Circuits for Datalog Provenance

Daniel Deutch
Tel Aviv University
danielde@post.tau.ac.il

Tova Milo
Tel Aviv University
milo@post.tau.ac.il

Sudeepa Roy
University of Washington
sudeepa@cs.washington.edu

Val Tannen
University of Pennsylvania
val@cis.upenn.edu

ABSTRACT

The annotation of the results of database queries with provenance information has many applications. This paper studies provenance for datalog queries. We start by considering provenance representation by (positive) Boolean expressions, as pioneered in the theories of incomplete and probabilistic databases. We show that even for linear datalog programs the representation of provenance using Boolean expressions incurs a super-polynomial size blowup in data complexity. We address this with an approach that is novel in provenance studies, showing that we can construct in PTIME poly-size (data complexity) provenance representations as Boolean circuits. Then we present optimization techniques that embed the construction of circuits into seminaive datalog evaluation, and further reduce the size of the circuits. We also illustrate the usefulness of our approach in multiple application domains such as query evaluation in probabilistic databases, and in deletion propagation. Next, we study the possibility of extending the circuit approach to the more general framework of semiring annotations introduced in earlier work. We show that for a large and useful class of provenance semirings, we can construct in PTIME poly-size circuits that capture the provenance.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; H.2.4 [**Database Management**]: Systems—*Relational databases*; H.2.m [**Database Management**]: Miscellaneous

General Terms

Algorithms, Theory

Keywords

Datalog, Provenance, Circuit, Semiring, Time complexity, Lower bounds

1. INTRODUCTION

Recording provenance information for query results, that explains the computational process leading to their generation, is now a common technique [22, 9, 32, 13, 37] with applications such as view maintenance, trust assessment, or query answering in probabilistic databases (see, *e.g.*, [26, 36, 38]). This paper focuses on provenance for *datalog queries* [2] and proposes a novel, circuit-based provenance representation, as well as an efficient way to compute provenance, whenever possible, using this representation. Datalog has recently re-gained popularity in the Web-context and is used in diverse applications, from data extraction in Web pages to communication routing [6, 7], and in commercial systems [25, 17, 3]. Finding suitable provenance mechanism for datalog is thus of both theoretical and practical importance. As was explained in [27], the presence of recursion creates new challenges in modeling provenance for datalog: in general an answer tuple can be derived in infinitely many different ways. Fortunately, previous work has identified provenance models whose elements are intrinsically finite and that nonetheless are appropriate for datalog, albeit by necessity providing only a summary of the full provenance. As we show in this paper, even tracking such a finite summary must be done with care to avoid super-polynomial blow-ups in size.

To deal with such blow-ups, we propose a provenance model for datalog that is based on *circuits*. Circuits are extensively studied, but to our knowledge this is the first work that proposes to use circuits for database provenance. Besides the space efficiency of this representation (proved in the sequel), it further has the potential of allowing data provenance to benefit from developments such as circuit optimizations and the use of circuits for parallel computing. Further comparisons to other approaches for provenance are given in the Related Work section. The work presented in this paper is organized as follows.

Circuits for Positive Boolean Provenance (Section 3). We start by considering a particular, yet important, case where provenance information is in fact *positive* Boolean formulas over some finite domain of variables, used to annotate tuples. For reasons that will become apparent when we generalize beyond Boolean provenance, we refer to relations whose tuples are associated with positive Boolean annotations, as PosBool(X)-relations. We note that PosBool(X)-relations form a particular class of c-tables [29], which was proven useful in the context of incomplete and probabilistic databases [36]. In these contexts one commonly uses PosBool(X)-relations as representation systems by consid-

10.5441/002/icdt.2014.22

ering the set of possible worlds they define via valuations to the underlying variables. Then, given such a database D with PosBool(X)-relations and a query, the goal is to compute a PosBool(X)-relation R that is a representation of all results of the query when applied to possible worlds of D. In provenance terminology, we *propagate* Boolean provenance annotations from D to R.

Importantly, we show that for Datalog (unlike e.g. positive relational algebra), such propagation of Boolean formulas that is *faithful* (i.e. the output encodes correct possible worlds) may lead to an inevitable super-polynomial blow-up of the size of output representation with respect to that of the input representation. Specifically for input of size n, the output size may be $n^{\Omega(\log n)}$.

This blowup stems from the use of *formulas* as annotations. So instead, we propose to capture annotations with (positive) *Boolean circuits*. Simply put, a positive Boolean circuit on a set of variables X is a DAG whose "sinks" are associated with elements of X and other nodes are associated with the connectives $\wedge$ and $\vee$. It is well-known that *Boolean circuits* may capture the same Boolean functions as formulas, while leading in some cases to an asymptotically smaller size. Indeed, we show that by using circuits as annotations, and propagating them instead of formulas, we can guarantee a polynomial size representation of the output. Moreover, circuits allow sharing of sub-expressions so instead of computing a separate circuit for every output tuple, we show that we can generate a single circuit with multiple "entry" points, and use it for annotation of *all* output tuples. We analyze the worst case complexity of generating the circuit and demonstrate applications of the construction for deletion propagation and probability computation.

Efficient Provenance Generation (Section 4). The previously described construction is geared towards studying the worst-case complexity incurred by the circuit-based approach. As such, it may not be very efficient to implement directly, for the following reason. It is common practice in provenance propagation to compute the output provenance along side with computing the output relation itself. This allows provenance generation algorithms to benefit from query optimizations [1], and in principle to be executed by a provenance-aware query engine. However, the construction in Section 3 generates provenance separately from query evaluation.

To this end, we show how to generate provenance along side with a particularly efficient evaluation algorithm for Datalog queries, namely semi-naive evaluation. We show that the performance of the circuit generation algorithm and the obtained circuits size are effected by the same factors that dictate the performance of semi-naive evaluation; this means that provenance tracking can benefit from the semi-naive optimizations.

Semiring-based Provenance (Section 5). Finally, we go beyond Boolean annotations, and we consider *semiring annotations*. We note that PosBool(X) relations considered thus far are particular case of the more general notion of K-relations, i.e. relations for which tuple provenance is an element of a commutative semiring K. Building on and extending our development for the Boolean case, we consider K-circuits that compute values in an arbitrary commutative semiring K. We extend the desideratum of faithful representation to the case where the annotations come from such K. We then show that it is impossible to apply the circuit approach for arbitrary commutative semirings, even when restricting the attention to ω-*continuous* semirings. Intuitively, these are semirings that support infinite sums, required to capture the infinitely many derivations possible in the context of Datalog. However, we show that for a specific important class of semirings, namely *absorptive* ones, K-circuits can be used to efficiently capture their provenance. In fact we show a stronger result: we introduce a new provenance semiring called Sorp(X), which is the "most general absorptive semiring". Then we show that one can generate datalog provenance in the form of Sorp(X)-circuits and then *specialize* them in a correct way (to be defined) in any absorptive semiring. Such ability to compute a general representation and later specialize it was proven crucial in a variety of applications [27]. To conclude, we consider further provenance semirings beyond absorptive ones. There is a particular semiring that is ω-continuous but not absorptive. This is $Why(X)$ (capturing exactly the why-provenance [12]). We show that while a general construction fails at specializing to Why(X), there is a direct construction that is specific to $Why(X)$ that allows to capture datalog provenance in this semiring. Finally, we briefly consider the case of arbitrary commutative semirings, not necessarily ω-continuous, where full datalog provenance may not be captured. We propose for such semiring a practical approach, based on capturing only finitely many derivations. We conclude with a discussion on related work in Section 6.

[1]Indeed, we show that equivalent provenance circuits are obtained for equivalent queries.

2. BACKGROUND

First we review some background on datalog and provenance, focusing on PosBool(X)-databases.

2.1 Datalog

We assume that the reader is familiar with standard datalog concepts [2]. Here we review the terminology and we illustrate it with an example. A datalog program P consists of rules over a relational schema. The schema is partitioned into *extensional* (or *edb*) and *intensional* (or *idb*) relation names (viewed as predicates). Each rule has a head (one idb predicate) and a body (0 or more edb or idb predicates). We do not find it useful to distinguish an "output" relation among the idb ones. Instead, we think of a datalog program as computing all its idb relations, therefore its usual set semantics is a mapping from edb instances to idb instances. Given a database D, we thus use $P(D)$ to denote the idb instances computed as the result of evaluating P on the edb instances of D.

EXAMPLE 1. *We will use the following datalog program as our running example. Intuitively, it computes the transitive closure in a directed graph, then selects nodes on a path leading to b:*

$$T(x,y) \ :- \ R(x,y) \quad (1)$$
$$T(x,y) \ :- \ R(x,z), T(z,y) \quad (2)$$
$$S(x) \ :- \ T(x,b) \quad (3)$$

Here Σ_e consists of the single edb relation R and Σ_i consists of the two idb relations T and S. For example, given

A	B	Ann
a	a	p
a	b	q

(a) R

X	Ann
a	$q \vee (p \wedge q) \vee ((p \wedge p) \wedge q) \vee \ldots$

(b) S (recursive query)

A	B	Ann
a	a	$p \vee (p \wedge p)$
a	b	$q \vee (p \wedge q)$

(c) T (non-recursive query)

X	Ann
a	$q \vee (p \wedge q)$

(d) S (non-recursive query)

Figure 1: Running example: edb relation R and idb relations S, T, the annotations have *not* been simplified.

the edb instance $R = \{(a,a),(a,b)\}$ *the program computes the idb instance* $T = \{(a,a),(a,b)\}, S = \{a\}$.

Any tuple in the edb instances in D is called an *edb tuple* or *edb fact* (*e.g.*, $R(a,a)$ and $R(a,b)$), and any tuple in the idb instances generated by executing P on D is called an *idb tuple* or *idb fact* (*e.g.*, $T(a,a), T(a,b)$, and $S(a)$).

2.2 PosBool(X)-Databases

We next introduce the concept of provenance through a simple yet important restricted case, namely the one of positive Boolean annotations. The basic idea is that tuples are *annotated* with elements of some mathematical structure, in this case the equivalence class of all positive Boolean expressions over some set of Boolean indeterminate ("variables") X (as usual, two Boolean expressions are equivalent if they give the same truth value for any assignment). The set of all such equivalence classes is denoted PosBool(X), and a relation whose tuples are annotated with elements of PosBool(X) is called a PosBool(X)-relation. PosBool(X) captures enough provenance to serve in the intensional semantics of queries on incomplete [29] and probabilistic data [36].

The possible truth assignments to Boolean variables correspond to possible sub-instances ("possible worlds") of the relation, consisting exactly of tuples whose annotating formula is satisfied by the assignment.

Definition 1. *A* PosBool(X)*-relation is a pair* $(R, \mathtt{Ann})$ *where* R *is a (finite) relation and* Ann *is an annotation function, mapping the tuples of* R *to elements of* PosBool(X). *Given a truth assignment* α *for variables in* X *we define* $R_\alpha = \{t \in R : \ \alpha \vdash \mathtt{Ann}(t)\}$. *The set of possible worlds of* $(R, \mathtt{Ann})$ *is defined as* $\bigcup_\alpha R_\alpha$.

We will denote $(R, \mathtt{Ann})$ simply by R where the annotation Ann is clear from the context. A PosBool(X)-database is then a collection of PosBool(X)-relations, and the notion of possible worlds extends naturally.

Example 2. *Figure 1a depicts a* PosBool(X)*-relation* R*, with* $X = \{p, q\}$. *It admits four possible worlds corresponding to the assignments of true and false to* p *and* q*; for instance if* α *maps* p *to true and* q *to false then* R_α *consists of the single tuple* (a,a).

Intuitively, the annotation for a tuple in a PosBool(X)-relation corresponds to its provenance. We are then interested in the way provenance propagates through query evaluation. For positive relational algebra, the approach taken in [27] defines provenance propagation in a way that closely follows the propagation of data itself [2]. Intuitively, when two tuples are *joined*, then their Boolean annotation are combined with a *conjunction*, since intuitively the worlds in which the joined tuple is present are exactly those in which the two tuples are present. Similarly, *projected tuples* are associated with annotations that are the *disjunction* of all tuples that could *alternatively* be used to derive the projected tuples.

Provenance and data propagation is defined similarly for all positive relational algebra (RA^+) operators, and so given a RA^+ query Q and a PosBool(X)-database D, $Q(D)$ denoting the output PosBool(X)-relation is well-defined. We term this a *semantics of positive relational algebra for* PosBool(X)*-databases*. A fundamental property that such semantics should have is that $Q(D)$ is a *faithful representation*:

- (Faithful representation) For every PosBool(X)-database D, RA^+ query Q, and truth assignment α to Boolean variables in X, $Q(D_\alpha) = [Q(D)]_\alpha$.

Note that this implies that annotations in $Q(D)$ capture exactly the same possible worlds, but even more than that, these worlds correspond to the same truth assignments, as those obtained by first applying the assignment to all relations in D and then evaluating Q on the obtained relations [3].

Naturally, maintaining provenance requires some overhead in size. Another favorable property of the approach for RA^+ queries is that the size of obtained provenance is only polynomial in the size of the input instance:

- (Poly-size overhead) For every PosBool(X)-database D and RA^+ query Q, the size of $Q(D)$, including annotations, is polynomial in that of D.

Example 3. *To illustrate these concepts for a* RA^+ *query, replace* $T(z,y)$ *by* $R(z,y)$ *in the body of rule (2) of Example 1 to avoid recursion (which now computes all pairs reachable in either one or two steps). The tuple* $T(a,a)$ *can be obtained either directly from* $R(a,a)$ *(by rule (1)), or by joining* $R(a,a)$ *with* $R(a,a)$ *(by rule (2)).* $R(a,a)$ *is annotated with* p*. Thus the annotation of* $T(a,a)$ *has the form* $p \vee (p \wedge p)$ [4]*, intuitively meaning that either using* p *by itself, or using* p *twice can generate* $T(a,a)$ *(see Figure 1c). Annotation of tuple* $S(a)$*, obtained by projection of the tuple* $T(a,b)$*, is shown in Figure 1d.*

2.3 Datalog on PosBool(X)-databases

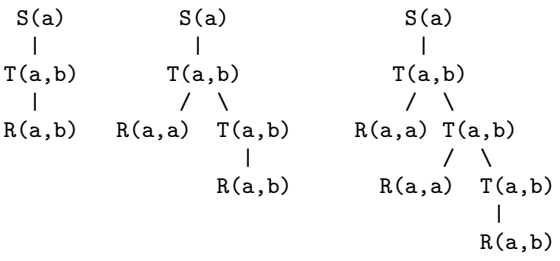
In the same spirit of extending positive relational algebra semantics to annotated relations, for every PosBool(X)-database D and a datalog program P, a *datalog semantics on* PosBool(X)*-databases* defines one or more PosBool(X)-relations collectively denoted by $P(D)$ (depending on the number of idb relations in P). This is done in [27] where

[2] The propagation for the particular Boolean case actually follows already from [29]

[3] The latter is a favorable property for applications such as deletion propagation and probabilistic databases

[4] Throughout this section, we deliberately avoid "simplifications" of Boolean expressions to emphasize the way in which they are generated. Simplifications are discussed in the following section. The non-simplified expressions will also be useful when we consider general semirings in Section 5.

the authors generalize datalog (proof-theoretic) semantics to annotated relations. We review this briefly. A *derivation tree* τ for an idb fact t represents one particular way to obtain t (at the root) from all the edb facts at the leaves of τ; three derivation trees for idb fact $t = S(a)$ in the recursive program in Example 1 are shown below.



This leads to a definition for the annotation $\mathtt{Ann}(t)$ of an idb fact t:

$$\mathtt{Ann}(t) \;=\; \bigvee_{\tau:\ \tau \text{yields } t} \;\; \bigwedge_{t':\ t' \text{ is a leaf}\atop \text{edb fact of } \tau} \mathtt{Ann}(t') \qquad (4)$$

In this formula, the conjunctions are over the (finitely many) annotations of the leaves of each derivation tree. However, the disjunction in this formula can be infinite (but countable), which raises additional problems. Obviously, every infinite Boolean formula is equivalent to a finite one (by idempotence and absorption). It follows from [27] that using this definition leads to a faithful representation for PosBool(X)-databases.

Example 4. *We have shown three derivation trees of idb fact $S(a)$ in our running example above. It is easy to see that in fact there are infinitely many derivation trees for $S(a)$, each of them repeat the leaf $R(a,a)$ zero or more times and then have the leaf $R(a,b)$. Thus*

$$\mathtt{Ann}(S(a)) = q \bigvee (p \wedge q) \bigvee (p \wedge p \wedge q) \bigvee \cdots$$

Naturally, simplifications of the Boolean annotations $\mathtt{Ann}(t)$ are possible to yield finite equivalent Boolean formulas (which is straightforward in the above example yielding $\mathtt{Ann}(S(a)) = q$). However, an important question is whether this finite formula has a poly-size representation in the size of the database instance D (which was the case for RA^+ queries); we address this question in the next section.

3. BOOLEAN PROVENANCE REPRESENTATION

The previous section introduces a formal definition for Boolean provenance for datalog programs on PosBool(X)-databases (by Equation (4)). This definition leads to a faithful representation, but the definition by itself is non-constructive since it may involve infinitely many derivation trees. The obvious choice for a finite representation of provenance is the equivalent Boolean formula of the (possibly infinite) expression in (4). However, our goal is to obtain a faithful representation which is not only finite, but also is of size polynomial in the size of the input database D. It turns out that for datalog, representation via (arbitrary) boolean formulas may fail to satisfy the desiderata. Recall that a datalog semantics for PosBool(X)-databases maps a PosBool(X)-database D and a datalog program P to one or more PosBool(X)-relations (see Section 2.3).

Theorem 1. *For any faithful datalog semantics, there exist datalog program P and* PosBool(X)*-database D such that provenance of the tuples in* PosBool(X)*-relations in $P(D)$ cannot be represented using Boolean formulas of size polynomial in $|D|$.*

Proof. We will use the result by Karchmer and Wigderson [31]: any monotone formula to test st-connectivity in a graph with n nodes has size $n^{\Omega(\log n)}$ (lower bound holds even for undirected graphs). Consider the datalog program P for transitive closure in Example 1, the database D corresponds to a complete graph n having a single edb relation R (the edge relation).

Assume by contradiction the existence of a datalog semantics that annotates each tuple in the idb PosBool(X)-relation T with poly-size Boolean formulas on the edb variables for edges $R(u,v)$; further, the annotation is faithful. Consider the annotation (provenance) of the tuple $T(s,t)$ for two fixed nodes s and t in the graph.

Since $P(D)$ is faithful, for all assignments α to the edb variables, $P(D_\alpha) = [P(D)]_\alpha$. Therefore, the poly-size Boolean formula annotating $T(s,t)$ restricted to α, $\mathtt{Ann}(T(s,t))_\alpha$, is *true* iff t is reachable from s in the graph defined by the edges in R_α, *i.e.*, the edges $R(u,v)$ for which $R(u,v)_\alpha = true$. This contradicts the lower bound in [31]. □

The above theorem motivates us to explore an alternative approach based on *circuit representation.*

3.1 Boolean Provenance Circuits

We review (monotone) Boolean circuits (see e.g. [5]):

Definition 2. *A Boolean circuit over* PosBool(X) *is a labeled Directed Acyclic Graph (DAG). The sinks are labeled by Boolean true, false, or elements of X, and the internal nodes are labeled by either $\wedge$ or $\vee$ (called $\wedge$-node and $\vee$-node respectively). The* size *of a circuit is the number of nodes in the circuit.*

In our context, a $\wedge$-node (resp. a $\vee$-node) will have children that are either $\vee$-nodes (resp. $\wedge$-nodes) or the sinks. Further, each $\vee$-node will be associated with a unique idb variable, and the label X of a non-Boolean sink node will be a unique edb variable[5]*. There will be one or more sources that are $\vee$-nodes.*

A circuit can be much more compact than an equivalent Boolean formula since it can reuse sub-expressions by its DAG structure. Indeed, when we allow the annotations of the tuples in $P(D)$ to be Boolean circuits, we get a poly-size faithful representation:

Theorem 2. *There exists a datalog semantics for* PosBool(X)*-databases such that given any datalog program P and* PosBool(X)*-database D,*

- *it represents provenance of the tuples in the* PosBool(X)*-relations in $P(D)$ as monotone Boolean circuits,* i.e.*, each idb tuple is annotated with a circuit,*
- *$P(D)$ is a faithful representation, and*
- *the size of $P(D)$ (including annotations) is polynomial in that of D.*

[5]For simplicity, we assume that each edb tuple in the input PosBool(X)-database is annotated with a single variable, general annotations can be easily handled by substitution.

We give a constructive proof of the theorem by giving an algorithm (Algorithm 1) to construct the circuits. However, to have an overall compact representation of the annotations, we will use a *shared circuit* for all the idb tuples in $P(D)$. The shared circuit will have multiple source nodes and each idb tuple t in $P(D)$ will have a unique source in the circuit; we will simply annotate t by the corresponding source node in the circuit.

Note. While Algorithm 1 presented below proves Theorem 2, it (unnecessarily) goes, for simplicity, through construction of a system of equations, and therefore is inefficient from a practical point of view. A much more efficient construction (Algorithm 2) is given in Section 4.

Description of Algorithm 1. Algorithm 1 is based on two prior work: The first is [27] that shows a construction of a system of equations (Boolean formulas) whose fixpoint is equivalent to the infinite expression defined above; and the second is [18] (also related here is [35]) that shows that computation of this fixpoint can be achieved through an iterative solution with a small, bounded number of iterations. The high-level construction is given in Algorithm 1, detailed next.

Procedure GenerateEquations (Line 1). We start by generating a *system of equations* EQS, defined as follows. We first generate all possible *candidate idb tuples*, by all possible assignments of the attributes of all idb predicates from the active domain[6]. Note that a candidate idb tuple may or may not be an actual idb tuple or idb fact in $P(D)$ (an idb fact has at least one valid derivation tree from the edb tuples in D). In the next section we give a more efficient algorithm that only considers idb tuples that are idb facts.

For each edb or candidate idb tuple t, we define an edb or idb variable X_t respectively. For edb tuples t, we will assume that $X_t =$ the annotation of t in D which play the role of *constants* in the system of equations; the idb variables act as *variables*. Let τ be an assignment of tuples $t_1^\tau, \cdots, t_{q_\tau}^\tau$ to the atoms in the body of a rule, such that the head of the rule under this assignment is t. The equation for t is the disjunction over all such assignments $X_t = \bigvee_\tau X_{t_1^\tau} \wedge \cdots \wedge X_{t_{q_\tau}^\tau}$ It follows from [27] that the fixpoint of this system of equations is equivalent to the semantics using the derivation trees in Section 2.1.

Example 5. *In our running example, the candidate idb tuples are* $T(a,a), T(a,b), T(b,a), T(b,b), S(a), S(b)$. *The system of equations is*

$$X_{T(a,a)} = p \vee p \wedge X_{T(a,a)} \qquad X_{S(a)} = X_{T(a,b)}$$
$$X_{T(a,b)} = q \vee p \wedge X_{T(a,b)} \qquad X_{S(b)} = X_{T(b,b)}$$

For edb facts $R(a,a)$ *and* $R(a,b)$, *the corresponding annotations* p, q *are used as constants.*

Generating the circuit (Lines 2-7). Now we have obtained a recursive system of equations, and we are looking for a fixpoint solution. It was shown in [18] that such solution can be achieved, using an iterative algorithm with $N+1$ iterations where N is the number of variables in the system (in our case, the number of idb variables appearing in EQS). The 0-th iteration sets all variables to 0 (in our case $false$) with constants unchanged, and then in the j-th iteration we set the value of variables based on the equations and the values of the variables in iteration $j-1$.

[6]The active domain contains values of all the attributes that actually appear in the database instance

Input: A datalog Program P; a PosBool(X)-annotated database D
Output: Circuit as annotation for each idb facts in $P(D)$

1 $EQS := GenerateEquations(P, D)$;
2 */*Create circuit*/* ;
3 Let N be the number of idb variables in EQS. For each idb variable X_t, create a Boolean $false$ sink node associated with $X_{t,0}$. ;
4 For every edb variable X_t, create a sink node labeled with X_t. ;
5 **for** $j = 1$ *to* $N+1$ **do**
6 | $AddCircuitLayer(j, j-1)$;
7 **end**
8 **return** *For every idb fact* t *in* $P(D)$, *annotate* t *with the circuit rooted at node* $X_{t,N+1}$

Algorithm 1: Generate PosBool(X) Circuit

There is a easy translation of the above procedural approach to a compact circuit. In Line 3 and 4 we create two types of sinks of the circuits: (i) $X_{t,0} = false$ nodes for a candidate idb tuple t, which corresponds to the $false$ assignments of the variables in iteration 0; and (ii) sink nodes X_t for edb tuples t.

Procedure AddCircuitLayer. Then we add $N+1$ layers to the circuit by the $AddCircuitLayer(j, j-1)$ procedure which simulates assignments in the j-th iteration using assignments in the $j-1$-th iteration for iteratively solving the system of equations. In the j-th iteration, we create a $\vee$-node associated with $X_{t,j}$, where t is a candidate idb tuple in EQS. If the equation for t in EQS has k terms, we add k $\wedge$-nodes as children of this $\vee$-node (called $\wedge$-children). If $X_{t_1} \cdots X_{t_\ell}$ is one of those k terms, the corresponding $\wedge$-node has ℓ children. The i-th child points to $X_{t_i,j-1}$ if t_i is a candidate idb tuple, and to the sink X_{t_i} if t_i is an edb tuple.

Output of the algorithm. The source nodes in the DAG at the top-most level $X_{t,N+1}$ correspond to the final circuit for X_t (equivalent to the final solution of the system of equations after $N+1$ iterations), which we return for each actual idb fact t in $P(D)$.

Example 6. *In our running example, the number of variables in* EQS *is* $N = 5$ *and so the obtained circuit will have* 6 *"layers". We show the connections in the first two layers in Figure 2.*

It is easy to see that Algorithm 1 runs in polynomial time and outputs circuits with size polynomial in the size of the database D (for a fixed datalog program P); a detailed proof will appear in the full version of the paper.

We further note that equivalent datalog programs generate equivalent provenance.

Proposition 1. *Two datalog programs* P_1 *and* P_2 *are set-equivalent if and only if for every* PosBool(X)*-annotated*

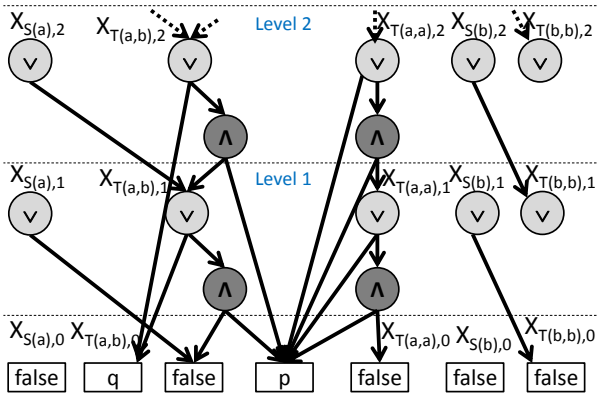


Figure 2: The first two layers of a circuit generated by Algorithm 1 (see Example 6)

database D it holds that $P_1(D) \equiv P_2(D)$.

The proof of the above proposition uses the faithfulness of representation: $P_1(D) \equiv P_2(D)$ holds if and only if there is equivalence of the positive Boolean formulas annotating results. To observe that this is correct, note that equivalence of the Boolean formulas holds if and only if they give the same truth value for every valuation, i.e. the result is equivalent under any truth assignment to PosBool(X). By the faithful representation property, this holds if and only if the queries are set-equivalent.

3.2 Applications

We next exemplify applications of provenance represented as Boolean circuits in deletion propagation and probability computation in probabilistic databases.

Deletion propagation. Let D' be a standard database (*i.e.*, not necessarily a PosBool(X)-database), τ is a subset of tuples in D', and P be a datalog program. Given P, D', and τ, the deletion propagation problem is to evaluate P on the database D'' obtained by deleting tuples in τ from D'.

PROPOSITION 2. *If we construct a* PosBool(X)*-database D from D' by annotating tuples in D' with distinct variables in X, and compute the (shared) Boolean circuit for the idb tuples in $P(D)$, then deletion propagation can be implemented in time linear in the size of the circuit through a bottom-up evaluation.*

EXAMPLE 7. *Given the circuit in Figure 2, suppose we want to delete (only) the tuple $R(a, b)$. As explained in [27] this corresponds to assigning q to $false$ and assigning p to $true$. Now we propagate the assignments of $true/false$ from the sinks ($p = true$, $q = false$ and sinks for idb variables set to $false$) up to the sources. An idb tuple t exists after deleting $R(a, b)$ if and only if it is evaluated to $true$.*

Probabilistic databases. PosBool(X)-relations can be extended to capture probabilistic databases [36] in a natural way, by associating independent probabilities [7] with elements of X. This in turn defines a probability distribution on possible worlds (the probability of a world is the sum of probabilities of truth assignments yielding it); see [36]. Query evaluation amounts to computing the probability of observing an idb tuple t in evaluation of a query on a possible world. The complexity of exact query evaluation is known to be hard even for very restricted cases, but we next show how the constructed circuits can be used as a tool for approximation for datalog. Such approximation has been considered in different contexts [16, 1], but the assumption in these work is that the input database is available. In practice, in many common scenarios one may only have the query result, and cannot access the database itself. For instance when it defines a view over the original database and a full database access is impossible due to access restrictions, lack of storage space, etc. [15]. We show here that *given only the provenance circuit*, one may still obtain an (absolute) approximation of the probability of result tuples. If the correct probability of a tuple is p then a randomized absolute approximation algorithm A computes given ϵ, δ, a probability p' such that $Pr(\mid p' - p \mid < \epsilon) > 1 - \delta$. Such randomized absolute approximation algorithm for datalog on probabilistic databases was presented in [16] *when the database is available.* In the full version of the paper, we show that such approximation is possible in the presence of only *query result including provenance information.* Note that a poly-time *relative approximation* is inacheivable (unless $P = BPP$) for datalog even in presence of the input database [16].

PROPOSITION 3. *Let D be a PosBool(X)-database, P a datalog program, $\Pi : X \mapsto [0, 1]$ a probability distribution on X, and t an idb tuple in $P(D)$. Given* only *the circuit representing the provenance of t and probabilities of the edb tuples (sink-nodes of the circuit) according to Π, one can compute in time polynomial in the size of the circuit a* randomized absolute approximation *of the probability of t to appear in $P(D')$, where D' is a possible world of D distributed according to Π.*

4. EFFICIENT CIRCUIT GENERATION FOR BOOLEAN PROVENANCE

Algorithm 1 in the previous section that generates provenance in circuit form has two shortcomings: (1) it may not be efficient, since it creates and iteratively considers all possible candidate idb tuples that are not necessarily idb facts, (2) unlike previously proposed approaches for propagating provenance through query evaluation (like positive relational algebra for K-relations), Algorithm 1 is somewhat detached from datalog evaluation. Namely, the computation of provenance of idb facts is not closely coupled with the computation of the facts. In this section we present a more efficient algorithm (Algorithm 2) that directly generates a circuit-based provenance while evaluating the datalog program by semi-naive evaluation and considers a candidate idb tuple if and only if it is an idb fact. In addition, we describe some other optimization techniques to improve both the time complexity to generate the circuit and space complexity to store the circuit.

[7] Since arbitrary formulas over X can be used to annotate facts, this will not necessarily entail independence between tuple probabilities.

Input: A Datalog Program P; a PosBool(X)-annotated database D
Output: Two levels (top and bottom) of the PosBool(X)-circuits for all idb facts w.r.t. P and D

```
1  For each edb fact t, create a sink-node labeled with the edb variable X_t. ;
2  while semi-naive evaluation continues do
3  |  Let i be the current iteration of semi-naive evaluation ;
4  |  while semi-naive evaluation considers generating an idb fact t in iteration i using existing edb and idb facts
   |     t_1, ..., t_q such that at least one of t_1, ..., t_q has been generated in iteration i − 1 do
5  |  |  if t is a new idb fact then
6  |  |  |  Create a new ∨-node for X_{t,up} at the top level, and a sink node for X_{t,bottom} at the bottom.
7  |  |  end
8  |  |  if the derivation is a new derivation then
9  |  |  |  Add a ∧-child to the ∨-node annotated by X_{t,up} ;
10 |  |  |  for every t' ∈ {t_1, ..., t_q} do
11 |  |  |  |  If t' is an edb (resp. idb) fact, connect the ∧-child to sink node for X_{t'} (resp. X_{t',down}) ;
12 |  |  |  end
13 |  |  end
14 |  end
15 end
```

Algorithm 2: Provenance circuits by semi-naive evaluation

4.1 Improving Space Complexity of Circuits

One of the major limitations of Algorithm 1 is the $\Omega(N^2)$ space complexity to store the circuit, where N is the number of candidate idb tuples (the circuit stores at least N $\vee$-nodes for the candidate idb tuples in $N+1$ layers) However, the connections from nodes in level ℓ to level $\ell-1$ are identical to the connections from level $\ell+1$ to level ℓ (see Figure 2). Therefore, we do not need to explicitly store all the $N+1$ levels; instead, it suffices to store only one set of connections between two levels called the *top* and *bottom* levels. For instance, in Figure 2, it would suffice to store nodes in level 1 ($X_{t,1}$), the sinks ($X_{t,0}$ and p, q), and the connection between them. Copy of the (candidate) idb tuple t at the top and bottom levels will be denoted by $X_{t,top}$ and $X_{t,bottom}$ respectively, whereas the edb facts t as sinks will be denoted by X_t. Initially, all variables $X_{t,bottom}$ are set to false. Then the assignments of the $X_{t,top}$ variables are evaluated for $N+1$ steps, setting the assignments of $X_{t,up}$ in step i to be the assignments of $X_{t,bottom}$ in step $i+1$. This improves the space complexity by a factor of $\Omega(N)$ and also improves the circuit generation time (proof will appear in the full version of the paper). We will output the circuit in this compact representation in Algorithm 2 presented below.

4.2 Direct Circuit Generation By Semi-Naive Evaluation

In any iteration i of *semi-naive* Datalog evaluation [2], edb facts and existing idb facts are used to generate new idb facts such that at least one idb fact used in the derivation is generated in the $i-1$-th iteration. If the standard semi-naive evaluation finds that an idb fact t has already been generated, it will simply ignore t if it is regenerated again later. Algorithm 2 presented in this section will run semi-naive evaluation, but in addition to storing new idb facts, it will store new derivations of (both new and old) idb facts.

Description of Algorithm 2. The circuit is stored in the compact representation described in Section 4.1. Line 1 creates sink-nodes annotated by X_t for edb facts t in D (denoted by p, q in Figure 2 or 3). If a new idb fact t is generated, two nodes $X_{t,top}$ and $X_{t,bottom}$ are created (Lines 5-7). But whenever a new derivation is found (for new or existing fact t), it is added (Lines 9-12) as a new $\wedge$-child of $X_{t,up}$.

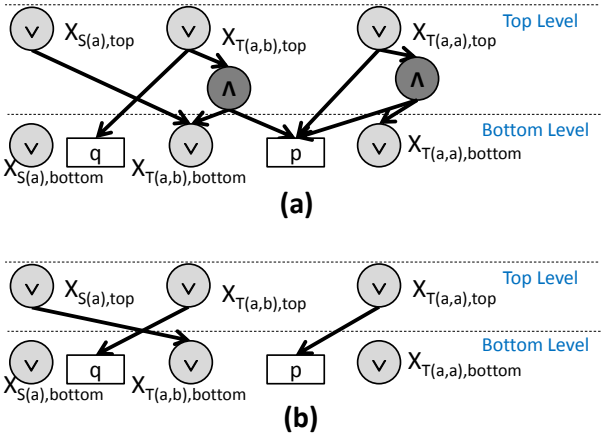


Figure 3: Improved circuits (a) by Algorithm 2, (b) after removing self-dependency

Example 8. *Both $X_{S(b)}, X_{T(b,b)}$ are not idb facts in our running example, and the nodes (and edges) corresponding to these candidate idb tuples would not appear in the circuit (see Figure 3 (a)), and the time to evaluate the final assignment will also be smaller. The improvement can be much more significant for larger programs and inputs.*

Algorithm 2 gives improvements over Algorithm 1 by considering the actual idb facts which can be much fewer than candidate idb tuples, and by storing a single layer instead of all the layers explicitly (apart from the fact that now the circuit generation is embedded within semi-naive evaluation): The space and time complexity analysis of Algorithm 2 will appear in the full version of the paper.

4.3 Optimization: Removing Self-Dependency

An optimization for simplifying the circuit for PosBool(X) is to ignore self-dependency of variables. If a minterm (conjunction of edb or idb variables) in the equation for idb variable X_t contains X_t itself, then that minterm can be safely

ignored while keeping the solution of the system of equations unchanged:

EXAMPLE 9. *Removing self-dependency in Figure 3 (a) leads to further simplified circuit in Figure 3 (b).*

The correctness of this optimization will be shown in the full version of the paper. More optimizations may be employed, such as no further evaluation of a $\vee$-node (resp. $\wedge$-node) if one child evaluates to true (resp. false). Another optimization based on *tuple-dependency graphs*, similar to *precedence graphs* used for improving the efficiency of the basic seminaive algorithm [2], will be discussed in the full version of the paper.

5. BEYOND BOOLEAN ANNOTATIONS

So far we have focused on a particular, simple yet important, provenance model, namely the case where provenance is captured by positive Boolean expressions. The work on provenance semirings [27] goes far beyond Boolean annotations, to arbitrary commutative semirings. Various provenance semirings have been proven useful in a variety of applications, including access control and computing cost associated with tuples. We first review the basic definitions, then we consider datalog evaluation on annotated databases beyond the Boolean case, charting the extent to which the results extend.

5.1 Semirings and K-relations

We briefly recall semirings and their use in provenance, and refer the reader to [27, 28] for further details. An algebraic structure $(K, +_K, \cdot_K, 0_K, 1_K)$ is a commutative semiring if each of $(K, +_K, 0_K)$ and $(K, \cdot_K, 1_K)$ is a commutative monoid (*i.e.*, both $+_K$ and $\cdot_K$ are associative and commutative, and $0_K, 1_K$ are their respective neutral elements), $\cdot_K$ is distributive over $+_K$, and $a \cdot_K 0_K = 0 \cdot_K a = 0_K$. A *semiring homomorphism* is a mapping $h : K \to K'$ where K, K' are semirings, and $h(0_K) = 0_{K'}, h(1_K) = 1_{K'}$, $h(a +_K b) = h(a) +_{K'} h(b)$, $h(a \cdot_K b) = h(a) \cdot_{K'} h(b)$. We will omit the subscripts from $+_K, \cdot_K$ when they are clear from the context.

The two operations of a semiring capture abstractly information usage: $\cdot$ corresponds to *joint* usage while $+$ corresponds to *alternative* usage. Annotating a tuple with 0_K signifies that the tuple is *absent*. Given a tuple t (in some relation) we use $\mathtt{Ann}(t)$ to denote the annotation of t. A K-relation is then a finite relation whose tuples are all K-*annotated*, *i.e.*, annotated with elements of a semiring K. A K-database is a schema-indexed set of K-relations; these generalize the concepts introduced in 2 where $K = \text{PosBool}(X)$.

Semiring annotation propagation algorithms have been devised for various query languages [27, 20, 28, 4, 21]. Given an input K-database and a query, these algorithms annotate the result of the query by producing an output K-relation. Therefore, these propagation algorithms define a semantics (we will call it K-*semantics*) for various query languages over the K-relation data model.

This framework has a good deal of generality. Two basic semirings, the Boolean semiring $(\mathbb{B}, \vee, \wedge, \text{false}, \text{true})$ and the natural numbers semiring $(\mathbb{N}, +, \cdot, 0, 1)$ give us $\mathbb{B}$-relations as the usual set semantics relations and, respectively, $\mathbb{N}$-relations as the bag semantics relations (here the annotation of a tuple t is its multiplicity, *i.e.*, the number of times t occurs). We have also already seen the commutative semiring $(\text{PosBool}(X), \vee, \wedge, \text{false}, \text{true})$. This semiring, as well as, *e.g.*, $\mathbb{N}[X]$, $\text{Why}(X)$, $\text{Lin}(X)$ and $\text{Trio}(X)$ (all discussed below) are examples of what we call *provenance semirings*, captured by the generic notation $\text{Prov}(X)$. The elements of such a $\text{Prov}(X)$ are expressions built using $+$ and $\cdot$ from *variables* in a set X. The variables in X are used to annotate the tuples of an input database (hence a finite X suffices). Therefore the $\text{Prov}(X)$-semantics for a query on such a $\text{Prov}(X)$-database produces an output $\text{Prov}(X)$-relation, and provenance of a tuple t in the output is the $\text{Prov}(X)$-annotation $\mathtt{Ann}(t)$ of the tuple.

$\mathbb{N}[X]$ (introduced in [27]) is the commutative semiring of multivariate *polynomials* in variables from X with coefficients from $\mathbb{N}$. It occupies a special position since it is the commutative semiring freely generated by X, that is, for any commutative semiring K, any function $X \to K$ extends uniquely to a homomorphism $\mathbb{N}[X] \to K$. The elements of $(\text{Why}(X), \cup, \Cup, \emptyset, \{\emptyset\})$ semiring are finite sets of finite subsets of X where $x \in X$ corresponds to $\{\{x\}\}$; $\cup$ is the standard set union, whereas for $W, Z \in \text{Why}(X)$, $W \Cup Z = \{A \cup B | A \in W, B \in Z\}$. $\text{Why}(X)$ captures the (witness) why-provenance of [12]. The minimal witness provenance of [12] is captured by $\text{PosBool}(X)$ discussed in previous sections (see [28]). Due to space restrictions, we refer to [28] for the definitions of the provenance semiring $\text{Lin}(X)$ (which captures the "contributing tuples" model from [14]) and $\text{Trio}(X)$ (which captures the model in [9]).

The papers [27, 26] also introduced a technique for using provenance in applications such as deletion propagation and trust assessment. The semiring framework supports this, as long as applications are captured by annotating tuples with elements of a commutative semiring K such that the following property holds:

Provenance specialization We say that $\text{Prov}(X)$ *specializes correctly* to K, if any valuation $v : X \to K$ extends uniquely to a homomorphism $h_v : \text{Prov}(X) \to K$.

For example, $\mathbb{N}[X]$ specializes correctly to any commutative semiring while $\text{PosBool}(X)$ specializes correctly only to distributive lattices. If a semiring K is used to specialize provenance we will refer to it as a *meta-domain*. Now, given a query and an input database, suppose that we have annotated the input tuples with distinct variables in X, and we have computed the $\text{Prov}(X)$-semantics so that each output tuple has a $\text{Prov}(X)$-annotation. Suppose that subsequently we wish to do an application that requires the computation of a K-semantics. The K-annotation of the input database corresponds to a valuation $v : X \to K$. If $\text{Prov}(X)$ specializes correctly to K, we can use the resulting homomorphism $h_v : \text{Prov}(X) \to K$ by applying it to the $\text{Prov}(X)$-annotations of the output tuples. We obtain the K-annotations of the K-semantics provided the following property holds:

Commutation with homomorphisms We say that a query language and its semiring semantics satisfies *commutation with homomorphisms* if for any query Q in the language, for any commutative semirings K_1, K_2, and any homomorphism $h : K_1 \to K_2$, we have $h(Q(D)) = Q(h(D))$ for any K_1-database D.

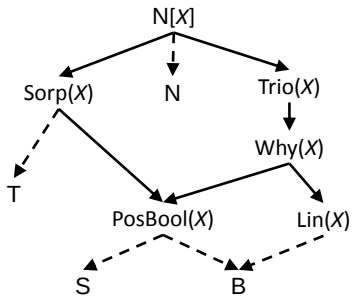


Figure 4: Provenance semiring hierarchy and specialization

This non-trivial property was proved for quite a few query languages and semiring semantics [27, 20, 4].

We already mentioned two meta-domain semirings: $\mathbb{B}$ can be used in deletion propagation and $\mathbb{N}$ can be used in multiplicity maintenance. Then we have the *tropical semiring* $\mathbb{T} = (\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$ which can be used for capturing *cost*; multiplication (joint use) corresponds to adding the cost of tuples used, while addition (alternative use) corresponds to taking the smallest cost among the alternatives. Cost can also be used to assess trust when interpreted as "cost of trusting". Finally we have semirings for capturing access control/security clearances; a simple one (introduced in [20] and refined in [4]) is $(\mathbb{S}, \min, \max, \text{absent}, \text{public})$ where $\mathbb{S} = \{\text{public} < \text{confidential} < \text{secret} < \text{topsecret} < \text{absent}\}$.

The provenance semirings and their ability to support provenance specialization is captured by the diagram in Figure 4 based on the hierarchy described in [28] (Sorp(X) will be described in Section 5.3). A solid arrow from $\text{Prov}_1(X)$ to $\text{Prov}_2(X)$ indicates that the first one is *more informative* than the second (formally, there exists a surjective homomorphism that is the identity on X). A dotted arrow from a Prov(X) to K indicates that Prov(X) specializes correctly to K. Clearly if $\text{Prov}_1(X)$ is more informative than $\text{Prov}_2(X)$ and $\text{Prov}_2(X)$ specializes correctly to K, then so does $\text{Prov}_1(X)$. Note also that we can use the provenance semirings themselves as meta-domains and that any provenance semiring specializes correctly to a less informative one (it can be useful, *e.g.*, to specialize $\mathbb{N}[X]$ or Why(X) to Lin(X) and thus compute the set of contributing tuples).

5.2 Extended Datalog Semantics

It is natural to similarly extend the datalog semantics to K-relations (Section 2.3), to obtain (see [27]):

$$\texttt{Ann}(t) \;=\; \sum_{\tau:\ \tau \text{ yields } t} \;\; \prod_{\substack{t':\ t' \text{ is a leaf} \\ \text{edb fact of } \tau}} \texttt{Ann}(t')$$

where $\sum$ and $\prod$ respectively denote the $+$ and $\cdot$ operations on the semiring K. However, this definition involves possibly infinite sums, and unlike PosBool(X), these sums may not be well-defined in arbitrary commutative semirings. An important subclass in which such infinite sums are well-defined is that of ω-continuous commutative semirings, defined as follows.

Definition 3. *(ω-**continuous**) Given a semiring K, define the binary relation $\sqsubseteq$ such that $a \sqsubseteq b$ if and only if $\exists c \in K$ such that $a + c = b$. We say that K is ω-continuous if (1) $\sqsubseteq$ is a partial order, (2) every (infinite) ω-chain $a_0 \sqsubseteq a_1 \ldots$ has a least upper bound $\sup a_i$, and (3) $\forall a \quad a + \sup a_i = \sup(a + a_i)$ and $a \cdot \sup a_i = \sup(a \cdot a_i)$.*

A homomorphism h between ω-continuous semirings is said to be ω-continuous if it preserves least upper bounds of ω-chains: $h(\sup a_i) = \sup h(a_i)$.

Going back to the provenance semirings of Section 5.1, we note that *for finite* X, PosBool(X), Why(X), and Lin(X) are ω-continuous because they are also finite, while $\mathbb{N}[X]$ and Trio(X) are not, even for finite X. Among the meta-domains, $\mathbb{B}$, $\mathbb{S}$, and $\mathbb{T}$ are ω-continuous, while $\mathbb{N}$ is not. $\mathbb{N}$ is easy to embed in an ω-continuous semiring by adding ∞, $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ while $\mathbb{N}[X]$ can be embedded in $\mathbb{N}^\infty[[X]]$, the semiring of formal power series (infinite polynomials) with variables from X and coefficients from $\mathbb{N}^\infty$.

As shown in [27], the discussion in Section 5.1 is also valid for datalog queries, provided we consider only ω-continuous semirings and ω-continuous homomorphisms. Furthermore, $\mathbb{N}^\infty[[X]]$ is the free ω-continuous commutative semiring generated by X (all valuations extend uniquely to ω-continuous homomorphisms), just as $\mathbb{N}[X]$ played this role for all commutative semirings.

Naturally, we ask how appropriate is $\mathbb{N}^\infty[[X]]$ as *the* provenance semiring for datalog. As we have observed, it correctly specializes provenance to the meta-domains $\mathbb{B}$ (this is the generalization of the *faithfulness* requirement that we stated for PosBool(X)-semantics), and also $\mathbb{S}$, $\mathbb{T}$, and $\mathbb{N}^\infty$. Poly-size overhead however does not even make sense since the elements of $\mathbb{N}^\infty[[X]]$ are infinite formal sums [8].

Still, one might expect that we would be able to compute for each output tuple a finite provenance expression (*i.e.*, an expression built from variables using $\cdot_K$ and $+_K$) that is "general" enough to at least specialize correctly in semirings where all elements are finite (such as Why(X)). If that were so, then, as in the case of PosBool(X), we could try to find poly-size circuits for such expressions. Interestingly, this is not the case

Theorem 3. *It is not possible to annotate with finite provenance expressions the output of datalog programs such that they specialize to the semiring* Why(X) *with the same results as the (correct) specialization of the provenance given by* $\mathbb{N}^\infty[[X]]$.

Intuitively, this means that one can not construct provenance circuits for Datalog in a "general" semiring, so that correct results may be "read" in Why(X). (We will later show what can still be done for Why(X). See Theorem 5.) So instead, we will look for a new provenance semiring, that (1) specializes correctly to many "interesting" semirings (but not to Why(X)), and (2) admit poly-size circuit representation for datalog provenance, that yields correct results for the semirings it specializes to. The new provenance semiring that we will construct is called Sorp(X), and as can be observed in Fig. 4 it specializes correctly to "interesting" semirings such as $\mathbb{T}$, $\mathbb{B}$ and $\mathbb{S}$ (actually to any distributive lattice). We will show in Section 5.3 that Sorp(X) admits poly-size circuit representations with the desired properties.

Intuitively, in Sorp(X) we will keep track of how many times a tuple was used (i.e. every derivation is represented

[8] We will discuss other possible representations in the related work section

as a bag of the tuples it have used) but we do not keep track of "redundant" derivations; these are derivations whose bag of tuples is bag-included in a different derivation. This is similar to the passage from witness (why) provenance to minimal witness provenance [12]. In a sense, only the "most economical" derivations are kept.

EXAMPLE 10. *Consider an idb fact with two derivations, one that uses an edb fact p twice and an edb fact q three times, and one that uses both p and q once. The corresponding provenance expression is* $p^2q^3 + pq$ *The bag of facts used in the second derivation is bag-included in that used in the first. Thus the first will not be recorded in* $\mathrm{Sorp}(X)$ *(i.e., the above provenance expression will be equivalent to pq.*

Towards a definition of $\mathrm{Sorp}(X)$, we say that a commutative semiring $(K, +, \cdot, 0, 1)$ is *absorptive* if the following identity holds: $a+a\cdot b = a$. Notice that an absorptive semiring is also *idempotent*, i.e., + operation is idempotent (take $b = 1$). Now, consider the semiring of polynomials $\mathbb{N}[X]$ and let $\sim$ be the smallest congruence relation on $\mathbb{N}[X]$ that identifies polynomials according to absorption. We define

$$\mathrm{Sorp}(X) = \mathbb{N}[X]/\sim$$

i.e., semiring of congruence classes modulo $\sim$. Based on the universality property of $\mathbb{N}[X]$, it follows that $\mathrm{Sorp}(X)$ is the free commutative absorptive semiring generated by X (i.e. the "most general" such semiring). Because $\mathrm{PosBool}(X)$ and $\mathbb{T}$ are absorptive, it follows that $\mathrm{Sorp}(X)$ is more informative than $\mathrm{PosBool}(X)$ and specializes correctly to $\mathbb{T}$.

The congruence relation leads to a quite natural characterization of the elements of $\mathrm{Sorp}(X)$ as follows. By idempotence every monomial with a coefficient > 1 is congruent with a monomial with coefficient 1. Therefore, in what follows, by "monomial" we mean just a product of powers of variables, at least one of which has exponent > 0. We say that the monomial m_1 *absorbs* monomial m_2 if $m_2 = m_1 \cdot m$ for some other monomial m (since then $m_1 + m_2 \sim m_1$). Clearly, this is a strict partial order on monomials. We say that a polynomial is an *antichain* if it is a sum of monomials such that none of them absorbs another.

EXAMPLE 11. *The polynomial* $pq^2 + p^3q$ *is an antichain since neither of the monomials absorbs the other. Note that it cannot be further simplified using the congruence axioms. In contrast,* p^2q^3+pq *is not an antichain, and indeed we have* $p^2q^3 + pq = pq \cdot (pq^2 + 1) = pq \cdot (1 + 1 \cdot pq^2) \sim pq \cdot 1 = pq$.

In general we have:

PROPOSITION 4. *Let* $\gamma \in \mathrm{Sorp}(X)$ *be a* $\sim$*-congruence class of polynomials from* $\mathbb{N}[X]$*. Exactly one of the following holds:*

(i) All the polynomials in γ *are* $\sim$*-congruent to 0.*

(ii) All the polynomials in γ *are* $\sim$*-congruent to 1.*

(iii) All the polynomials in γ *are* $\sim$*-congruent to exactly one antichain polynomial.*

Therefore, we have the following nice representation for the elements of $\mathrm{Sorp}(X)$: they are the antichain polynomials together with 0 and 1. Finally we have the following essential property:

PROPOSITION 5. *If X is finite, then* $\mathrm{Sorp}(X)$ *is* ω*-continuous and the* $\mathrm{Sorp}(X)$*-semantics for datalog specializes correctly to any commutative absorptive* ω*-continuous semiring.*

The proof is omitted, but the basic idea is that an ascending ω-chain of antichain polynomials must become "stationary" after finitely many steps otherwise we get an infinite strictly descending sequence of natural numbers (that appear as exponents in the monomials).

5.3 Extended Circuit-Based Semantics

We next extend the circuit-based construction to semirings beyond $\mathrm{PosBool}(X)$, starting with $\mathrm{Sorp}(X)$. For that, we first note that the notion of Boolean circuit can be generalized to circuits over an arbitrary semiring $(K, +, \cdot, 0, 1)$ in a straightforward manner. Leaves are associated with basic elements of K, and internal nodes are $+_K$-node or $\cdot_K$-node (generalizing $\vee$- and $\wedge$-nodes for $\mathrm{PosBool}(X)$); we call these *K-circuits*. A top-down reading of the circuit now yields a semiring element as an expression involving the + and $\cdot$ operations.

In addition to being quite natural and general, the semiring $\mathrm{Sorp}(X)$ can indeed serve as a provenance semiring in our context. Intuitively, the following result means that we can compute provenance circuits for datalog in $\mathrm{Sorp}(X)$, and then specialize it correctly in *every absorptive semiring*, including every semiring that is "below" $\mathrm{Sorp}(X)$ in Figure 4.

In the construction we will use the *absorption* property, favorable for circuit generation.

THEOREM 4. *We can compute a circuit-based representation with poly-size overhead for the* $\mathrm{Sorp}(X)$*-semantics for datalog.*

PROOF. (sketch) The algorithm to produce a K-circuit follows exactly the structure of Algorithm 1, replacing every occurrence of $\vee$ with $+_K$ and of $\wedge$ with $\cdot_K$. The algorithm itself guarantees a result with polynomial-size overhead. The fact that the result of the algorithm computes indeed the $\mathrm{Sorp}(X)$-semantics for datalog follows from the fact that $\mathrm{Sorp}(X)$ is absorptive and therefore satisfies the conditions in [18]. □

Optimizations. The optimizations employed in Section 4 to embed computation of provenance as part of the seminaive evaluation, go through to the settings of $\mathrm{Sorp}(X)$-provenance, and we can also employ optimizations that utilize the congruence axioms that we have enforced, to further simplify the circuit.

EXAMPLE 12. *Reconsider the circuit in Fig. 2, and observe that by replacing* $\wedge$ *by* $\cdot$ *and* $\vee$ *by* $+$*, we obtain a provenance circuit in* $\mathrm{Sorp}(X)$*. Observe that according to the equivalence axioms in* $\mathrm{Sorp}(X)$*, the circuit rooted at* $X_{S(a),2}$ *can be simplified to once including a single node labeled q. This corresponds to the intuition that the "simplest" derivation of S(a) involves only a single use of the tuple* $T(a, b)$*. Now, consider a homomorphism to the tropical semiring. Intuitively, such homomorphism corresponds to associating numeric values (costs) with base facts; then the provenance of derived facts is the minimal cost of a derivation. For our*

example, the cost for S(a) will be the total cost of the shortest path *from a to b. In our simple example, this is exactly the cost c assigned to q (i.e. the cost of the edge* (a, b)*).*

Further, the self-dependency can be ignored for some semirings like distributive lattices and tropical semirings. To conclude this section, we revisit Figure 4, and briefly consider Datalog provenance semirings that do not appear "below" $\mathrm{Sorp}(X)$ in the semiring hierarchy. We first revisit the non-absorptive semiring $Why(X)$.

Beyond absorptive semirings. We note that the $Why(X)$-semiring is non-absorptive and so does not fit the framework. Further observe that the proof of Theorem 3 indicates that this can not be remedied by choosing a different provenance semiring. We may, however, compute provenance *directly* for $Why(X)$, through a dedicated construction.

THEOREM 5. *There is a* $\mathrm{Why}(X)$*-semantics for datalog that satisfies poly-size overhead, faithfulness of representation, and commutation with homomorphisms.*

PROOF SKETCH. Our algorithm for constructing the $Why(X)$ circuit follows lines similar to that of Algorithm 1, except that the fixpoint computation (and hence the layering in the construction of the circuit) is repeated here $((|E| \times |W| + 1)) \times |E|) + 1$ times, where E is the set of the equations generated in Algorithm 1 and W is the size of the $Why(X)$-database provenance. (Recall that, in contrast, for $\mathrm{Sorp}(X)$, the iterations depend only on the number of variables; the size of the provenance need not be accounted for).

To prove the correctness of this construction we consider the set of equations E constructed in Algorithm 1. It has been shown in [18] that one can construct from E a corresponding context free grammar G whose derivation trees capture the sequence of fix-point iterations of E, in the sense that, when viewed as semiring expressions, their sum evaluates to the same value as the corresponding fixpoint computation. We show that, when considering this sum, it suffices to look at derivation trees of of depth at most $((|E| \times |W| + 1)) \times |W|) + 1$. And hence one needs to repeat the fixpoint computation for E at most that number of times. The proof is based on the observation that deeper trees can be pruned to form a smaller tree that evaluates to the same provenance value, and thus do not contribute to the sum (In $Why(X)$ only distinct elements may add value). □

We have now considered all semirings in Figure 4: we have shown that there is no "good" circuit-based semantics for $\mathbb{N}^\infty[[X]]$. This motivated the introduction of a novel semiring $\mathrm{Sorp}(X)$ and we have shown that the circuit construction for $\mathrm{PosBool}(X)$ extends naturally to $\mathrm{Sorp}(X)$ and all "less general" semirings. Finally we have noted that $\mathrm{Why}(X)$ is not "less general" than $\mathrm{Sorp}(X)$ and have proposed a dedicated construction for it. Throughout the section we have assumed that the semiring of interest is ω-continuous. To conclude we briefly consider semirings beyond this class.

Beyond ω-continuous semirings. For commutative semirings that are not ω-continuous (notable examples are $\mathbb{N}[X]$ and $\mathrm{Trio}(X)$), provenance is in general not even well-defined. Still, for tuples with finitely many derivations, the provenance expression is in fact a finite sum (which may nevertheless be of super-polynomial size, if computed directly as a sum). To obtain provenance semantics that satisfies the desiderata when restricted to tuples with finitely many derivations, we may apply the following procedure: (1) Detect which tuples have infinitely many derivations, (2) Generate circuits for tuples with finitely many derivations. Step (1) can be done using the algorithm of [33]. As for step (2), observe that for tuples with finitely many derivations we may obtain a non-recursive equation system. From that we can obtain a poly-size circuit, using our general algorithm. Note that ω-continuity is required in the construction only to handle recursion, and so is not required for finitely many derivations.

6. RELATED WORK AND CONCLUSIONS

We have proposed in this paper to base the foundations of provenance for Datalog on the notion of circuits. We have shown that circuits are effective in capturing provenance even in cases where the traditional use of provenance polynomials incurs super-polynomial provenance size. We have presented algorithms and optimization techniques for propagating provenance based on the approach, and we have shown its usefulness of the approach in different application domains such as probabilistic databases and access-control.

As mentioned earlier, provenance has been studied in various lines of work, using different formalisms (see e.g. [19, 11, 37, 14, 12, 9, 10]). For capturing provenance of Datalog queries, three other approaches have been proposed. We have already mentioned the representation of provenance as an infinite sum [27] and its shortcomings. An alternative [27, 24] is to use a system of equations (such as the one generated as an intermediate step in our circuit generation algorithm) as the final provenance representation. Our approach has two advantages with respect to that one: (1) circuits, and in particular *circuit complexity* in terms of circuit size and its use in the context of parallel evaluation have been extensively studied for decades as a fundamental problem in complexity theory (*e.g.* see [5, 8]). Basing our provenance model on circuits allows us to benefit from these foundations; (2) when using (for applications such as deletion propagation) the equation system as the provenance representation, one needs to iteratively "solve" the system. Some computation done in these iterations is in fact done as part of provenance circuit generation, i.e. "offline". Similar accounts apply to the comparison with the possible use of *provenance graphs* [30]. Finally, we note that while our work is the first to consider circuits in the context of datalog provenance, particular kinds of circuits have been previously used to trace provenance in different contexts such as data mining tasks [34, 23].

We have performed some preliminary experiments that indicate the superiority of using circuits for deletion propagation, with respect to the alternative solutions of iteratively solving the system of equations, or simply re-evaluating the datalog program. A full-scale experimental study is left as future work, as well as further circuit-specific optimizations.

Last, we mention that Boolean c-tables in the context of datalog (in fact, datalog in presence of contradictions) were studied in [1]. However, conditions in c-tables may involve negation; this is unlike our Boolean provenance case. The restriction to positive formulas imposed in our case allows to extend the framework to general provenance semirings. On

the other hand, the use of negation in [1] does not generalize to semirings, but yields a poly-size c-table based on Boolean formulas, for datalog with contradictions.

Acknowledgments

This research was partially supported by the European Research Council under the European Community's 7th Framework Programme (FP7/2007-2013) / ERC grant MoDaS, grant agreement 291071, by the National Science Foundation (NSF IIS 1217798), the Israeli Ministry of Science, the Isreali Science Foundation (ISF), and by the US-Israel Binational Science Foundation (BSF).

7. REFERENCES

[1] S. Abiteboul, D. Deutch, and V. Vianu. Deduction with contradictions in datalog. In *ICDT*, 2014. *To appear.*

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: datalog in time and space. In *Datalog '10*, 2010.

[4] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, 2011.

[5] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[6] B. T. Loo et al. Declarative networking: Language, execution and optimization. In *SIGMOD*, 2006.

[7] R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with lixto. *VLDB Journal*, 2001.

[8] Y. Ben-Asher, E. Fisher, G. Haber, and V. Tartakovsky. Fast evaluation of boolean circuits based on two-players game and optical connectivity circuits. In *ICPP*, 2012.

[9] O. Benjelloun, A.D. Sarma, A.Y. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17, 2008.

[10] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB J.*, 14(4), 2005.

[11] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Syst.*, 33(4), 2008.

[12] P. Buneman, S. Khanna, and W.C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[13] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[14] Y. Cui, J. Widom, and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2), 2000.

[15] D. Deutch, Z. Ives, T. Milo, and V. Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*, 2013.

[16] D. Deutch, C. Koch, and T. Milo. On probabilistic fixpoint and markov chain query languages. In *PODS*, 2010.

[17] Jason Eisner and Nathaniel Wesley Filardo. Dyna: Extending datalog for modern ai. In *Datalog*, 2010.

[18] J. Esparza and M. Luttenberger. Solving fixed-point equations by derivation tree analysis. In *CALCO*, 2011.

[19] Robert Fink, Larisa Han, and Dan Olteanu. Aggregation in probabilistic databases via knowledge compilation. *PVLDB*, 5(5):490–501, 2012.

[20] J.N. Foster, T.J. Green, and V. Tannen. Annotated XML: queries and provenance. In *PODS*, 2008.

[21] F. Geerts, G. Karvounarakis, V. Christophides, and I. Fundulaki. Algebraic structures for capturing the provenance of sparql queries. In *ICDT*, 2013.

[22] F. Geerts and A. Poggi. On database query languages for k-relations. *J. Applied Logic*, 8(2), 2010.

[23] B. Glavic, J. Siddique, P. Andritsos, and R. J. Miller. Provenance for data mining. In *TaPP*, 2013.

[24] Gösta Grahne. *The Problem of Incomplete Information in Relational Databases*. Springer, 1991.

[25] T. J. Green, M. Aref, and G. Karvounarakis. Logicblox, platform and language: A tutorial. In *Datalog*, 2012.

[26] T. J. Green, G. Karvounarakis, Z. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007.

[27] T.J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[28] Todd J. Green. Containment of conjunctive queries on annotated relations. In *ICDT*, pages 296–309, 2009.

[29] T. Imielinski and W. Lipski. Incomplete information in relational databases. *J. ACM*, 31(4), 1984.

[30] Z. Ives, T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. Talukdar, M. Jacob, and F. Pereira. The ORCHESTRA collaborative data sharing system. *SIGMOD Rec.*, 37, 2008.

[31] Mauricio Karchmer and Avi Wigderson. Monotone circuits for connectivity require super-logarithmic depth. In *STOC*, pages 539–550, 1988.

[32] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 4(1), 2010.

[33] Inderpal Singh Mumick and Oded Shmueli. Finiteness properties of database queries. In *Australian Database Conference*, pages 274–288, 1993.

[34] Dan Olteanu and Sebastiaan J. van Schaik. Dagger: clustering correlated uncertain data (to predict asset failure in energy networks). In *KDD*, 2012.

[35] Marc D. Riedel and Jehoshua Bruck. Cyclic boolean circuits. *Discrete Appl. Math.*, 160(13-14), 2012.

[36] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[37] S. Vansummeren and J. Cheney. Recording provenance for sql queries and updates. *IEEE Data Eng. Bull.*, 30(4), 2007.

[38] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *SIGMOD*, 2010.