Relational Programming

Yisu Remy Wang

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

Doctor of Philosophy

University of Washington

2023

Reading Commitee:

Dan Suciu, Chair

Zachary Tatlock

Andrew Lumsdaine

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Relational Programming

Yisu Remy Wang

Chair of the Supervisory Committee:
Dan Suciu
Paul G. Allen School of Computer Science & Engineering

Relational databases have become one of the most important software components in the world, ubiquitous in computing systems from smartphones to data centers. Nevertheless, traditional relational databases are struggling to meet the demands of modern data analytics, which involve more and more complex computation over an increasing variety of data. In this dissertation, we reinvigorate relational data processing with a new language foundation for writing relational programs, a new algorithm for the relational join, and techniques to optimize relational queries. We extend the reach of the relational paradigm in both expressiveness and efficiency, ushering a new era of relational programming.

Contents

Acknowledgments

By sheer chance I became a student of my advisor Dan Suciu, thus beginning my journey into the world of databases research. Dan has taught me so much – not only everything I know about databases, but also to think and speak with clarity, and to "get to the bottom of it". But above all, to be kind. Because what matters most is not the facts and figures, but the community of people in pursuit of knowledge.

I'm also proud to call myself a student of Zach Tatlock, even though he's never been my "advisor on paper". When I was lost and confused, Zach welcomed me into the PLSE lab with open arms, as he has done for countless others, and supported me unconditionally. Getting a PhD has been the hardest thing I've ever done, and I knew I could do it because Zach believed in me.

I'm fortunate to call both the PLSE lab and the UWDB lab my home, and I've learned as much from my peers as I have elsewhere. Among many others, I want to thank Max Willsey, Chandrakana Nandi, Yihong Zhang, Jonathan Leang and Shana Hutchison for sharing this journey with me.

Since I was little, my family have exhausted every last resource to give me the best education possible. They taught me what it truly means to sacrifice. I hope they will be pleased to know that I have taken on a mission to educate the next generation of computer scientists.

When it rains in Seattle, it's always sunny in the small apartment I share with my fiancée, Ellen Wu. Thanks for putting up with me, and let's go on many more adventures together!

Chapter 1

Introduction

Fifty years after its initial proposal by Edgar F. Codd [Cod70], the relational model has cemeted its status as the de-facto data model in nearly all modern databases. It provides *data independence*, which has allowed data systems to scale to unprecedented sizes while guaranteeing performance and correctness. The success of the relational model is witnessed by the popularity of SQL, uniquitous in computing systems from smartphones to data centers.

Nevertheless, traditional relational databases are struggling to meet the demands of modern data analytics. Today's data analytics involve new kinds of data, as well as new kinds of computation over such data. For example, machine learning workloads involve linear algebra operations running over data stored as matrices. For another example, graph analytics require iterative algorithms over graph data. Running these workloads in existing relational databases is both cumbersome and slow: SQL is a poor language for expressing the computation, and the systems are not optimized for such workloads.

This dissertation is motivated by the question: *How can we renew relational database systems to support modern data analytics?* Towards that end, we propose a new language foundation for writing relational programs; a new algorithm for the relational join; and techniques to optimize

relational queries. Together, these three ingredients make up the basis for a new generation of relational systems that are more expressive and more efficient. Using these systems, the analyst may author entire application programs instead of simple queries *relationally*, entering the era of *relational programming*.

1.1 Motivation and Contributions

The main query language for relational data, SQL, is found today in a wide range of applications and devices. The reason for its success is the *data independence principle*, which separates the declarative model from the physical implementation [Cod70], and enables advanced implementation techniques, such as cost-based optimizations, indices, materialized views, incremental view maintenance, parallel evaluation, and many others, while keeping the same simple, declarative interface to the data unchanged.

But analysts today often need to perform tasks that require iteration over the data. Gradient descent, clustering, page-rank, network centrality, inference in knowledge graphs are some examples of common tasks in data science that require iteration. While SQL has introduced recursion since 1999 (through Common Table Expressions, CTE), it has many cumbersome limitations and is little used in practice [McS22].

The need to support recursion in a declarative language led to the development of Datalog in the mid 80s [Via21a]. Datalog adds recursion to the relational query language, yet enjoys several elegant properties: it has a simple, declarative semantics; its naïve bottom-up evaluation algorithm always terminates; and it admits a few powerful optimizations, such as semi-naïve evaluation and magic set rewriting. Datalog has been extensively studied in the literature; see [GHLZ13] for a survey and [MTKW18, Via21a] for historical notes. We will also briefly review the semantics and execution of Datalog in Chapter 3.

However, Datalog is not the answer to modern needs, because it only supports monotone queries over sets. Most tasks in data science today require the interleaving of recursion and aggregate computation. Aggregates are not monotone under set inclusion, and therefore they are not supported by the framework of pure Datalog. Neither SQL'99 nor popular open-source Datalog systems like Soufflé [JSS16] allow recursive queries to have aggregates. While several proposals have been made to extend Datalog with aggregation [GGZ91, RS92, GGZ95, MSZ13b, SYZ15, SYI+16a, ZYDI16, ZYD+17, ZYI+18, CDI+18, GWM+19, ZDG+19, DLW+19, ZDG+21], these extensions are at odds with the elegant properties of Datalog and have not been adopted by either Datalog systems or SQL engines.

The first contribution of this dissertation is a foundation for a query language that supports both recursion and aggregation. Our language, called Datalog°, is a generalization of Datalog. Intuitively, Datalog works over sets which are functions from tuples to booleans. Datalog° instead works over functions from tuples to values from a semiring. For example, setting the semiring to be the reals allows us to define recursive tensor programs in Datalog°, and the using the min-plus tropical semiring $(\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$ lets us compute shortest paths over weighted graphs. Datalog° retains many of the elegant properties of Datalog, while extending its expressiveness to support aggregation.

To evaluate any relational program, including Datalog° programs, classic Datalog programs, or even SQL queries without recursion, a key operation is the relational join. The join allows us to combine data from different sources, as well as compose computation from different programs. Most mainstream databases today evaluate a query by joining two relations at a time. In other words, they implement *binary join* algorithms. Over the last decade, worst-case optimal join (WCOJ) [NPRR12, Vel14, NRR13, Ngo18] has emerged as a breakthrough in the design of efficient join algorithms. It can be asymptotically faster than traditional binary joins, all the while remaining simple to understand and implement [NRR13]. However, traditional binary join algorithms have

benefited from decades of research and engineering. Techniques like column-oriented layout, vectorization, and query optimization have contributed compounding constant-factor speedups, making it challenging for WCOJ to be competitive in practice.

The second contribution of this dissertation is a new join algorithm, called Free Join, that unifies WCOJ and binary join. We propose several new techniques to make Free Join outperform both binary join and WCOJ:

1. An algorithm to convert any binary join plan to a Free Join plan that runs as fast or faster.

2. A new data structure called COLT (for *Column-Oriented Lazy Trie*), adapting the classic column-oriented layout to improve the trie data structure used in WCOJ.

3. A vectorized execution algorithm for Free Join.

While a faster join algorithm improves the performance of relational programs at the operator level, modern data analytics presents abundant opportunities for higher-level optimizations. For example, machine learning pipelines heavily rely on linear algebra operations, and the order to carry out these operations can have a significant impact on performance. Consider the Linear Algebra (LA) expression $sum((X - UV^T)^2)$ which defines a typical loss function for approximating a matrix X with a low-rank matrix UV^T. Here, $sum()$ computes the sum of all matrix entries in its argument, and A^2 squares the matrix A element-wise. Suppose X is a sparse, 1M x 500k matrix, and suppose U and V are dense vectors of dimensions 1M and 500k respectively. Thus, UV^T is a rank 1 matrix of size 1M x 500k, and computing it naively requires 0.5 trillion multiplications, plus memory allocation. Fortunately, the expression is equivalent to $sum(X^2) - 2U^TXV + U^TU * V^TV$. Here U^TXV is a scalar that can be computed efficiently by taking advantage of the sparsity of X, and, similarly, U^TU and V^TV are scalar values requiring only 1M and 500k multiplications respectively.

Optimization opportunities like this are ubiquitous in machine learning programs. State-of-the-art optimizing compilers such as SystemML [Boe19], OptiML[SLB+11], and Cumulon[HBY13] commonly implement syntactic rewrite rules that exploit the algebraic properties of the LA expressions. For example, SystemML includes a rule that rewrites the preceding example to a specialized operator[1] to compute the result in a streaming fashion. However, such syntactic rules fail on the simplest variations, for example SystemML fails to optimize $sum((X + UV^T)^2)$, where we just replaced $-$ with $+$. Moreover, rules may interact with each other in complex ways. In addition, complex ML programs often have many common subexpressions (CSE) that further interact with syntactic rules, for example the same expression UV^T may occur in multiple contexts, each requiring different optimization rules.

The third contribution of this dissertation is SPORES, a novel optimization approach for complex linear algebra programs that leverages relational algebra as an intermediate representation to completely represent the search space. SPORES first transforms LA expressions into traditional Relational Algebra (RA) expressions consisting of joins, unions and aggregates. It then performs a cost-based optimizations on the resulting Relational Algebra expressions, using only standard identities in RA. Finally, the resulting RA expression is converted back to LA, and executed.

Beyond linear algebra, the prevalence of recursion in modern workloads also calls for novel optimization techniques. Yet the optimization problem for recursive queries is little studied. Although various data systems support recursion and iteration in one form or another, most systems only optimize within each iteration. The few systems that optimize across iterations apply limited techniques, like magic set optimization or semi-naïve evaluation, which are restricted to postitive queries.

For example, consider the following Datalog query, computing the distance between all pairs

[1] See the SystemML Engine Developer Guide for details on the weighted-square loss operator `wsloss`.

of nodes in a graph:

```
path(X,Y,L) :- edge(X,Y,L).
path(X,Y,L) :- edge(X,Z,L1), path(Z,Y,L2), L=L1+L2.
dist(X,Y,D) :- D=min{ L | path(X,Y,L) }.
```

We will cover the semantics of Datalog in Chapter 3, but intuitively explain the query here. The first rule says that there is a path of length L from X to Y if there is an edge from X to Y of length L. The second rule says that there is a path of length $L_1 + L_2$ from X to Y if there is an edge from X to Z of length L_1, and a path of length L_2 from Z to Y. Finally, the last rule says that the distance from X to Y is the minimum length of all paths from X to Y. The program is intuitive yet inefficient: to find the shortest path between two nodes, it first computes *all* paths between the nodes. A graph with cycles also contains an infinite number of paths, and the program will not terminate. A different query, in our new Datalog$^\circ$ language, computes the shortest paths more efficiently:

```
path(X, Y) = min(edge(X, Y), min_Z{ edge(X, Z) + path(Z, Y) })
```

Here we treat `path` and `edge` as functions from pairs of nodes to some length. The program essentially follows the Bellman-Ford algorithm. Initially we assign the distance of every pair to be the length of the edge between them. Then, in each iteration, we update the distance in a breadth-first manner, adding the length of an edge from a previously found distance. This new program is more efficient and terminates on cyclic graphs with postitive lengths. It is natural to hope a query optimizer can rewrite the first program to the second, and perform similar optimization for other recursive queries.

The fourth and final contribution of this dissertation is a new query optimization framework for recursive queries. Our framework replaces a recursive program with another, equivalent recursive program, whose body may be quite different, and thus focuses on optimizing

the recursive program as a whole, not on optimizing its body in isolation; the latter can be done separately, using standard query optimization techniques. Our optimization is based on a novel rewrite rule for recursive programs, called the FGH-rule, which we implement using *program synthesis*, a technique developed in the programming languages and verification communities. We introduce a new method for inferring loop invariants, which extends the reach of the FGH-rule, and also show how to use global constraints on the data for semantic optimizations using the FGH-rule.

1.2 Chapter Organization & Publications

The rest of this dissertation is organized as follows:

- Chapter 2 covers related work to relational programming.
- Chapter 3 reviews the basics of relational data processing.
- Chapter 4 introduces the Datalog° language as well as an algorithm to evaluate its programs.
- Chapter 5 presents the Free Join algorithm.
- Chapter 6 presents the SPORES system for optimizing linear algebra programs.
- Chapter 7 presents our approach to optimizing recursive queries using the FGH-rule.
- Chapter 8 concludes the dissertation and discusses future work.

Chapter 4 contains material from our PODS paper [ANP+22] and SIGMOD Record article [KNP+22]. The full version of the PODS paper [KNP+22] contains additional proofs as well as detailed discussions on several extensions to Datalog°. Chapter 5 is based on our SIGMOD paper [WWS23]. Chapter 6 is based on our VLDB paper [WHS+20], and another project using the same techniques

to optimize deep learning inference was published at MLSys [YPW+21]. and Chapter 7 is based on our SIGMOD paper [WAN+22]. The equality saturation system used in projects throughout this dissertation was developed in a series of publications with my collaborators [NWZ+21, ZWWT22, ZWF+23, WNW+21].

Chapter 2

Related Work

In this chapter we cover related work to this dissertation. First, we discuss prior proposals for relational programming languages. Then we review the literature on join algorithms, with an emphasis on multi-way joins. Finally, we cover existing techniques for optimizing relational programs.

2.1 Relational Programming Languages

We are not the first to use the term "relational programming". It appeared as early as 1981 in a paper by Bruce J. MacLennan [Mac81], where it is described as "a style of programming in which entire relations are manipulated rather than individual data". The term was also used as a synonym for logic programming a la Prolog [Col90], where the fundamental building blocks are relations. While there is, to the best of our knowledge, no surviving implementation of MacLennan's relational programming language, many Prolog systems are still being used and actively developed [Zho12, BCC+97, DC+01, Pro21, WSTL12]. Both MacLennan and proponents of Prolog emphasize one aspect of their languages: thanks to a relational foundation, the programmer

can specify complex logic at a very high level, resulting in concise and elegant programs. As a result, such langauges found success in classic AI applications like expert systems, which require sophisticated symbolic reasoning [KLB+22].

More recently, a class of languages called miniKanren has renewed interest in relational programming [Byr09, BHF12, RVB19, FBK05]. The designers of miniKanren were motivated by the "impurity" of Prolog: since Prolog evaluation is based on backtracking, Prolog programmers frequently resort to imperative features like the "cut" operator to prune the enormous search space. In contrast, miniKanren has been touted as "purely logical" or "purely relational", in that it more strictly adheres to the declarative paradigm. Another distinguishing feature of miniKanren is its simple design: the core language consists of only three logical operators: the equality constraint `==`, disjunction `conde`, and variable introduction `fresh`. Thanks to its simplicity, miniKanren has been embedded in dozens of programming languages.

A closer line of work to us centers around the Datalog language. We will cover Datalog in more detail in Chapter 3, but note here that it is a "pure" subset of Prolog. There are well-established declarative semantics for Datalog based on proof theory and model theory [AHV95]. As we will see, Datalog is also delightfully simple: a Datalog program is just a set of inference rules, and the naïve evaluation algorithm for Datalog just applies the rules in a loop until a fixpoint.

Nevertheless, the core Datalog language is very limited. It does not allow negation, arithmetic, or aggregation, all of which are essential for modern data analysis. Researchers have therefore proposed various extensions to Datalog to support such constructs. In this dissertation, we focus on extensions that aim to support *aggregates* in Datalog. These include, but are not limited to, the standard `MIN`, `MAX`, `SUM`, and `COUNT` aggregates in SQL.

The main challenge in having aggregates is that they are not *monotone* under set inclusion, yet monotonicity is crucial for the declarative semantics of Datalog, and optimizations like semi-naïve evaluation. Approaches to resolve the tension between aggregates and monotonicity mainly

follow two strategies: break the program into *strata*, or generalize the order relation to ensure that aggregates become monotone.

Stratified Aggregates The simplest way to add aggregates to Datalog while staying monotone is to disallow aggregates in recursion. Proposed by Mumick et al. [MPR90], the idea is inspired by stratified negation, where every negated relation must be computed in a previous stratum. Stratifying aggregates has the benefit that the semantics, evaluation algorithms, and optimizations for classic Datalog can be applied unchanged to each stratum. However, stratification limits the programs one is allowed to write – in Chapter 4 we show two versions of the same program, one stratified and one not, yet the latter is more efficient. The stratification requirement can also be a cognitive burden on the programmer. In fact, the most general notion of stratification, dubbed "magic stratification" [MPR90], involves both a syntactic condition and a semantic condition defined in terms of derivation trees.

Generalized Ordering In this dissertation, we follow the approach that restores monotonicity by generalizing the ordering on which monotonicity is defined. The key idea is that, although a program is not monotone according to the $\subseteq$ ordering on sets, we can pick another order under which P **is** monotone. Ross and Sagiv [RS92] define the ordering[1] $P \sqsubseteq P'$ as:

$$\forall (x, y, d) \in P, \exists (x, y, d') \in P' : (x, y, d) \sqsubseteq (x, y, d')$$
$$\text{where } (x, y, d) \sqsubseteq (x', y', d') \stackrel{\text{def}}{=} x = x' \wedge y = y' \wedge d \geq d'$$

That is, P increases if we replace (x, y, d) with (x, y, d') where $d' < d$, for example $\{(a, c, 21)\} \sqsubseteq \{(a, c, 11)\}$. In general, to define a generalized ordering we need to view a relation as a map from a tuple to an element in some ordered set S. For example, the relation P maps a pair of vertices

[1] If (x, y) is not a key in P, then $\sqsubseteq$ is only a preorder.

(x, y) to a distance d. We will call such generalized relations S-relations. Different approaches in existing work have modeled S using different algebraic structures: Ross and Sagiv [RS92] require it to be a complete lattice, Conway et al. [CMA+12] require only a semilattice, whereas Green et al. [GKT07] require S to be an ω-continuous semiring. These proposals bundle the ordering together with two operations $\otimes$ and/or $\oplus$. In this dissertation, we follow this line of work and ensure monotonicity by generalizing the ordering. However, in contrast to prior work we *decouple* operations on S-relations from the ordering, and allow one to freely mix and match the two as long as monotonicity is respected.

Other Approaches There are other approaches to support aggregates in Datalog that do not fall into the two categories above. We highlight a few of them here. Ganguly et al. [GGZ91] model min and max aggregates in Datalog with negation, thereby supporting aggregates via semantics defined for negation. Mazuran et al. [MSZ13a] extend Datalog with counting quantification, which additionally captures `SUM`. Kemp and Stuckey [KS91] extend the well-founded semantics [GRS91, Gel89] and stable model semantics [GL88] of Datalog to support recursive aggregates. They show that their semantics captures various previous attempts to incorporate aggregates into Datalog. They also discuss a semantics using *closed* semirings, and observe that under such a semantics some programs may not have a unique stable model. Semantics of aggregation in Answer Set Programming has been extensively studied [FPL11, GZ14, Alv16]. Liu and Stoller [LS22] give a comprehensive survey of this space.

2.2 Join Algorithms

Join algorithms have been studied since the birth of relational databases, and there are thousands of papers focusing on improving the performance of joins. Algorithms for computing the join

roughly fall into two categories: binary joins and multi-way joins. We will provide a more in-depth comparison of the two paradigms in Chapter 5, and highlight a few distinguishing features here.

Compared to multi-way joins, binary joins are by far more frequently implemented in practice [GPB+22, SK91, RM19]. This is largely because the former is more modular: to compute the join of multiple relations, we can chain together a sequence of binary joins. Modularity simplifies implementation, and also makes it easy to optimize, parallelize, incrementalize, and distribute the query, or make it more robust to failures.

On the other hand, multi-way joins can be more efficient than binary joins [GBC98, KKW99, AH00]. By processing multiple relations at the same time, multi-way joins can avoid materializing intermediate results. It also has the freedom to reorder the relations at runtime, taking advantage of better statistics computed during execution. More recently, researchers have shown a kind of multi-way join algorithms to be *Worst-case optimal* [NPRR12, Vel14, NRR13, Ngo18], making them the algorithms to have stronger asymptotic guarantees than the classic binary join algorithms.

Given that each paradigm has its own strength, researchers have proposed to combine the two within the same system [FBS+20, ALT+17, MS19], much like how modern database systems implement both hash joins and sort-merge joins. We will discuss these systems in more detail in Chapter 5. Our approach is motivated by the same idea of bringing the best of both worlds, but instead of implementing both binary join and multi-way join, we unify and generalize them into a single join algorithm.

2.3 Optimizing Relational Programs

There is a vast body of literature for both relational query optimization and optimizing compilers for machine learning. Since we optimize machine learning programs through a relational lens, our work relates to research in both fields. As we have pointed out, numerous state-of-the-

art optimizing compilers for machine learning resort to syntactic rewrites and heuristics to optimize linear algebra expressions [Boe19] [SLB+11] [HBY13]. We perform optimization based on a relational semantics of linear algebra and holistically explore the complex search space. A majority of relational query optimization focus on join order optimization [Gra95] [MN06a] [MN08] [SAC+79a]; we optimize programs with join (product), union (addition), and aggregate (sum) operations. Sum-product optimization considers operators other than join while optimizing relational queries. Recent years have seen a line of excellent theoretical and practical research in this area [KNR16] [JPR16]. These work gives significant improvement for queries involving $\times$ and $\sum$, but fall short of LA workloads that occur in practice. We step past these frameworks by incorporating common subexpressions and incorporating addition (+).

2.3.1 Relational Algebra and Linear Algebra

Elgamal et al. [ELB+17] envisions SPOOF, a compiler for machine learning programs that leverages relational query optimization techniques for LA sum-product optimization. We realize this vision by providing the translation rules from LA to RA and the relational equality rules that completely represents the search space for sum-product expressions. One important distinction is, Elgamal et al. proposes *restricted relational algebra* where every expression must have at most two free attributes. This ensures every relational expression in every step of the optimization process to be expressible in LA. In contrast, we remove this restriction and only require the optimized output to be in linear algebra. This allows us to trek out to spaces not covered by linear algebra equality rules and achieve completeness. In addition to sum-product expressions, Elgamal et al. also considers selection and projection operations like selecting the positive entries of a matrix. We plan to explore supporting selection and projection in the future. Elgamal et al. also proposes compile-time generation of fused operators, which is implemented by Boehm et al. [BRH+18].

SPORES can replace operations with existing fused operators when it is beneficial, and we plan to explore combining sum-product rewrite with fusion generation in the future.

MorpheusFI by Li et al. [LCK19] and LARA by Hutchison et al. [HHS17] explore optimizations across the interface of machine learning and database systems. In particular, MorpheusFI speeds up machine learning algorithms over large joins by pushing computation into each joined table, thereby avoiding expensive materialization. LARA implements linear algebra operations with relational operations and shows competitive optimizations alongside popular data processing systems. Schleich et al.[SOC16] and Khamis et al.[KNN+17] explore in-database learning, which aims to push entire machine learning algorithms into the database system. We contribute in this space by showing that even without a relational engine, the relational abstraction can still benefit machine learning tasks as a powerful intermediate abstraction. Kotlyar et.al. [KPS97] explore compiling sparse linear algebra via a relational abstraction. We contribute by providing a simple set of rewrite rules and prove them complete.

2.3.2 Low-level Code Generation

Novel machine learning compilers including TACO [KKC+17], TVM [CZY+18], TensorComprehension [VZT+18] and Tiramisu [BRR+19] generate efficient low-level code for kernel operators. These kernels are small expressions that consist of a few operators. For example the MATTRANSMUL kernel in TACO implements $\alpha A^T x + \beta z$. The kernels are of interest because they commonly appear in machine learning programs, and generating efficient low-level implementation for them can greatly impact performance. However, these compilers cannot perform algebraic rewrite on large programs as SPORES does. For example, TACO supports only plus, multiply and aggregate, whereas SPORES supports any custom functions as discussed in Section 6.2.3; Tiramisu requires tens of lines of code just to specify matrix multiply which is a single operator in SPORES. Fur-

thermore, the basic polyhedral model in Tiramisu and TensorComprehension does not support sparse matrices. Sparse extensions exist, but require the user to make subtle tradeoffs between expressivity and performance [SHO18]. At a high level, we view these kernel compilers as complementary to SPORES. The former can provide efficient kernel implementation just like the fused operators in SystemML, and we can easily include these kernels in SPORES for whole-program rewrite. The TASO compiler [JPT+19] combines kernel-level rewrite with whole-program rewrite, and is also driven by a set of equality rules like SPORES. However, it induces significant overhead – generating operator graphs with just up to 4 operators takes 5 minutes, and while [JPT+19] does not include detailed time for compiling whole programs, it reports the compilation finishes in "less than ten minutes". In contrast, SPORES takes seconds instead of minutes in compilation.

2.3.3 Recursive Program Optimization and Program Synthesis

Our work on optimizing recursive programs was partially inspired by the PreM condition, described by Zaniolo et al. [ZYD+17], which, as we shall explain, is a special case of the FGH-rule. Unlike our system, their implementation required the programmer to check the PreM manually, then perform the corresponding optimization. Seveal prior systems leveraged SMT-solvers to reason about query languages [VGdHT09, CWWC17, GCI+17, SRLS17, WDLC18]; but none of these consider recursive queries. Datalog synthesizers have been described in [AKNS17, SLZ+18, SRHN19, WSC+20, RMZ+20]. Their setting is different from ours: the specification is given by input-output examples, and the synthesizer needs to produce a program that matches all examples. A design choice that we made, and which sets us further aside from the previous systems, is to use an existing CEGIS system, Rosette; thus, we do not aim to improve the CEGIS system itself, but optimize the way we use it.

Chapter 3

Background

This chapter reviews basic concepts in relational query processing. We cover relations and (unions of) conjunctive queries in Section 3.1, relational algebra with an emphasis on joins in Section 3.2, and Datalog in Section 3.3. By the end of this chapter, the reader will have the knowledge necessary to build a very simple evaluator for Datalog.

3.1 Relations and (Unions of) Conjunctive Queries

Let D be a set of values, for example the set of natural numbers, or the set of ASCII strings. A *tuple* over D is of the form $(t_1, t_2, \ldots, t_k)$ where each t_i is a value in D, and k is called the *arity* of the tuple. A *relation* over D is a set of tuples over D, each with the same arity which we call the arity of the relation. Note that many popular database systems allow duplicate tuples in a relation; in other words they follow the so-called *bag semantics*. We will focus on the set semantics in this section, and in Chapter 4 we will show how to support bag semantics by extending the relations with an algebraic structure called *semiring*. Many databases also prescribe a *schema* for each relation. We omit the schema because it is not necessary for the conjunctive query notation

that we follow in this thesis. However, we will occasionally use SQL queries in our examples to make them more readible to SQL programmers.

Example 3.1. Let $D = \{1, 2, \ldots, n\}$ represent a set of n vertices in a graph. A binary relation E over D represents the (directed) edges in the graph. Let $E \stackrel{\text{def}}{=} \{(x, y) \mid x, y \in D\}$, then E is the complete graph on D.

Given a set of relations $R_1, R_2, \ldots, R_n$, a *conjunctive query* (CQ) is of the form:

$$Q(\boldsymbol{X}) \text{ :- } R_1(\boldsymbol{X}_1), R_2(\boldsymbol{X}_2), \ldots, R_n(\boldsymbol{X}_n).$$

where each $\boldsymbol{X}_i$ (and $\boldsymbol{X}$) is a tuple consisting of variables and constants. $\boldsymbol{X}_i$ must have the same arity as R_i, and the arity of $\boldsymbol{X}$ is called the *arity* of the query. Each $R_i(\boldsymbol{X}_i)$ (and $Q(\boldsymbol{X})$) is called an *atom*. When the context is clear, we will use Q to refer to both the query and its output relation. The query Q is called *safe* if each variable in $\boldsymbol{X}$ also appears in some $\boldsymbol{X}_i$. We will only consider safe queries in this thesis. A variable only appearing in the body is called a *bound* variable. Otherwise, it is called a *free* variable. Q is called a *full* conjunctive query if it has no bound variables. The conjunctive query computes in Q the set $\{\boldsymbol{X} \mid \exists_{\boldsymbol{X}_{\text{bound}}} : \bigwedge_{i \in [1 \ldots n]} \boldsymbol{X}_i \in R_i\}$ where $\boldsymbol{X}_{\text{bound}}$ is the set of bound variables.

Example 3.2. Using the relation E from Example 3.1, the query $Q(X, Z)$:- $E(X, Y), E(Y, Z)$. computes all pairs of vertices (X, Z) such that there is a vertex Y connected to both X and Z. Formally, the set $\{(x, z) \mid \exists y : (x, y) \in E \wedge (y, z) \in E\}$. In other words, it finds all paths of length 2 in the graph. The equivalent SQL query, assuming E has schema `(head, tail)`, is the following:

```
SELECT E1.head, E2.tail
  FROM E AS E1, E AS E2
 WHERE E1.tail = E2.head
```

Finally, a set of multiple conjunctive queries with the same head relation defines a *union of conjunctive queries* (UCQ). It has the form:

$$Q(\boldsymbol{X}) \text{ :- } R_{i_1}(\boldsymbol{X}_1), R_{i_2}(\boldsymbol{X}_2), \ldots, R_{i_m}(\boldsymbol{X}_m)$$
$$Q(\boldsymbol{Y}) \text{ :- } R_{j_1}(\boldsymbol{Y}_1), R_{j_2}(\boldsymbol{Y}_2), \ldots, R_{j_n}(\boldsymbol{Y}_n)$$
$$\ldots$$

As the name suggests, a UCQ computes the union of each conjunctive query.

3.2 Relational Algebra

Relational queries are compiled to relational algebra before they can be executed by a database system. The relational join is the central operation in relational algebra, as it connects data from different relations and composes computation from different queries. In this section we describe the semantics of the join as well as a simple algorithm to compute it. We also review other relational algebra operators, namely selection, projection, and union.

A *full* conjunctive query Q where no atom contains constants or duplicate variables is called a *natural join* over the relations in Q. We will only consider the natural join in this thesis, and will simply call it *join*. A *binary* join is a join involving only two relations. We can compute the natural join of multiple relations by joining two relations at a time. Given a conjunctive

$Q(\boldsymbol{X})$:- $R_1(\boldsymbol{X}_1), R_2(\boldsymbol{X}_2), \ldots, R_n(\boldsymbol{X}_n)$, define the following queries:

$$
\begin{aligned}
Q_2(\boldsymbol{X}_1 \cup \boldsymbol{X}_2) &\text{ :- } R_1(\boldsymbol{X}_1), R_2(\boldsymbol{X}_2) \\
Q_3(\boldsymbol{X}_1 \cup \boldsymbol{X}_2 \cup \boldsymbol{X}_3) &\text{ :- } Q_2(\boldsymbol{X}_1 \cup \boldsymbol{X}_2), R_3(\boldsymbol{X}_3) \\
&\vdots \\
Q_n(\boldsymbol{X}_1 \cup \boldsymbol{X}_2 \cup \cdots \cup \boldsymbol{X}_n) &\text{ :- } Q_{n-1}(\boldsymbol{X}_1 \cup \boldsymbol{X}_2 \cup \cdots \cup \boldsymbol{X}_{n-1}), R_n(\boldsymbol{X}_n)
\end{aligned}
$$

We abuse "$\boldsymbol{X}_1 \cup \boldsymbol{X}_2$" to mean a tuple containing all variables in $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ without duplicates, in any order, and similarly for "$\boldsymbol{X}_1 \cap \boldsymbol{X}_2$". It is easy to check that $Q_n = Q$. Indeed, most modern database systems compute the natural join of multiple relations by joining two relations at a time.

A simple yet effective algorithm to compute the binary join is called *hash join*, and it is implemented in nearly every database system. Algorithm 1 shows the pseudocode for hash join. Given a full conjunctive query (without constants) of two relations $Q(\boldsymbol{X}_1 \cup \boldsymbol{X}_2)$:- $R(\boldsymbol{X}_1), S(\boldsymbol{X}_2)$, let $\boldsymbol{X} \stackrel{\text{def}}{=} \boldsymbol{X}_1 \cap \boldsymbol{X}_2$. We pick one of the relations to be the *left* relation and the other to be the *right* relation. Suppose we pick R to be the left relation, and S to be the right relation. We first initialize s to be a empty hash map where each key will be (a portion of) a tuple in S, mapped to a vector of tuples in S. Then we iterate over each tuple in S, binding the values to $\boldsymbol{X}_2$. In each iteration, we insert $\boldsymbol{X}_2$ to the vector mapped to by the key $\boldsymbol{X}$ (recall $\boldsymbol{X}$ contains a subset of the variables in $\boldsymbol{X}_2$). Next, we iterate over each tuple in R, binding the values to $\boldsymbol{X}_1$, and look up the hash map to find all tuples in s that match the values in $\boldsymbol{X}$. For each match, we concatenate the tuples from each relation, and output the result.

For each binary join, the decision of which relation to be the left and which to be the right can have a significant impact on performance. Because we need to build a hash map for the right relation, using a large relation on the right incurs heavy overhead. And when the right relation

Algorithm 1: Hash join of $R(\boldsymbol{X}_1)$ and $S(\boldsymbol{X}_2)$, using S as the right relation.

```
s ← new hash map;
for X_2 ∈ S do
    s[X].insert(X_2);
end
for X_1 ∈ R do
    for X_2 ∈ s[X] do
        output (X_1 ∪ X_2);
    end
end
```

is itself a join, we would have to materialize the join before we can build the hash map. Instead, when a "sub-join" is on the left, we can simply iterate over its results as they are produced. For this reason, databases frequently implement the so-called *left-deep linear* join, where every right relation is a base relation, and the left-most relation is (usually) the largest relation.

To support conjunctive queries with constants, duplicate variables, and those that are not full, we need two additional operators from relational algebra, namely selection (σ) and projection (Π). The selection operator σ takes as arguments an atom $R(\boldsymbol{X})$ and a predicate θ over $\boldsymbol{X}$, and returns a subset of the relation R containing only the tuples that satisfy θ. For example, $\sigma_{X=1}E(X, Y)$ returns all edges in E that start at vertex 1. For another example, $\sigma_{X=Y}E(X, Y)$ returns all edges in E that are self-loops; it is equivalent to repeating the variable X, i.e., $E(X, X)$. The projection operator Π takes as arguments an atom $R(\boldsymbol{X})$ and a set of variables $\boldsymbol{Y} \subseteq \boldsymbol{X}$, and returns a relation containing only attributes corresponding to the variables in $\boldsymbol{Y}$. For example, $\Pi_X E(X, Y)$ returns all source vertices in E. Selection can be implemented with a simple loop over the tuples in the input relation. The same goes for projection in *bag* semantics, whereas in set semantics, projection requires looping over the tuples while taking care to remove duplicates. Finally, computing UCQs requires the union operator $\cup$, which simply takes the set union of the relations. Popular databases also implement several additional relational algebra operators such as aggregation. In Chapter 4

we will show how to support aggregation as a generalization of projection, when we extend the relational algebra with semirings.

Example 3.3. To compute the query $Q(X)$:- $E(X,Y), E(Y,X)$ (set of vertices that loop back to themselves in two steps), we first compute the join $J(X,Y,Z)$:- $E(X,Y), E(Y,Z)$, then select the tuples where $X = Z$ with $S(X,Y,Z)$:- $\sigma_{X=Z}J(X,Y,Z)$. then project out Y and Z with $Q(X)$:- $\Pi_X S(X,Y,Z)$.

Although we have presented the evaluation of a query by computing a sequence of intermediate queries, like J and S in Example 3.3, in practice the database system will combine the different relational algebra operators and compute the result "in one go". This approach to avoid the intermediates is called *pipelining*. For example, to compute Q in Example 3.3, we may fuse together the inner loop of the join with the loops of the selection and projection, as shown in Algorithm 2.

Algorithm 2: Pipelined execution of Q in Example 3.3.

```
1  e ← new hash map ;
2  for (Y, Z) ∈ E do
3  |  e[Y].insert((Y, Z));
4  end
5  for (X, Y) ∈ E do
6  |  for (Y, Z) ∈ e[Y] do
7  |  |  if X = Z then
8  |  |  |  output X;
9  |  |  end
10 |  end
11 end
```

3.3 Datalog

Many applications in modern data analytics require recursion. For example, we may want to detect unreachable nodes in a network. This can be achieved by computing the transitive closure

of the edges in the network, which can be expressed as a recursive query. Datalog is a query language designed to support recursion, and we review its semantics and exeuction in this section.

A Datalog program is a set of conjunctive queries. Each conjunctive query is also called a *rule*, where different rules may share the same head atom. If a relation never appears in any head atom, we call it an input relation or an *extensional database* (EDB) relation. Otherwise, we call it an output relation or an *intensional database* (IDB) relation.

Example 3.4. The following Datalog program computes the transitive closure of the relation E.

$$P(X, Y) \text{ :- } E(X, Y).$$
$$P(X, Z) \text{ :- } P(X, Y), E(Y, Z).$$

The meaning of a Datalog program can be defined as the solution (or *model*) of a set of logical constraints. This is known as the *model theoretic semantics* of Datalog. Specifically, we interpret each atom as a predicate, and treat each rule as a $\forall$-quantified logical implication from the conjunction of the body atoms to the head atom. For example, the second rule in Example 3.4 is interpreted as $\forall X, Y, Z : P(X, Y), E(Y, Z) \implies P(X, Z)$. Then the result of the Datalog program is the *smallest* relations such that all constraints are satisfied. The emphasis on *smallest* is important, because otherwise we may simply set every relation R with arity k to contain all tuples $\{(t_1, t_2, \ldots, t_k) \mid t_i \in D \text{ for each } i\}$ to satisfy the constraints.

Example 3.5. Setting $E \stackrel{\text{def}}{=} \{(1, 2), (2, 3), (3, 4), (4, 5)\}$, the Datalog program in Example 3.4 computes the set of all paths in the graph E, namely $P = \{(x, y) \mid 1 \leq x < y \leq 5\}$.

An alternative semantics for Datalog is the *fixpoint semantics*. It is equivalent to the model theoretic semantics, but also tells us how to evaluate the Datalog program. For each IDB relation R_i, the set of all rules with R_i in the head form a union of conjunctive queries. This UCQ defines a

function F_i from the relations in the rule bodies to R_i, which we can extend to be a function from all IDB relations to R_i (relations not in the bodies are simply ignored). We write F for the set of all F_i, and F is then a function from all IDB relations to all IDB relations. We call F the immediate consequence operator (ICO) of the Datalog program. To evaluate the Datalog program, as shown in Algorithm 3, we initialize all IDB relations to be empty, then repeatedly apply F to update them, until they no longer change. This is also known as the *naïve evaluation* of the Datalog program.

Algorithm 3: Naïve evaluation of a Datalog program. I is the set of IDB relations.

```
1 I ← ∅;
2 while I changes do
3 |   I ← F(I);
4 end
5 return I
```

Example 3.6. The evaluation of the program in Example 3.4 is shown here:

$$
\begin{array}{ll}
 & P \\
\text{iter 1} & \{(1,2),(2,3),(3,4),(4,5)\} \\
\text{iter 2} & P^{(1)} \cup \{(1,3),(2,4),(3,5)\} \\
\text{iter 3} & P^{(2)} \cup \{(1,4),(2,5)\} \\
\text{iter 4} & P^{(3)} \cup \{(1,5)\} \\
\text{iter 5} & P^{(4)}
\end{array}
$$

Here we use $P^{(i)}$ to mean the content of relation P at iteration i. Note that in each iteration, each relation contains all tuples from the previous iteration. This is always true for any Datalog program, and we say that Datalog evaluation is *inflationary*.

An important property of Datalog is that the naïve evaluation always terminates, and it does so in polynomial time. However, the naïve evaluation is inefficient in practice. As shown in

Example 3.6, each iteration computes the relations from scratch, even though most tuples are already in the relations from the previous iteration. A more efficient algorithm, the *semi-naïve evaluation*, computes only the new tuples in each iteration. It works by tracking a *delta relation* for each relation, which contains the new tuples in each iteration. To compute a delta relation, every rule in the Datalog program is replaced with a set of delta rules. Given a rule containing n IDB atoms (R_1 to R_n) and m EDB atoms (S_1 to S_m):

$$R(\boldsymbol{X}) \text{ :- } R_1(\boldsymbol{X}_1), R_2(\boldsymbol{X}_2), \ldots, R_n(\boldsymbol{X}_n), S_1(\boldsymbol{Y}_1), S_2(\boldsymbol{Y}_2), \ldots, S_m(\boldsymbol{Y}_m).$$

we generate n delta rules, where the i-th rule has a delta relation for the i-th IDB atom, and all EDB atoms are unchanged:

$$\Delta R(\boldsymbol{X}) \text{ :- } \Delta R_1(\boldsymbol{X}_1), R_2(\boldsymbol{X}_2), \ldots, R_n(\boldsymbol{X}_n), S_1(\boldsymbol{Y}_1), S_2(\boldsymbol{Y}_2), \ldots, S_m(\boldsymbol{Y}_m).$$

$$\Delta R(\boldsymbol{X}) \text{ :- } R_1(\boldsymbol{X}_1), \Delta R_2(\boldsymbol{X}_2), \ldots, R_n(\boldsymbol{X}_n), S_1(\boldsymbol{Y}_1), S_2(\boldsymbol{Y}_2), \ldots, S_m(\boldsymbol{Y}_m).$$

$$\Delta R(\boldsymbol{X}) \text{ :- } R_1(\boldsymbol{X}_1), R_2(\boldsymbol{X}_2), \ldots, \Delta R_n(\boldsymbol{X}_n), S_1(\boldsymbol{Y}_1), S_2(\boldsymbol{Y}_2), \ldots, S_m(\boldsymbol{Y}_m).$$

Taking these delta rules, we define a new "delta ICO" for the program which we denote δF. Given IDB relations I and their delta relations ΔI, $\delta F(I, \Delta I)$ computes the new delta relations. The semi-naïve evaluation then proceeds as shown in Algorithm 4. In each iteration, we apply the delta rules to compute the delta relations ΔI, while taking care to remove already-discovered tuples in I from ΔI. We take the union of ΔI and the previous I to compute the new I, and stop when ΔI is empty.

Algorithm 4: Semi-naïve evaluation of a Datalog program.

```
1 I ← F(∅), ΔI ← F(∅) ;
2 while ΔI ≠ ∅ do
3 |   ΔI ← δF(I, ΔI) − I;
4 |   I ← I ∪ ΔI
5 end
6 return I
```

Example 3.7. The delta rules for the program in Example 3.4 are as follows:

$$\Delta P(X, Y) \text{ :- } E(X, Y).$$

$$\Delta P(X, Z) \text{ :- } \Delta P(X, Y), E(Y, Z).$$

The execution of semi-naïve evaluation is shown here:

	ΔP	P
iter 1	$\{(1,2),(2,3),(3,4),(4,5)\}$	$\Delta P^{(1)}$
iter 2	$\{(1,3),(2,4),(3,5)\}$	$P^{(1)} \cup \Delta P^{(2)}$
iter 3	$\{(1,4),(2,5)\}$	$P^{(2)} \cup \Delta P^{(3)}$
iter 4	$\{(1,5)\}$	$P^{(3)} \cup \Delta P^{(4)}$
iter 5	$\emptyset$	$P^{(4)}$

For consistency with the naïve algorithm, we count the initialization code in line 1 of Algorithm 4 as "iteration 1". In every iteration, although we compute the same P relation as the naïve algorithm, we only need to join ΔP with E. Because ΔP gets smaller as time goes on, every iteration runs faster in the semi-naïve algorithm.

Chapter 4

The Datalog$^\circ$ Language

In this chapter we propose a foundation for a query language that supports both recursion and aggregation. Our proposal is based on the concept of K-relations, introduced in a seminal paper by Green, Karvounarakis, and Tannen [GKT07]. In a K-relation, tuples are mapped to a fixed semiring. Standard relations (sets) are $\mathbb{B}$-relations where tuples are mapped to the Boolean semiring $\mathbb{B}$, relations with duplicates (bags) are $\mathbb{N}$-relations, sparse tensors are $\mathbb{R}$ -relations, and so on. Queries over K-relations are the familiar relational queries, where the operations $\wedge, \vee$ are replaced by the operations $\otimes, \oplus$ in the semiring; importantly, an existential quantifier $\exists$ becomes an $\oplus$-aggregate operator. K-relations are a very powerful abstraction, because they open up the possibility of adapting query processing and optimization techniques to other domains [ANR16].

Our first contribution is to introduce an extension of Datalog to K-relations. We call the language Datalog$^\circ$ (pronounced "Datalog-Oh"), where the superscript $\circ$ represents a (semi)-ring. Datalog$^\circ$ has a declarative semantics based on the least fixpoint, and supports both recursion and aggregates. We illustrate throughout this chapter its utility through several examples that are typical for recursive data processing. In order to define the least fixpoint semantics of Datalog$^\circ$, the semiring needs to be partially ordered. For this purpose, we introduce an algebraic structure called

a *Partially Ordered Pre-Semiring (POPS)*, which generalizes the more familiar naturally ordered semirings. This generalization is necessary for some applications. For example, the bill-of-material program (Example 4.6) is naturally expressed over the lifted reals, $\mathbb{R}_\perp$, which is a POPS that is not naturally ordered.

Our second contribution is to extend the *semi-naïve algorithm* to Datalog°, under certain restrictions on the POPS. This should be viewed as an illustration of the potential for applying advanced optimizations to Datalog°: in Chapter 7 we introduce a simple, yet powerful optimization technique for Datalog°, and showed, among other things, that magic set rewriting can be obtained using several applications of that rule.

The remainder of this chapter is organized as follows. We define POPS in Sec. 4.1 and give several examples. In Sec. 4.2 we define Datalog° formally, and give several examples. Sec. 4.3 presents a generalization of semi-naïve evaluation to Datalog°.

4.1 Partially Ordered Pre-Semirings (POPS)

In this section, we review the basic algebraic notions of (pre-)semirings, P-relations, and sum-product queries. We also introduce an extension called partially ordered pre-semiring (POPS).

4.1.1 (Pre-)Semirings and POPS

Definition 4.1 ((Pre-)semiring)**.** A *pre-semiring* [GM08] is a tuple $\boldsymbol{S} = (S, \oplus, \otimes, 0, 1)$ where $\oplus$ and $\otimes$ are binary operators on S for which $(S, \oplus, 0)$ is a commutative monoid, $(S, \otimes, 1)$ is a monoid, and $\otimes$ distributes over $\oplus$. When the *absorption rule* $x \otimes 0 = 0$ holds for all $x \in S$, we call $\boldsymbol{S}$ a *semiring*.[1] When $\otimes$ is commutative, then we say that the pre-semiring is *commutative*. In this dissertation we only consider commutative pre-semirings, and we will simply refer to them as pre-semirings.

[1] Some references, e.g. [KW08], define a semiring without absorption.

In any (pre-)semiring $\boldsymbol{S}$, the relation $x \preceq_S y$ defined as $\exists z : x \oplus z = y$, is a *preorder*, which means that it is reflexive and transitive, but it is not anti-symmetric in general. When $\preceq_S$ is anti-symmetric, then it is a partial order, and is called the *natural order* on $\boldsymbol{S}$; in that case we say that $\boldsymbol{S}$ is *naturally ordered*.

Example 4.1. Some simple examples of pre-semirings are the Booleans $(\mathbb{B} \stackrel{\text{def}}{=} \{0,1\}, \vee, \wedge, 0, 1)$, the natural numbers $(\mathbb{N}, +, \times, 0, 1)$, and the real numbers $(\mathbb{R}, +, \times, 0, 1)$. We will refer to them simply as $\mathbb{B}, \mathbb{N}$ and $\mathbb{R}$. The natural order on $\mathbb{B}$ is $0 \preceq_{\mathbb{B}} 1$ (or false $\preceq_{\mathbb{B}}$ true); the natural order on $\mathbb{N}$ is the same as the familiar total order $\leq$ of numbers. $\mathbb{R}$ is not naturally ordered, because $x \preceq_{\mathbb{R}} y$ holds for every $x, y \in \mathbb{R}$. Another useful example is the *tropical semiring* $\mathsf{Trop}^+ = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$, where the natural order $x \preceq y$ is the *reverse* order $x \geq y$ on $\mathbb{R}_+ \cup \{\infty\}$.

A key idea we introduce in this dissertation is the decoupling of the partial order from the algebraic structure of the (pre-)semiring. The decoupling allows us to inject a partial order when the (pre-)semiring is not naturally ordered, or when we need a *different* order from the natural order.

Definition 4.2 (POPS)**.** A *partially ordered pre-semiring* (POPS) is a tuple $\boldsymbol{P} = (P, \oplus, \otimes, 0, 1, \sqsubseteq)$, where $(P, \oplus, \otimes, 0, 1)$ is a pre-semiring, $(P, \sqsubseteq)$ is a poset, and $\oplus, \otimes$ are *monotone*[2] operators under $\sqsubseteq$. In this dissertation, we will assume that every poset $(P, \sqsubseteq)$ has a minimum element denoted by $\bot$.

A POPS satisfies the identities $\bot \oplus \bot = \bot$ and $\bot \otimes \bot = \bot$, because, by monotonicity and the fact that $(P, \oplus, 0)$ and $(P, \otimes, 1)$ are commutative monoids, we have $\bot \oplus \bot \sqsubseteq \bot \oplus 0 = \bot$, and $\bot \otimes \bot \sqsubseteq \bot \otimes 1 = \bot$. We say that the multiplicative operator $\otimes$ is *strict* if the identity $x \otimes \bot = \bot$ holds for every $x \in P$. Throughout this dissertation we will assume that $\otimes$ is strict, unless otherwise stated.

[2]Monotonicity means $x \sqsubseteq x'$ and $y \sqsubseteq y'$ imply $x \oplus y \sqsubseteq x' \oplus y'$ and $x \otimes y \sqsubseteq x' \otimes y'$.

4.1.2 Polynomials over POPS

Fix a POPS $\boldsymbol{P} = (P, \oplus, \otimes, 0, 1, \sqsubseteq)$. We are interested in vector-valued multivariate functions on P defined by composing $\oplus$ and $\otimes$. These functions are multivariate polynomials. Writing polynomials in $\boldsymbol{P}$ using the symbols $\oplus, \otimes$ is cumbersome and difficult to parse. Consequently, we replace them with $+, \cdot$ when the underlying POPS $\boldsymbol{P}$ is clear from context; furthermore, we will also abbreviate a multiplication $a \cdot b$ with ab. As usual, a^k denotes the product of k copies of a, where $a^0 \stackrel{\text{def}}{=} 1$.

Let $x_1, \ldots, x_N$ be N variables. A *monomial* (on $\boldsymbol{P}$) is an expression of the form:

$$m \stackrel{\text{def}}{=} c \cdot x_1^{k_1} \cdot \cdots \cdot x_N^{k_N} \tag{4.1}$$

where $c \in P$ is some constant. Its *degree* is $\deg(m) \stackrel{\text{def}}{=} k_1 + \cdots + k_N$. A (multivariate) *polynomial* is a sum:

$$f(x_1, \ldots, x_N) \stackrel{\text{def}}{=} m_1 + m_2 + \cdots + m_q \tag{4.2}$$

where each m_i is a monomial. The polynomial f defines a function $P^N \to P$ in the obvious way, and, with some abuse, we will denote by f both the polynomial and the function it defines. Notice that f is monotone in each of its arguments.

A *vector-valued polynomial function* on $\boldsymbol{P}$ is a function $\boldsymbol{f} : P^N \to P^M$ whose component functions are polynomials. In particular, the vector-valued polynomial function is a *tuple of polynomials* $\boldsymbol{f} = (f_1, \ldots, f_M)$ where each f_i is a polynomial in variables $x_1, \ldots, x_N$.

We note a subtlety when dealing with POPS: when the POPS is not a semiring, then we cannot "remove" monomials by setting their coefficient $c = 0$, because 0 is not absorbing. Instead, we must ensure that they are not included in the polynomial (4.2). For example, consider the POPS of the lifted reals, $\boldsymbol{R}_\perp$, and the linear polynomial $f(x) = ax + b$. If we set $a = 0$, we do not obtain the

constant function $g(x) = b$, because $f(\bot) = a\bot + b = \bot + b = \bot \neq g(\bot) = b$. We just have to be careful to not include monomials we don't want.

4.1.3 P-Relations

Fix a relational vocabulary, $\sigma = \{R_1, \ldots, R_m\}$, where each R_i is a relation name, with an associated arity. Let D be an infinite domain of constants, for example the set of all strings over a fixed alphabet, or the set of natural numbers. Recall that an instance of the relation R_i is a finite subset of $D^{\text{arity}(R_i)}$, or equivalently, a mapping $D^{\text{arity}(R_i)} \to \mathbb{B}$ assigning 1 to all tuples present in the relation. Following [GKT07], we generalize this abstraction from $\mathbb{B}$ to an arbitrary POPS $\boldsymbol{P}$.

Given a relation name $R_i \in \sigma$, a *ground atom* of R_i is an expression of the form $R_i(\boldsymbol{u})$, where $\boldsymbol{u} \in D^{\text{arity}(R_i)}$. Let $\mathsf{GA}(R_i, D)$ denote the set of all ground atoms of R_i, and $\mathsf{GA}(\sigma, D) \stackrel{\text{def}}{=} \bigcup_i \mathsf{GA}(R_i, D)$ denote the set of all ground atoms over the entire vocabulary σ. The set $\mathsf{GA}(\sigma, D)$ is the familiar Herbrand base in logic programming. Note that each ground atom is prefixed by a relation name.

Let $\boldsymbol{P}$ be a POPS. A *$\boldsymbol{P}$-instance for σ* is a function $I : \mathsf{GA}(\sigma, D) \to \boldsymbol{P}$ with *finite support*, where the support is defined as the set of ground atoms that are mapped to elements *other than* $\bot$. For example, if $\boldsymbol{P}$ is a naturally ordered semiring, then the support of the function I is the set of ground atoms assigned to a non-zero value. The *active domain* of the instance I, denoted by $\mathsf{ADom}(I)$, is the finite set $\mathsf{ADom}(I) \subseteq D$ of all constants that occur in the support of I. We denote by $\mathsf{Inst}(\sigma, D, \boldsymbol{P})$ the set of $\boldsymbol{P}$-instances over the domain D. When σ consists of a single relation name, then we call I a *$\boldsymbol{P}$-relation.*

An equivalent way to define a $\boldsymbol{P}$-instance is as a function $I : \mathsf{GA}(\sigma, D_0) \to \boldsymbol{P}$, for some finite subset $D_0 \subseteq D$; by convention, this function is extended to the entire set $\mathsf{GA}(\sigma, D)$ by setting $I(a) := \bot$ for all $a \in \mathsf{GA}(\sigma, D) \setminus \mathsf{GA}(\sigma, D_0)$. The set $\mathsf{Inst}(\sigma, D_0, \boldsymbol{P})$ is isomorphic to P^N, where $N = |\mathsf{GA}(\sigma, D_0)|$, and, throughout this dissertation, we will identify a $\boldsymbol{P}$-instance with a tuple in

P^N.

Thus, a $\boldsymbol{P}$-instance involves two domains: D, which is called the *key space*, and the POPS $\boldsymbol{P}$, which is called the *value space*. For some simple illustrations, a $\mathbb{B}$-relation is a standard relation where every ground tuple is either true or false, while an $\mathbb{R}^+$-relation[3] is a sparse tensor.

4.1.4 (Sum-)Sum-Product Queries on POPS

In the Boolean world, conjunctive queries and union of conjunctive queries are building-blocks for relational queries. Analogously, in the POPS world, we introduce the concepts of *sum-product queries* and *sum-sum-product queries*. In the simplest setting, these queries have been studied in other communities (especially AI and machine learning as reviewed below). In our setting, we need to introduce one extra feature called "conditional", in order to cope with the fact that 0 is not absorptive.

Fix two disjoint vocabularies, $\sigma, \sigma_{\mathbb{B}}$; the relation names in σ will be interpreted over a POPS $\boldsymbol{P}$, while those in $\sigma_{\mathbb{B}}$ will be interpreted over the Booleans. Let D be a domain, and $V = \{X_1, \ldots, X_p\}$ a set of "key variables" whose values are over the key space D. They should not be confused with variables used in polynomials, which are interpreted over the POPS $\boldsymbol{P}$; we refer to the latter as "value variables" to contrast them with the key variables. We use upper case for key variables, and lower case for value variables. A *σ-atom* is an expression of the form $R_i(\boldsymbol{X})$, where $R_i \in \sigma$ and $\boldsymbol{X} \in (V \cup D)^{\text{arity}(R_i)}$.

Definition 4.3. A (conditional) *sum-product query*, or *sum-product rule* is an expression of the form

$$T(X_1, \ldots, X_k) \text{ :- } \bigoplus_{X_{k+1}, \ldots, X_p} \{R_1(\boldsymbol{X}_1) \otimes \cdots \otimes R_m(\boldsymbol{X}_m) \mid \Phi(V)\} \tag{4.3}$$

[3]Recall the $\mathbb{R}$ semiring is not a POPS.

where T is a new relation name of arity k, each $R_j(\boldsymbol{X}_j)$ is a σ-atom, and Φ is a first-order (FO) formula over $\sigma_{\mathbb{B}}$, whose free variables are in $V = \{X_1, \ldots, X_p\}$. The LHS of :- is called the *head*, and the RHS the *body* of the rule. The variables $X_1, \ldots, X_k$ are called *free variables* of the query (also called *head variables*), and $X_{k+1}, \ldots, X_p$ are called *bound variables*.

Without the conditional term Φ, the problem of computing efficiently sum-products over semirings has been extensively studied both in the database and in the AI literature. In databases, the query optimization and evaluation problem is a special case of sum-product computation over the value-space of Booleans (set semantics) or natural numbers (bag semantics). The functional aggregate queries (FAQ) framework [ANR16] extends the formulation to queries over multiple semirings. In AI, this problem was studied by Shenoy and Schafer [SS88], Dechter [Dec97], Kohlas and Wilson [KW08] and others. Surveys and more examples can be found in [AM00, Koh03]. These methods use a sparse representation of the $\boldsymbol{P}$-relations, consisting of a collection of the tuples in their support.

The use of a conditional Φ in the sum-product is non-standard, but it is necessary for sum-product expressions over a POPS that is not a semiring, as we illustrate next.

Example 4.2. Let $E(X, Y)$ be a $\mathbb{B}$-relation (i.e. a standard relation), representing a graph. The following sum-product expression over $\mathbb{B}$ computes all pairs of nodes connected by a paths of length 2:

$$T(X, Z) \text{ :- } \exists_Y \left(E(X, Y) \wedge E(Y, Z)\right)$$

This is a standard conjunctive query [AHV95] (where the semantics of quantification over Y is explicitly written). Here $\sigma = \{E\}$, and $\sigma_{\mathbb{B}} = \emptyset$: we do not need the conditional term Φ yet.

For the second example, consider the same graph given by $E(X, Y)$, and let $C(X)$ be an $\mathbb{R}_\perp$-relation associating to each node X a real number representing a cost, or $\perp$ if the cost is unknown;

now $\sigma = \{C\}, \sigma_{\mathbb{B}} = \{E\}$. The following sum-product expression computes the total costs of all neighbors of X:

$$T(X) \text{ :- } \sum_Y \{C(Y) \mid E(X,Y)\} \tag{4.4}$$

Usually, conditionals are avoided by using an indicator function $1_{E(X,Y)}$, which is defined to be 1 when $E(X,Y)$ is true and 0 otherwise, and writing the rule as $T(X)$:- $\sum_Y \left(1_{E(X,Y)} \cdot C(Y)\right)$. But this does not work in $\mathbb{R}_\perp$, because, when Y is mapped to a non-neighboring node which so happens to have an unknown cost (while all neighbors' costs are known), we have $C(Y) = \perp$. In this case, $1_{E(X,Y)} \cdot C(Y) = 0 \cdot \perp = \perp$, instead of 0. Since $x + \perp = \perp$ in $\mathbb{R}_\perp$, the result is also $\perp$. One may ask whether we can re-define the POPS $\mathbb{R}_\perp$ so that $\perp \cdot 0 = 0$, but we show in Lemma 4.1 that this is not possible. The explicit conditional in (4.4) allows us to restrict the range of Y only to the neighbors of X.

We now formally define the semantics of (conditional) sum-product queries. Due to the subtlety with POPS, we need to consider an alternative approach to evaluating the results of sum-product queries: first compute the *provenance polynomials* of the query (4.3) to obtain the component polynomials of a vector-valued function, then evaluate these polynomials. The provenance polynomials, or simply provenance, are also called lineage, or groundings in the literature [GKT07].

Given an input instance $I_{\mathbb{B}} \in \mathsf{Inst}(\sigma_{\mathbb{B}}, D, \mathbb{B})$, $I \in \mathsf{Inst}(\sigma, D, \boldsymbol{P})$, and let $D_0 \subseteq D$ be the finite set consisting of their active domains and all constants occurring in the sum-product expression (4.3). Let $N \stackrel{\text{def}}{=} |\mathsf{GA}(\sigma, D_0)|$ and $M \stackrel{\text{def}}{=} |\mathsf{GA}(T, D_0)| = |D_0|^k$ be the number of input ground atoms and output ground atoms respectively.

To each of the N input atoms $\mathsf{GA}(\sigma, D_0)$ we associate a unique POPS variable $x_1, \ldots, x_N$. (Recall that we use upper case for key variables and lower case for value variables.) Abusing notation,

we also write $x_{R(\boldsymbol{u})}$ to mean the variable associated to the ground atom $R(\boldsymbol{u})$. Recall that V is the set of key variables, and we define a *valuation* to be a function $\theta : V \to D_0$. When applied to the body of the rule (4.3), the valuation θ defines the following monomial:

$$\theta(\text{body}) \stackrel{\text{def}}{=} x_{R_1(\theta(\boldsymbol{X}_1))} \cdot x_{R_2(\theta(\boldsymbol{X}_2))} \cdot \ldots \cdot x_{R_m(\theta(\boldsymbol{X}_m))} \tag{4.5}$$

The *provenance polynomial* [GKT07] of the output tuple $T(\boldsymbol{a}) \in \mathsf{GA}(T, D_0)$ is the following:

$$f_{T(\boldsymbol{a})}(X_1, \ldots, X_k) \stackrel{\text{def}}{=} \sum_{\substack{\theta: V \to D_0, \\ \theta(X_1,\ldots,X_k)=\boldsymbol{a}, \\ I_{\mathbb{B}} \models \Phi[\theta]}} \theta(\text{body}) \tag{4.6}$$

In other words, we consider only valuations θ that map the head variables to the tuple $\boldsymbol{a}$ and satisfy the FO sentence Φ. There are M provenance polynomials, one for each tuple in $\mathsf{GA}(T, D_0)$, and they define an M-tuple of polynomials in N variables, $\boldsymbol{f}$, which in turn defines a function $\boldsymbol{f} : \boldsymbol{P}^N \to \boldsymbol{P}^M$. The semantics of the query (4.3) is defined as the value of this polynomial on the input instance $I \in \mathsf{Inst}(\sigma, D_0, \boldsymbol{P})$, when viewed as a tuple $I \in \boldsymbol{P}^N$.

Note that, once we have constructed the provenance polynomial, we no longer need to deal with the conditional Φ, because the grounded version does not have Φ anymore. In most of the rest of the chapter we will study properties of vector-valued functions whose components are these provenance polynomials.

We notice that, as defined, our semantics depends on the choice of the domain D_0: if we used a larger finite domain $D_0' \supseteq D_0$, then the provenance polynomials will include additional spurious monomials, corresponding to the spurious grounded tuples in D_0'. Traditionally, these spurious monomials are harmless, because their value is 0. However, in our setting, their value is $\bot$, and they may change the result. This is precisely the role of the conditional Φ in (4.3): to control the

range of the variables and ensure that the semantics is *domain independent*, meaning a query produces the same result for every domain that contains the active domain. All examples in this dissertation are written such that they are domain independent.

Finally, (conditional) sum-sum-product queries are defined in the natural way:

Definition 4.4. A (conditional) *sum-sum-product query* or *sum-sum-product rule* has the form:

$$T(X_1, \ldots, X_k) \text{ :- } E_1 \oplus \cdots \oplus E_q \tag{4.7}$$

where $E_1, E_2, \ldots, E_q$ are the bodies of sum-product expressions (4.3), each with the same free variables $X_1, \ldots, X_k$.

The provenance polynomials of a sum-sum-product query are defined as the sum of the provenance polynomials of the expressions $E_1, \ldots, E_q$.

For a simple illustration, we show a modification of (4.4) where we include in the total sum $T(X)$ the cost of X:

$$T(X) \text{ :- } C(X) + \sum_Y \{C(Y) \mid E(X, Y)\}$$

4.1.5 Properties and Examples of POPS

We end this section by presenting several properties of POPS and illustrating them with a few examples.

Extending Pre-semirings to POPS

If $\boldsymbol{S}$ is a pre-semiring, then we say that a POPS $\boldsymbol{P}$ *extends* $\boldsymbol{S}$ if $S \subseteq P$ (S and P are their domains), and the operations $\oplus, \otimes, 0, 1$ in $\boldsymbol{S}$ are the same as those in $\boldsymbol{P}$. We describe three procedures to extend a

pre-semiring $\boldsymbol{S}$ to a POPS $\boldsymbol{P}$, inspired by abstract interpretations in programming languages [CC77].

Representing Undefined The *lifted POPS* is $\boldsymbol{S}_\bot = (S \cup \{\bot\}, \oplus, \otimes, 0, 1, \sqsubseteq)$, where $x \sqsubseteq y$ iff $x = \bot$ or $x = y$, and the operations $\oplus, \otimes$ are extended to $\bot$ by setting $x \oplus \bot = x \otimes \bot = \bot$. Notice that $\boldsymbol{S}_\bot$ is not a semiring, because 0 is not absorbing: $0 \otimes \bot \neq 0$. Here $\bot$ represents *undefined*.

Representing Contradiction The *completed POPS* is $\boldsymbol{S}_\bot^\top = (S \cup \{\bot, \top\}, \oplus, \otimes, 0, 1, \sqsubseteq)$, where $x \sqsubseteq y$ iff $x = \bot$, $x = y$, or $y = \top$ and the operations $\oplus, \otimes$ are extended to $\bot, \top$ as follows: $x \oplus \bot = x \otimes \bot = \bot$ for all x (including $x = \top$), and $x \oplus \top = x \otimes \top = \top$ for all $x \neq \bot$. Here $\bot, \top$ represent undefined and contradiction respectively. Intuitively: $\bot$ is the empty set $\emptyset$, each element $x \in \boldsymbol{S}$ is a singleton set consisting of one value, and $\top$ is the entire set S.

Representing Incomplete Values More generally, define $\mathcal{P}(\boldsymbol{S}) = (\mathcal{P}(S), \oplus, \otimes, \{0\}, \{1\}, \subseteq)$. It consists of all subsets of $\boldsymbol{S}$, ordered by set inclusion, where the operations $\oplus, \otimes$ are extended to sets, e.g. $A \oplus B = \{x \oplus y \mid x \in A, y \in B\}$. Here $\bot = \emptyset$ represents undefined, $\top = S$ represents contradiction, and, more generally, every set represents some degree of incompleteness.

A lifted POPS $\boldsymbol{S}_\bot$ is never a semiring, because $\bot \otimes 0 = \bot$, and the reader may ask whether there exists an alternative way to extend it to a POPS that is also a semiring, i.e. $0 \otimes x = 0$. For example, we can define $\mathbb{N} \cup \{\bot\}$ as a semiring by setting $x + \bot = \bot$ for all x, $0 \cdot \bot = 0$ and $x \cdot \bot = \bot$ for $x > 0$: one can check that the semiring laws hold. However, this is not possible in general. We prove:

Lemma 4.1. If $\boldsymbol{S}$ is any POPS extension of $(\mathbb{R}, +, \cdot, 0, 1)$, then $\boldsymbol{S}$ is not a semiring, i.e. it fails the absorption law $0 \cdot x = 0$.

Proof. Let $\boldsymbol{S} = (S, +, \cdot, 0, 1, \sqsubseteq)$ be any POPS that is an extension of $\mathbb{R}$. In particular $\mathbb{R} \subseteq S$ and S

has a minimal element $\bot$. Since 0, 1 are additive and multiplicative identities, we have:

$$\bot + 0 = \bot \qquad\qquad \bot \cdot 1 = \bot$$

We claim that the following more general identities hold:

$$\forall x \in \mathbb{R} : \bot + x = \bot \qquad\qquad \forall x \in \mathbb{R} \setminus \{0\} : \bot \cdot x = \bot$$

To prove the first identity, we use the fact that + is monotone in $\boldsymbol{S}$ and $\bot$ is the smallest element, and derive $\bot + x \sqsubseteq (\bot + (y - x)) + x = \bot + y$ for all $x, y \in \mathbb{R}$. This implies $\bot + x = \bot + y$ for all x, y and the claim follows by setting $y = 0$. The proof of the second identity is similar: first observe that $\bot \cdot x \sqsubseteq (\bot \cdot \frac{y}{x}) \cdot x = \bot \cdot y$ hence $\bot \cdot x = \bot \cdot y$ for all $x, y \in \mathbb{R} \setminus \{0\}$, and the claim follows by setting $y = 1$.

Assuming $\boldsymbol{S}$ is a semiring, it satisfies the absorption law: $\bot \cdot 0 = 0$. We prove now that $0 = \bot$. Choose any $x \in \mathbb{R} \setminus \{0\}$, and derive:

$$\bot = \bot + \bot = (\bot \cdot x) + (\bot \cdot (-x)) = \bot \cdot (x + (-x)) = \bot \cdot 0 = 0.$$

The middle identity follows from distributivity. From $0 = \bot$, we conclude that 0 is the smallest element in $\boldsymbol{S}$. Then, for every $x \in \mathbb{R}$, we have $x = x + 0 \sqsubseteq x + (-x) = 0$, which implies $x = 0$, $\forall x \in \mathbb{R}$, which is a contradiction. Thus, $\boldsymbol{S}$ is not a semiring. □

The POPS THREE

Consider the following POPS: $\mathsf{THREE} \stackrel{\text{def}}{=} (\{\bot, 0, 1\}, \vee, \wedge, 0, 1, \leq_k)$, where:

- $\vee, \wedge$ have the semantics of 3-valued logic [Fit85]. More precisely, define the *truth ordering*

$0 \leq_t \bot \leq_t 1$ and set $x \vee y \stackrel{\text{def}}{=} \max_t(x, y)$, $x \wedge y \stackrel{\text{def}}{=} \min_t(x, y)$. We note that this is precisely Kleene's three-valued logic [Fit91].

- $\leq_k$ is the *knowledge order*, defined as $\bot <_k 0$ and $\bot <_k 1$.

THREE is not the same as the lifted Booleans, $\mathbb{B}_\bot$, because in the latter $0 \wedge \bot = \bot$, while in THREE we have $0 \wedge \bot = 0$. [ANP+22] show how to use THREE to support negation in Datalog°.

Stable Semirings

We illustrate two examples of semirings that are *stable*, a property defined formally in [ANP+22]. In short, a semiring S is stable if for every $u \in S$, the sequence $u^{(i)} \stackrel{\text{def}}{=} 1 \oplus u \oplus u^2 \oplus \cdots \oplus u^i$ converges after p terms, where $u^i \stackrel{\text{def}}{=} u \otimes u \otimes \cdots \otimes u$ (i times). That is, $u^{(p)} = u^{(p+1)}$ for some p. Both examples are adapted from [GM08, Example 7.1.4] and [GM08, Chapt.8, Sec.1.3.2] respectively. If A is a set and $p \geq 0$ a natural number, then we denote by $\mathcal{P}_p(A)$ the set of subsets of A of size p, and by $\mathcal{B}_p(A)$ the set of bags of A of size p. We also define

$$\mathcal{P}_{\mathtt{fin}}(A) \stackrel{\text{def}}{=} \bigcup_{p \geq 0} \mathcal{P}_p(A) \qquad\qquad \mathcal{B}_{\mathtt{fin}}(A) \stackrel{\text{def}}{=} \bigcup_{p \geq 0} \mathcal{B}_p(A).$$

We denote bags as in $\{\{a, a, a, b, c, c\}\}$. Given $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{P}_{fin}(\mathbb{R}_+ \cup \infty)$, we denote by:

$$\boldsymbol{x} \cup \boldsymbol{y} \stackrel{\text{def}}{=} \text{set union of } \boldsymbol{x}, \boldsymbol{y} \qquad\qquad \boldsymbol{x} + \boldsymbol{y} \stackrel{\text{def}}{=} \{u + v \mid u \in \boldsymbol{x}, v \in \boldsymbol{y}\}$$

Similarly, given $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{B}_{fin}(\mathbb{R}_+ \cup \infty)$, we denote by:

$$\boldsymbol{x} \uplus \boldsymbol{y} \stackrel{\text{def}}{=} \text{bag union of } \boldsymbol{x}, \boldsymbol{y} \qquad\qquad \boldsymbol{x} + \boldsymbol{y} \stackrel{\text{def}}{=} \{\{u + v \mid u \in \boldsymbol{x}, v \in \boldsymbol{y}\}\}$$

Example 4.3. For any bag $\boldsymbol{x} = \{\{x_0, x_1, \ldots, x_n\}\}$, where $x_0 \leq x_1 \leq \ldots \leq x_n$, and any $p \geq 0$, define:

$$\min_p(\boldsymbol{x}) \stackrel{\text{def}}{=} \{\{x_0, x_1, \ldots, x_{\min(p,n)}\}\}$$

In other words, $\min_p$ returns the smallest $p+1$ elements of the bag $\boldsymbol{x}$. Then, for any $p \geq 0$, the following is a semiring:

$$\mathsf{Trop}_p^+ \stackrel{\text{def}}{=} (\mathcal{B}_{p+1}(\mathbb{R}_+ \cup \{\infty\}), \oplus_p, \otimes_p, \mathbf{0}_p, \mathbf{1}_p)$$

where:

$$\boldsymbol{x} \oplus_p \boldsymbol{y} \stackrel{\text{def}}{=} \min_p(\boldsymbol{x} \uplus \boldsymbol{y}) \qquad \mathbf{0}_p \stackrel{\text{def}}{=} \{\infty, \infty, \ldots, \infty\}$$
$$\boldsymbol{x} \otimes_p \boldsymbol{y} \stackrel{\text{def}}{=} \min_p(\boldsymbol{x} + \boldsymbol{y}) \qquad \mathbf{1}_p \stackrel{\text{def}}{=} \{0, \infty, \ldots, \infty\}$$

For example, if $p = 2$ then $\{\{3,7,9\}\} \oplus_2 \{\{3,7,7\}\} = \{\{3,3,7\}\}$ and $\{\{3,7,9\}\} \otimes_2 \{\{3,7,7\}\} = \{\{6,10,10\}\}$. The following identities are easily checked, for any two finite bags $\boldsymbol{x}, \boldsymbol{y}$:

$$\min_p(\min_p(\boldsymbol{x}) \uplus \min_p(\boldsymbol{y})) = \min_p(\boldsymbol{x} \uplus \boldsymbol{y}) \qquad \min_p(\min_p(\boldsymbol{x}) + \min_p(\boldsymbol{y})) = \min_p(\boldsymbol{x} + \boldsymbol{y}) \tag{4.8}$$

This implies that, an expression in the semiring Trop_p^+ can be computed as follows. First, convert $\oplus, \otimes$ to $\uplus, +$ respectively, compute the resulting bag, then apply $\min_p$ only once, on the final result. When $p = 0$, then $\mathsf{Trop}_p^+ = \mathsf{Trop}^+$.

Example 4.4. Fix a real number $\eta \geq 0$, and denote by $\mathcal{P}_{\leq \eta}(\mathbb{R}_+ \cup \{\infty\})$ the set of nonempty, finite sets $\boldsymbol{x} = \{x_0, x_1, \ldots, x_p\}$ where $\min(\boldsymbol{x}) \leq \max(\boldsymbol{x}) \leq \min(\boldsymbol{x}) + \eta$. Given any finite set

$\boldsymbol{x} \in \mathcal{P}_{\mathtt{fin}}(\mathbb{R}_+ \cup \{\infty\})$, we define

$$\min_{\leq\eta}(\boldsymbol{x}) \stackrel{\text{def}}{=} \{u \mid u \in \boldsymbol{x}, u \leq \min(\boldsymbol{x}) + \eta\}$$

In other words, $\min_{\leq\eta}$ retains from the set $\boldsymbol{x}$ only the elements at distance $\leq \eta$ from its minimum. The following is a semiring:

$$\mathsf{Trop}^+_{\leq\eta} \stackrel{\text{def}}{=} (\mathcal{P}_{\leq\eta}(\mathbb{R}_+ \cup \{\infty\}), \oplus_{\leq\eta}, \otimes_{\leq\eta}, \mathbf{0}_{\leq\eta}, \mathbf{1}_{\leq\eta})$$

where:

$$\begin{aligned} \boldsymbol{x} \oplus_{\leq\eta} \boldsymbol{y} &\stackrel{\text{def}}{=} \min_{\leq\eta}(\boldsymbol{x} \cup \boldsymbol{y}) & \mathbf{0}_{\leq\eta} &\stackrel{\text{def}}{=} \{\infty\} \\ \boldsymbol{x} \otimes_{\leq\eta} \boldsymbol{y} &\stackrel{\text{def}}{=} \min_{\leq\eta}(\boldsymbol{x} + \boldsymbol{y}) & \mathbf{1}_{\leq\eta} &\stackrel{\text{def}}{=} \{0\} \end{aligned}$$

For example, if $\eta = 6.5$ then: $\{3,7\} \oplus_{\leq\eta} \{5,9,10\} = \{3,5,7,9\}$ and $\{1,6\} \otimes_{\leq\eta} \{1,2,3\} = \{2,3,4,7,8\}$. The following identities are easily checked, for any two finite sets $\boldsymbol{x}, \boldsymbol{y}$:

$$\min_{\leq\eta}(\min_{\leq\eta}(\boldsymbol{x}) \cup \min_{\leq\eta}(\boldsymbol{y})) = \min_{\leq\eta}(\boldsymbol{x} \cup \boldsymbol{y}) \quad \min_{\leq\eta}(\min_{\leq\eta}(\boldsymbol{x}) + \min_{\leq\eta}(\boldsymbol{y})) = \min_{\leq\eta}(\boldsymbol{x} + \boldsymbol{y}) \tag{4.9}$$

It follows that expressions in $\mathsf{Trop}^+_{\leq\eta}$ can be computed as follows: first convert $\oplus, \otimes$ to $\cup, +$ respectively, compute the resulting set, and apply the $\min_{\leq\eta}$ operator only once, on the final result. Notice that, when $\eta = 0$, then we recover again $\mathsf{Trop}^+_{\leq\eta} = \mathsf{Trop}^+$.

The reader may wonder why Trop^+_p is defined to consist of bags of $p+1$ numbers, while $\mathsf{Trop}^+_{\leq\eta}$ is defined on sets. The main reason is for consistency with [GM08]. We could have defined either semirings on either sets or bags, and both identities (4.8) and (4.9) continue to hold, which is

sufficient to prove the semiring identities. However, the *stability* property, defined and proved in [ANP+22], holds for $\mathsf{Trop}^+_{\leq \eta}$ only if it is defined over sets; in contrast, Trop^+_p is stable for either sets or bags.

4.2 Datalog°

We define here the language Datalog°, which generalizes Datalog from traditional relations to $\boldsymbol{P}$-relations, for some POPS $\boldsymbol{P}$. As in Datalog, the input relations to the program will be called Extensional Database Predicates, EDB, and the computed relations will be called Intensional Database Predicates, IDB. Each EDB can be either a $\boldsymbol{P}$-relation, or standard relation, i.e. a $\mathbb{B}$-relation, and we denote by $\sigma \stackrel{\text{def}}{=} \{R_1, \ldots, R_m\}$ and $\sigma_{\mathbb{B}} \stackrel{\text{def}}{=} \{B_1, \ldots, B_k\}$ the two vocabularies. All IDBs are $\boldsymbol{P}$-relations, and their vocabulary is denoted by $\tau = \{T_1, \ldots, T_n\}$.

A Datalog° program Π consists of n sum-sum-product rules $r_1, \ldots, r_n$ (as in Definition 4.4), where each rule r_i has the IDB T_i in the head:

$$\begin{aligned} r_1 &: T_1(\ldots) \text{ :- } E_{11} \oplus E_{12} \oplus \cdots, \\ &\ldots \\ r_n &: T_n(\ldots) \text{ :- } E_{n1} \oplus E_{n2} \oplus \cdots, \end{aligned} \tag{4.10}$$

and each E_{ij} is a sum-product expression as in (4.3). The program Π is said to be *linear* if each sum-product expression E_{ij} contains at most one IDB predicate.

4.2.1 Least Fixpoint of the Immediate Consequence Operator

The *Immediate Consequence Operator* (ICO) of a program Π is the function $F : \mathsf{Inst}(\sigma, D, \boldsymbol{P}) \times \mathsf{Inst}(\sigma_{\mathbb{B}}, D, \mathbb{B}) \times \mathsf{Inst}(\tau, D, \boldsymbol{P}) \to \mathsf{Inst}(\tau, D, \boldsymbol{P})$, that takes as input an instance $(I, I_{\mathbb{B}})$ of the EDBs

and an instance J of the IDBs, and computes a new instance $F(I, I_{\mathbb{B}}, J)$ of the IDBs by evaluating each sum-sum-product rule. By fixing the EDBs, we will view the ICO as function from IDBs to IDBs, written as $F(J)$. We define the *semantics* of the Datalog° program (4.10) as the least fixpoint of the ICO F, when it exists.

Algorithm 5: Naïve evaluation for Datalog°

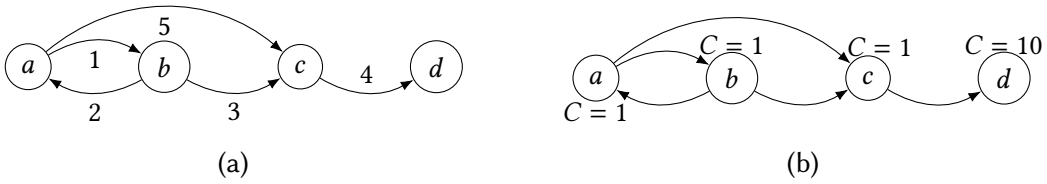
```
J^(0) ← ⊥;                 // In a naturally ordered semiring this is the same as J^(0) ← 0
for t ← 0 to ∞ do
    J^(t+1) ← F(J^(t));
    if J^(t+1) = J^(t) then
        Break
    end
end
return J^(t)
```

The Naïve Algorithm for evaluating Datalog° is shown in Algorithm 5, and it is quite similar to that for standard, positive Datalog with set semantics. We start with all IDBs at $\bot$, then repeatedly apply the ICO F, until we reach a fixpoint. The algorithm computes the increasing sequence $\bot \sqsubseteq F(\bot) \sqsubseteq F^{(2)}(\bot) \sqsubseteq \cdots$ When the algorithm terminates we say that it *converges*; in that case it returns the least fixpoint of F. Otherwise we say that the algorithm *diverges*.

While every pure Datalog program is guaranteed to have a least fixpoint, this no longer holds for Datalog° programs. [ANP+22] fully characterizes the POPS that ensure the convergence of every Datalog° program. In short, every Datalog° program converges over a POPS $\boldsymbol{P}$ if and only if $\boldsymbol{P}$ is *stable*, a property defined in [ANP+22].

4.2.2 Examples

We illustrate Datalog° with two examples. When the POPS $\boldsymbol{P}$ is a naturally ordered semiring, then we will use the following *indicator function* $[C]_0^1$, which maps a Boolean condition C to either



(a) (b)

Figure 4.1: A graph illustrating Example 4.5 (a) and Example 4.6 (b)

$0 \in \boldsymbol{P}$ or $1 \in \boldsymbol{P}$, depending on whether C is false or true. We write the indicator function simply as $[C]$, when the values $0, 1$ are clear from the context. An indicator function can be desugared by replacing $\{[C] \otimes P_1 \otimes \cdots \otimes P_k \mid \Phi\}$ with $\{P_1 \otimes \cdots \otimes P_k \mid \Phi \wedge C\}$. When $\boldsymbol{P}$ is not naturally ordered, then we will not use indicator functions, see Example 4.2.

Example 4.5. Let the EDB and IDB vocabularies be $\sigma = \{E\}$ and $\tau = \{L\}$, where E is binary and L is unary. Consider the following Datalog° program:

$$L(X) \text{ :- } [X = a] \oplus \bigoplus_{Z} (L(Z) \otimes E(Z, X)) \tag{4.11}$$

where $a \in D$ is some constant in the domain. We show three different interpretations of the program, over three different naturally ordered semirings. First, we interpret it over the semiring of Booleans. In this case, the program can be written in a more familiar notation:

$$L(X) \text{ :- } [X = a]_0^1 \vee \exists_Z (L(Z) \wedge E(Z, X))$$

This is the reachability program, which computes the set of nodes X reachable from the node a. The indicator function $[X = a]_0^1$ returns 0 or 1, depending on whether $X \neq a$ or $X = a$.

Next, let's interpret it over Trop^+. In that case, the indicator function $[X = a]_\infty^0$ returns ∞

when $X \neq a$, and returns 0 when $X = a$. The program becomes:

$$L(X) \text{ :- } \min\left((\texttt{if } X = a \texttt{ then 0 else } \infty), \min_Z(L(Z) + E(Z,X))\right)$$

This program solves the Single-Source-Shortest-Path (SSSP) problem with source vertex a. Consider the graph in Fig. 4.1(a). The active domain consists of the constants a, b, c, d, and the naïve evaluation algorithm converges after 5 steps, as shown here:

	$L(a)$	$L(b)$	$L(c)$	$L(d)$
$L^{(0)}$	∞	∞	∞	∞
$L^{(1)}$	0	∞	∞	∞
$L^{(2)}$	0	1	5	∞
$L^{(3)}$	0	1	4	9
$L^{(4)}$	0	1	4	8
$L^{(5)}$	0	1	4	8

Third, let's interpret it over Trop_p^+, defined in Example 4.3. Assume for simplicity that $p = 1$. In that case the program computes, for each node X, the bag $\{\{l_1, l_2\}\}$ of the lengths of the two shortest paths from a to X. The indicator function $[X = a]$ is equal to $\{\{0, \infty\}\}$ when $X = a$, and equal to $\{\{\infty, \infty\}\}$ otherwise. The reader may check that the program converges to:

$$L(a) = \{\{0, 3\}\} \qquad L(b) = \{\{1, 4\}\} \qquad L(c) = \{\{4, 5\}\} \qquad L(d) = \{\{8, 9\}\}$$

Finally, we can interpret it over $\mathsf{Trop}_{\leq\eta}^+$, the semiring in Example 4.4. In that case the program computes, for each X, the set of all possible lengths of paths from a to X that are no longer than the shortest path plus η.

Example 4.6. A classic problem that requires the interleaving of recursion and aggregation is the bill-of-material (see, e.g., [ZYI+18]), where we are asked to compute, for each part X, the total cost of X, of all sub-parts of X, all sub-sub-parts of X, etc. The EDB and IDB schemas are $\sigma_{\mathbb{B}} = \{E\}$, $\sigma = \{C\}$, $\tau = \{T\}$. The relation $E(X, Y)$ is a standard, Boolean relation, representing the fact hat "X has a subpart Y"; $C(X)$ is an $\mathbb{N}$-relation or an $\mathbb{R}_\perp$-relation (to be discussed shortly) representing the cost of X; and $T(X)$ is the *total cost* of X, that includes the cost of all its (sub-)subparts. The Datalog° program is:

$$T(X) \text{ :- } C(X) + \sum_Y \{T(Y) \mid E(X, Y)\}$$

When the graph defined by E is a tree then the program computes correctly the bill-of-material. We are interested, however, in what happens when the graph encoded by E has cycles, as illustrate with Fig. 4.1(b). The grounded program is:[4]

$$T(a) \text{ :- } C(a) + T(b) + T(c)$$
$$T(b) \text{ :- } C(b) + T(a) + T(c)$$
$$T(c) \text{ :- } C(c) + T(d)$$
$$T(d) \text{ :- } C(d)$$

We consider two choices for the POPS. First, the naturally ordered semiring $(\mathbb{N}, +, *, 0, 1)$. Here the program diverges, since the naïve algorithm will compute ever increasing values for $T(a)$ and $T(b)$, which are on a cycle. Second, consider the lifted reals $\mathbb{R}_\perp = (\mathbb{R} \cup \{\perp\}, +, *, 0, 1, \sqsubseteq)$. Now the

[4]Strictly speaking, we should have introduced POPS variables, $x_{T(a)}, x_{T(b)}, \ldots$, but, to reduce clutter, we show here directly the grounded atoms instead of their corresponding POPS variable.

program converges in 3 steps, as can be seen below:

	$T(a)$	$T(b)$	$T(c)$	$T(d)$
T_0	$\bot$	$\bot$	$\bot$	$\bot$
T_1	$\bot$	$\bot$	$\bot$	10
T_2	$\bot$	$\bot$	11	10
T_3	$\bot$	$\bot$	11	10

4.2.3 Extensions

We discuss here several extensions of Datalog° that we believe are needed in a practical implementation.

Case Statements Sum-products can be extended w.l.o.g. to include case statements of the form:

$$T(x_1, \ldots, x_k) \coloneq \texttt{case}\ \ C_1 : E_1;\quad C_2 : E_2; \cdots ; [\ \texttt{else}\ \ E_n]$$

where $C_1, C_2, \ldots$ are conditions and $E_1, E_2, \ldots$ are sum-product expressions. This can be desugared to a sum-sum-product:

$$T(x_1, \ldots, x_k) \coloneq \{E_1 \mid C_1\} \oplus \{E_2 \mid \neg C_1 \wedge C_2\} \oplus \cdots \oplus \{E_n \mid \neg C_1 \wedge \neg C_2 \cdots\}$$

and therefore the least fixpoint semantics and the convergence results in [ANP+22] continue to hold. For example, we may compute the prefix-sum of a vector V of length 100 as follows:

$$W(i) \coloneq \texttt{case}\ \ i = 0 : V(0);\quad i < 100 : W(i-1) + V(i);$$

Multiple Value Spaces Our discussion so far assumed that all rules in a Datalog° program are over a single POPS. In practice one often wants to perform computations over multiple POPS. In that case we need to have some predefined functions mapping between various POPS; if these are monotone, then the least fixpoint semantics still applies, otherwise the program needs to be *stratified*. We illustrate with an example, which uses two POPS: $\mathbb{R}_+$ and $\mathbb{B}$.

Example 4.7. We illustrate the *company control* example from [RS92, Example 3.2]. $S(X,Y) = n \in \mathbb{R}_+$ represents the fact that company X owns a proportion of n shares in company Y. We want to compute the predicate $C(X,Y)$, representing the fact that the company X *controls* company Y, where control is defined as follows: X controls Y if the sum of shares it owns in Y plus the sum of shares in Y owned by companies controlled by X is > 0.5. The program is adapted directly from [RS92]:

$$\begin{aligned} CV(X,Z,Y) &\text{ :- } [X=Z] * S(X,Y) + [C(X,Z)] * S(Z,Y) \\ T(X,Y) &\text{ :- } \sum_Z \{CV(X,Z,Y) \mid \text{Company}(Z)\} \\ C(X,Y) &\text{ :- } [T(X,Y) > 0.5] \end{aligned}$$

The value of $CV(X,Z,Y)$ is the fraction of shares that X owns in Y, through its control of company Z; when $X = Z$ then this fraction includes $S(X,Y)$. The value of $T(X,Y)$ is the total amount of shares that X owns in Y. The last rule checks whether this total is > 0.5: in that case, X controls Y.

The EDB and IDB vocabularies are $\sigma = \{S\}$, $\sigma_{\mathbb{B}} = \{\text{Company}\}$, $\tau = \{CV, T\}$, $\tau_{\mathbb{B}} = \{C\}$. The IDBs CV, T are $\mathbb{R}_+$-relations, C is a standard $\mathbb{B}$-relation. The mapping between the two POPS is achieved by the indicator function $[\Phi] \in \mathbb{R}_+$, which returns 0 when the predicate Φ is false, and 1 otherwise. All rules are monotone, w.r.t. to the natural orders on $\mathbb{R}_+$ and $\mathbb{B}$, and, thus, the least-fixpoint semantics continues to apply to this program. But the convergence results

in [ANP+22] apply only to fixpoints of polynomials, while the grounding of our program is no longer a polynomial.

Interpreted functions over the key-space A practical language needs to allow interpreted functions over the key space, i.e. the domain D, as illustrated by this simple example:

$$\texttt{Shipping}(cid, \texttt{date} + 1) \text{ :- } \texttt{Order}(cid, \texttt{date})$$

Here $\texttt{date} + 1$ is an interpreted function applied to $\texttt{date}$. Interpreted functions over D may cause the active domain to grow indefinitely, leading to divergence; the results in [ANP+22] apply only when the active domain is fixed, enabling a definition of the grounding of the Datalog° program.

Keys to Values Finally, a useful extension is to allow key values to be used as POPS values, when the types are right. For example, if $\texttt{Length}(X, Y, C)$ is Boolean relation, where a tuple (X, Y, C) represents the fact that there exists a path of length C from X to Y, then the we can compute the length of the shortest path as the following rule of the tropical semiring Trop^+:

$$\texttt{ShortestLength}(X, Y) \text{ :- } \min_C \left([\texttt{Length}(X, Y, C)]_\infty^0 + C \right)$$

The key variable C became an atom over the tropical semiring.

4.3 Semi-naïve Evaluation for Datalog°

We now introduce an extension of the semi-naïve evaluation algorithm to Datalog°. As discussed in Chapter 3, it is known that the naïve evaluation algorithm is inefficient, because at each iteration it repeats all the computations from the previous iterations. The more efficient semi-naïve algorithm, described in Section 3.3, keeps track of the changes between successive iterations, and applies

the rules only to the changes. However, the semi-naïve algorithm is defined only for programs that are monotone under set inclusion, and the systems that implement it enforce monotonicity, preventing the use of aggregation in recursion. We adapt the semi-naïve algorithm to Datalog°, under certain restrictions of the POPS $\boldsymbol{P}$, thus enabling the algorithm to be applied to programs with aggregation in recursion. Specifically, our semi-naïve algorithm applies to Datalog° programs over any *sparse* POPS, defined below. Surprisingly, it also works when the semiring is unordered, as we will discuss in Sec. 4.3.2.

Definition 4.5. A POPS $\boldsymbol{P}$ is *sparse* if $\bot = 0$.

Every naturally ordered POPS is sparse, but the converse is not true. For example, consider the POPS $\boldsymbol{N}^2 \stackrel{\text{def}}{=} (\mathbb{N}^2, +^2, \times^2, (0,0), (1,1))$, where $+^2$ and $\times^2$ mean to apply the operations component-wise, and the elements in $\boldsymbol{N}^2$ are ordered lexicographically. $\boldsymbol{N}^2$ is sparse but not naturally ordered, because $(1,2) \leq (2,1)$ but $\nexists x \in \boldsymbol{N}^2 : (1,2) + x = (2,1)$.

4.3.1 The Semi-naïve Algorithm for Sparse Datalog°

We can evaluate any Datalog° program over sparse POPS using the semi-naïve algorithm. Similar to the semi-naïve evaluation of classic Datalog, we keep track of a *delta* relation for each IDB relation. In the following we first define a simple, yet inefficient, rule to compute the delta relation which we then refine to be more efficient. Given a rule T_i :- $F_i(T_1, \ldots, T_n)$, the simple delta rule is defined as follows:

$$\delta_i^{(t)} \text{ :- } \sum_{\substack{T_1', \ldots, T_n', \\ T_i' \in \{T_i^{(t-2)}, \delta_i^{(t-1)}\}, \\ \exists i : T_i' = \delta_i^{(t-1)}}} F_i(T_1', \ldots, T_n')$$

In words, each term under sum evaluates F_i on arguments $T_1', \ldots, T_n'$, where each argument T_i' is either the IDB relation from two iterations ago, namely $T_i^{(t-2)}$, or the delta relation from the

Algorithm 6: Semi-naïve evaluation for Datalog° over sparse POPS

```
J^(0) ← 0;
J^(1) ← F(0), δJ^(1) ← F(0) ;
for t ← 2 to ∞ do
    δ^(t) ← δF(J^(t−2), δJ^(t−1)) ;
    J^(t) ← J^(t−1) ⊕ δ^(t);
    if δ^(t) = 0 then
        break
    end
end
return J^(t)
```

previous iteration, namely $\delta_i^{(t-1)}$. We evaluate F_i on all such sets of arguments, except for the set that does not contain any $\delta_i^{(t-1)}$. Note that there are exponentially many such argument sets, which we will improve to linearly many in our refined definition. The sum of these results then becomes the current delta relation $\delta_i^{(t)}$. We will denote this delta rule as δF_i, and use δF to denote the set of all delta rules. We can now define the semi-naïve algorithm using these delta rules, as shown in Algorithm 6. We initialize the IDB relations at time 0 to be empty, then apply F once to compute the IDB relations and the delta relations at time 1. In each iteration of the for-loop, we apply the delta rules to compute the new delta relations $\delta^{(t)}$. We then update the IDB relations by adding the delta relations to them. We repeat until the delta relations are all empty, which means the evaluation has converged. Note that the current algorithm lacks the difference operator from the classic semi-naïve algorithm for Datalog. We will introduce a generalized difference $\ominus$ as an additional optimization in Sec. 4.3.3.

To guarantee the semi-naïve algorithm is correct, the key property of the delta rule is that it computes the difference between two iterations of the naïve algorithm:

Proposition 4.1. Let $T_i^{(t)}$ and $T_i^{(t+1)}$ be an IDB relation T_i at successive iterations t and $t+1$ during naïve evaluation. Let $\delta_i^{(t+1)}$ be the delta relation for T_i at iteration $t+1$. Then $T_i^{(t)} \oplus \delta_i^{(t+1)} = T_i^{(t+1)}$.

Proof. We prove by induction on t. The proposition holds at $t = 0$. For $t \geq 1$, we assume w.l.o.g. that F is a single monomial $T_1 \otimes T_2 \otimes \cdots \otimes T_n$. Then we have:

$$\begin{aligned}
T_i^{(t+1)} &= T_1^{(t)} \otimes T_2^{(t)} \otimes \cdots \otimes T_n^{(t)} && \text{by definition of naïve evaluation} \\
&= (T_1^{(t-1)} \oplus \delta_1^{(t)}) \otimes \cdots \otimes (T_n^{(t-1)} \oplus \delta_n^{(t)}) && \text{by the inductive hypothesis} \\
&= T_i^{(t)} \oplus \delta_i^{(t+1)} && \text{by definition of } \delta_i^{(t+1)}
\end{aligned}$$

□

Although this simple version of the semi-naïve algorithm requires exponentially many delta rules, it can already improve the efficiency of naïve evaluation on certain programs.

Example 4.8. We now demonstrate the efficiency of the semi-naïve algorithm on the Bill-of-Material example over $\mathbb{N}$:

$$T(X) \text{ :- } C(X) + \sum_{Y | E(X,Y)} T(Y)$$

In words, we want to compute the total cost $T(X)$ of a component X, which is defined to be the individual cost $C(X)$ of X plus the sum of the total costs of all subcomponents Y of X. Suppose there are four components a, b, c, d, the subpart relation $E = \{(a, b), (b, c), (c, d)\}$, and each individual component has cost 1. The naïve evaluation of the program proceeds as in Fig. 4.2 (left). Note that the values for a, b and c are recomputed at each iteration, even after they stop changing. Under semi-naïve evaluation, we avoid recomputing the same values. This is shown on the right of Fig. 4.2. If a tuple's value does not change after an iteration, it is shown in gray. A tuple in the delta relation whose value is 0 is also shown in gray. Recall $0 = \bot$ in a sparse semiring, and therefore a tuple whose value is 0 is not stored. This means we indeed avoid (re-)computing the grayed-out entries.

	$T(a)$	$T(b)$	$T(c)$	$T(d)$
$T^{(0)}$	0	0	0	0
$T^{(1)}$	1	1	1	1
$T^{(2)}$	1	2	2	2
$T^{(3)}$	1	2	3	3
$T^{(4)}$	1	2	3	4

	$T(a)$	$T(b)$	$T(c)$	$T(d)$
$T^{(0)}, \delta^{(1)}$	0, 1	0, 1	0, 1	0, 1
$T^{(1)}, \delta^{(2)}$	1, 0	1, 1	1, 1	1, 1
$T^{(2)}, \delta^{(3)}$	1, 0	2, 0	2, 1	2, 1
$T^{(3)}, \delta^{(4)}$	1, 0	2, 0	3, 0	3, 1
$T^{(4)}, \delta^{(5)}$	1, 0	2, 0	3, 0	4, 0

Figure 4.2: Naïve (left) and semi-naïve (right) evaluation of the Bill-of-Material program.

The current definition of the delta rule contains exponentially many terms in the number of IDB predicates, which is prohibitive for long rules. An improved definition requires only linearly many terms. Let $T_1, \ldots, T_n$ be any permutation of the body IDB relations. We can then compute the delta relation as follows:

$$\begin{aligned}
\delta^{(t)} =& F(\delta_1^{(t-1)}, T_2^{(t-1)}, T_3^{(t-1)}, \ldots, T_n^{(t-1)}) \\
&+ F(T_1^{(t-2)}, \delta_2^{(t-1)}, T_3^{(t-1)}, \ldots, T_n^{(t-1)}) \\
&+ \quad \cdots \\
&+ F(T_1^{(t-2)}, \ldots, T_{n-2}^{(t-2)}, \delta_{n-1}^{(t-1)}, T_n^{(t-1)}) \\
&+ F(T_1^{(t-2)}, \ldots, T_{n-2}^{(t-2)}, T_{n-1}^{(t-2)}, \delta_n^{(t-1)})
\end{aligned}$$

That is, the i-th term has the δ relation at position i, $T^{(t-2)}$ relations before position i, and $T^{(t-1)}$ relations after position i. This alternative formulation can be seen as a "factorization" of the previous formulation by using the distributivity of $\times$ over $+$. It is easy to verify the two formulations are equivalent.

4.3.2 Datalog over Unordered Semirings

Although Datalog° is defined over POPS which requires an ordering of the pre-semiring values, many pracitical Datalog variants lack an ordering and simply rely on an *operational* semantics that iterates the rules until convergence (the semantics is undefined if the rules diverge). Suprisingly, the semi-naïve algorithm remains valid for unordered pre-semirings. This is because the proof of Proposition 4.1 does not rely on any ordering of the pre-semiring values. One important detail is necessary to ensure efficiency: just as we do not store $\bot$-values in relations over POPS, we also need to omit 0-values in relations over unordered pre-semirings. Doing so guarantees we only update entries that change their values after each iteration.

Example 4.9. Observe that Example 4.8 remains valid over the (unordered) pre-semiring over $\mathbb{R}$.

One may be tempted to think that the semi-naïve algorithm works for every POPS with any arbitrary order. However, it is not the case. For example, in the lifted semiring $\mathbb{R}_\bot$ we do not have $0 = \bot$. Since we must reserve the meaning of a missing tuple to indicate that tuple has value $\bot$, we cannot omit tuples with value 0. The semi-naïve algorithm for $\mathbb{R}_\bot$ still produces the correct result, but it does not save any work.

4.3.3 Further Optimizations

In some cases we can further improve the efficiency of the semi-naïve algorithm by removing redundant tuples from the delta relations. We achieve this with a generalized difference operation $\ominus$, which is defined as follows:

$$x \ominus y = \begin{cases} 0 & \text{if } x + y = y \\ x & \text{otherwise} \end{cases}$$

The intuition is simple: if adding a new value x does not change the result, we can omit x from the delta relation. With this, we can change the delta computation on line 4 of Algorithm 6 to use $\ominus$:

$$\delta^{(t)} \leftarrow \delta F(I^{(t-2)}, \delta I^{(t-1)}) \ominus I^{(t-1)}$$

For the POPS $\mathbb{N}$, the only value x such that $x \ominus y = 0$ is $x = 0$. One example where we can remove non-zero elements with $\ominus$ is the POPS $\mathbb{N}^{\leq 10} \stackrel{\text{def}}{=} (\{n \in \mathbb{N} \mid n \leq 10\}, +^{\leq 10}, \times^{\leq 10}, 0, 1)$, where $x +^{\leq 10} y \stackrel{\text{def}}{=} \min(10, x + y)$ and similar for $\times^{\leq 10}$. In other words, $\mathbb{N}^{\leq 10}$ is the POPS $\mathbb{N}$ "capped at" 10. Let $x \in \mathbb{N}^{\leq 10}$, $x = 1$, then $x \ominus 10 = 0$ because $x +^{\leq} 10 = 10$. Another example is the tropical semiring Trop^+ where $\forall x \geq y : x \ominus y = 0$.

Finally, when the POPS is idempotent (meaning $x \oplus x \oplus y = x \oplus y$), we only need to track the IDB relations at time $t - 1$ and can discard those at time $t - 2$. Specifically, we replace every $t - 2$ with $t - 1$ in each delta rule and in Algorithm 6. At this point, we recover exactly the semi-naïve algorithm for classic Datalog which is the same as Datalog$^\circ$ over the POPS $\mathbb{B}$.

4.4 Conclusions

A massive number of application domains demand us to move beyond the confine of the Boolean world: from program analysis [CC77, NNH99], graph algorithms [Car79, LT80, LRT79], provenance [GKT07], formal language theory [Kui97], to machine learning and linear algebra [Roc, ABC$^+$16]. Semiring and poset theory – of which POPS is an instance – is the natural bridge connecting the Boolean island to these applications.

The bridge helps enlarge the set of problems Datalog$^\circ$ can express in a very natural way. The possibilities are endless. For example, amending Datalog$^\circ$ with an interpreted function such as sigmoid will allow it to express typical neural network computations. Adding another semiring

to the query language (in the sense of FAQ [ANR16]) helps express rectilinear units in modern deep learning. At the same time, the bridge facilitates the porting of analytical ideas from datalog to analyze convergence properties of the application problems, and to carry over optimization techniques such as semi-naïve evaluation.

This chapter established part of the bridge. There are many interesting problems left open; we mention a few here.

The question of whether a Datalog° program over p-stable POPS converges in polynomial time in p and the size of the input database is open. This is open even for linear programs. Our result on Trop_p indicates that the linear case is likely in PTIME.

We have discussed in Sec. 4.2 several extensions to Datalog° that we believe are necessary in practice. It remains open whether our convergence results continue to hold for those extensions.

Negation can be added to Datalog° as an interpreted predicate. The question is, can we extend semantics results (such as stable model semantics) from general datalog / logic programming to Datalog° with negation?

Beyond exact solution and finite convergence, as mentioned in the introduction, it is natural in some domain applications to have approximate fixpoint solutions, which will allow us to trade off convergence time and solution quality. A theoretical framework along this line will go a long way towards making Datalog° deal with real machine learning, linear algebra, and optimization problems.

Chapter 5

The Free Join Algorithm

Over the last decade, worst-case optimal join (WCOJ) algorithms [NPRR12, Vel14, NRR13, Ngo18] have emerged as a breakthrough in one of the most fundamental challenges in query processing: computing joins efficiently. Such an algorithm can be asymptotically faster than traditional binary joins, all the while remaining simple to understand and implement [NRR13]. These algorithms opened up a flourishing field of research, leading to both theoretical results [NRR13, KNS17] and practical implementations [Vel14, ALT+17, FBS+20, MS19].

Over time, a common belief took hold: "WCOJ is designed for cyclic queries". This belief is rooted in the observation that, when the query satisfies a property called *cyclicity*, WCOJ enjoys lower asymptotic complexity than traditional algorithms [NRR13]. But when the query is acyclic, classic algorithms like the Yannakakis algorithm [Yan81] are already asymptotically optimal. Moreover, traditional binary join algorithms have benefited from decades of research and engineering. Techniques like column-oriented layout, vectorization, and query optimization have contributed compounding constant-factor speedups, making it challenging for WCOJ to be competitive in practice. This has lead many instantiations of WCOJ, including Umbra [FBS+20], Emptyheaded [ALT+17], and Graphflow [MS19], to adopt a hybrid approach: using WCOJ to

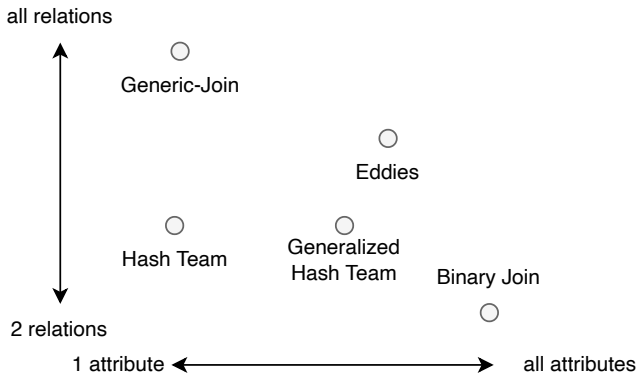


Figure 5.1: Design space of join algorithms.

process parts of the query, and resorting to traditional algorithms (usually binary join) for the rest. Having two different algorithms in the same system requires changing and potentially duplicating existing infrastructure like the query optimizer. This introduces complexity, and hinders the adoption of WCOJ.

The dichotomy of WCOJ versus binary join has led researchers and practitioners to view the algorithms as opposites. In this dissertation, we break down this dichotomy by proposing a new framework called Free Join that unifies WCOJ and binary join. We propose several new techniques to make Free Join outperform both binary join and WCOJ: we design an algorithm to convert any binary join plan to a Free Join plan that runs as fast or faster; we design a new data structure called COLT (for *Column-Oriented Lazy Trie*), adapting the classic column-oriented layout to improve the trie data structure used in WCOJ; and we propose a vectorized execution algorithm for Free Join.

To explain these contributions we provide some context. In this dissertation, we focus on algorithms based on hashing, and choose Generic Join [NRR13] as a representative of WCOJ algorithms. A crucial difference between Generic Join and binary join lies in the way they process each join operation. Binary join processes two relations at a time, and joins on *all attributes* shared

between these two relations. In contrast, Generic Join processes one attribute at a time, and joins *all relations* that share that attribute. This suggests a design space of join algorithms, where each join operation may process any number of attributes and relations. Figure 5.1 shows this design space which also covers classic multiway join algorithms like Hash Team [GBC98], Generalized Hash Team [KKW99] and Eddies [AH00]. Being able to join on any number of variables and relations frees us from the constraints of all existing algorithms mentioned above.

Our new framework, Free Join, covers the entire design space, thereby generalizing and unifying existing algorithms. The starting observation is that the execution of a *left-deep linear binary join plan* (reviewed in Sec. 5.1) is already very similar to Generic Join. While Generic Join (also reviewed in Sec. 5.1) is traditionally specified as a series of nested loops [NRR13], the push-based model [Neu11, KLK$^+$18] for executing a left-deep linear binary plan is also implemented, similarly, as nested loops. The two algorithms also process each join operation similarly: each binary hash join iterates over tuples on one relation, and for each tuple probes into the hash table of another relation; each loop level in Generic Join iterates over the keys of a certain trie, and probes into several other tries for each key. This inspired us to unify hash tables and hash tries into the same data structure, and develop Free Join using iteration and probing as key operations. This finer-grained view of join algorithms allows Free Join to generalize and unify existing algorithms, while precisely capturing each of them.

Free Join takes as input an already optimized binary join plan, and converts it into a new kind of plan that we call a Free Join plan. It then optimizes the Free Join plan, resulting in a plan that sits in between binary join and Generic Join, combining the benefits of both. On one hand Free Join fully exploits the design space in Figure 5.1. On the other hand, by starting from an already optimized binary plan, Free Join takes advantage of existing cost-based optimizers; in our system we used binary plans produced by the optimizer of DuckDB [RM20, Raa22].

Next, we address the main source of inefficiency in Generic Join: the need to construct a trie

on each relation in the query [MS19, FBS+20]. In contrast, a binary join plan needs to build a hash map only for each relation on the right-hand side of a join, and simply iterates over the relation on the left. As a result, a query optimizer commonly places the largest relation on the left to minimize the cost of hash building. One simple optimization in Free Join is that we do not built a trie for tables that are left children, mimicking the binary plans. However, we go a step further, and introduce the *Column-Oriented Lazy Trie* (COLT) data structure, which builds the inner subtries lazily, by creating each subtrie on demand. We note that this builds on an earlier idea in [FBS+20]. As the name suggests, COLT adapts the lazy trie data structure in [FBS+20] to use a column-oriented layout. And unlike the original lazy trie which builds at least one trie level per table, COLT completely eliminates the cost of trie building for left tables.

Finally, we describe a method for incorporating vectorized processing in Free Join, allowing it to collect multiple data values before entering the next iteration level. The standard Generic Join processes one data value at a time, but, as is the case in traditional query engines, this leads to poor cache locality. Vectorized execution [PMAJ01] was proposed for binary join to improve its locality by processing data values in batch. By breaking down join operations into iterations and probes, Free Join gives rise to a simple vectorized execution algorithm that breaks each iteration into chunks and groups together batches of probes. Our proposal is to our knowledge the first vectorized execution algorithm for Generic Join.

We implemented Free Join as a standalone Rust library, and compared it with two baselines: 1. our own Generic Join implementation in Rust, and 2. the binary hash join implemented in DuckDB [RM20, Raa22], a state-of-the-art in-memory database. We found that, on acyclic queries, Free Join is up to 19.36x faster than binary join, and up to 31.6x faster than Generic Join; on cyclic queries, Free Join is up to 15.45x faster than binary join, and up to 4.08x faster than Generic Join.

While optimizers for binary plans have been developed and improved over decades [SAC+79b], little is known about optimizing Generic Join. A Generic Join plan consists of a total order on its

variables, and its run time depends on the choice of this order. But since the theoretical analysis of Generic Join guarantees worst case optimality for *any* variable order, it is a folklore belief that Generic Join is more robust than binary join plans when the optimizer makes a bad choice. To test the robustness of binary join, Generic Join, and Free Join, We also conducted experiments measuring their performance when given a bad query plan. We found that Generic Join is indeed the least sensitive, while Free Join, like binary joins, suffers more from the poor optimization choices of the optimizer, since both rely on a cost-based optimized plan. However, Generic Join starts from worse baseline than Free Join. In other words, Free Join takes better advantage of a good plan, when available, than Generic Join does.

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

1. Free Join, a framework unifying existing join algorithms (Section 5.2).
2. An algorithm to converting a binary plan into an optimized Free Join plan (Section 5.3.1).
3. COLT, a column-oriented lazy trie data structure (Section 5.3.2).
4. A vectorized execution algorithm for Free Join (Section 5.3.3).
5. Experiments evaluating the algorithms and optimizations (Section 5.4).

5.1 Background

This section defines basic concepts and reviews background on binary join and Generic Join.

5.1.1 Basic Concepts

In previouse chapters we have used upper case variables for key values, and reserved lower case variables for semiring values. As we do not deal with semirings in this chapter, we will use lower

case variables for key values to reduce clutter. Recall from Section 3.2 that a *full conjunctive query* has the following form:

$$Q(\boldsymbol{x}) \text{ :- } R_1(\boldsymbol{x}_1), \ldots, R_m(\boldsymbol{x}_m). \tag{5.1}$$

where each variable in a body atom $R_i(\boldsymbol{x}_i)$ also appears in the head $R(\boldsymbol{x})$. We will assume that the query does not have *self-joins*, meaning no two atoms share the same relation name. This is without loss of generality: if two atoms have the same relation name, then we simply rename one of them. Our system also supports selections, projections, and aggregation. We assume that the selections are pushed down to the base tables, thus the atom R_i in (5.1) may include a selection over a base table; in particular, all variables in the atom $R_i(\boldsymbol{x}_i)$ are distinct. Similarly, projections and aggregates are performed after the full join, hence none of them is shown in (5.1).

Example 5.1. Consider the following SQL query:

```
SELECT r.x, s.u, t.u
  FROM R as r, M as s, M as t -- schema: R(x,y), M(u,v,w)
 WHERE s.w > 30 AND t.v = t.w
   AND r.y = s.u AND s.v = t.u AND t.v = r.x
```

Then we denote by $S = \Pi_{uv}(\sigma_{w>30}(M))$ and $T = \Pi_{uv}(\sigma_{v=w}(M))$, and write the query as:

$$Q_\triangle(x, y, z) \text{ :- } R(x, y), S(y, z), T(z, x).$$

We call this query the *triangle query* over the relations R, S, T.

It is often convenient to view the conjunctive query (5.1) as a hypergraph. The *query hypergraph* of Q consists of vertices $\mathcal{V}$ and edges $\mathcal{E}$, where the set of nodes $\mathcal{V}$ is the set of variables occurring in Q, and the set of hyperedges $\mathcal{E}$ is the set of atoms in Q. The hyperedge associated to the atom

$R(\boldsymbol{x}_i)$ is defined as the set consisting of the nodes associated to the variables $\boldsymbol{x}_i$. As standard, we say that the query Q is *acyclic* if its associated hypergraph is α-acyclic [Fag83]

5.1.2 Binary Join

The standard approach to computing a conjunctive query (5.1) is to compute one binary join at a time. A *binary plan* is a binary tree, where each internal node is a join operator $\bowtie$, and each leaf node is one of the base tables (atoms) $R_i(\boldsymbol{x}_i)$ in the query (5.1). The plan is a *left-deep linear plan*, or simply left-deep plan, if the right child of every join is a leaf node. If the plan is not left-deep, then we call it *bushy*. For example, $(R \bowtie S) \bowtie (T \bowtie U)$ is a bushy plan, while $((R \bowtie S) \bowtie T) \bowtie U$ is a left-deep plan. We do not treat specially right-deep or zig-zag plans, but simply consider them to be bushy.

In this dissertation we consider only hash-joins, which are the most common types of joins in database systems. The standard way to execute a bushy plan is to decompose it into a series of left-deep linear plans. Every join node that is a right child becomes the root of a new subplan, which is first evaluated, and its result materialized, before the parent join can proceed. As a consequence, every binary plan, bushy or not, becomes a collection of left-deep plans. In our system we decompose bushy plans exactly this way, and we will focus on left-deep linear plans in the rest of this chapter. For example, the bushy plan $(R \bowtie S) \bowtie (T \bowtie U)$ is converted into two plans: $P_1 = T \bowtie U$ and $P_2 = (R \bowtie S) \bowtie P_1$; both are left-deep plans.

To reduce clutter, we represent a left-deep plan $(\cdots((R_1 \bowtie R_2) \bowtie R_3) \cdots \bowtie R_{m-1}) \bowtie R_m$ as $[R_1, R_2, \ldots, R_m]$. Evaluation of a left-deep plan is done using pipelining. The engine iterates over each tuple in the left-most base table R_1; each tuple is probed in R_2; each of the matching tuple is further probed in R_3, etc.

Example 5.2. A possible left-deep linear plan for $Q_\triangle$ is $[R, S, T]$, which represents $(R(x, y) \bowtie$

```
for (x, y) in R:
  s = S[y]?
  for (y, z) in s:
    t = T[x,z]?
    for (x, z) in t:
      output(x, y, z)
```

(a) Binary join.

```
for a in R.x ∩ T.x:
  r = R[a]; t = T[a]
  for b in r.y ∩ S.y:
    s = S[b]
    for c in s.z ∩ t.z:
      output(a, b, c)
```

(b) Generic Join.

Figure 5.2: Execution of binary join and Generic Join for $Q_\triangle$. The notation `S[y]?` performs a lookup on S with the key y, and continues to the enclosing loop if the lookup fails. Binary join iterates over *tuples*, Generic Join iterates over *values*.

$S(y,z)) \bowtie T(z,x)$. To execute this plan, we first build a hash table for S keyed on y, where each Y maps to a vector of (y,z) tuples, and a hash table for T keyed on x and z, each mapped to a vector of (x,z) tuples[1]. Then the execution proceeds as shown in Figure 5.2a. For each tuple (x,y) in R, we first probe into the hash table for S using y to get a vector of (y,z) tuples. We then loop over each (y,z) and probe into the hash table for T using x and z. Each successful probe will return a vector of (x,z) tuples, and we output the tuple (x,y,z) for each (x,z).

5.1.3 Generic Join

Generic Join was introduced in [NRR13] and is the simplest worst-case optimal join algorithm. It is based on the earlier Leapfrog Triejoin algorithm [Vel14]. Generic Join computes the query Q in (5.1) through a series of nested loops, where each loop iterates over a variable (not a tuple). Concretely, Generic Join chooses arbitrarily a variable x, computes the intersection of all x-columns of all relations containing x, and for each value a in this intersection it computes the residual query $Q[a/x]$, where every relation R that contains x is replaced with $\sigma_{x=a}(R)$. In pseudocode:

GJ: **for** a **in** $\bigcap\{\Pi_x(R_i) \mid R_i$ contains $x\}$

[1]When the relations are bags, then the hash table may contain duplicate tuples, or store separately the multiplicity. We also note that the question what exactly to store in the hash table (e.g. copies of the tuples, or pointers to the tuple in the buffer pool) has been studied for a long time, see [Gra93].

```
compute Q[a/x]  \\ run GJ on Q with one fewer variable
```

If the query Q has k variables, then there are k nested loops in Generic Join. In the inner most loop, Generic Join outputs the tuple of constants, one from each iteration.[2] We notice that a plan for Generic Join consists of a total order of the variables of the query, which we denote as $[x_1, x_2, \ldots, x_k]$. Assuming that the intersection above is done optimally (see below), the algorithm is provably worst-case-optimal, for any choice of the variable order.

Example 5.3. Fig. 5.2b illustrates the pseudocode for Generic Join on the query $Q_\triangle$, using the variable order $[x, y, z]$. We denoted $\Pi_x(R)$ by $R.x$, and denoted (with some abuse) $\sigma_{x=a}(R)$ by $R[a]$.

While binary joins use hash tables, an implementation of Generic Join uses a *hash trie*, one for each relation in the query. The hash-trie is a tree, whose depth is equal to one plus the number of attributes of the relation, and where each node is either an empty leaf node,[3] or a hash map mapping each atomic value to another node. We will call the *level* of a node to be the distance from the root, i.e. the root has level 0, its children level 1, etc. The hash-trie completely represents the relation: every root-to-leaf path corresponds precisely to one tuple in the relation. Generic Join uses the hash-trie as follows. In order to compute $\sigma_{x=a}(R)$, it simply probes the current hash table for the value $x = a$, and returns the corresponding child. To compute an intersection $\Pi_x(R_1) \cap \Pi_x(R_2) \cap \cdots$, it selects the trie with the fewest keys, say R_1, then iterates over every value a in the keys for R_1 and probes it in each of the hash-maps for $R_2, R_3, \ldots$; this is a provably optimal algorithm for the intersection.

Example 5.4. Consider the query $Q_\triangle$ and the Generic Join plan $[x, y, z]$. We first build a hash trie each for R, S, and T. Each trie has three levels including the leaf. Level 0 of R is keyed on x, level 1 is keyed on y, level 2 contains empty leaf nodes, and similarly for S and T. Consider again the

[2]For bag semantics, it multiplies their multiplicities.
[3]For bag semantics, we store in the leaf the multiplicity of the tuple.

pseudocode in Figure 5.2b. The first loop intersects level 0 of the R-trie and the T-trie. For each value a in the intersection, we retrieve the corresponding children $R[a]$ and $T[a]$ respectively; these are at level 1. The second loop intersects the hash map $R[a]$ (at level 1) with the level 0 hash-map of S. For each value b in the intersection it retrieves the corresponding children (levels 2 and 1 respectively), and, finally, the innermost loop intersects the S- and T-hash maps (both at level 2), and outputs (a, b, c) for each c in the intersection. So far we have assumed set semantics; if the relations have bag semantics, then we simply multiply the tuple multiplicities on the leaves (level 3).

5.1.4 Binary Join v.s. Generic Join

Binary join and Generic Join each have their own advantages and disadvantages. Generic Join became popular because of its asymptotic performance guarantee: [NRR13] proved the algorithm is *worst-case optimal* for *any variable order*, in the sense that its run time is bounded by the largest possible size of its output, called AGM bound [AGM13]. For example, Generic Join executes $Q_\triangle$ in time $\sqrt{|R| \cdot |S| \cdot |T|}$, which is $n^{3/2}$ when all relations have size n; in contrast, a binary join plan can take $\Omega(n^2)$. We note, however, that this formula does not include the preprocessing time needed to construct the tries. For example, if T is significantly larger than R, S, then the run time of Generic Join is $\ll |T|$, yet during preprocessing Generic Join needs to read the entire relation T. On the other hand, binary join has been a staple of database systems for decades. The hash table data structure is simpler than hash tries and is cheaper to build. Techniques like vectorized execution and column-oriented layout have also made binary join practically efficient, but these optimizations have not been adapted for Generic Join. Binary join plans are known to be very sensitive to the choice of the optimizer: poor plans perform catastrophically bad [LGM+15]. In contrast, although the runtime performance of Generic Join does depend on the variable order,

some researchers believe that Generic Join is less sensitive to poor variable orders, in part because it is worst-case optimal.

5.2 Free Join

In this section we introduce the Free Join framework. We start by presenting the Generalized Hash Trie (GHT) which is the data structure used in Free Join (Section 5.2.1). Next we introduce the Free Join plan that specifies how to execute a query with Free Join (Section 5.2.2). Finally, we describe the Free Join algorithm, which takes as input a collection of GHTs and a Free Join plan, and computes the query according to the plan (Section 5.2.3).

We will show how each of the above components generalizes and unifies the corresponding components in binary join and Generic Join: the GHT generalizes hash tables and hash tries, the Free Join plan generalizes binary plans and Generic Join plans, and the Free Join algorithm generalizes binary join and Generic Join.

Throughout this section we will make use of the *clover query* $Q_\clubsuit$ in Figure 5.3a. Figure 5.3b visualizes the input relations for this query. Note that $Q_\clubsuit$ is *acyclic*.

5.2.1 The Generalized Hash Trie

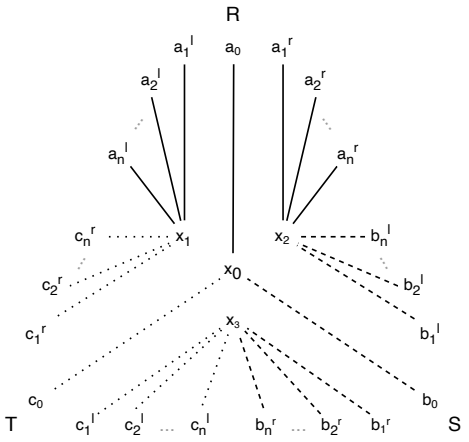
To unify binary join and Generic Join, we first need to unify the data structures they work over. We propose the Generalized Hash Trie which generalizes both the hash table used in binary join and the hash trie used in Generic Join.

Definition 5.1 (Generalized Hash Trie (GHT))**.** A GHT is a tree where each leaf is a vector of tuples, and each internal node is a hash map whose keys are tuples, and each key maps to a child node.

$$Q_\clubsuit(x, a, b, c) \text{ :- } R(x, a), S(x, b), T(x, c)$$

$$R = \{(x_0, a_0)\} \cup \{(x_1, a_i^l), (x_2, a_i^r) \mid i \in [1 \ldots n]\}$$
$$S = \{(x_0, b_0)\} \cup \{(x_2, b_i^l), (x_3, b_i^r) \mid i \in [1 \ldots n]\}$$
$$T = \{(x_0, c_0)\} \cup \{(x_3, c_i^l), (x_1, c_i^r) \mid i \in [1 \ldots n]\}$$

(a) $Q_\clubsuit$ and inputs.



(b) Visualization of the input relations.

Figure 5.3: (5.3a) the clover query $Q_\clubsuit$, and an input instance. (5.3b) visualization of the instance in Fig. 5.3a. The solid (top) edges form the relation R, the dashed (right) edges form the relation S, and the dotted (left) edges form the relation T. The relations join on the attribute in the center. The only output tuple consists of the three edges in the center.

```
interface GHT {
  # fields
  relation: String, vars: Vec<String>
  # constructor
  fn new(name: String, schema: Vec<Vec<String>>) -> Self
  # methods
  fn iter() -> Iterator<Tuple>
  fn get(key: Tuple) -> Option<GHT> }
```

Figure 5.4: The GHT interface.

We will reuse the terminology defined for tries, including *level*, *node*, and *leaf*, etc., for GHTs. We will also use the terms GHT and *trie* interchangeably when the context is clear. The *schema* of a GHT is the list $[\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_\ell]$ where $\boldsymbol{y}_k$ are the attribute names of the key at level k.

The hash trie used in Generic Join is a GHT where each key is a tuple of size one, and the last level stores empty vectors, each of which represents a leaf. The hash table used in binary join is very similar to a GHT with only two levels, where level 0 stores the keys and level 1 stores vectors of tuples. A small difference is that, in the GHT, the concatenation of a tuple from level 0 with a tuple from level 1 forms a tuple in the relation, whereas each whole tuple is stored directly in a hash table. We will show in Section 5.3.2 how the COLT data structure more faithfully captures the structure of a hash table. Figure 5.5 shows two examples of GHTs.

We use GHTs to represent relations, and attach metadata as well as access methods to each GHT, to be used by the Free Join algorithm. The GHT interface is shown in Figure 5.4. The `relation` field stores the relation name. A sub-trie inherits its name from its parent. The `vars` field stores parts of the relation's schema: if the trie is a vector of tuples, `vars` is the schema of each tuple; if the trie is a map, `vars` is the schema of each key. The constructor method `new` creates a new GHT from the named relation, where an n-th level trie has variables matching the n-th element of the `schema` argument, and the values along each path from the root to a leaf of the GHT form a tuple in the relation.

Example 5.5. Both GHTs in Figure 5.5 represent relation S from the clover query $Q_\clubsuit$ in Figure 5.3a. The GHT on the left (a hash trie) was created by calling the constructor method `new` with the schema `[[x],[b],[]]`, so the top-level trie has the schema `[x]`, each second-level trie has the schema `[b]`, and each third-level trie (a leaf) has the empty schema `[]`. The GHT on the right (a hash table) was created by calling `new` with the schema `[[x],[b]]`. It has only two levels, with schema `[x]` and `[b]`, respectively. Note that each b value in the hash trie is hashed and stored as a

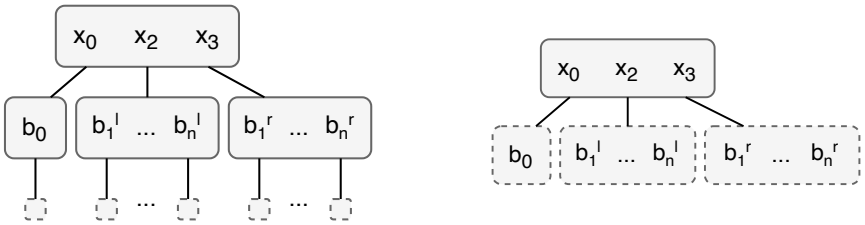


Figure 5.5: Two GHTs. The one on the left is also a hash trie, and the one on the right is similar to a hash table. Each box with solid border stores hash keys, and each box with dashed border is a vector of tuples. An empty box is an empty vector, representing a leaf.

key, while the b values in the hash table are simply stored in vectors.

The methods `iter` and `get` provide access to values stored in the trie. If the trie is a map, `get(key)` returns the sub-trie mapped to `key`, if any. Calling `get` on a vector returns `None`. If the trie is a vector, `iter()` returns an iterator over the tuples in the vector; calling `iter` on a map returns an iterator over the keys.

Example 5.6. On the second GHT in Figure 5.5, calling `iter` returns an iterator over the values $[x_0, x_2, x_3]$. Calling `get` with the key x_2 returns the sub-trie which is the vector $[b_1^l, \ldots, b_n^l]$. Calling `iter` on this sub-trie returns an iterator over $[b_1^l, \ldots, b_n^l]$.

5.2.2 The Free Join plan

A Free Join plan specifies how the Free Join algorithm should be executed. It generalizes and unifies binary join plans and Generic Join plans. Recall that a left-deep linear plan for binary join is a sequence of relations; it need not specify the join attributes, since all shared attributes are joined. In contrast, a Generic Join plan is a sequence of variables; it need not specify the relations, since all relations on each variable are joined. A Free Join plan may join on any number of variables and relations at each step, and therefore needs to specify both explicitly.

To help define the Free Join plan, we introduce two new concepts, called *subatom* and *parti-*

tioning. Fix the query Q in Eq. (5.1):

Definition 5.2. A *subatom* of an atom $R_i(\boldsymbol{x}_i)$ is an expression $R_i(\boldsymbol{y})$ where $\boldsymbol{y}$ is a subset of the variables $\boldsymbol{x}_i$. A *partitioning* of the atom $R_i(\boldsymbol{x}_i)$ is a set of subatoms $R_i(\boldsymbol{y}_1), R_i(\boldsymbol{y}_2), \ldots$ such that $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots$ are a partition of $\boldsymbol{x}_i$.

We now define the Free Join plan using these concepts.

Definition 5.3 (Free Join Plan)**.** Fix a conjunctive query Q. A Free Join *plan* is a list $[\phi_1, \ldots, \phi_m]$, where each ϕ_k is a list of subatoms of Q, called a *node*. The nodes are required to *partition the query*, in the sense that, for every atom $R_i(\boldsymbol{x}_i)$ in the query, the set of all its subatoms occurring in all nodes must form a partitioning of $R_i(\boldsymbol{x}_i)$. We denote by $vs(\phi_k)$ the set of variables in all subatoms of ϕ_k. The variables *available to* ϕ_k are all the variables of the preceding nodes:

$$avs(\phi_k) = \bigcup_{j<k} vs(\phi_j)$$

We will define shortly a *valid plan*, but first we show an example.

Example 5.7. The following is an Free Join plan for $Q_\clubsuit$:

$$[[R(x,a), S(x)], [S(b), T(x)], [T(c)]] \tag{5.2}$$

To execute the first node we iterate over each tuple (x, a) in R and use x to probe into S; for each successful probe we execute the second node: we iterate over each b in $S[x]$, then use x to probe into T; finally the third node iterates over c in $T[x]$. The reader may notice that this corresponds precisely to the left-deep plan $(R(x,a) \bowtie S(x,b)) \bowtie T(x,c)$. Another Free Join plan for $Q_\clubsuit$ is:

$$[[R(x), S(x), T(x)], [R(a)], [S(b)], [T(c)]] \tag{5.3}$$

This plan corresponds to the Generic Join plan $[x, a, b, c]$. Intuitively, here we start by intersecting $R.x \cap S.x \cap T.x$, then, for each x in the intersection, we retrieve the values of a, b, and c from R, S, and T, and output their Cartesian product.

Not all Free Join plans are valid, and only valid plans can be executed. We execute each Free Join node by iterating over one relation in that node, and probe into the others. Therefore, the values used in each probe must be available, either from the same node or a previous one.

Definition 5.4. A Free Join plan is *valid* if for every node ϕ_k the following two properties hold. (a) No two subatoms share the same relation, and (b) there is a subatom containing all variables in $vs(\phi_k) - avs(\phi_k)$. We call such an subatom a *cover* for ϕ_k, and write $cover(\phi_k)$ for the set of covers.

We will assume only valid plans in the rest of this chapter. To simplify the presentation, in this section we assume that each node Φ_k, has *one* subatom designated as cover, and will always list it as the first subatom in Φ_k. We will revisit this assumption in Sec. 5.3, and allow for multiple covers.

Example 5.8. Both plans in Example 5.7 are valid. The covers for the 3 nodes for Eq. (5.2) are $R(x, a)$, $S(b)$, and $T(c)$, respectively. For the plan in Eq. (5.3), the covers for the 4 nodes are $R(x), R(a), S(b), T(c)$; for the first node we could have also chosen $S(x)$ or $T(x)$ as cover.

Example 5.9. An example of an *invalid* plan for the clover query has one single node containing all relations and variables:

$$[[R(x, a), S(x, b), T(x, c)]]$$

Intuitively, we cannot execute it: if we iterate over, say R, then we bind two variables x and a, but to lookup S we need the key (x, b).

```
fn join(all_tries, plan, tuple):
  if plan == []:
    output(tuple)
  else:
    tries = [ t ∈ all_tries | t.relation ∈ plan[0] ]
    # iterate over the cover
    @outer for t in tries[0].iter():
      subtries = [ iter_r.get(t) ]
      tup = tuple + t
      # probe into other tries
      for trie in tries[1..]:
        key = tup[trie.vars]
        subtrie = trie.get(key)
        if subtrie == None: continue @outer
        subtries.push(subtrie)
      new_tries = all_tries[tries ↦ subtries]
      join(new_tries, plan[1:], tup)
```

Figure 5.6: The Free Join algorithm.

5.2.3 Execution of the Free Join Plan

The execution of a Free Join plan has two phases: the build phase and the join phase. The build phase constructs the GHTs for the relations in the query, by calling the constructor method `new` on each relation with the appropriate schema. The join phase works over the GHTs to compute the join of the relations.

Build Phase

The build phase constructs a GHT for each relation (atom) $R_i(\boldsymbol{x}_i)$, as follows. If the plan partitions the atom into the subatoms $R_i(\boldsymbol{y}_0), R_i(\boldsymbol{y}_1), \ldots, R_i(\boldsymbol{y}_{\ell-1})$, then the schema of its GHT is the list $[\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_{\ell-1}, []]$. Recall that the last level of a GHT is a vector instead of a hash map. As an optimization, if the last subatom $R_i(\boldsymbol{y}_{\ell-1})$ is the cover of its node, then we drop the last [] from the schema, in other words, we construct a vector for the $\boldsymbol{y}_{\ell-1}$. After computing the schema for

each relation, we call the constructor method `new` on each relation and its computed schema to build the GHTs.

Example 5.10. Consider the plan in Eq. (5.2) for the clover query $Q_\clubsuit$. The GHT schemas for R, S, and T are $[[x, a]]$, $[[x], [b]]$, and $[[x], [c]]$ respectively. Thus, R is a flat vector of tuples, and each of S and T is a hash map of vectors of values. Consider now the triangle query $Q_\triangle$ and the plan $[[R(x, y), S(y), T(x)], [S(z), T(z)]]$. The GHT schemas for R, S, T are $[[x, y]]$, $[[y], [z]]$, and $[[x], [z], []]$: in other words R is stored as a vector, S is a hash-map of vectors, and T is a hash-map of hash-maps of vectors.

Join Phase

The pseudo-code for the Free Join algorithm is shown in Figure 5.6. The `join` method takes as input the GHTs, the Free Join plan, and the current tuple initialized to be empty. If the plan is empty, we output the tuple (line 3). Otherwise, we work on the first node in the plan and intersect relevant tries (line 5). We iterate over tuples in the covering relation, which is the first trie in the node (line 7). Then, we use values from `t` and the `tuple` argument as keys to probe into the other tries (line 8-15). To construct a key for a certain trie, we find the values mapped from the trie's schema variables in `t` and `tuple` (line 12). If any probe fails, we continue to the next tuple in the outer loop. If all probes succeed, we replace the original tries with the subtries returned by the probes, and recursively call `join` on the new tries and the rest of the plan (line 16-17).

The recursive definition may obscure the essence of the Free Join algorithm, so we provide some examples where we unroll the recursion. We introduce some convenient syntax to simplify the presentation. We write `for (x,y,...) in T:` to introduce a for-loop iterating over `T`, binding the values of each tuple in `T.iter()` to the variables `x,y,...`. We write `r = R[t]?` to bind the result of `R.get(t)` to `r`; if the lookup fails, we continue to the next iteration of the enclosing loop.

```
R = GHT("R",[["x","a"]])
S = GHT("S",[["x"],["b"]])
T = GHT("T",[["x"],["c"]])
for (x, a) in R:
  s = S[x]?
  for b in s:
    t = T[x]?
    for c in t:
      output(x, a, b, c)
```

(a) Binary Free Join.

```
# same as the left
# ...
# ...
for (x, a) in R:
  s = S[x]?
  t = T[x]?
  for b in s:
    for c in t:
      output(x, a, b, c)
```

(b) Factorized Free Join.

```
R = GHT("R",[["x"],["a"]])
# same as the left
# ...
for x in R:
  r=R[x]?; s=S[x]?; t=T[x]?
  for a in r:
    for b in s:
      for c in t:
        output(x, a, b, c)
```

(c) Generic Free Join.

Figure 5.7: Execution of Free Join for the clover query.

In other words, `r = R[t]?` is equivalent to:

```
r = R.get(t); if r.is_none(): continue
```

Example 5.11. Consider the plan in Eq. (5.2) for the clover query $Q_\clubsuit$. Figure 5.7a shows its execution; ignore the underlined instruction for now. In the build phase, we construct a flat vector for R and a hash table for each of S and T. In the join phase, for the node $[R(x,a), S(x)]$ we iterate over R and probe into S, while for the second node $[S(b), T(x)]$, we iterate over the second level of S and probe into T. Finally, the third loop iterates over the second level of T and outputs the result.

Example 5.12. Consider now the plan in Eq. (5.3) for $Q_\clubsuit$. Its execution is shown in Figure 5.7c. We construct hash tables for R, S, and T, keyed on x. The first loop level intersects the three relations on x, and subsequent loop levels take the Cartesian product of the relations on a, b, and c.

Note that Fig. 5.7a follows the execution of binary hash join with the plan $[R, S, T]$, whereas Fig. 5.7c follows the execution of Generic Join with the plan $[x, a, b, c]$. We will describe Fig. 5.7b later.

5.2.4 Discussion

Free Join plans generalize both traditional binary plans and Generic Join. One limitation so far is our assumption that the cover is chosen at optimization time. This was convenient for us to illustrate how to avoid constructing some hash maps, by storing the last level of a GHT as vector, when it corresponds to a cover. In contrast, Generic Join computes the intersection $R_1.x \cap R_2.x \cap \cdots$ by iterating over the smallest set, hence it chooses the "cover" at run time. We will address this in the next section by describing COLT, a data structure that constructs the GHT lazily, at run time, allowing us to choose the cover during the *join phase*.

5.3 Optimizing the Free Join Plan

In the previous section we have introduced Free Join plans and their associated data structures, the GHTs. We have seen that a Free Join plan is capable of covering the entire design space in Fig. 5.1, from traditional join plans to Generic Join. In this section we describe how to build, optimize, and speedup the execution of a Free Join plan. We start from a conventional binary plan produced by a query optimizer, and convert it into an optimized Free Join plan (Section 5.3.1). Next, we introduce the COLT data structure to greatly reduce the cost of building the hash tries (Section 5.3.2). We present a simple vectorized execution algorithm for Free Join (Section 5.3.3), and finally, we discuss how Free Join relates to Generic Join (Section 5.3.4).

5.3.1 Building and Optimizing a Free Join Plan

Our system starts from an optimized binary plan produced by a traditional cost-based optimizer; in particular, we use DuckDB's optimizer [RM20, Raa22]. We decompose a bushy plan into a set of left-deep plans, as described in Sec. 5.1, then convert each left-deep plan into an equivalent

```
fn binary2fj(bin_plan):
  fj_plan = []; r = bin_plan[0]
  φ₀ = [ r(r.schema) ]; φ = φ₀ # iterate over left relation
  for s in bin_plan[1:]:
    φ.push(s(s.schema ∩ avs(φ))) # probe w/ available vars
    fj_plan.push(φ)
    φ = [ s(s.schema - avs(φ)) ] # iterate over probe result
  fj_plan.push(φ)
  return fj_plan
```

Figure 5.8: Translating a binary plans to a Free Join plan.

Free Join plan. Finally, we optimize the converted Free Join plan, resulting in a plan that can be anywhere between a left-deep plan or a Generic Join plan.

The conversion from a binary plan to an equivalent Free Join plan is done by the function `binary2fj` in Figure 5.8. We begin by adding the full atom of the left relation as the first subatom in the first Free Join plan node. Then we iterate over the remaining relations in the binary join plan. For each relation, we add a subatom whose variables are the intersection of the relation's schema with the available variables at the current Free Join plan node. Then we create a new join node, adding to it the relation with the remaining variables.

Example 5.13. The binary plan $[R, S, T]$ for the clover query $Q_\clubsuit$ is converted into the Free Join plan shown in Eq. (5.2). For another example, consider a chain query:

$$Q \text{ :- } R(x, y), S(y, z), T(z, u), W(u, v).$$

The left-deep plan $[R, S, T, W]$ is converted into:

$$[[R(x, y), S(y)], [S(z), T(z)], [T(u), W(u)], [W(v)]]$$

So far the algorithm in Figure 5.8 produces a Free Join plan that is equivalent to the left-deep

```
fn factor(plan):
  @outer: for i in [1..n-1].reverse():
    φ = plan[i]; φ' = plan[i-1]
    for α in φ:
      if α.vars ⊆ avs(φ) ∧ α.relation ∉ φ' :
        φ.remove(α); φ'.push(α)
      else: continue @outer
```

Figure 5.9: Factorizing a Free Join plan.

plan. Next, we optimize the Free Join plan. The main idea behind our optimization is to bring the query plan closer to Generic Join, without sacrificing the benefits of binary join.

For intuition, let us revisit the clover query $Q_\clubsuit$, and its execution depicted in Fig. 5.7a (as explained in Example 5.11). Consider the input shown in Fig. 5.3b. Both relations R and S are skewed on the value x_2, and their join will produce n^2 tuples, namely $\{(x_2, a_i, b_j) \mid i, j \in [1..n]\}$. This means the body of the second loop in Figure 5.7a is executed n^2 times. However, the n^2 tuples are only to be discarded by the join with T which does not contain x_2.

There is a simple fix to the inefficiency: we can pull the underlined lookup on T in Figure 5.7a out of the loop over s to filter out redundant tuples early. This results in the nested loops in Figure 5.7b which runs in $O(n)$ time, because the two lookups in the first loop already filter the result to a single tuple. At the logical level, we convert the first Free Join plan into the second Free Join plan:

Naive plan (Eq. (5.2)):	$[[R(x,a), S(x)], [S(b), T(x)], [T(c)]]$
Optimized plan:	$[[R(x,a), S(x), T(x)], [S(b)], [T(c)]]$

While this is closer to the Generic Join in Figure 5.7c, it differs in that it still uses the same GHTs built for original plan, without the need for an additional hash table for R.

More generally, we will optimize a Free Join plan by *factoring out* lookups, i.e. by moving a subatom from a node Φ_i to the node Φ_{i-1}. In doing so we must ensure that the plan is still valid, and also avoid accidental slowdowns. For example, we cannot factor the lookups on S and T beyond the outermost loop, because that loop binds the variable x used in the lookups.

The optimization algorithm for Free Join plans is shown in Figure 5.9. We traverse the plan in reverse order visiting each node. For each node, if there is an atom whose variables are all available before that node, and if the previous node does not contain an atom of the same relation, we move the atom to the previous node. These two checks ensure the factored plan remains valid. The last line in the algorithm ensures we factor lookups *conservatively*. That is, we factor out a lookup only if all previous lookups in the same node have also been factored out. Doing so respects the lookup ordering given by the original cost-based optimizer, since scrambling this ordering may inadvertently slow down the query. It should be clear that factoring out any lookup will always improve performance.

5.3.2 COLT: Column-Oriented Lazy Trie

The original Generic Join algorithm builds a hash trie for each input relation. A left-deep plan avoids building a hash table on the left most relation, since it only needs to iterate over it, and this is an important optimization, since the left most relation is often the largest one. Building a subtrie can also be wasteful when that subtrie's parent is pruned away by an earlier join, in which case the subtrie will never be used. To address that, we describe here how to build the tries *lazily*: we only build the trie for a (sub-)relation at runtime, if and when we need to perform a lookup, or need to iterate over a prefix of its tuples. This idea leads to our new data structure called Column-Oriented Lazy Trie, or COLT for short. In our system the raw data is stored column-wise, in main memory, and each column is stored as a vector, as standard in column-oriented databases [ABH+13].

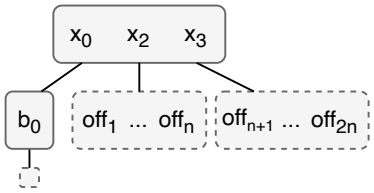


Figure 5.10: A COLT for the relation S in Fig. 5.3a. Each off_i is an integer representing an offset into the table S.

Definition 5.5. A *COLT* is a tree where each leaf is a vector of offsets into the base relation, and each internal node is a hash map mapping a tuple to a child node.

A COLT tree need not be balanced, and there can be both hash maps and vectors at the same tree level. Fig. 5.10 illustrates a COLT tree for the instance S of the clover query $Q_\clubsuit$.

COLT Implements the GHT interface in Figure 5.4, and its implementation is shown in Figure 5.11. As before, COLT stores a reference to the relation it represents, as well as the GHT schema computed from the plan. Consider a relation with n tuples. The COLT tree is initialized with a single node consisting of the vector $[0, \ldots, n-1]$, i.e. one offset to each tuple. COLT implements the `get` and `iter` methods lazily. When `get` is called, we check if the current node is a hash map or a vector. In the first case, we simply perform a lookup in the map. In the second case, we first replace the current vector with a hash map, whose children are vectors of offsets. Notice that this requires iterating over the current vector of offsets, accessing the tuple in the base table, inserting the key in the hash map, and inserting the offset in the corresponding child. Consider now a call to `iter`. If the current node is a hash map, then we return an iterator over it. If it is a vector, then we check if it is a suffix of the relation schema: if yes, then we simply iterate over that vector (and access the tuples via their offsets), otherwise we first materialize the current hash map as explained above, and return an iterator over the hash map.

As a simple but effective optimization, we do not initialize the COLT tree to the single node $[0, 1, \ldots, n-1]$, but instead iterate directly over the base table, if required. If no `get` is performed

```
struct COLT {
  relation, schema, vars,
  data = Map(HashMap<Tuple, COLT>) | Vec<Vec<u64>> }

impl GHT for COLT:
  fn new(relation, schema):
    COLT { relation, schema, schema[0],
           data = [ 0, 1, ..., relation.len - 1 ] }

  fn iter():
    match self.data:
      Map(m) => m.keys().iter(),
      Vec(v) =>
          if is_suffix(self.vars, relation.schema):
            v.map(|i| cols = self.relation[self.vars];
                      cols.map(|c| c[i]) )
          else: self.force(); self.iter()

  fn get(key): self.force(); self.get_map.get(key)

  fn force():
    match self.data:
      Map(m) => {} # already forced, do nothing
      Vec(v) =>
        map = new()
        for i in v:
          cols = self.relation[self.vars]
          k = cols.map(|col| col[i])
          if map[k] is None: # make a new, empty COLT
            map[k] = COLT { relation: self.relation,
                            schema: self.schema[1..],
                            data: [] }
          map[k].data.push(i)
        self.data = Map(map)
```

Figure 5.11: The COLT data structure.

on this table, then we have completely eliminated the cost of building any auxiliary structure on this table. Thus, the Free Join plan can be equivalent to a left-deep plan that avoids building a hash table on the left-most relation. COLT is also closer to the structure of traditional hash tables, which, in some implementations, map a key to a vector of pointers to tuples.

Example 5.14. Consider an extension of the clover query $Q_\clubsuit$:

$$Q(x, a, b, c) \text{ :- } R(x, a), S(x, b), T(x, c), \underline{U(b)}.$$

Generic Join builds a 2-level hash trie for each of R, S, and T, as well as a 1-level hash trie for U. Consider the Free Join plan $[[R(x, a), S(x), T(x)], [U(b), S(b)], [T(c)]]$. Free Join executes the first node of the plan by iterating over R directly, without constructing any auxiliary structure. For each tuple (x, a) in R, it looks up x in S and T. Upon the first lookup, COLT builds the first level of the GHT for S and T, i.e. a hash map indexed by the x values. Assuming the database instance for R, S, T shown in Fig. 5.3a, the result of $R.x \cap S.x \cap T.x$ has only one value, x_0, thus, Free Join executes the second node for only one value x_0. Here it needs to intersect $U(b)$ and $S(b)$. Assume for the moment that Free Join chooses $U(b)$ to be the cover, on the first lookup in S, COLT will expand the second level, arriving at Figure 5.10: notice that all other b values in S will never be inserted in the hash table. More realistically, Free Join follows the principle in Generic Join and chooses $S(b)$ as cover, because it is the smallest: it builds a hash map for U, but none for the 2nd level of S.

The example highlights a divergence between Generic Join and traditional plans. To intersect $R_1.x \cap R_2.x \cap \ldots$, Generic Join choose to iterate over the smallest relation, which results in the best runtime *ignoring* the build time. A traditional join plan will iterate over the largest relation, because then it needs to build hash tables only on the smaller relations. Currently, we follow Generic Join, and plan to explore alternatives in the future.

```
@outer for ts in tries[0].iter_batch(batch_size):
  tup_subtries = [(tuple + t, [ tries[0].get(t) ]) | t ∈ ts]
  for trie in tries[1..]:
    for (tup, subtries) in tup_subtries:
      subtrie = trie.lookup(tup[trie.vars])
      if subtrie is None:
        tup_subtries.remove((tup, subtries))
      else: subtries.append(subtrie)
  for (tup, subtries) in tup_subtries:
    new_tries = all_tries[tries ↦ subtries]
    join(new_tries, plan[1:], tup)
```

Figure 5.12: Vectorized execution for Free Join.

5.3.3 Vectorized Execution

The Free Join algorithm as presented in Figure 5.6 suffers from poor temporal locality. In the body of the outer loop, we probe into the same set of relations for each tuple. However, these probes are interrupted by the recursive call at the end, which is itself a loop interrupted by further recursive calls.

A simple way to improve locality is to perform a batch of probes before recursing, just like the classic vectorized execution for binary join. Concretely, we replace the `iter` method with a new method `iter_batch(batch_size)` which returns up to `batch_size` tuples at a time. If there are less than `batch_size` tuples left, it returns all the remaining tuples. Then we replace the outer loop in Figure 5.6 with the one in Figure 5.12. For each batch of tuples, we create a vector pairing each tuple to its subtrie in `tries[0]`. Then for each trie to be probed, we iterate over the vector and look up each tuple from the trie. If the lookup succeeds, we append the subtrie to the vector of tries paired with the tuple. If it fails, we remove the tuple to avoid probing it again. Finally, with each tuple and the subtries it pairs with, we recursively call `join`.

5.3.4 Discussion

COLT is a lazy data structure, sharing a similar goal with database cracking [IKM07a, IKM07b], where an index is constructed incrementally, by performing a little work during each lookup. Another connection is to Factorized Databases [OS16] – we intentionally used the term "factor" when describing how we optimize Free Join plans to suggest this connection. Concretely, we can view the trie data structure as a factorized representation of a relation, where keys of the same hash map are combined with union, and tuples are formed by taking the product of values at different levels. Practically, we can use this factorized representation to compress large outputs, saving time and space during materialization.

As we discussed at the end of Section 5.2, in order match the optimality of Generic Join, the Free Join algorithm needs to choose dynamically the "cover", i.e. the relation over which to iterate. To achieve this, we first find *all* covers for each node, then make a simple change to the Free Join algorithm in Figure 5.6: we simply choose to iterate over the cover whose trie has the fewest keys. For that we insert the following code right before the outer loop in Figure 5.6:

```
trie[0] = covers(plan[0]).min_by(|t| t.keys().len)
trie[1..] = # the rest of the tries
```

When we use COLTs, we cannot know the exact number of keys in a vector unless we force it into a hash map. In that case we use the length of the vector as an estimate.

Example 5.15. Consider the triangle query $Q_\triangle$, and the following Free Join plan:

$$[[R(x), T(x)], [R(y), S(y)], [S(z), T(z)]]$$

Each subatom is a cover of its own node. On the outermost loop, we iterate over R if it has fewer x values, and otherwise we iterate over T. On the second loop level we make a decision picking

between S and a subtrie of R, *for each subtrie of R*. Finally, on the innermost loop we pick between the subtries of S and T.

5.4 Experiments

We implemented Free Join as a standalone Rust library. The main entry point of the library is a function that takes a binary join plan (produced and optimized by DuckDB), and a set of input relations. The system converts the binary plan to a Free Join plan, optimizes it, then runs it using COLT and vectorized execution. We compare Free Join against two baselines: our own Generic Join implementation in Rust, and the binary hash join implemented in the state-of-art in-memory database DuckDB [RM20, Raa22]. We evaluate their performance on the popular Join Order Benchmark (JOB) [LGM+15] and the LSQB benchmark [MLK+21]. In addition, we compare against Kùzu [FJC+23], a system that implements Generic Join. Kùzu is the current iteration of the Graphflow system [MS19]. We ask three research questions:

1. How does Free Join compare to binary and Generic Join, on acyclic and cyclic queries?
2. What is the impact of COLT and vectorization on Free Join?
3. How sensitive is Free Join to the query optimizer's quality?

5.4.1 Setup

While we had easy access to optimized join plans produced by DuckDB, we did not find any system that produces optimized Generic Join plans, or can take an optimized plan as input. We therefore implement a Generic Join baseline ourselves, by modifying Free Join to fully construct all tries, and removing vectorization. We chose as variable order for Generic Join the same as for

Free Join.[4]

Both the JOB and the LSQB benchmarks focus on joins. JOB contains 113 acyclic queries with an average of 8 joins per query, whereas LSQB contains both cyclic and acyclic queries. Each query in the benchmarks only contains base-table filters, natural joins, and a simple group-by at the end, and no null values. JOB works over real-world data from the IMDB dataset, and LSQB uses synthetic data. We exclude 5 queries from JOB that return empty results, since such empty queries are known to introduce reproducibility issues[5]. We use the first 5 queries from LSQB; the other 4 queries require anti-joins or outer joins which we do not support.

We ran all our experiments on a MacBook Air laptop with Apple M1 chip and 16GB memory. All systems are configured to run single-threaded in main memory, and we leave all of DuckDB's configurations to be the default. All systems are given the same binary plan optimzed by DuckDB. To answer our third research question, we needed to hijack DuckDB's optimizer to produce a poor plan. We did this by modifying its cardinality estimator to always return 1. Since we are only interested in the performance of the join algorithm, we exclude the time spent in selection and aggregation when reporting performance. This excluded time takes up on average less than 1% of the total execution time.

5.4.2 Run time comparison

Our first set of experiments compare the performance of Free Join, Generic Join, and binary join on the JOB and LSQB benchmarks. For each query in the benchmarks, we invoke DuckDB to obtain an optimized binary plan, and provide the plan to our Free Join and Generic Join implementation. We run LSQB with the scaling factors 0.1, 0.3, 1, and 3, as some queries run out of memory with larger scaling factors.

[4] Free Join defines only a partial order; we extended it to a total order.

[5] See GitHub issue: `https://github.com/gregrahn/join-order-benchmark/issues/11`

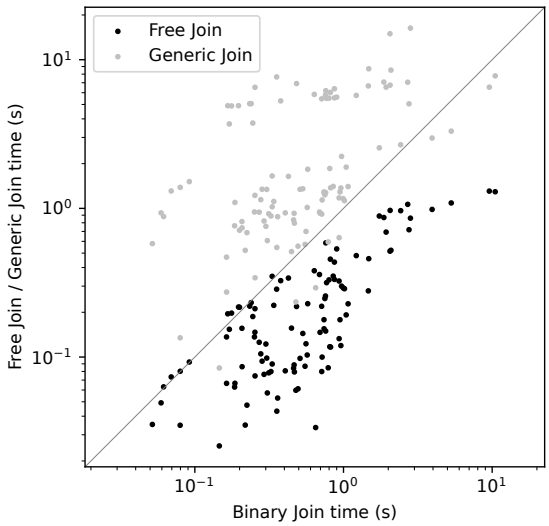


Figure 5.13: Run time comparison on JOB. Each black dot compares the run time of a query on Free Join and binary join, and a black dot below the diagonal means Free Join is faster. The gray dots compare Generic Join and binary join similarly.

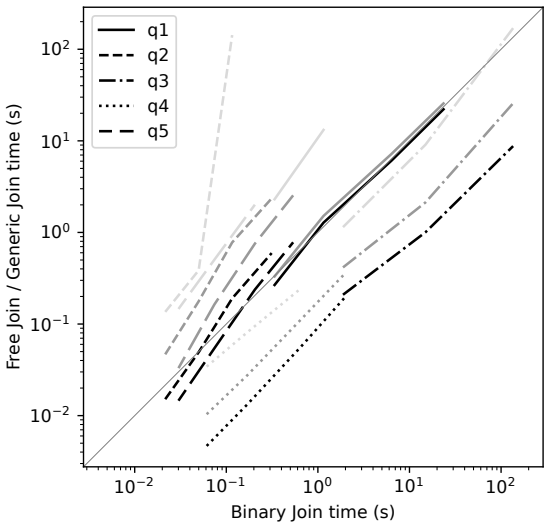


Figure 5.14: Run time comparison on LSQB. Each line is a query running on increasing scaling factors (0.1, 0.3, 1, 3). The black lines compare Free Join with binary join, the gray lines compare our Generic Join baseline with binary join, and the light gray lines compare Kùzu with binary join.

Figure 5.13 compares the run time of Free Join and Generic Join against binary join on JOB queries. We see that almost all data points for Free Join are below the diagonal, indicating that Free Join is faster than binary join. On the other hand, the data points for Generic Join are largely above the diagonal, indicating that Generic Join is slower than both binary join and Free Join. On average (geometric mean), Free Join is 2.94x faster than binary join and 9.61x faster than Generic Join. The maximum speedups of Free Join against binary join and Generic Join are 19.36x and 31.6x, respectively, while the minimum speedups are 0.85x (17% slowdown) and 2.63x.

We zoom in onto a few interesting queries for a deeper look. The slowest query under DuckDB is Q13a, taking over 10 seconds to finish. Generic Join runs slightly faster, taking 7 seconds, whereas Free Join takes just over 1 second. The query plan for this query reveals the bottleneck for binary join: the first 3 binary joins are over 4 very large tables, and two of the joins are many-to-many joins, exploding the intermediate result to contain over 100 million tuples. However, all 3 joins are on the same attribute; in other words they are quite similar to our clover query $Q_\clubsuit$. As a result, Generic Join and Free Join simply intersects the relations on that join attribute, expanding the remaining attributes only after other more selective joins. This data point appears to confirm a folklore that claims WCOJ algorithms are more resistant to poor query plans. After all, binary join could have been faster, had the query plan ordered the more selective joins first. We expand on this point with more experiments evaluating each algorithm's robustness against poor plans in Section 5.4.4.

On a few queries Free Join runs slightly slower than binary join, as shown by the data points over the diagonal. The binary plans for these queries are all bushy, and each query materializes a large intermediate relation. We have not spent much effort optimizing for materialization, and we implement a simple strategy: for each intermediate that we need to materialize, we store the tuples containing all base-table attributes in a simple vector. Future work may explore more efficient materialization strategies, for example only materializing attributes that are needed by future

joins.

Figure 5.14 compares the performance of Free Join and Generic Join against binary join on LSQB queries. Each line corresponds to one query running on scaling factors 0.1, 0.3, 1, and 3. The black lines are for Free Join, gray lines for our own Generic Join baseline, and light gray lines for Kùzu. Kùzu errors when loading data for SF3; it did not finish after 10 minutes for q1 SF 1. DuckDB also took over 10 minutes running q3 SF 3. These instances do not show up in the figure. We can see Kùzu takes consistently longer than our Generic Join implementation on all queries across scaling factors. This shows that our Generic Join implementation is a reasonable baseline to compare against. On cyclic queries, Free Join is up to 15.45x (q3) faster than binary join, and up to 4.08x (q2) faster than Generic Join. On acyclic queries Free Join is up to 13.07x (q4) and 3.25x (q5) faster than binary join and Generic Join, respectively. On q3 and q4 both Free Join and Generic Join consistently outperform binary join on all scaling factors. q3 contains many cycles, whereas q4 is a star query, so the superior performance of Free Join and Generic Join is expected. Surprisingly, despite q2 being a cyclic query, Free Join is only slightly faster on smaller inputs and is even slightly slower on larger inputs. This is the opposite of the common believe that WCOJ algorithms should be faster on cyclic queries. The query plan reveals that there are no skewed joins, and so binary join suffers no penalty. Our experience shows that, in practice, the superiority of WCOJ algorithms like Free Join and Generic Join is not solely determined by the cyclicity of the query; the presence of skew in the data is another important factor.

Unlike the JOB queries, in LSQB the output size (before aggregation) is much larger than the input size. This leads to a large amount of time spent in constructing the output, which involves random accesses to retrieve the data values for each tuple. We therefore implemented the optimization in Section 5.3.4 to factorize the output. This made q1 significant faster, as shown in Figure 5.15, while other queries are not affected.

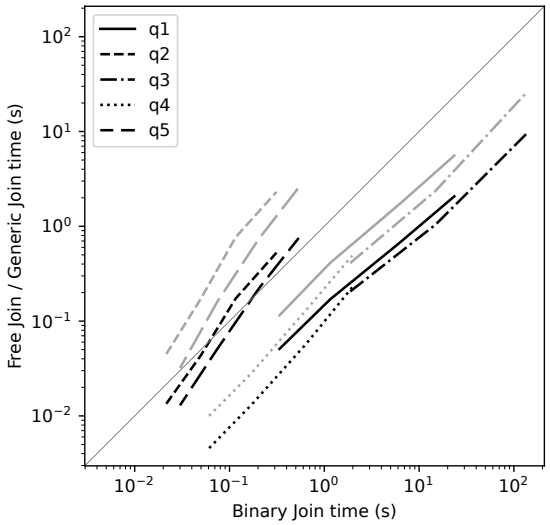


Figure 5.15: Run time comparison on LSQB w/ factorization.

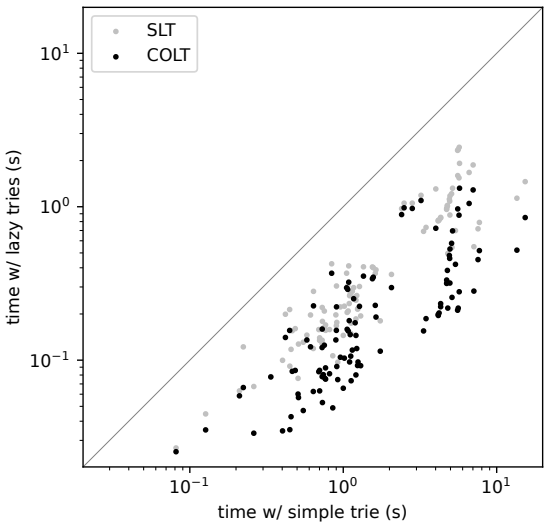


Figure 5.16: Impact of COLT.

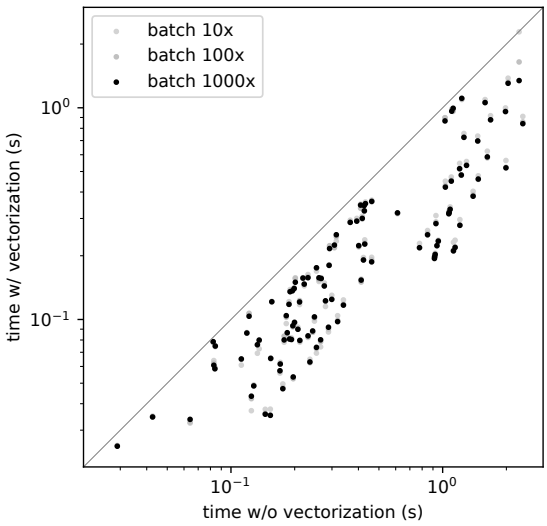


Figure 5.17: Impact of vectorization.

5.4.3 Impact of COLT and Vectorization

The three key ingredients that make Free Join efficient are 1. our algorithm to optimize the Free Join plan by factoring, 2. the COLT data structure, and 3. the vectorized execution algorithm. We conduct an ablation study to evaluate the performance impact of these components. But if we do not optimize the Free Join plan converted from a binary plan and execute it as-is, Free Join would behave identically to binary join. Since we have already compared Free Join with binary join in Section 5.4.2, we do not repeat it here. Therefore, our ablation study includes two sets of experiments, evaluating the impact of COLT and vectorization respectively.

Figure 5.16 compares the run time of Free Join using different trie data structures. The baseline fully expands each trie ahead of time, and we call this *simple trie*. Another data structure, *simple lazy trie* (SLT), expands the first level of each trie ahead of time, while expanding the inner levels lazily. This is the same strategy as proposed by [FBS+20]. Finally, COLT is our column-oriented lazy trie. In all three cases, we use the default vectorization batch size 1000. The experiments show the average (geometric mean) speedup of COLT is 1.91x and 8.47x, over SLT and simple trie respectively, and the maximum speedup over them is 11.01x and 26.29x, respectively.

Figure 5.17 compares the run time of Free Join using different vectorization batch sizes. The baseline uses no vectorization, i.e., we set the batch size to 1. Then we adjust the batch size among 10, 100, and 1000. The data does not show significant performance differences among the different batch sizes – it appears *any amount of vectorization is better than none*. For short-running queries, a smaller batch size perform slightly better, and for longer running queries a larger batch size wins. We conjecture this is due to a smaller batch having less overhead, leading to lower latency, while a larger batch size speeds up large joins better, leading to better throughput. Overall, using the default batch size 1000 leads to an average (geometric mean) speedup of 2.12x, and a maximum speedup of 5.33x over non-vectorized Free Join.

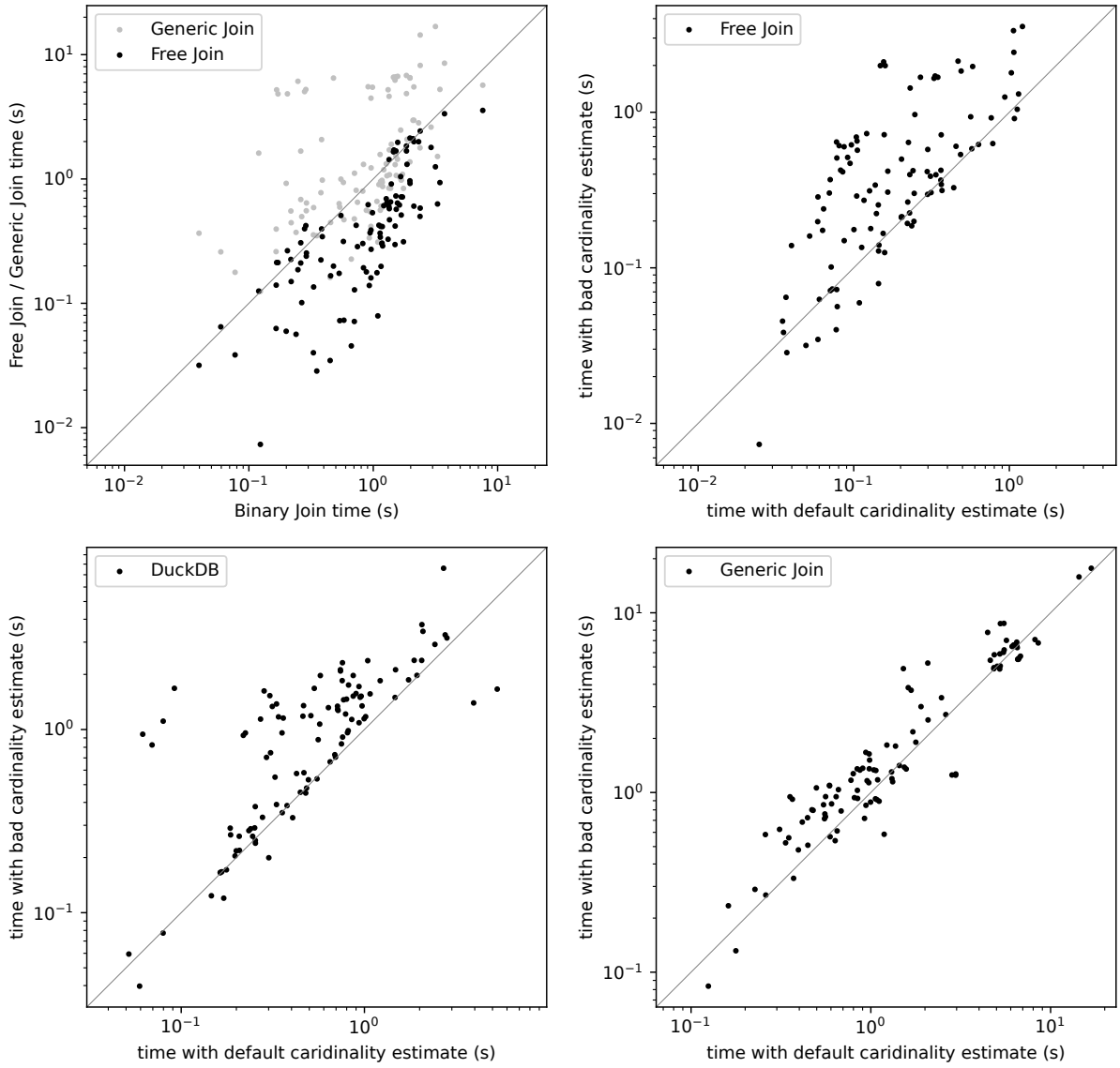


Figure 5.18: Run time of join algorithms with good and bad cardinality estimates. The first plot compares the run time of Free Join, Generic Join and binary join on JOB with bad cardinality estimates. The remaining three plots compare the run time of each algorithm with good and bad cardinality estimates on JOB.

5.4.4 Robustness Against Poor Plans

Our last set of experiments compares Free Join, Generic Join and binary join on their sensitivity to the quality of the query plan. Many believe WCOJ algorithms suffer less from poor query planning, due to its asymptotic guarantees. Our experience with Q13 from Section 5.4.2 also seems to confirm this intuition. However, our experimental results tell a different story. As the first plot in Figure 5.18 shows, the relative performance of the three algorithms stays the same with good and bad plans, with Free Join being the fastest, Generic Join the slowest, and binary join in the middle. However, as shown in Figure 5.18, Free Join seems to slow down as much as binary join when the plan is bad (there are many points far above the diagonal). It turns out with a poor cardinality estimate, DuckDB routinely outputs bushy plans that materialize large results. We have noted in Section 5.4.2 that our materialization strategy is simplistic, so with larger intermediates it leads to more severe slowdown. In comparison, Generic Join slows down less (the data points are close to the diagonal). However, it was the slowest to begin with, and since overheads like trie building dominates Generic Join's run time, a bad plan does not make it much slower. Overall, we believe Free Join can be more robust to bad plans with faster materialization.

5.5 Limitations and Future Work

With Free Join we have made a first step to bring together the worlds of binary join and WCOJ algorithms. There are three obvious limitations that require future work. Our current system is main memory only. If the data were to reside on disk, then COLT could be quite inefficient, because it requires repeated, random accesses to the data. Another limitation is that the optimization of a Free Join plan is split into two phases: a traditional cost-based optimization (currently done by DuckDB), followed by a heuristic-based optimization of the Free Join plan (factorization). This

has the advantage of reusing an existing cost-based optimizer, but the disadvantage is that an integrated optimizer may be able to find better plans. Third, our current optimizer does not make use of existing indices. It is known that the optimization problem for join plans becomes harder in the presence of foreign key indices [LGM+15], and we expect the same to hold for Free Join plans. All these limitations require future work. As we have designed Free Join to closely capture binary join while also generalizing it, we hope the solutions to these problems can also be smoothly transferred from binary join to Free Join.

Finally, we have made several observations during this project, some of them quite surprising (to us), which we believe deserve a future study. We observed that a major bottleneck is the materialization of intermediate results in bushy plans; an improved materialization algorithm may speed up Free Join on bushy plans. One promising idea is to be more aggressively lazy and keep COLTs unexpanded during materialization, which essentially leads to a factorized representation of the intermediates. We also observed that, contrary to common belief, a cyclic query does not necessarily mean WCOJ algorithms are faster, and an acyclic query does not mean they are slow. A natural question is thus "when exactly are WCOJ algorithms faster than binary join?" Answering this question will also help us design a better optimizer for Free Join. The optimizer can output a plan closer to WCOJ when it expects major speedups. We note that the query optimizer by [FBS+20] switches between Generic Join and binary join depending on the estimated cardinality. In contrast, an optimizer for Free Join should smoothly transform a Free Join plan to fully explore the design space between the two extremes of binary join and Generic Join. Finally, we realized that, rather surprisingly, Generic Join and traditional joins diverge in their choice of the inner table (called the cover in this chapter): Generic Join requires that to be the smallest (otherwise it is not optimal), while a traditional plan will chose it to be the largest (to save the cost of computing its hash table). Future work is required for a better informed decision for the choice of the inner relation.

Chapter 6

Optimizing Linear Algebra

Consider the Linear Algebra (LA) expression $sum((X - UV^T)^2)$ which defines a typical loss function for approximating a matrix X with a low-rank matrix UV^T. Here, $sum()$ computes the sum of all matrix entries in its argument, and A^2 squares the matrix A element-wise. Suppose X is a sparse, 1M x 500k matrix, and suppose U and V are dense vectors of dimensions 1M and 500k respectively. Thus, UV^T is a rank 1 matrix of size 1M x 500k, and computing it naively requires 0.5 trillion multiplications, plus memory allocation. Fortunately, the expression is equivalent to $sum(X^2) - 2U^TXV + U^TU * V^TV$. Here U^TXV is a scalar that can be computed efficiently by taking advantage of the sparsity of X, and, similarly, U^TU and V^TV are scalar values requiring only 1M and 500k multiplications respectively.

Optimization opportunities like this are ubiquitous in machine learning programs. State-of-the-art optimizing compilers such as SystemML [Boe19], OptiML[SLB+11], and Cumulon[HBY13] commonly implement syntactic rewrite rules that exploit the algebraic properties of the LA expressions. For example, SystemML includes a rule that rewrites the preceding example to a specialized operator[1] to compute the result in a streaming fashion. However, such syntactic rules

[1] See the SystemML Engine Developer Guide for details on the weighted-square loss operator `wsloss`.

fail on the simplest variations, for example SystemML fails to optimize $sum((X + UV^T)^2)$, where we just replaced $-$ with $+$. Moreover, rules may interact with each other in complex ways. In addition, complex ML programs often have many common subexpressions (CSE), that further interact with syntactic rules, for example the same expression UV^T may occur in multiple contexts, each requiring different optimization rules.

In this chapter we describe SPORES, a novel optimization approach for complex linear algebra programs that leverages relational algebra as an intermediate representation to completely represent the search space. SPORES first transforms LA expressions into traditional Relational Algebra (RA) expressions consisting of joins, unions and aggregates. It then performs cost-based optimizations on the resulting Relational Algebra expressions, using only standard identities in RA. Finally, the resulting RA expression is converted back to LA, and executed.

A major advantage of SPORES is that the rewrite rules in RA are *complete*, meaning they can be used to prove equivalence of any two expressions. Linear Algebra seems to require an endless supply of clever rewrite rules, but, in contrast, by converting to RA, we can prove that our set of just 13 rules are complete. Just like in Chapter 4 for Datalog$^\circ$, the RA expressions in this chapter are over K-relations [GKT07]; a tuple $X(i, j)$ is no longer true or false, but has a numerical value, e.g. 5, which could be interpreted, e.g., as the multiplicity of that tuple. In other words, the RA expressions that result from LA expressions are interpreted over bags instead of sets. The problem of checking equivalence of queries under bag semantics has a unique history. Chaudhuri and Vardi first studied the containment and equivalence problem under bag semantics, and claimed that two conjunctive queries are equivalent iff they are isomorphic: Theorem 5.2 in [CV93]. However, a proof was never produced. A rather long proof for this claim was given for the bag-set semantics in [CSN05]. Green provided a comprehensive proof, showing that two unions of conjunctive queries (UCQ) are equivalent under bag semantics iff they are isomorphic, by using sophisticated techniques involving multiple semirings [Gre09]. The completeness result

of our 13 rules relies on a similar result, but stated for tensors rather than bags; we present here a simple and self-contained proof in Sec. 6.1.3. We note that, in contrast, *containment* of two UCQs with bag semantics is undecidable [IR95]; we do not consider containment in this dissertation. Finally, we prove that our optimizer rules are sufficient to convert any RA expression into its *canonical form*, i.e. to an UCQ, and thus can, in principle, discover all equivalent rewritings.

However, we faced a major challenge in trying to exploit the completeness of the rules. The search space is very large, typically larger than that encountered in standard database optimizers, because of the prevalence of unions +, large number of aggregates $\sum$, and frequent common subexpressions. To tackle this, SPORES adopts and extends a technique from compilers called *equality saturation* [TSTL11]. It uses a data structure called the E-Graph [Nel80] to compactly represent the space of equivalent expressions, and equality rules to populate the E-Graph, then leverages constraint solvers to extract the optimal expression from the E-Graph. We extend equality saturation with rule sampling and use a greedy extraction algorithm to quickly cover vast portions of the search space, and trade the guarantee of optimality for shorter compile time.

We have integrated SPORES into SystemML [Boe19], and show that it can derive all hand-coded rules of SystemML. We evaluated SPORES on a spectrum of machine learning tasks, showing competitive performance improvement compared with more mature heuristic-based optimizers. Our optimizer rediscovers all optimizations by the latter, and also finds new optimizations that contribute to up to 10X speedup.

We make the following contributions in this dissertation:

1. We describe a novel approach for optimizing complex Linear Algebra expressions by converting them to Relational Algebra, and prove the completeness of our rewrite rules (Sec. 6.1).

2. We present a search algorithm based on Equality Saturation that can explore a large search space while using little memory (Sec. 6.2).

3. We conduct an empirical evaluation of the optimizer using several real-world machine learning tasks, and demonstrate it's superiority over an heuristics-driven optimizer in SystemML (Sec. 6.3).

6.1 Representing the Search Space

6.1.1 Rules R_{LR}: from LA to RA and Back

In this section we describe our approach of optimizing LA expressions by converting them to RA. The rules converting from LA to RA and back are denoted R_{LR}.

To justify our approach, let us revisit our example loss function written in LA and attempt to optimize it using standard LA identities. Here we focus on algebraic rewrites and put aside concerns about the cost model. Using the usual identities on linear algebra expressions, one may attempt to rewrite the original expression as follows:

$$
\begin{aligned}
& sum((X - UV^T)^2) \\
=& sum((X - UV^T) * (X - UV^T)) \\
=& sum(X^2 - 2X * UV^T + (UV^T)^2) \\
=& sum(X^2) - 2sum(X * UV^T) + sum((UV^T)^2)
\end{aligned}
$$

At this point we are stuck trying to rewrite $sum(X * UV^T)$ (recall that $*$ is element-wise multiplication); it turns out to be equal to $sum(U^T XV)$, for any matrices X, U, V (and it is equal to the scalar $U^T XV$ when U, V are column vectors), but this does not seem to follow from standard LA identities like associativity, commutativity, and distributivity. Similarly, we are stuck trying to rewrite $sum((UV^T)^2)$ to $sum(U^T U * V^T V)$. Current systems manually add syntactic rewrite rules,

Table 6.1: Representation of expressions in LA and RA.

	A			x		$A * x^T$			Ax	
LA	$\begin{bmatrix} 0 & 5 \\ 7 & 0 \end{bmatrix}$			$\begin{bmatrix} 3 \\ 2 \end{bmatrix}$		$\begin{bmatrix} 0 & 10 \\ 21 & 0 \end{bmatrix}$			$\begin{bmatrix} 10 \\ 21 \end{bmatrix}$	
RA	i	j	#	j	#	i	j	#	i	#
	1	2	5	1	3	1	2	10	1	10
	2	1	7	2	2	2	1	21	2	21

whenever such a special case is deemed frequent enough to justify extending the optimizer.

Instead, our approach is to expand out the LA expression element-wise. For example, assuming for simplicity that U, V are column vectors, we obtain

$$
\begin{aligned}
&sum((UV^T)^2) = \textstyle\sum_{i,j}(U_i \times V_j) \times (U_i \times V_j) \\
&= \textstyle\sum_{i,j}(U_i \times U_i) \times (V_j \times V_j) \\
&= (\textstyle\sum_i U_i \times U_i) \times (\sum_j V_j \times V_j) \\
&= U^T U \times V^T V
\end{aligned}
$$

The expressions using indices represent Relational Algebra expressions. More precisely, we interpret every vector, or matrix, or tensor, as a K-relation [GKT07] over the reals. In other words we view X_{ij} as a tuple $X(i, j)$ whose "multiplicity" is the real value of that matrix element. We interpret point-wise multiply as natural join; addition as union; sum as aggregate; and matrix multiply as aggregate over a join[2]. Table 6.1 illustrates the correspondence between LA and RA. We treat each matrix entry A_{ij} as the multiplicity of tuple (i, j) in relation A under bag semantics. For example $A_{2,1} = 7$, therefore the tuple $(2, 1)$ has multiplicity of 7 in the corresponding relation. The relation schema stores the size of each dimension. $A * x^T$ denotes element-wise multiplication,

[2]In the implementation, we use outer join for point-wise multiply and addition, where we multiply and add the matrix entries accordingly. In this chapter we use join and union to simplify presentation.

Table 6.2: LA and RA operators. The type $M_{M,N}$ is a matrix of size $M \times N$; $[i, j]$ is a list of attribute names; $R_{i:M,j:N}$ is a relation with attributes i of size M and j of size N; S_1, S_2, S, and U are sets of attributes. `elemmult` and `elemplus` are broadcasting.

	name	type	syntax
LA	mmult	$M_{M,L} \times M_{L,N} \to M_{M,N}$	AB
	elemmult	$M_{M,N} \times M_{M,N} \to M_{M,N}$	$A * B$
	elemplus	$M_{M,N} \times M_{M,N} \to M_{M,N}$	$A + B$
	rowagg	$M_{M,N} \to M_{M,1}$	$sum_{row}A$
	colagg	$M_{M,N} \to M_{1,N}$	$sum_{col}A$
	agg	$M_{M,N} \to M_{1,1}$	$sumA$
	transpose	$M_{M,N} \to M_{N,M}$	A^T
conv.	bind	$M_{M,N} \times [i, j] \to R_{i:M,j:N}$	$_{[i,j]}A$
	unbind	$R_{i:M,j:N} \times [i, j] \to M_{M,N}$	$_{[-i,-j]}A$
RA	join	$R_{S_1} \times R_{S_2} \to R_{S_1 \cup S_2}$	$A \times B$
	union	$R_{S_1} \times R_{S_2} \to R_{S_1 \cup S_2}$	$A + B$
	agg	$R_S \times U \to R_{S \setminus U}$	$\sum_U A$

where each element A_{ij} of the matrix is multiplied with the element x_j of the row-vector x^T. In RA it is naturally interpreted as the natural join $A(i, j) \bowtie x(j)$, which we write as $A(i, j) \times x(j)$. Similarly, Ax is the standard matrix-vector multiplication in LA, while in RA it becomes a query with a group by and aggregate, which we write as $\sum_j A(i, j) \times x(j)$. Our K-relations are more general than bags, since the entry of a matrix can be a real number, or a negative number; they correspond to K-relations over the semiring of reals $(\mathbb{R}, 0, 1, +, \times)$.

1. $A * B \to {}_{[-i,-j]}({}_{[i,j]}A \times {}_{[i,j]}B)$.
2. $A + B \to {}_{[-i,-j]}({}_{[i,j]}A + {}_{[i,j]}B)$.
3. $sum_{row}A \to [-i, \_] \sum_j {}_{[i,j]}A$. Similar for sum_{col}, sum.
4. $AB \to {}_{[-i,-k]} \sum_j ({}_{[i,j]}A \times {}_{[j,k]}B)$.
5. $A^\mathsf{T} \to {}_{[-j,-i]}{}_{[i,j]}A$.
6. $A - B \to A + (-1) * B$

Figure 6.1: LA-to-RA Ruleset R_{LR}.

We now describe the general approach in SPORES. The definition of LA and RA are in Table 6.2. LA consists of seven operators, which are those supported in SystemML [Boe19]. These operators all implement sum-product operations and take up the majority of run time in machine learning programs as we show in Section 6.3.2. We also support common operations like division and logarithm as we discuss in Section 6.2.3. RA consists of only three operators: $\times$ (natural join), + (union), and $\sum$ (group-by aggregate). Difference is represented as $A - B = A + (-1)B$ (this is difference in $\mathbb{R}$; we do not support bag difference, i.e. difference in $\mathbb{N}$ like $3 - 5 = 0$, because there is no corresponding operation in LA), while selection can be encoded by multiplication with relations with 0/1 entries. We call an expression using these three RA operators an *RPlan*, for Relational Plan, and use the terms RPlan and RA/relational algebra interchangeably. Finally, there are two operators, *bind* and *unbind* for converting between matrices/vectors and K-relations.

The translation from LA to RA is achieved by a set of rules, denoted R_{LR}, and shown in Figure 6.1. The bind operator $[i,j]$ converts a matrix to a relation by giving attributes i, j to its two dimensions; the unbind operator $[-i,-j]$ converts a relation back to a matrix. For example, $[-j,-i]\,[i,j]\,A$ binds A's row indices to i and its column indices to j, then unbinds them in the opposite order, thereby transposing A.

SPORES translates an LA expression into RA by first applying the rules R_{LR} in Figure 6.1 to each LA operator, replacing it with an RA operator, preceded by *bind* and followed by *unbind*. Next, it eliminates consecutive unbind/bind operators, possibly renaming attributes, e.g. $[k,l]\,[-i,-j]\,A$ becomes $A[i \to k, j \to l]$, which indicates that the attributes i and j in A's schema should be renamed to k and l, by propagating the rename downward into A. As a result, the entire LA expression becomes an RA expression (RPlan), with *bind* on the leaves and *unbind* at the top.

1. $A \times (B + C) = A \times B + A \times C$
2. $\sum_i (A + B) = \sum_i A + \sum_i B$
3. If $i \notin A$, $A \times \sum_i B = \sum_i (A \times B)$ (else rename i)
4. $\sum_i \sum_j A = \sum_{i,j} A$
5. If $i \notin Attr(A)$, then $\sum_i A = A \times dim(i)$
6. $A + (B + C) = +(A, B, C)$ (assoc. & comm.)
7. $A \times (B \times C) = \times(A, B, C)$ (assoc. & comm.)

Figure 6.2: RA equality rules R_{EQ}.

6.1.2 Rules R_{EQ} : from RA to RA

The equational rules for RA consists of seven identities shown in Figure 6.2, and denoted by R_{EQ}. The seven rules are natural relational algebra identities, where $\times$ corresponds to natural join, + to union (of relations with the same schema) and $\sum_i$ to group-by and aggregate. In rule 5, $i \notin Attr(A)$ means that i is not an attribute of A, and $dim(i)$ is the dimension of index i. For a very simple illustration of this rule, consider $\sum_i 5$. Here 5 is a constant, i.e. a relation of zero arity, with no attributes. The rule rewrites it to $5dim(i)$, where $dim(i)$ is a number representing the dimension of i.

6.1.3 Completeness of the Optimization Rules

As we have seen at the beginning of this section, when rewriting LA expressions using identities in linear algebra we may get stuck. Instead, by rewriting the expressions to RA, the seven identities in R_{EQ} are much more powerful, and can discover more rewrites. We prove here that this approach is *complete*, meaning that, if two LA expressions are semantically equivalent, then their equivalence can be proven by using rules R_{EQ}. The proof consists of two parts: (1) the rules R_{EQ} are sufficient to convert any RA expression e to its *normal form* (also called *canonical form*) $C(e)$, and back,

(2) two RA expressions e, e' are semantically equivalent iff they have isomorphic normal forms, $C(e) \equiv C(e')$.

We first give formal definitions for several important constructs. First, we interpret a relation as a function from *tuples* to a *semiring*. For simplicity we assume all attributes have the same domain $\mathbf{D}$. Fix a semiring $\mathbf{S}$. Recall from Chapter 4 that an $\mathbf{S}$-***relation*** is a function $A : \mathbf{D}^a \to \mathbf{S}$ where a is the *arity* of the relation. When the domain is $\mathbf{D} = [n]$ for some natural number n and $\mathbf{S} = \mathbb{R}$, then an $\mathbb{R}$-relation is a tensor over the reals. Next we define expressions over operators from Table 6.2. Recall the definitions of *monomials*, *sum-product expressions*, and *sum-sum product expressions* from Chapter 4. To reduce clutter, we will call sum-product expressions a *term*, and sum-sum product expressions a *polynomial*. An *expression* in RA is any polynomial. Note that a polynomial can have only one term, and a term can have an empty aggregate. As a reminder, such an expression are built up from atoms of the for $R(x_1, \ldots, x_a)$ the operations + and $\times$, and aggregation $\sum_x e$. Because the order of consecutive aggregates does not matter, we write $\sum_{\{x_1,\ldots,x_n\}} e$ for $\sum_{x_1} \cdots \sum_{x_n} e$.

We write $fv(e)$ and $bv(e)$ for the set of free and bound variables in e, respectively. We interpret every expression e as a lambda expression $\lambda fv(e).e$ where the parameters $fv(e)$ may follow a given order, e.g. from an unbind operator if e was converted from LA[3]. In the body of the lambda expression, any atom $R(x_1, \ldots, x_a)$ evaluates to some $s \in \mathbf{S}$, and $+, \times$ and $\sum$ compute over $\mathbf{S}$. We say two RA expressions are equivalent iff they evaluate to the same result given any same inputs:

Definition 6.1. *(Equivalence of Expressions)* Fix expressions e_1, e_2 over the relation symbols $R_1, \ldots, R_n$. We say that e_1, e_2 are ***equivalent*** over the semiring $\mathbf{S}$ and the domain $\mathbf{D}$ if they have the same free variables, and for all interpretations $\boldsymbol{I} = (R_1^I, \ldots, R_n^I)$ where $R_j^I : \mathbf{D}^{a_j} \to \mathbf{S}$ for $j = 1, \ldots, n$ the two expressions return the same answer, $e_1(\boldsymbol{I}) = e_2(\boldsymbol{I})$. We write $e_1 =_{\mathbf{S},\mathbf{D}} e_2$ to mean e_1 and e_2 are equivalent. When $e_1 =_{\mathbf{S},\mathbf{D}} e_2$ for all domains $\mathbf{D}$, then we abbreviate $e_1 =_{\mathbf{S}} e_2$; when this holds

[3]A bind operator $[i, j]$ converts a matrix A to an atom $A(i, j)$.

for all semirings S, then we write $e_1 = e_2$.

Before we formally define our canonical form, we need to define two syntactical relationships between our expressions, namely *homomorphism* and *isomorphism*. Fix terms $t = \sum_{\boldsymbol{x}} m$ and $t' = \sum_{\boldsymbol{x}'} m'$, and let $f : \boldsymbol{x} \to \boldsymbol{x}'$ be any function. Let $r \in bag(m)$ be an atom of m. We write $f(r)$ for the result of applying f to all variables of r. We write $f(bag(m))$ for the bag obtained by applying f to each atom $r \in bag(m)$.

Definition 6.2. *(Homomorphism)* Fix two terms t, t'. A ***homomorphism***, $f : t \to t'$, is a function $f : \boldsymbol{x} \to \boldsymbol{x}'$ such that $f(bag(m)) = bag(m')$.

Example 6.1. Let $t_1 = \sum_{vws} A(i,v) \times B(v,w) \times A(i,s)$ $t_2 = \sum_{jk} A^2(i,j) \times B(j,k)$ and $t_3 = \sum_{jk} A(i,j) \times B(j,k)$, and consider the function $f : \{v,w,s\} \to \{j,k\}$ defined by $v \mapsto j, w \mapsto k, s \mapsto j$. Then this is a homomorphism $f : t_1 \to t_2$. On the other hand f is *not* a homomorphism from $t_1 \to t_3$, because $f(bag(t_1)) = \{A(i,j), B(j,k), A(i,j)\}$ contains the atom $A(i,j)$ twice, while $bag(t_3)$ contains it only once.

Notice that t_1, t_2 must have exactly the same free variables. By convention, we extend f to be the identity on the free variables. The following facts are easily verified:

Example 6.1. Every homomorphism $f : t_1 \to t_2$ is a ***surjective*** function $vars(t_1) \to vars(t_2)$.

Example 6.2. Homomorphisms are closed under composition.

A stronger correspondence between terms is an isomorphism:

Definition 6.3 (Term Isomorphism)**.** Fix two terms t, t'. An ***isomorphism*** is a homomorphism $f : t \to t'$ that is a bijection from $vars(t)$ to $vars(t')$. If an isomorphism exists, then we say that t, t' are ***isomorphic*** and write $t \equiv t'$.

Lemma 6.1. Fix two terms t_1 and t_2. If there exists hommorphisms $f : t_1 \to t_2$ and $g : t_2 \to t_1$ then the terms are isomorphic, $t_1 \equiv t_2$. More generally, any cycle of homomorphisms $t_1 \to t_2 \to t_3 \to \ldots \to t_n \to t_1$ implies that all terms are isomorphic.

Proof. The composition $g \circ f$ is a homomorphism $t_1 \to t_1$ which, by Fact 6.1, is a surjective function $vars(t_1) \to vars(t_1)$; since $vars(t_1)$ is a finite set, it follows that $g \circ f$ is a bijection, hence so are f and g. □

We are now ready to formally define the canonical form for RA expressions:

Definition 6.4. *(Canonical Form)* An RPlan expression (as defined in Table 6.2) is ***canonical*** if it is a polynomial containing no isomorphic terms.

We can canonicalize any expression by pulling + to the top and pushing × to the bottom, while combining isomorphic terms $c_1 t + c_2 t$ into $(c_1 + c_2)t$:

Lemma 6.2. For every RPlan expression there is a canonical expression equivalent to it.

Proof. The proof is a standard application of the rewrite rules R_{EQ} in Figure 6.2. □

We can identify canonical expressions syntactically using term isomorphism:

Definition 6.5. *(Isomorphic Canonical Expressions)* Fix two R-polynomials $e = c_0 + c_1 t_1 + \cdots + c_n t_n$ and $e' = c'_0 + c'_1 t'_1 + \cdots + c'_m t'_m$. We say that e and e' are ***isomorphic*** if $m = n$, $c_0 = c'_0$, and there exists a permutation $\sigma : [n] \to [n]$ such that $\forall i \in [n]$, $c_i = c'_{\sigma(i)}$, and $t_i \equiv t'_{\sigma(i)}$.

In other words, e, e' are isomorphic if they are essentially the same expression, up to commutativity of + and up to replacing terms t_i with isomorphic terms $t'_{\sigma(i)}$. In particular, isomorphic expressions have same free variables. Our ultimate goal is to identify canonical form isomorphism with equivalence. That is, two canonical expressions are equivalent iff they are isomorphic. For

any R-polynomial $e = c_0 + c_1 t_1 + \cdots c_n t_n$, we denote by $|vars(e)| \stackrel{\text{def}}{=} \max_i(|vars(t_i)|)$. Our main result is the following:

Theorem 6.1. (**Isomorphism Captures Equivalence**) Let e_1, e_2 be two canonical expressions. Then the following conditions are equivalent:

1. $e_1 \equiv e_2$
2. $e_1 = e_2$ 3. $e_1 =_{\mathbb{C}} e_2$ 4. $e_1 =_{\mathbb{R}} e_2$ 5. $e_1 =_{\mathbb{N}} e_2$
6. $e_1 =_{\mathbb{N},\mathbf{D}} e_2$, for some finite domain $\mathbf{D}$ s.t. $|\mathbf{D}| \geq \max(|vars(e_1)|, |vars(e_2)|)$.

The implications (1) $\Rightarrow$ (2) $\Rightarrow \cdots \Rightarrow$ (6) are straightforward. We prove below that (6) $\Rightarrow$ (1). In other words, we prove that, if e_1, e_2 are equivalent over the semiring of natural numbers $\mathbb{N}$ and over some domain "large enough", then their canonical forms must be isomorpic. Requiring $|\mathbf{D}|$ to be large enough is necessary, because otherwise two non-isomorphic expressions may be equivalent. For example, if we restrict the relations X, Y to be matrices of dimensions 1×1, then the expressions $\sum_{i,j} X(i,j) \times Y(i,j)$ and $\sum_{i,j} X(i,j) \times Y(j,i)$ have the same semantics, but different canonical form. For another example, if x, y, z are vectors of length 2, these two expressions are equivalent: $\sum_{i,j,k} x(i) \times y(j) \times z(k) + 2 \sum_i x(i) \times y(i) \times z(i)$ and $\sum_{i,j} x(i) \times y(i) \times z(j) + \sum_{i,j} x(i) \times y(j) \times z(i) + \sum_{i,j} x(j) \times y(i) \times z(i)$, although they are not equivalent when x, y, z are vectors of length ≥ 3.

Proof. We prove (6) $\Rightarrow$ (1). Assume that $e_1(\boldsymbol{I}) = e_2(\boldsymbol{I})$ for all interpretations $\boldsymbol{I}$ over the domain $\mathbf{D}$. We start by observing that the constant terms in e_1 and e_2 must be equal, i.e. if $e_1 = c_0 + \ldots$, $e_2 = c_0' + \ldots$ then $c_0 = c_0'$. This follows by choosing $\boldsymbol{I}$ to consist of empty relations, in which case $e_1(\boldsymbol{I}) = c_0$ and $e_2(\boldsymbol{I}) = c_0'$, proving $c_0 = c_0'$. Thus, we can cancel the constant terms and assume w.l.o.g. that e_1, e_2 have no constant terms. Next, we show that we can assume w.l.o.g. that e_1 and e_2 have no free variables. Otherwise, denote by $\boldsymbol{x}$ the free variables in e_1 and e_2 (they must be the same in

order for e_1, e_2 to be equivalent), and define $e_1' = \sum_{\boldsymbol{x}} e_2$ and $e_2' = \sum_{\boldsymbol{x}} e_2$. It is easy to check that e_1', e_2' are also equivalent and, if we prove that they are isomorphic, then so are e_1, e_2. Thus, we will assume w.l.o.g. that e_1, e_2 have no free variables. Suppose that e_1 and e_2 contain two terms that are isomorphic: that is, e_1 contains $c_i t_i$, e_2 contains $c_j' t_j'$, and $t_i \equiv t_j'$. In particular, $t_i = t_j'$ i.e. they are also equivalent. Assuming $c_i \geq c_j'$, we subtract $c_j' t_j'$ from both e_1 and e_2; now e_1 contains $(c_i - c_j')t_i$, while e_2 no longer contains t_j'. By repeating this process, we remove any pair of isomorphic terms from e_1, e_2. If e_1, e_2 were isomorphic, then after this process we remove all terms and both e_1, e_2 becomes 0. Suppose by contradiction that this is not the case, thus $e_1 = c_1 t_1 + c_2 t_2 + \ldots c_m t_m$, $e_2 = c_1' t_1' + \cdots + c_n' t_n'$, and, denoting $T \stackrel{\text{def}}{=} \{t_1, \ldots, t_m, t_1', \ldots, t_n'\}$ the set of terms in both expressions, no two terms in T are isomorphic. Assuming $m > 0$ or $n > 0$, we prove that $e_1 =_{\mathbb{N},\mathbf{D}} e_2$ is a contradiction.

Let us denote by $t < t'$ when there exists a homomorphism $t \to t'$. Then $<$ defines a partial order on T, i.e. there is no $<$-cycle, otherwise Lemma 6.1 would imply that some terms are isomorphic. Let $t_1 \in T$ be any minimal element under $<$, in other words there is no $t' \in T$ s.t. $t' < t_1$. Assume w.l.o.g. that t_1 is a term in e_1. We will construct an instance $\boldsymbol{I}$ that is "canonical" for t_1, and prove that $e_1(\boldsymbol{I}) \neq e_2(\boldsymbol{I})$. Assume $t_1 = \sum_{\boldsymbol{x}} r_1^{k_1} \times \cdots r_m^{k_m}$, where $r_1, \ldots, r_k$ are distinct atoms, and $r_i^{k_i} \stackrel{\text{def}}{=} r_i \times \cdots \times r_i$ (k_i times). Let $n = |\boldsymbol{x}|$, and recall that, by assumption, $|\mathbf{D}| \geq n$. Choose any injective function $\theta : \boldsymbol{x} \to \mathbf{D}$; to reduce clutter we assume w.l.o.g. that $\mathbf{D} = \{1, 2, \ldots, n\}$ and $\theta(x_1) = 1, \ldots, \theta(x_n) = n$. Let $u_1, \ldots, u_m$ be m variables over $\mathbb{N}$, one for each distinct atom in t_1. We define the canonical $\boldsymbol{I}$ as follows. For each relational symbol R, and for any atom $r_i = R(x_{j_1}, \ldots, x_{j_a})$ that uses the symbol R, we define $R^I(j_1, \ldots, j_a) \stackrel{\text{def}}{=} u_i$; for all other tuples in $\mathbf{D}^a$ we define $R(\ldots) = 0$. This completes the definition of $\boldsymbol{I}$. We make two claims. First, $t_1(\boldsymbol{I})$ is a multivariate polynomial containing the monomial $c u_1^{k_1} \cdots u_m^{k_m}$. To see this, write $t = \sum_{\boldsymbol{y}} r_1' \times \cdots \times r_q'$, and observe that $t(\boldsymbol{I}) = \sum_{\tau:\boldsymbol{y}\to\mathbf{D}} \tau(r_1') \times \cdots \times \tau(r_q')$. When $\tau = \theta$ the R-monomial $\tau(r_1') \times \cdots \times \tau(r_q') = \theta(r_1') \times \cdots \times \theta(r_q')$ is precisely $u_1^{k_1} \cdots u_m^{k_m}$. Second, we claim that,

for any other term $t \in T$, its value $t(\boldsymbol{I})$ on the canonical instance is some multivariate polynomial in $u_1, \ldots, u_m$ that does *not* contain the monomial $u_1^{k_1} \cdots u_m^{k_m}$. Indeed, suppose it contained this monomial: then we prove that there exists a homomorphism $t \to t_1$, contradicting the assumption that t_1 is minimal. To see this, consider again $t(\boldsymbol{I}) = \sum_{\tau:\boldsymbol{y}\to\mathbf{D}} \tau(r'_1) \times \cdots \times \tau(r'_q)$. If this expression includes the monomial $u_1^{k_1} \cdots u_m^{k_m}$, then for some function $\tau : \boldsymbol{y} \to \mathbf{D}$, the bag $\{\theta'(r'_1), \ldots, \theta'(r'_q)\}$ must contain precisely the atom $\theta(r_1)$ k_1-times, the atom $\theta(r_2)$ k_2-times, etc. But that means that τ is a homomorphism $t \to t_1$ (since $\mathbf{D}$ and $vars(t_1)$ are isomorphic via θ), contradicting our assumption that t_1 is minimal.

Thus, we have established that both $e_1(\boldsymbol{I})$ and $e_2(\boldsymbol{I})$ are multivariate polynomials in $u_1, \ldots, u_m$, but the first expression contains the monomial $u_1^{k_1} \cdots u_m^{k_m}$ while the second does not contain it. Since $e_1 = e_2$, these two polynomials must have the same values for all choices of natural numbers $u_1, \ldots, u_m \in \mathbb{N}$. It is well known from classical algebra that, in this case, the two polynomials are identical, which is a contradiction.

For a simple illustration, assume $e_1 = t_1$ and $e_2 = t_2$, where $t_1 = \sum_{x,y,z} R(x,y) \times R(y,z) \times R(z,x)$ and $t_2 = \sum_i R(i,i)^3$. They are not isomorphic, and our proof essentially constructs an instance $\boldsymbol{I}$ on which their answers differ. Since we have a homomorphism $t_1 \to t_2$ but not vice versa, the instance is the canonical instance for t_1, i.e. $R^I(1,2) \stackrel{\text{def}}{=} u_1$, $R^I(2,3) \stackrel{\text{def}}{=} u_2$, $R^I(3,1) \stackrel{\text{def}}{=} u_3$, and all the other entries are 0. Then it is easy to verify that $t_1(\boldsymbol{I}) = 3u_1u_2u_3$ (there are three isomorphisms $t_1 \to t_1$), while $t_2(\boldsymbol{I}) = 0$. Notice that the canonical instance for t_2, $R^I(1,1) \stackrel{\text{def}}{=} u_1$ and all other entries are 0, does not make the two expressions different: $t_1(\boldsymbol{I}) = t_2(\boldsymbol{I}) = u_1^3$. □

We are now ready to establish the completeness of RA equalities, by showing any equivalent LA expressions can be rewritten to each other through the translation rules R_{LR} and the canonicalization rules R_{EQ}:

Theorem 6.2. (**Completeness of** R_{EQ}) Two LA expressions are semantically equivalent if and

only if their relational form can be rewritten to each other by following R_{EQ}.

In the following $R_{LR}(e)$ translates LA expression e into RA and $C(e)$ returns the normal form of e.

Proof. Translating e_1 and e_2 to RA preserves semantics under R_{LR}. By Lemma 6.2, normalizing $R_{LR}(e_1)$ and $R_{LR}(e_2)$ preserves semantics. By Theorem 6.1,

$$R_{LR}(e_1) =_{\mathbb{R}} R_{LR}(e_2) \iff C(R_{LR}(e_1)) \equiv C(R_{LR}(e_2))$$

Since every rule in R_{EQ} is reversible, the right-hand-side is true iff $R_{LR}(e_1)$ and $R_{LR}(e_2)$ can be rewritten to each other via R_{EQ}. □

6.2 Exploring the Search Space

With a complete representation of the search space by relational algebra, our next step is to explore this space and find the optimal expression in it. Traditional optimizing compilers commonly resort to heuristics to select from available rewrites to apply. SystemML implements a number of heuristics for its algebraic rewrite rules, and we discuss a few categories of them here.

Competing or Conflicting Rewrites The same expression may be eligible for more than one rewrites. For example, $sum(AB)$ rewrites to $sum(sum_{col}(A)^T * sum_{row}(B))$, but when both A and B are vectors the expression can also be rewritten to a single dot product. SystemML then implements heuristics to only perform the first rewrite when the expression is not a dot product. In the worst case, a set of rules interacting with each other may create a quadratic number of such conflicts, complicating the codebase.

Order of Rewrites Some rewrite should be applied after others to be effective. For example, X/y could be rewritten to $X * 1/y$ which may be more efficient, since SystemML provides efficient

implementation for sparse multiplication but not for division. This rewrite should occur before constant folding; otherwise it may create spurious expressions like $X/(1/y) \rightarrow X * (1/(1/y))$, and without constant folding the double division will persist. However, a rewrite like $1/(1 + exp(-X)) \rightarrow sigmoid(X)$ should come after constant folding, in order to cover expressions like $(3-2)/(1+exp(-X))$. Since SystemML requires all rewrites to happen in one phase and constant folding another, it has to leave out[4] rewrites like $X/y \rightarrow X * 1/y$.

DEPENDENCY ON INPUT/PROGRAM PROPERTIES Our example optimization from $sum((X - UV^T)^2)$ to $sum(X^2) - 2U^TXV + U^TU * V^TV$ improves performance only if X is sparse. Otherwise, computing X^2 and $X * UV^T$ would both create dense intermediates. Similarly, some rewrites depend on program properties like common subexpressions. Usually, these rewrites only apply when the matched expression shares no CSE with others in order to leverage common subexpression elimination. Testing input and program properties like this becomes tedious over a large code base, adding burden to implementation and maintenance.

COMPOSING REWRITES Even more relevant to us is the problem of composing larger rewrites out of smaller ones. Our equality rules R_{EQ} are very fine-grained, and any rule is unlikely to improve performance on its own. Our example optimization from $sum((X - UV^T)^2)$ to $sum(X^2) - 2U^TXV + U^TU * V^TV$ takes around 10 applications of R_{EQ} rules. If an optimizer applies rewrites one by one, it is then very difficult, if not impossible, for it to discover the correct sequence of rewrites that compose together and lead to the best performance.

Stepping back, the challenge of orchestrating rewrites is known as the *phase-ordering problem* in compiler optimization. Tate et al. [TSTL11] proposed a solution dubbed *equality saturation* which we adapt and extend in SPORES.

[4]Another reason to leave out this rewrite is that $X * 1/y$ rounds twice, whereas X/y only rounds once.

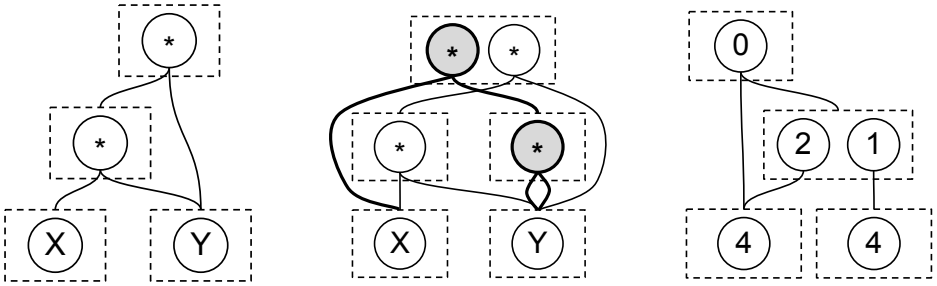


Figure 6.3: Left: E-Graph representing $(X \times Y) \times Y$, and the graph after applying associativity to the root (middle). New nodes are in gray. Each dashed box is an E-Class. Right: the CSE problem. Each node shows its cost.

6.2.1 Equality Saturation

Equality saturation optimizes an expression in two steps:

Saturation: given the input expression, the optimizer enumerates equivalent expressions and collects them into a compact representation called the E-Graph [Nel80].

Extraction: given a cost function, the optimizer selects the optimal expression from the E-Graph. An expression is represented by a subgraph of the E-Graph, and the optimizer uses a constraint solver to find the subgraph equivalent to the input that is optimal according to the cost function.

The E-Graph Data Structure

An E-Graph represents sets of equivalent expressions. A node in the graph is called an E-Class, which contains the root operators of a set of equivalent expressions. The edges are similar to the edges in an abstract syntax tree; but instead of pointing from an operator directly to a child, each edge points from an operator to an E-Class of expressions. For example, in Figure 6.3 the top class in the middle represents the set of equivalent expressions $\{(X \times Y) \times Y, X \times (Y \times Y)\}$. Note that the class represents two expressions, each with 2 appearances of Y and one appearance of X, whereas each variable only appears once in the E-Graph. This is because the E-Graph makes sure

its expressions share all possible common subexpressions. As the size of the graph grows, this compression becomes more and more notable; in some cases a graph can represent a number of expressions exponential to its size [TSTL11]. We take advantage of this compression in SPORES to efficiently cover vast portions of the search space. If saturation, as described below, carries out to convergence, the E-Graph represents the search space exhaustively.

An E-Graph can also be seen as an AND-OR DAG over expressions. Each E-Class is an OR node whose children are equivalent expressions from which the optimizer chooses from. Each operator is an AND node whose children must all be picked if the operator itself is picked. In this dissertation we favor the terms E-Graph and E-Class to emphasize each OR node is an equivalence class.

Saturating the E-Graph

At the beginning of the optimization process, the optimizer instantiates the graph by inserting the nodes in the syntax tree of the input expression one by one in post order. For example, for input $(X \times Y) \times Y$, we construct the left graph in Figure 6.3 bottom-up. By inserting in post order, we readily exploit existing common subexpressions in the input. Once the entire input expression is inserted, the optimizer starts to extend the graph with new expressions equivalent to the input. It considers a list of equations, and matches either side of the equation to subgraphs of the E-Graph. If an equation matches, the optimizer then inserts the expression on the other side of the equation to the graph. For example, applying the associativity rule extends the left graph in Figure 6.3 with $X \times (Y \times Y)$, resulting in the right graph. Figure 6.4 shows the pseudo code for this process. While inserting new expressions, the optimizer checks if any subexpression of the new expression is already in the graph. If so, it reuses the existing node, thereby exploiting all possible common-subexpressions to keep the E-Graph compact. In Figure 6.3, only two $\times$ are added since the variables X and Y are already in the graph. Once the entire new expression has been added,

```
def saturate(egraph, equations):
  for eq in equations:
    matches = egraph.match(eq.lhs)
    for eclass in matches:
      ec = egraph.add(eq.rhs)
      egraph.merge(eclass, c)
```

```
def add(expr):
  ID = egraph.find(expr)
  if ID != NULL:
    return ID
  else:
    cids = expr.children.map(add)
    ID = egraph.insert(expr.op, cids)
    return ID
```

Figure 6.4: Equality saturation pseudocode.

the optimizer then merges the newly created E-Class at its root with the E-Class containing the matched expression, asserting them equal. Importantly, the optimizer also propagates the congruent closure of this new equality. For example, when $A+A$ is merged with $2\times A$, the optimizer also merges $(A+A)^2$ with $(2\times A)^2$. Figure 6.4 shows the pseudo code for adding an expression to E-Graph. This process of match-and-insert is repeated until the graph stops changing, or reaching a user-specified bound on the number of saturation iterations. If this process does converge, that means no rule can add new expressions to the graph any more. If the set of rules are complete, as is our R_{EQ}, convergence of saturation implies the resulting E-Graph represents the transitive closure of the equality rules applied to the initial expression. In other words, it contains *all* expressions equivalent to the input under the equality rules.

The outer loop that matches equations to the graph can be implemented by a more efficient algorithm like the Rete algorithm [For82] when the number of equations is large. However, we did not find matching to be expensive and simply match by traversing the graph. Our implementation uses the E-Graph data structure from the `egg` [WWF+20] library.

Dealing with Expansive Rules

While in theory equality saturation will converge with well-constructed rewrite rules, in practice the E-Graph may explode for certain inputs under certain rules. For example, a long chain of multiplication can be rewritten to an exponential number of permutations under associativity and commutativity (AC rules). If we apply AC rules everywhere applicable in each iteration, the graph would soon use up available memory. We call this application strategy the *depth-first* strategy because it eagerly applies expansive rules like AC. AC rules by themselves rarely affect performance [KKL15], and SystemML also provides the fused `mmchain` operator that efficiently computes multiplication chains, so permuting a chain is likely futile. In practice, AC rules are useful because they can enable other rewrites. Suppose we have a rule $R_{factor} : A{\times}X{+}B{\times}X \rightarrow (A{+}B){\times}X$ and an expression $U \times Y + Y \times V$. Applying commutativity to $Y \times V$ would then transform the expression to be eligible for R_{factor}. With this insight, we change each saturation iteration to sample a limited number of matches to apply per rule, instead of applying all matches. This amounts to adding `matches = sample(matches, limit)` between line 3 and line 4 in Figure 6.4. Sampling encourages each rule to be considered equally often and prevents any single rule from exploding the graph. This helps ensure good exploration of the search space when exhaustive search is impractical. But when it is possible for saturation to converge and be exhaustive, it still converges with high probability when we sample matches. Our experiments in Section 6.3.3 show sampling always preserve convergence in practice.

Extracting the Optimal Plan

A greedy strategy to extract the best plan from the saturated E-Graph is to traverse the graph bottom-up, picking the best plan at each level. This assumes the best plan for any expression also contains the best plan for any of its subexpressions. However, the presence of common

subexpressions breaks this assumption. In the right-most graph in Figure 6.3 each operator node is annotated with its cost. Between the nodes with costs 1 and 2, a greedy strategy would choose 1, which incurs total cost of $1 + 4 = 5$. The greedy strategy then needs to pick the root node with cost 0 and the other node with cost 4, incurring a total cost of 9. However, the optimal strategy is to pick the nodes with 0, 2 and share the same node with cost 4, incurring a total cost of 6.

We follow Tate et.al. [TSTL11] and handle the complexity of the search problem with a constraint solver. We assign a variable to each operator and each E-Class, then construct constraints over the variables for the solver to select operators that make up a valid expression. The solver will then optimize a cost function defined over the variables; the solution then corresponds to the optimal expression equivalent to the input. We implement both the greedy strategy and the solver-based strategy and compare them in Section 6.3.3.

Constraint Solving and Cost Function

We encode the problem of extracting the cheapest plan from the E-Graph with integer linear programming (ILP). Figure 6.5 shows this encoding. For each operator in the graph, we generate a boolean variable B_{op}; for each E-Class we generate a variable B_c. For the root class, we use the variable B_r. Constraint $F(op)$ states that if the solver selects an operator, it must also select all its children; constraint $G(c)$ states that if the solver selects an E-Class, it must select at least one of its members. Finally, we assert B_r must be selected, which constrains the extracted expression to be in the same E-Class as the unoptimized expression. These three constraints together ensure the selected nodes form a valid expression equivalent to the unoptimized input. Satisfying these constraints, the solver now minimizes the cost function given by the total cost of the selected operators. Because each B_{op} represents an operator node in the E-Graph which can be shared by multiple parents, this encoding only assigns the cost once for every shared common subexpression. In our implementation, we use Gurobi [GO19] to solve the ILP problem.

$$Constraints \equiv B_r \wedge \bigwedge_{op} F(op) \wedge \bigwedge_{c} G(c)$$

$$F(op) \equiv B_{op} \rightarrow \bigwedge_{c \in op.children} B_c$$

$$G(c) \equiv B_c \rightarrow \bigvee_{op \in c.nodes} B_{op}$$

$$\textbf{minimize} \sum_{op} B_{op} \cdot C_{op} \textbf{ s.t. } Constraints$$

Figure 6.5: ILP constraint and objective for extraction.

$$\mathbf{S}[X \times Y] = min(\mathbf{S}[X], \mathbf{S}[Y])$$
$$\mathbf{S}[X + Y] = min(1, \mathbf{S}[X] + \mathbf{S}[Y])$$
$$\mathbf{S}[\sum_i X] = min(1, |i| \cdot \mathbf{S}[X])$$

Figure 6.6: Sparsity estimation. We define $sparsity = nnz/size$, i.e. a 0 matrix has sparsity 0.0^5. $|i|$ is the size of the aggregated dimension.

Each operation usually has cost proportional to the output size in terms of memory allocation and computation. Since the size of a matrix is proportional to its the number of non-zeroes (nnz), we use SystemML's estimate of nnz as the cost for each operation. Under our relational interpretation, this corresponds to the cardinality of relational queries. We use the simple estimation scheme in Figure 6.6, which we find to work well. We rely on SystemML's estimation for non-sum-product operators. Future work can hinge on the vast literature on sparsity and cardinality estimation to improve the cost model.

6.2.2 Schema and Sparsity as Class Invariant

In the rules R_{EQ} used by the saturation process, Rule (3) If $i \notin A$, $A \times \sum_i B = \sum_i (A \times B)$ contains a condition on attribute i which may be deeply nested in the expression. This means the optimizer cannot find a match with a simple pattern match. Fortunately, all expressions in the same class must contain the same set of free attributes (attributes not bound by aggregates). In other words, the set of free variables is invariant under equality. This corresponds precisely to the schema of a database - equivalent queries must share the same schema. We therefore annotate each class with its schema, and also enable each equation to match on the schema.

In general, we find class invariants to be a powerful construct for programming with E-Graphs. For each class we track as class invariant if there is a constant scalar in the class. As soon as all the children of an operator are found to contain constants, we can fold the operator with the constant it computes. This seamlessly integrates constant folding with the rest of the rewrites. We also treat sparsity as a class invariant and track it throughout equality saturation. Because our sparsity estimation is conservative, equal expressions that use different operators may have different estimates. But as soon as we identify them as equal, we can merge their sparsity estimates

[5]Some may find this definition counter-intuitive; we define it so to be consistent with SystemML.

by picking the tighter one, thereby improving our cost function. Finally, we also take advantage of the schema invariant during constraint generation. Because we are only interested in RA expressions that can be translated to LA, we only generate symbolic variables for classes that have no more than two attributes in their schema. This prunes away a large number of invalid candidates and helps the solver avoid wasting time on them. We implement class invariants using `egg`'s Metadata API.

6.2.3 Translation, Fusion and Custom Functions

Since equality saturation can rewrite any expression given a set of equations, we can directly perform the translation between LA and RA within saturation, simply by adding the translation rules R_{LR} from Figure 6.1. Furthermore, saturation has flexible support for custom functions. The simplest option is to treat a custom functions as a black box, so saturation can still optimize below and above them. With a little more effort, we have the option to extend our equations R_{EQ} to reason about custom functions, removing the optimization barrier. We take this option for common operators that are not part of the core RA semantics, e.g. square, minus and divide. In the best scenario, if the custom function can be modeled by a combination of basic operators, we can add a rule equating the two, and retain both versions in the same graph for consideration. In fact, this last option enables us to encode fused operators and seamlessly integrate fusion with other rewrite rules. As a result, the compiler no longer need to struggle with ordering fusion and rewrites, because saturation simultaneously considers all possible ordering. We note that although supporting custom functions require additional rules in SPORES, these rules are all identities, and they are much simpler than the heuristics rules in SystemML which need to specify when to fire a rule and how each rule interacts with another. Finally, although SystemML does not directly expose "physical" operators, e.g. different matrix multiplication algorithms, SPORES

readily supports optimization of physical plans. For example, we could use two distinct operators for two matrix multiplication algorithms, and both would always appear in the same E-Class. Both operators would share the same child E-Classes, therefore the additional operator only adds one node for every class that contains a matrix multiply.

6.2.4 Saturation v.s. Heuristics

Using equality saturation, SPORES elegantly remedies the drawbacks of heuristics mentioned in the beginning of section 6.2. First, when two or more conflicting rewrites apply, they would be added to the same E-Class, and the extraction step will pick the more effective one based on the global cost estimate. Second, there is no need to carefully order rewrites, because saturation simultaneously considers all possible orders. For example, when rules R_1 and R_2 can rewrite expression e to either $R_1(R_2(e))$ or $R_2(R_1(e))$, one iteration of saturation would add $R_1(e)$ and $R_2(e)$ to the graph, and another iteration would add both $R_1(R_2(e))$ and $R_2(R_1(e))$ to the same E-Class. Third, rules do not need to reason about their dependency on input or program properties, because extraction uses a global cost model that holistically incorporates factors like input sparsity and common subexpressions. Finally, every rule application in saturation applies one step of rewrite on top of those already applied, naturally composing complex rewrites out of simple ones.

6.2.5 Integration within SystemML

We integrate SPORES into SystemML to leverage its compiler infrastructure. SPORES plugs into the algebraic rewrite pass in SystemML; it takes in a DAG of linear algebra operations, and outputs the optimized DAG. Within SPORES, it first translates the LA DAG into relational algebra, performs equality saturation, and finally translates the optimal expression back into LA. We obtain matrix characteristics such as dimensions and sparsity estimation from SystemML. Since we did not focus

our efforts in supporting various operators and data types unrelated to linear algebra computation (e.g. string manipulation), we only invoke SPORES on important LA expressions from the inner loops of the input program.

6.3 Evaluation

We evaluate SPORES to answer three research questions about our approach of relational equality saturation:

- **Section 6.3.1**: can SPORES derive hand-coded rewrite rules for sum-product optimization?
- **Section 6.3.2**: can SPORES find optimizations that lead to greater performance improvement than hand-coded rewrites and heuristics?
- **Section 6.3.3**: does SPORES induce compilation overhead afforded by its performance gain?

We ran experiments on a single node with Intel E74890 v2 @ 2.80GHz with hyper-threading, 1008 GB RAM, 1 Nvidia P100 GPU, 8TB disk, and Ubuntu 16.04.6. We used OpenJDK 1.8.0, Apache Hadoop 2.7.3, and Apache Spark 2.4.4. Spark was configured to run locally with 6 executors, 8 cores/executor, 50GB driver memory, and 100GB executor memory. Our baselines are from Apache SystemML 1.2.0 and TensorFlow r2.1. We compile all TensorFlow functions with XLA through `tf.function`, and enable GPU.

6.3.1 Completeness of Relational Rules

Theoretically, our first hypothesis is validated by the fact that our relational equality rules are complete w.r.t. linear algebra semantics. To test the rules' coverage in practice, our first set of experiments check if SPORES can derive the hand-coded sum product rewrite rules in SystemML.

To do this, we input the left hand side of each rule into SPORES, perform equality saturation, then check if the rule's right hand side is present in the saturated graph. The optimizer is able to derive all 84 sum-product rewrite rules in SystemML using relational equality rules. Refer to [WHL+20] for a list of these rewrites. We believe replacing the 84 ad-hoc rules with our translation rules R_{LR} and equality rules R_{EQ} would greatly simplify SystemML's codebase. Together with equality saturation, our relational rules can also lead to better performance, as we demonstrate in the next set of experiments.

6.3.2 Run Time Measurement

We compare SPORES against SystemML's native optimizations for their performance impact. As baseline, we run SystemML with optimization level 2 (`opt2`), which is its default and includes all advanced rewrites like constant folding and common subexpression elimination. We additionally enable SystemML's native sum-product rewrites and operator fusion. When using SPORES, we disable SystemML's native sum-product rewrites, which means disabling the 84 rules discussed in Section 6.3.1. We compile and execute 5 real-world algorithms including Generalized Linear Model (GLM), Multinomial Logistic Regression (MLR), Support Vector Machine (SVM), Poisson Nonnegative Matrix Factorization (PNMF), and Alternating Least Square Factorization (ALS). We configure GLM and MLR to learn a probit model as a binary classifier. We take the implementation of these algorithms from SystemML's performance benchmark suite [Sys]. All algorithms were used as benchmarks in previous optimization research [ELB+17] [BRH+18]. We use the same input datasets from [BRH+18], specifically the Amazon books review dataset (Amazon/A) [HM16], the Airline on-time performance dataset (Flights/F) [(AS], the Netflix movie rating dataset (Netflix/N) [Kag], and the MNIST8M dataset (MNIST/M) [Bot]. In order to fit the computation in memory, we down sample each dataset to obtain inputs of small, medium and large sizes. For the Amazon

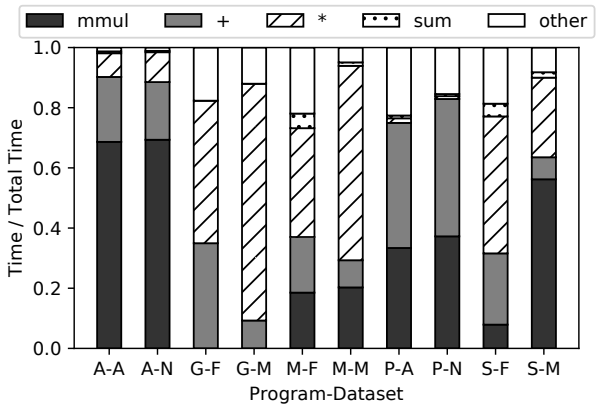


Figure 6.7: Run-time profile of benchmark programs.

and Netflix data, AS/NS contian 25k reviews, AM/NM 50k, and AL/NL 100k. We convert the data to a review matrix, where columns are items and rows are customers, then use it as input to ALS and PNMF. For the Flights and MNIST datasets, FS/MS contain 2.5M rows, FM/MM 5M, and FL/ML 10M. We use these datasets as input to GLM / MLR / SVM. Each algorithm learns if a flight is delayed more than 5 hours, or if an image shows the digit 2. In TensorFlow experiments we generate random inputs that match the size of the intermediate data in the corresponding benchmark. Our approach focuses on optimizing sum-product operations with the assumption that these operations take up the majority of run time in machine learning programs. We test this assumption by profiling our benchmark programs on the largest version of each dataset. Figure 6.7 shows that for each program on each input dataset, sum-product operations including matrix multiply, addition, point-wise multiply and summation together take up the great majority of run time (from 77.4% to 98.9%). For other heavy-hitting operations, we implement standard rewrite rules as discussed in Section 6.2.3. Figure 6.8 shows the program run time under SPORES optimization against SystemML's optimization. SPORES is competitive with the hand-coded rules in SystemML: for GLM and SVM, SPORES discovers the same optimizations as SystemML does. For ALS, MLR and PNMF, SPORES found new optimizations that lead to up to 10X speedup. We

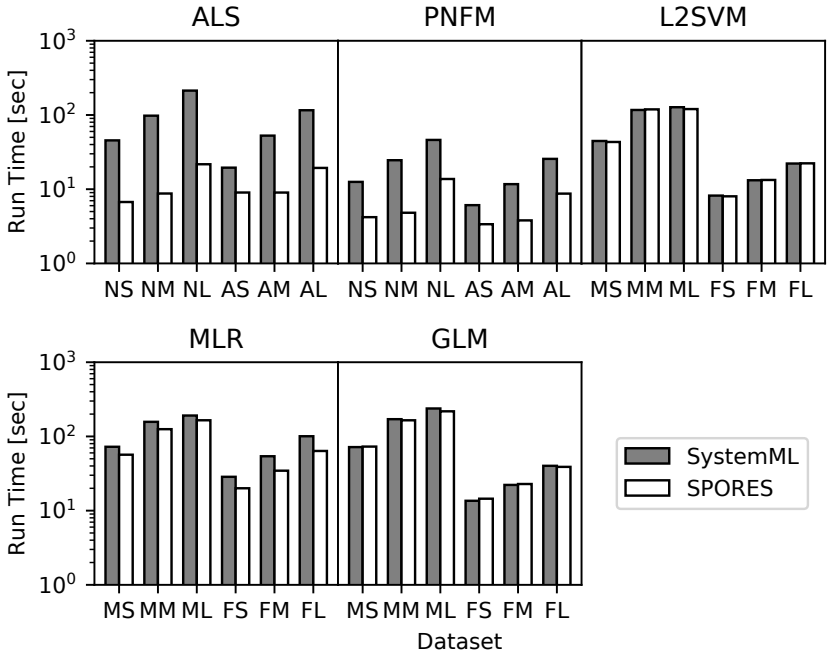


Figure 6.8: Run time under SystemML/SPORES compilation.

next analyze each benchmark in detail.

For **ALS**, SPORES leads to up to 10X speedup beyond SystemML's optimizations using our relational rules. Investigating the optimized code reveals the speedup comes from a rather simple optimization: SPORES expands $(UV^T - X)V$ to $UV^TV - XV$ to exploit the sparsity in X. Before the optimization, all three operations (2 matrix multiply and 1 minus) in the expression create dense intermediates because U and V are dense. After the optimization, XV can be computed efficiently thanks to the sparsity in X. UV^TV can be computed in one go without intermediates, taking advantage of SystemML's `mmchain` operator for matrix multiply chains. Although the optimization is straightforward, it is counter-intuitive because one expects computing $A(B+C)$ is more efficient than $AB + AC$ if one does not consider sparsity. For the same reason, SystemML simply does not consider distributing the multiplication and misses the optimization.

For **PNMF**, the speedup of up to 3.5X using RA rules attributes to rewriting $sum(WH)$ to

$sum_{col}(W) \cdot sum_{row}(H)$ which avoids materializing a dense intermediate WH. Interestingly, SystemML includes this rewrite rule but did not apply it during optimization. In fact, SystemML only applies the rule when WH does not appear elsewhere, in order to preserve common subexpression. However, although WH is shared by another expression in PNMF, the other expression can also be optimized away by another rule. Because both rules uses heuristics to favor sharing CSE, neither fires. This precisely demonstrates the limitation of heuristics.

For **MLR**, SPORES leads to up to 1.3X speedup. The important optimization[6] is $P * X - P * sum_{row}(P) * X$ to $P * (1 - P) * X$, where P is a column vector. This takes advantage of the `sprop` fused operator in SystemML to compute $P * (1 - P)$, therefore allocating only one intermediate. Note that the optimization factors out P, which is the exact opposite to the optimization in ALS that distributes multiply. Naive rewrite rules would have to choose between the two directions, or resort to heuristics to break ties.

In summary, SPORES improves performance consistently for ALS, PNMF and MLR across different data sizes. The impact of sparsity on performance can be gleaned from the particular optimizations: ALS optimization takes advantage of sparsity, while PNMF and MLR optimizations apply for either dense or sparse inputs.

For **SVM** and **GLM**, equality saturation finds the same optimizations as SystemML does, leading to speedup mainly due to operator fusion. Upon inspection, we could not identify better optimizations for **SVM**. For **GLM**, however, we discovered a manual optimization that should improve performance in theory, but did not have an effect in practice since SystemML cannot accurately estimate sparsity to inform execution.

[6] Simplified here for presentation. In the source code P and X are not variables but consist of subexpressions.

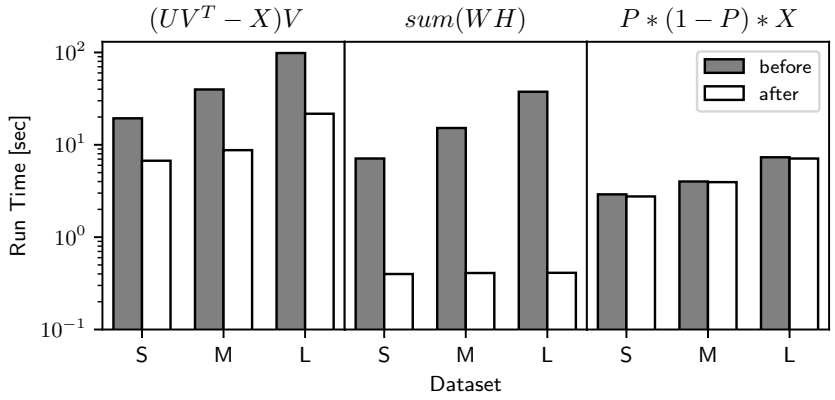


Figure 6.9: Run time before- and after-rewrite in TensorFlow.

Comparison Against TensorFlow

We ran additional experiments in TensorFlow to see if it can also benefit from SPORES's optimizations. For each of the 3 rewrites we discussed in Section 6.3.2, we coded the expressions before- and after-rewrite in TensorFlow. Then we compile each version with TensorFlow XLA and measure its run time. Figure 6.9 shows up to 90X and 50X speedup from the rewrites taken from ALS ($(UV^T - X)V$) and PNMF ($sum(WH)$) respectively. Upon inspection of the compiled code, we found XLA performs no optimization on the input expressions, likely due to certain heuristics preferring the unoptimized versions. For MLR ($P * (1 - P) * X$), XLA compiles the before- and after-rewrite expressions to the same code, so the run time stays the same. We were unable to compile our full benchmarks with XLA because the latter cannot compile certain operations on sparse matrices. When this improves, we expect to see SPORES bring its full potential to TensorFlow.

6.3.3 Compilation Overhead

In our initial experiments, SPORES induces nontrivial compilation overhead compared to SystemML's native rule-based rewrites. Figure 6.10 (sampling, ILP extraction) shows the compile

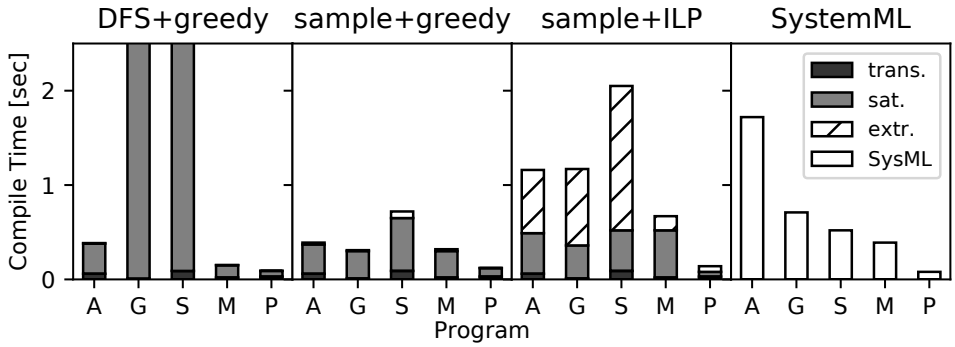


Figure 6.10: Compile time breakdown for different saturation and extraction strategies. Depth-first saturation reaches the 2.5s timeout compiling GLM and SVM.

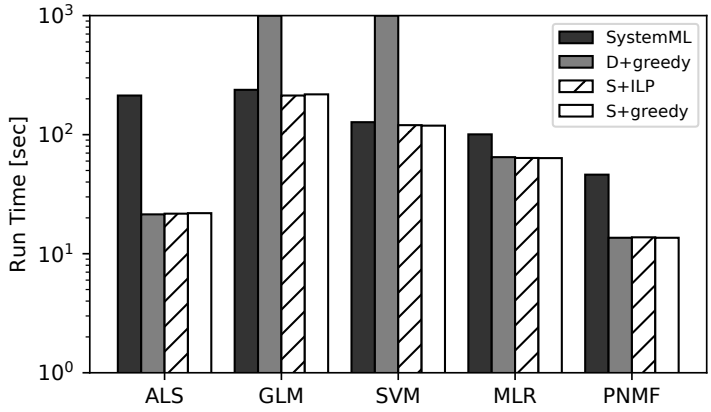


Figure 6.11: Performance impact of different saturation and extraction strategies. S is saturation with sampling, and D is depth-first saturation. Depth-first saturation runs into timeout compiling GLM and SVM.

time breakdown for each benchmark, and the majority of time is spent in the ILP solver. We therefore experiment with the greedy algorithm described in Section 6.2.1 to see if we can trade off guarantees of optimality for a shorter compile time. Figure 6.11 shows the performance impact of greedy extraction, and Figure 6.10 (sampling, greedy extraction) shows the compile time with it. Greedy extraction significantly reduces compile time without sacrificing *any* performance gain! This is not surprising in light of the optimizations we discussed in Section 6.3.2: all of these optimizations improve performance regardless of common subexpressions, so they are selected by both the ILP-based and the greedy extractor.

We also compare saturation with sampling against depth-first saturation in terms of performance impact and compile time. Recall the depth-first saturation strategy applies all matches per rule per iteration. As Figure 6.10 shows, sampling is slightly slower for ALS, MLR and PNMF, but resolves the timeout for GLM and SVM. This is because sampling takes longer to converge when full saturation is possible, and otherwise prevents the graph from blowing up before reaching the iteration limit. Indeed, saturation converges for ALS, MLR and PNMF, which means SPORES can guarantee the optimality of its result under the given cost model. Saturation does not not converge before reaching the iteration limit for GLM and SVM because of deeply nested $*$ and $+$ in the programs. Convergence may come as a surprise despite E-Graph's compaction – expansive rules like associativity and commutativity commonly apply in practice. However, the expression DAGs we encounter are often small (no more than 15 operators), and large DAGs are cut into small pieces by optimization barriers like uninterpreted functions.

Figure 6.10 compares the overall DAG compilation overhead of SystemML against SPORES with different extraction strategies. Note that the overhead of SystemML also includes optimizations unrelated to sum-product rewrites that are difficult to disentangle, therefore it only gives a sense of the base compilation time and does not serve as head-to-head comparison against SPORES. Although SPORES induces significant compilation overhead in light of the total DAG compilation

Table 6.3: Numerical characteristics of compiled programs.

Optimizer	SysML	SPORES	SysML	SPORES
Dataset	Airline		MNIST	
SVM (*acc.*)	82.4%	82.4%	96.8%	96.8%
MLR (R^2)	0.773	0.773	0.742	0.742
GLM (R^2)	0.618	0.618	0.671	0.671
Dataset	Netflix		Amazon	
PNMF (*iter.*)	18	18	36	36
ALS (*iter.*)	8	8	9	9

time of SystemML, the overhead is afforded by its performance gain. As we did not focus our efforts on reducing compile time, we believe there is plenty room for improvement, for example organizing rewrite rules to speed up saturation.

6.3.4 Numerical Considerations

Although our rewrite rules preserve semantics for the reals, they do not preserve semantics under floating point arithmetics. We therefore run experiments to see if SPORES sacrifices numerical accuracy. For L2SVM, we compare the accuracy of the trained model under SPORES / SystemML optimization; for MLR and GLM we compare the R^2 value; PNMF and ALS terminate after some loss falls below a threshold, therefore we compare the number of iterations until termination. Table 6.3 shows all these statistics are identical under SPORES / SystemML optimzation. Although SPORES does not optimize for numerical accuracy, equality saturation was used by Herbie [PSWT15] for that exact purpose. We are actively collaborating with Herbie's authors to develop a multi-objective optimizer for accuracy and run time.

6.4 SPORES Limitations and Future Work

We intend SPORES to be used to optimize machine learning programs that perform linear algebra operations. As such, SPORES is not a linear algebra solver, and operations like matrix inversion and calculating eigenvalues are out of scope. SPORES does not include a matrix decomposition operator, but the programmer can implement decomposition algorithms like our ALS and PNMF benchmarks. The operations we support already cover a variety of algorithms as we show in the evaluation. Similar scope is shared by existing research [BRH+18] [ELB+17] [KKC+17]. Although SPORES does not focus on deep learning, its design can be adapted to optimize deep models. We are experimenting to incorporate the identity rules from TASO into our framework. It would be challenging to extend the completeness theorem to the complex operators used in deep learning, but we expect our extension of equality saturation can find high-impact optimizations with short compile time. Finally, although our rewrite rules are complete, we had to resort to rule sampling and greedy extraction to cut down the overhead. Future work can investigate more intelligent rule application and extraction strategies. For example, the optimizer can learn which rules more likely lead to performance improvement and prioritizes firing those rules. Another direction is to incorporate plus into theoretical sum-product frameworks like FAQ [KNR16] and guarantee optimality.

Chapter 7

Optimizing Recursive Queries

Most database systems are designed to support primarily non-recursive (loop-free) queries. Their optimizers are based on the rule-driven, cost-based Volcano architecture, designed specifically for optimizing non-recursive query plans. However, most data science and machine learning workloads today involve some form of recursion or iteration. Examples include finding the connected components of a graph, computing the page rank, computing the network centrality, minimizing an objective function using gradient descent, etc. The importance of supporting recursive queries has been noted by system designers. Some modern data analytics systems, like Spark or Tensorflow, support for-loops. The SQL standard defines a limited form of recursive queries, using the `with` construct, and some popular engines, like Postgres or SQLite, do support this restricted form of recursion.

The Datalog language, covered in Chapter 3, is designed specifically for recursive queries, and it is gaining in popularity [AMC+10, RL18, Via21b, HGL11, SYI+16b, FZZ+19, SGL15, WBH15, FLVE20]. But the optimization problem for recursive queries is much less studied. The naïve evaluation of Datalog repeatly applies the rules in a loop, until a fixpoint is reached. Datalog engines typically optimize the loop body, without optimizing the actual loop. The few systems

that do, apply only limited optimization techniques, like magic set optimization and semi-naive evaluation, which are restricted to positive queries.

In this chapter we describe a new query optimization framework for recursive queries. Our framework replaces a recursive program with another, equivalent recursive program, whose body may be quite different, and thus focuses on optimizing the recursive program as a whole, not on optimizing its body in isolation; the latter can be done separately, using standard query optimization techniques. Our optimization is based on a novel rewrite rule for recursive programs, called the FGH-rule, which we implement using *program synthesis*, a technique developed in the programming languages and verification communities. We introduce a new method for inferring loop invariants, which extends the reach of the FGH-rule, and also show how to use global constraints on the data for semantic optimizations using the FGH-rule.

The FGH-Rule At the core of our approach is a novel, yet very simple rewrite rule, called the FGH-rule (pronounced *fig-rule*), which can be used to prove that two recursive programs are equivalent, even when their loop bodies are quite different. We show that the FGH-rule can express previously known optimizations for Datalog, including magic sets and semi-naive evaluation, and also a wide range of new optimizations. The optimized program is often significantly more efficient than the original program, and sometimes can have a strictly lower asymptotic complexity. We implemented a source-to-source optimizer using the FGH-rule, evaluated its effectiveness on several Datalog systems, and observed speedups of up to 4 orders of magnitude (Sec. 7.6).

For a taste of the FGH-optimization, consider the following example, from [ZYD+17, ZYI+18]: compute the connected components of an undirected graph $E(x, y)$. The Datalog program in Fig. 7.1 (a) achieves this by first computing the transitive closure relation $TC(x, y)$, then computing a min-aggregate query assigning to every node x the smallest label $L[y]$ of all nodes y reachable from x. In contrast, the optimized program in Fig. 7.1 (b) computes directly the CC label of every node x as the minimum of its own label and the smallest CC label of its neighbors, using a single

$$TC(x, y) \text{ :- } [x = y] \vee \exists z(E(x, z) \wedge TC(z, y))$$
$$CC[x] \text{ :- } \min_y\{L[y] \mid TC(x, y)\}$$

(a)

$$CC[x] \text{ :- } \min(L[x], \min_y\{CC[y] \mid E(x, y)\})$$

(b)

Figure 7.1: Unoptimized (a) and optimized (b) Datalog program for the connected components of an undirected graph.

recursive rule with min-aggregation. The space complexity of the transitive closure is $O(n^2)$, which, in practice, is prohibitively expensive on large graphs. On the other hand, the optimized query has space complexity $O(n)$.

Pattern Matching vs. Query Synthesis Applying the FGH-rule is an instance of *query rewriting using views*. In that problem we are given a set of view expressions and a query, and the task is to rewrite the query to use the view expressions rather than the base relations. This problem has been extensively studied in the literature [Hal01], and today's database systems perform it using pattern matching [GL01]. This is a form of transformational synthesis, where every candidate query rewriting is guaranteed to be correct, because it is obtained by applying a limited set of manually crafted rules (patterns), which are guaranteed to be correct. However, the FGH-rule often requires exploring a very large space, which cannot be covered by a limited set of rules. In this chapter we propose to use for this purpose *counterexample-guided inductive synthesis* (CEGIS), a technique designed for program sketching [STB+06, TJ07]. When applied to our context, we call this technique *query synthesis*. Unlike pattern matching, query synthesis explores a much larger space, by examining rewritings that are not necessarily correct, and need to be checked for correctness by a verifier (`z3` in our system). The verifier also produces a small counterexample

database for each rejected candidate, and these counterexamples are collected by the synthesizer and used to produce only candidate rewritings that pass all the previous counterexamples, which significantly prunes the search space of the synthesizer. We report in Sec. 7.6 synthesis times of less than 1 second, even for complex queries that use global constraints and require inferring loop invariants.

Monotone Queries and Semiring Semantics Datalog is, by definition, restricted to monotone queries. But queries that contain aggregates or negation (expressed in SQL via subqueries) are not monotone, and most systems that support recursion prohibit the combination of aggregates and recursion. This has two shortcomings: it limits what kind of queries the user can express, and also prevents many of our FGH-rewritings. For example, the simple computation of connected components in Fig. 7.1 (a) can be expressed in PostgreSQL, or in SQLite, or in Soufflé, because the first rule uses only recursion and the second rule uses only aggregation. However, none of these systems accepts the query in Fig. 7.1 (b), because it combines recursion and aggregation.[1] In order to express such queries, we work with the Datalog$^\circ$ language introduced in Chapter 4 that supports recursive aggregates by extending Datalog with semirings.

Loop Invariants One difficulty in reasoning about loops in programming languages is the need to discover loop invariants. Some (but not all) applications of the FGH-rule also require the discovery of loop invariants. We describe a novel technique for inferring loop invariants for Datalog$^\circ$ programs, by combining symbolic execution with equality saturation, and using a verifier. We execute symbolically the recursive program for a very small number of iterations (five in our system), obtain query expressions for the IDBs (the recursive predicates), and construct all identities satisfied by the IDBs. Then, we retain only candidates that hold at each iteration, and check each candidate for correctness using the SMT solver. By inferring and using loop invariants

[1] Prior work [GGZ91, SGL15] has proposed extending Datalog with min and max aggregates by explicitly redefining the semantics of recursive rules with aggregates. Our approach keeps the standard least fixpoint semantics, but generalizes the semiring.

we show that we can significantly improve some instances of magic-set optimizations from the literature: we call the new optimization *beyond magic*.

Constraints and Semantic Optimizations Optimizations that are conditioned on certain constraints on the database are known as *semantic optimizations* [RS94]. SQL optimizers routinely use key constraints and foreign key constraints to optimize queries. More powerful optimizations can be performed using the chase and back-chase framework [DPT99, PDST00], and these include optimizations under inclusion constraints, or conditional functional dependencies, or tuple generating constraints. However, all constraints that are useful for optimizing non-recursive queries are *local*. In contrast, the FGH-rule optimizes recursive queries, and therefore it can also exploit *global* constraints. For example, suppose the database represents a graph, and the global constraint states that the graph is a tree. This global constraint does not help optimize non-recursive queries, but can be used to great advantage to optimize some recursive queries; we give details in Sec. 7.1.3.

Equality Saturation Throughout our optimizer we need to manage symbolic expressions of queries, and their equivalence classes, as defined by a set of rules. This is achieved once again with equality saturation (EQSAT), a technique introduced in Chapter 6. We show how to use EQSAT for checking equality under constraints, inferring loop invariants, and "denormalization" (which is essentially query rewriting using views).

Contributions In summary, the main contribution of this chapter consists of a new, principled and powerful method for optimizing recursive queries. We make the following specific contributions:

- We introduce a simple optimization rule for recursive queries, called the FGH-rule (Sec. 7.1).
- We show the FGH-rule captures known optimizations (magic sets, PreM, semi-naive), (Sec. 7.1.1), new optimizations (Sec. 7.1.2), and optimizations under global constraints (Sec. 7.1.3).

- We present our novel framework for query optimization via the FGH-rule (Sec. 7.2).
- We describe how an SMT solver (Sec. 7.3) and a CEGIS system (Sec. 7.4) can be profitably integrated into our FGH-optimizer.
- We describe how to use an EQSAT system for various tasks in the FGH optimizer: loop-invariant inference, denormalization, and checking equivalence under constraints (Sec. 7.5).

7.1 The FGH-Rule

In this section we introduce a simple rewrite rule that allows us to rewrite an iterative program to another, possibly more efficient program. Then, we illustrate how this rule, when applied to Datalog$^\circ$ programs, can express several known optimizations in the literature, as well as some new ones.

Consider an iterative program that repeatedly applies a function F until some termination condition is satisfied, then applies a function G that returns the final answer Y:

$$\begin{aligned} &X \leftarrow X_0 \\ &\texttt{loop } X \leftarrow F(X) \texttt{ end loop} \\ &Y \leftarrow G(X) \end{aligned} \tag{7.1}$$

We call this an FG-program. The FGH-rule (pronounced *FIG-rule*) provides a sufficient condition for the final answer Y to be computed by the alternative program, called the GH-program:

$$\begin{aligned} &Y \leftarrow G(X_0) \\ &\texttt{loop } Y \leftarrow H(Y) \texttt{ end loop} \end{aligned} \tag{7.2}$$

Theorem 7.1 (The FGH-Rule)**.** If the following identity holds:

$$G(F(X)) = H(G(X)) \tag{7.3}$$

then the FG-program (7.1) is equivalent to the GH-program (7.2).

Proof. Let $X_0, X_1, X_2, \ldots$ denote the intermediate values of the FG-program, and $Y_0, Y_1, Y_2, \ldots$ those of the GH-program. By the FGH-rule, the following diagram commutes, proving the claim:

$$\begin{array}{ccccccccc} X_0 & \xrightarrow{F} & X_1 & \xrightarrow{F} & X_2 & \xrightarrow{F} & \cdots & \xrightarrow{F} & X_n \\ \downarrow{\scriptstyle G} & & \downarrow{\scriptstyle G} & & \downarrow{\scriptstyle G} & & & & \downarrow{\scriptstyle G} \\ Y_0 & \xrightarrow{H} & Y_1 & \xrightarrow{H} & Y_2 & \xrightarrow{H} & \cdots & \xrightarrow{H} & Y_n \end{array}$$

□

In this chapter we will apply the FGH-rule to optimize Datalog° programs. In this context, F is the ICO of the Datalog° program, X is the tuple of all its IDB predicates, and Y are the answer-IDB predicates. We will also make the natural assumption that G maps the initial state X_0 of the IDBs of the program (7.1) to the initial state Y_0 of (7.2). For example, if both programs are traditional Datalog programs, then the initial state consists of all IDBs being the empty set, which we denote, with some abuse, by $X_0 = \emptyset$, even when X consists of several mutually recursive IDBs. Similarly, $Y_0 = \emptyset$. Typically, G is a conjunctive query, which maps $\emptyset$ to $\emptyset$, and in that case the theorem implies that, if Eq. (7.3) holds, then the following Datalog° programs Π_1, Π_2 return the same answer Y:

$$\begin{array}{lll} \Pi_1: & X \text{ :- } F(X) & \Pi_2: \quad Y \text{ :- } H(Y) \\ & Y \text{ :- } G(X) & \end{array} \tag{7.4}$$

More generally, however, the theorem does not care about the termination condition of the

FG-programs (7.1). It only assumes that the GH-program is executed the same number of iterations as the FG-program. However, it follows immediately that, if F reaches a fixpoint, then so does H:

Corollary 7.1. If the FG-program reaches a fixpoint after n steps (meaning: $X_n = X_{n+1}$) then the GH-program also reaches a fixpoint after n steps ($Y_n = Y_{n+1}$). The converse fails: the GH-program may converge much faster than the FG-program.

In summary, the optimization proceeds as follows. Given an FG-program defined by the query expressions F and G, find a new query expression H such that the identity $G \circ F = H \circ G$ holds, then replace the FG-program with the GH-program. We will describe this process in detail in Sec. 7.2. In the remainder of this section we present several examples showing that the FGH-rule can express several known optimizations, like magic set rewriting, and new optimizations, like semantic optimizations using global constraints.

7.1.1 Simple Examples

Example 7.1 (Connected Components)**.** Consider the computation of the connected components of a graph, which is a well-known target of query optimization in the literature, see e.g., [ZYI+18]. The program is given in Fig. 7.1 (a), and its optimized version in Fig. 7.1 (b). The three transformations F, G, H are as follows:

$$F(TC) \stackrel{\text{def}}{=} TC' \qquad \text{where} \qquad TC'(x,y) \stackrel{\text{def}}{=} [x = y] \vee \exists z(E(x,z) \wedge TC(z,y))$$

$$G(TC) \stackrel{\text{def}}{=} CC \qquad \text{where} \qquad CC[x] \stackrel{\text{def}}{=} \min_y\{L[y] \mid TC(x,y)\}$$

$$H(CC) \stackrel{\text{def}}{=} CC' \qquad \text{where} \qquad CC'[x] \stackrel{\text{def}}{=} \min(L[x], \min_y\{CC[y] \mid E(x,y)\})$$

To check the FGH-rule, we compute $CC_1 \stackrel{\text{def}}{=} G(F(TC)) = G(TC')$, and $CC_2 \stackrel{\text{def}}{=} H(G(TC)) = H(CC)$, both shown in Fig. 7.2, and observe that it becomes identical to CC_1 after renaming the variables

$$
\begin{aligned}
CC_1[x] &\stackrel{\text{def}}{=} \min_y\{L[y] \mid TC'(x,y)\} \\
&= \min_y\{L[y] \mid [x=y] \vee \exists z(E(x,z) \wedge TC(z,y))\} \\
&= \min(L[x], \min_y\{L[y] \mid \exists z(E(x,z) \wedge TC(z,y))\}) \\
&= \min(L[x], \min_{y,z}\{L[y] \mid E(x,z) \wedge TC(z,y)\}) \\
CC_2[x] &\stackrel{\text{def}}{=} \min(L[x], \min_y\{CC[y] \mid E(x,y)\}) \\
&= \min(L[x], \min_y\{\min_{y'}\{L[y'] \mid TC(y,y')\} \mid E(x,y)\}) \\
&= \min(L[x], \min_{y',y}\{L[y'] \mid E(x,y) \wedge TC(y,y')\})
\end{aligned}
$$

Figure 7.2: Computing CC_1 and CC_2 from Example 7.1.

y', y to y, z respectively.

Example 7.2 (PreM Property)**.** Zaniolo et al. [ZYD+17] define the *Pre-mappability* rule (PreM), and prove that, under this rule, one Datalog program with ICO F is equivalent to another program with a simpler ICO. The PreM property is a restricted form of the FGH-rule, more precisely it asserts that the identity $G(F(X)) = G(F(G(X)))$ holds. In this case one can simply define H as $H(X) = G(F(X))$, and the FGH-rule holds. The PreM rule is more restricted than the FGH-rule, in two ways. First, the types of the IDBs of the F-program and the H-program must be the same. Second, the new query H is uniquely defined, namely $H \stackrel{\text{def}}{=} G \circ F$. While this simplifies the optimizer significantly, it also limits the type of optimizations that are possible under PreM.

Example 7.3 (Simple Magic)**.** The simplest application of magic set optimization [BMSU86, MP94, MFPR90] converts *transitive closure* to *reachability*. More precisely, it rewrites this program:

$$
\begin{aligned}
\Pi_1: \quad & TC(x,y) \text{ :- } [x=y] \vee \exists z(TC(x,z) \wedge E(z,y)) \\
& Q(y) \text{ :- } TC(a,y)
\end{aligned} \tag{7.5}
$$

$$F(TC) \stackrel{\text{def}}{=} TC' \text{ where } TC'(x,y) \stackrel{\text{def}}{=} [x = y] \vee \exists z(TC(x,z) \wedge E(z,y))$$
$$G(TC) \stackrel{\text{def}}{=} Q \quad \text{where} \quad Q(y) \stackrel{\text{def}}{=} TC(a,y)$$
$$H(Q) \stackrel{\text{def}}{=} Q' \quad \text{where} \quad Q'(y) \stackrel{\text{def}}{=} [y = a] \vee \exists z(Q(z) \wedge E(z,y))$$

Figure 7.3: Expressions F, G, H in Example 7.3.

where a is some constant, into this program:

$$\Pi_2 : \qquad Q(y) \text{ :- } [y = a] \vee \exists z(Q(z) \wedge E(z,y)) \tag{7.6}$$

This is a powerful optimization, because it reduces the run time from $O(n^2)$ to $O(n)$. Several Datalog systems support some form of magic set optimizations. We check that (7.5) is equivalent to (7.6) by verifying the FGH-rule. The functions F, G, H are shown in Fig. 7.3. One can verify that $G(F(TC)) = H(G(TC))$, for any relation TC. Indeed, after converting both expressions to normal form, we obtain $G(F(TC)) = H(G(TC)) = P$, where:

$$P(y) \stackrel{\text{def}}{=} [y = a] \vee \exists z(TC(a,z) \wedge E(z,y))$$

We prove in [WKN+22] that, given a sideways information passing strategy (SIPS) [BR91] every magic set optimization [BPRM91] over a Datalog program can be proven correct using a sequence of applications of the FGH-rule.

Example 7.4 (Generalized Semi-Naive Evaluation)**.** The naïve evaluation algorithm for (positive) Datalog re-discovers each fact from step t again at steps $t+1, t+2, \ldots$ The *semi-naive algorithm* aims at avoiding this, by computing only the new facts. We *generalize* the semi-naive evaluation from the Boolean semiring to any ordered pre-semiring $\boldsymbol{S}$, and prove it correct using the FGH-rule.

We require $\boldsymbol{S}$ to be a complete distributive lattice and $\oplus$ to be idempotent, and define the "minus" operation as: $b \ominus a \stackrel{\text{def}}{=} \bigwedge\{c \mid b \preceq a \oplus c\}$, then prove using the FGH-rule the following programs equivalent:

Π_1 :	Π_2 :	
$X_0 := \emptyset;$	$Y_0 := \emptyset;$	$\Delta_0 := F(\emptyset) \ominus \emptyset;\ (= F(\emptyset))$
`loop` $X_t := F(X_{t-1});$	`loop`	$Y_t := Y_{t-1} \oplus \Delta_{t-1};$
		$\Delta_t := F(Y_t) \ominus Y_t;$

To prove their equivalence, we define $G(X) \stackrel{\text{def}}{=} (X, F(X) \ominus X)$, $H(X, \Delta) \stackrel{\text{def}}{=} (X \oplus \Delta, F(X \oplus \Delta) \ominus (X \oplus \Delta))$, and then we prove that $G(F(X)) = H(G(X))$ by exploiting the fact that $\boldsymbol{S}$ is a complete distributive lattice. In practice, we compute the difference $\Delta_t = F(Y_t) \ominus Y_t = F(Y_{t-1} \oplus \Delta_{t-1}) \ominus F(Y_{t-1})$ using an efficient differential rule that computes $\delta F(Y_{t-1}, \Delta_{t-1}) = F(Y_{t-1} \oplus \Delta_{t-1}) \ominus F(Y_{t-1})$, where δF is an *incremental update* query for F, i.e., it satisfies the identity $F(Y) \oplus \delta F(Y, \Delta) = F(Y \oplus \Delta)$.

Thus, semi-naive query evaluation generalizes from standard Datalog over the Booleans to Datalog$^\circ$ over any complete distributive lattice with idempotent $\oplus$, and, moreover, is a special case of the FGH-rule. However, the semi-naive program (more precisely, function H) is no longer monotone, while our synthesizer (described in Sec. 7.4) is currently restricted to infer monotone functions H. For that reason we do not synthesize the semi-naive algorithm; instead we apply it using pattern-matching as the last optimization step.

7.1.2 Loop Invariants

More advanced uses of the FGH-rule require a loop-invariant, $\phi(X)$. By refining Theorem 7.1 with a loop invariant we obtain the following corollary:

Corollary 7.2. Let $\phi(X)$ be any predicate satisfying the following three conditions:

$$\phi(X_0) \tag{7.7}$$

$$\phi(X) \Rightarrow \phi(F(X)) \tag{7.8}$$

$$\phi(X) \Rightarrow (G(F(X)) = H(G(X))) \tag{7.9}$$

then the FG-program (7.1) is equivalent to the GH-program (7.2).

To prove the corollary, we consider the restriction of the function F to values X that satisfy ϕ. Conditions (7.7) and (7.8) state that ϕ is a loop invariant for the FG-program (7.1), while condition (7.9) is the FGH-rule applied to the restriction of F to ϕ.

Example 7.5 (Beyond Magic)**.** By using loop-invariants, we can perform optimizations that are more powerful than standard magic set rewritings. For a simple illustration, consider the following program:

$$\Pi_1: \qquad \begin{array}{c} TC(x,y) \text{ :- } [x=y] \vee \exists z(E(x,z) \wedge TC(z,y)) \\ Q(y) \text{ :- } TC(a,y) \end{array} \tag{7.10}$$

which we want to optimize to:

$$\Pi_2: \qquad Q(y) \text{ :- } [y=a] \vee \exists z(Q(z) \wedge E(z,y)) \tag{7.11}$$

Unlike the simple magic program in Example 7.3, here rule (7.10) is right-recursive. As shown in [BR91], the magic set optimization using the standard sideways information passing optimization [AHV95] yields a program that is more complicated than our program (7.11). Indeed, consider a graph that is simply a directed path $a_0 \rightarrow a_1 \rightarrow \cdots \rightarrow a_n$ with $a = a_0$. Then, even with magic set

optimization, the right-recursive rule (7.10) needs to derive *quadratically many* facts of the form $T(a_i, a_j)$ for $i \leq j$, whereas the optimized program (7.11) can be evaluated in linear time. Note also that the FGH-rule cannot be applied directly to prove that the program (7.10) is equivalent to (7.11). To see this, denote by $P_1 \stackrel{\text{def}}{=} G(F(TC))$ and $P_2 \stackrel{\text{def}}{=} H(G(TC))$, and observe that P_1, P_2 are defined as:

$$P_1(y) \stackrel{\text{def}}{=} [y = a] \vee \exists z(E(a, z) \wedge TC(z, y))$$
$$P_2(y) \stackrel{\text{def}}{=} [y = a] \vee \exists z(TC(a, z) \wedge E(z, y))$$

In general, $P_1 \neq P_2$. The problem is that the FGH-rule requires that $G(F(TC)) = H(G(TC))$ for *every* input TC, not just the transitive closure of E. However, the FGH-rule *does* hold if we restrict TC to relations that satisfy the following loop-invariant $\phi(TC)$:

$$\exists z_1(E(x, z_1) \wedge TC(z_1, y)) \Leftrightarrow \exists z_2(TC(x, z_2) \wedge E(z_2, y)) \tag{7.12}$$

If TC satisfies this predicate, then it follows immediately that $P_1 = P_2$, allowing us to optimize the program (7.10) to (7.11). It remains to prove that ϕ is indeed an invariant for the function F. The base case (7.7) holds because both sides of (7.12) are empty when $TC = \emptyset$. It remains to check $\phi(TC) \Rightarrow \phi(F(TC))$. Let us denote $TC' \stackrel{\text{def}}{=} F(TC)$, then we need to check that, if (7.12) holds, then the predicate $\Psi_1(x, y) \stackrel{\text{def}}{=} \exists z_1(E(x, z_1) \wedge TC'(z_1, y))$ is equivalent to the predicate $\Psi_2(x, y) \stackrel{\text{def}}{=} \exists z_2(TC'(x, z_2) \wedge E(z_2, y))$. We expand both predicates in Fig. 7.4, where we renamed z to z_2 in the last line of Ψ_1, and renamed z to z_1 in Ψ_2. Their equivalence follows from the assumption (7.12).

$$
\begin{aligned}
\Psi_1(x,y) \equiv & \exists z_1\,(E(x,z_1) \wedge ([z_1 = y] \vee \exists z(E(z_1,z) \wedge TC(z,y))))\\
\equiv & \exists z_1\,(E(x,z_1) \wedge [z_1 = y] \vee E(x,z_1) \wedge \exists z(E(z_1,z) \wedge TC(z,y)))\\
\equiv & E(x,y) \vee \exists z_1(E(x,z_1) \wedge \exists z(E(z_1,z) \wedge TC(z,y)))\\
\equiv & E(x,y) \vee \exists z_1(E(x,z_1) \wedge \exists z_2(E(z_1,z_2) \wedge TC(z_2,y)))\\
\Psi_2(x,y) \equiv & \exists z_2\,(([x = z_2] \vee \exists z(E(x,z) \wedge TC(z,z_2))) \wedge E(z_2,y))\\
\equiv & E(x,y) \vee \exists z, z_2(E(x,z) \wedge TC(z,z_2) \wedge E(z_2,y))\\
\equiv & E(x,y) \vee \exists z(E(x,z) \wedge \exists z_2(TC(z,z_2) \wedge E(z_2,y)))\\
\equiv & E(x,y) \vee \exists z_1(E(x,z_1) \wedge \exists z_2(TC(z_1,z_2) \wedge E(z_2,y)))
\end{aligned}
$$

Figure 7.4: Predicates Ψ_1 and Ψ_2 from Example 7.5.

7.1.3 Semantic Optimization Under Constraints

Semantic optimization refers to optimization rules that hold when the database satisfies certain constraints [RS94]. For example, most database systems today can optimize key/foreign-key joins by simply removing the join when the table containing the key is not used anywhere else in the query.

A priori knowledge on the structure of the underlying data may often provide additional potential for optimization. For instance, in [BMSU86], the counting and reverse counting methods are presented to further optimize the same-generation program if it is known that the underlying graph is acyclic. We present a principled way of exploiting such a priori knowledge. As we show here, recursive queries have the potential to use *global* constraints on the data during semantic optimization; for example, the query optimizer may exploit the fact that the graph is a tree, or the graph is connected.

Let Γ denote a set of constraints on the EDBs. Then, the FGH-rule (7.3) needs to be be checked only for EDBs that satisfy Γ. We illustrate this with an example:

Example 7.6 (Semantic Optimization)**.** Consider a hierarchy of subparts consisting of two re-

lations: $\texttt{SubPart}(x, y)$ indicates that y is a subpart of x, and $\texttt{Cost}[x] \in \mathbb{N}$ represents the cost of the part x. We want to compute, for each x, the total cost $Q[x]$ of all its subparts, sub-subparts, etc. Since the hierarchy can, in general, be a DAG, we first need to compute the transitive closure, before summing up the costs of all subparts, sub-subparts, etc:

$$\Pi_1 : \qquad \begin{aligned} S(x, y) &\text{ :- } [x = y] \vee \exists z\, (S(x, z) \wedge \texttt{SubPart}(z, y)) \\ Q[x] &\text{ :- } \sum_y \{\texttt{Cost}[y] \mid S(x, y)\} \end{aligned} \tag{7.13}$$

The first rule, defining the S predicate, is over the $\mathbb{B}$ semiring, while the second rule, defining Q, is over the $\mathbb{N}_\perp$ semiring. Consider now the case when our subpart hierarchy is a tree. Then, we can compute the total cost much more efficiently, using the following program:

$$\Pi_2 : \qquad Q[x] \text{ :- } \texttt{Cost}[x] + \sum_z \{Q[z] \mid \texttt{SubPart}(x, z)\} \tag{7.14}$$

Optimizing the program (7.13) to (7.14) is an instance of *semantic optimization*, since this only holds if the database instance is a tree. We do this in three steps. We define the constraint Γ stating that the data is a tree; using Γ we infer a loop-invariant Φ of the program Π_1; using Γ and Φ we prove the FGH-rule, concluding that Π_1 is equivalent to Π_2.

The constraint Γ is the conjunction of the following statements:

$$\forall x_1, x_2, y\, (\texttt{SubPart}(x_1, y) \wedge \texttt{SubPart}(x_2, y) \Rightarrow x_1 = x_2) \tag{7.15}$$

$$\forall x, y\, (\texttt{SubPart}(x, y) \Rightarrow T(x, y)) \tag{7.16}$$

$$\forall x, y, z\, (T(x, z) \wedge \texttt{SubPart}(z, y) \Rightarrow T(x, y)) \tag{7.17}$$

$$\forall x, y\, (T(x, y) \Rightarrow x \neq y) \tag{7.18}$$

The first asserts that y is a key in $\texttt{SubPart}(x, y)$. The last three are an Existential Second Order Logic (ESO) statement: they assert that there exists some relation $T(x, y)$ that contains SubPart, is transitively closed, and irreflexive. Next, we infer the following loop-invariant of the program Π_1:

$$\Phi : S(x, y) \Rightarrow [x = y] \vee T(x, y) \tag{7.19}$$

Finally, we check the FGH-rule, under the assumptions Γ, Φ. Denote by $P_1 \stackrel{\text{def}}{=} G(F(S))$ and $P_2 \stackrel{\text{def}}{=} H(G(S))$. To prove $P_1 = P_2$ we simplify P_1 using the assumptions Γ, Φ, as shown in Fig. 7.5. We explain each step. Line 2-3 are inclusion/exclusion. Line 4 uses the fact that the term on line 3 is = 0, because the loop invariant implies:

$$\begin{aligned}
S(x, z) \wedge \texttt{SubPart}(z, y) &\Rightarrow ([x = z] \vee T(x, z)) \wedge \texttt{SubPart}(z, y) && \text{by (7.19)} \\
&\equiv \texttt{SubPart}(x, y) \vee (T(x, z) \wedge \texttt{SubPart}(z, y)) \\
&\Rightarrow T(x, y) \vee T(x, y) && \text{by (7.17)} \\
&\equiv T(x, y) \\
&\Rightarrow x \neq y && \text{by (7.18)}
\end{aligned}$$

Line 5 follows from the fact that y is a key in $\texttt{SubPart}(z, y)$. A direct calculation of $P_2 = H(G(S))$ results in the same expression as line 5 of Fig. 7.5, proving that $P_1 = P_2$.

7.2 Architecture of FGH-Optimization

In the rest of this chapter we describe our synthesis-based FGH-optimizer, whose architecture is shown in Fig. 7.6. We optimize one stratum at a time. We denote by Π_1 one stratum of the

$$
\begin{aligned}
P_1[x] = &\sum_y \{\texttt{Cost}[y] \mid [x = y] \vee \exists z\, (S(x,z) \wedge \texttt{SubPart}(z,y))\} \\
=&\texttt{Cost}[x] + \sum_y \{\texttt{Cost}[y] \mid \exists z\, (S(x,z) \wedge \texttt{SubPart}(z,y))\} \\
&- \sum_y \{\texttt{Cost}[y] \mid [x = y] \wedge \exists z\, (S(x,z) \wedge \texttt{SubPart}(z,y))\} \\
=&\texttt{Cost}[x] + \sum_y \{\texttt{Cost}[y] \mid \exists z\, (S(x,z) \wedge \texttt{SubPart}(z,y))\} \\
=&\texttt{Cost}[x] + \sum_y \sum_z \{\texttt{Cost}[y] \mid (S(x,z) \wedge \texttt{SubPart}(z,y))\}
\end{aligned}
$$

Figure 7.5: Transformation of $P_1 \stackrel{\text{def}}{=} G(F(S))$ in Example 7.6.

input program, denote by X its recursive IDBs, by Y its output IDBs, and by F, G the ICO and the output operator respectively; see Eq. (7.4). The optimizer also takes as input a database constraint, Γ. The optimizer starts by inferring the loop invariant Φ; this is discussed in Sec. 7.5. Next, the optimizer needs to find H such that $\Gamma \wedge \Phi \models (G(F(X)) = H(G(X)))$. To reduce clutter we will often abbreviate this to $\Gamma \models (G(F(X)) = H(G(X)))$, assuming that Γ incorporates Φ. The optimizer makes two attempts at synthesizing H: it first tries using a simpler rule-based synthesizer, and, if that fails, then it tries the state-of-the-art Counterexample-Guided Inductive Synthesis (CEGIS). This is described in Sec. 7.4. Finally, H (or the original program if the FGH-optimization failed) is further transformed using generalized semi-naive optimization, as we already described in Sec. 7.1.1. Notice that stratification ensures that no interpreted functions are applied to the IDBs X; they can still be applied to the EDBs, or occur in predicates.

The FGH-optimization is an instance of *query rewriting using views* [Hal01, GL01]. Denoting by $Q \stackrel{\text{def}}{=} G(F(X))$ and $V \stackrel{\text{def}}{=} G(X)$, one has to rewrite the query Q using the view(s) V, in other words $Q = H(V)$. This is a *total* rewriting, in the sense that H may no longer refer to the IDBs X. This problem is NP-complete for UCQs with set semantics [LMSS95], in NP for UCQs with bag

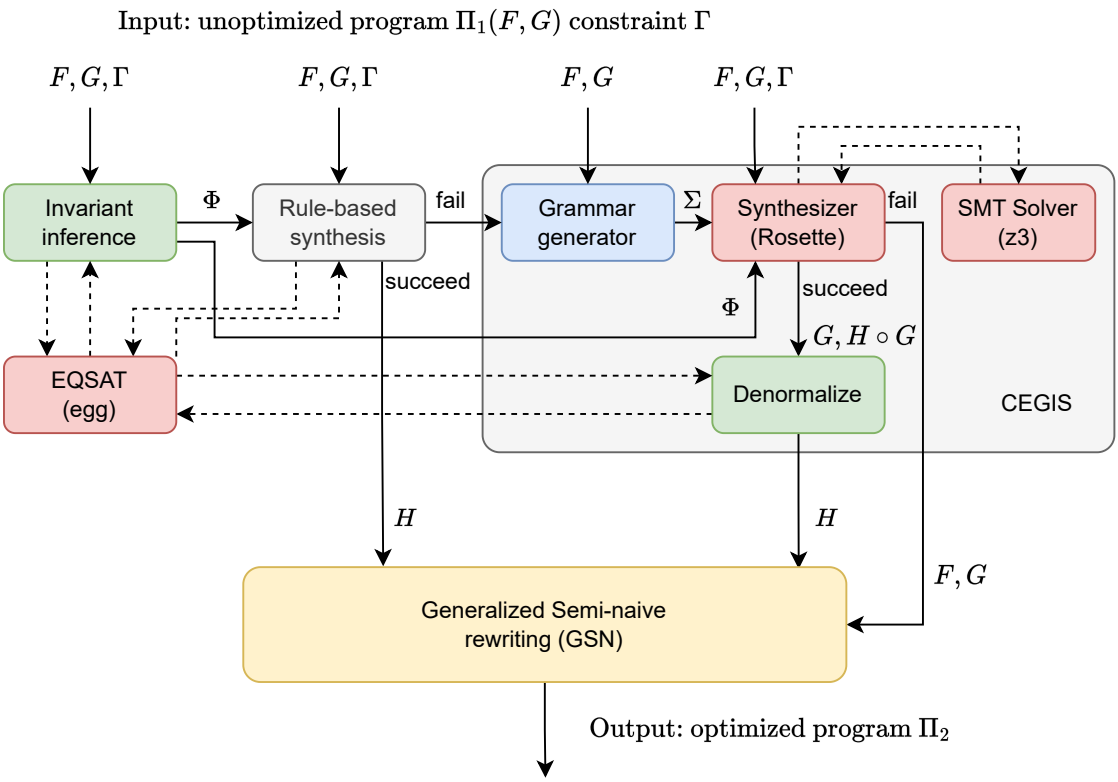


Figure 7.6: The architecture of the FGH-optimizer. The input is the unoptimized program Π_1, consisting of the functions F, G and the database constraint Γ. The output consists of the optimized program Π_2, see Eq. (7.4). Blue boxes are described in Section 7.4 and the green boxes in Section 7.5. The yellow box (generalized semi-naive optimization) is described in Section 7.1.1. The red boxes represent three state-of-the-art systems: Rosette is a CEGIS system [STB+06, TJ07, TB13], z3 is an SMT solver [dMB08], and EGG is an EQSAT system [WNW+21].

semantics[2], and undecidable for realistic SQL queries that include aggregates and arithmetic [GL01]. Systems that support query rewriting using views are rule-based, and apply a set of hand crafted, predefined patterns; our first attempt to synthesize H is also rule-based. Such synthesizers usually cannot take advantage of database constraints, but we will show in Sec. 7.5 how to exploit the constraint Γ in the rule-based synthesizer. However, rule-based rewriting explores a limited space, which is insufficient for many FGH-optimizations. In a seminal paper [STB+06] Solar-Lezama proposed an alternative to rule-based transformation, called *Counterexample-Guided Inductive Synthesis*, CEGIS: the synthesizer produces potentially incorrect candidates, and an SMT solver verifies their correctness. In the FGH-optimizer we use a program synthesizer, Rosette [TB13], to synthesize H.

At a conceptual level, program synthesis has two abstract steps: *generate* H, and *verify* $G(F(X)) = H(G(X))$. While the verifier is not used explicitly, it is used implicitly in the synthesizer, and we describe it in Sec. 7.3. Then we describe the synthesizer in Sec. 7.4.

7.3 Verification

We introduced the FGH-rule in Sec. 7.1 and showed several examples. In order to apply the rule, one needs to check the identity (7.3), $F(G(X)) = G(H(X))$. In this section we describe how we verify this identity. This step is implicit in both boxes *Rule-based Synthesis* and CEGIS in Fig 7.6. The identity can be checked in one of two ways: by applying a predefined set of identity rules (as currently done by most query optimizers), or by using an SMT solver.

[2]This follows from the fact that, under bag semantics, two UCQ queries are equivalent iff they are isomorphic. [Gre09, WHS+20].

7.3.1 Rule-based Test

Let $P_1 = G(F(X))$, $P_2 = H(G(X))$. To check $P_1 = P_2$, the *rule-based test* first normalizes both expressions into a sum-sum-product expression (Eq. (4.7)) via the semiring axioms, then checks if the expressions are isomorphic: if yes, then $P_1 = P_2$, otherwise we assume $P_1 \neq P_2$. The treatment of a constraint Γ will be discussed in Sec. 7.5. This test can be visualized as follows:

$$P_1 \xrightarrow{\text{axioms}} \text{normalize}(P_1) \simeq \text{normalize}(P_2) \xleftarrow{\text{axioms}} P_2 \tag{7.20}$$

where $\simeq$ denotes isomorphism. The Rule-based test is sound. When both P_1, P_2 are over the $\mathbb{N}^\infty$ semiring and have no interpreted functions then it is also complete [Gre09, WHS+20]. This simple test motivates the need for a complete set of axioms that allows any semiring expression to be normalized. The axioms include standard semiring axioms, and axioms about summations and free variables `fv`. For example, in order to prove $CC_1 = CC_2$ in Example 7.1 (with semiring notation in Figure 7.7) one needs all three axioms below:

$$\bigoplus_x \bigoplus_y (\cdots) = \bigoplus_{x,y} (\cdots) \tag{7.21}$$

$$A \otimes \bigoplus_x B = \bigoplus_x A \otimes B \text{ when } x \notin \texttt{fv}(A) \tag{7.22}$$

$$\bigoplus_x (A(x) \otimes [x = y]) = A(y) \tag{7.23}$$

7.3.2 SMT Test

When the expressions P_1, P_2 are over a semiring other than $\mathbb{N}^\infty$, or they contain interpreted functions, then the rule-based test is insufficient and we use an SMT solver for our verifier. We still normalize the expressions using our axioms, because today's solvers cannot reason about

$$CC_1[x] = \bigoplus_y L[y] \otimes \left([x = y]^0_\infty \oplus \bigoplus_z [E(x,z)]^0_\infty \otimes [TC(z,y)]^0_\infty\right)$$
$$CC_2[x] = L[x] \oplus \bigoplus_y \left(\bigoplus_{y'} L[y'] \otimes [TC(y,y')]^0_\infty\right) \otimes [E(x,y)]^0_\infty$$

Figure 7.7: CC_1 and CC_2 in semiring notation; their normal forms are isomorphic.

bound/free variables (as needed in axioms (7.21)-(7.23)). The *SMT test* is captured by the following figure:

$$P_1 \xrightarrow{\text{axioms}} \text{normalize}(P_1) \xleftrightarrow{\text{SMT}} \text{normalize}(P_2) \xleftarrow{\text{axioms}} P_2 \tag{7.24}$$

Example 7.7 (APSP100)**.** Consider a labeled graph E where $E[x, y]$ represents the cost of the edge x, y. The following query over Trop computes the all-pairs shortest path up to length of 100:

$$\begin{aligned} D[x,y] &\text{ :- } \texttt{if } x = y \texttt{ then 0 else } \min_z (D[x,z] + E[z,y]) \\ Q[x,y] &\text{ :- } \min(D[x,y], 100) \end{aligned} \tag{7.25}$$

The program is inefficient because it first computes the full path length, only to cap it later to 100. By using the FGH-rule we get:

$$Q[x,y] \text{ :- } \texttt{if } x = y \texttt{ then 0 else } \min\left(\min_z \left(Q[x,z] + E[z,y]\right), 100\right) \tag{7.26}$$

We show how to verify that (7.26) is equivalent to (7.25). Denote by $P_1 \stackrel{\text{def}}{=} G(F(D))$ and $P_2 \stackrel{\text{def}}{=} H(G(D))$ (where F, G, H are the obvious functions in the two programs defining Q). After we

de-sugar, convert to semiring expressions, and normalize, they become:

$$P_1[x,y] = \left(0 \otimes [x=y]_\infty^0\right) \oplus \left(\bigoplus_z D[x,z] \otimes E[z,y]\right) \oplus 100$$

$$P_2[x,y] = \left(0 \otimes [x=y]_\infty^0\right) \oplus \left(\bigoplus_z D[x,z] \otimes E[z,y]\right) \oplus \left(100 \otimes \bigoplus_z E[z,y]\right) \oplus 100$$

In the normalized expressions we push the summations past the joins, i.e., we apply rule (7.22) from right to left, thus we write $100 \otimes \bigoplus(\cdots)$ instead of $\bigoplus(100 \otimes \cdots)$: we give the rationale below. At this point, the normalized P_1 and P_2 are not isomorphic, yet they are equivalent if they are interpreted in Trop. We explain below in detail how the solver can check that. In this particular semiring, the identity $100 = \left(100 \otimes \bigoplus_z E[z,y]\right) \oplus 100$ holds since it becomes $100 = \min(100 + \min_z E[z,y], 100)$ with $E[z,y] \geq 0$, once we replace the uninterpreted operators $\oplus, \otimes$ with $\min, +$.

Implementation We describe how we implemented the SMT test $\Gamma \models P_1 = P_2$ using a solver, now also taking the database constraint Γ into account, where P_1, P_2 are the expressions $G \circ F$ and $H \circ G$. We used the `z3` solver [dMB08], but our discussion applies to other solvers as well. We need to normalize P_1, P_2 before using the solver, because solvers require all axioms to be expressed in First Order Logic. They cannot encode the axioms (7.21)-(7.23), because they are referring to free variables, which is a meta-logical condition not expressible in First Order Logic. Once normalized, we encode the equality as a first-order logic formula, and assert its negation, asking the solver to check if $\Gamma \wedge (P_1 \neq P_2)$ is satisfiable. The solver returns UNSAT, a counterexample, or UNKNOWN. UNSAT means the identity holds. When it returns a counterexample, then the identity fails, and the counterexample is given as input to the synthesizer (Sec. 7.4). UNKNOWN means that it could neither prove nor disprove the equivalence and we assume $P_1 \neq P_2$. For the theory of reals with

$+, *$, despite its decidability, `z3` often timed out in our experiments. We therefore used the theory of integers, and `z3` never timed out or returned UNKNOWN in our experiments.

We encode every $\boldsymbol{S}$-relation $R(x_1, \ldots, x_n)$ as an uninterpreted function $R : \mathbb{N} \times \cdots \times \mathbb{N} \to \boldsymbol{S}$, where $\boldsymbol{S}$ is the *interpreted* semiring, i.e., $\mathbb{B}$, Trop, $\mathbb{N}^\infty$, etc. We represent natural numbers as integers with nonnegativity assertions, and represent the sets $\mathbb{N}^\infty, \mathbb{N}_\perp, \mathbb{R}_\perp$ as union types. Operators supported by the solver, like $+, *, \min, -$, are entered unchanged; we treat other operators as uninterpreted functions. Unbounded aggregation, like $\bigoplus_x e(x)$, poses a challenge: there is no such operation in any SMT theory. Here we use the fact that P_1 and P_2 are normalized sum-sum-product expressions:

$$P_1 = \left(\bigoplus_{x_1} e_1\right) \oplus \left(\bigoplus_{x_2} e_2\right) \oplus \cdots \qquad\qquad P_2 = \left(\bigoplus_{x_1'} e_1'\right) \oplus \left(\bigoplus_{x_2'} e_2'\right) \oplus \cdots$$

Assume first that each x_i is a single variable. We ensure that all the variables $x_1, x_2, \ldots$ in P_1 are distinct, by renaming them if necessary. Next, we replace each expression $\bigoplus_{x_i} e_i$ with $u(x_i, e_i)$ where u is an uninterpreted function. Finally, we ask the solver to check

$$\Gamma \models \left(u(x_1, e_1) \oplus u(x_2, e_2) \oplus \cdots = u(x_1', e_1') \oplus u(x_2', e_2') \oplus \cdots\right)$$

This procedure is sound, because if the identity $u(x, e) = u(x', e')$ holds, then $x = x'$ (they are the same variable) and $e = e'$, which means that $\bigoplus_x e = \bigoplus_{x'} e'$. Moreover, when synthesizing P_2, we will ensure that the generator includes the variables $x_1, x_2, \ldots$ present in P_1 to achieve a limited form of completeness, see Sec. 7.4. Finally, if a summation is over multiple variables, we simply nest the uninterpreted function, i.e., write $\bigoplus_{x,y} e$ as $u(x, u(y, e))$.

Example 7.8. We now finish Example 7.7. After introducing the uninterpreted functions described

above, we obtain:

$$P_1 = \min(0 + w(x,y), u(z, D[x,z] + E[z,y]), 100)$$
$$P_2 = \min(0 + w(x,y), u(z, D[x,z] + E[z,y]), 100 + u(z, E[z,y]), 100)$$

where $w(x,y)$ is an uninterpreted function representing $[x = y]_\infty^0$, and u is our uninterpreted function encoding summation. The solver proves that the two expressions are equal, given that $w \geq 0$ and $u \geq 0$. Notice that it was critical to factorize the term 100: had we not done that, then the expression $100 + u(z, E[z,y])$ would be $u(z, 100 + E[z,y])$ and the identity $P_1 = P_2$ no longer holds.

Discussion Readers unfamiliar with First Order Logic may be puzzled by our statement that the identity $u(x,e) = u(x',e')$ holds iff $x = x'$ and $e = e'$. In order to explain this, it helps to first review the basic definitions of validity and satisfiability in logic. A statement is "valid" if it is true for all interpretations of its uninterpreted symbols. For example, the equality $f(x) + y = y + f(x)$ is valid over integers, because it holds for all function f and all values of x and y. A statement is "satisfiable" if there exists interpretations of its uninterpreted symbols that make the statement true. A statement is valid iff its negation is not satisfiable. In our case, the statement $u(x,e) = u(x',e')$ is valid if the equality is true for all possible interpretations of u, x, x'. For example, suppose we asked the solver to check whether $u(x, 2(x+1)) = u(y, 2y+2)$ is valid. To answer this question, we negate the statement and ask the `z3` solver whether the negation is satisfiable: $u(x, 2(x+1)) \neq u(y, 2y+2)$. One can easily satisfy this with pen and paper, e.g., $x = 1, y = 2, u(a,b) = a + b$, then $u(x, 2(x+1)) = 5, u(y, 2y+2) = 8$. `z3` also answers "yes", and

provides the following example for the inequality[3]:

$$x = 0, y = 38, u(a, b) = \text{if } a = 38 \wedge b = 78 \text{ then } 6 \text{ else } 4$$

Therefore, the identity $u(x, 2(x + 1)) = u(y, 2y + 2)$ is not valid. In contrast, suppose we asked the solver whether $u(x, 2(x+1)) = u(x, 2x+2)$ is valid. Its negation is $u(x, 2(x+1)) \neq u(x, 2x+2)$, and `z3` returns UNSAT, which means that the identity is valid. In general, the identity $u(x, e) = u(x', e')$ is valid iff $x = x'$ and $e = e'$.

7.4 Synthesis

We have seen in Sec. 7.3 how to use an SMT solver to check the identity $G(F(X)) = H(G(X))$. We are now ready to discuss the core of the FGH-optimizer: given the query expressions F, G, find H such that the identity $G(F(X)) = H(G(X))$ holds; recall that we denote these expressions by P_1, P_2. As for verification, this can be done by using only rewriting, or using program synthesis with an SMT solver. We are also given a database constraint Γ, and we assume that we have already added to it the loop invariant Φ.

7.4.1 Rule-based Synthesis

The optimizer first attempts to synthesize H using rule-based rewriting. This process is akin to our initial verifier that relies only on normalization and isomorphism checking.

$$P_1 \xrightarrow{\text{axioms}} \text{normalize}(P_1) \xrightarrow{\text{axioms}} P_2 \tag{7.27}$$

[3]Please refer to the documentation of `z3` for how models for uninterpreted functions are constructed.

There is no obvious way to "denormalize" an expression, since many expressions share the same normal form. We used for this purpose an equality saturation system (EQSAT), also used for multiple tasks of the FGH-optimizer, see Fig 7.6. We describe EQSAT in Sec. 7.5.

7.4.2 Counterexample-based Synthesis

The rule-based synthesis (7.27) explores only correct rewritings P_2, but its space is limited by the hand-written axioms. The alternative approach, pioneered in the programming language community [STB+06], is to synthesize candidate programs P_2 from a much larger space, then using an SMT solver to verify their correctness. This technique, called Counterexample-Guided Inductive Synthesis, or CEGIS, can find rewritings P_2 even in the presence of interpreted functions, because it exploits the theory of the underlying domain. As a first attempt it can be described as follows (we will revise it below):

$$P_1 \xrightarrow{\text{axioms}} \text{normalize}(P_1) \xrightarrow{\text{CEGIS}} P_2 \tag{7.28}$$

Brief Overview of CEGIS

We give a brief overview of the CEGIS system, Rosette [TJ07, TB13], that we used in our optimizer. Understanding its working is important in order to optimize its usage for FGH-optimization. The input to Rosette consists of a *specification* and a *grammar*, and the goal is to synthesize a program defined by the grammar and that satisfies the specification. The main loop is implemented with a pair of *dueling* SMT-solvers, the *generator* and the *checker*. In our setting, the inputs are the query P_1, the database constraint Γ, and a small grammar Σ (described below). The specification is $\Gamma \models (P_1 = P_2)$, where P_2 is defined by the grammar Σ. The generator generates syntactically correct programs P_2, and the verifier checks $\Gamma \models (P_1 = P_2)$. In the most naive attempt, the

generator could blindly generate candidates $P_2, P_2', P_2'', \ldots$, until one is found that the verifier accepts. This is hopelessly inefficient. The first optimization in CEGIS is that the verifier returns a small counterexample database instance D for each unsuccessful candidate P_2, i.e., $P_1(D) \neq P_2(D)$. When considering a new candidate P_2, the generator checks that $P_1(D_i) = P_2(D_i)$ holds for all previous counterexamples $D_1, D_2, \ldots$, by simply evaluating the queries P_1, P_2 on the small instance D_i. This significantly reduces the search space of the generator.

CEGIS applies a second optimization, where it uses the SMT solver itself to generate the next candidate P_2, as follows. It requires a fixed recursion depth for the grammar Σ; in other words we can assume w.l.o.g. that Σ is non-recursive. Then it associates a symbolic Boolean variable $b_1, b_2, \ldots$ to each choice of the grammar. The grammar Σ can be viewed now as a BDD (binary decision diagram) where each node is labeled by a choice variable b_j, and each leaf by a completely specified program P_2. The search space of the generator is now completely defined by the choice variables b_j, and Rosette uses the SMT solver to generate values for these Boolean variables such that the corresponding program P_2 satisfies $P_1(D_i) = P_2(D_i)$, for all counterexample instances D_i. This significantly speeds up the choice of the next candidate P_2.

Using Rosette

To use Rosette, we need to define the specification and the grammar. A first attempt is to simply define some grammar for H, with the specification $\Gamma \models (G(F(X)) = H(G(X)))$. This does not work, since Rosette uses the SMT solver to check the identity: as explained in Sec. 7.3.2, modern SMT solvers have limitations that require us to first normalize $G(F(X))$ and $H(G(X))$ before checking their equivalence. Even if we modify Rosette to normalize $H(G(X))$ during verification, there is still no obvious way to incorporate normalization into the program generator driven by the SMT solver. Instead, we define a grammar Σ for normalize$(H(G(X)))$ rather than for H, and

then specify:

$$\Gamma \models \text{normalize}(G(F(X))) = \text{normalize}(H(G(X)))$$

Then, we denormalize the result returned by Rosette, in order to extract H, using the *denormalization* module in Fig. 7.6, described in Sec. 7.5. In summary, our CEGIS-approach for FGH-optimization can be visualized as follows:

$$P_1 \xrightarrow{\text{axioms}} \text{normalize}(P_1) \xrightarrow{\text{CEGIS}} \text{normalize}(P_2) \xrightarrow{\text{axioms}} P_2 \tag{7.29}$$

The choice of the grammar Σ is critical for the FGH-optimizer. If it is too restricted, then the optimizer will be limited too, if it is too general, then the optimizer will take a prohibitive amount of time to explore the entire space. We briefly describe our design at a high level. Recall that X denotes multiple IDBs, and the query $G(X)$ may also return multiple intermediate relations. In our system $G(X)$ is restricted to return a single relation, so we will assume that $Y = G(X)$ is a single IDB. The expression G is known to us, and is a sum-sum-product expression, see Eq. (4.7),

$$G(X) = G_1(X) \oplus \cdots \oplus G_m(X)$$

where each $G_i(X)$ is a sum-product expression, using the IDBs X and/or the EDBs.

To generate `normalize`$(H(G(X)))$, we group its sum-products by the number of occurrences of Y:

$$\texttt{normalize}(H(Y)) = H^{(0)} \oplus H^{(1)}(Y) \oplus \cdots \oplus H^{(k_{\max})}(Y)$$

where $H^{(k)}$ is a sum-sum-product $H^{(k)} = Q_1 \oplus Q_2 \oplus \cdots$ s.t. each Q_i contains exactly k occurrences of Y, and an arbitrary number of EDBs (it may not contain the IDBs X). We choose $k_{\max}$ as the

largest number of recursive IDBs X that occur in any rule of the original program $F(X)$, e.g., if the original program was linear, then $k_{\max} \stackrel{\text{def}}{=} 1$. We obtain:

$$\texttt{normalize}(H(G(X))) =$$
$$H^{(0)} \oplus \texttt{normalize}(H^{(1)}(G(X))) \oplus \cdots \oplus \texttt{normalize}(H^{(k_{\max})}(G(X)))$$

The grammar Σ is shown in Fig. 7.8. The start symbol, A, generates a sum matching the expression above. A_0 generates $H^{(0)}$, which is a sum of sum-product terms without any occurrence of Y. Recall from Sec. 7.3.2 that the expression $u(z, Q)$ denotes $\bigoplus_z Q$. E is one of the EDBs, and Z is a non-terminal for which we define rules $Z \to z_1|z_2|\cdots|z_m|z'_1|z'_2\cdots$ where $z_1, \ldots, z_m$ are variables that already occur in $\texttt{normalize}(G(F(X)))$, and $z'_1, z'_2, \ldots$ is some fixed set of fresh variable names. A_k generates $\texttt{normalize}(H^{(k)}(G(X)))$, which is a sum of sum-products, each with exactly k occurrence of Y. As stated in Fig. 7.8, the rules for A_k are incorrect. For example consider A_1: the m non-terminals $A_{11}, \ldots, A_{1m}$ should have identical derivations, instead of being expanded independently. For example, assume $G = G_1 \oplus G_2$ (thus $m = 2$) and we want H to be one of $E_1 \otimes Y$ or $E_2 \otimes Y$ or $E_3 \otimes Y$. Then, $\texttt{normalize}(H(G(X)))$ can be one of the following three expressions $E_1 \otimes G_1 \oplus E_1 \otimes G_2$ or $E_2 \otimes G_1 \oplus E_2 \otimes G_2$ or $E_3 \otimes G_1 \oplus E_3 \otimes G_2$. However, the grammar $A_1 \to A_{11} \oplus A_{12}$ also generates incorrect expressions $E_1 \otimes G_1 \oplus E_2 \otimes G_2$, because A_{11}, A_{12} can choose independently the IDB E_1, E_2, or E_3. We fix this by exploiting the choice variables in Rosette: we simply use the same variables in $A_{11}, A_{12}, \ldots$ ensuring that all these non-terminals make exactly the same choices. We note that our current system is restricted to linear programs, hence $k_{\max} = 1$.

Discussion

Even though our grammar is restricted to $k_{\max} = 1$, it is more complex than Fig 7.8, in order to further reduce the search space. We use more non-terminals to better control which variables z

$$
\begin{aligned}
A &\to A_0 \oplus A_1 \oplus \cdots \oplus A_{k_{\max}},\\
A_0 &\to Q_0 \mid Q_0 \oplus A_0,\ \ Q_0 \to\ u(Z, Q_0) \mid Q_0 \otimes Q_0 \mid E(Z, Z, \cdots, Z),\\
A_1 &\to A_{11} \oplus \cdots \oplus A_{1m},\ \ A_2 \to\ A_{211} \oplus \cdots \oplus A_{2mm},\ \ A_3 \to A_{3111} \oplus \cdots\\
A_{1i} &\to Q_{1i} \mid Q_{1i} \oplus A_{1i},\ \ Q_{1i} \to u(Z, Q_{1i}) \mid Q_{1i} \otimes Q_0 \mid G_i(X), \qquad i = 1, m\\
A_{2ij} &\to Q_{2ij} \oplus A_{2ij}, \qquad Q_{2ij} \to u(Z, Q_{2ij}) \mid Q_{1i} \otimes G_j(X), \qquad i, j = 1, m\\
A_{3ij\ell} &\to Q_{3ij\ell} \oplus A_{3ij\ell}, \quad Q_{3ij\ell} \to u(Z, Q_{3ij\ell}) \mid Q_{2ij} \otimes G_\ell(X), \quad i, j, \ell = 1, m
\end{aligned}
$$

Figure 7.8: Grammar Σ for `normalize`$(H(G(X)))$, for $k_{\max} = 3$.

can be used where, and we also consider the choice of including entire subexpressions that occur in the original program P_1, since they are often reused in the optimized program. The synthesizer would require many trials to find them, had we not included them explicitly.

7.5 Equality Saturation

Throughout the FGH-optimizer we need to manipulate expressions, apply rules, and manage equivalent expressions. This problem is common to all query optimizers. Instead of implementing our own expression manager, we again leverage the equality saturation (EQSAT) system as we did in Chapter 6. Specifically, we used egg [WNW+21] to implement the green boxes in the architecture shown in Fig. 7.6.

We describe how we use EGG in the FGH-optimizer. First, we use it to extend the Rule-based test (Sec. 7.3.1) to account for a constraint Γ. By design, the e-graph makes it easy to infer the equivalence $P_1 = P_2$ from a set of rules. Suppose we want to check such an equivalence conditioned on Γ. We may assume w.l.o.g. that Γ is a logical implication, $\Delta \Rightarrow \Theta$ since all database constraints are expressed this way. We convert it into an equivalence $\Delta \wedge \Theta = \Delta$, and insert it into the e-graph, then check for equivalence $P_1 = P_2$.

Second, we use the e-graph to denormalize an expression. More precisely, recall from Sec. 7.4.1

Program	Type	Con.	Inv.	Dataset	#op
Beyond Magic (BM)	rule-based	No	Yes	twitter, epinions, wiki	6
Connected Components (CC)	rule-based	No	No	twitter, epinions, wiki	6
Single Source Shortest Path (SSSP)	rule-based	No	No	twitter, epinions, wiki	17
Sliding Window Sum (WS)	CEGIS	No	Yes	Vector of Numbers	15
Betweenness Centrality (BC)	CEGIS	No	No	Erdős–Rényi Graphs	43
Graph Radius (R)	CEGIS	Yes	Yes	Random Recursive Trees	12
Multi-level Marketing (MLM)	CEGIS	Yes	Yes	Random Recursive Trees	6

Figure 7.9: Experimental Setup. "Type" is the type of synthesis required to optimize the program. "Con." means whether the input database has constraints. "Inv." means whether optimizing the program requires loop invariants. "#op" is the number of operations in the program.

that we attempt to synthesize H by denormalizing $P_1 \stackrel{\text{def}}{=} \text{normalize}(F(G(X)))$, in other words, writing it in the form $H(G(X))$. For that we add $G(X)$ to the e-graph, observe in which e-class it is inserted, and replace that e-class with a new node Y. The root of the new e-graph represents many equivalent expressions, and each of them is a candidate for H. We choose the expression H that has the smallest AST *and* does not have any occurrence of the IDBs X.

Finally, we use the e-graph to infer the loop invariants. We do this by symbolically executing the recursive program F for up to 5 iterations, and compute the symbolic expressions of the IDBs X: $X_0, X_1, \ldots$ Using an e-graph we represent all identities satisfied by these (distinct!) expressions. The identities that are satisfied by every X_i are candidate loop invariants: for each of them we use the SMT solver to check if they satisfy Eq. (7.8) from Sec. 7.1.2.

7.6 Evaluation

We implemented a source-to-source FGH-optimizer, based on Fig. 7.6. The input is a program Π_1, given by F, G, and a database constraint Γ, and the output is an optimized program H. We evaluated it on three Datalog systems, and several programs from benchmarks proposed by prior research [SYI+16b, FZZ+19]; we also propose new benchmarks that perform standard data analysis

Program	BM	CC	SSSP	R	MLM	BC	WS
Invariance inference	0.092	0	0	0.129	0.132	0	0
Synthesis	0.004	0.005	0.004	0.284	0.299	1.2	0.821
Total	0.096	0.005	0.004	0.413	0.431	1.2	0.821
Opt. / Exec. (max%-min%)	.82 - .16	.04-.01	.24-.002	.41-.07	.76-.09	6.3-.51	7.4-.66

Program	R	MLM	BC	WS
Search space	10	20	132	94

Figure 7.10: Optimization time in seconds, optimization time over execution time, and size of the search space.

tasks. We did not modify any of the three Datalog engines. We asked two major questions:

1. How effective is our source-to-source optimization, given that each system already supports a range of optimizations?

2. How much time does the actual FGH optimization take?

7.6.1 Setup

There is a great number of commercial and open-source Datalog engines in the wild, but only a few support aggregates in recursion. We were able to identify five major systems with such support: SociaLite [SGL15], Myria [WBH15], the DeALS family of systems (DeALS [SYZ15], BigDatalog [SYI+16a], and RaDlog [GWM+19]), RecStep [FZZ+19], and Dyna [FLVE20]. Prior work [SYI+16b] reports SociaLite and Myria are consistently slower than newer systems, so we do not include them in our experiments. Dyna is designed to experiment with novel language semantics and not for data analytics, and we were not able to run our benchmarks without errors using it. Systems in the DeALS family are similar to each other; we pick BigDatalog because it is open source and runs our benchmarks without errors; we include RecStep for the same reasons. Both BigDatalog and RecStep are multi-core systems. Finally, we run experiments on an

unreleased commercial system X, which is single core. As we shall discuss, X is the only one that supports all features for our benchmarks.

We conducted all experiments on a server running CentOS 8.3.2011. The server has a total of 1008GB memory, and 4 Intel Xeon CPU E7-4890 v2 2.80GHz CPUs, each with 15 cores and 30 threads. We ran seven benchmarks, shown in Fig 7.9. BM and CC are Examples 7.5 and 7.1; MLM is basically Example 7.6. CC, SSSP and MLM are from [SYI+16b], the others are designed by us. R and MLM require a database constraint stating that the data is a tree. BM, R, and MLM each have a non-trivial loop invariant that is inferred by the optimizer. Our optimizer requires each program to consist of two rules, one each for F and G, and so a meaningful metric for program size is the number of semiring operations. These numbers are listed in the last column of Fig 7.9. Our benchmark programs are comparable in size to those used in prior work [SYI+16b, FZZ+19]. All programs are available in our git repository. The real-world datasets twitter [ML12], epinions [RAD03], and wiki [LHK10] are from the popular SNAP collection [LK14]. We follow the setting in [SYI+16b, FZZ+19] when generating the synthetic graphs. We additionally generate random recursive trees with an exponential decay, modeling the decay of association in multi-level marketing [EKTZ11]. For WS, we input the vector $[1, \ldots, n]$, since the values of the entries do not affect run time. In general, we used smaller datasets than [SYI+16b, FZZ+19] because some of our experiments run single-threaded.

7.6.2 Run Time Measurement

For each program-dataset pair, we measure the run times of three programs: original, with the FGH-optimization, and with the FGH-optimization and the generalized semi-naive (GSN, for short) transformation. We report only the speedups relative to the original program in Fig. 7.11 and 7.12. In some cases the original program timed out our preset limit of 3 hours, where we

report the speedup against the 3 hours mark. In some other cases the original program ran out of memory and we mark them with "o.o.m." in the figure. The *absolute* runtimes are irrelevant for our discussion, since we want to report the effect of *adding* our optimizations. (We also do not have permission to report the runtimes of X.) All three systems already perform semi-naive evaluation on the original program, since that is expressed over the Boolean semiring. But the FGH-optimized program is over a different semiring (except for BM), and GSN has non-stratifiable rules with negation, which are supported only by system X; we report GSN only for system X. While the benchmarks in Fig. 7.11 were on real datasets, those in Fig. 7.12 use synthetic data, for multiple reasons: we did not have access to a good tree dataset needed in the R and MLM benchmarks, BC timed out on our real data (BC is computationally expensive), and WS uses only a simple array. A benefit of synthetic data is that we can report how the optimizations scale with the data size. Unfortunately, the FGH-optimized programs in Fig. 7.12 require recursion with SUM aggregation, which is not supported by BigDatalog or RecStep; this is in contrast with those in Fig. 7.11, which require recursion with MIN aggregation which is supported by all systems.

Findings

Figure 7.11 shows the results of the first group of benchmarks optimized by the rule-based synthesizer. Overall, we observe our optimizer provides consistent and significant (up to 4 orders of magnitude) speedup across systems and datasets. Only a few datapoints indicate the optimization has little effect: BM and CC on wiki under BigDatalog, and SSSP on wiki under X. This is due to the small size of the wiki dataset: both the optimized and unoptimized programs finish very quickly, so the run time is dominated by system overhead which cannot be optimized away. We also note that (under X) GSN speeds up SSSP but slows down CC (note the log scale). The latter occurs because the Δ-relations for CC are very large, and as a result the semi-naive evaluation has the same complexity as the naive evaluation; but the semi-naive program is more complex

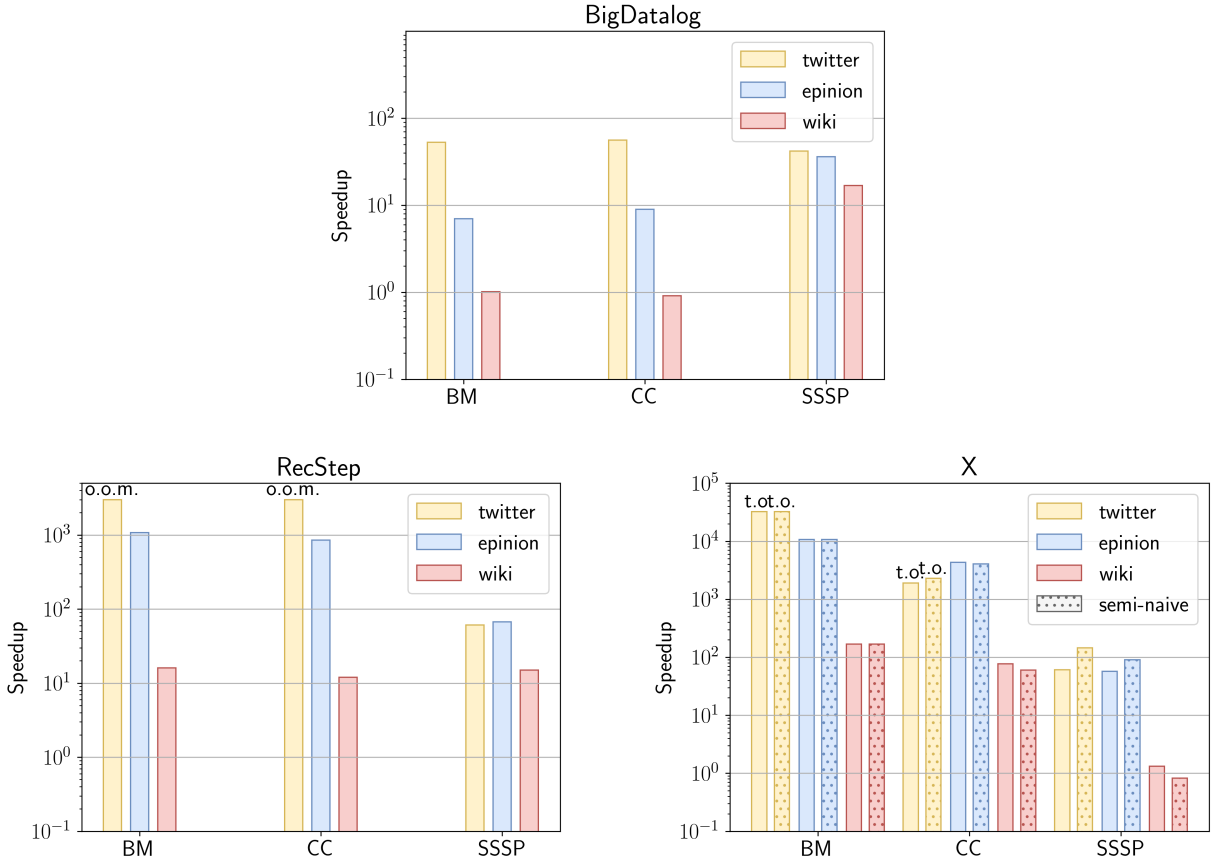


Figure 7.11: Speedup of the optimized v.s. original program; higher is better; t.o. means the original program timed out after 3 hours, in which case we report the speedup against 3 hours; o.o.m. means the original program ran out of memory.

and incurs a constant slowdown. GSN has no effect on BM because the program is in the boolean semiring, and X already implements the standard semi-naive evaluation. Optimizing BM with FGH on BigDatalog sees a significant speedup even though the systems already implements magic set rewrite, because the optimization depends on a loop invariant.[4] Overall, both the semi-naive and naive versions of the optimized program are significantly faster than the unoptimized program.

Figure 7.12 shows the results of the second group of benchmarks, which required CEGIS. Since we used synthetic data, we examined here the asymptotic behavior of the optimization as a function of the data size. The most advanced optimization was for BC, which leads essentially to Brandes' algorithm [Bra01]: its effect is dramatic. R and MLM rely on semantic optimization for a tree. We generated two synthetic trees, a random recursive tree with expected depth of $O(\log n)$ and one with exponential decay with expected depth of $O(n)$. Since the benefit of the optimization depends on the depth, we see a much better asymptotic behavior in the second case. Here, too, the optimizations were always improving the runtime.

7.6.3 Optimization Time and Search Space

CEGIS can quickly become very expensive if its search space is large, and, for that reason, we have designed the grammar generator carefully to reduce the search space without losing generality. Fig. 7.10 reports the runtime of the synthesizer (in seconds) for both rule-based synthesis and CEGIS, and the size of the search space. The rule-based synthesizer runs in milliseconds, while CEGIS took over 1s for BC (our hardest benchmark). These numbers are close to those demanded by modern query optimizers, and represent only a tiny portion of the total runtime of the optimized query. Optimization time takes less than 1% of the query run time for all benchmarks except for BC and WS on the smallest input data. To our surprise, our grammar managed to narrow the

[4] BigDatalog can optimize the left-recursive version of BM (7.5) to obtain similar speedup, via the classic magic set rewrite.

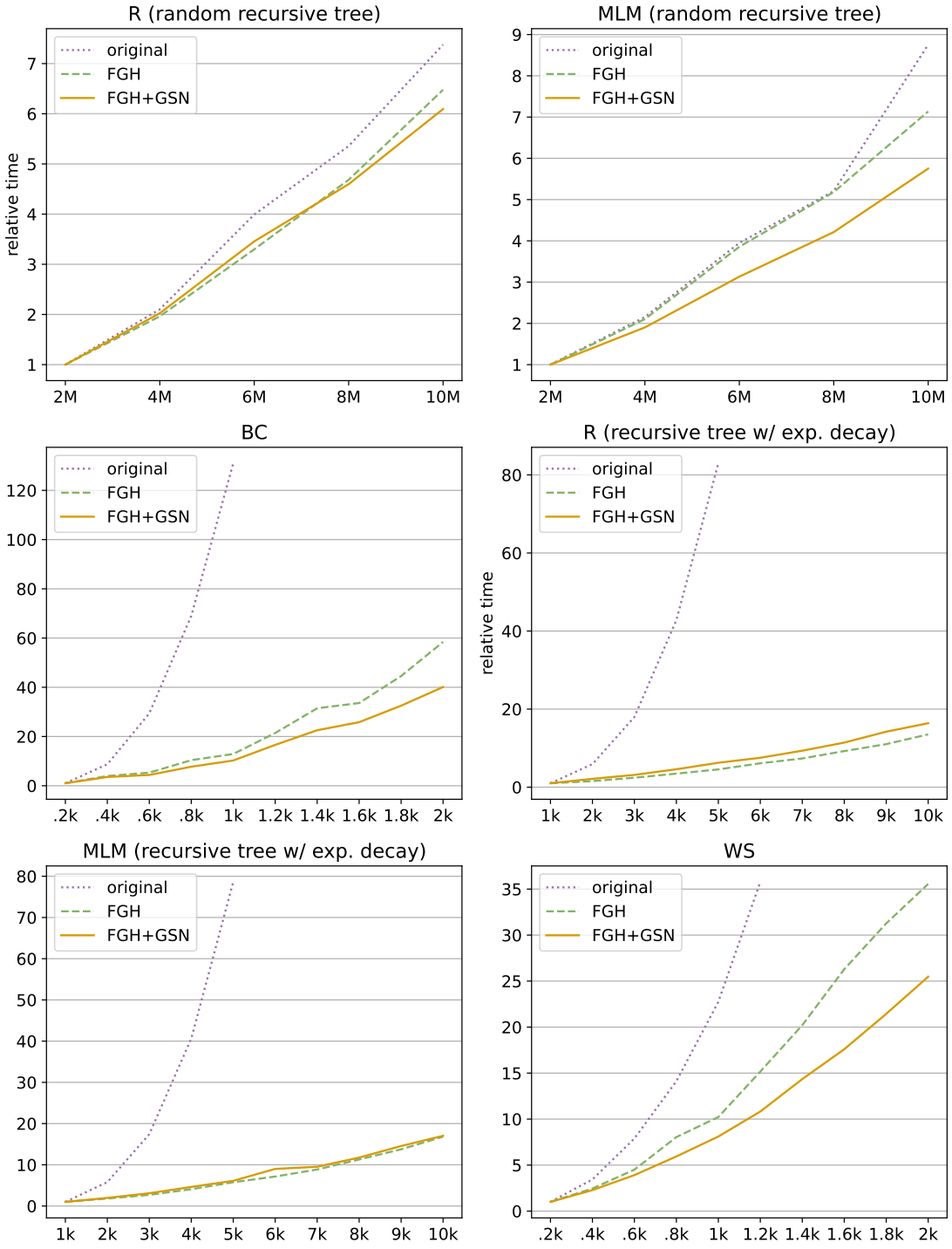


Figure 7.12: Runtime increase as a function of the data size; lower is better.

search space considerably, to no more than 132 candidates, which (in hindsight) explains the low optimization times. The search space can grow rapidly, and even exponentially, as the size of the input program grows. Our optimizer optimizes a single stratum at a time, focusing on improving critical "basic blocks" of a program. Our benchmark programs demonstrate a wide range of data analysis computation can be expressed succinctly using just a few semiring operations, and optimization can have a dramatic impact on performance.

7.6.4 Summary

We conclude that our optimizer can significantly speedup already optimized Datalog systems, either single-core or multi-core. GSN can, sometimes, further improve the runtime. We achieved this using a rather small search space, which led to fast optimization.

7.7 Summary and Discussion

We have presented a new optimization method for recursive queries, which generalizes many previous optimizations described in the literature. We implemented it using a CEGIS and an EQSAT system. Our experiments have shown that this optimization is beneficial, regardless of what other optimizations a Datalog system supports. We discuss here some limitations and future work.

Our current implementation is restricted to linear programs, but our techniques apply to nonlinear programs as well. Non-linear programs require a more complex grammar Σ; this is likely to increase the search space, and possibly increase the optimization time. We leave this exploration to future work.

Our current optimizer is heuristic-based, and future work needs to integrate it with a cost model. This, however, will be challenging, because very little work exists for estimating the cost of

recursive queries. this chapter applies a simple cost-model. We use the arity of the IDB predicate as a proxy for a simple asymptotic cost model, because N^{arity} is the size bound of the output, when N is the size of the active domain. This simple cost-model is currently used by the commercial DB system mentioned in this chapter. If the optimized program reduces the arity, then it is assessed to have lower cost.

Two limitations of our current implementation are the fact that we currently do not "invent" new IDBs for the optimized query, and do not apply the FGH-optimizer repeatedly. Both would be required to support more advanced magic set optimizations.

Our initial motivation for this work came from a real application, which consists of a few hundred Datalog rules that were computationally very expensive, and required a significant amount of manual optimizations. Upon close examination, at a very high level, the manual optimization that we performed could be described, abstractly, as a *sliding window* optimization (WS in Fig. 7.9), which is one of the simplest instantiations of the FGH-rule. Yet, our current system is far from able to optimize automatically programs with hundreds of rules: we leave that for future work.

Chapter 8

Conclusions and Future Outlook

In this dissertation we have enhanced the relational paradigm of data processing to support more complex data and programs. We first extended the traditional Datalog language with support for recursive aggregates, enabling the expression of a wide range of data processing tasks. We then described a new algorithm to speed up the relation join, a key operation in relational data processing. We presented techniques to optimize linear algebra programs through a relational lens, and finally showed how to optimize recursive programs over relational data. Our approach follows the core principle of the relational model: to allow the programmer to specify computation in a high-level, declarative language, and let the system optimize the program for efficient execution.

Nevertheless, to realize the full potential of the relational model in modern data processing, there are still many challenges to overcome. Namely, the relational model offers *data independence*, meaning the programmer needs only to write the program logic once, and the system will automatically handle optimization, parallelism, distribution, incremetal maintenance, persistence, fault tolerance, etc. Addressing each of these challenges in the context of modern data processing is a research topic of its own right. In the following, we point out a few promising directions that will bring us closer towards our vision of relational programming.

Scheduling Languages. Building a relational data system is hard. Half a century after the relational model was first proposed, we are still perfecting the arts and crafts of non-recursive simple queries over tabular data. We believe the relational future of programming is much more tangible, if we rethink the relationship between the data analyst and the data system. In a perfect world, the analyst need only specify the program logic in the highest possible level of abstraction, and the system will fill in all the details automatically. Building such a fully autonomous system is indeed a daunting task. The rise of *scheduling languages* like Halide [RBA+13] in domain-specific computing provides a more pragmatic alternative. The programmer may start with a simple, yet inefficient, program that is already executable (perhaps for prototyping purposes). Then, instead of relying on the system to automatically optimize their program, the programmer *guides* the system to gradually refine the program to be more efficient. For example, one may instruct the system to unroll or fuse certain loops, or arrange data into tiles to take advantage of SIMD instructions. In fact, some database systems already provide mechanisms for the programmer to specify hints to the optimizer, and the most performance-critical queries are already decorated with hand-picked join orders, cardinality estimates, and index selections. We believe the first practical, high-performance relational programming system will be built on top of such a scheduling language.

Modular Architecture. Traditionally, database systems are built as monolithic systems. Partly due to the commercial nature of many database systems, and partly due to the rapid evolution of hardware and software, each system is built from scratch around a new architectural design. This means building every new system requires significant effort that is not reusable by other systems. However, the design of many components of a database system has largely stablized: query optimizers usually follow the CASCADE framework [Gra95], with the execution engine implementing either the pull-based or push-based model [KLK+18], and the data are stored as either rows or columns [ABH+13]. Even within the optimizer, most systems implement variants of the

dynamic programming algorithm [MN06b] which interacts with a cardinality estimator to find the cheapest join order. We therefore advocate developing modular, reusable components for relational data systems, and designing new system architectures by composing these components. There is already encourage progress in this direction, for example the Apache Calcite project [BCRH+18] and the Orca optimizer [SAR+14]. Again taking inspiration from the programming languages community, the database community should aim to build a collection of database components like the LLVM project did for compilers [LA04].

Testing and Analyzing Relation Programs. In the real world, software development is an iterative process. No software is guarenteed to be bug-free in the beginning, and requirements change over time. Just like for general purpose programming languages, tools to test and analyze relational programs are essential for programmer productivity and software quality. Testing and analyzing relational programs presents unique challenges. For example, each relational program usually touches a large amount of data, so debugging the program on the full dataset can be prohibitively expensive. As we have seen in this dissertation, the data system applies advanced optimizations to the program, which means the program that is executed is usually very different from the original program written by the programmer. Although there is increasing interest in testing and analyzing relational data *systems* [RS20b, RS20c, RS20a], we believe the problem of testing and analyzing relational programs also deserves attention from the community.

Bibliography

[ABC+16] 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 Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016.

[ABH+13] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Found. Trends Databases*, 5(3):197–280, 2013.

[AGM13] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. 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 261–272. ACM, 2000.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AKNS17] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of datalog programs. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 689–706. Springer, 2017.

[ALT+17] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[Alv16] Mario Alviano. Evaluating Answer Set Programming with non-convex recursive aggregates. *Fundam. Informaticae*, 2016.

[AM00] Srinivas M. Aji and Robert J. McEliece. The generalized distributive law. *IEEE Trans. Inf. Theory*, 46(2):325–343, 2000.

[AMC+10] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 262–281. Springer, 2010.

[ANP+22] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. Convergence of datalog over (pre-) semirings. In Leonid Libkin and Pablo Barceló, editors, *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 105–117. ACM, 2022.

[ANR16] 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.

[(AS] American Statistical Association (ASA). Airline on-time performance dataset. `stat-computing.org/dataexpo/2009/the-data.html`.

[BCC+97] Francisco Bueno, Daniel Cabeza, Manuel Carro, Manuel Hermenegildo, P López-Garcıa, and Germán Puebla. The ciao prolog system. *Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM)*, 95:96, 1997.

[BCRH+18] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, pages 221–230, 2018.

[BHF12] William E Byrd, Eric Holk, and Daniel P Friedman. minikanren, live and untagged: Quine generation via relational interpreters (programming pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, pages 8–29, 2012.

[BMA+19] 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*. ACM, 2019.

[BMSU86] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In Avi Silberschatz, editor, *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*, pages 1–15. ACM, 1986.

[Boe19] Matthias Boehm. Apache systemml. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019.

[Bot] Leon Bottou. The infinite mnist dataset. `leon.bottou.org/projects/infimnist`.

[BPRM91] Isaac Balbin, Graeme S. Port, Kotagiri Ramamohanarao, and Krishnamurthy Meenakshi. Efficient bottom-up computation of queries on stratified databases. *J. Log. Program.*, 11(3&4):295–344, 1991.

[BR91] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *J. Log. Program.*, 10(3&4):255–299, 1991.

[Bra01] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[BRH+18] 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. *PVLDB*, 11(12):1755–1768, 2018.

[BRR+19] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 193–205. IEEE, 2019.

[Byr09] William E Byrd. *Relational programming in miniKanren: techniques, applications, and implementations*. PhD thesis, Indiana University, 2009.

[Car79] Bernard Carré. *Graphs and networks*. The Clarendon Press, Oxford University Press, New York, 1979. Oxford Applied Mathematics and Computing Science Series.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[CDI+18] Tyson Condie, Ariyam Das, Matteo Interlandi, Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Scaling-up reasoning and advanced analytics on bigdata. *Theory Pract. Log. Program.*, 18(5-6):806–845, 2018.

[CMA+12] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *SOCC*, 2012.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[Col90] Alain Colmerauer. An introduction to prolog iii. *Communications of the ACM*, 33(7):69–90, 1990.

[CSN05] Sara Cohen, Yehoshua Sagiv, and Werner Nutt. Equivalences among aggregate queries with negation. *ACM Trans. Comput. Log.*, 6(2):328–360, 2005.

[CV93] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *Real* conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC, USA*, pages 59–70, 1993.

[CWWC17] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[CZY+18] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 3393–3404, 2018.

[DC+01] Daniel Diaz, Philippe Codognet, et al. Design and implementation of the gnu prolog system. *Journal of Functional and Logic Programming*, 6(2001):542, 2001.

[Dec97] Rina Dechter. Bucket elimination: a unifying framework for processing hard and soft constraints. *Constraints An Int. J.*, 2(1):51–55, 1997.

[DLW+19] Ariyam Das, Youfu Li, Jin Wang, Mingda Li, and Carlo Zaniolo. Bigdata applications from graph analytics to machine learning by aggregates in recursion. In Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Inclezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang, editors, *Proceedings 35th International Conference on Logic Programming (Technical Communications),*

ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019, volume 306 of *EPTCS*, pages 273–279, 2019.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[DPT99] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 459–470. Morgan Kaufmann, 1999.

[EKTZ11] Yuval Emek, Ron Karidi, Moshe Tennenholtz, and Aviv Zohar. Mechanisms for multi-level marketing. In *Proceedings of the 12th ACM conference on Electronic commerce*, pages 209–218, 2011.

[ELB+17] 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 *CIDR*, 2017.

[Fag83] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.

[FBK05] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The reasoned schemer*. MIT Press, 2005.

[FBS+20] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.

[Fit85] Melvin Fitting. A kripke-kleene semantics for logic programs. *J. Log. Program.*, 2(4):295–312, 1985.

[Fit91] Melvin Fitting. Kleene's logic, generalized. *J. Log. Comput.*, 1(6):797–810, 1991.

[FJC+23] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu graph database management system. In *CIDR*, 2023.

[FLVE20] Matthew Francis-Landau, Tim Vieira, and Jason Eisner. Evaluation of logic programs with built-ins and aggregation: A calculus for bag relations. In *13th International Workshop on Rewriting Logic and Its Applications*, pages 49–63, April 2020.

[For82] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[FPL11] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in Answer Set Programming. *Artif. Intell.*, 2011.

[FZZ+19] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. Scaling-up in-memory datalog processing: Observations and techniques. *Proc. VLDB Endow.*, 12(6):695–708, 2019.

[GBC98] Goetz Graefe, Ross Bunker, and Shaun Cooper. Hash joins and hash teams in microsoft SQL server. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 86–97. Morgan Kaufmann, 1998.

[GCI+17] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. Verifying equivalence of spark programs. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 282–300. Springer, 2017.

[Gel89] Allen Van Gelder. The alternating fixpoint of logic programs with negation. In Avi Silberschatz, editor, *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*, pages 1–10. ACM Press, 1989.

[GGZ91] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Minimum and maximum predicates in logic programming. In Daniel J. Rosenkrantz, editor, *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 29-31, 1991, Denver, Colorado, USA*, pages 154–163. ACM Press, 1991.

[GGZ95] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Extrema predicates in deductive databases. *J. Comput. Syst. Sci.*, 51(2):244–259, 1995.

[GHLZ13] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Found. Trends Databases*, 5(2):105–195, 2013.

[GKT07] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In Leonid Libkin, editor, *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40. ACM, 2007.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming,*

Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes), pages 1070–1080. MIT Press, 1988.

[GL01] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. In Sharad Mehrotra and Timos K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 331–342. ACM, 2001.

[GM08] Michel Gondran and Michel Minoux. *Graphs, dioids and semirings*, volume 41 of *Operations Research/Computer Science Interfaces Series*. Springer, New York, 2008. New models and algorithms.

[GO19] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2019.

[GPB+22] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. Sqlite: past, present, and future. *Proceedings of the VLDB Endowment*, 15(12):3535–3547, 2022.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[Gra95] Goetz Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3), 1995.

[Gre09] Todd J. Green. Containment of conjunctive queries on annotated relations. In Ronald Fagin, editor, *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings*, volume 361 of *ACM International Conference Proceeding Series*, pages 296–309. ACM, 2009.

[GRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[GWM+19] Jiaqi Gu, Yugo H. Watanabe, William A. Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark. In Boncz et al. [BMA+19], pages 467–484.

[GZ14] Michael Gelfond and Yuanlin Zhang. Vicious circle principle and logic programs with aggregates. *Theory Pract. Log. Program.*, 2014.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[HBY13] Botong Huang, Shivnath Babu, and Jun Yang. Cumulon: optimizing statistical data analysis in the cloud. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1–12. ACM, 2013.

[HGL11] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1213–1216, New York, NY, USA, 2011. Association for Computing Machinery.

[HHS17] Dylan Hutchison, Bill Howe, and Dan Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. *CoRR*, abs/1703.07342, 2017.

[HM16] Ruining He and Julian J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, and Ben Y. Zhao, editors, *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 507–517. ACM, 2016.

[IKM07a] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 68–78. www.cidrdb.org, 2007.

[IKM07b] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 413–424. ACM, 2007.

[IR95] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Trans. Database Syst.*, 20(3):288–324, 1995.

[JPR16] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. Ajar: Aggregations and joins over annotated relations. In *PODS*. ACM, 2016.

[JPT+19] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 47–62. ACM, 2019.

[JSS16] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23,*

2016, Proceedings, Part II, volume 9780 of *Lecture Notes in Computer Science*, pages 422–430. Springer, 2016.

[Kag] Kaggle. Netflix prize data. `kaggle.com/netflix-inc/netflix-prize-data`.

[KKC+17] 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.

[KKL15] David Kernert, Frank Köhler, and Wolfgang Lehner. Spmacho - optimizing sparse linear algebra expressions with probabilistic density estimation. In Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martín Ugarte, Jan Van den Bussche, and Jan Paredaens, editors, *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, pages 289–300. OpenProceedings.org, 2015.

[KKW99] Alfons Kemper, Donald Kossmann, and Christian Wiesner. Generalised hash teams for join and group-by. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 30–41. Morgan Kaufmann, 1999.

[KLB+22] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V Hermenegildo, Jose F Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, et al. Fifty years of prolog and beyond. *Theory and Practice of Logic Programming*, 22(6):776–858, 2022.

[KLK+18] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, 2018.

[KNN+17] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-database learning with sparse tensors. *CoRR*, abs/1703.04780, 2017.

[KNP+22] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. Datalog in wonderland. *SIGMOD Rec.*, 51(2):6–17, 2022.

[KNR16] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. In *PODS*. ACM, 2016.

[KNS17] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In 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.

[Koh03] Jürg Kohlas. *Information algebras - generic structures for inference.* Discrete mathematics and theoretical computer science. Springer, 2003.

[KPS97] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Euro-Par '97 Parallel Processing, Third International Euro-Par Conference, Passau, Germany, August 26-29, 1997, Proceedings*, volume 1300 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 1997.

[KS91] David B. Kemp and Peter J. Stuckey. Semantics of logic programs with aggregates. In *Logic Programming*, 1991.

[Kui97] Werner Kuich. Semirings and formal power series: their relevance to formal languages and automata. In *Handbook of formal languages, Vol. 1*, pages 609–677. Springer, Berlin, 1997.

[KW08] Jürg Kohlas and Nic Wilson. Semiring induced valuation algebras: Exact and approximate local computation algorithms. *Artif. Intell.*, 172(11):1360–1399, 2008.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[LCK19] Side Li, Lingjiao Chen, and Arun Kumar. Enabling and optimizing non-linear feature interactions in factorized linear algebra. In Boncz et al. [BMA+19], pages 1571–1588.

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[LHK10] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 1361–1370, 2010.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In Mihalis Yannakakis and Serge Abiteboul, editors, *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles*

of Database Systems, May 22-25, 1995, San Jose, California, USA, pages 95–104. ACM Press, 1995.

[LRT79] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16(2):346–358, 1979.

[LS22] Yanhong A. Liu and Scott D. Stoller. Recursive rules with aggregation: A simple unified semantics. In *LFCS*, 2022.

[LT80] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9(3):615–627, 1980.

[Mac81] Bruce J MacLennan. Introduction to relational programming. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 213–220, 1981.

[McS22] Frank McSherry. Recursion in materialize. `https://github.com/frankmcsherry/blog/blob/master/posts/2022-12-25.md`, 2022. Accessed: December 2022.

[MFPR90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, pages 247–258. ACM Press, 1990.

[ML12] Julian J McAuley and Jure Leskovec. Learning to discover social circles in ego networks. In *NIPS*, volume 2012, pages 548–56. Citeseer, 2012.

[MLK+21] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. LSQB: a large-scale subgraph query benchmark. In Vasiliki Kalavri and Nikolay Yakovets, editors, *GRADES-NDA '21: Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Virtual Event, China, 20 June 2021*, pages 8:1–8:11. ACM, 2021.

[MN06a] Guido Moerkotte and Thomas Neumann. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *VLDB*, 2006.

[MN06b] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 930–941. ACM, 2006.

[MN08] Guido Moerkotte and Thomas Neumann. Dynamic Programming Strikes Back. In *SIGMOD*, 2008.

[MP94] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, pages 103–114. ACM Press, 1994.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, 1990.

[MS19] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, 2019.

[MSZ13a] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. A declarative extension of horn clauses, and its significance for Datalog and its applications. *Theory Pract. Log. Program.*, 2013.

[MSZ13b] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. Extending the power of datalog recursion. *VLDB J.*, 22(4):471–493, 2013.

[MTKW18] David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. Datalog: concepts, history, and outlook. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 3–100. ACM / Morgan & Claypool, 2018.

[Nel80] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Stanford, CA, USA, 1980. AAI8011683.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011.

[Ngo18] 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, New York, NY, USA, 2018. ACM.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer-Verlag, Berlin, 1999.

[NPRR12] 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, New York, NY, USA, 2012. ACM.

[NRR13] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.

[NWZ+21] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–28, 2021.

[OS16] Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Rec.*, 45(2):5–16, 2016.

[PDST00] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A chase too far? 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 273–284. ACM, 2000.

[PMAJ01] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. Block oriented processing of relational database operations in modern computer architectures. In Dimitrios Georgakopoulos and Alexander Buchmann, editors, *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 567–574. IEEE Computer Society, 2001.

[Pro21] Scryer Prolog. Scryer prolog. *URL: https://github. com/mthom/scryer-prolog.[Accessed 11 March, 2022]*, 2021.

[PSWT15] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 1–11. ACM, 2015.

[Raa22] Mark Raasveldt. Duckdb - A modern modular and extensible database system. In Satyanarayana R. Valluri and Mohamed Zait, editors, *1st International Workshop on Composable Data Management Systems, CDMS@VLDB 2022, Sydney, Australia, September 9, 2022*, 2022.

[RAD03] Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. Trust management for the semantic web. In *International semantic Web conference*, pages 351–368. Springer, 2003.

[RBA+13] 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.

[RL18] Timothy Roscoe and Boon Thau Loo. Declarative networking. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[RM19] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019.

[RM20] Mark Raasveldt and Hannes Mühleisen. Data management for data science - towards embedded analytics. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[RMZ+20] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. *Proc. ACM Program. Lang.*, 4(POPL):62:1–62:27, 2020.

[Roc] Tim Rocktäschel. Einsum is all you need - Einstein summation in deep learning. `https://rockt.github.io/2018/04/30/einsum`.

[RS92] Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In Moshe Y. Vardi and Paris C. Kanellakis, editors, *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*, pages 114–126. ACM Press, 1992.

[RS94] Raghu Ramakrishnan and Divesh Srivastava. Semantics and optimization of constraint queries in databases. *IEEE Data Eng. Bull.*, 17(2):14–17, 1994.

[RS20a] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1140–1152. ACM, 2020.

[RS20b] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA):211:1–211:30, 2020.

[RS20c] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 667–682. USENIX Association, 2020.

[RVB19] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. Certified semantics for minikanren. In *Proceedings of the 2019 miniKanren and Relational Programming Workshop*, pages 80–98, 2019.

[SAC+79a] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[SAC+79b] 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.

[SAR+14] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. Orca: a modular query optimizer architecture for big data. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 337–348. ACM, 2014.

[SGL15] Jiwon Seo, Stephen Guo, and Monica S. Lam. Socialite: An efficient graph query language based on datalog. *IEEE Trans. Knowl. Data Eng.*, 27(7):1824–1837, 2015.

[SHO18] Michelle Mills Strout, Mary W. Hall, and Catherine Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018.

[SK91] Michael Stonebraker and Greg Kemnitz. The postgres next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[SLB+11] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 609–616. Omnipress, 2011.

[SLZ+18] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, pages 515–527. ACM, 2018.

[SOC16] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. 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 3–18. ACM, 2016.

[SRHN19] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. Synthesizing datalog programs using numerical relaxation. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6117–6124. ijcai.org, 2019.

[SRLS17] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. Optimizing big-data queries using program synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 631–646. ACM, 2017.

[SS88] Prakash P. Shenoy and Glenn Shafer. Axioms for probability and belief-function proagation. In Ross D. Shachter, Tod S. Levitt, Laveen N. Kanal, and John F. Lemmer, editors, *UAI '88: Proceedings of the Fourth Annual Conference on Uncertainty in Artificial Intelligence, Minneapolis, MN, USA, July 10-12, 1988*, pages 169–198. North-Holland, 1988.

[STB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006.

[SYI+16a] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1135–1149, 2016.

[SYI+16b] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1135–1149, New York, NY, USA, 2016. Association for Computing Machinery.

[Sys] SystemML. Systemml performance tests. `https://github.com/apache/systemml/tree/master/scripts/perftest`.

[SYZ15] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in deals. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 867–878. IEEE Computer Society, 2015.

[TB13] Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 135–152. ACM, 2013.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[TSTL11] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.

[Vel14] 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, Athens, Greece, 2014. OpenProceedings.org.

[VGdHT09] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. Symbolic query exploration. In Karin K. Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, volume 5885 of *Lecture Notes in Computer Science*, pages 49–68. Springer, 2009.

[Via21a] Victor Vianu. Datalog unchained. In Leonid Libkin, Reinhard Pichler, and Paolo Guagliardo, editors, *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021*, pages 57–69. ACM, 2021.

[Via21b] Victor Vianu. Datalog unchained. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'21, page 57–69, New York, NY, USA, 2021. Association for Computing Machinery.

[VZT+18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[WAN+22] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. Optimizing recursive queries with progam synthesis. 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 79–93. ACM, 2022.

[WBH15] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *Proc. VLDB Endow.*, 8(12):1542–1553, August 2015.

[WDLC18] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.*, 2(POPL):56:1–56:29, 2018.

[WHL+20] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *CoRR*, abs/2002.07951, 2020.

[WHS+20] 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.

[WKN+22] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. Optimizing recursive queries with program synthesis. *CoRR*, abs/2202.10390, 2022.

[WNW+21] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[WSC+20] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. Data migration using datalog program synthesis. *Proc. VLDB Endow.*, 13(7):1006–1019, 2020.

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[WWF+20] Max Willsey, Yisu Remy Wang, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. egg: Easy, efficient, and extensible e-graphs, 2020.

[WWS23] Yisu Remy Wang, Max Willsey, and Dan Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM Manag. Data*, 1(2):150:1–150:23, 2023.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

[YPW+21] Yichen Yang, Phitchaya Mangpo Phothilimthana, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In Alex Smola, Alex Dimakis, and Ion Stoica, editors, *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org, 2021.

[ZDG+19] Carlo Zaniolo, Ariyam Das, Jiaqi Gu, Youfu Li, Mingda Li, and Jin Wang. Monotonic properties of completed aggregates in recursive queries. *CoRR*, abs/1910.08888, 2019.

[ZDG+21] Carlo Zaniolo, Ariyam Das, Jiaqi Gu, Youfu Li, Mingda Li, and Jin Wang. Developing big-data application as queries: an aggregate-based approach. *IEEE Data Eng. Bull.*, 44(2):3–13, 2021.

[Zho12] Neng-Fa Zhou. The language features and architecture of b-prolog. *Theory and Practice of Logic Programming*, 12(1-2):189–218, 2012.

[ZWF+23] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *CoRR*, abs/2304.04332, 2023.

[ZWWT22] Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. Relational e-matching. *Proc. ACM Program. Lang.*, 6(POPL):1–22, 2022.

[ZYD+17] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *Theory Pract. Log. Program.*, 17(5-6):1048–1065, 2017.

[ZYDI16] Carlo Zaniolo, Mohan Yang, Ariyam Das, and Matteo Interlandi. The magic of pushing extrema into recursion: Simple, powerful datalog programs. In Reinhard Pichler and Altigran Soares da Silva, editors, *Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City, Panama, May 8-10, 2016*, volume 1644 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[ZYI+18] Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. Declarative bigdata algorithms via aggregates and relational database dependencies. In Dan Olteanu and Barbara Poblete, editors, *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali,*

Colombia, May 21-25, 2018, volume 2100 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.