Evaluating Datalog over Semirings: A Grounding-based Approach

HANGDONG ZHAO, University of Wisconsin-Madison, USA
SHALEEN DEEP, Microsoft, USA
PARASCHOS KOUTRIS, University of Wisconsin-Madison, USA
SUDEEPA ROY, Duke University, USA
VAL TANNEN, University of Pennsylvania, USA

Datalog is a powerful yet elegant language that allows expressing recursive computation. Although Datalog evaluation has been extensively studied in the literature, so far, only loose upper bounds are known on how fast a Datalog program can be evaluated. In this work, we ask the following question: given a Datalog program over a naturally-ordered semiring σ, what is the tightest possible runtime? To this end, our main contribution is a general two-phase framework for analyzing the data complexity of Datalog over σ: first ground the program into an equivalent system of polynomial equations (i.e. grounding) and then find the least fixpoint of the grounding over σ. We present algorithms that use structure-aware query evaluation techniques to obtain the smallest possible groundings. Next, efficient algorithms for fixpoint evaluation are introduced over two classes of semirings: (1) finite-rank semirings and (2) absorptive semirings of total order. Combining both phases, we obtain state-of-the-art and new algorithmic results. Finally, we complement our results with a matching fine-grained lower bound.

CCS Concepts: • **Theory of computation → Database query processing and optimization (theory)**.

Additional Key Words and Phrases: Datalog, Semirings, Evaluation, Grounding

ACM Reference Format:
Hangdong Zhao, Shaleen Deep, Paraschos Koutris, Sudeepa Roy, and Val Tannen. 2024. Evaluating Datalog over Semirings: A Grounding-based Approach. *Proc. ACM Manag. Data* 2, 2 (PODS), Article 90 (May 2024), 26 pages. https://doi.org/10.1145/3651591

1 INTRODUCTION

Datalog is a recursive query language that has gained prominence due to its expressivity and rich applications across multiple domains, including graph processing [43], declarative program analysis [42, 44], and business analytics [23]. Most of the prior work focuses on Datalog programs over the Boolean semiring (this corresponds to the standard relational join semantics): popular examples include same generation [5], cycle finding, and pattern matching. Many program analysis tasks such as Dyck-reachability [31, 32, 41], context-free reachability [36], and Andersen's analysis [4] can also be naturally cast as Datalog programs over the Boolean domain. However, modern data analytics frequently involve aggregations over recursion. Seminal work by Green et al. [24] established the semantics of Datalog over the so-called K-relations where tuples are

Authors' addresses: Hangdong Zhao, Department of Computer Sciences, University of Wisconsin-Madison, Madison, USA, hangdong@cs.wisc.edu; Shaleen Deep, Jim Gray Systems Lab, Microsoft, USA, shaleen.deep@microsoft.com; Paraschos Koutris, Department of Computer Sciences, University of Wisconsin-Madison, Madison, USA, paris@cs.wisc.edu; Sudeepa Roy, Department of Computer Science, Duke University, Durham, USA, sudeepa@cs.duke.edu; Val Tannen, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, USA, val@seas.upenn.edu.

ACM 2836-6573/2024/5-ART90
https://doi.org/10.1145/3651591

annotated by the domain of a fixed semiring. Recently, Abo Khamis et al. [28] proposed Datalog°, an elegant extension of Datalog, and established key algebraic properties that governs convergence of Datalog°. The authors further made the observation that the convergence rate of Datalog can be studied by confining to the class of *naturally-ordered* semirings (formally defined in Section 2).

A parallel line of work has sought to characterize the complexity of Datalog evaluation. The general data complexity of Datalog is P-complete [11, 20], with the canonical P-complete Datalog program:

$$T(x_1) \leftarrow U(x_1).$$
$$T(x_1) \leftarrow T(x_2) \wedge T(x_3) \wedge R(x_2, x_3, x_1).$$

Some fragments of Datalog can have lower data complexity: the evaluation for non-recursive Datalog is in AC_0, whereas evaluation for Datalog with linear rules is in NC and thus efficiently parallelizable [3, 45]. However, all such results do not tell us how efficiently we can evaluate a given Datalog program. As an example, the program (1) can be evaluated in linear time with respect to the input size m [22]. A deeper understanding of precise upper bounds for Datalog is important, given that it can capture practical problems (all of which are in P) across various domains as mentioned before.

Unfortunately, even for the Boolean semiring, the current general algorithmic techniques for Datalog evaluation typically aim for an imprecise polynomial bound instead of specifying the tightest possible exponent. Semi-naïve or naïve evaluation only provides upper bounds on the number of iterations, ignoring the computational cost of each iteration. For example, program (1) can be evaluated in at most $O(m)$ iterations, but the cost of an iteration can be as large as $\Theta(m)$. Further, going beyond the Boolean semiring, obtaining tight bounds for evaluating general Datalog programs over popular semirings (such as absorptive semirings with a total order which are routinely used in data analytics [39, 40]) is unclear. In particular, proving correctness of program evaluation is not immediate and the impact of the semiring on the evaluation time remains unknown.

Endeavors to pinpoint the exact data complexity for Datalog fragments have focused on the class of Conjunctive Queries (CQs) [26, 29, 37] and union of CQs [8], where most have been dedicated to develop faster algorithms. When recursion is involved, however, exact runtimes are known only for restricted classes of Datalog. Seminal work of Yannakakis [50] established a $O(n^3)$ runtime for chain Datalog programs (formally defined in Section 2.3), where n is the size of the active domain. Such programs have a direct correspondence to context-free grammars and capture a fundamental class of static analysis known as context-free reachability (CFL). When the chain Datalog program corresponds to a regular grammar, the runtime can be further improved to $O(m \cdot n)$[1]. An $O(n^3)$ algorithm was proposed for the Datalog program that captures Andersen's analysis [36]. Recently, Casel and Schmid [9] studied the fine-grained complexity of evaluation, enumeration, and counting problems over regular path queries (also a Datalog fragment) with similar upper bounds. However, none of the above techniques generalize to arbitrary Datalog programs.

Our Contribution. In this paper, we ask: given a Datalog program P over a naturally-ordered semiring σ, what is the *tightest possible runtime as a function of the input size m and the size of the active domain n*? Our main contributions are as follows.

We propose a general, two-phased framework for evaluation of P over σ. The first phase *grounds* P into an equivalent system of polynomial equations G. Though constructing groundings naïvely is rather straightforward, we show that groundings of smaller size are attainable via tree decomposition techniques. The second phase evaluates the least fixpoint of G over the underlying semiring σ. We show that finite-rank semirings and absorptive semirings with total order, two routinely-used classes of semirings in practice, admit efficient algorithms for least fixpoint computation. We apply

[1] m denotes the size of the input data and n denotes the size of the active domain for a Datalog program throughout the paper.

our framework to prove state-of-the-art and new results for practical Datalog fragments (e.g. linear Datalog).

Further, we establish tightness of our results by demonstrating a matching lower bound on the running time (conditioned on the popular min-weight ℓ-clique conjecture) and size of the grounded program (unconditional) for a class of Datalog programs.

2 PRELIMINARIES

In this section, we describe the concepts used in the rest of the paper.

2.1 Datalog

We review standard Datalog [1] that consists of a set of *rules* over a set of *extensional* and *intensional* relations (simply referred to as EDB and IDB respectively, henceforth). EDBs correspond to relations in a given database, each comprising a set of EDB tuples, whereas IDBs are derived by the rules. The head of each rule is an IDB, and the body consists of zero or more EDBs and IDBs as a conjunctive query defining how the head IDB is derived. We illustrate with the example of transitive closure on a binary EDB R denoting edges in a directed graph, a single IDB T, and two rules:

$$T(x_1, x_2) \leftarrow R(x_1, x_2), \tag{1}$$

$$T(x_1, x_2) \leftarrow T(x_1, x_3) \wedge R(x_3, x_2) \tag{2}$$

In standard Datalog, IDBs are derived given EDBs (equivalently, evaluation is over the Boolean semiring), whereas as discussed in Section 1, the evaluation can also be considered over other semirings where the EDBs are annotated with elements of that semiring.

2.2 Semirings and Their Properties

Semirings. A tuple $\sigma = (\boldsymbol{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a *semiring* if $\oplus$ and $\otimes$ are binary operators over $\boldsymbol{D}$ for which:

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

σ is a *commutative semiring* if $\otimes$ is commutative. Examples include the Boolean semiring $\mathbb{B} = (\{\mathsf{false}, \mathsf{true}\}, \vee, \wedge, \mathsf{false}, \mathsf{true})$, the natural numbers semiring $(\mathbb{N}, +, \cdot, 0, 1)$, and the real semiring $(\mathbb{R}, +, \cdot, 0, 1)$.

Naturally-Ordered Semirings. In a semiring σ, the relation $x \sqsubseteq y$ (or $y \sqsupseteq x$), defined as $\exists z : x \oplus z = y$, is reflexive and transitive, but not necessarily anti-symmetric. If it is anti-symmetric, then $(\boldsymbol{D}, \sqsubseteq)$ is a poset ($\mathbf{0}$ is the minimum element since $\mathbf{0} \oplus a = a$) and we say that σ is *naturally-ordered*. For example, $\mathbb{N}$ is naturally-ordered with the usual order $\leq$. However, $\mathbb{R}$ is not naturally-ordered, since $x \sqsubseteq y$ for every $x, y \in \mathbb{R}$. We focus on naturally-ordered commutative semirings in this paper (simply referred to as semirings from now).

ω-complete & ω-continuous Semirings. An ω-chain in a poset is a sequence $a_0 \sqsubseteq a_1 \sqsubseteq \dots$ with elements from the poset. A naturally-ordered semiring σ is *ω-complete* if every (infinite) ω-chain $a_0 \sqsubseteq a_1 \sqsubseteq \cdots$ has a least upper bound $\sup a_i$, and is *ω-continuous* if also $\forall a \in \boldsymbol{D}$, $a \oplus \sup a_i = \sup (a \oplus a_i)$ and $a \otimes \sup a_i = \sup (\mathsf{a} \otimes \mathsf{a_i})$.

Rank. The *rank* of a strictly increasing ω-chain $a_0 \sqsubset a_1 \sqsubset \cdots \sqsubset a_r$ is r ($x \sqsubset y$ if $x \sqsubseteq y$ and $x \neq y$). We say that a semiring σ has rank r if every strictly increasing sequence has rank at most r. Any semiring over a finite domain has constant rank (e.g. $\mathbb{B}$ has rank 1).

We highlight two special classes of naturally-ordered semirings to be studied in this paper: dioids and absorptive semirings.

Dioids. A *dioid* σ is a semiring where $\oplus$ is idempotent, i.e. $a \oplus a = a$ for any $a \in \boldsymbol{D}$. A dioid is naturally ordered, i.e. $a \sqsubseteq b$ if $a \oplus b = b$. The natural order satisfies the monotonicity property: for all $a, b, c \in \boldsymbol{D}$, $a \sqsubseteq b$ implies $a \oplus c \sqsubseteq b \oplus c$ and $a \otimes c \sqsubseteq b \otimes c$.

Absorptive Semirings. A semiring σ is *absorptive* if $\mathbf{1} \oplus a = \mathbf{1}$ for every $a \in \boldsymbol{D}$. An absorptive semiring is also called 0-*stable* [28]. An absorptive semiring is a dioid and $\mathbf{0} \sqsubseteq a \sqsubseteq \mathbf{1}$, for any $a \in \boldsymbol{D}$. An equivalent characterization of absorptiveness is in the Appendix of the full version of the paper [52]. Simple examples of absorptive semirings are the Boolean semiring and the *tropical semiring* $\mathsf{Trop}^+ = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$ (that can be used to measure the shortest path), where its natural order $x \sqsubseteq y$ is the reverse order $x \geq y$ on $\mathbb{R}_+ \cup \{\infty\}$.

2.3 Datalog over Semirings

Next we describe the syntax of Datalog over a semiring σ. Here EDBs are considered as σ-relations, i.e., each tuple in each EDB R is annotated by an element from the domain of σ. Tuples not in the EDB are annotated by $\mathbf{0}$ implicitly. Standard relations are essentially $\mathbb{B}$-relations. When a Datalog program is evaluated on such σ-relations, IDB σ-relations are derived. Here, conjunction ($\wedge$) is interpreted as $\otimes$, and disjunction as $\oplus$ of σ, as discussed in [24].

Sum-product Queries. The Datalog program (1) has two rules of the same head, thus an IDB tuple can be derived by either rule (i.e. a disjunction over rules). Following Abo Khamis et al. [28], we combine all rules with the same IDB head into one using disjunction. Hence the program under a semiring σ is written as:

$$T(x_1, x_2) \leftarrow R(x_1, x_2) \oplus \left(\bigoplus_{x_3} T(x_1, x_3) \otimes R(x_3, x_2) \right). \tag{3}$$

where $\oplus$ corresponds to alternative usage (or disjunction) [24] and the implicit $\exists x_3$ in the second rule of (1) is made explicit by $\bigoplus_{x_3}$. Let ℓ be the number of variables $x_1, x_2, \cdots, x_\ell$ in a Datalog program as in Eqn. (3). For $J \subseteq [\ell]$, where $[\ell] = \{1, 2, \ldots, \ell\}$, $\mathbf{x}_J$ denotes the set of variables $\{x_j \mid j \in J\}$. In the rest of the paper, w.l.o.g., we consider a Datalog program as a set of rules of distinct IDB heads, where each rule has the following form:

$$T(\mathbf{x}_H) \leftarrow \varphi_1(\mathbf{x}_H) \oplus \varphi_2(\mathbf{x}_H) \oplus \ldots \tag{4}$$

Here $\mathbf{x}_H$ denotes the set of head variables. The body of a rule is a *sum* (i.e., $\oplus$) over one or more *sum-product queries* (defined below) $\varphi_1(\mathbf{x}_H), \varphi_2(\mathbf{x}_H), \ldots$ corresponding to different derivations of the IDB $T(\mathbf{x}_H)$ over a semiring σ. Formally, a *sum-product (in short, sum-prod) query* φ over σ has the following form:

$$\varphi(\mathbf{x}_H) : \quad \bigoplus_{\mathbf{x}_{[\ell] \setminus H}} \bigotimes_{J \in \mathcal{E}} T_J(\mathbf{x}_J), \tag{5}$$

where (1) $\mathbf{x}_H$ is the set of head variables, (2) each T_J is an EDB or IDB predicate, and (3) $([\ell], \mathcal{E})$ is the *associated hypergraph* of φ with hyperedges $\mathcal{E} \subseteq 2^{[\ell]}$ and vertex set as $\mathbf{x}_{[\ell]}$.

For every Datalog program in the sum-product query form, we will assume that there is a unique IDB that we identify as the *target* (or *output*) of the program. We use $\mathsf{arity}(P)$ to denote the maximum number of variables contained in any IDB of a program P.

Monadic, Linear & Chain Datalog. We say that a Datalog program P is *monadic* if every IDB is unary (i.e. $\mathsf{arity}(P) = 1$); a Datalog program is *linear* if every sum-prod query in every rule contains at most one IDB (e.g., the program in Eq. (3) is linear). A *chain* query is a sum-prod query over

binary predicates as follows:

$$\varphi(x_1, x_{k+1}) : \bigoplus_{\mathbf{x}_{\{2,\ldots,k\}}} T_1(x_1, x_2) \otimes T_2(x_2, x_3) \otimes \cdots \otimes T_k(x_k, x_{k+1}).$$

A *chain* Datalog program is a program where every rule consists of one or a sum of multiple chain queries. A chain Datalog program corresponds to a Context-Free Grammar (CFG) [50].

Least Fixpoint Semantics. The least fixpoint semantics of a Datalog program P over a semiring σ is defined as the standard Datalog, through its immediate consequence operator (ICO). An ICO applies all rules in P exactly once over a given instance and uses all derived facts (and the instance) for the next iteration. The naive evaluation algorithm iteratively applies ICO until a least fixpoint is reached (if any). In general, naive evaluation of Datalog over a semiring σ may not converge, depending on σ. Kleene's Theorem [13] shows that if σ is ω-continuous and the semiring σ is ω-complete, then the naive evaluation converges to the least fixpoint. Following prior work [16, 24], we assume that σ is ω-continuous and ω-complete.

σ-equivalent. Two Datalog programs P, G are σ-equivalent if for any input EDB instance annotated with elements of a semiring σ, the targets of P and G are identical σ-relations when least fixpoints are reached for both.

Parameters & Computational Model. We use m to denote the sum of sizes of the input EDB σ-relations to a Datalog program (henceforth referred to as the "*input size is m*"). We use n to denote the size of the active domain of EDB σ-relations (i.e., the number of distinct constants that occur in the input). If $\text{arity}(P) = k$, then $n \leq m \cdot k$. We assume *data complexity* [47], i.e. the program P size (the total number of predicates and the variables) is a constant. We use the standard word-RAM model with $O(\log m)$-bit words and unit-cost operations [10] for all complexity results. $\widetilde{O}$ hides poly-logarithmic factors in the size of the input.

3 FRAMEWORK AND MAIN RESULTS

This section presents a general framework to analyze the data complexity of Datalog programs. We show how to use this framework to obtain both state-of-the-art and new algorithmic results.

A common technique to measure the runtime of a Datalog program is to multiply the number of iterations until the fixpoint is reached with the cost of each ICO evaluation. Although this method can show a polynomial bound in terms of data complexity, it cannot generate the tightest possible upper bound. Our proposed algorithmic framework allows us to decouple the semiring-dependent fixpoint computation from the structural properties of the Datalog program. It splits the computation into two distinct phases, where the first phase concerns the *logical structure* of the program, and the second the *algebraic structure* of the semiring.

Grounding Generation. In the first phase, the Datalog program is transformed into a σ-equivalent *grounded program*, where each rule contains only constants. A *grounding* of a grounded Datalog program is a system of polynomial equations where we assign to each grounded EDB atom a distinct coefficient e in the semiring and to each grounded IDB atom a distinct variable x. Our goal in this phase is to construct a σ-equivalent grounding G of the smallest possible size $|G|$. The size of G is measured as the total number of coefficients and variables in the system of polynomial equations.

Grounding Evaluation. In the second phase, we need to deal with evaluating the least fixpoint of G over the semiring σ. The fixpoint computation now depends on the structure of the semiring σ, and we can completely ignore the underlying logical structure of the program. Here, we need to construct algorithms that are as fast as possible w.r.t. the size of the grounding $|G|$. For example, it is known that over the Boolean semiring, the least fixpoint of G can be computed in time $O(|G|)$ [22].

Table 1. Summary of the grounding results. The $\widetilde{O}$ notation hides polylog factors in m (total input size) and n (size of the active domain). k denotes the arity of the program. See Section 6 for the definitions of fhw, subw.

Semiring	Class of Programs	IDB Arity	Grounding Size & Time	Result
all	rulewise-acyclic	k	$O(n^{k-1} \cdot (m + n^k))$	Theorem 3.2
all	rulewise free-connex acyclic	k	$O(m + n^k)$	Proposition 5.3
all	linear and rulewise-acyclic	2	$O(n \cdot m)$	Proposition 5.4
all	fractional hypertree-width $\leq$ fhw	k	$O(n^{k-1} \cdot (m^{\text{fhw}} + n^{k \cdot \text{fhw}}))$	Proposition 6.4
dioid	submodular width $\leq$ subw	k	$\widetilde{O}(n^{k-1} \cdot (m^{\text{subw}} + n^{k \cdot \text{subw}}))$	Theorem 3.3
dioid	submodular width $\leq$ subw	1	$\widetilde{O}(m^{\text{subw}})$	Corollary of Theorem 3.3
dioid	free-connex submodular width $\leq$ fsubw	k	$\widetilde{O}(m^{\text{fsubw}} + n^{k \cdot \text{fsubw}})$	Proposition 6.2
dioid	linear and submodular width $\leq$ subw	k	$\widetilde{O}(n^{k-1} m^{\text{subw}-1} \cdot (m + n^k))$	Proposition 6.3

3.1 Grounding Generation

The first phase of the framework generates a grounded program.

Example 3.1. We will use as an example the following variation of program (3), over an arbitrary semiring σ:

$$T(x_1, x_2) \leftarrow R(x_1, x_2) \oplus \left(\bigoplus_{x_3} T(x_1, x_3) \otimes U(x_3) \otimes R(x_3, x_2) \right).$$

We introduced an unary atom U to capture properties of the nodes in the graph. Consider an instance where R contains two edges, (a, b) and (b, c), and U contains three nodes, a, b, c. One possible σ-equivalent grounded Datalog program (over any semiring σ) is:

$$T(a, b) \leftarrow R(a, b) \oplus T(a, a) \otimes U(a) \otimes R(a, b), \qquad T(a, a) \leftarrow \mathbf{0}$$
$$T(a, c) \leftarrow T(a, b) \otimes U(b) \otimes R(b, c), \qquad T(b, c) \leftarrow R(b, c).$$

This corresponds to the following system of polynomial equations:

$$x_{ab} = e_{ab} \oplus x_{aa} \otimes f_a \otimes e_{ab} \qquad x_{aa} = \mathbf{0}$$
$$x_{ac} = x_{ab} \otimes f_b \otimes e_{bc} \qquad x_{bc} = e_{bc}.$$

As there is a one-to-one correspondence between a grounded program and its grounding, we will only use the term grounding from now on. The naive way to generate a σ-equivalent grounded program is to take every rule and replace the variables with all possible constants. However, this may create a grounding of very large size. Our key idea is that we can optimize the size of the generated grounding by exploiting the logical structure of the rules.

Upper Bounds. Our first main result (Section 5) considers the body of every sum-product query in the Datalog program is acyclic (formally Definition 5); we call such a program *rulewise-acyclic*

THEOREM 3.2. *Let P be a rulewise-acyclic Datalog program over some semiring σ, with input size m, active domain size n, and* $\mathsf{arity}(P) \leq k$. *Then, we can construct a σ-equivalent grounding in time (and has size)* $O(n^{k-1} \cdot (m + n^k))$.

A direct result of the above theorem is that for *monadic* Datalog, where $\mathsf{arity}(P) = 1$, we obtain a grounding of size $O(m)$, which is essentially optimal. The main technical idea behind Theorem 3.2 is to construct the join tree corresponding to each rule, and then decompose the rules following the structure of the join tree.

It turns out that, in analogy to conjunctive query evaluation, we can also get good bounds on the size of the grounding by considering a width measure of tree decompositions of the rules. We show the following theorem in Section 6, which relates the grounding size to the maximum *submodular width* (Section 6) across all rules.

THEOREM 3.3. *Let P be a Datalog program where* $\mathsf{arity}(P) \leq k$, subw *is the maximum submodular width across all rules of P, and σ is a dioid and suppose the input size is m, and the active domain size is n. Then, we can construct a σ-equivalent grounding in time (and has size)* $\widetilde{O}(n^{k-1} \cdot (m^{\mathsf{subw}} + n^{k \cdot \mathsf{subw}}))$.

The above theorem requires the semiring to be idempotent w.r.t. the $\oplus$ operation (i.e. a dioid). For a general semiring, the best we show is that we can replace subw with the weaker notion of fractional hypertree width (Proposition 6.4). Sections 5 and 6 show refined constructions that improve the grounding size when the Datalog program has additional structure (e.g., linear rules); we summarize these results in Table 1.

Lower Bounds. One natural question is: do Theorem 3.2 and 3.3 attain the best possible grounding bounds? This is unlikely to be true for specific Datalog fragments (e.g. linear Datalog has a tighter grounding as shown in Proposition 6.3); however, we can show optimality for a class of programs (proof can be found in Appendix B).

THEOREM 3.4. *Take any integer $k \geq 2$ and any rational number $w \geq 1$ such that $k \cdot w$ is an integer. There exists a (non-linear) Datalog program P over the tropical semiring* Trop^+ *with* $\mathsf{arity}(P) \leq k$, *such that:*

(1) $\mathsf{subw}(P) = w$,
(2) any Trop^+*-equivalent grounding has size* $\Omega(n^{k-1+kw})$,
(3) under the min-weight ℓ-Clique hypothesis [33], no algorithm that evaluates P has a runtime of $O(n^{k-1+kw-o(1)})$.

3.2 Grounding Evaluation

Given a grounding G for a Datalog program, we now turn to the problem of computing the (least fixpoint) solution for this grounding. In particular, we study *how fast can we evaluate G under different types of semirings as a function of its size.* Ideally, we would like to have an algorithm that computes the fixpoint using $O(|G|)$ semiring operations; however, this may not be possible in general. In Section 4, we will show fast evaluation strategies for two different classes of semirings:

(i) semirings of finite rank, and (ii) semirings that are totally ordered and absorptive. Our two main results can be stated as follows.

Theorem 3.5. *We can evaluate a grounding G over any semiring of rank r using $O(r \cdot |G|)$ semiring operations.*

Theorem 3.6. *We can evaluate a grounding G over any absorptive semiring with total order using $O(|G| \log |G|)$ semiring operations.*

Many semirings of practical interest are captured by the above two classes. The Boolean semiring in particular is a semiring of rank 1. Hence, we obtain as a direct corollary the result in [22]:

Corollary 3.7. *We can evaluate a grounding G over the Boolean semiring in time $O(|G|)$.*

The *set semiring* $(2^K, \cup, \cap, \emptyset, K)$ for a finite set K has rank $|K|$. In fact, all bounded distributive lattices have constant rank. As another example, the *access control* semiring $(\{P, C, S, T, 0\}, \min, \max, 0, P)$ [19] employs a constant number of security classifications, where P = PUBLIC, C = CONFIDENTIAL, S = SECRET, T = TOP-SECRET and $P \sqsubset C \sqsubset S \sqsubset T \sqsubset 0$ is the total order for levels of clearance. For all these semirings, their grounding can be evaluated in time $O(|G|)$ by Theorem 3.5. The class of absorptive semiring with total order contains Trop^+ that has infinite rank. Hence, Theorem 3.5 cannot be applied, but using Theorem 3.6, we can evaluate G over Trop^+ in time $O(|G| \log |G|)$.

3.3 Applications

Finally, we discuss algorithmic implications of our general framework. In particular, we demonstrate that our approach captures as special cases several state-of-the-art algorithms for tasks that can be described in Datalog. Table 2 summarizes some of our results that are straightforward applications of our framework.

To highlight some of our results, Dikjstra's algorithm for single-source shortest path is a special case of applying Theorem 3.6 after we compute the grounding of the program. As another example, Yannakakis [50] showed that a binary (i.e. EDB/IDB arities are at most 2) rulewise-acyclic programs can be evaluated in time $O(n^3)$. By Theorem 3.2 (let $k = 2$), an equivalent grounding of size $O(n^3)$ can be constructed in time $O(n^3)$ for such programs. Then by Corollary 3.7, we can evaluate the original program in $O(n^3)$ time, thus recovering the result of Yannakakis.

If the binary rulewise-acyclic Datalog program is also linear, it can be a evaluated in $O(m \cdot n)$ time [50]. We generalize this result to show that the $O(m \cdot n)$ bound holds for any linear rulewise-acyclic Datalog program with IDB arity at most 2, even if the EDB relations are of higher arity (see Table 2, Proposition 5.4).

As another corollary of Theorem 3.3 and Corollary 3.7, monadic Datalog can be evaluated in time $\widetilde{O}(m^{\text{subw}})$. This is surprising in our opinion, since $\widetilde{O}(m^{\text{subw}})$ is the best known runtime for Boolean CQs. Hence, this result tells us that the addition of recursion with unary IDB does not really add to the runtime of evaluation!

4 GROUNDING EVALUATION

This section presents algorithms for evaluating a grounding over two types of commonly-used semirings [28, 39, 40]. First, Section 4.1 presents a procedure that transforms the grounding to one with a more amenable structure, called a *2-canonical grounding*. Then, we present evaluation algorithms for semirings of rank r (Section 4.2), and for absoprtive semirings that are totally ordered (Section 4.3).

Table 2. Summary of runtime results. The $\widetilde{O}$ notation hides polylog factors in m (total input size) and n (size of the active domain). k denotes the arity of the program. See Section 6 for the definition of subw.

Task	Semiring	Program	Runtime
APSP [17, 49]	Tropical	$T(x_1, x_2) \leftarrow R(x_1, x_2) \oplus \bigoplus_{x_3} T(x_1, x_3) \otimes R(x_3, x_3)$	$\widetilde{O}(m \cdot n)$
Single-source Shortest Path (SSSP) [15]	Tropical	$T(x_1) \leftarrow U(x_1) \oplus \bigoplus_{x_2} T(x_2) \otimes R(x_2, x_1)$	$\widetilde{O}(m)$
CFL reachability [50]	Boolean	chain programs	$O(n^3)$
	Boolean	IDB arity ≤ 2 & rulewise-acyclic	$O(n^3)$
Regular Path Queries All-pairs [6, 9]	Boolean	linear chain programs	$O(m \cdot n)$
CFL-APSP [46, 50]	Tropical	chain programs	$\widetilde{O}(n^3)$
Monadic acyclic Datalog [21, 22]	Boolean	monadic rulewise-acyclic	$O(m)$
Monadic Datalog [21, 22]	Boolean	monadic	$\widetilde{O}(m^{\text{subw}})$
Linear Datalog [25]	Boolean	IDB arity $\leq k$	$\widetilde{O}(n^{k-1} m^{\text{subw}-1} \cdot (m + n^k))$
Datalog	Boolean	IDB arity $\leq k$	$\widetilde{O}(n^{k-1} \cdot (m^{\text{subw}} + n^{k \cdot \text{subw}}))$

4.1 2-canonical Grounding

We say that a grounding G is *2-canonical* if every equation in G is of the form $y = x \oplus z$ or $y = x \otimes z$. As a first step of our evaluation, we first transform the given grounding into a 2-canonical form using the following Lemma 4.1.

Lemma 4.1. *A grounding G can be transformed into a σ-equivalent 2-canonical grounding of size at most $4|G|$ in time $O(|G|)$.*

Proof. The construction works in two steps. In the first step, we rewrite all monomials. Consider a monomial $\boldsymbol{m} = x_1 \otimes x_2 \otimes \ldots x_n$. If $n = 1$, then simply let $\boldsymbol{m} = x_1 \otimes \mathbf{1}$. Otherwise ($n \geq 2$), we rewrite $\boldsymbol{m}$ as follows. We introduce $n - 1$ new variables $y_1, \ldots, y_{n-1}$, replace $\boldsymbol{m}$ with y_{n-1} and add the following equations in the system:

$$y_1 = x_1 \otimes x_2, \quad y_2 = y_1 \otimes x_3, \quad \ldots \quad y_{n-1} = y_{n-2} \otimes x_n$$

This transformation increases the size of the system by at most a factor of two. Indeed, the monomial contributes n to $|G|$, and the new equations contribute $1 + 2(n - 2) \leq 2n$.

After the first step, we are left with equations that are either a product of two elements, or a sum of the form $x = x_1 \oplus x_2 \oplus \ldots x_n$ (w.l.o.g., we can assume no equations of the form $x = y$). Here, we apply the same idea as above, replacing multiplication with addition. This transformation will

increase the size of the system by another factor of two using the same argument as before. The equivalence of the new grounding follows from the associativity property of both $\oplus, \otimes$. □

We demonstrate Lemma 4.1 with an example.

Example 4.2. Consider the polynomial equation showed up in Section 3: $x_{ab} = e_{ab} \oplus x_{aa} \otimes f_a \otimes e_{ab}$. Its rewriting (via substitutions) following the proof of Lemma 4.1 is:

$$x_{ab} = e_{ab} \oplus y_1, \quad y_1 = x_{aa} \otimes y_2, \quad y_2 = f_a \otimes e_{ab}$$

Here y_1, y_2 are new IDB variables introduced to make the system of polynomial equations 2-canonical.

4.2 Finite-rank Semirings

We now present a grounding evaluation algorithm (Algorithm 1) over a semiring of rank r, which is used to prove Theorem 3.5 (see subsection A.2).

Algorithm 1: Grounding evaluation over a rank r semiring

```
   Input  : grounding G
1  transform G to be 2-canonical via Lemma 4.1
2  construct a hash table E, such that for every variable x in G, E[x] is the set of all equations
     that contain x in the right-hand side ;  init an empty queue q
3  foreach EDB variables e in G do h(e) ← e ;
4  foreach IDB variables x in G do h(x) ← 0 ;
5  foreach EDB variables e in G do insert e into q ;
6  while q is not empty do
7  |  pop x from q
8  |  forall (y = x ⊛ z) ∈ E[x] do                                   // ⊛ ∈ {⊕, ⊗}
9  |  |  if h(y) ≠ h(x) ⊛ h(z) then
10 |  |  |  h(y) ← h(x) ⊛ h(z) insert y into q
11 return h(·)
```

The key idea of the algorithm is to compute the least fixpoint in a fine-grained way. Instead of updating all equations in every iteration, we will carefully choose a subset of equations to update their left-hand side variables. In particular, at every step we will pick a new variable (Line 7) and then only update the equations that contain this variable[2] (Line 8-10). Because the semiring has rank r, the value of each variable cannot be updated more than r times; hence, we are guaranteed that each equation will be visited only $2 \cdot r$ times. Moreover, since the grounding is 2-canonical, updating each variable needs only one $\oplus$ or $\otimes$ operation; the latter property would not be possible if we had not previously transformed the grounding into a 2-canonical one.

4.3 Absorptive Semirings with Total Order

We present a Dijkstra-style algorithm (Algorithm 2) for grounding evaluation over an absorptive semiring with $\sqsubseteq$ being a total order to prove Theorem 3.6 (see subsection A.3). It builds upon prior work [39] by further optimizing the original algorithm to achieve the almost-linear runtime.

[2]We assume that look-ups, inserts, and deletes on hash tables cost constant time. In practice, hashing can only achieve amortized constant time. Therefore, whenever we claim constant time for hash operations, we mean amortized constant time.

Algorithm 2: Grounding evaluation over an absorptive semiring with total order

Input : a grounding G

1 **transform** G to be 2-canonical via Lemma 4.1
2 **construct** a hash table E, such that for every variable x in G, $E[x]$ is the set of all equations that contain x in the right-hand side ; **init** $\mathcal{F} \leftarrow \emptyset$
3 **foreach** *EDB variables* e *in* G **do** $h(e) \leftarrow e$;
4 **foreach** *IDB variables* x *in* G **do** $h(x) \leftarrow \mathbf{0}$;
5 **init** an empty priority queue q of decreasing values w.r.t. $\sqsupseteq$
6 **forall** *IDB variables* $x = y \circledast z$ *in* G **do** // $\circledast \in \{\oplus, \otimes\}$
7 $\quad h(x) \leftarrow h(y) \circledast h(z)$; **insert** x into q of value $h(x)$
8 **while** q *is non-empty* **do**
9 $\quad$ **pop** a variable x of the max value from q
10 $\quad \mathcal{F} \leftarrow \mathcal{F} \cup \{x\}$;
11 $\quad$ **forall** $(y = x \circledast z) \in E[x] \wedge y \notin \mathcal{F}$ **do**
12 $\quad\quad$ **if** $h(y) \neq h(x) \circledast h(z)$ **then**
13 $\quad\quad\quad h(y) \leftarrow h(x) \circledast h(z)$
14 $\quad\quad\quad$ **insert** y into q of value $h(y)$
15 **return** $h(\cdot)$

The algorithm follows the same idea as Algorithm 1 by carefully updating only a subset of equations at every step. However there are two key differences. First, while Algorithm 1 is agnostic to the order in which the newly updated variables are propagated to the equations (hence the use of a queue), Algorithm 2 needs to always pick the variable with the current maximum value w.r.t. the total order $\sqsubseteq$. To achieve this, we need to use a priority queue instead of a queue, which is the reason of the additional logarithmic factor in the runtime. The second difference is that once a variable is updated once, it gets "frozen" and never gets updated again (see Lines 10-11). We show in the detailed proof of correctness (Appendix A) that it is safe to do this and still reach the desired fixpoint.

5 GROUNDING OF ACYCLIC DATALOG

In this section, we study how to find an efficient grounding (in terms of space usage and time required) of a Datalog program over a semiring σ. First, we introduce *rulewise-acyclicity* of a program using the notion of *tree decompositions*.

Example 5.1. We ground the APSP program (3). First, $R(x_1, x_2)$ produces $O(|R|)$ groundings, since it is an EDB. The latter sum-prod query has a $O(|R| \cdot n)$ grounding, since for each tuple $R(x_3, x_2)$, we should consider all active domain of x_1. Hence, the equivalent grounding is of size $O(|R| + |R| \cdot n) = O(m \cdot n)$.

Tree Decompositions. Let φ be a sum-prod query (5) and $([\ell], \mathcal{E})$ be its associated hypergraph. A *tree decomposition* of φ is a tuple $(\mathcal{T}, \chi)$ where (i) $\mathcal{T}$ is an undirected tree, and (ii) χ is a mapping that assigns to every node $t \in V(\mathcal{T})$ a set of variables $\chi(t) \subseteq [\ell]$, called the *bag* of t, such that

(1) every hyperedge $J \in \mathcal{E}$ is a subset of some $\chi(t)$, $t \in V(\mathcal{T})$;
(2) (running intersection property) for each $i \in [\ell]$, the set $\{t \mid i \in \chi(t)\}$ is a non-empty (connected) sub-tree of $\mathcal{T}$.

A tree decomposition is a *join tree* if $\chi(t) \in \mathcal{E}$ for all $t \in V(\mathcal{T})$.

Acyclicity. A sum-prod query φ is said to be *acyclic* if its associated hypergraph admits a join tree. The GYO reduction [51] is a well-known method to construct a join tree for φ. We say that a rule of a Datalog program is *acyclic* if every sum-prod query in its body is acyclic and a program is *rulewise-acyclic* if it has only acyclic rules.

We formally prove Theorem 3.2, by introducing a grounding algorithm for rulewise-acyclic programs attaining the desired bounds; the pseudocode can be found in [52].

Proof of Theorem 3.2. The σ-equivalent grounding G is constructed rule-wise from P. Take any acyclic rule from P with head $T(\mathbf{x}_H)$. Note that its arity $|H| \le k$. We ground each sum-prod query $\varphi(\mathbf{x}_H)$ in this rule one by one. To ground a single φ (which is of the form (5)), we construct a join tree $(\mathcal{T}, \chi)$ of φ. Each bag in the join tree will correspond to an atom in the body of φ. We root $\mathcal{T}$ from any atom that contains at least one of the variables in $\mathbf{x}_H$ (i.e. head variables), say s_0. This orients the join tree. For any node s, define $(V_s, \mathcal{E}_s)$, where $V_s \subseteq [\ell], \mathcal{E}_s \subseteq \mathcal{E}$, to be the hypergraph constructed by only taking the bags from the subtree of $\mathcal{T}$ rooted at t as hyperedges. Note that $(V_{s_0}, \mathcal{E}_{s_0}) = ([\ell], \mathcal{E})$ is the associated hypergraph of φ. Let $H_s \subseteq H$ be the head variables that occur in this subtree (so at root, $H_{s_0} = H$). Further, rename the head atom $T(\mathbf{x}_H)$ to be $U_{e_{(\perp,s_0)}}(\mathbf{x}_H)$, with $\perp$ being an imaginary parent node of the root s and $e_{(\perp,s_0)} = H$.

Next, we use a recursive method Ground to ground φ. Let r be the parent of s in $\mathcal{T}$. A Ground$(s, U_{e_{(r,s)}})$ call at a node s grounds the sum-prod query

$$U_{e_{(r,s)}}(\mathbf{x}_{e_{(r,s)}}) \leftarrow \bigoplus_{\mathbf{x}_{V_s \setminus e_{(r,s)}}} \bigotimes_{J \in \mathcal{E}_s} T_J(\mathbf{x}_J). \tag{6}$$

Hence, calling Ground$(s_0, U_{e_{(\perp,s_0)}})$ at s_0 suffices to ground φ. We describe the procedure Ground$(s, U_{e_{(r,s)}})$, which has three steps:

(i) *Refactor.* Define for each child t of s (i.e. $(s,t) \in E(\mathcal{T})$, $E(\mathcal{T})$ being the directed edges of $\mathcal{T}$), $e_{(s,t)} = (\chi(s) \cap \chi(t)) \cup H_t$ and

$$S = \chi(s) \cup \bigcup_{t:(s,t)\in E(\mathcal{T})} e_{(s,t)}.$$

We refactor as follows: $U_{e_{(r,s)}}(\mathbf{x}_{e_{(r,s)}}) = \bigoplus_{\mathbf{x}_{V_s \setminus e_{(r,s)}}} \bigotimes_{J \in \mathcal{E}_s} T_J(\mathbf{x}_J)$

$$\overset{(1)}{=} \bigoplus_{\mathbf{x}_{V_s \setminus e_{(r,s)}}} T_{\chi(s)}(\mathbf{x}_{\chi(s)}) \otimes \bigotimes_{t:(s,t)\in E(\mathcal{T})} \bigotimes_{J \in \mathcal{E}_t} T_J(\mathbf{x}_J)$$

$$\overset{(2)}{=} \bigoplus_{\mathbf{x}_{S \setminus e_{(r,s)}}} T_{\chi(s)}(\mathbf{x}_{\chi(s)}) \otimes \bigotimes_{t:(s,t)\in E(\mathcal{T})} \left(\bigoplus_{\mathbf{x}_{V_t \setminus e_{(s,t)}}} \bigotimes_{J \in \mathcal{E}_t} T_J(\mathbf{x}_J) \right)$$

$$\overset{(3)}{=} \bigoplus_{\mathbf{x}_{S \setminus e_{(r,s)}}} T_{\chi(s)}(\mathbf{x}_{\chi(s)}) \otimes \bigotimes_{t:(s,t)\in E(\mathcal{T})} U_{e_{(s,t)}}(\mathbf{x}_{e_{(s,t)}})$$

where (1) regroups the product into the node s and subtrees rooted at each child t of s, (2) safely pushes aggregation over every child t (since any $i \in V_t \setminus e_{(s,t)}$ only occurs in the subtree rooted at t, otherwise by the running intersection property, $i \in \chi(s) \cap \chi(t) \subseteq e_{(s,t)}$, a contradiction), and (3) replaces every child sum-prod query by introducing a new IDB $U_{e_{(s,t)}}(\mathbf{x}_{e_{(s,t)}})$ and a corresponding query:

$$U_{e_{(s,t)}}(\mathbf{x}_{e_{(s,t)}}) \leftarrow \bigoplus_{\mathbf{x}_{V_t \setminus e_{(s,t)}}} \bigotimes_{J \in \mathcal{E}_t} T_J(\mathbf{x}_J). \tag{7}$$

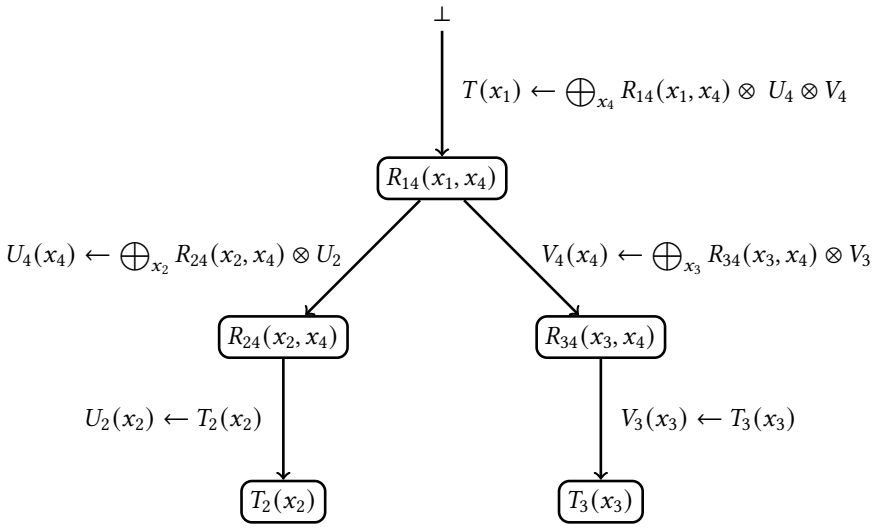

Fig. 1. A join tree with its corresponding rewriting.

(*ii*) *Ground.* Instead of grounding the sum-prod query (6) as a whole, we ground the σ-equivalent (but refactored) query

$$U_{e_{(r,s)}}(\mathbf{x}_{e_{(r,s)}}) \leftarrow \bigoplus_{\mathbf{x}_{S \setminus e_{(r,s)}}} T_{\chi(s)}(\mathbf{x}_{\chi(s)}) \otimes \bigotimes_{t:(s,t)\in E(\mathcal{T})} U_{e_{(s,t)}}(\mathbf{x}_{e_{(s,t)}}).$$

Notice that S is exactly the set of variables appear in the body of the refactored query. We ground this query as follows: for each possible tuple (say $\mathbf{c}_{\chi(s)}$) in $T_{\chi(s)}$, we add a grounded rule for each tuple (say $\mathbf{c}_{S\setminus\chi(s)}$) that can be formed over the schema $\mathbf{x}_{S\setminus\chi(s)}$ using the active domain of each variable. This is done by taking the attribute values for each variable from $\mathbf{c}_{\chi(s)}$ and $\mathbf{c}_{S\setminus\chi(s)}$, and substituting it in every predicate (EDB, IDB, or the head) of the query.

(*iii*) *Recurse.* Now every child t of s has an introduced IDB $U_{e_{(s,t)}}$. For each t, we call into $\textsc{Ground}(t, U_{e_{(s,t)}})$ to recursively ground the new rule (7). If there are no children (i.e. s is a leaf node), the $\textsc{Ground}$ call on s terminates immediately.

We argue the grounding size and time at the grounding step (*ii*), for any node s when grounding the refactored query. Recall that r is the parent node of s. We consider all possible tuples over $\mathbf{x}_S$ (variables in the body of the refactored query). First, $T_{\chi(s)}$ is of size $O(m)$ if it is an EDB, or $O(n^k)$ if it is an IDB. Next, for the number of tuples that can be constructed over $\mathbf{x}_{S\setminus\chi(s)}$, note that

$$S = \chi(s) \cup \bigcup_{t:(s,t)\in E(\mathcal{T})} e_{(s,t)} = \chi(s) \cup \bigcup_{t:(s,t)\in E(\mathcal{T})} H_t \subseteq \chi(s) \cup H_s$$

where the inclusion holds since $H_t \subseteq H_s$ if t is a child of s. Observe that if $H_s \subsetneq H$, then $|H_s| \leq k-1$; otherwise (i.e., $H_s = H$), since we have rooted the join tree to a node that contains at least one head variable, at least one of the head variables in H_s must belong to $\chi(s)$ (again by the running intersection property), in which case also $|H_s \setminus \chi(s)| \leq k-1$. Thus, $|S \setminus \chi(s)| \leq |(\chi(s) \cup H_s) \setminus \chi(s)| = |H_s \setminus \chi(s)| \leq k-1$. Hence, the groundings inserted for any intermediate rule is of size (and in time) $O(n^{k-1} \cdot (m + n^k))$, i.e. the product of the cardinality of $T_{\chi(s)}$ and the number of possible tuples that can be formed over the schema $\mathbf{x}_{S\setminus\chi(S)}$. □

Example 5.2. We illustrate Theorem 3.2 with the acyclic rule:

$$T(x_1) \leftarrow \bigoplus_{\mathbf{x}_{234}} T_2(x_2) \otimes T_3(x_3) \otimes R_{24}(\mathbf{x}_{24}) \otimes R_{34}(\mathbf{x}_{34}) \otimes R_{14}(\mathbf{x}_{14})$$

A join tree is drawn in Figure 1 with new intermediate IDBs and rules (one per edge of the join tree) introduced recursively by the proof of Theorem 3.2. In the transformed program, each rule produces $O(m+n) = O(m)$ groundings. Hence, its overall size is $O(m)$.

We now provide a step-by-step walk-through of how to evaluate the rule by following the tree decomposition in Figure 1. For succinctness, we call the nodes in the join tree by their corresponding assigned atoms. Following the rooted join tree in Figure 1, we start from calling Ground at the root node R_{14}, i.e. $\textsc{Ground}(R_{14}, T(x_1))$:

(*i*) (*Refactor*) The root node R_{14} has two children and thus we introduce two new IDBs $U_4(x_4)$ for the node R_{24} and $V_4(x_4)$ for the node R_{34}. This corresponds to the following *refactoring* of the original rule:

$$\begin{aligned} T(x_1) &= \bigoplus_{\mathbf{x}_{234}} T_2(x_2) \otimes T_3(x_3) \otimes R_{24}(x_2, x_4) \otimes R_{34}(x_3, x_4) \otimes R_{14}(x_1, x_4) \\ &= \underbrace{\bigoplus_{x_4} R_{14}(x_1, x_4) \otimes \left(\bigoplus_{x_2} R_{24}(x_2, x_4) \otimes T_2(x_2) \right) \otimes \left(\bigoplus_{x_3} R_{34}(x_3, x_4) \otimes T_3(x_3) \right)}_{\textit{(push } \bigoplus \textit{ over two children)}} \\ &= \underbrace{\bigoplus_{x_4} R_{14}(x_1, x_4) \otimes U_4(x_4) \otimes V_4(x_4)}_{\textit{(introduce two new IDBs)}} \end{aligned}$$

along with two new rules for the intermediate IDBs:

$$\begin{aligned} U_4(x_4) &\leftarrow \bigoplus_{x_2} R_{24}(x_2, x_4) \otimes T_2(x_2) \\ V_4(x_4) &\leftarrow \bigoplus_{x_3} R_{34}(x_3, x_4) \otimes T_3(x_3). \end{aligned}$$

(*ii*) (*Ground*) We ground the refactored rule $T(x_1) \leftarrow \bigoplus_{x_4} R_{14}(x_1, x_4) \otimes U_4(x_4) \otimes V_4(x_4)$ (i.e. the top rule shown in Figure 1) by taking all tuples in the EDB R_{14} and that naturally subsumes all possible tuples with non-zero annotations (i.e. $R_{14}(x_1, x_4) \otimes U_4(x_4) \otimes V_4(x_4) \neq \mathbf{0}$). Recall that tuples not in the EDB R_{24} has an implicit annotation of $\mathbf{0}$ and thus do not contribute to the sum-product. This gives a grounding of size $O(|R_{14}|) = O(m)$.

(*iii*) (*Recurse*) We recursively call Ground on the children nodes, i.e. $\textsc{Ground}(R_{24}, U_4(x_4))$ and $\textsc{Ground}(R_{34}, V_4(x_4))$. To avoid repeating, we only demonstrate $\textsc{Ground}(R_{24}, U_4(x_4))$ next.

Again, following the join tree rooted at R_{24}, we call $\textsc{Ground}(R_{24}, U_4(x_4))$:

(*i*) (*Refactor*) The node R_{24} has one child $T_2(x_2)$ and thus we introduce a new IDB $U_2(x_2)$. This corresponds to the following *refactoring* of the original rule:

$$\begin{aligned} U_4(x_4) &= \bigoplus_{x_2} R_{24}(x_2, x_4) \otimes T_2(x_2) && \textit{(no more variables to push } \textstyle\bigoplus\textit{)} \\ &= \bigoplus_{x_2} R_{24}(x_2, x_4) \otimes U_2(x_2) && \textit{(introduce a new IDB)} \end{aligned}$$

along with a new rule for the intermediate IDB:

$$U_2(x_2) \leftarrow T_2(x_2).$$

(ii) (*Ground*) We ground the refactored rule

$$U_4(x_4) \leftarrow \bigoplus_{x_2} R_{24}(x_2, x_4) \otimes U_2(x_2)$$

by taking all tuples in the EDB $R_{24}(x_2, x_4)$. This gives a grounding of size $O(|R_{24}|) = O(m)$.

(iii) (*Recurse.*) We recursively call $\textsc{Ground}(T_2, U_2(x_2))$ next.

Finally, we call $\textsc{Ground}(T_2, U_2(x_2))$. However, since T_2 is a leaf node, there is no refactoring and further recursion. We simply ground the rule: $U_2(x_2) \leftarrow T_2(x_2)$ by taking all values in the active domain of x_2. This gives a grounding of size $O(n)$ (recall $n \le m$).

Free-connexity. Take a tree decomposition $(\mathcal{T}, \chi)$ of a sum-prod query $\varphi(\mathbf{x}_H)$ rooted at a node $r \in V(\mathcal{T})$. Let $\mathsf{TOP}_r(x)$ be the highest node in $\mathcal{T}$ containing x in its bag. We say that $(\mathcal{T}, \chi)$ is *free-connex w.r.t. r* if for any $x \in H$ and $y \in [\ell] \setminus H$, $\mathsf{TOP}_r(y)$ is not an ancestor of $\mathsf{TOP}_r(x)$ [48]. We say that $(\mathcal{T}, \chi)$ is free-connex if it is free-connex w.r.t. some $r \in V(\mathcal{T})$. An acyclic query is free-connex if it admits a free-connex join tree. If every sum-prod query in every rule of a program P admits a free-connex join tree, then P is *rulewise free-connex acyclic*. Theorem 5.3 (see Table 1) shows that in such a case, there is a much tighter bound (dropping the n^{k-1} term).

Proposition 5.3. *Let P be a rulewise free-connex acyclic Datalog program over some semiring σ with input size m, active domain size n, and* $\mathsf{arity}(P)$ *is at most k. Then, a σ-equivalent grounding can be constructed in time (and has size) $O(m + n^k)$.*

Linear Acyclic Programs. If $\mathsf{arity}(P) \le 2$, Theorem 3.2 states that a rulewise-acyclic Datalog program admits a grounding of size $O(n^3)$. If the program is also linear (e.g. same generation [5]), we strengthen the upper bound to $O(m \cdot n)$ (see Table 1). The proof can be found in [52] and an example is shown in Appendix C.

Proposition 5.4. *Let P be a linear rulewise-acyclic Datalog program over some semiring σ with input size m, active domain size n, and* $\mathsf{arity}(P)$ *is at most 2. Then, we can construct a σ-equivalent grounding in time (and has size) $O(m \cdot n)$.*

6 GROUNDING OF GENERAL DATALOG

This section introduces an algorithm for grounding any Datalog program over an arbitary dioid via the PANDA algorithm [29] (or PANDA, for short). PANDA is introduced to evaluate CQs (i.e. sum-prod queries over the *Boolean semiring*).

Submodular width. A function $f : 2^{[\ell]} \mapsto \mathbb{R}_+$ is a non-negative *set function* on $[\ell]$ ($\ell \ge 1$). The set function is *monotone* if $f(X) \le f(Y)$ whenever $X \subseteq Y$, and is *submodular* if $f(X \cup Y) + f(X \cap Y) \le f(X) + f(Y)$ for all $X, Y \subseteq [\ell]$. A non-negative, monotone, submodular set function h such that $h(\emptyset) = 0$ is a *polymatroid*. Let φ be a sum-prod query (5). Let Γ_ℓ be the set of all polymatroids h on $[\ell]$ such that $h(J) \le 1$ for all $J \in \mathcal{E}$. The *submodular width* of φ is

$$\mathsf{subw}(\varphi) := \max_{h \in \Gamma_\ell} \min_{(\mathcal{T}, \chi) \in \mathcal{F}} \max_{t \in V(\mathcal{T})} h(\chi(t)), \tag{8}$$

where $\mathcal{F}$ is the set of all *non-redundant* tree decompositions of φ. A tree decomposition is *non-redundant* if no bag is a subset of another. Abo Khamis et al. [29] proved that non-redundancy ensures that $\mathcal{F}$ is finite, hence the inner minimum is well-defined. We define the *free-connex*

submodular width of φ, $\mathsf{fsubw}(\varphi)$, by restricting $\mathcal{F}$ to the set of all non-redundant free-connex tree decompositions.

We extend subw and fsubw to Datalog. The *submodular width* of a program P (i.e. $\mathsf{subw}(P)$) is the maximum $\mathsf{subw}(\varphi)$ across all sum-prod queries in P. Likewise, the *free-connex submodular width* of P (i.e. $\mathsf{fsubw}(P)$) is the maximum $\mathsf{fsubw}(\varphi)$ over all sum-prod queries in P. If P is rulewise-acyclic, $\mathsf{subw}(P) = 1$.

This section describes the key ideas of the algorithm underlying Theorem 3.3. We demonstrate the grounding algorithm step-by-step on the following concrete example.

Example 6.1. The following Datalog computes a diamond-pattern reachability starting from some node a set $U(x_1)$,

$$T(x_1) \leftarrow U(x_1) \tag{9}$$

$$T(x_1) \leftarrow \bigoplus_{\mathbf{x}_{234}} T(x_3) \otimes R_{32}(\mathbf{x}_{32}) \otimes R_{21}(\mathbf{x}_{21}) \otimes R_{34}(\mathbf{x}_{34}) \otimes R_{41}(\mathbf{x}_{41}) \tag{10}$$

A simple (but suboptimal) grounding is to ground the 4-cycle join first, materializing the opposite nodes of the cycle, i.e.

$$\mathsf{Cycle}(x_1, x_3) \leftarrow \bigoplus_{\mathbf{x}_{24}} R_{32}(x_3, x_2) \otimes R_{21}(x_2, x_1) \otimes R_{34}(x_3, x_4) \otimes R_{41}(x_4, x_1)$$

$$T(x_1) \leftarrow U(x_1) \oplus \bigoplus_{\mathbf{x}_3} T(x_3) \otimes \mathsf{Cycle}(x_1, x_3)$$

PANDA evaluates the first rule (essentially a non-recursive sum-prod query) in time $\widetilde{O}(m^{3/2} + n^2)$, since $\mathsf{Cycle}(x_1, x_3)$ has a grounding (or output) of size $O(n^2)$.

We ground the rule in Equation 10 that involves a 4-cycle join. In particular, our grounding algorithm constructs a grounding of size $\widetilde{O}(m^{3/2})$ (note the missing n^2 additive factor).

(*i*) *Refactor.* Let $([4], \mathcal{E})$ be the associated hypergraph of the above cyclic recursive rule, where $\mathcal{E} = \{\{3\}, \{3, 2\}, \{2, 1\}, \{3, 4\}, \{4, 1\}\}$. We construct the non-redundant tree decompositions $(\mathcal{T}_1, \chi_1)$, $(\mathcal{T}_2, \chi_2)$ (there are only two), both having two nodes $\{v_1, v_2\}$ and one edge $\{(v_1, v_2)\}$, where the bags are

$$\chi_1(v_1) = [3], \chi_1(v_2) = \{3, 4, 1\};\ \chi_2(v_1) = \{4, 1, 2\}, \chi_2(v_2) = \{2, 3, 4\}.$$

This allows us to rewrite and factorize the cyclic body into two acyclic sum-prod queries, one for each decomposition:

$$\begin{aligned}\varphi(x_1) &= \bigoplus_{\mathbf{x}_{234}} T(x_3) \otimes R_{32} \otimes R_{21} \otimes R_{34} \otimes R_{41} \\ &= \bigoplus_{\mathbf{x}_{234}} \underbrace{T \otimes R_{32} \otimes R_{21}}_{B_{[3]}(\mathbf{x}_{[3]})} \otimes \underbrace{R_{34} \otimes R_{41}}_{B_{341}(\mathbf{x}_{341})} \oplus \underbrace{R_{21} \otimes R_{41}}_{B_{412}(\mathbf{x}_{412})} \otimes \underbrace{T \otimes R_{32} \otimes R_{34}}_{B_{234}(\mathbf{x}_{234})}\end{aligned}$$

where the second line uses the idempotence of $\oplus$. The underbraces introduce new IDBs and rules (e.g. $B_{[3]} \leftarrow T \otimes R_{32} \otimes R_{21}$ corresponds to the bag $\chi_1(v_1)$ of $\mathcal{T}_1$ and likewise for B_{341}, B_{234} and B_{412}).

(*ii*) *Grounding Bags.* A naïve grounding of (say) $B_{[3]}$ is of size $O(m \cdot n)$ (cartesian product of R_{32} with domain of x_1), which is suboptimal. Instead, we use PANDA to ground the new IDBs.

Let $Q(\mathbf{x}_{[4]})$ be the set of tuples such that a tuple $\mathbf{c}_{[4]} \in Q$ if and only if its annotation $T(\mathbf{c}_3) \otimes R_{32}(\mathbf{c}_{32}) \otimes R_{21}(\mathbf{c}_{21}) \otimes R_{34}(\mathbf{c}_{34}) \otimes R_{41}(\mathbf{c}_{41}) \neq \mathbf{0}$. Here, $\mathbf{c}_J$ with $J \subseteq [4]$ is a shorthand for the projection of $\mathbf{c}_{[4]}$ onto the variables in J. The PANDA algorithm outputs one table for every new IDB (to differentiate, denote the PANDA output tables as $B^*_{[3]}$, B^*_{341}, B^*_{234} and B^*_{412}) such that: (1) each table

is of size $\widetilde{O}(m^{3/2})$ since subw$(P) = 3/2$; (2) $Q(\mathbf{x}_{[4]})$ is equal to the union of two CQs, one for each decomposition, i.e.

$$(B^*_{[3]}(\mathbf{x}_{[3]}) \bowtie B^*_{341}(\mathbf{x}_{341})) \cup (B^*_{412}(\mathbf{x}_{412}) \bowtie B^*_{234}(\mathbf{x}_{234})) = Q(\mathbf{x}_{[4]})$$

Using PANDA output tables, we ground the four new IDBs, e.g. for each tuple $\mathbf{c}_{[3]} \in B^*_{[3]}$, we add a grounded rule: $B_{[3]}(\mathbf{c}_{[3]}) \leftarrow T(\mathbf{c}_3) \otimes R_{32}(\mathbf{c}_{32}) \otimes R_{21}(\mathbf{c}_{21})$. The same step is applied to all IDBs in total time and size $\widetilde{O}(m^{3/2})$. This ensures that each tuple in $Q(\mathbf{x}_{[4]})$ is present in at least one of $B_{[3]} \otimes B_{341}$ or $B_{412} \otimes B_{234}$, with the correct annotation being preserved , i.e. for any $\mathbf{c}_{[4]} \in Q(\mathbf{x}_{[4]})$,

$$T \otimes R_{32} \otimes R_{21} \otimes R_{34} \otimes R_{41} = (B_{[3]} \otimes B_{341}) \oplus (B_{412} \otimes B_{234})\,.$$

(*iii*) *Grounding Acyclic Sub-queries.* With the grounded bags, we now rewrite $\varphi(x_1)$ as a sum of two acyclic sum-prod queries:

$$\varphi(x_1) = \left[\bigoplus_{\mathbf{x}_{234}} B_{[3]}(\mathbf{x}_{[3]}) \otimes B_{341}(\mathbf{x}_{341})\right] \oplus \left[\bigoplus_{\mathbf{x}_{234}} B_{412} \otimes B_{234}\right]$$

We ground the two acyclic sub-queries (one per decomposition) using the construction in the proof of Theorem 3.2. For example, if we root both trees at v_1, we get a rewriting:

$$U_{31}(\mathbf{x}_{31}) \leftarrow \bigoplus_{x_4} B_{341} \qquad U_{24}(\mathbf{x}_{24}) \leftarrow \bigoplus_{x_3} B_{234}$$

$$\varphi(x_1) \leftarrow \left[\bigoplus_{\mathbf{x}_{23}} B_{[3]} \otimes U_{31}(\mathbf{x}_{31})\right] \oplus \left[\bigoplus_{\mathbf{x}_{24}} B_{412} \otimes U_{24}(\mathbf{x}_{24})\right]$$

guaranteeing that every intermediate IDB has a grounding of size $\widetilde{O}(m^{3/2})$. Thus, the total size of the grounding for P is $\widetilde{O}(m^{3/2})$.

We show a free-connex version (Proposition 6.2) by restricting the first refactoring step to use only free-connex decompositions. Though fsubw $\geq$ subw, a n^{k-1} factor is shaved off from the bound.

Proposition 6.2. *Let P be a Datalog program with input size m, active domain size n,* arity$(P) \leq k$. fsubw *is its free-connex submodular width. Let σ be a dioid. Then, a σ-equivalent grounding can be constructed in time (and has size)* $\widetilde{O}(m^{\text{fsubw}} + n^{k\cdot\text{fsubw}})$.

Linear Programs. For linear programs, a careful analysis yields the following improved result.

Proposition 6.3. *Let P be a linear Datalog program where* arity$(P) \leq k$*, input size m, and active domain size n. Let σ be a dioid. Then, a σ-equivalent grounding can be constructed in time (and has size)* $\widetilde{O}(n^{k-1}m^{\text{subw}-1} \cdot (m + n^k))$.

Fractional Hypertree-width. So far, all our results only apply to dioids. However, we can extend our results to *any* semiring by using the InsideOut algorithm [2]. Similar to subw(φ) (8), the *fractional hypertree-width* of a sum-prod query φ, i.e. fhw(φ), is defined as

$$\text{fhw}(\varphi) := \min_{(\mathcal{T},\chi)\in\mathcal{F}} \max_{h\in\Gamma_\ell} \max_{t\in V(\mathcal{T})} h(\chi(t))$$

The fractional hypertree-width of a program P is the maximum fhw over all sum-prod queries, denoted as fhw(P). We show Proposition 6.4 for the grounding size over *any* semiring (see Table 1).

Proposition 6.4. *Let P be a Datalog program with input size m, active domain size n, and* arity$(P) \leq k$*. Let σ be a naturally-ordered semiring. Then, a σ-equivalent grounding can be constructed in time (and has size)* $O(n^{k-1} \cdot (m^{\text{fhw}} + n^{k\cdot\text{fhw}}))$.

7 RELATED WORK

In this section, we present a brief overview of the prior works that are related to our work.

Complexity of Datalog. Data complexity of special fragments of Datalog has been explicitly studied. Gottlob et al. [22] showed that monadic acyclic Datalog can be evaluated in linear time w.r.t the size of the program plus the size of the input data. Gottlob et al. [21] defined the fragment *Datalog LITE* as the set of all stratified Datalog queries whose rules are either monadic or guarded. The authors showed that this fragment can also be evaluated in linear time as monadic acyclic Datalog. This result also follows from a generalization of Courcelle's theorem [12] by Flum et al. [18]. Our framework subsumes these results as an application. Lutz et al. [34] studied efficient enumeration of *ontology-mediated queries* that are acyclic and free-connex acyclic. The reader may refer to Green at al. [23] for an in-depth survey of Datalog rewriting, Datalog with disjunctions, and with integrity constraints. However, there is no principled study on general Datalog programs in parameterized complexity despite its prevalence in the literature of join query evaluation [2, 29, 35, 38].

Datalog over Semirings. Besides Datalog° [28], recent work [25] has looked at the convergence rate for linear Datalog°. The evaluation of Datalog over absorptive semirings with a total order has also been studied [39] where the key idea is to transform the program (and its input) into a weighted hypergraph and use Knuth's algorithm [30] for evaluation. Our paper recovers and extends this result by formalizing the proof of correctness via the concept of asynchronous Kleene chains, and showing a precise runtime for evaluating a grounding over such semirings. None of the works mentioned above focus on finding the smallest possible grounding, a key ingredient to show the tightest possible bounds.

Datalog Provenance Computation and Circuits. Algorithms for Datalog provenance computation provides an alternative way to think of Datalog evaluation. Deutch et al. [14] initiated the study of circuits for database provenance. They show that for a Datalog program having $|G|$ groundings for IDBs, a circuit for representing Datalog provenance for the $\mathrm{Sorp}(X)$ semiring (absorptive semirings are a special case of Sorp) can be built using only $|G|+1$ layers. As an example, for APSP, the circuit construction and evaluation cost $O(n^4)$ time. An improvement of the result [27] showed that for APSP, a monotone arithmetic circuit of size $O(n^3)$ can be constructed by mimicking the dynamic programming nature of the Bellman-Ford algorithm. We use this improvement to show a circuit unconditional lower bound on the grounding size. Ramusat et al. [40] have also shown that Djikstra's algorithm can be coupled with the semi-naïve evaluation for Datalog provenance.

8 CONCLUSION

This paper introduces a general two-phased framework that uses the structure of a Datalog program to construct a tight grounding, and then evaluates it using the algebraic properties of the semiring. Our framework successfully recovers state-of-the-art results for popular programs (e.g. chain programs, APSP), and uncovers new results (e.g. for linear Datalog). We also show a matching lower bound (both for running time and space requirement) for a class of Datalog programs. Future work includes efficient evaluation over broader classes of semirings [28], circuit complexity of Datalog over semirings, and general grounding lower bounds.

ACKNOWLEDGMENTS

This work was done in part while Zhao, Koutris, Roy, and Tannen were visiting the Simons Institute for the Theory of Computing. We gratefully acknowledge the support of NSF via awards IIS-2008107, IIS-2147061, and IIS-1910014.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.

[2] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '16)*. New York, NY, USA.

[3] Foto Afrati and Christos H Papadimitriou. 1993. The parallel complexity of simple logic programs. *Journal of the ACM (JACM)* 40, 4 (1993), 891–916.

[4] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph. D. Dissertation. Citeseer.

[5] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*. 1–15.

[6] Pablo Barceló Baeza. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*. 175–188.

[7] Nofar Carmeli, Batya Kenig, Benny Kimelfeld, and Markus Kröll. 2021. Efficiently enumerating minimal triangulations. *Discret. Appl. Math.* 303 (2021), 216–236.

[8] Nofar Carmeli and Markus Kröll. 2021. On the Enumeration Complexity of Unions of Conjunctive Queries. *ACM Transactions on Database Systems (TODS)* 46, 2 (2021), 1–41.

[9] Katrin Casel and Markus L. Schmid. 2021. Fine-Grained Complexity of Regular Path Queries. In *ICDT (LIPIcs, Vol. 186)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:20.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[11] Stavros Cosmadakis. 1999. Inherent complexity of recursive queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 148–154.

[12] Bruno Courcelle. 1990. Graph rewriting: An algebraic and logic approach. In *Formal Models and Semantics*. Elsevier, 193–242.

[13] Brian A Davey and Hilary A Priestley. 2002. *Introduction to lattices and order*. Cambridge university press.

[14] Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. 2014. Circuits for Datalog Provenance.. In *ICDT*, Vol. 3. Citeseer, 2014.

[15] Edsger W Dijkstra. 2022. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*. 287–290.

[16] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2010. Newtonian Program Analysis. *J. ACM* 57, 6, Article 33 (nov 2010), 47 pages.

[17] Robert W Floyd. 1962. Algorithm 97: shortest path. *Commun. ACM* 5, 6 (1962), 345.

[18] Jörg Flum, Markus Frick, and Martin Grohe. 2002. Query evaluation via tree-decompositions. *Journal of the ACM (JACM)* 49, 6 (2002), 716–752.

[19] J. Nathan Foster, Todd J. Green, and Val Tannen. 2008. Annotated XML: Queries and Provenance. In *PODS*. ACM, 271–280.

[20] Michael R Garey. 1997. Computers and intractability: A guide to the theory of np-completeness, freeman. *Fundamental* (1997).

[21] Georg Gottlob, Erich Grädel, and Helmut Veith. 2002. Datalog LITE: A deductive query language with linear time model checking. *ACM Transactions on Computational Logic (TOCL)* 3, 1 (2002), 42–79.

[22] Georg Gottlob and Christoph Koch. 2004. Monadic datalog and the expressive power of languages for Web information extraction. *J. ACM* 51, 1 (2004), 74–113.

[23] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Found. Trends Databases* 5, 2 (2013), 105–195.

[24] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 31–40.

[25] Sungjin Im, Benjamin Moseley, Hung Ngo, and Kirk Pruhs. 2023. On the Convergence Rate of Linear Datalogo over Stable Semirings. *arXiv preprint arXiv:2311.17664* (2023).

[26] Manas Joglekar and Christopher Ré. 2018. It's All a Matter of Degree - Using Degree Information to Optimize Multiway Joins. *Theory Comput. Syst.* 62, 4 (2018), 810–853.

[27] Stasys Jukna. 2015. Lower Bounds for Tropical Circuits and Dynamic Programs. *Theory Comput. Syst.* 57, 1 (2015), 160–194.

[28] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. Convergence of Datalog over (Pre-) Semirings. In *PODS*. ACM, 105–117.

[29] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *PODS*. ACM, 429–444.

[30] Donald E Knuth. 1977. A generalization of Dijkstra's algorithm. *Inform. Process. Lett.* 6, 1 (1977), 1–5.
[31] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 780–793.
[32] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2021. On the complexity of bidirected interleaved Dyck-reachability. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.
[33] Andrea Lincoln, Virginia Vassilevska Williams, and R. Ryan Williams. 2018. Tight Hardness for Shortest Cycles and Paths in Sparse Graphs. In *SODA*. SIAM, 1236–1252.
[34] Carsten Lutz and Marcin Przybylko. 2022. Efficiently enumerating answers to ontology-mediated queries. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 277–289.
[35] Dániel Marx. 2010. Can You Beat Treewidth? *Theory Comput.* 6, 1 (2010), 85–112.
[36] Anders Alnor Mathiasen and Andreas Pavlogiannis. 2021. The fine-grained and parallel complexity of andersen's pointer analysis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29.
[37] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40.
[38] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* (2014).
[39] Yann Ramusat, Silviu Maniu, and Pierre Senellart. 2021. A Practical Dynamic Programming Approach to Datalog Provenance Computation. *CoRR* abs/2112.01132 (2021).
[40] Yann Ramusat, Silviu Maniu, and Pierre Senellart. 2021. Provenance-Based Algorithms for Rich Queries over Graph Databases. In *EDBT*. OpenProceedings.org, 73–84.
[41] Thomas W. Reps. 1998. Program analysis via graph reachability. *Inf. Softw. Technol.* 40, 11-12 (1998), 701–726.
[42] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *CC*. ACM, 196–206.
[43] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis. *Proc. VLDB Endow.* 6, 14 (2013), 1906–1917. https://doi.org/10.14778/2556549.2556572
[44] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (2015), 1–69.
[45] Jeffrey D. Ullman and Allen Van Gelder. 1988. Parallel Complexity of Logical Query Programs. *Algorithmica* 3 (1988), 5–42.
[46] Leslie Valiant. 1974. General context-free recognition in less than cubic time. (1974).
[47] Moshe Y Vardi. 1982. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*. 137–146.
[48] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *SIGMOD Conference*. ACM, 1969–1981.
[49] Stephen Warshall. 1962. A theorem on boolean matrices. *Journal of the ACM (JACM)* 9, 1 (1962), 11–12.
[50] Mihalis Yannakakis. 1990. Graph-Theoretic Methods in Database Theory. In *PODS*. ACM Press, 230–242.
[51] Clement Tak Yu and Meral Z Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979*. IEEE, 306–312.
[52] Hangdong Zhao, Shaleen Deep, Paraschos Koutris, Sudeepa Roy, and Val Tannen. 2024. Evaluating Datalog over Semirings: A Grounding-based Approach. *arXiv preprint arXiv:2403.12436* (2024).

A MISSING DETAILS FROM SECTION 4

A.1 Asynchronous Fixpoint Evaluation

The result in this part on asynchronous evaluation for Datalog is folklore, but we formalize it in a rigorous and general way. The definitions and lemmas established are used in the proof of Theorem 3.5 and Theorem 3.6, to show convergence.

Let G be a grounding containing k rules over σ. Suppose we have k posets $\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_k$ with minimum elements $\bot_1, \ldots, \bot_k$ respectively. Let $\{f_i\}_{i\in[k]}$ be a family of k motonone functions, where

$$f_i : \mathbf{L}_1 \times \cdots \times \mathbf{L}_k \to \mathbf{L}_i,$$

associates to the right-hand side of the i-th rule (the multivariate polynomial) in G. Let $\bot = (\bot_1, \ldots, \bot_k)$ and the ICO of G is

$$\boldsymbol{f} = (f_1, \ldots, f_k) : \mathbf{L}_1 \times \cdots \times \mathbf{L}_k \to \mathbf{L}_1 \times \cdots \times \mathbf{L}_k.$$

The Kleene sequence (or ascending Kleene chain) attempts to evaluate the grounding (if there is a fixpoint) through the following (ascending) sequence:

$$\boldsymbol{h}^{(0)} = \bot, \quad \boldsymbol{h}^{(1)} = \boldsymbol{f}(\boldsymbol{h}^{(0)}), \quad \boldsymbol{h}^{(2)} = \boldsymbol{f}(\boldsymbol{h}^{(1)}), \quad \ldots$$

Here $\boldsymbol{h}^{(t)} = (h_1^{(t)}, \ldots, h_k^{(t)})$, $t \geq 0$, is the vector of values for the posets at the t-th iteration (applying $\boldsymbol{f}$ for t times).

We say that the Kleene sequence converges in t steps if $\boldsymbol{h}^{(t+1)} = \boldsymbol{h}^{(t)}$. Note that the Kleene sequence updates $\boldsymbol{h}^{(t)}$ by using the values of the previous iteration simultaneously. We show that we can obtain convergence if $\boldsymbol{h}^{(t)}$ is updated asynchronously.

Definition A.1. An *asynchronous Kleene chain* is a chain where $\boldsymbol{\hbar}^{(0)} = \bot$ and for any $t \geq 0$:

$$\hbar_i^{(t+1)} = f_i(\hbar_1^{(t_1)}, \hbar_2^{(t_2)}, \ldots, \hbar_n^{(t_n)}), \text{ for some } 0 \leq t_i \leq t.$$

In other words, the current values of each variable can be updated by looking at possibly stale values of the other variables. When $t_1 = t_2 = \cdots = t$, we get exactly a Kleene chain. The following proposition tells us that any asynchronous Kleene chain is always below the Kleene chain.

Proposition A.2. *Let $\boldsymbol{\hbar}^{(0)}, \boldsymbol{\hbar}^{(1)}, \ldots$ be an asynchronous Kleene chain. For every $t \geq 0$, $\boldsymbol{\hbar}^{(t)} \sqsubseteq \boldsymbol{h}^{(t)}$.*

Proof. We will use the inductive hypothesis that for every $t' \leq t$, $\boldsymbol{\hbar}^{(t')} \sqsubseteq \boldsymbol{h}^{(t')}$. For the base case $t = 0$, we have $\boldsymbol{\hbar}^{(0)} = \bot = \boldsymbol{h}^{(0)}$. For the inductive case, assume that $\boldsymbol{\hbar}^{(t')} \sqsubseteq \boldsymbol{h}^{(t')}$ for every $t' \leq t$. Then, we can write:

$$\hbar_j^{(t+1)} = f_i(\hbar_1^{(t_1)}, \hbar_2^{(t_2)}, \ldots, \hbar_n^{(t_n)}) \sqsubseteq f_i(h^{(t_1)}, h^{(t_2)}, \ldots, h^{(t_n)}) \sqsubseteq f_i(h^{(t)}, h^{(t)}, \ldots, h^{(t)}) = h_i^{(t+1)}$$

The first inequality follows from the inductive hypothesis, and the second inequality is implied by the monotonicity of the Kleene chain. Both inequalities use the monotonicity of the function f_i. □

A.2 Proof of Theorem 3.5

In this part, we complete the proof of the theorem, by showing that Algorithm 1 correctly computes the solution with the desired runtime.

Proof of Theorem 3.5. Algorithm 1 terminates when no variable changes value, in which case we have obtained a fixpoint. By Proposition A.2, we have reached the least fixpoint (since we can view the updates as an asynchronous Kleene chain).

We now analyze its runtime. The hash table construction and the variable initialization cost time $O(|G|)$. For the while loop, observe that each operation in Line 7-10 costs $O(1)$ time. Hence, it

suffices to bound the number of times we visit each equation in G. For this, note that to consider the equation $y = x \circledast z$, either x or z needs to be updated. However, since the semiring has rank r, each variable can be updated at most r times. Hence, an equation can be considered at most $2r$ times in the while loop. □

A.3 Proof of Theorem 3.6

In the following, we assume G to be a 2-canonical grounding after applying Lemma 4.1. First, we establish the following two lemmas throughout the while-loop of Algorithm 2. The first one is the following:

Lemma A.3. *For any IDB variable x in G, $\mathbf{0} \sqsubseteq x^{(1)} \sqsubseteq x^{(2)} \sqsubseteq \cdots$, where $x^{(0)} = \mathbf{0}, x^{(1)}, x^{(2)}, \ldots$ are the values of x in the asynchronous Kleene chain constructed from the while-loop of Algorithm 2.*

Proof. The base case is the first pop, say it pops the variable x_1. Trivially, $\mathbf{0} \sqsubseteq x_1^{(1)} = x_1^{(2)} = \cdots$ because x_1 is fixed onwards. Then, for an IDB variable $y \notin E[x]$, $y^{(1)} = y^{(2)}$, else $y^{(1)} = \mathbf{0} \circledast z^{(0)} \sqsubseteq x_1^{(1)} \circledast z^{(1)} = y^{(2)}$.

Now in the inductive case (at the $(i+1)$-th pop of the variable x_i), the inductive hypothesis is that for each IDB variable x, $\mathbf{0} \sqsubseteq x^{(1)} \sqsubseteq x^{(2)} \sqsubseteq \cdots \sqsubseteq x^{(i)}$. Then, for any IDB variable y, $y^{(i)} = x_i^{(i-1)} \circledast z^{(i-1)} \sqsubseteq x_i^{(i)} \circledast z^{(i)} = y^{(i+1)}$ by monotonicity of both $\oplus, \otimes$. □

The second invariant on the while-loop of Algorithm 2 is stated as the following:

Lemma A.4. *The values of IDB variables popped off the queue are non-increasing w.r.t. to $\sqsubseteq$.*

Proof. Let $x^{(i)}$ $(i \geq 1)$ denote the value of variable x at the i-th pop. If $(x_1, x_2, \ldots)$ is the sequence of variables that get popped off the queue, then we want to show that $\mathbf{1} \sqsupseteq x_1^{(1)} \sqsupseteq x_2^{(2)} \sqsupseteq \cdots \sqsupseteq \mathbf{0}$. We prove it via induction. Observe that x_i will bind to $x_i^{(i)}$ after the i-th pop. Thus, $x_i^{(i)} = x_i^{(j)}$ for any $j \geq i$.

The base case is trivial, since $\mathbf{1} \sqsupseteq s \sqsupseteq \mathbf{0}$ holds for any $s \in \boldsymbol{D}$. Now, inductively, suppose we pop x_i $(i \geq 2)$ at some point and $\mathbf{1} \sqsupseteq x_1^{(1)} \sqsupseteq x_2^{(2)} \sqsupseteq \cdots \sqsupseteq x_i^{(i)}$ is the sequence of already popped values. We will show that $x_i^{(i)} \sqsupseteq x_{i+1}^{(i+1)}$.

We distinguish two cases. Suppose that x_i is not in the right-hand side of the equation of x_{i+1}. Then, $x_i^{(i)} \sqsupseteq x_{i+1}^{(i)} = x_{i+1}^{(i+1)}$, where the first inequality comes from the fact that $x_i^{(i)}$ is popped off as a max element in the queue at iteration i.

Otherwise, we have in G exactly one equation of the form $x_{i+1} = x_i \circledast z$ (z may be a EDB coefficient). Now there are two cases depending on the operator:

(1) $x_{i+1} = x_i \otimes z$. By superiority of $\otimes$, $x_i^{(i)} \sqsupseteq x_i^{(i)} \otimes z^{(i)} = x_{i+1}^{(i+1)}$.
(2) $x_{i+1} = x_i \oplus z$. If $x_i^{(i)} \sqsupseteq z^{(i)}$, then $x_{i+1}^{(i+1)} = x_i^{(i)}$. Otherwise, $x_i^{(i)} \sqsubset z^{(i)}$. If z is an IDB variable, we get a contradiction since x_i is not a max value at the i-th pop. If z is an EDB coefficient, then $z^{(i-1)} = z^{(i)}$ and thus
$$x_{i+1}^{(i)} = x_i^{(i-1)} \oplus z^{(i-1)} = x_i^{(i-1)} \oplus z^{(i)} \sqsupseteq z^{(i)} \sqsupset x_i^{(i)}$$
However, $x_{i+1}^{(i)} \sqsupset x_i^{(i)}$ is a contradiction since then x_{i+1} would have been the i-th pop instead of x_i.

□

Now we are ready to present the proof for Theorem 3.6. In particular, we will use the two lemmas established above (Lemmas A.3 and A.4) to show that the algorithm terminates and returns the desired fixpoint. Then we will analyze its runtime, as predicated in Theorem 3.6.

Proof of Theorem 3.6. We show that Algorithm 2 terminates and yields a fixpoint, and by Proposition A.2, we get the least fixpoint for free because the Dijkstra-style while-loop is a special case of asynchronous Kleene chain. First, it is easy to see that Algorithm 2 terminates because the size of q always decreases by 1 (popped IDB variables will not be pushed back into the queue, using $\mathcal{F}$ for bookkeeping of already popped variables).

We now show that the returned $h(\cdot)$ is a fixpoint. To prove the fixpoint, we show that once the queue pops off an IDB variable x as some value $h(x)$, the equation with x at the left-hand side in G, say $x = y \circledast z$, always holds from that point onwards. We prove this by contradiction. Suppose that $x = y \circledast z$ causes the earliest violation in the underlying asynchronous Kleene chain. By Lemma A.3, the values of y, z are non-decreasing w.r.t. $\sqsubseteq$. Therefore, the only way to violate $x = y \circledast z$ is that the value of an IDB variable y (or z) strictly increases after x is popped off the queue. On the other hand, the queue will eventually pop off that IDB variable (y or z) $\sqsupset x$, since Algorithm 2 always terminates and y, z can not grow smaller by Lemma A.3. Yet, it is a direct contradiction to Lemma A.4.

Lastly, we provide the runtime analysis. Using similar arguments in the proof of Theorem 3.5, Lines 1-7 of Algorithm 2 can execute in $O(|G|)$ time. For its while-loop, each iteration binds an IDB variable x and calls the insert operations of the priority queue q at most $\deg(x)$ times, where $\deg(x)$ denotes the number of the equations in $E[x]$. A 2-canonical grounding has that $\sum_{x \in E} \deg(x) = 2|G|$. Using a classic max heap implementation of a priority queue, inserts (or updates) of values can run in $O(\log |G|)$ time. Thus, the overall runtime is

$$O\left(|G| + \sum_{x \in E} \deg(x) \cdot \log |G|\right) = O(|G| \cdot \log |G|).$$

□

B LOWER BOUNDS

In this section, we will prove the lower bounds stated in Theorem 3.4. Our lower bounds are based on the following proposition, which constructs a class of Datalog programs with the appropriate width that decides whether a undirected graph G contains a clique.

Proposition B.1. *Take an integer $k \geq 2$ and a rational number $w \geq 1$ such that $k \cdot w$ is an integer. There is a (non-recursive) Datalog program P with arity k such that* $\mathrm{subw}(P) \leq w$, *and the target Boolean IDB predicate $Q()$ for P decides whether a undirected graph G has a clique of size $\ell = (k-1) + k \cdot w$.*

Proof. Let $R(x, y)$ be the binary predicate that represents the edges of G, and $V(x)$ be the unary predicate that represents the nodes of G. It will be helpful to distinguish the ℓ variables of the clique we want to compute into k variables $x_1, \ldots, x_k$ and the remaining $\ell - k$ variables $y_1, y_2, \ldots, y_{\ell-k}$. Note that $\ell - k = k \cdot w - 1 \geq 2 - 1 = 1$, so there is at least one y-variable.

Initially, we compute all cliques of size k in the graph:

$$T(z_1, \ldots, z_k) \leftarrow V(z_1) \otimes V(z_2) \otimes \cdots \otimes V(z_k).$$
$$C(z_1, \ldots, z_k) \leftarrow T(z_1, \ldots, z_k) \otimes \bigotimes_{1 \leq i < j \leq k} R(z_i, z_j).$$

Both rules are acyclic (so their submodular width is 1) and the arity of the two intensional predicates we introduce is k. Next, we introduce the following rule:

$$S(x_1, \ldots, x_k) \leftarrow \bigoplus_{y_1, \ldots, y_{\ell-k}} U_1[x_1, y_1, y_2, \ldots, y_{\ell-k}] \otimes U_2[x_2, y_1, y_2, \ldots, y_{\ell-k}] \otimes \ldots$$

$$\otimes U_k[x_k, y_1, y_2, \ldots, y_{\ell-k}].$$

where $U_i[x_i, y_1, y_2, \ldots, y_{\ell-k}]$ stands for the product of the following sequence of atoms:

$$\{C(z_1, \ldots, z_k) \mid z_1, \ldots, z_k \text{ are all possible sets of size } k \text{ from } \{x_i, y_1, y_2, \ldots, y_{\ell-k}\}\}$$

Note that this is well-defined, since $\ell - k + 1 = k \cdot w$ and $w \geq 1$. We will argue that the submodular width of this rule for $S(x_1, \ldots, x_k)$ is w. The idea is that every tree decomposition has the exact same set of bags: $\{x_i, y_1, y_2, \ldots, y_{\ell-k}\}$ for $i = 1, \ldots \ell$. This follows from [7] and the fact that the clique-graph of the hypergraph is chordal (i.e. any cycle in the clique-graph of length greater than 3 has a chord).

Now, each bag has $\ell - k + 1$ variables. Pick a set of $\ell - k + 1$ hyperedges (of size k) such that each variable is covered exactly k times (this selection is possible because every possible k-sized hyperedge is in the hypergraph). By assigning a weight of $1/k$ to any such hyperedge, we have a fractional edge cover of size $(\ell - k + 1)/k = w$.

Finally, we need to check that each tuple in S forms an k-clique, which can be done through the following rule:

$$Q() \leftarrow \bigoplus_{x_1, x_2, \ldots, x_k} S(x_1, \ldots, x_k) \otimes \bigotimes_{1 \leq i < j \leq k} R(x_i, x_j).$$

Observe that the last rule is also acyclic, so its submodular width is 1. □

B.1 Missing Proof for Part (1) of Theorem 3.4

Provenance Polynomials. Consider a non-recursive Datalog program P over a semiring σ with a single Boolean target $Q()$. Then, we can generate a provenance polynomial for P by interpreting each EDB fact $R(\mathbf{c})$ as a variable $x_{\mathbf{c}}$. We call this the provenance polynomial of P. The non-recursive condition guarantees that the provenance polynomial is finite.

Circuits over Semirings. We define a circuit F over a semiring σ to be a Directed Acyclic Graph (DAG) with input nodes (with fan-in 0) variables $x_1, \ldots, x_m$ and the constants $\mathbf{0}, \mathbf{1} \in \boldsymbol{D}$. Every other node is labelled by $\oplus$ or $\otimes$ and has fan-in 2; these nodes are called $\oplus$-gates and $\otimes$-gates respectively. An output gate of F is any gate with fanout 0. The size of the circuit F, denoted $|F|$, is the number of gates in F. We say that F *evaluates a polynomial* f over the semiring σ if for any input values that the variables $x_1, \ldots, x_m$ take over the domain $\boldsymbol{D}$ of the semiring, the output value of the circuit is the corresponding value of f.

Proposition B.2. *Let G be a non-recursive grounded Datalog program over a semiring σ with a single Boolean target $Q()$. Then, there is a circuit F over σ that evaluates the provenance polynomial of G and F has size $O(|G|)$.*

Proof. Given G, we will construct a semiring circuit F as follows. Every EDB fact $R_J(\mathbf{c}_J)$ maps to a distinct input gate $z_{R_J(\mathbf{c}_J)}$ of the circuit. For every IDB fact $S_J(\mathbf{c}_J)$, we introduce an $\oplus$-gate $z_{S_J(\mathbf{c}_J)}$. Take any rule r with head $S_J(\mathbf{c}_J)$ and body $T_{J_1}(\mathbf{c}_{J_1}), \ldots, T_{J_k}(\mathbf{c}_{J_k})$. We then create a subcircuit of the form

$$(((z_{T_{J_1}(\mathbf{c}_{J_1})} \otimes z_{T_{J_2}(\mathbf{c}_{J_2})}) \otimes z_{T_{J_3}(\mathbf{c}_{J_3})}) \ldots)$$

and name the top gate of this subcircuit as $y^r_{S_J(\mathbf{c}_J)}$. Let $r_1, r_2, \ldots, r_s$ be the rules with the same head $S_J(\mathbf{c}_J)$. Then, we make $z_{S_J(\mathbf{c}_J)}$ be the top gate of the subcircuit

$$\left(y^{r_1}_{S_J(\mathbf{c}_J)} \oplus y^{r_2}_{S_J(\mathbf{c}_J)}\right) \oplus y^{r_3}_{S_J(\mathbf{c}_J)} \cdots$$

The gate corresponding to $z_{q()}$ is the output gate. We now bound the size of the circuit F. Note that for every rule with k atoms in the body we introduce $(k-1)$ $\otimes$-gates and one $\oplus$-gate. This concludes the proof. □

We will use the following proposition, which is an unconditional lower bound on the size of a semiring circuit that computes the k-clique.

Proposition B.3 ([27]). *Any semiring circuit that evaluates the k-clique polynomial over* $\mathsf{Trop}^+ = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$ *has size* $\Omega(n^k)$.

We are now ready to prove the first part of Theorem 3.4.

Proof of Theorem 3.4, part (1). Consider the program P from Proposition B.1. This program returns the minimum weight of a k-clique in a graph G, where $k = \ell - 1 + \ell \cdot w$. Let f_P be the provenance polynomial for P. From Proposition B.3, we know that any semiring circuit that evaluates f_P over the Trop^+ has size $\Omega(n^k)$. Applying finally Proposition B.2, we get that the size of any grounded program equivalent to P over Trop^+ must also be $\Omega(n^k)$. □

B.2 Missing Proof for Part (2) of Theorem 3.4

Conditional Lower Bounds. Using the construction of Proposition B.1, we can obtain directly a conditional lower bound on the running time using problems from fine-grained complexity. In particular, we consider the *min-weight ℓ-Clique hypothesis* [33]: there is no algorithm that can find a ℓ-Clique of minimum total edge weight in an n-node graph with nonnegative integer edge weights in time $O(n^{\ell-c})$ for some constant $c > 0$. Observing that running the Datalog program of Proposition B.1 over the tropical semiring corresponds to solving the min weight ℓ-Clique problem, we obtain the following proposition, which proves part (2) of Theorem 3.4.

Proposition B.4. *Take any integer $k \geq 2$ and any rational number $w \geq 1$ such that $k \cdot w$ is an integer. There is a (non-recursive, non-linear) Datalog program P over the tropical semiring with IDB and EDB arities at most k such that* $\mathsf{subw}(P) = w$, *such that no algorithm can compute it in time $O(n^{kw+k-1-o(1)})$ assuming the min-weight k-Clique hypothesis.*

C EXAMPLE APPLICATION OF PROPOSITION 5.4

Example C.1. Consider the following linear rule (essentially a chain query) of a Datalog program with the only IDB $T(x_2, x_3)$ in the body:

$$T(x_1, x_5) \leftarrow \bigoplus_{\mathbf{x}_{234}} R_{12}(x_1, x_2) \otimes T(x_2, x_3) \otimes R_{34}(x_3, x_4) \otimes R_{45}(x_4, x_5).$$

Suppose that we elect R_{12} as the root bag for the join tree shown in Figure C.1. Applying the construction of Theorem 3.2 on the join tree is suboptimal: the grounding of the rule $T_{25}(x_2, x_5) \leftarrow \bigoplus_{x_3} T(x_2, x_3) \otimes T_{35}(x_3, x_5)$ requires $O(n^3)$ since every possible combination of (x_2, x_3, x_5) in the active domain must be considered from $T(x_2, x_3) \otimes T_{35}(x_3, x_5)$. In constrast, Proposition 5.4 avoids the situation where an intermediate rule contains more than one IDB in the body. The algorithm runs in a top-down fashion first from the IDB node and constructs the grounding for $T_{25}(x_2, x_5)$ via the following:

$$T_{24}(x_2, x_4) \leftarrow \bigoplus_{x_3} T(x_2, x_3) \otimes R_{34}(x_3, x_4), \qquad T_{25}(x_2, x_5) \leftarrow \bigoplus_{x_4} T_{24}(x_2, x_4) \otimes R_{45}(x_4, x_5).$$

Now both intermediate rules have a grounding of $O(m \cdot n)$. Finally, we ground the rule $T(x_1, x_5) \leftarrow \bigoplus_{x_2} R_{12}(x_1, x_2) \otimes T_{25}(x_2, x_5)$ in $O(m \cdot n)$ time. Indeed, one can check that it is an σ-equivalent

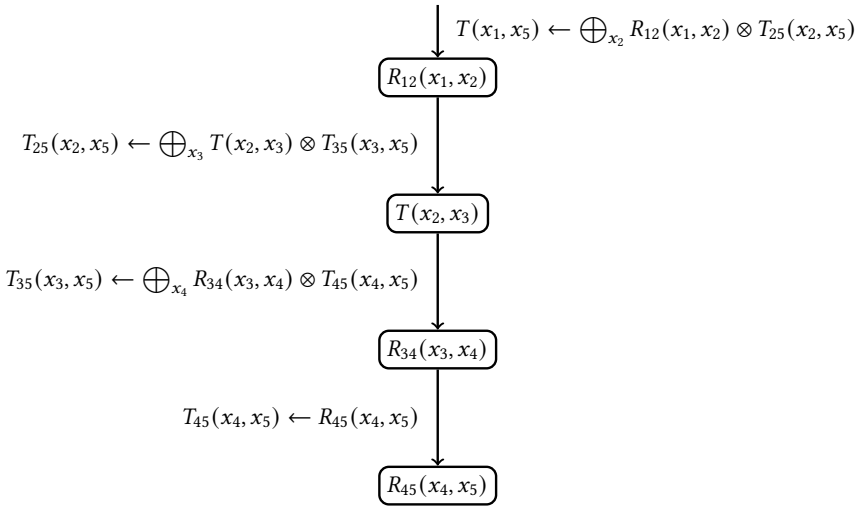

Fig. 2. A join tree with its corresponding rewriting for Example C.1 when using Theorem 3.2.

rewriting of the program since

$$
\begin{aligned}
\bigoplus_{x_2} R_{12}(x_1, x_2) \otimes T_{25}(x_2, x_5) &= \underbrace{\bigoplus_{x_2} R_{12}(x_1, x_2) \otimes \left(\bigoplus_{x_4} T_{24}(x_2, x_4) \otimes R_{45}(x_4, x_5) \right)}_{\text{(substitute } T_{25}(x_2, x_5)\text{)}} \\
&= \underbrace{\bigoplus_{\mathbf{x}_{24}} R_{12}(x_1, x_2) \otimes T_{24}(x_2, x_4) \otimes R_{45}(x_4, x_5)}_{\text{(pull out the summation } \bigoplus_{x_4}\text{)}} \\
&= \underbrace{\bigoplus_{\mathbf{x}_{24}} R_{12}(x_1, x_2) \otimes \left(\bigoplus_{x_3} T(x_2, x_3) \otimes R_{34}(x_3, x_4) \right) \otimes R_{45}(x_4, x_5)}_{\text{(substitute } T_{24}(x_2, x_4)\text{)}} \\
&= \underbrace{\bigoplus_{\mathbf{x}_{234}} R_{12}(x_1, x_2) \otimes T(x_2, x_3) \otimes R_{34}(x_3, x_4) \otimes R_{45}(x_4, x_5)}_{\text{(pull out the summation } \bigoplus_{x_3}\text{)}} \\
&= \underbrace{T(x_1, x_5)}_{\text{(by definition)}}
\end{aligned}
$$

Received December 2023; revised February 2024; accepted March 2024