Optimizing Recursive Queries with Program Synthesis

Yisu Remy Wang*
University of Washington
relationalAI
USA

Mahmoud Abo Khamis
relationalAI
USA

Hung Q. Ngo
relationalAI
USA

Reinhard Pichler*
TU Wien
Austria

Dan Suciu*
University of Washington
relationalAI
USA

ABSTRACT

Most work on query optimization has concentrated on loop-free queries. However, data science and machine learning workloads today typically involve recursive or iterative computation. In this work, we propose a novel framework for optimizing recursive queries using methods from program synthesis. In particular, we introduce a simple yet powerful optimization rule called the "FGH-rule" which aims to find a faster way to evaluate a recursive program. The solution is found by making use of powerful tools, such as a program synthesizer, an SMT-solver, and an equality saturation system. We demonstrate the strength of the optimization by showing that the FGH-rule can lead to speedups up to 4 orders of magnitude on three, already optimized Datalog systems.

CCS CONCEPTS

• **Information systems** → **Query optimization**.

KEYWORDS

Datalog; Recursive Aggregate; Program Synthesis; Semirings

ACM Reference Format:
Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing Recursive Queries with Program Synthesis. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517827

1 INTRODUCTION

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.

Datalog is a language designed specifically for recursive queries, and it is gaining in popularity [3, 12, 14, 22, 35, 37, 38, 48, 49]. But the optimization problem for recursive queries is much less studied. A datalog program consists of multiple rules, defining several, mutually recursive relations, and one distinguished relation name which is the output of the program. The effect of the program consist of repeatedly applying the rules, sometimes called the *body* of the program, until a fixpoint is reached, then it returns the output relation. 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 paper 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. 8).

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

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3517827

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

(a)

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

(b)

Figure 1: Unoptimized (a) and optimized (b) Datalog program for the connected components of an undirected graph.

For a taste of the FGH-optimization, consider the following example, from [54, 55]: compute the connected components of an undirected graph $E(x, y)$. The Datalog program in Fig. 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. 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 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 [21], and today's database systems perform it using pattern matching [16]. 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 paper we propose to use *counterexample-guided inductive synthesis* (CEGIS) for this purpose, which is a technique designed for program sketching [43, 46]. 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. 8 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. This ensures that every query has a well-defined semantics, namely the least fixpoint of its immediate consequence operator. Existing optimizations for Datalog, like semi-naive evaluation and magic set rewriting, apply mainly to monotone queries. Even stratified negation can (if at all) only be handled by imposing appropriate restrictions [44]. 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. 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. 1 (b), because it combines recursion and aggregation.[1] In order to express such queries, in this paper we propose an extension of Datalog, following the approach in [18], where the relations are interpreted over *ordered semirings*.

A semiring is an algebraic structure with two operations, $\oplus, \otimes$. Traditional Datalog corresponds to the Boolean semiring, where these two operators are $\vee, \wedge$, while the query in Fig. 1 (b) is over the Tropical semiring, where the two operators are min, + (reviewed in Sec. 2). We call this extension of Datalog to ordered semirings Datalog°, pronounced "Datalogo", where the circle represents the semiring. In Datalog° recursion is still restricted to monotone[2] queries, but monotone queries in Datalog° include queries with aggregates, over an appropriate semiring. The query in Fig. 1 (b) is monotone over the (ordered) tropical semiring.

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° 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 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* [32]. SQL optimizers routinely use key constraints and foreign key constraints to optimize queries. More powerful optimizations can be performed using the chase and backchase framework [10, 30], 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. 3.3.

[1] Prior work [15, 37] has proposed extending Datalog with min and max aggregates by explicitly re-defining the semantics of recursive rules with aggregates. Our approach keeps the standard least fixpoint semantics, but generalizes the semiring.
[2] This monotonicity is over the partial order from the ordered semiring.

Equality Saturation Systems Throughout our optimizer we need to manage symbolic expressions of queries, and their equivalence classes, as defined by a set of rules. We uses for this purpose a state-of-the-art Equality Saturation System (EQSAT), EGG [53]. We show how to use EQSAT for checking equality under constraints, inferring loop invariants, and "denormalization" (which is essentially query rewriting using views).

Related Work Our work was partially inspired by the PreM condition, described by Zaniolo et al. [54], 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 [8, 19, 36, 47, 50]; but none of these consider recursive queries. Datalog synthesizers have been described in [2, 31, 41, 42, 51]. 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.

Contributions In summary, the main contribution of this paper 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. 3).
- We show the FGH-rule captures known optimizations (magic sets, PreM, semi-naive), (Sec. 3.1), new optimizations (Sec. 3.2), and optimizations under global constraints (Sec. 3.3).
- We present our novel framework for query optimization via the FGH-rule (Sec. 4).
- We describe how an SMT solver (Sec. 5) and a CEGIS system (Sec. 6) 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).

2 BACKGROUND

Datalog A *relation* of arity k is a finite subset of D^k, where D is a fixed domain. The abbreviations EDB and IDB stand for *Extensional Database* and *Intensional Database*, and represent the base relations and the computed relations respectively. A *rule* has the form:

$$R_0(\texttt{vars}) \text{ :- } R_1(\texttt{vars}_1) \wedge \cdots \wedge R_m(\texttt{vars}_m)$$

where R_0 is an IDB, and $R_1, \ldots, R_m$ are IDBs or EDBs. The rule is *safe* if every variable occurs in at least some predicate in the body, and the rule is *linear* if its body contains at most one IDB. A *Datalog program* consists of a set of possibly mutually recursive rules. Usually, only a subset of the IDB predicates are returned to the user, and we will call them the *answer* IDBs. The *Immediate Consequence Operator*, ICO, is the mapping on the IDB predicates that consists of one application of all the Datalog rules. The *semantics* of a Datalog program is given by the least fixpoint of its ICO. The *naive evaluation algorithm* consists of repeatedly applying the ICO until the IDBs no longer change.

In this paper we will combine multiple rules with the same head into a single rule by OR-ing their bodies, and writing explicitly all existential quantifiers. This is a common convention used in the literature, see e.g., [13]. For example the following datalog program, which computes the transitive closure of a relation E,

$$TC(x, y) \text{ :- } E(x, y)$$
$$TC(x, y) \text{ :- } E(x, z) \wedge TC(z, y)$$

becomes $TC(x, y)$:- $E(x, y) \vee \exists z(E(x, z) \wedge TC(z, y))$.

(Pre-)Semirings A *pre-semiring* is a tuple $\boldsymbol{S} = (S, \oplus, \otimes, \bar{0}, \bar{1})$ where $\oplus$ is commutative, both $\oplus, \otimes$ are associative, have identities $\bar{0}$ and $\bar{1}$ respectively, and $\otimes$ distributes over $\oplus$. When $\otimes$ is commutative, then we call $\boldsymbol{S}$ a *commutative* pre-semiring. All pre-semirings in this paper are commutative, and we will simply refer to them as pre-semirings. When the equality $x \otimes \bar{0} = \bar{0}$ holds for all x, then it is called a *semiring*. An *ordered* pre-semiring is a pre-semiring with a partial order $\preceq$, where both $\oplus, \otimes$ are monotone operations. When the partial order is defined by $x \preceq y$ iff $\exists z, x \oplus z = y$ then it is called the *natural order*. Examples of ordered (pre-)semirings are the Booleans $\mathbb{B} = (\{0, 1\}, \vee, \wedge, 0, 1)$, the closed natural numbers $\mathbb{N}^\infty = (\mathbb{N} \cup \{\infty\}, +, *, 0, 1)$, the tropical semiring $\text{Trop} = (\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$, the reversed tropical semiring $\text{Trop}^r = (\mathbb{N}, \max, +, 0, 0)$, the lifted naturals and lifted reals $\mathbb{N}_\perp = (\mathbb{N} \cup \{\perp\}, +, *, 0, 1)$, $\mathbb{R}_\perp = (\mathbb{R} \cup \{\perp\}, +, *, 0, 1)$, where $\perp + x = \perp * x = \perp$. The structures $\mathbb{B}, \mathbb{N}^\infty$, Trop are semirings, the others are pre-semirings. $\mathbb{B}, \mathbb{N}^\infty$, Trop, and Trop^r are naturally ordered. Confusingly (!!), the order relation on Trop is the reverse one: ∞ is the smallest, and 0 is the largest element. The order relation in $\mathbb{N}_\perp$ and $\mathbb{R}_\perp$ is given by $\perp \preceq x$ for all x: they are ordered pre-semirings but not naturally ordered.[3]

***S*-relations** An $\boldsymbol{S}$-relation R is a function that associates to each tuple $t \in D^k$ a value in the semiring, $R[t] \in \boldsymbol{S}$. In this context, $\boldsymbol{S}$ is called the *value space* of the relation R, while the domain D of its attributes is called the *key space*. $\boldsymbol{S}$-relations were first introduced[4] by Green et al. [18] in order to model data provenance. A $\mathbb{B}$-relation is a set, an $\mathbb{N}^\infty$-relation is a bag (with possibly infinite multiplicities), an $\mathbb{R}_\perp$-relation is a tensor (with possibly undefined entries).

Queries Consider a relational schema $R_1, R_2, \ldots$ over a pre-semiring $\boldsymbol{S}$. A positive (relational algebra) *query* is a relational algebra expression using selections, projections, joins, and unions (no difference operator in the positive fragment). The most common definition of the relational algebra restricts the predicates used in selections to equality predicates, $x = y$. In this paper we follow [18] and allow arbitrary predicates $p(x, y, \ldots)$ over the value space, including disequality $x \neq y$, inequality $x < y$, or any other interpreted predicate. Green [18] showed that positive relational algebra extends naturally to an arbitrary semiring $\boldsymbol{S}$. When $\boldsymbol{S}$ is the Boolean semiring, then this coincides with the set semantics of relational algebra, and when $\boldsymbol{S}$ is the semiring of natural numbers, then it coincides with bag semantics.

[3] Note that we define Trop and Trop^r over the natural numbers rather than the reals. The motivation for this slight deviation from the standard definition of these semirings will become clear in Section 5: the support of integer theories by the SMT-solver z3.
[4] Under the name K-relations.

Normal Forms Alternatively, a query can be described using rules, as follows. A *sum-product* query is an expression

$$T(x_1,\ldots,x_k) \text{ :- } \bigoplus_{x_{k+1},\ldots,x_p \in D} A_1 \otimes \cdots \otimes A_m \tag{1}$$

where each A_u is a *relational atom* of the form $R_i(x_{t_{1_i}},\ldots,x_{t_{k_i}})$, or some interpreted predicate such as $x_i > 5x_j + 3$. The variables $x_1,\ldots,x_k$ are called *free variables*, or *head variables*, and the others are called *bound variables*. A *sum-sum-product* query has the form:

$$Q(x_1,\ldots,x_k) \text{ :- } T_1(x_1,\ldots,x_k) \oplus \cdots \oplus T_q(x_1,\ldots,x_k) \tag{2}$$

where $T_1, T_2, \ldots, T_q$ are sum-product expressions with the same head variables $x_1,\ldots,x_k$. When the semiring is $\mathbb{B}, \mathbb{N}^\infty$ and the interpreted predicates are restricted to equality predicates, these queries are (Unions of) Conjunctive Queries (UCQs) under set semantics, or under bag semantics; when the semiring is $\mathbb{R}_\perp$, the sum-products are tensor expressions, sometimes called *Einsum expressions* [34]. Every positive relational algebra query Q can be converted into a sum-sum-product expression, which we call the *normal form* of Q.

Datalog° Let $\boldsymbol{S}$ be an ordered pre-semiring. A Datalog° program consists of a set of (possibly recursive) sum-sum-product rules (2) over $\boldsymbol{S}$-relations. We allow two extensions to the expressions (1) and (2): the summation in (1) may be restricted by some Boolean predicate, and we also allow an atom A in (1) to be an *interpreted function*. One important interpreted function is the cast operator $[-]_{\bar{0}}^{\bar{1}} : \mathbb{B} \to \boldsymbol{S}$, which maps 0 to $\bar{0}$ and 1 to $\bar{1}$ and therefore, for any predicate P, $[P]_{\bar{0}}^{\bar{1}}$ is an atom in the pre-semiring $\boldsymbol{S}$. For example, $[x < y]_{\bar{0}}^{\bar{1}}$ is $\bar{0} \in \boldsymbol{S}$ when $x \geq y$ and $\bar{1} \in \boldsymbol{S}$ when $x < y$; when $\bar{0}, \bar{1}$ are clear from the context, we drop them and write simply $[x < y]$. We treat interpreted functions in a similar way to negation in standard Datalog, and require a program to be *stratified*, such that the interpreted functions are applied only to EDBs or to IDBs defined in earlier strata. This implies that the ICO of that stratum is a monotone function in the IDBs defined by that stratum, and its semantics is defined as its least fixpoint. Abo Khamis et al. [23] proved that any Datalog° program over the semirings discussed in this section (except for $\mathbb{N}^\infty$ and Trop^r) converges in polynomial time in the size of the input database.

Example 2.1. Consider the body of the rule in Fig. 1(b). The relations L, CC are over the tropical semiring, while E is over the Boolean semiring. Formally, its body is a sum-sum-product expression, with a Boolean predicate:

$$L[x] \oplus \bigoplus_y \{CC[y] \mid E(x,y)\}$$

Here the summation $\bigoplus_y$ is restricted to those values y that satisfy the predicate $E(x,y)$. Equivalently, we can rephrase it as:

$$L[x] \oplus \bigoplus_y \left(CC[y] \otimes [E(x,y)]_\infty^0\right)$$

where $[-]_\infty^0$ is the cast operator from $\mathbb{B}$ to Trop; it maps $0, 1$ to $\infty, 0$ respectively. Alternatively, suppose that we represent a label $v = L[x]$ using a standard, Boolean-valued relation $L(x,v)$, where x is a key, and v is the numerical value (label). Then, instead of the atom $L[x]$ we would write $\bigoplus_v \{v \mid L(x,v)\}$, or $\bigoplus_v (v \otimes [L(x,v)]_\infty^0)$. Here v is considered to be an atom.

3 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° 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:

$$\begin{aligned} &X \leftarrow X_0 \\ &\texttt{loop } X \leftarrow F(X) \texttt{ end loop} \\ &Y \leftarrow G(X) \end{aligned} \tag{3}$$

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:

$$\begin{aligned} &Y \leftarrow G(X_0) \\ &\texttt{loop } Y \leftarrow H(Y) \texttt{ end loop} \end{aligned} \tag{4}$$

Theorem 3.1 (The FGH-Rule). *If the following identity holds:*

$$G(F(X)) = H(G(X)) \tag{5}$$

then the FG-program (3) *is equivalent to the GH-program* (4).

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 paper 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 (3) to the initial state Y_0 of (4). 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. (5) holds, then the following Datalog° programs Π_1, Π_2 return the same answer Y:

$$\begin{array}{lll} \Pi_1: & X \text{ :- } F(X) & \Pi_2: \quad Y \text{ :- } H(Y) \\ & Y \text{ :- } G(X) & \end{array} \tag{6}$$

More generally, however, the theorem does not care about the termination condition of the FG-programs (3). 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 3.2. *If the FG-program reaches a fixpoint after n steps (meaning: $X_n = X_{n+1}$) then the GH-program also reaches a fixpoint*

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

Figure 2: Computing CC_1 and CC_2 from Example 3.3.

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. 4. 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.

3.1 Simple Examples

Example 3.3 (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., [55]. The program is given in Fig. 1 (a), and its optimized version in Fig. 1 (b). The three transformations F, G, H are as follows:

$$
\begin{aligned}
&F(TC) \stackrel{\text{def}}{=} TC' \quad \text{where} \quad TC'(x,y) \stackrel{\text{def}}{=} [x=y] \vee \exists z(E(x,z) \wedge TC(z,y))\\
&G(TC) \stackrel{\text{def}}{=} CC \quad \text{where} \quad CC[x] \stackrel{\text{def}}{=} \min_y\{L[y] \mid TC(x,y)\}\\
&H(CC) \stackrel{\text{def}}{=} CC' \quad \text{where} \quad CC'[x] \stackrel{\text{def}}{=} \min(L[x], \min_y\{CC[y] \mid E(x,y)\})
\end{aligned}
$$

To check the FGH-rule, we compute $CC_1 \stackrel{\text{def}}{=} G(F(TC)) = G(TC')$, then compute $CC_2 \stackrel{\text{def}}{=} H(G(TC)) = H(CC)$, both shown in Fig. 2, and observe that it becomes identical to CC_1 after renaming the variables y', y to y, z respectively.

Example 3.4 (PreM Property). Zaniolo et al. [54] 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 3.5 (Simple Magic). The simplest application of magic set optimization [5, 28, 29] converts *transitive closure* to *reachability*.

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

Figure 3: Expressions F, G, H in Example 3.5.

More precisely, it rewrites this program:

$$
\begin{aligned}
\Pi_1: \quad & TC(x,y) \text{ :- } [x=y] \vee \exists z(TC(x,z) \wedge E(z,y))\\
& Q(y) \text{ :- } TC(a,y) \qquad (7)
\end{aligned}
$$

where a is some constant, into this program:

$$
\Pi_2: \quad Q(y) \text{ :- } [y=a] \vee \exists z(Q(z) \wedge E(z,y)) \qquad (8)
$$

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) is equivalent to (8) 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, 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 the full version of this paper that, given a sideways information passing strategy (SIPS) [6] every magic set optimization [4] over a Datalog program can be proven correct using a sequence of applications of the FGH-rule.

Example 3.6 (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° 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. 6) 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.

3.2 Loop Invariants

More advanced uses of the FGH-rule require a loop-invariant, $\phi(X)$. By refining Theorem 3.1 with a loop invariant we obtain the following corollary:

Corollary 3.7. *Let $\phi(X)$ be any predicate satisfying the following three conditions:*

$$\phi(X_0) \tag{9}$$
$$\phi(X) \Rightarrow \phi(F(X)) \tag{10}$$
$$\phi(X) \Rightarrow (G(F(X)) = H(G(X))) \tag{11}$$

then the FG-program (3) *is equivalent to the GH-program* (4).

To prove the corollary, we consider the restriction of the function F to values X that satisfy ϕ. Conditions (9) and (10) state that ϕ is a loop invariant for the FG-program (3), while condition (11) is the FGH-rule applied to the restriction of F to ϕ.

Example 3.8 (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:

$$\begin{aligned} \Pi_1: \quad & TC(x,y) \text{ :- } [x=y] \vee \exists z(E(x,z) \wedge TC(z,y)) \\ & Q(y) \text{ :- } TC(a,y) \end{aligned} \tag{12}$$

which we want to optimize to:

$$\Pi_2: \quad Q(y) \text{ :- } [y=a] \vee \exists z(Q(z) \wedge E(z,y)) \tag{13}$$

Unlike the simple magic program in Example 3.5, here rule (12) 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 (13). 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 (12) needs to derive *quadratically many* facts of the form $T(a_i, a_j)$ for $i \le j$, whereas the optimized program (13) can be evaluated in linear time. Note also that the FGH-rule cannot be applied directly to prove that the program (12) is equivalent to (13). 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 $\phi(TC)$:

$$\exists z_1(E(x,z_1) \wedge TC(z_1,y)) \Leftrightarrow \exists z_2(TC(x,z_2) \wedge E(z_2,y)) \tag{14}$$

If TC satisfies this predicate, then it follows immediately that $P_1 = P_2$, allowing us to optimize the program (12) to (13). It remains to prove that ϕ is indeed an invariant for the function F. The base case (9) holds because both sides of (14) are empty when $TC = \emptyset$. It remains to check $\phi(TC) \Rightarrow \phi(F(TC))$. Let us denote

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

Figure 4: Predicates Ψ_1 and Ψ_2 from Example 3.8.

$TC' \stackrel{\text{def}}{=} F(TC)$, then we need to check that, if (14) 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. 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 (14).

3.3 Semantic Optimization Under Constraints

Semantic optimization refers to optimization rules that hold when the database satisfies certain constraints [32]. 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 [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.

Let Γ denote a set of constraints on the EDBs. Then, the FGH-rule (5) needs to be be checked only for EDBs that satisfy Γ. We illustrate this with an example:

Example 3.9 (Semantic Optimization). Consider a hierarchy of subparts consisting of two relations: $\texttt{SubPart}(x,y)$ 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:

$$\begin{aligned} \Pi_1: \quad & S(x,y) \text{ :- } [x=y] \vee \exists z\,(S(x,z) \wedge \texttt{SubPart}(z,y)) \\ & Q[x] \text{ :- } \sum_y \{\texttt{Cost}[y] \mid S(x,y)\} \end{aligned} \tag{15}$$

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:

$$\Pi_2: \quad Q[x] \text{ :- } \texttt{Cost}[x] + \sum_z \{Q[z] \mid \texttt{SubPart}(x,z)\} \tag{16}$$

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

Figure 5: Transformation of $P_1 \stackrel{\text{def}}{=} G(F(S))$ in Example 3.9.

Optimizing the program (15) to (16) 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:

$$\forall x_1, x_2, y\,(\texttt{SubPart}(x_1, y) \wedge \texttt{SubPart}(x_2, y) \Rightarrow x_1 = x_2) \quad (17)$$

$$\forall x, y\,(\texttt{SubPart}(x, y) \Rightarrow T(x, y)) \quad (18)$$

$$\forall x, y, z\,(T(x, z) \wedge \texttt{SubPart}(z, y) \Rightarrow T(x, y)) \quad (19)$$

$$\forall x, y\,(T(x, y) \Rightarrow x \neq y) \quad (20)$$

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:

$$\Phi : S(x, y) \Rightarrow [x = y] \vee T(x, y) \quad (21)$$

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. 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:

$$
\begin{aligned}
S(x,z) \wedge \texttt{SubPart}(z,y) &\Rightarrow ([x=z] \vee T(x,z)) \wedge \texttt{SubPart}(z,y) && \text{by (21)} \\
&\equiv \texttt{SubPart}(x,y) \vee (T(x,z) \wedge \texttt{SubPart}(z,y)) \\
&\Rightarrow T(x,y) \vee T(x,y) && \text{by (19)} \\
&\equiv T(x,y) \\
&\Rightarrow x \neq y && \text{by (20)}
\end{aligned}
$$

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. 5, proving that $P_1 = P_2$.

4 ARCHITECTURE OF FGH-OPTIMIZATION

In the rest of the paper we describe our synthesis-based FGH-optimizer, whose architecture is shown in Fig. 6. We optimize one stratum at a time. We denote by Π_1 one stratum of the 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. (6). 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. Next, the optimizer needs to find H such that

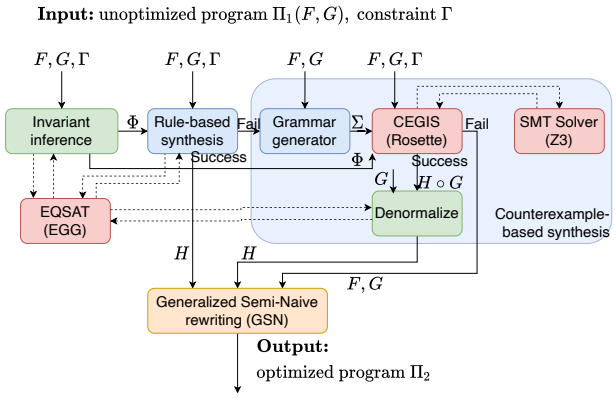

Figure 6: The architecture of the FGH-optimizer. The input is the unoptimized program Π_1, consisting of the functions F, G and the database constraint Γ. The output consists of the optimized program Π_2, see Eq. (6). Blue boxes are described in Section 6 and the green boxes in Section 7. The yellow box (generalized semi-naive optimization) is described in Section 3.1. The red boxes represent three state-of-the-art systems: Rosette is a CEGIS system [43, 45, 46], z3 is an SMT solver [9], and EGG is an EQSAT system [53].

$\Gamma \wedge \Phi \models (G(F(X)) = H(G(X)))$. 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. 6. 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. 3.1. 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* [16, 21]. 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 [26], in NP for UCQs with bag semantics[5], and undecidable for realistic SQL queries that include aggregates and arithmetic [16]. 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 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 [43] Solar-Lezama proposed an alternative to rule-based transformation, called *Counterexample-Guided Inductive Synthesis*, CEGIS: the synthesizer produces potentially incorrect candidates,

[5]This follows from the fact that, under bag semantics, two UCQ queries are equivalent iff they are isomorphic. [17, 52].

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

Figure 7: CC_1 and CC_2 in semiring notation; their normal forms are isomorphic.

and an SMT solver verifies their correctness. In the FGH-optimizer we use a program synthesizer, Rosette [45], 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. 5. Then we describe the synthesizer in Sec. 6.

5 VERIFICATION

We introduced the FGH-rule in Sec. 3 and showed several examples. In order to apply the rule, one needs to check the identity (5), $F(G(X)) = G(H(X))$. In this section we describe how we verify this identity. This step is implicit in both boxes *Rule-based Synthesis* and CEGIS in Fig 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.

5.1 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. (2)) 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. This test can be visualized as follows:

$$P_1 \xrightarrow{\text{axioms}} \text{normalize}(P_1) \simeq \text{normalize}(P_2) \xleftarrow{\text{axioms}} P_2 \qquad (22)$$

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 [17, 52]. 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 3.3 (with semiring notation in Figure 7) one needs all three axioms below:

$$\bigoplus_x \bigoplus_y (\cdots) = \bigoplus_{x,y} (\cdots) \qquad (23)$$
$$A \otimes \bigoplus_x B = \bigoplus_x A \otimes B \text{ when } x \notin \texttt{fv}(A) \qquad (24)$$
$$\bigoplus_x (A(x) \otimes [x = y]) = A(y) \qquad (25)$$

5.2 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 bound/free variables (as needed in axioms (23)-(25)). The *SMT test* is captured by the following figure:

$$P_1 \xrightarrow{\text{axioms}} \text{normalize}(P_1) \xleftrightarrow{\text{SMT}} \text{normalize}(P_2) \xleftarrow{\text{axioms}} P_2 \qquad (26)$$

Example 5.1 (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:

$$D[x,y] \coloneq \texttt{if } x = y \texttt{ then 0 else } \min_z (D[x,z] + E[z,y])$$
$$Q[x,y] \coloneq \min(D[x,y], 100) \qquad (27)$$

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:

$$Q[x,y] \coloneq \texttt{if } x = y \texttt{ then 0 else } \min\left(\min_z (Q[x,z] + E[z,y]), 100\right) \qquad (28)$$

We show how to verify that (28) is equivalent to (27). 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 desugar, convert to semiring expressions, and normalize, they become:

$$P_1[x,y] = (0 \otimes [x = y]^0_\infty) \oplus \left(\bigoplus_z D[x,z] \otimes E[z,y]\right) \oplus 100$$
$$P_2[x,y] = (0 \otimes [x = y]^0_\infty) \oplus \left(\bigoplus_z D[x,z] \otimes E[z,y]\right) \oplus \left(100 \otimes \bigoplus_z E[z,y]\right) \oplus 100$$

In the normalized expressions we push the summations past the joins, i.e., we apply rule (24) 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 [9], 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 (23)-(25), 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. 6). 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:

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

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. 6. 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 5.2. We now finish Example 5.1. 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[6]:

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

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 $e = e'$.

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

6 SYNTHESIS

We have seen in Sec. 5 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.

6.1 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.

$$P_1 \xrightarrow{\text{axioms}} \text{normalize}(P_1) \xrightarrow{\text{axioms}} P_2 \tag{29}$$

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 6. We describe EQSAT in Sec. 7.

6.2 Counterexample-based Synthesis

The rule-based synthesis (29) 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 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):

$$P_1 \xrightarrow{\text{axioms}} \text{normalize}(P_1) \xrightarrow{\text{CEGIS}} P_2 \tag{30}$$

6.2.1 Brief Overview of CEGIS. We give a brief overview of the CEGIS system, Rosette [45, 46], 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.

6.2.2 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. 5.2, 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 $\text{normalize}(H(G(X)))$ 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. 6, described in Sec. 7. In summary, our CEGIS-approach for FGH-optimization can be visualized as follows:

$$P_1 \xrightarrow{\text{axioms}} \text{normalize}(P_1) \xrightarrow{\text{CEGIS}} \text{normalize}(P_2) \xrightarrow{\text{axioms}} P_2 \quad (31)$$

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. (2),

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

where each $G_i(X)$ is a sum-product expression, Eq. (1), using the IDBs X and/or the EDBs.

To generate $\texttt{normalize}(H(G(X)))$, 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:

$$\begin{aligned}
&\texttt{normalize}(H(G(X))) = \\
&\quad H^{(0)} \oplus \texttt{normalize}(H^{(1)}(G(X))) \oplus \cdots \oplus \texttt{normalize}(H^{(k_{\max})}(G(X)))
\end{aligned}$$

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

Figure 8: Grammar Σ for $\texttt{normalize}(H(G(X)))$, for $k_{\max} = 3$.

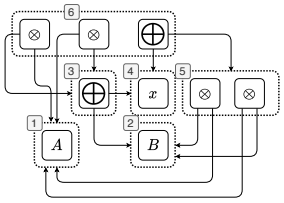

Figure 9: Example e-graph.

The grammar Σ is shown in Fig. 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. 5.2 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. 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 $k_{\max} = 1$.

6.2.3 Discussion. Even though our grammar is restricted to $k_{\max} = 1$, it is more complex than Fig 8, in order to further reduce the search space. We use more non-terminals to better control which variables z 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 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 adopt a state-of-the-art rewriting system dubbed Equality Saturation (EQSAT). Specifically, we used EGG [53] to implement the green boxes in the architecture shown in Fig. 6.

An EQSAT system maintains a data structure called an e-graph that compactly represents a set of expressions, together with an equivalence relation over this set. Each e-graph consists of a set of e-classes, each e-class consists of a set of e-nodes, and each e-node is a function symbol with e-classes as children. Figure 9 shows an e-graph representing the two expressions in Eq. (24), their subexpressions, and other equivalent expressions. Each e-class (dotted box) represents a class of equivalent expressions. For example e-class 5 represents $A \otimes B$ and $B \otimes A$, which are equivalent by commutativity. e-class 6 represents four equivalent expressions (including the two choices in e-class 5).

The EQSAT system maintains separately a collection of *rules*, each represented by a pair of patterns. For example, one rule may state that $\otimes$ is commutative: $x \otimes y = y \otimes x$. The e-graph can efficiently add a new expression to its collection, insert a new rule, and match a given expression against the e-graph.

We describe how we use EGG in the FGH-optimizer. First, we use it to extend the Rule-based test (Sec. 5.1) 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. 6.1 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. (10) from Sec. 3.2.

8 EVALUATION

We implemented a source-to-source FGH-optimizer, based on Fig. 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 [12, 38]; we also propose new benchmarks that perform standard data analysis 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?

8.1 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 [37], Myria [49], the DeALS family of systems (DeALS [40], BigDatalog [39], and RaDlog [20]), RecStep [12], and Dyna [14]. Prior work [38] 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 10. BM and CC are Examples 3.8 and 3.3; MLM is basically Example 3.9. CC, SSSP and MLM are from [38], 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 10. Our benchmark programs are comparable in size to those used in prior work [12, 38]. All programs are available in our git repository. The real-world datasets twitter [27], epinions [33], and wiki [24] are from the popular SNAP collection [25]. We follow the setting in [12, 38] when generating the synthetic graphs. We additionally generate random recursive trees with an exponential decay, modeling the decay of association in multi-level marketing [11]. 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 [12, 38] because some of our experiments run single-threaded.

8.2 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. 11 and 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

Program	Synthesis Type	Constraint?	Invariant?	Dataset	Size (# ops)
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 10: Experimental Setup

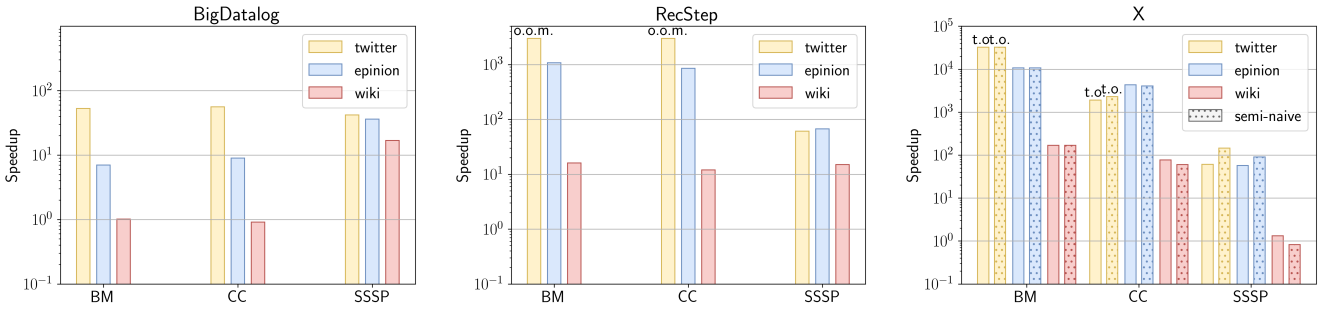

Figure 11: Speedup of the optimized v.s. original program; higher is better; t.o. means the original program timed out after 3 hours, in which case we report the speedup against 3 hours; o.o.m. means the original program ran out of memory.

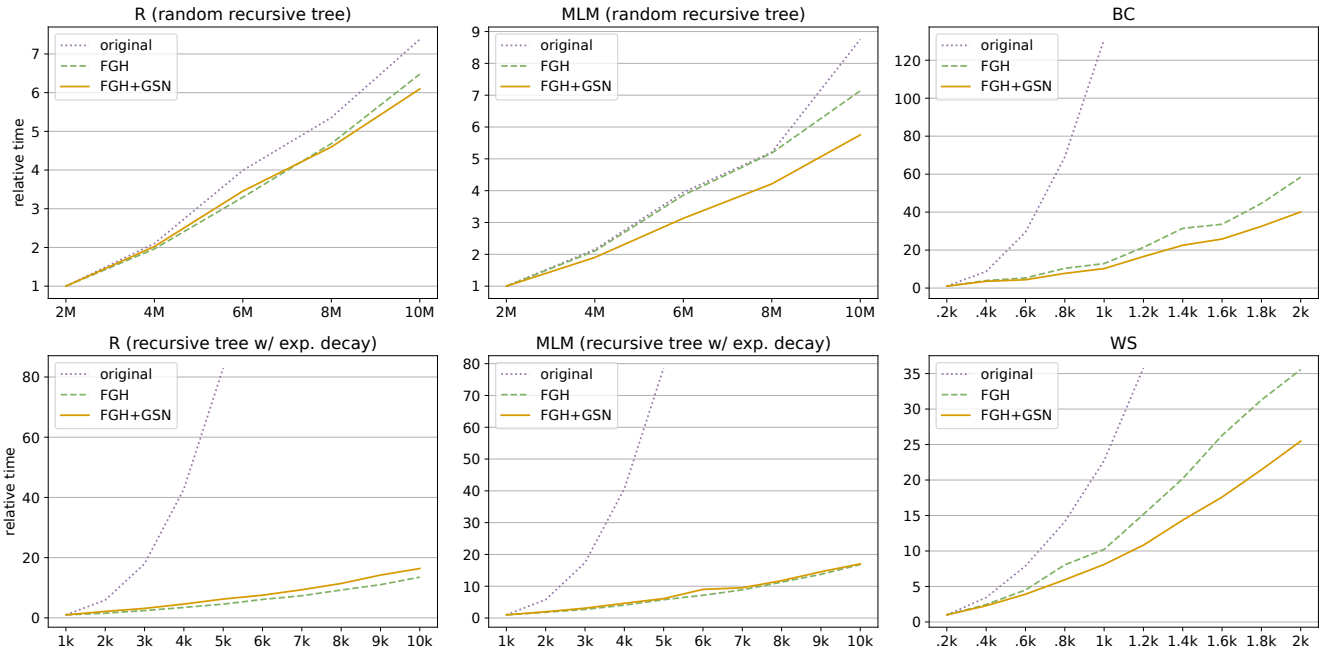

Figure 12: Runtime increase as a function of the data size; lower is better.

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 13: Optimization time in seconds, optimization time over execution time, and size of the search space.

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. 11 were on real datasets, those in Fig. 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. 12 require recursion with SUM aggregation, which is not supported by BigDatalog or RecStep; this is in contrast with those in Fig. 11, which require recursion with MIN aggregation which is supported by all systems.

8.2.1 Findings. Figure 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 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.[7] Overall, both the semi-naive and naive versions of the optimized program are significantly faster than the unoptimized program.

Figure 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 [7]: 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.

8.3 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. 13 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 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.

8.4 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.

9 CONCLUSION

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 paper 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 the paper. 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. 10), 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.

[7] BigDatalog can optimize the left-recursive version of BM (7) to obtain similar speedup, via the classic magic set rewrite.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/

[2] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10416)*, J. Christopher Beck (Ed.). Springer, 689–706. https://doi.org/10.1007/978-3-319-66158-2_44

[3] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2010. Dedalus: Datalog in Time and Space. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6702)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.). Springer, 262–281. https://doi.org/10.1007/978-3-642-24206-9_16

[4] Isaac Balbin, Graeme S. Port, Kotagiri Ramamohanarao, and Krishnamurthy Meenakshi. 1991. Efficient Bottom-UP Computation of Queries on Stratified Databases. *J. Log. Program.* 11, 3&4 (1991), 295–344. https://doi.org/10.1016/0743-1066(91)90030-S

[5] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*, Avi Silberschatz (Ed.). ACM, 1–15. https://doi.org/10.1145/6012.15399

[6] Catriel Beeri and Raghu Ramakrishnan. 1991. On the Power of Magic. *J. Log. Program.* 10, 3&4 (1991), 255–299. https://doi.org/10.1016/0743-1066(91)90038-Q

[7] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[8] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. 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. http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[10] Alin Deutsch, Lucian Popa, and Val Tannen. 1999. Physical Data Independence, Constraints, and Optimization with Universal Plans. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 459–470. http://www.vldb.org/conf/1999/P44.pdf

[11] Yuval Emek, Ron Karidi, Moshe Tennenholtz, and Aviv Zohar. 2011. Mechanisms for multi-level marketing. In *Proceedings of the 12th ACM conference on Electronic commerce*. 209–218.

[12] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. 2019. Scaling-Up In-Memory Datalog Processing: Observations and Techniques. *Proc. VLDB Endow.* 12, 6 (2019), 695–708. https://doi.org/10.14778/3311880.3311886

[13] Melvin Fitting. 1991. Bilattices and the Semantics of Logic Programming. *J. Log. Program.* 11, 1&2 (1991), 91–116. https://doi.org/10.1016/0743-1066(91)90014-G

[14] Matthew Francis-Landau, Tim Vieira, and Jason Eisner. 2020. Evaluation of Logic Programs with Built-Ins and Aggregation: A Calculus for Bag Relations. In *13th International Workshop on Rewriting Logic and Its Applications*. 49–63.

[15] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. 1991. Minimum and Maximum Predicates in Logic Programming. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 29-31, 1991, Denver, Colorado, USA*, Daniel J. Rosenkrantz (Ed.). ACM Press, 154–163. https://doi.org/10.1145/113413.113427

[16] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, Sharad Mehrotra and Timos K. Sellis (Eds.). ACM, 331–342. https://doi.org/10.1145/375663.375706

[17] Todd J. Green. 2009. Containment of conjunctive queries on annotated relations. In *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings (ACM International Conference Proceeding Series, Vol. 361)*, Ronald Fagin (Ed.). ACM, 296–309. https://doi.org/10.1145/1514894.1514930

[18] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, Leonid Libkin (Ed.). ACM, 31–40. https://doi.org/10.1145/1265530.1265535

[19] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. 2017. Verifying Equivalence of Spark Programs. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 282–300. https://doi.org/10.1007/978-3-319-63390-9_15

[20] Jiaqi Gu, Yugo H. Watanabe, William A. Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. 2019. RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 467–484. https://doi.org/10.1145/3299869.3324959

[21] Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB J.* 10, 4 (2001), 270–294. https://doi.org/10.1007/s007780100054

[22] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and Emerging Applications: An Interactive Tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 1213–1216. https://doi.org/10.1145/1989323.1989456

[23] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2021. *Convergence of Datalog over (Pre-) Semirings*. arXiv:2105.14435v1 [cs.DB]

[24] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 1361–1370.

[25] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[26] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. 1995. Answering Queries Using Views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, Mihalis Yannakakis and Serge Abiteboul (Eds.). ACM Press, 95–104. https://doi.org/10.1145/212433.220198

[27] Julian J McAuley and Jure Leskovec. 2012. Learning to discover social circles in ego networks.. In *NIPS*, Vol. 2012. Citeseer, 548–56.

[28] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. Magic is Relevant. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 247–258. https://doi.org/10.1145/93597.98734

[29] Inderpal Singh Mumick and Hamid Pirahesh. 1994. Implementation of Magic-sets in a Relational Database System. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, Richard T. Snodgrass and Marianne Winslett (Eds.). ACM Press, 103–114. https://doi.org/10.1145/191839.191860

[30] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. 2000. A Chase Too Far?. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 273–284. https://doi.org/10.1145/342009.335421

[31] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2020. Provenance-guided synthesis of Datalog programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 62:1–62:27. https://doi.org/10.1145/3371130

[32] Raghu Ramakrishnan and Divesh Srivastava. 1994. Semantics and Optimization of Constraint Queries in Databases. *IEEE Data Eng. Bull.* 17, 2 (1994), 14–17. http://sites.computer.org/debull/94JUN-CD.pdf

[33] Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. 2003. Trust management for the semantic web. In *International semantic Web conference*. Springer, 351–368.

[34] Tim Rocktäschel. [n.d.]. Einsum is all you need - Einstein summation in deep learning. https://rockt.github.io/2018/04/30/einsum.

[35] Timothy Roscoe and Boon Thau Loo. 2018. Declarative Networking. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. https://doi.org/10.1007/978-1-4614-8265-9_1220

[36] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. 2017. Optimizing Big-Data Queries Using Program Synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 631–646. https://doi.org/10.1145/3132747.3132773

[37] Jiwon Seo, Stephen Guo, and Monica S. Lam. 2015. SociaLite: An Efficient Graph Query Language Based on Datalog. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1824–1837. https://doi.org/10.1109/TKDE.2015.2405562

[38] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1135–1149. https://doi.org/10.1145/2882903.2915229

[39] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. 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*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1135–1149. https://doi.org/10.1145/2882903.2915229

[40] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). IEEE Computer Society, 867–878. https://doi.org/10.1109/ICDE.2015.7113340

[41] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-guided synthesis of Datalog programs. In *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*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 515–527. https://doi.org/10.1145/3236024.3236034

[42] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. Synthesizing Datalog Programs using Numerical Relaxation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 6117–6124. https://doi.org/10.24963/ijcai.2019/847

[43] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *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*, John Paul Shen and Margaret Martonosi (Eds.). ACM, 404–415. https://doi.org/10.1145/1168857.1168907

[44] K. Tuncay Tekle and Yanhong A. Liu. 2019. Extended Magic for Negation: Efficient Demand-Driven Evaluation of Stratified Datalog with Precise Complexity Guarantees. In *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019 (EPTCS, Vol. 306)*, Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Inclezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang (Eds.). 241–254. https://doi.org/10.4204/EPTCS.306.28

[45] Emina Torlak and Rastislav Bodík. 2013. Growing solver-aided languages with rosette. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 135–152. https://doi.org/10.1145/2509578.2509586

[46] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *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 (Lecture Notes in Computer Science, Vol. 4424)*, Orna Grumberg and Michael Huth (Eds.). Springer, 632–647. https://doi.org/10.1007/978-3-540-71209-1_49

[47] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. 2009. Symbolic Query Exploration. In *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5885)*, Karin K. Breitman and Ana Cavalcanti (Eds.). Springer, 49–68. https://doi.org/10.1007/978-3-642-10373-5_3

[48] Victor Vianu. 2021. Datalog Unchained. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Virtual Event, China) *(PODS'21)*. Association for Computing Machinery, New York, NY, USA, 57–69. https://doi.org/10.1145/3452021.3458815

[49] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. 2015. Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1542–1553. https://doi.org/10.14778/2824032.2824052

[50] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2018. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.* 2, POPL (2018), 56:1–56:29. https://doi.org/10.1145/3158144

[51] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration using Datalog Program Synthesis. *Proc. VLDB Endow.* 13, 7 (2020), 1006–1019. https://doi.org/10.14778/3384345.3384350

[52] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 11 (2020), 1919–1932. http://www.vldb.org/pvldb/vol13/p1919-wang.pdf

[53] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434304

[54] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. 2017. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory Pract. Log. Program.* 17, 5-6 (2017), 1048–1065. https://doi.org/10.1017/S1471068417000436

[55] Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. 2018. Declarative BigData Algorithms via Aggregates and Relational Database Dependencies. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018 (CEUR Workshop Proceedings, Vol. 2100)*, Dan Olteanu and Barbara Poblete (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-2100/paper2.pdf