Datalog in Wonderland*

Mahmoud Abo Khamis
RelationalAI

Hung Q. Ngo
RelationalAI

Reinhard Pichler
TU Wien

Dan Suciu
University of Washington

Yisu Remy Wang
University of Washington

ABSTRACT

Modern data analytics applications, such as knowledge graph reasoning and machine learning, typically involve recursion through aggregation. Such computations pose great challenges to both system builders and theoreticians: first, to derive simple yet powerful abstractions for these computations; second, to define and study the semantics for the abstractions; third, to devise optimization techniques for these computations.

In recent work we presented a generalization of Datalog called Datalog°, which addresses these challenges. Datalog° is a simple abstraction, which allows aggregates to be interleaved with recursion, and retains much of the simplicity and elegance of Datalog. We define its formal semantics based on an algebraic structure called Partially Ordered Pre-Semirings, and illustrate through several examples how Datalog° can be used for a variety of applications. Finally, we describe a new optimization rule for Datalog°, called the FGH-rule, then illustrate the FGH-rule on several examples, including a simple magic-set rewriting, generalized semi-naïve evaluation, and a bill-of-material example, and briefly discuss the implementation of the FGH-rule and present some experimental validation of its effectiveness.

1. INTRODUCTION

The database community has developed an *elegant* abstraction for recursive computations, in the context of Datalog [1, 45]. Query evaluation and optimization techniques such as semi-naïve evaluation and magic-set transformation have led to efficient implementations [24, 3, 4]. Datalog and its optimization, however, are not sufficiently powerful to handle the computations commonly found in the modern data stack; for example, new applications often involve iterative computations with aggregations such as summation or minimization over the reals. Even under the Boolean domain, clean and practical fixpoint semantics require strong assumptions on the input rules such as stratification [1]. In addition to difficult semantic questions, semi-naïve evaluation and magic-set transformation also impose strong assumptions about the input such as monotonicity in terms of set-containment. These assumptions typically do not hold in the new world.

A *powerful* abstraction to address these challenges can be found in the study of aggregation with the aid of semirings [21]. This research line generalizes relational algebra beyond sets and multisets. The semiring semantics can capture a wide range of operations including tensor algebra.

In recent works [27, 26, 46, 47], we combine the elegance of Datalog and the power of semiring abstractions into a new language called Datalog°. We studied its semantics, optimization, and convergence behavior. In particular, we derived a novel optimization primitive called the FGH-rule, which serves as a key component for generalizing both semi-naïve evaluation and magic-set transformations to Datalog°. This article highlights our findings.

Semantics of Datalog°. A Datalog program is a collection of (unions of) conjunctive queries, operating on relations. Analogously, a Datalog° program is a collection of (sum-) sum-product queries on S-relations[1] for some (pre-)semiring S [20]. In short, Datalog° is like Datalog, where $\wedge, \vee$ are replaced by $\otimes, \oplus$. Recall that an S-relation is a function from the set of tuples to a (pre-) semiring S; the domain of the tuples is called the *key space*, while S is the *value space*. The value space of standard relations is the Booleans (see Sec. 3).

EXAMPLE 1.1. *A matrix A over the real numbers is an $\mathbb{R}$-relation, where each tuple $A[i,j]$ has the value a_{ij}; $\mathbb{R}$ denotes the sum-product semiring $(\mathbb{R}, +, \cdot, 0, 1)$. Both the objective and gradient of the* ridge linear regression *problem $\min_x J(x)$, with $J(x) = \frac{1}{2}\|Ax-b\|^2 + (\lambda/2)\|x\|^2$, are expressible in* Datalog°, *because they*

*Suciu and Wang were partially supported by NSF IIS 1907997 and NSF IIS 1954222. Pichler was supported by the Austrian Science Fund (FWF):P30930.

[1] *K-relations* were introduced by Green et al. [21]; we call them *S-relations* in this paper where S stands for "semiring".

are sum-sum-product queries. The gradient $\nabla J(x) = A^\top Ax - A^\top b + \lambda x$*, for example, is the following sum-sum-product query:*

$$\nabla[i] \text{ :- } \sum_{j,k} a[k,i] \cdot a[k,j] \cdot x[j] + \sum_j -1 \cdot a[j,i] \cdot b[j] + \lambda \cdot x[i]$$

The gradient has the same dimensionality as x, and the group-by variable is i. Gradient descent is an algorithm to find the solution of $\nabla J(x) = 0$*, or equivalently to solve for a fixpoint solution to the* Datalog$^\circ$ *program* $x = f(x)$ *where* $f(x) = x - \alpha \nabla J(x)$ *for some step-size* α.[2]

EXAMPLE 1.2. *The* all-pairs shortest paths *(APSP) problem is to compute the shortest path length* $P[x,y]$ *between any pair* x, y *of vertices in the graph, given the length* $E[x,y]$ *of edges in the graph. The value space of* $E[x,y]$ *can be the reals* $\mathbb{R}$ *or the non-negative reals* $\mathbb{R}_+$*. The APSP problem in* Datalog$^\circ$ *can be expressed very compactly as:*

$$P[x,y] \text{ :- } \min(E[x,y], \min_z(P[x,z] + E[z,y])) \tag{1}$$

where $(\min, +)$ *are the "addition" and "multiplication" operators in the tropical semirings; see Ex. 3.1.*

By changing the semiring, Datalog$^\circ$ is able to express similar problems in exactly the same way. For example, (1) becomes transitive closure over the Boolean semiring, the $p+1$ shortest paths over the Trop_p^+ semiring, and so forth.

In Datalog, the least fixpoint semantics was defined w.r.t. set inclusion [1]. To generalize this semantics for Datalog$^\circ$, we generalize set inclusion to a partial order over S-relations. We define a *partially ordered, pre-semiring* (POPS, Sec. 3) to be any pre-semiring S [20] with a partial order, where both $\oplus$ and $\otimes$ are monotone operations. Thus, in Datalog$^\circ$ the value space is always some POPS. Given this partial order, the semantics of a Datalog$^\circ$ program is defined naturally, as the least fixpoint of the immediate consequence operator.

Optimizations. While Datalog is designed for iteration, Datalog engines typically optimize only the loop body but not the actual loop. The few systems that do, are limited to a small number of hard-coded optimizations, like magic-set rewriting and semi-naïve evaluation. Datalog$^\circ$ supports these classic optimizations, and more. We describe these optimizations in Sec. 4.1, but give here a brief preview, and start by illustrating how the semi-naïve algorithm extends to Datalog$^\circ$. Consider the program computing the transitive closure of E:

$$P(x,y) \text{ :- } E(x,y) \vee \bigvee_z (P(x,z) \wedge E(z,y)) \tag{2}$$

We use parentheses like $E(x,y)$ for standard relations, whose value space is the set of Booleans, and use square brackets, like $E[x,y]$, when the value space is something else. The rule (2) deviates only slightly from standard Datalog syntax, in that it uses explicit conjunction and disjunction, and binds the variable z explicitly. After initializing $P_0(x,y) = \delta_0(x,y) = E(x,y)$, at the t'th iteration, the semi-naïve algorithm does the following:

$$\begin{aligned} \delta_t(x,y) &= \left(\bigvee_z (\delta_{t-1}(x,z) \wedge E(z,y)) \right) \setminus P_{t-1}(x,y) \\ P_t(x,y) &= P_{t-1}(x,y) \cup \delta_t(x,y) \end{aligned} \tag{3}$$

By computing the δ relation so, we avoid re-deriving many facts in each iteration. Another way to see this is that, when δ is much smaller than P, then the join between δ and E in (3) is much cheaper than the join between P and E in (2). The set difference operation aims precisely to keep δ small. Somewhat surprisingly, the same principle can be extended to Datalog$^\circ$, as we illustrate next.

EXAMPLE 1.3 (APSP-SN). *The semi-naïve algorithm for the APSP problem (Example 1.2) is:*

$$\begin{aligned} \delta_t[x,y] &= \left(\min_z (\delta_{t-1}[x,z] + E[z,y]) \right) \ominus P_{t-1}[x,y] \\ P_t[x,y] &= \min(P_{t-1}[x,y], \delta_t[x,y]) \end{aligned} \tag{4}$$

The difference operator $\ominus$ *is defined as follows:*

$$b \ominus a = \begin{cases} b & \text{if } b < a \\ \infty & \text{if } b \geq a \end{cases}$$

As in the standard semi-naïve algorithm, our goal is to keep δ *small, by storing only tuples with a finite value,* $\delta[x,y] \neq \infty$*. We use the* $\ominus$ *operator for that purpose. Consider the rule* (4)*. If* $b = \delta_{t-1}[x,z] + E[z,y]$ *is the newly discovered path length, and* $a = P_{t-1}[x,y]$ *is the previously known path length, then* $b \ominus a$ *is finite iff* $b < a$*, i.e. only when the new path length strictly decreases. Correctness of the semi-naïve algorithm follows from the identity* $\min(a, b \ominus a) = \min(a,b)$*. We note that, recently, Budiu et al. [7] have developed a very general incremental view maintenance technique, which also leads to the semi-naïve algorithm, for the case when the value space is restricted to an abelian group.*

We have introduced in [47] a simple, yet very general optimization rule, called the FGH-rule. The semi-naïve algorithm is one instance of the FGH-rule, but so are many other optimizations, as we illustrate in Sec. 4. For

[2] In practice, the fixpoint computation for (non-accelerated) gradient descent is more complicated, where α is dynamically adjusted using variants of back-tracking line-search [37].

a brief preview, we illustrate the FGH-optimization with the following example:

EXAMPLE 1.4. *Soufflé [24] is a popular open source Datalog system that supports aggregates, but does not allow aggregates in recursive rules. This means that we cannot write APSP as in rule* (1)*. The common workaround is to stratify the program: we first compute the lengths of* all paths *between each pair of vertices, then take the minimum:*

$$\begin{aligned} P_{all}(x,y,d) &\text{ :- } E(x,y,d).\\ P_{all}(x,y,d_1+d_2) &\text{ :- } E(x,z,d_1), P_{all}(z,y,d_2).\\ P[x,y] &= \min_d\{d \mid P_{all}(x,y,d)\} \end{aligned}$$

Of course, this program diverges on graphs with cycles, and is quite inefficient on acyclic graphs. The FGH-rule rewrites this naïve program into (1)*.*

Organization. Sec. 2 discusses related work; Sec. 3 introduces the syntax and semantics of Datalog°; Sec. 4 describes the FGH-framework for optimizing Datalog° programs, including magic-set transformation and semi-naïve evaluation; finally, Sec. 5 concludes.

2. RELATED WORK

Researchers have extended Datalog in many ways to enhance its expressiveness. Some of these extensions also reveal opportunities for various optimizations. This section surveys some existing research on Datalog extensions (Sec. 2.1) and their optimization (Sec. 2.2).

2.1 Datalog Extensions

Pure Datalog is very spartan: neither negation nor aggregate is allowed. Therefore, the literature on Datalog extensions is vast, and we will not attempt to cover the whole space. Instead, 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. Consider the APSP Example 1.2. We could attempt to write it in Datalog, by extending the language with a min-aggregation:

$$\begin{aligned} P(x,y,d) &\text{ :- } E(x,y,d)\\ P(x,y,\min(d)) &\text{ :- } P(x,z,d_1), E(z,y,d_2), d = d_1+d_2 \end{aligned} \tag{5}$$

However, the second rule is not monotone w.r.t. set inclusion. This is a subtle, but important point. For example, fix $E = \{(b,c,20),(b',c,10)\}$: if P is $\{(a,b,1)\}$, then the output of the rule is $\{(a,c,21)\}$, but when P is the superset $\{(a,b,1),(a,b',1)\}$, then the output is $\{(a,c,11)\}$, which is not a superset of the previous output, $\{(a,c,21)\} \not\supseteq \{(a,c,11)\}$.

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. [35], the idea is inspired by stratified negation, where every negated relation must be computed in a previous stratum. Ex. 1.4 is stratified: the first two rules form the first stratum and compute P_{all} to a fixpoint in regular Datalog[3], and the last rule performs the min aggregate in its own 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 – Ex. 1.2 is not stratified, and would therefore be invalid. Since Ex. 1.2 is equivalent to but more efficient than Ex. 1.4, disallowing the former leads to suboptimal performance. The stratification requirement can also be a cognitive burden on the programmer. In fact, the most general notion of stratification, dubbed "magic stratification" [35], involves both a syntactic condition and a semantic condition defined in terms of derivation trees.

Generalized Ordering. In this article, we follow the approach that restores monotonicity by generalizing the ordering on which monotonicity is defined. The key idea is that, although a program like that shown in Eq. (5) is not monotone according to the $\subseteq$ ordering on sets, we can pick another order under which P **is** monotone. Ross and Sagiv [39] define the ordering[4] $P \sqsubseteq P'$ 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 (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 [39] require it to be a complete lattice, Conway et al. [9] require only a semilattice, whereas Green et al. [21] require S to be an ω-continuous semiring. These proposals bundle the order-

[3]The second rule uses the built-in function $+$.

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

ing together with two operations $\otimes$ and/or $\oplus$. In this article, 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. [14] model min and max aggregates in Datalog with negation, thereby supporting aggregates via semantics defined for negation. Mazuran et al. [34] extend Datalog with counting quantification, which additionally captures `SUM`. Kemp and Stuckey [25] extend the well-founded semantics [16, 15] and stable model semantics [17] 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 [12, 18, 2]. Liu and Stoller [33] give a comprehensive survey of this space.

2.2 Optimizing Extended Datalog

New opportunities for optimization emerge once we extend Datalog to support aggregation. We have already seen one instance in Ex. 1.4 and Ex. 1.2. Intuitively, the optimization can be seen as applying the group-by pushdown rule to recursive programs; or, it can be seen as a variant of the magic-set transformation, where we push the aggregate instead of a predicate into recursion. The optimizations we study in this article are inspired by a long line of work done by Carlo Zaniolo and his collaborators [14, 51, 50, 49]. Indeed, our FGH-rule is a generalization of Zaniolo et al.'s notion of pre-mappability (PreM) [51], which evolved from earlier ideas on pushing extrema (min / max aggregate) into recursion [50]. A different and more recent solution to aggregate pushdown by Shkapsky et al. [42] is to use set-monotonic aggregation operators. Unlike PreM, pushing monotonic aggregates requires no preconditions, but may result in slightly less efficient programs.

In addition to new optimizations like aggregate pushdown, classic techniques including semi-naïve evaluation and magic-set transformation also exhibit interesting twists in extended Datalog variants. For example, Conway et al. [9] generalize the set-based semi-naïve evaluation into one over S-relations where S is a semilattice, and Mumick et al. [35] adapt magic-set transformation to work in the presence of aggregate-in-recursion. In this article, we will show how the FGH-rule can capture both the magic-set transformation and the general semi-naïve evaluation.

To evaluate a recursive Datalog$^\circ$ program is to solve fixpoint equations over semirings, which has been studied in the automata theory [28], program analysis [10, 36], and graph algorithms [8, 32, 31] communities since the 1970s. (See [40, 23, 29, 20, 52] and references thereof). The problem took slighly different forms in these domains, but at its core, it is to find a solution to the equation $x = f(x)$, where $x \in S^n$ is a vector over the domain S of a semiring, and $f : S^n \to S^n$ has multivariate polynomial component functions. A literature review on this fixpoint problem can be found in [26].

3. DATALOG$^\circ$

Datalog$^\circ$ extends Datalog in two ways. First, all relations are S-relations over some semiring S. Second, the semiring needs to be partially ordered; more precisely, it needs to be a POPS.

POPS A *partially ordered pre-semiring*, or POPS, is a tuple $\boldsymbol{S} = (S, \oplus, \otimes, \mathbf{0}, \mathbf{1}, \sqsubseteq)$, where:

- $(S, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a *pre-semiring*, meaning that $\oplus, \otimes$ are commutative and associative, have identities $\mathbf{0}$ and $\mathbf{1}$ respectively, and $\otimes$ distributes over $\oplus$.
- $\sqsubseteq$ is a partial order with a minimal element, $\bot$.
- Both $\oplus, \otimes$ are monotone operations w.r.t. $\sqsubseteq$.

Pre-semirings have been studied intensively [20], and we need to straighten up some terminology before proceeding. A pre-semiring only requires $\oplus$ to be commutative: if $\otimes$ is also commutative, then it is called a commutative pre-semiring. All pre-semirings in this paper are commutative. If $x \otimes \mathbf{0} = \mathbf{0}$ holds for all x, then $\boldsymbol{S}$ is called a *semiring*. We call $\otimes$ *strict* if $x \otimes \bot = \bot$ for all x; throughout this paper we will assume that $\otimes$ is strict.

When the relation $x \sqsubseteq y$ is defined by $\exists z : x \oplus z = y$, then $\sqsubseteq$ is called the *natural order*. In that case, the minimal element is $\bot = \mathbf{0}$. Naturally ordered semirings appear often in the literature [20, 21, 11], but we do *not* require POPS to be naturally ordered (see Example 3.2).

$\boldsymbol{S}$-Relations Fix a POPS $\boldsymbol{S}$, and a domain D, which, for simplicity, we will assume to be finite. An *$\boldsymbol{S}$-relation* is a function $R : D^k \to S$. We call D^k the *key space* and $\boldsymbol{S}$ the *value space*. When $\boldsymbol{S}$ is the set of Booleans, which we denote $\mathbb{B}$, then a $\mathbb{B}$-relation is a standard relation, i.e. a set. Next, we need to define *sum-product*, and *sum-sum-product* expressions, which are generalizations of Conjunctive Queries (CQ), and Unions of Conjunctive Queries (UCQ):

$$T[x_1, \ldots, x_k] ::= \bigoplus_{x_{k+1}, \ldots, x_p \in D} \{A_1 \otimes \cdots \otimes A_m \mid C\} \quad (6)$$

$$F[x_1, \ldots, x_k] ::= T_1[x_1, \ldots, x_k] \oplus \cdots \oplus T_q[x_1, \ldots, x_k] \quad (7)$$

The sum-product expression (6) defines a new $\boldsymbol{S}$-relation T, with head variables $x_1,\ldots,x_k$. It consists of a summation of products, where the bound variables $x_{k+1},\ldots,x_p$ range over the domain D, and may be further restricted to satisfy a condition C. Each factor A_u is either a *relational atom*, $R_i[x_{t_1},\ldots,x_{t_{k_i}}]$, where R_i is a relation name from some given vocabulary, or an *equality predicate*, $[x_t = x_s]$. The sum-sum-expression (7) is a sum of sum-product expressions, with the restriction that all summands have the same head variables.

Datalog$^\circ$ The input to a Datalog$^\circ$ program consists of m EDB predicates[5] $\boldsymbol{E} = (E_1,\ldots,E_m)$, and the output consists of n IDB predicates $\boldsymbol{P} = (P_1,\ldots,P_n)$. The Datalog$^\circ$ program has one rule for each IDB:

$$\begin{aligned} P_1[\mathtt{vars}_1] &\text{ :- } F_1[\mathtt{vars}_1] \\ &\cdots \\ P_n[\mathtt{vars}_n] &\text{ :- } F_n[\mathtt{vars}_n] \end{aligned} \tag{8}$$

where each $F_i[\mathtt{vars}_i]$ is a sum-sum-product expression using the relation symbols $E_1,\ldots,E_m,P_1,\ldots,P_n$. We say that the program is *linear* if each product contains at most one IDB predicate.

Semantics The tuple of all n sum-sum-product expressions $\boldsymbol{F} = (F_1,\ldots,F_n)$ is called the *Immediate Consequence Operator*, or ICO. Fix an instance of the EDB relations $\boldsymbol{E}$. The ICO defines a function $\boldsymbol{P} \mapsto \boldsymbol{F}(\boldsymbol{E},\boldsymbol{P})$ that maps the IDB instances $\boldsymbol{P}$ to new IDB instances. The *semantics* of a Datalog$^\circ$ program is the least fixpoint of the ICO, when it exists. Equivalently, the Datalog$^\circ$ program is the result returned by the following naïve evaluation algorithm:

$$\begin{aligned} &\boldsymbol{P}_0 = \bot; \quad t = 0; \\ &\mathtt{repeat}\ \ \boldsymbol{P}_{t+1} = \boldsymbol{F}(\boldsymbol{E},\boldsymbol{P}_t); \\ &\qquad\quad t = t+1; \\ &\mathtt{until}\ \ \boldsymbol{P}_t = \boldsymbol{P}_{t-1} \end{aligned}$$

The reader may have recognized that Datalog$^\circ$ is quite similar to Datalog, with minor changes: the operations $\vee,\wedge$ are replaced with $\oplus,\oplus$, and multiple rules for the same IDB predicate are combined into a single sum-sum-product rule. Importantly, Datalog$^\circ$ retains the same simple fixpoint semantics, but it generalizes from sets to $\boldsymbol{S}$-relations. We illustrate this with two examples.

EXAMPLE 3.1. *Consider the one-rule program:*

$$P[x,y] \text{ :- } E[x,y] \oplus \bigoplus_z (P[x,z] \otimes E[z,y]) \tag{9}$$

We will interpret it over several POPS and use it to compute quite different things.

[5]EDB and IDB stand for extensional database and intentional database respectively [1].

Booleans *Choose* $\mathbb{B} = (\{0,1\},\vee,\wedge,0,1,\leq)$ *to be the value space, where* $0 \leq 1$. *Then the program in Eq.* (9) *becomes the transitive closure program in Eq.* (2).

Tropical Semiring $\mathsf{Trop}^+ = (\mathbb{R}_+,\min,+,\infty,0,\geq)$ *is a naturally ordered POPS, called the* tropical semiring. *When we choose it as value space, then the program in Eq.* (9) *becomes the APSP program in Eq.* (1). *We briefly illustrate its semantics on a graph with three nodes* a,b,c *and edges (we show only entries with value* $< \infty$*):*

$$E[a,b] = 1 \qquad E[a,c] = 10 \qquad E[b,c] = 1$$

During the iterations $t = 0,1,2,\ldots$ *of the naïve algorithm, P "grows" as follows (notice that the order relation in* Trop^+ *is the reverse of the usual one, thus* ∞ *is the smallest value):*

	$P[a,b]$	$P[a,c]$	$P[b,c]$
$t=0$	∞	∞	∞
$t=1$	1	10	1
$t=2$	1	2	1

p-Tropical *We can use the same program over a different POPS to compute the* $p+1$ *shortest paths, for some fixed number* $p \geq 0$. *We need some notations. If* $\boldsymbol{x} = \{\!\{x_0 \leq x_1 \leq \ldots \leq x_n\}\!\}$ *is a bag of numbers, then we denote by* $\min_p(\boldsymbol{x}) \stackrel{def}{=} \{\!\{x_0,x_1,\ldots,x_{\min(p,n)}\}\!\}$. *In other words,* $\min_p$ *retains the smallest* $p+1$ *elements of the bag* $\boldsymbol{x}$. *The p-tropical semiring is:*

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

where $\mathcal{B}_{p+1}$ *represents bags of* $p+1$ *elements, and:*[6]

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

Trop_p^+ *is naturally ordered. Now the program* (9) *computes the length of the* $p+1$ *shortest paths from x to y.*

η-Tropical *Finally, we illustrate how the same program can be used to compute the length of all paths that differ from the shortest path by* $\leq \eta$, *for some fixed* $\eta \geq 0$. *Given any finite set* $\boldsymbol{x}$ *of real numbers, define:*

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

In other words, $\min_{\leq\eta}$ *retains from the set* $\boldsymbol{x}$ *the elements at distance* $\leq \eta$ *from its minimum. Define the POPS:*

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

where $\mathcal{P}_{\leq\eta}$ *is the set of finite sets* $\boldsymbol{x}$ *where* $\max(\boldsymbol{x}) - \min(\boldsymbol{x}) \leq \eta$, *and:*

$$\boldsymbol{x} \oplus_{\leq\eta} \boldsymbol{y} \stackrel{def}{=} \min_{\leq\eta}(\boldsymbol{x} \cup \boldsymbol{y}) \quad \boldsymbol{x} \otimes_{\leq\eta} \boldsymbol{y} \stackrel{def}{=} \min_{\leq\eta}(\boldsymbol{x}+\boldsymbol{y})$$
$$\mathbf{0}_{\leq\eta} \stackrel{def}{=} \{\infty\} \qquad \mathbf{1}_{\leq\eta} \stackrel{def}{=} \{0\}$$

[6]For sets or bags $\boldsymbol{x},\boldsymbol{y}$: $\boldsymbol{x}+\boldsymbol{y} \stackrel{\text{def}}{=} \{u+v \mid u \in \boldsymbol{x}, v \in \boldsymbol{y}\}$.

$\mathsf{Trop}^+_{\leq \eta}$ *is naturally ordered. The program* (9) *computes now the length of all paths that differ by* $\leq \eta$ *from the shortest.*

EXAMPLE 3.2. *Consider now the* bill-of-material *example: the relation* $\texttt{SupPart}(x,z)$ *represents the fact that* z *is a subpart of* x*, and the goal is to compute, for each* x*, the total cost of all its direct and indirect subparts. This is written in* Datalog$^\circ$ *as follows:*

$$Q[x] \text{ :- } \texttt{Cost}[x] + \sum_z \{Q[z] \mid \texttt{SubPart}(x,z)\} \quad (10)$$

We will interpret it over two different POPS:

Natural Numbers *We start by interpreting the program over the POPS* $(\mathbb{N}, +, *, 0, 1, \leq)$*. If the* `SubPart` *hierarchy is a tree, then Eq.* (10) *computes correctly the total cost, as intended. If* `SubPart` *is a DAG, then the program may over-count some costs; we will return to this issue in Ex. 4.5. For now, we consider termination, in the case when* `SubPart` *happens to have a cycle: in that case the program diverges.*

Lifted Reals (or Lifted Naturals) *Alternatively, consider the following POPS:* $\mathbb{R}_\perp = (\mathbb{R} \cup \{\perp\}, +, *, 0, 1, \sqsubseteq)$*, where* $x + \perp = x \times \perp = \perp$ *for all* x*, and* $a \sqsubseteq b$ *if* $a = b$ *or* $a = \perp$*. This POPS is not naturally ordered. When we interpret the program in Eq.* (10) *over* $\mathbb{R}_\perp$*, then it always converges, even on a graph with cycles, because all nodes on a cycle will converge with* $Q[x] = \perp$*.*

Note that, in the above examples, only $\mathbb{R}_\perp$ (likewise $\mathbb{N}_\perp$) is a pre-semiring. All other POPS discussed here are actually semirings.

4. OPTIMIZING DATALOGO

Traditional Datalog has two major advantages: first, it has a clean declarative semantics; second, it has some powerful optimization techniques such as the semi-naïve evaluation, magic-set rewriting, and the *PreM* optimization [51]. Datalog$^\circ$ generalizes both: we have seen its semantics in Sec. 3, while here we show (following [26, 47]) that the previous optimizations are special cases of a general, yet very simple optimization rule, which we call the FGH-rule (pronounced "fig-rule").

4.1 The FGH-Rule

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:

$$\begin{aligned} &X \leftarrow X_0 \\ &\texttt{loop } X \leftarrow F(X) \texttt{ end loop} \\ &Y \leftarrow G(X) \end{aligned} \quad (11)$$

We call this an FG-program. The FGH-rule provides a sufficient condition to compute the final answer Y by another program, called the GH-program:

$$\begin{aligned} &Y \leftarrow G(X_0) \\ &\texttt{loop } Y \leftarrow H(Y) \texttt{ end loop} \end{aligned} \quad (12)$$

The FGH-Rule [47] states: if the following identity holds:

$$G(F(X)) = H(G(X)) \quad (13)$$

then the FG-program (11) and the GH-program (12) are equivalent. We supply here a "proof by picture" of the claim:

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

Our goal is to use the FGH-rule to optimize Datalog$^\circ$ programs, and we proceed as follows. Consider two Datalog$^\circ$ programs Π_1 and Π_2 given below:

$$\Pi_1: \quad \begin{aligned} X &\text{ :- } F(X) \\ Y &\text{ :- } G(X) \end{aligned} \qquad \Pi_2: \quad Y \text{ :- } H(Y)$$

Here X and Y are tuples of IDBs (for example $X = (P_1, \ldots, P_n)$ with the notation in Sec. 3), and F, G, H represent sum-sum-product expressions over these IDBs. In both cases, only the IDBs Y are returned. Then, if the FGH-rule (13) holds, and, moreover, $G(\perp) = \perp$, then Π_1 is equivalent to Π_2. We notice that, under these conditions, if Π_1 terminates, then Π_2 terminates as well.

We illustrate several applications of the FGH-rule in Sec. 4.2, then describe its implementation in an optimizer in Sec. 4.3.

4.2 Applications of the FGH-Rule

We start with some simple applications. Throughout this section we assume that the function H is given; we discuss in Sec. 4.3 how to synthesize H.

EXAMPLE 4.1 (CONNECTED COMPONENTS). *We are given an undirected graph, with edge relation* $E(x,y)$*, where each node* x *has a unique numerical label* $L[x]$*. The task is to compute for each node* x*, the minimum label* $CC[x]$ *in its connected component. This program is a well-known target of query optimization in the literature [51]. A naïve approach is to first compute the reflexive and transitive closure of* E*, then apply a* min*-aggregate:*

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

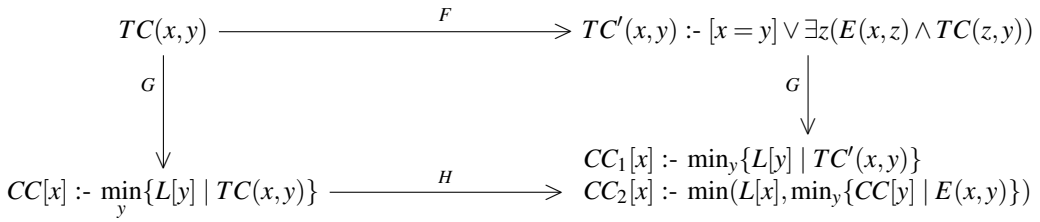

Figure 1: Visualization of the FGH-rule used in Example 4.1.

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

Figure 2: Computing CC_1 and CC_2 from Ex. 4.1.

An optimized program interleaves aggregation and recursion:

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

We prove their equivalence, by checking the FGH-rule which is shown in Fig. 1. More precisely, we need to check that $CC_1 \stackrel{def}{=} G(F(TC)) = G(TC')$ *is equivalent to* $CC_2 \stackrel{def}{=} H(G(TC)) = H(CC)$*, which is shown in Fig. 2.*

EXAMPLE 4.2 (SIMPLE MAGIC). *The simplest application of magic-set optimization [5, 6] converts* transitive closure *to* reachability*, by rewriting this program:*

$$\Pi_1: \quad \begin{array}{l} TC(x,y) \text{ :- } [x=y] \vee \exists z(TC(x,z) \wedge E(z,y)) \\ \quad Q(y) \text{ :- } TC(a,y) \end{array} \tag{14}$$

where a is some constant, into this program:

$$\Pi_2: \quad Q(y) \text{ :- } [y=a] \vee \exists z(Q(z) \wedge E(z,y)) \tag{15}$$

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* (14) *is equivalent to* (15) *by verifying the FGH-rule. The functions* F, G, H *are shown in Fig. 3. One can verify that* $G(F(TC)) = H(G(TC))$*, for any relation TC. Indeed, after converting both expressions to normal form (i.e. in sum-sum-product form), we obtain*

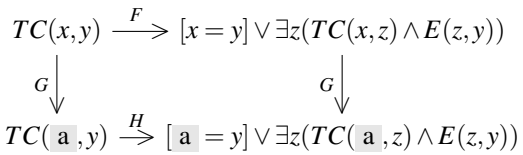

Figure 3: Expressions F, G, H in Ex. 4.2.

$G(F(TC)) = H(G(TC)) = P$*, where:*

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

Replacing $TC(a,\_)$ *by* $Q(\_)$ *now yields precisely program* Π_2 *in (15). We show in the full version of our paper [47] that, given a sideways information passing strategy (SIPS) [6] every magic-set optimization [5] over a Datalog program can be proven correct by a sequence of FGH-rule applications.*

EXAMPLE 4.3 (GENERAL SEMI-NAÏVE). *The algorithm for the naïve evaluation of (positive) Datalog re-discovers each fact from step* t *again at steps* $t+1, t+2, \ldots$ *The* semi-naïve algorithm *aims at avoiding this, by computing only the new facts. We* generalize *the semi-naïve evaluation from the Boolean semiring to any POPS* $\mathbf{S}$*, and prove it correct using the FGH-rule. We require* $\mathbf{S}$ *to be a complete distributive lattice and* $\oplus$ *to be idempotent, and define the "minus" operation as:* $b \ominus a \stackrel{def}{=} \bigwedge\{c \mid b \sqsubseteq a \oplus c\}$*, then prove using the FGH-rule the following programs equivalent:*

Π_1:	Π_2:
$X_0 := \mathbf{0}$;	$Y_0 := \mathbf{0}$;
	$\Delta_0 := F(\mathbf{0}) \ominus \mathbf{0}$; $(= F(\mathbf{0}))$
`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:

$$G(X) \stackrel{def}{=} (X, F(X) \ominus X)$$

$$H(X, \Delta) \stackrel{def}{=} (X \oplus \Delta, F(X \oplus \Delta) \ominus (X \oplus \Delta))$$

Then we prove that $G(F(X)) = H(G(X))$ *by exploiting the fact that* $\mathbf{S}$ *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-naïve 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.*

We remark that the FGH-rule is a generalization of an optimization rule introduced by Zaniolo et al. [51] and called *Pre-mappability*, or PreM. The PreM property asserts that the identity $G(F(X)) = G(F(G(X)))$ holds: in this case one can define H as $H(X) = G(F(X))$, and the FGH-rule holds automatically. The PreM rule is more restricted than the FGH-rule, in two ways: the types of the IDBs of the F-program and the H-program must be the same, and the new query H is uniquely defined by F and G, which limits the type of optimizations that are possible under PreM.

Loop Invariants We now describe a more powerful application of the FGH-rule, which uses loop invariants. The general principle is the following. Let $\phi(X)$ be any predicate satisfying the following three conditions:

$$\begin{aligned} &\phi(X_0) \\ &\phi(X) \Rightarrow \phi(F(X)) \\ &\phi(X) \Rightarrow (G(F(X)) = H(G(X))) \end{aligned} \tag{16}$$

then the FG-program (11) and the GH-program (12) are equivalent. This *conditional* FGH-rule is very powerful; we briefly illustrate it with an example.

EXAMPLE 4.4 (BEYOND MAGIC). *Consider the following program:*

$$\Pi_1: \quad \begin{array}{l} TC(x,y) \text{ :- } [x=y] \vee \exists z(E(x,z) \wedge TC(z,y)) \\ Q(y) \text{ :- } TC(a,y) \end{array} \tag{17}$$

which we want to optimize to:

$$\Pi_2: \quad Q(y) \text{ :- } [y=a] \vee \exists z(Q(z) \wedge E(z,y)) \tag{18}$$

Unlike the simple magic program in Ex. 4.2, here rule (17) *is right-recursive. As shown in [6], the magic-set optimization using the standard sideways information passing optimization [1] yields a program that is more complicated than our program* (18)*. Indeed, consider a graph that is simply a directed path* $a_0 \to a_1 \to \cdots \to a_n$ *with* $a = a_0$*. Then, even with magic-set optimization, the right-recursive rule* (17) *needs to derive* quadratically many *facts of the form* $T(a_i, a_j)$ *for* $i \le j$*, whereas the optimized program* (18) *can be evaluated in linear time. Note also that the FGH-rule cannot be applied directly to prove that the program* (17) *is equivalent to* (18)*. To see this, denote by* $P_1 \stackrel{def}{=} G(F(TC))$ *and* $P_2 \stackrel{def}{=} H(G(TC))$*, and observe that* P_1, P_2 *are defined as:*

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

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* $\phi(TC)$*:*

$$\exists z_1(E(x,z_1) \wedge TC(z_1,y)) \Leftrightarrow \exists z_2(TC(x,z_2) \wedge E(z_2,y)) \tag{19}$$

If TC satisfies this predicate, then it follows immediately that $P_1 = P_2$*, allowing us to optimize program* (17) *to* (18)*. It remains to prove that* ϕ *is indeed an invariant for the function F. The base case* (16) *holds because both sides of* (19) *are empty when* $TC = \emptyset$*. It remains to check* $\phi(TC) \Rightarrow \phi(F(TC))$*. Denote* $TC' \stackrel{def}{=} F(TC)$*, then we need to check that, if* (19) *holds, then the predicate* $\Psi_1(x,y) \stackrel{def}{=} \exists z_1(E(x,z_1) \wedge TC'(z_1,y))$ *is equivalent to the predicate* $\Psi_2(x,y) \stackrel{def}{=} \exists z_2(TC'(x,z_2) \wedge E(z_2,y))$*. Using* (19) *we can prove the equivalence of the predicates* Ψ_1 *and* Ψ_2*.*

We describe in [47] how to infer the loop invariant given a program and constraints on the input.

Semantic optimization Finally, we illustrate how the FGH-rule takes advantage of database constraints [38]. In general, a priori knowledge of database constraints can lead to more powerful optimizations. For instance, in [5], 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. We will denote by Γ the set of constraints on the EDBs. Then, the FGH-rule (13) needs to be be checked only for EDBs that satisfy Γ, as we illustrate in this example:

EXAMPLE 4.5. *Consider again the bill-of-material problem in Ex. 3.2.* `SubPart`(x,y) *indicates that y is a subpart of x, and* `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. Recall from Ex. 3.2 that, if we insist on interpreting the program (10) over the natural numbers or reals (and not the lifted naturals* $\mathbb{N}_\perp$ *or lifted reals* $\mathbb{R}_\perp$*), then a cyclic graph will cause the program to diverge. Even if the subpart hierarchy is a DAG, we have to be careful not to double count costs. Therefore, we first compute the transitive closure, and then sum up all costs:*

$$\Pi_1: \quad S(x,y) \text{ :- } [x=y] \vee \exists z\,(S(x,z) \wedge \texttt{SubPart}(z,y)) \qquad (20)$$
$$Q[x] \text{ :- } \sum_y \{\texttt{Cost}[y] \mid S(x,y)\}$$

Consider now the case when our subpart hierarchy is a tree. Then, we can compute the total cost much more efficiently by using the program in Eq. (10)*, repeated here for convenience:*

$$\Pi_2: \quad Q[x] \text{ :- } \texttt{Cost}[x] + \sum_z \{Q[z] \mid \texttt{SubPart}(x,z)\} \qquad (21)$$

Optimizing the program (20) *to* (21) *is an instance of* semantic optimization*, since this only holds if the database instance is a tree. We do this in three steps. First, we define the constraint* Γ *stating that the data is a tree. Second, using* Γ *we infer a loop-invariant* Φ *of the program* Π_1*. Finally, using* Γ *and* Φ *we prove the FGH-rule, concluding that* Π_1 *and* Π_2 *are equivalent. The constraint* Γ *is the conjunction of the following statements:*

$$\forall x_1, x_2, y.\, \texttt{SubPart}(x_1,y) \wedge \texttt{SubPart}(x_2,y) \Rightarrow x_1 = x_2$$
$$\forall x, y.\, \texttt{SubPart}(x,y) \Rightarrow T(x,y)$$
$$\forall x, y, z.\, T(x,z) \wedge \texttt{SubPart}(z,y) \Rightarrow T(x,y) \qquad (22)$$
$$\forall x, y.\, T(x,y) \Rightarrow x \neq y \qquad (23)$$

The first asserts that y is a key in $\texttt{SubPart}(x,y)$*. The last three are an Existential Second Order Logic 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*:*

$$\Phi : S(x,y) \Rightarrow [x=y] \vee T(x,y) \qquad (24)$$

Finally, we check the FGH-rule, under the assumptions Γ, Φ*. Denote by* $P_1 \stackrel{def}{=} G(F(S))$ *and* $P_2 \stackrel{def}{=} H(G(S))$*. To prove* $P_1 = P_2$ *we simplify* P_1 *using the assumptions* Γ, Φ*, as shown in Fig. 4. 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:*

$$\begin{aligned}
& S(x,z) \wedge \texttt{SubPart}(z,y) \\
\Rightarrow & ([x=z] \vee T(x,z)) \wedge \texttt{SubPart}(z,y) && \textit{by } (24) \\
\equiv & \texttt{SubPart}(x,y) \vee (T(x,z) \wedge \texttt{SubPart}(z,y)) \\
\Rightarrow & T(x,y) \vee T(x,y) \equiv T(x,y) && \textit{by } (22) \\
\Rightarrow & x \neq y && \textit{by } (23)
\end{aligned}$$

The last line 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. 4, proving that* $P_1 = P_2$*.*

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

Figure 4: Transformation of $P_1 \stackrel{\text{def}}{=} G(F(S))$ in Ex. 4.5.

4.3 Program Synthesis

In order to use the FGH-rule, the optimizer has to do the following: given the expressions F, G find the new expression H such that $G(F(X)) = H(G(X))$. We will denote $G(F(X))$ and $H(G(X))$ by P_1, P_2 respectively. There are two ways to find H: using rewriting, or using program synthesis with an SMT solver.

Rule-based Synthesis In *query rewriting using views* we are given a query Q and a view V, and want to find another query Q' that answers Q by using only the view V instead of the base tables X; in other words, $Q(X) = Q'(V(X))$ [22, 19]. The problem is usually solved by applying rewrite rules to Q, until it only uses the available views. The problem of finding H is an instance of query rewriting using views, and one possibility is to approach it using rewrite rules; for this purpose we used the rule engine egg [48], a state-of-the-art *equality saturation* system [47].

Counterexample-based Synthesis Rule-based synthesis 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 [43], is to generate candidate programs P_2 from a much larger space, then using an SMT solver to verify 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 semantics of the underlying domain.

Rosette We briefly review Rosette [44], the CEGIS system used in our optimizer. The input to Rosette consists of a *specification* and a *grammar*, and the goal is to synthesize a program defined by the grammar 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 (including the loop invariant), 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 naïve attempt, the generator could blindly generate candidates $P_2, P_2', P_2'', \ldots$, until one is accepted by the verifier; this is hopelessly inefficient. A first optimization in CEGIS is that the verifier returns a small counterexample database instance X for each unsuccessful candidate P_2, i.e., $P_1(X) \neq P_2(X)$. When considering a new candidate P_2, the generator checks that $P_1(X_i) = P_2(X_i)$ holds for all previous counterexamples $X_1, X_2, \ldots$, by simply evaluating the queries P_1, P_2 on the small instance X_i. This significantly reduces the search space of the generator. A second optimization is to use the SMT solver itself to generate the next candidate P_2, as follows. We assume that the grammar Σ is non-recursive, and associate a symbolic Boolean variable $b_1, b_2, \ldots$ to each choice of the grammar. The grammar Σ can be viewed now as a 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 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 $P_1(X_i) = P_2(X_i)$, for all previous counterexample instances X_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 provide the specification $\Gamma \models (G(F(X)) = H(G(X)))$ and supply the grammar of Datalog$^\circ$. This does not work, since Rosette uses the SMT solver to check the identity, and 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 could 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 for $\texttt{normalize}(H(G(X)))$ rather than for H, and then specify:

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

Then, we denormalize the result returned by Rosette, in order to extract H. In summary, our CEGIS-approach for FGH-optimization can be visualized as follows:

$$P_1 \xrightarrow{\texttt{normalize}} P_1' \xrightarrow{\text{CEGIS}} P_2' \xrightarrow{\texttt{denormalize}} P_2 \qquad (25)$$

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 refer the reader to [47] for details on how we constructed the grammar Σ.

4.4 Experimental results

We implemented an optimizer for Datalog$^\circ$ programs. 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 [41, 13]; in [47] we also propose new benchmarks that perform standard data analysis tasks. We did not modify any of the three Datalog engines. Our experiments aim to answer the question: *How effective is our source-to-source optimization, given that each system already supports a range of optimizations?*

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 chose 2 open source research systems, BigDatalog [41] and RecStep [13], and an unreleased commercial system X for our experiments. Both BigDatalog and RecStep are multi-core systems. The commercial system X is single core. As we shall discuss, X is the only one that supports all features for our benchmarks. In this paper we cover 3 out of the 7 benchmark programs used in [47]: Ex. 4.4 (BM), Ex. 4.1 (CC), and Single-source Shortest Path (SSSP) from [41]. The real-world datasets twitter, epinions, and wiki are from the popular SNAP collection [30].

Run Time Measurement For each program-dateset pair we measure the runtime of three programs: the original, with the FGH-optimization, and with the FGH-optimization and the generalized semi-naïve transformation (GSN). We report the speedups relative to the original program in Fig. 5. Where the original program timed out after 3 hours, we report the speedup against 3 hours. In some other cases the original program ran out of memory and we mark them with "o.o.m." in the figure. All three systems already perform semi-naïve evaluation on the original program 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.

Findings. Figure 5 shows the results of the first group of benchmarks optimized by the rule-based synthesizer. The optimizer provides significant (up to 4 orders of magnitude) speedup across systems and datasets. In a few cases, for BM and CC on wiki under BigDatalog, and SSSP on wiki under X, the optimization has little effect. This is due to the small size of the wiki dataset: both the optimized and unoptimized programs finish instantly, so the run time is dominated by optimization overhead. 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-naïve evaluation has the same complexity as the naïve evaluation; but the semi-naïve program is more complex and incurs a constant slowdown. GSN has no effect on BM because the program is in the

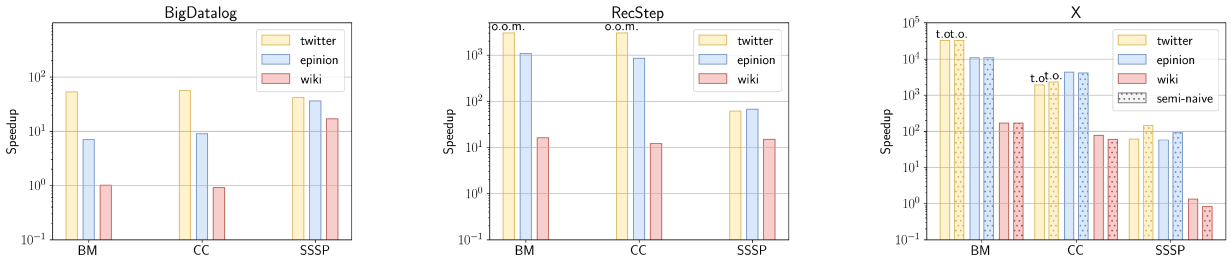

Figure 5: Speedup of the optimized v.s. original program; higher is better; t.o. means the original program timed out after 3h, and we report the speedup against 3 hours; o.o.m. means the original program ran out of memory.

Boolean semiring, and X already implements the standard semi-naïve evaluation. Optimizing BM with FGH on BigDatalog leads to significant speedup, although the system already supports magic-set rewrite, because the optimization depends on a loop invariant. Both the semi-naïve and naïve versions of the optimized program are significantly faster than the unoptimized program.

5. CONCLUSIONS

We presented Datalog°, a recursive language which combines the elegant syntax and semantics of Datalog, with the power of semiring abstraction. This combination allows Datalog° to express iterative computations prevalent in modern data analytics, yet retain the declarative least fixpoint semantics of Datalog. We also presented a novel optimization rule called the FGH-rule, and techniques for optimizing Datalog° by program synthesis using the FGH-rule. Experimental results were presented to validate the theory. There are interesting open problems relating to Datalog° convergence properties and its optimization; we refer to [26, 47] for details.

6. REFERENCES

[1] ABITEBOUL, S., HULL, R., AND VIANU, V. *Foundations of Databases*. Addison-Wesley, 1995.

[2] ALVIANO, M. Evaluating Answer Set Programming with non-convex recursive aggregates. *Fundam. Informaticae* (2016).

[3] ALVIANO, M., FABER, W., LEONE, N., PERRI, S., PFEIFER, G., AND TERRACINA, G. The disjunctive Datalog system DLV. In *Datalog* (2010).

[4] AREF, M., TEN CATE, B., GREEN, T. J., KIMELFELD, B., OLTEANU, D., PASALIC, E., VELDHUIZEN, T. L., AND WASHBURN, G. Design and implementation of the LogicBlox system. In *SIGMOD* (2015).

[5] BANCILHON, F., MAIER, D., SAGIV, Y., AND ULLMAN, J. D. Magic sets and other strange ways to implement logic programs. In *PODS* (1986).

[6] BEERI, C., AND RAMAKRISHNAN, R. On the power of magic. *J. Log. Program.* (1991).

[7] BUDIU, M., MCSHERRY, F., RYZHYK, L., AND TANNEN, V. DBSP: automatic incremental view maintenance for rich query languages. *CoRR abs/2203.16684* (2022).

[8] CARRÉ, B. *Graphs and networks*. The Clarendon Press, Oxford University Press, New York, 1979.

[9] CONWAY, N., MARCZAK, W. R., ALVARO, P., HELLERSTEIN, J. M., AND MAIER, D. Logic and lattices for distributed programming. In *SOCC* (2012).

[10] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (1977).

[11] DANNERT, K. M., GRÄDEL, E., NAAF, M., AND TANNEN, V. Semiring provenance for fixed-point logic. In *CSL* (2021).

[12] FABER, W., PFEIFER, G., AND LEONE, N. Semantics and complexity of recursive aggregates in Answer Set Programming. *Artif. Intell.* (2011).

[13] FAN, Z., ZHU, J., ZHANG, Z., ALBARGHOUTHI, A., KOUTRIS, P., AND PATEL, J. M. Scaling-up in-memory Datalog processing: Observations and techniques. *Proc. VLDB Endow.* (2019).

[14] GANGULY, S., GRECO, S., AND ZANIOLO, C. Minimum and maximum predicates in logic programming. In *PODS* (1991).

[15] GELDER, A. V. The alternating fixpoint of logic programs with negation. In *PODS* (1989).

[16] GELDER, A. V., ROSS, K. A., AND SCHLIPF, J. S. The well-founded semantics for general logic programs. *J. ACM* (1991).

[17] GELFOND, M., AND LIFSCHITZ, V. The stable model semantics for logic programming. In *Logic Programming* (1988).

[18] GELFOND, M., AND ZHANG, Y. Vicious circle principle and logic programs with aggregates. *Theory Pract. Log. Program.* (2014).

[19] GOLDSTEIN, J., AND LARSON, P. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD* (2001).
[20] GONDRAN, M., AND MINOUX, M. *Graphs, dioids and semirings*. Springer, New York, 2008.
[21] GREEN, T. J., KARVOUNARAKIS, G., AND TANNEN, V. Provenance semirings. In *PODS* (2007).
[22] HALEVY, A. Y. Answering queries using views: A survey. *VLDB J.* (2001).
[23] HOPKINS, M. W., AND KOZEN, D. Parikh's theorem in commutative Kleene algebra. In *LICS* (1999).
[24] JORDAN, H., SCHOLZ, B., AND SUBOTIC, P. Soufflé: On synthesis of program analyzers. In *CAV* (2016).
[25] KEMP, D. B., AND STUCKEY, P. J. Semantics of logic programs with aggregates. In *Logic Programming* (1991).
[26] KHAMIS, M. A., NGO, H. Q., PICHLER, R., SUCIU, D., AND WANG, Y. R. Convergence of Datalog over (pre-) semirings. *CoRR abs/2105.14435* (2021).
[27] KHAMIS, M. A., NGO, H. Q., PICHLER, R., SUCIU, D., AND WANG, Y. R. Convergence of datalog over (pre-) semirings. In *PODS* (2022).
[28] KUICH, W. Semirings and formal power series: their relevance to formal languages and automata. In *Handbook of formal languages, Vol. 1*. Springer, Berlin, 1997.
[29] LEHMANN, D. J. Algebraic structures for transitive closure. *Theor. Comput. Sci.* (1977).
[30] LESKOVEC, J., AND KREVL, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.
[31] LIPTON, R. J., ROSE, D. J., AND TARJAN, R. E. Generalized nested dissection. *SIAM J. Numer. Anal.* (1979).
[32] LIPTON, R. J., AND TARJAN, R. E. Applications of a planar separator theorem. *SIAM J. Comput.* (1980).
[33] LIU, Y. A., AND STOLLER, S. D. Recursive rules with aggregation: A simple unified semantics. In *LFCS* (2022).
[34] MAZURAN, M., SERRA, E., AND ZANIOLO, C. A declarative extension of horn clauses, and its significance for Datalog and its applications. *Theory Pract. Log. Program.* (2013).
[35] MUMICK, I. S., PIRAHESH, H., AND RAMAKRISHNAN, R. The magic of duplicates and aggregates. In *VLDB* (1990).
[36] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of program analysis*. Springer-Verlag, Berlin, 1999.
[37] NOCEDAL, J., AND WRIGHT, S. J. *Numerical Optimization*. Springer, 1999.
[38] RAMAKRISHNAN, R., AND SRIVASTAVA, D. Semantics and optimization of constraint queries in databases. *IEEE Data Eng. Bull.* (1994).
[39] ROSS, K. A., AND SAGIV, Y. Monotonic aggregation in deductive databases. In *PODS* (1992).
[40] ROTE, G. Path problems in graphs. In *Computational graph theory*, Comput. Suppl. Springer, Vienna, 1990.
[41] SHKAPSKY, A., YANG, M., INTERLANDI, M., CHIU, H., CONDIE, T., AND ZANIOLO, C. Big data analytics with Datalog queries on spark. In *SIGMOD* (2016).
[42] SHKAPSKY, A., YANG, M., AND ZANIOLO, C. Optimizing recursive queries with monotonic aggregates in DeALS. In *ICDE* (2015).
[43] SOLAR-LEZAMA, A., TANCAU, L., BODÍK, R., SESHIA, S. A., AND SARASWAT, V. A. Combinatorial sketching for finite programs. In *ASPLOS* (2006).
[44] TORLAK, E., AND BODÍK, R. Growing solver-aided languages with Rosette. In *Onward!* (2013).
[45] VIANU, V. Datalog unchained. In *PODS* (2021).
[46] WANG, Y. R., KHAMIS, M. A., NGO, H. Q., PICHLER, R., AND SUCIU, D. Optimizing recursive queries with progam synthesis. In *SIGMOD* (2022).
[47] WANG, Y. R., KHAMIS, M. A., NGO, H. Q., PICHLER, R., AND SUCIU, D. Optimizing recursive queries with program synthesis. *CoRR abs/2202.10390* (2022).
[48] WILLSEY, M., NANDI, C., WANG, Y. R., FLATT, O., TATLOCK, Z., AND PANCHEKHA, P. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, POPL (2021).
[49] ZANIOLO, C., DAS, A., GU, J., LI, Y., LI, M., AND WANG, J. Monotonic properties of completed aggregates in recursive queries. *CoRR abs/1910.08888* (2019).
[50] ZANIOLO, C., YANG, M., DAS, A., AND INTERLANDI, M. The magic of pushing extrema into recursion: Simple, powerful Datalog programs. In *AMW* (2016).
[51] ZANIOLO, C., YANG, M., DAS, A., SHKAPSKY, A., CONDIE, T., AND INTERLANDI, M. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory Pract. Log. Program.* (2017).
[52] ZIMMERMANN, U. Linear and combinatorial optimization in ordered algebraic structures. *Ann. Discrete Math.* (1981).