Towards Elastic Incrementalization for Datalog

David Zhao
University of Sydney
dzha3983@uni.sydney.edu.au

Mukund Raghothaman
University of Southern California
raghotha@usc.edu

Pavle Subotić
Microsoft
pavlesubotic@microsoft.com

Bernhard Scholz
University of Sydney
bernhard.scholz@sydney.edu.au

ABSTRACT

Various incremental evaluation strategies for Datalog have been developed which reuse computations for small input changes. These methods assume that incrementalization is always a better strategy than re-computation. However, in real-world applications such as static program analysis, re-computation can be cheaper than incrementalization for large updates.

This work introduces a novel elastic incremental approach that has two different strategies that can be selectively applied. We call the first strategy a *Bootstrap* strategy that recomputes the entire result for high-impact changes, and the second is an *Update* strategy that performs an incremental update for low-impact changes. Our approach allows for a lightweight Bootstrap strategy that is suitable for high-impact changes, with the trade-off that Update may require more work for small changes. We demonstrate our approach on real world applications and compare our elastic incremental approach to existing methods.

ACM Reference Format:
David Zhao, Mukund Raghothaman, Pavle Subotić, and Bernhard Scholz. 2018. Towards Elastic Incrementalization for Datalog. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Logic languages such as Datalog have seen wide-spread adoption in recent years in areas such as static program analysis [2, 6, 9, 11], declarative networking [4, 17, 42], security analysis [29], business applications [3] and machine learning [24]. The main reasons for the wide-spread adoption have been the availability of high-performance logic engines [3, 14, 19] and the ease of expressing programs declaratively, i.e., computations can be expressed succinctly, providing means for rapid prototyping of scientific and industrial applications.

The standard evaluation strategy for Datalog programs is to find the resulting output when given a set of facts and logical rules. The facts are the input to the logical computation defined by the rules, and the output is the logical result of the computation.

Woodstock '18, June 03–05, 2018, Woodstock, NY

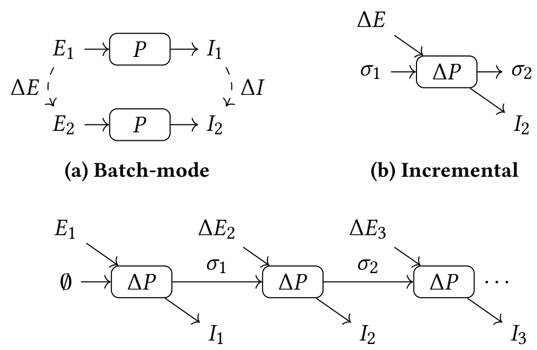
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

However, it has been observed [23, 27] that many real-world applications re-compute most of their results with slight variations of their input. Hence, several state-of-the-art Datalog engines have proposed incremental evaluation techniques [21, 23, 26] to facilitate streaming, i.e., the evaluation uses the previous result with a given change in its input.

Successful applications of incremental Datalog have operated on several assumptions: (1) that the impact, i.e., number of overall tuple changes, is proportional to the update size, and (2) that the use case exhibits a continuous stream of small impact updates. Indeed for several use cases [27] these assumptions tend to hold. However, for other notable use cases such as program analysis in a continuous integration/continuous delivery (CI/CD) setup [7, 8, 38] these assumptions do not hold.

Static analyses written in Datalog can consist of hundreds or thousands of highly recursive rules and relations [6, 10]. Due to the complexity of the ruleset, one can no longer assume that update size is proportional to impact size. For example, in our experimental evaluation on the Doop program analysis framework, we found large variability in the *impact* of updates. This can be explained by the connectivity of points-to analyses, where even small changes in the input may substantially change pointer sets of variables. Moreover, when static analyzers are deployed in CI/CD pipelines there is no guarantee that updates will be structurally small. For instance, when the code base is updated, the initial change is often a refactor or a new feature implementation. Such code changes typically result in large changes to the input of an analysis. These changes may then be followed by smaller changes as a result of minor review suggestions, but as we show, even these smaller input changes cannot ensure small impacts. Thus, we argue the success of incremental evaluation techniques on such use cases requires minimizing the overhead of evaluating large impact updates.

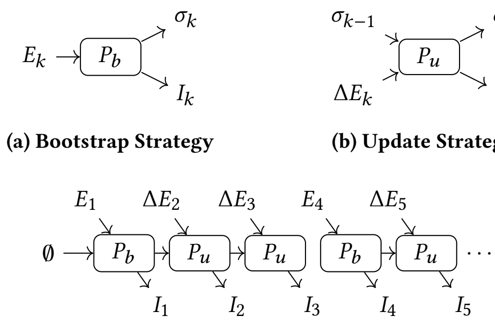
Consider Fig. 1a that illustrates a generic incremental computation setup. Given a Datalog program P and two inputs E_1 and E_2, also known as Extensional Database (EDB). A standard *batch-mode* evaluation runs the program P with E_1 and E_2 independently, to produce the results I_1 and I_2, also known as Intensional Database (IDB). However, assume that only a small portion of the input (denoted by ΔE) and output (denoted by ΔI) changes in E_2 and I_2. In such a scenario, we assume that many computations for producing I_2 are repeated. An incremental evaluation ΔP can recycle computations from a previous evaluation. The functional block of an incremental evaluation [21, 23, 26] is illustrated in Fig. 1b. A *Computational State* σ_1 encodes the previous computations for I_1 in a special format so that the computations can be reused. With state σ_1 and the change in input ΔE, the incremental evaluation



(a) Batch-mode **(b) Incremental**

(c) Stream Processing with Incremental

Figure 1: Batch-mode vs. Incrementalized Evaluation



(a) Bootstrap Strategy **(b) Update Strategy**

(c) Elastic Streaming

Figure 2: Elastic Incremental Evaluation

produces the output I_2 and the new computational state σ_2. The streaming setup of incremental evaluation is shown in Fig. 1c. A series of updates to the EDB is provided over time via ΔE_i. We call one stage in the stream an *epoch*. For the first epoch, we use the empty state as computational state and E_1 as ΔE_1 to produce I_1 and the state σ_1. Any subsequent change ΔE_i in the EDB is processed by using the previous computational state σ_{i-1} to generate I_i.

State-of-the art incremental evaluation frameworks [23, 25, 26] use a comprehensive computational state so that small updates can be performed efficiently. Because of their comprehensive computational state, initiating a stream with current frameworks can be prohibitively slow and cannot be used to react to large updates. For example, when a static program analysis seeks to reuse previous computations for a large code refactoring, significant portions of the control flow graph may have been replaced. In such a use case, an incremental evaluation will essentially perform two computations, one to delete the old control flow graph, and one to compute the new control flow graph with additional overheads caused by the incrementalization. Therefore, these heavyweight updates are better served by an evaluation strategy that is closer to standard batch-mode evaluation augmented with state for the future updates to be performed incrementally.

In this work, we propose an *elastic* incremental evaluation scheme called *Bootstrap-Update*. Our approach has two distinct strategies to evaluate an update: a specialized *Bootstrap* denoted as P_b (Fig. 2a), and *Update* denoted as P_u (Fig. 2b). The specialized Bootstrap resembles an augmented batch-mode evaluation that produces the computational state from scratch to allow subsequent updates, whereas Update is an incremental evaluation strategy.

Our approach proposes a novel *sparse* encoding that maintains a *lightweight* state σ. Our state exhibits a worst-case space complexity of $O(|I|)$ (i.e. linear in the size of the output) whereas existing incremental encodings [23, 26], have a worst-case space complexity of $O(m|I|)$ where m is the number of fixpoint iterations in the semi-naive evaluation algorithm [1]. Our lightweight state allows for an accelerated Bootstrap algorithm that can handle high-impact updates by efficiently recomputing the state from scratch, with the trade-off that the Update strategy may require more work for smaller updates. Furthermore, we provide a simple heuristic for choosing the appropriate strategy: we rerun the bootstrap when the incremental update takes more than a fraction (as a *switching parameter*) of the last bootstrap's runtime. This switching parameter typically depends on the behavior of each individual application, and how large a typical update is for that application. Our solution operates under the insight that if we have comparable performance with batch mode Datalog evaluation on large impact updates and a small slow down on low impact updates we will have an overall net gain by selective application of incremental evaluation.

We have integrated our elastic Bootstrap-Update incremental evaluation in the open source high performance Datalog compiler Soufflé [20]. We have performed an extensive evaluation on a number of use cases which show the utility of our approach when compared to existing techniques on both large and small updates. We also provide a discussion of the practical considerations for building incremental evaluation in Soufflé that include relational data structures, parallelization and scheduling strategies.

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

(1) We present a new problem - that incremental evaluation should be *elastic*, i.e., it should be sensitive towards the impact of an update.
(2) We present a novel incremental evaluation, using a sparse proof counting encoding, exhibiting superior performance and lower memory overhead for elastic use cases.
(3) We extend Soufflé, an open-source Datalog evaluation engine for elastic incremental evaluation and propose several engine optimizations for superior performance.
(4) We provide an extensive experimental evaluation validating the utility of our contribution.

2 BACKGROUND

In this section, we provide an example to explain the background of Datalog evaluation.

2.1 Example: Datalog Pointer Analysis

We present an example Datalog program analysis that computes the pairs of variables that may *alias* in a source program.

Figure 3a shows an fragment of object-oriented source code, which is encoded in the form of relations in Figure 3b, and represented diagrammatically in Figure 4. From this encoded relational

```
L1: a = new O();
L2: b = a;

L3: c = new P();
L4: d = new P();

L5: c.f = a;
L6: e = d.f;
L7: b = c.f;
L8: a = b;
```

(a) Input Program

```
new(a, L1).
assign(b, a).

new(c, L3).
new(d, L4).

store(c, f, a).
load(e, d, f).
load(b, c, f).
assign(a, b).
```

(b) EDB Tuples

```
vpt(Var, Obj) :- new(Var, Obj).       //r1
vpt(Var, Obj) :- assign(Var, Var2),
                 vpt(Var2, Obj).      //r2
vpt(Var, Obj) :- load(Var, Y, F),
                 store(P, F, Q),
                 vpt(Q, Obj),
                 vpt(P, Obj2),
                 vpt(Y, Obj2).        //r3
alias(Var1, Var2) :- vpt(Var1, Obj),
                     vpt(Var2, Obj). //r4
```

(c) Datalog Pointer Analysis

Figure 3: Program Analysis Datalog Setup

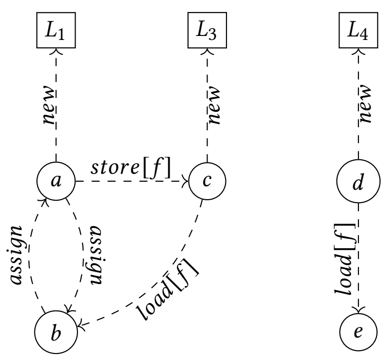


Figure 4: Pointer Input Diagram

representation of the source program, a (field sensitive but flow-insensitive [37]) pointer analysis is written in Datalog, in Figure 3c. In this analysis, the input relations (also known as extensional database, or EDB) are `new`, `assign`, `load`, and `store`, each of which represent a certain type of operation in the source program. During the analysis, the Datalog specification computes output relations (also known as intensional database, or IDB) `vpt`, which relates variables and the objects that they point to, and `alias`, which relates pairs of variables which may point to the same object.

This logic specification consists of four *rules* (here labelled `r1` through `r4`). Each rule is a Horn clause consisting of two parts: the *predicate* on the left of the implication sign (`:-`) is the *head*, and the set of predicates on the right is the *body*. Each predicate consists of a *relation name* and a sequence of constants and variables of appropriate arity. For example, the rule

```
vpt(Var,Obj) :- assign(Var,Var2), vpt(Var2,Obj).
```

has the predicate `vpt(Var,Obj)` as the head, and the two predicates `assign(Var,Var2)` and `vpt(Var2,Obj)` as the body. Negation and constraints are omitted for now, but are discussed in more detail in Section 3.3.

A predicate may be *instantiated*, where all its variables are mapped to constants to form a *tuple*. An instantiated rule is a rule where each predicate is instantiated, such that the variable mappings are compatible between all the predicates. A Datalog rule is read from right to left as a universally quantified implication: "for all rule instantiations, if every tuple in the body is derivable, then the corresponding tuple for the head is also derivable".

2.2 Semi-Naïve Evaluation

To evaluate a Datalog specification, modern engines use a *bottom-up* approach, which begins from the input tuples, and in each step attempts to derive more tuples using an *immediate consequence operator* $\Gamma_P(I) = I \cup \{t \mid r = t \text{ :- } t_1, \ldots, t_n,\ \text{each } t_i \in I\}$ such that r is a valid instantiation of a rule in P with each $t_i \in I$. The evaluation ends when a *fixed-point* is reached. Many Datalog solvers improve on this bottom-up strategy by utilizing *semi-naïve* evaluation. Semi-naïve evaluation proposes two main optimizations: (1) Stratification: the Datalog specification is split into *strata*. Firstly, a precedence graph of relations is computed, where for relations R_{body} and R_{head}, there is an edge from R_{body} to R_{head} if R_{body} appears in the body of a rule with R_{head} in the head. Then, each strongly connected component of the precedence graph forms a stratum. Each stratum is evaluated in a bottom-up fashion as a separate fixpoint computation in order based on the topological order of SCCs. The input to a particular stratum is the relations in the previous strata in the precedence graph. (2) New knowledge optimization: within a single stratum, the evaluation is optimized in each iteration by considering the new tuples generated in the previous iteration. A new tuple is generated in the current iteration only if it directly depends on tuples generated in the previous iteration, therefore avoiding the recomputation of tuples already computed in prior iterations.

The standard semi-naïve evaluation is presented in Algorithm 1 for a single stratum. The inputs for the algorithm are E, the input set of tuples (since this is a single stratum, the input may be EDB tuples, or tuples from earlier strata), and P, the set of Datalog rules forming the stratum.

Algorithm 1 Semi-Naïve(E, P)

1: $\Delta_0 \leftarrow E$
2: **for all** $k \in \{1, 2, \ldots\}$ **do**
3: $\quad I_{k-1} \leftarrow \bigcup_{0 \le i < k} \Delta_i$
4: $\quad \Delta_k \leftarrow \Pi_P[I_{k-1} \mid \Delta_{k-1}] \setminus I_{k-1}$
5: $\quad$ **if** $\Delta_k = \emptyset$ **then**
6: $\quad\quad$ **return** I_{k-1}

This algorithm begins by initializing the delta and the full set of tuples from the input (line 1). In the fix-point loop, line 4 is the critical line, evaluating the Datalog rules. This line uses notation adapted from [26], which introduces a *rule evaluation operator*, Π, where

$$\Pi_P[I \mid \Delta] = \left\{ t \;\middle|\; \begin{array}{l} t \texttt{ :- } t_1, \ldots, t_n \text{ is instantiation of rule in } P \text{ where} \\ \{t_1, \ldots, t_n\} \subseteq I \text{ and } \{t_1, \ldots, t_n\} \cap \Delta \neq \emptyset \end{array} \right\}$$

Here, Π_P finds the head tuples of all rules in P instantiated from tuples in I, where at least one body tuple also exists in Δ. For the rest of this paper, the program P is omitted from Π_P where it is clear. The dependence on Δ is the new knowledge optimization in semi-naïve evaluation. By requiring that at least one body tuple for each rule derivation is contained in Δ_{k-1}, the algorithm ensures that new tuples are only generated from tuples that were new in the previous iteration.

Algorithm 1 continues by merging the newly discovered tuples into the full relation (line 3), and if a fix-point has been reached (i.e., no new tuples are generated), then the evaluation ends (line 5).

As a concrete example of semi-naïve evaluation, consider the recursive stratum containing `vpt` in the running example. In the initialization phase, the algorithm simply copies the inputs. Therefore, in iteration 0,

$$\Delta_0 = I_0 = \left\{ \begin{array}{l} \texttt{new(a, L1)}, \texttt{new(c, L3)}, \texttt{new(d, L4)}, \texttt{assign(a, b)}, \\ \texttt{assign(b, a)}, \texttt{store(c, f, a)}, \texttt{load(e, d, f)}, \texttt{load(b, c, f)} \end{array} \right\}$$

In iteration 1, note that the `vpt` relation is empty in I_0. Therefore, only the non-recursive rule `r1` can be applied, generating

$$\begin{aligned} \Delta_1 &= \{\texttt{vpt(a, L1)}, \texttt{vpt(c, L3)}, \texttt{vpt(d, L4)}\} \\ I_1 &= I_0 \cup \Delta_1 \end{aligned}$$

Using I_1 and Δ_1, the algorithm can now apply the recursive rules of `vpt` as well. Rule `r1` no longer applies, since there are no tuples from relation `new` in Δ_1. From rule `r2`, we can derive `vpt(b,L1)` from the instantiation `vpt(b,L1) :- assign(b,a), vpt(a,L1)`. From rule `r3`, we can again derive `vpt(b,L1)`, from `vpt(b,L1) :- load(b,c,f), store(c,f,a), vpt(a,L1), vpt(c,L3), vpt(c,L3)`. Therefore, these two derivations generate the same tuple, and so,

$$\begin{aligned} \Delta_2 &= \{\texttt{vpt(b, L1)}\} \\ I_2 &= I_1 \cup \Delta_2 \end{aligned}$$

In iteration 3, rule `r2` can generate `vpt(a,L1)`. However, this tuple is already contained in I_2, and therefore $I_3 = I_2$ and a fixpoint is reached.

2.3 Incremental Datalog Evaluation

Incremental evaluation refers to a procedure to *update* the result of the Datalog computation given some changes in the input, without performing a full recomputation. An incremental evaluation proceeds in *epochs*, where each epoch represents one round of updates, i.e., inserting/deleting tuples from the input, and computing the new result and state. We refer to the inserted and deleted tuples as the *diff*. For the workflow in Fig. 2c, each I_k represents the result of epoch k, and each ΔE_k represents the corresponding diff. To summarize, the central problem of incremental evaluation is as follows:

Definition 2.1 (Incremental Evaluation). Given a Datalog program P, an input data set E, the result $P(E)$, an insertion set E^+ and a deletion set E^-, compute the result $P((E \cup E^+) \setminus E^-)$.

The cost of an update is usually measured by its impact. Typically high impact changes result in more computation overhead.

Definition 2.2 (Incremental Update Impact). The impact of an update is the number of IDB tuples changed as a consequence of the update i.e., ΔI.

We note that while the state-of-the-art incremental evaluation strategies, such as DRed [13], its related strategies [15, 16, 26], and counting-based algorithms [23, 25] have proven worthwhile for applications where each update has a small impact on the computed result, we have observed that this assumption does not hold in general for all incremental workloads. For a concrete example, consider our running example. We may remove the line `L6` in Figure 3a as part of an update to the software. This removed line would result in the input tuple `load(e,d,f)` being removed. From the graph in Figure 4, this only affects a single edge, and does not affect the connected component containing `a`, `b`, and `c`. Therefore, computing the result after performing this update should take advantage of this separation, and this update has *small* impact. However, imagine also removing the line `L1` as part of the same software update. Then, the input tuple `new(a,L1)` would be deleted, and both connected components in Figure 4 would be affected. This results in an update with *large* impact, where half of the tuples in `vpt` are deleted, and all of the tuples in `alias` are deleted. In these situations, where both small and large updates may be present, state-of-the-art incremental evaluation strategies may not be effective.

3 ELASTIC INCREMENTAL EVALUATION

This section describes our algorithms for elastic incremental evaluation. Recall from Fig. 2 that we have two evaluation procedures, one to initialize the computation state and one to incrementally update it. We call these evaluations *Bootstrap* and *Update* strategies, respectively (see Fig. 2c). Our Bootstrap strategy mimics a standard semi-naïve evaluation that also computes the computational state to allow subsequent updates. The bootstrap strategy either initiates the streaming or is a restart strategy for large updates. Recall that the update strategy needs a notion of computational state σ, which is carried from one epoch to the next. Traditionally, this computational state involves a long vector of numbers per tuple in the IDB [23, 25], where each number represents a count in each iteration of the fix-point computation. In the worst case, the length of the vector is determined by the number of iterations m in the fix-point computation. Hence, the state may exhibit a worst-case space complexity of $O(m|I|)$ where $|I|$ is the size of the output.

Our approach maintains a lightweight state, where each tuple is associated with a sparsified version of the traditional state, maintaining only two numbers per tuple. Its worst-case space complexity is $O(|I|)$. Our lightweight computational state σ shortens the runtime of the Bootstrap evaluation so that it can be used for high-impact updates with the trade-off that the Update evaluation strategy may require more work. When given an incremental update, we provide a heuristic for switching between both strategies. Apart from the initial epoch, we first attempt using the Update strategy. If it times out (the time-out is set to some fraction using a *switching parameter* of the previous Bootstrap's runtime strategy), we discard its partial state and produce the output and the computational state from scratch using Boostrap. The time-out is dependent on the application and needs to be fine-tuned appropriately.

The computational state of our approach is formed by two numbers per tuple: The first number is a *proof count* (the number of ways that the tuple can be derived in the iteration when it can be deduced the first time), and the second number is the iteration in which the tuple is first derived.

We introduce some notation for describing our approach. We define a sequence of sets $\langle D_1, D_2, \ldots \rangle$ where set D_k denotes the set of *rule instantiations*. Set $D_k = \{(t \text{ :- } t_1, \ldots, t_n)\}$ contains all the rule instantiations that are computed in iteration k. The proof support count of tuple t in iteration k is the number of rule instantiations $(t \text{ :- } t_1, \ldots, t_n)$ whose head is t. For the sake of simplicity, we define $\mathcal{N}^\#$ as a sequence of counting multisets for describing the proof support of tuples. We use the standard definition of multisets, where each $\mathcal{N}^\#_k = \{(t \mapsto c)\}$ denotes the number of rule instantiation t :- $t_1, \ldots, t_n$ for tuple t in D_k. For notational convenience, we will express the elements with multiplicities $t \mapsto c$ as t^c.

3.1 Bootstrap Algorithm

The Bootstrap algorithm is a specialized counting algorithm for efficiently computing the sequence of multisets from scratch mimicking a semi-naive evaluation producing the computational state as a side-effect. For example, consider our running example. In the initial phase, the input E becomes iteration 0, where the counting semantics mean that every tuple has a count of 1. Therefore,

$$\mathcal{N}^\#_0 = \left\{ \begin{array}{l} \texttt{new(a,L1)}^1, \texttt{new(c,L3)}^1, \texttt{new(d,L4)}^1, \texttt{assign(a,b)}^1, \\ \texttt{assign(b,a)}^1, \texttt{store(c,f,a)}^1, \texttt{load(e,d,f)}^1, \texttt{load(b,c,f)}^1 \end{array} \right\}$$

In iteration 1, we apply the non-recursive rule `r1`. In this case, all tuples have a count of 1:

$$\mathcal{N}^\#_1 = \left\{ \texttt{vpt(a,L1)}^1, \texttt{vpt(c,L3)}^1, \texttt{vpt(d,L4)}^1 \right\}$$

In iteration 2, however, the counting semantics causes a divergence from the standard semi-naïve evaluation. Recall that the tuple `vpt(b,L1)` is derivable from two rules:

(1) `vpt(b, L1) :- assign(b, a), vpt(a, L1)`, and
(2) `vpt(b, L1) :- load(b, c, f), store(c, f, a), vpt(a, L1), vpt(c, l3), vpt(c, l3)`

Therefore, `vpt(b, L1)` has a count of 2 in iteration 2:

$$\mathcal{N}^\#_2 = \left\{ \texttt{vpt(b,L1)}^2 \right\}$$

In iteration 3, no new tuples are derivable. Therefore, a fixpoint has been reached, and the Datalog evaluation ends.

We present the bootstrap algorithm in Algorithm 2. The main extension from the standard semi-naïve evaluation (Algorithm 1) is that the algorithm generates $\mathcal{N}^\#$, the sequence of multisets, in contrast to the standard sets in standard semi-naïve. To compute these multisets, we first introduce a version of the rule evaluation operator that computes sets of *rule instantiations*:

$$\Pi^D_P[I \mid I_{\text{in}}] =$$
$$\left\{ (t \text{ :- } t_1, \ldots, t_n) \;\middle|\; \begin{array}{l} t \text{ :- } t_1, \ldots, t_n \text{ in } P \text{ where } \{t_1, \ldots, t_n\} \subseteq I \\ \text{and } \{t_1, \ldots, t_n\} \cap I_{\text{in}} \neq \emptyset \end{array} \right\}$$

From the Π^D_P operator, we can define a counting version:

$$\Pi^\#_P[I \mid I_{\text{in}}] =$$
$$\left\{ t^v \;\middle|\; v = \#\text{rule instantiations } (t \text{ :- } t_1, \ldots, t_n) \in \Pi^D_P[I \mid I_{\text{in}}] \right\}$$

where v is the number of ways that the tuple t can be derived.

Algorithm 2 presents the light-weight bootstrap algorithm for a single stratum. Its structure is almost identical to the standard semi-naïve evaluation algorithm. The main difference for Bootstrap is that it maintains a separate sequence of multisets $\mathcal{N}^\#$, where each $\mathcal{N}^\#_k$ is similar to Δ_k of semi-naïve, and contains all the new tuples computed in iteration k. The algorithm begins by initializing $\mathcal{N}^\#_0$ to be equal to E (line 1). Here, the assignment of a set to a multiset is defined as $\mathcal{N}^\#_0 = \{(t^1) \mid t \in E\}$, where every element of E is taken with a count of 1. In the fixpoint loop, the algorithm first creates a set projection of the current iteration's multiset (line 3), where the operator taking the support of a multiset is defined as $\text{Supp}(\mathcal{N}^\#_{k-1}) = \{t \mid (t^c) \in \mathcal{N}^\#_{k-1} \text{ and } c > 0\}$. Intuitively, $\text{Supp}(\mathcal{N})$ is the set of tuples in $\mathcal{N}$ with count greater than 0. The algorithm also computes the full state of the relations up to iteration $k-1$ (line 4), in the same way as the semi-naïve algorithm. These two auxiliary sets, B_{k-1} and I_{k-1}, are used in the rule evaluation on line 5. This rule evaluation computes all tuples that are new in the current iteration. The set minus operation is an abuse of notation, operating as a filter to exclude any tuples that were computed in earlier iterations. The algorithm exits if there are no new tuples generated in the current iteration, returning the evaluation state $(E, \mathcal{N}^\#)$ (line 6).

Algorithm 2 Bootstrap(E)

1: $\mathcal{N}^\#_0 \leftarrow E$ $\triangleright$ $\mathcal{N}^\#_0 \leftarrow \{(t^1) \mid t \in E\}$
2: **for all** $k \in \{1, 2, \ldots\}$ **do**
3: $\quad N_{k-1} \leftarrow \text{Supp}(\mathcal{N}^\#_{k-1})$
4: $\quad I_{k-1} \leftarrow \cup_{0 \le i \le k-1} B_i$
5: $\quad \mathcal{N}^\#_k \leftarrow \{(t^v) \in \Pi^\#[I_{k-1} \mid B_{k-1}] \mid t \notin I_{k-1}\}$
6: $\quad$ **if** $\text{Supp}(\mathcal{N}^\#_k) = \emptyset$ **then**
7: $\qquad$ **return** $(E, \mathcal{N}^\#)$

Correctness. To demonstrate the correctness of Algorithm 2, we need to show that it computes the same resulting set of tuples as standard semi-naïve evaluation (Algorithm 1). To do this, we need to demonstrate two basic properties: (*a*) each N_k of Bootstrap is equal to Δ_k of semi-naïve, and (*b*) both Bootstrap and semi-naïve evaluation terminate after the same number of iterations. To show this, we introduce the following lemma:

Lemma 3.1. *Given a Datalog program P, for all A, B such that $B \subseteq A$, $\text{Supp}(\Pi^\#_P[A \mid B]) = \Pi_P[A \mid B]$.*

This property can be shown since a tuple $t \in \Pi_P[A \mid B]$ if and only if there is a rule instantiation that computes it. If this is the case, then the same rule instantiation also fits $\Pi^\#_P[A \mid B]$ with a count of at least one. As a corollary, we can show that Bootstrap and semi-naïve both produce the same set of tuples in each iteration.

Lemma 3.2. *Given a Datalog program P and an input set E, each I_{k-1} of Bootstrap is equal to I_{k-1} of semi-naïve.*

This can be shown by an induction over k, since $\mathrm{Supp}(\mathcal{N}_i^\#) = \Delta_i$ (from Lemma 3.1) for each iteration i, then the result in each iteration must be identical to semi-naïve. Note that both Bootstrap and semi-naïve terminate after the same number of iterations since $\mathrm{Supp}(\mathcal{N}_i^\#) = \Delta_i$ for every iteration i, and therefore $\mathrm{Supp}(\mathcal{N}_i^\#) = \emptyset$ if and only if $\Delta_i = \emptyset$. Therefore, both algorithms terminate after the same number of iterations, and thus produce the same set of resulting tuples.

The computational state generated by Bootstrap is sparse since a tuple can only be contained at most in a single set $\mathcal{N}_i^\#$. The exclusion clause in Line 5, i.e., $t \notin I_{k-1}$, ensures that property.

3.2 Incremental Update Algorithm

The Update algorithm is a procedure that takes a computational state, either computed by Bootstrap or by a previous Update, and a set of changes to the inputs. The algorithm updates the computational state to reflect the changes that result from the input changes and produces the output for the epoch.

At a high level, there are two cases for how a tuple update is handled: (1) A tuple is inserted if it is the head of an instantiated rule where either (*a*) one of the tuples in the body of the rule is newly inserted in the current epoch, (*b*) the same head tuple was deleted in a prior iteration and there is an alternative derivation in the current iteration. (2) A tuple is deleted if it is the head of an instantiated rule where either (*a*) one of the tuples in the body of the rule is deleted in the current epoch, or (*b*) an alternative derivation is found for the tuple in an earlier iteration current epoch.

The incremental update algorithm also introduces extended notation over Bootstrap. The rule evaluation notation, $\Pi_P[I \mid I_{\text{in}}]$, denotes tuples resulting from rules in P instantiated from I, with at least one body tuple also in I_{in}. This is extended with

$$\Pi_P^\#[I \mid I_1 \mid I_2] = \left\{ t^v \;\middle|\; \begin{array}{l} v = \text{number of rule instantiations } t \;{:}-\; t_1, \ldots, t_n \\ \text{in } P \text{ where } \{t_1, \ldots, t_n\} \subseteq I \text{ and } \{t_1, \ldots, t_n\} \cap I_1 \neq \emptyset \\ \text{and } \{t_1, \ldots, t_n\} \cap I_2 \neq \emptyset \end{array} \right\}$$

This notation derives tuples from rule instantiations where at least one body tuple is from I_1, and also at least one body tuple is from I_2. In Update, I_1 and I_2 would be the deltas from semi-naïve evaluation and the diffs from the incremental update, allowing it to compute tuples that are newly changed in the current iteration of the current epoch. Additionally, the algorithm uses $\oplus$ and $\ominus$, the standard multiset addition and subtraction operators for operations involving multisets.

The update algorithm computes the updates to the sequence of multisets $\mathcal{N}^\#$, which result from applying the insertions and deletions to the input. The algorithm also makes use of a number of auxiliary sets: I_k^o and I_k maintain the full sets of tuples up to iteration k for the previous and current epoch respectively, I_k^- and I_k^+ maintain the tuples that are deleted and inserted respectively up to iteration k, and N_k^o and N_k are the simple set projections of $\mathcal{N}_k^{\#o}$ and $\mathcal{N}_k^\#$ and store the tuples that are new in iteration k in the previous and current epoch respectively.

Algorithm 3 is presented for a single stratum, and takes the state the previous epoch $(E, \mathcal{N}^{\#o})$, and the incremental update (E^-, E^+) consisting of a set of tuples to be deleted and a set of tuples to

Algorithm 3 Update($(E, \mathcal{N}^{\#o})$, (E^-, E^+))

Ensure: $E^- \subseteq E$, $E \cap E^+ = \emptyset$

1: $\mathcal{N}_0^\# \leftarrow E \setminus E^- \cup E^+$
2: $I_0^- \leftarrow E^-$
3: $I_0^+ \leftarrow E^+$
4: **for all** $k \in \{1, 2, \ldots\}$ **do**
5: $\quad N_{k-1} \leftarrow \mathrm{Supp}(\mathcal{N}_{k-1}^\#)$
6: $\quad N_{k-1}^o \leftarrow \mathrm{Supp}(\mathcal{N}_{k-1}^{\#o})$
7: $\quad I_{k-1} \leftarrow \cup_{0 \leq i \leq k-1} N_i$
8: $\quad I_{k-1}^o \leftarrow \cup_{0 \leq 1 \leq k-1} N_i^o$
9: $\quad \mathcal{N}_k^\# \leftarrow \mathcal{N}_k^{\#o} \ominus (\Pi^\#[I_{k-1}^o \mid N_{k-1}^o \mid I_{k-1}^-] \setminus I_{k-1}^o)$
$\qquad \oplus (\Pi^\#[I_{k-1} \mid N_{k-1} \mid I_{k-1}^+] \setminus I_{k-1})$
$\qquad \oplus (I_{k-1}^- \cap \Pi^\#[I_{k-1}^o \cap I_{k-1} \mid N_{k-1}] \setminus I_{k-1})$
10: $\quad \mathcal{N}_k^\# \leftarrow \{(t^v) \in \mathcal{N}_k^\# \mid t \notin I_{k-1}^+\}$
11: $\quad I_k^- \leftarrow (I_{k-1}^- \setminus N_k) \cup (N_k^o \setminus I_k)$
12: $\quad I_k^+ \leftarrow (I_{k-1}^+ \setminus N_k^o) \cup (N_k \setminus I_k^o)$
13: $\quad$ **if** $\mathcal{N}_k^\# = \mathbf{0}$ **then**
14: $\qquad$ **return** $(E \setminus E^- \cup E^+, \mathcal{N}^\#)$

be inserted, respectively. Note that $\mathcal{N}^\#$ may be the IDB sequence from the bootstrap stage, $\mathcal{B}^\#$, or it may be the result of a previous incremental update. The algorithm begins by initializing the state of the input by applying E^- and E^+, and storing the result in $\mathcal{N}_0^\#$ (line 1). Then, the algorithm initializes the sets I_0^- and I_0^+ to be the updates in iteration 0.

In the fixpoint loop, the rule evaluation on line 9 is the core part of this algorithm. This step starts with the multiset of tuples from the previous epoch and applies deletions and insertions resulting from applying Datalog rules. The deletion term, $\Pi^\#[I_{k-1}^o \mid N_{k-1}^o \mid I_{k-1}^-] \setminus I_{k-1}^o$, computes tuples that are deleted in the current iteration as a result of a derivation where the body contains both a tuple in the delta (N_{k-1}^o) and a deleted tuple (I_{k-1}^-). This abuses notation (similarly to line 5 of Bootstrap) to exclude tuples that were in earlier iterations in the previous epoch, preventing over-deletion since the tuples would not be present in the current iteration due to sparsification. The insertion term, $\Pi^\#[I_{k-1} \mid N_{k-1} \mid I_{k-1}^+] \setminus I_{k-1}$, computes tuples that are inserted as a result of the body of a derivation containing an inserted tuple. Tuples that already exist in previous iterations (i.e., tuples that are contained in I_{k-1}) are excluded to maintain the sparsification invariant. The re-discovery term, $I_{k-1}^- \cap \Pi[I_{k-1}^o \cap I_{k-1}^n \mid N_{k-1}]$, computes tuples that are deleted in previous iterations I_{k-1}^-, but where an alternative derivation exists in the current iteration. This re-discovery rule applies in the situation where a tuple is deleted from some iteration, but can still be derived in a later iteration. In this case, the re-discovery term computes this later derivation.

The sparsification term (line 10) does not perform any rule evaluation, but instead excludes tuples from iteration k that were inserted in an earlier iteration (as a result of a new derivation). These tuples should be deleted to maintain the sparsification invariant that a tuple is only present in a single iteration in any given epoch.

The algorithm continues by updating the I_k^- and I_k^+ sets (lines 11 and 12). Computing I_k^- (line 11) takes the deletion set from the previous iteration I^-_{k-1} and excludes the tuples that are newly computed in the current iteration N_k, along with tuples that are deleted in the current iteration ($N^o_k \setminus I^n_k$). Similarly, computing I^+_k (line 12) takes the insertion set from the previous iteration, and excludes tuples that already existed in the current iteration in the previous epoch (since these tuples already existed, so are not *newly* inserted in the current epoch), along with tuples that are inserted in the current iteration.

Correctness. To show the correctness of our incremental update algorithm, we must show that it computes the same sequence of multisets as if we had applied bootstrap to the altered input. In other words, we need to show that given a Datalog program P, an input set E, a deletion set E^- and an insertion set E^+, computing the result directly via Bootstrap($E^b = E \setminus E^- \cup E^+$) is equal to Update(Bootstrap(E), (E^-, E^+)). The central parts of the algorithm computing these results are lines 9 and 10. Before the final correctness proof, we need some intermediate properties of the N_k sets and the I^- and I^+ sets. The next important properties are that the validity properties of the E sets (i.e., that $E^+ \cap E = \emptyset$ and $E^- \subseteq E$) also hold for the I^o, I^-, and I^+ sets during the incremental update algorithm. Similar properties relating I^- and I^+ sets to the current epoch's I sets are also required. The eventual goal is to show that $I_k = I^o_k \setminus I^-_k \cup I^+_k$ for each iteration k, which is an important result for showing the correctness of the rule evaluations.

Lemma 3.3. *For each iteration k, we have (1) $I^-_k \subseteq I^o_k$ and $I^-_k \cap I_k = \emptyset$, and (2) $I^+_k \cap I^o_k = \emptyset$ and $I^+_k \subseteq I_k$.*

To sketch the proof for this property, we perform an induction over the iterations. The base case holds because of the definition of I^-_0 and I^+_0. Then, for each subsequent iteration, consider line 11 of Algorithm 3. Here, I^-_k takes the value of $(I^-_{k-1} \setminus N_k) \cup (N^o_k \setminus I_k)$. In the first part of the union, the property holds for I^-_{k-1} by the induction hypothesis. In the second part of the union, N^o_k is a subset of I^o_k by definition. Therefore, $I^-_k \subseteq I^o_k$. By similar arguments on line 12, $I^+_k \subseteq I_k$. The second part of the property, i.e., that $I^-_k \cap I_k = \emptyset$ can be shown by a similar induction argument, again consider line 11.

As a corollary, we can show that the I^- and I^+ sets are correct.

Corollary 3.4. *For each iteration k, we have $I_k = I^o_k \setminus I^-_k \cup I^+_k$.*

It remains to be shown that Update is correct. Our criteria for correctness is that it computes the same sequence of multisets as if we had applied the bootstrap algorithm to the updated input, i.e., that $\mathcal{B}^\#_i = \mathcal{N}^\#_i$ for each iteration i. This is the central theorem for our correctness proof.

Theorem 3.5. *Given P, E, E^-, and E^+ as above, $\mathcal{N}^\#_i$ as computed by IncrementalUpdate(Bootstrap(E), (E^-, E^+)) is equal to $\mathcal{B}^\#_i$ as computed by Bootstrap($E \setminus E^- \cup E^+$) for each iteration i.*

The proof of Theorem 3.5 is an induction over the iterations, and in each step, it considers all four parts of lines 9 and 10. By arguments over which sets each tuple is contained in, and careful consideration of the subset relationships between them, we can show that the counting multisets are the same as those produced by Bootstrap.

Sparsification. Another important property of our elastic incremental evaluation strategy is the *sparsification invariant.*

Lemma 3.6 (Sparsification Invariant). *For each iteration k, the sets N_k are disjoint.*

This property ensures that every tuple is only computed in a single iteration, with this iteration being the earliest one in which it is computed.

Re-discovery rules as a notion of provenance. The re-discovery part of the rule evaluation part of Algorithm 3 is (the last part of line 9) is critical for maintaining the sparsification property of our algorithm. The rule evaluation $I^-_{k-1} \cap \Pi[I^o_{k-1} \cap I^n_{k-1} \mid N_{k-1}]$ states that we compute tuples that were deleted in an earlier iteration (i.e., exist in I^-_{k-1}), but an alternative derivation exists for the current iteration ($\Pi[I^o_{k-1} \cap I^n_{k-1} \mid N_{k-1}]$). This re-discovery rule is a notion of *provenance* for the deleted tuples. Provenance is defined as "discovering the derivations for a tuple", and this fits the process of finding derivations from $I^o_{k-1} \cap I^n_{k-1}$ for all the deleted tuples. Therefore, we adapt techniques from [41] to compute these re-discovery rules.

3.3 Stratified Negation and Constraints

Our algorithms thus far have omitted any notion of negation or constraints. However, both negation and constraints are powerful and common extensions of Datalog. Constraints are a simpler case than negation, and may take the form of arithmetic constraints such as `A < B` or `A != B` where A and B are *grounded* variables (i.e., variables also occurring in a positive body predicate) or constants. In an instantiated rule, a constraint is satisfied if the instantiated arithmetic constraint is satisfied. For example,

```
alias(Var1,Var2) :- vpt(Var1,Obj), vpt(Var2,Obj),
                    Var1 != Var2
```

is a rule with arithmetic constraints, and an instantiation of the rule only derives a tuple if the inequality constraint is satisfied by the values given to `Var1` and `Var2`.

Negation is more complicated than simple arithmetic constraints. Syntactically, negations are denoted as a negated predicate with the `!` symbol. For example, the rule

```
path(X,Z) :- edge(X,Y), path(Y,Z), !edge(X,Z)
```

computes all the paths in a graph which are not direct edges. A negated predicate must contain only grounded variables or constants, and a negated predicate is satisfied if and only if the corresponding tuple (resulting from an instantiation) is not computable. The standard semantics for negation in Datalog is *stratified negation.* In this semantics, recursive negation is not permitted, and any negated predicates must be of a relation from either input or a previous stratum. With this semantics, a negated predicate is similar to a constraint, where only a simple check of the input for a stratum is needed to determine whether it is satisfied or not. However, the truth value of a negation may change as a result of tuples being inserted or deleted from the negated relation. To adapt our Datalog evaluation algorithms to support stratified negation and constraints, the rule evaluation is extended to support these features. The rule evaluation operator, $\Pi^\#$ is extended so that

$$\Pi^\#_P[I \mid I_{\text{in}}] = \left\{ t^v \;\middle|\; \begin{array}{l} v = \#\text{instantiations } t \text{ :- } t_1, \ldots, t_n, !t_{n+1}, \ldots, !t_{n+m}, \psi \\ \text{in } P \text{ where } \{t_1, \ldots, t_n\} \subseteq I, \{t_1, \ldots, t_n\} \cap I_{\text{in}} \neq \emptyset, \\ \{t_{n+1} \ldots t_{n+m}\} \cap I = \emptyset, \text{ and } \psi \text{ is satisfied} \end{array} \right\}$$

where ψ denotes the instantiated arithmetic constraints occurring in the rule. Replacing the rule evaluation operator in Bootstrap (Algorithm 2) with this extended version allows the algorithm to support stratified negation and constraints. However for incremental update, the extension is more involved, since introducing negation also introduces new cases for deleting/inserting tuples. For example, consider the rule

```
path(X,Z) :- edge(X,Y), path(Y,Z), !edge(X,Z).
```

If we have a rule instantiation

```
path(a,c) :- edge(a,b), path(b,c), !edge(a,c)
```

where `edge(a,c)` is inserted as a result of an incremental update, then the head tuple `path(a,c)` must be deleted since the negation is no longer satisfied. The opposite situation may arise where the deletion of a tuple may lead to the insertion of a consequent tuple. Therefore, we further extend the rule evaluation operator so that

$$\Pi^\#_P[I \mid I_1 \mid I_2, I_2'] = \left\{ t^v \;\middle|\; \begin{array}{l} v = \#\text{instantiations } t \text{ :- } t_1, \ldots, t_n, !t_{n+1}, \ldots, !t_{n+m}, \psi \\ \text{in } P \text{ where } \{t_1, \ldots, t_n\} \subseteq I, \{t_1, \ldots, t_n\} \cap I_1 \neq \emptyset, \\ \{t_{n+1} \ldots t_{n+m}\} \cap I = \emptyset, \psi \text{ is satisfied, and} \\ (\{t_1, \ldots, t_n\} \cap I_2 \neq \emptyset \text{ or } \{t_{n+1} \ldots, t_{n+m}\} \cap I_2' \neq \emptyset) \end{array} \right\}$$

With this rule evaluation operator, a new tuple is derived if the rule instantiation contains body tuples from I, where at least one positive body tuple is also in I_1, and either there is a positive body tuple in I_2 or a negative body tuple in I_2'. Using this notation, the rule evaluation step of Algorithm 3 (line 9) becomes

$$\begin{aligned} \mathcal{N}_k^\# \leftarrow \mathcal{N}_k^\# &\ominus (\Pi^\#[I_{k-1}^o \mid N_{k-1}^o \mid I_{k-1}^-, I_0^+] \setminus I_{k-1}^o) \\ &\oplus (\Pi^\#[I_{k-1} \mid N_{k-1} \mid I_{k-1}^+, I_0^-] \setminus I_{k-1}) \\ &\oplus (I_{k-1}^- \cap \Pi[I_{k-1}^o \cap I_{k-1} \mid N_{k-1}]) \end{aligned}$$

where the first and second terms now handle stratified negation. The deletion term, $\Pi^\#[I_{k-1}^o \mid N_{k-1}^o \mid I_{k-1}^-, I_0^+] \setminus I_{k-1}^o$, now computes tuples that are deleted, either as a result of a deleted positive body tuple (I_{k-1}^-) or an inserted negated body tuple (I_0^+). We use iteration 0 for the negated tuples since stratified negation enforces that negations must be from the input of the current stratum. Similarly, the insertion term, $\Pi^\#[I_{k-1}^n \mid N_{k-1} \mid I_{k-1}^+, I_0^-] \setminus I_{k-1}^n$, computes tuples that are inserted either as a result of an inserted positive body tuple or a deleted negative body tuple. The other parts of the algorithms involve manipulating and merging relations, and are independent of the Datalog rules. Therefore, no changes are needed to support negation or constraints. Hence, with the extensions to the rule evaluation presented above, our algorithms fully support Datalog with stratified negation and constraints.

4 INTEGRATION INTO SOUFFLÉ

In this section we outline how are approach is integrated in the Soufflé Datalog engine, including several optimizations for incremental evaluation.

4.1 Core Implementation

Specialized data structures. Soufflé internally uses a highly specialized, parallel B-tree data structure to store relations. For incremental evaluation, we associate each tuple with an iteration number and a count. Therefore, the internal data structures must be extended to allow for these auxiliary attributes. Importantly, these auxiliary attributes may be *updated*, e.g., in the case of a new derivation being discovered, the count must be incremented. Thus, we implemented an update mechanism, along with adapting the existing optimistic locking mechanism to support parallel operation.

Rule evaluation. The standard rule evaluation algorithms in Soufflé are extended to support the extra operations in the incremental evaluation algorithms. Soufflé uses nested loop joins for evaluating rules, which incorporate extra conditions and existence checks to ensure correctness. For incremental evaluation, further specialized existence checks are required e.g., a tuple in `diff_minus` may not actually be deleted, and only one of its derivations is deleted. Therefore, we need a specialized existence check in the full relation to check its count to determine if the tuple is fully deleted or not. Moreover, separate versions of the rule evaluations are required for the bootstrap and incremental update algorithms.

Other operations. Along with the rule evaluation extensions, other operations such as merges between iterations, and a cleanup operation between epochs, are also required. In standard semi-naïve evaluation, at the end of each iteration, new tuples computed in that iteration are merged into the full relation, and this also becomes the delta for the following iteration. For incremental evaluation, further operations may take place, e.g., eager computation of the delta of the previous epoch, and eager computation of `diff_plus` and `diff_minus`. In between epochs, the incremental evaluation algorithms also require a cleanup stage, where the `diff_plus` and `diff_minus` relations are merged into the full relations to update the state in preparation for the following epoch.

4.2 Optimizations

Eager vs. lazy `diff_plus` *and* `diff_minus`*.* The `diff_plus` and `diff_minus` relations store tuples that are inserted and deleted in the current epoch, respectively. However, there is extra computation involved with the `diff_plus` and `diff_minus` relations, in lines 11 and 12. Here, a tuple in `diff_plus` may not actually be newly inserted - it may be a new derivation for a tuple that already existed. Similarly, a tuple in `diff_minus` may not actually be deleted - an alternative derivation may still hold. Thus, we need to check the full relation to determine if a tuple in `diff_plus` or `diff_minus` is actually inserted or deleted, respectively. This check may be performed eagerly during the merge step in each iteration, with results stored in separate relations `actual_diff_plus` and `actual_diff_minus`, or lazily inside the rule evaluation. For the sake of clarity, our algorithms are presented with eager diff computations, which can be seen in lines 11 and 12. A lazy diff version would incorporate this computation directly in the rule evaluation. This design decision is a tradeoff: eagerly computing `diff_plus` and `diff_minus` may result in wasted computation for tuples that are not considered in any rules, while lazy computation may mean the same check of the full relation is performed multiple times for a single tuple, if it occurs in multiple rule derivations. Our experiments, however, indicate that this tradeoff generally favors eager diffs, where it can amortize the checks for tuples which occur in multiple rule derivations. For our benchmarks, the difference is generally within 15% in favor of eager diffs, but it can provide up to 4× speed up in some situations.

Filtering for re-discovery rules. The elastic algorithm includes the notion of *re-discovery*, which is required due to its sparsification. In the re-discovery rules, the algorithm finds all tuples which have been deleted in an earlier iteration, but where an alternative derivation still exists for the current iteration. Naïvely, this could be done by instrumenting a rule as:

$$\mathtt{R} \ \mathtt{:-}\ \mathtt{diff\_minus\_R}, \mathtt{R}_1, \ldots, \mathtt{R}_k.$$

However, in some cases this can cause a problematic join, if there are few variables in common between the `diff_minus_R` atom and the remaining atoms in the rule. For example,

$$\mathtt{R}(x, y, z) \ \mathtt{:-}\ \mathtt{diff\_minus\_R}(x, y, z), \mathtt{R}_1(x, a), \mathtt{R}_2(y, a), \mathtt{R}_3(z, a).$$

may cause a duplication of work in $\mathtt{R}_1(x, a)$ if there are many tuples in `diff_minus_R` with the same x value. Our solution is to divide the `diff_minus` relation so that it never causes extra work.

$$\begin{aligned}\mathtt{R}(x, y, z) \ \mathtt{:-}\ &\mathtt{diff\_minus\_R}_x(x), \mathtt{R}_1(x, a), \mathtt{R}_2(y, a),\\ &\mathtt{diff\_minus\_R}_y(y), \mathtt{R}_3(z, a), \mathtt{diff\_minus\_R}_z(z).\end{aligned}$$

Dividing the `diff_minus` relations ensures that each variable only acts as a filter, and cannot multiply the work of the other atoms in the rule. Here, the x variable is scheduled first, since we assume that $\mathtt{diff\_minus\_R}_x(x)$ is smaller than $\mathtt{R}_1(x, a)$ (since we assume that changes between epochs are smaller than the full result). However, the other variables must be scheduled after their corresponding atom, to prevent a cross-product with the previous atom. This strategy of considering the attributes in each literal is an adaptation of ideas from worst-case optimal joins [28, 40]. Our benchmarks show that this technique is generally 2.5× faster than the naïve strategy, while in some situations it can be up to 15× faster.

Scheduling. Scheduling for join orders plays an important role in the performance of Datalog rules [18, 34, 35]. With incremental evaluation, the assumption that the diffs are smaller than the full relations allows for better heuristics for automatic scheduling. By using this assumption, scheduling `diff_plus` or `diff_minus` first in a rule evaluation generally improves performance by restricting the size of the search as early as possible. However, care must be taken to avoid cross-products. For example, consider the following rule:

$$\mathtt{R}(a, d) \ \mathtt{:-}\ \mathtt{R}_1(a, b), \mathtt{R}_2(b, c), \mathtt{diff\_minus\_R}_3(c, d).$$

In this case, moving $\mathtt{diff\_minus\_R}_3(c, d)$ to the front of the rule would create a cross-product with $\mathtt{R}_1(a, b)$, and may lead to worse performance than the original schedule. Hence, using simple automatic scheduling techniques, such as maximizing the number of bound variables in each atom, is crucial to maintain the performance of incremental evaluation.

5 EXPERIMENTAL EVALUATION

This experimental section aims to demonstrate the following claims:

Claim I: Inviability of single incremental evaluations on variable update use cases.

Claim II: The elastic incremental evaluation with a simple switch heuristic performs better compared to existing single strategy incremental evaluations, both in terms of runtime and memory usage, over a series of varying sized incremental updates.

Experimental Setup. Our experiments are run on an AMD Threadripper 2990WX machine with 128 GB memory, running Ubuntu 20.10 with GCC 10.2 used to generate all Soufflé executables. All experiments are run with 8 threads, and all I/O time is excluded from measurements.

We evaluate three versions of Soufflé: (1) Soufflé: Non-Incremental Soufflé engine. (2) Soufflé-counting: A baseline counting incremental algorithm implemented in Souffle with optimizations. (3) Soufflé-elastic: The implementation of the technique presented in this paper. When necessary we differentiate between elastic-update and elastic-bootstrap algorithms.

We also compare our approach to an industrial strength incremental Datalog engine, Differential Datalog (DDLog) [32], which uses Differential Dataflow [23] as a backend. DDLog with Differential Dataflow is a state-of-the-art incremental engine which uses a variant of the counting algorithm.

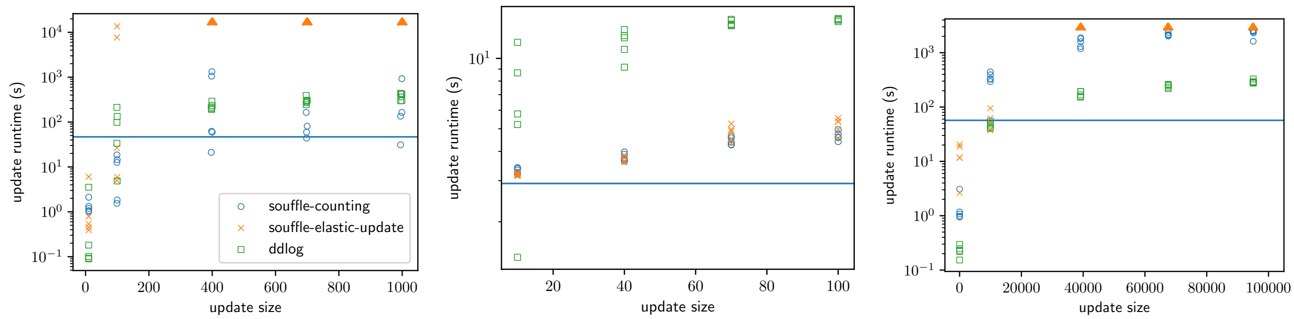
We perform our evaluations using a set of dynamic Datalog use cases adapted by Frank McSherry [1] for benchmarking incremental Datalog engines. The use cases are described below:

(1) Doop [6]: a points-to prorgam analysis framework for Java programs. This is a subset of the Doop program analysis library ported for DDLog. This use case exhibits characteristics: large number of rules, relations with complex recursion.
(2) CRDT: an implementation of a conflict-free replicated data type in Datalog. This use cases resembles an in between ruleset with a medium number of rules and relations of moderate complexity.
(3) Galen [30]: a medical ontology inference task implemented in Datalog. This use case represents a typical ontological use case consisting a a small number of rules and relations with simple recursive structure.

Some basic statistics for the benchmarks are included in Table 1. To evaluate the performance of incremental evaluation algorithms, sets of small, medium, and large updates were generated for each benchmark, by randomly choosing a subset of EDB tuples that are incrementally deleted and inserted.

Table 1: Benchmark Statistics

Benchmark	Number of rules	EDB size	IDB size
Doop	90	11,014,960	41,665,029
CRDT	31	259,778	2,668,247
Galen	6	976,552	24,483,561



(a) Doop **(b) CRDT** **(c) Galen**

Figure 5: Incremental update size vs. runtime. The horizontal line in each figure is the runtime of non-incremental Soufflé on the respective benchmark, and the upwards arrows indicate the time outs.

5.1 Single Strategy Incremental Evaluation

In these sets of experiments we only consider single strategy evaluations, that is, we only include our Update (elastic-update) evaluation and thus **do not switch to Bootstrap**. In these experiments we do not establish the supremacy of any one technique. Rather, we show that single strategies are not viable compared to non-incremental evaluation. The results for the runtime of incremental updates for each evaluation implementation are shown in Fig. 5. These results are computed for one *cycle* of an update set, where an update set is a randomly selected subset of EDB tuples, and a cycle consists of one epoch where the update set is deleted followed by one epoch where the update set is inserted. The horizontal line on each benchmark represents the runtime if non-incremental Soufflé were to perform the same task, i.e., running the whole benchmark twice from scratch. For each benchmark, there is a general trend that larger updates require more runtime. However, this performance is highly unpredictable, even if the size of the incremental update is a constant.

Consider the performance of incremental evaluation for Doop, in Fig. 5a. Here, there are five separate small update sets, which are each generated by randomly choosing 10 EDB tuples, and running one cycle. These small updates all finished within two seconds, which is vastly faster than non-incremental Soufflé. For these small incremental updates all evaluations were very fast on average due to their very low impact, only affecting up to 25 of the IDB tuples. DDlog and elastic-update performed well and on a par. Our general observation is that incremental evaluation is highly effective for these lightweight updates. For the 100 update size, the smallest impact was 88 IDB tuples, and the largest impact was 53,816 IDB tuples. As anticipated, this increased the variability of the results. The elastic-update, exhibited large extremities, finishing within 5 seconds for the fastest, while more than 5000 seconds for two update sets. DDLog, also had a high variance, with the fastest runtime being 5 seconds, and the slowest being 213 seconds, well over the non-incremental engine time. Curiously, the fastest incremental update was also one of the higher impact ones, affecting 22,347 IDB tuples, while the slowest affected 140 IDB tuples, indicating that neither the size of the EDB updates, nor the size of the impact are always helpful in predicting the runtime of the incremental update. While Soufflé-counting was generally faster than DDLog, it still exhibited a large variance, with runtimes ranging between 1.4 and 18 seconds. For the larger update sets, containing 400, 700, and 1000 tuples respectively, all evaluation strategies failed to compete with non-incremental Soufflé. For example, elastic-update was unable to complete any of the update sets within the time limit. These timed-out updates consisted of tuples that were deep in a complex recursive structure, indicating that the elastic-update algorithm does not handle these large impact updates well. Likewise the counting algorithms implemented in both DDLog and in Soufflé exhibited generally poor performance compared to non-incremental Soufflé. Furthermore, these larger updates exhibited even greater variability, particularly for Soufflé-counting.

The results for CRDT, in Fig. 5b tell a similar story. Here, even small updates consisting of 10 EDB tuples exhibit unpredictable and poor performance. In comparison to Doop, the small updates for CRDT have a much larger impact, affecting between 3,444 and 35,130 IDB tuples. However, even this larger impact is around 1% of the IDB, and even with these overall small impacts, the runtime of incremental update is considerably slower than re-running the computation from scratch in Soufflé. Similarly to Doop, the performance for larger updates only gets worse. For updates containing 40 EDB tuples, the runtimes varied between 9 and 13 seconds. While this variation is smaller than for Doop, the result still indicates that the performance of incremental evaluation is unpredictable. For larger updates containing 70 and 100 EDB tuples, DDLog was around 5× slower than non-incremental Soufflé, despite the update being only around 0.04% of the EDB and impacting only up to 3.4% of the IDB tuples. Update and Soufflé-counting were both more performant, however still slower than non-incremental Soufflé. It is interesting to note that the impact on the IDB tuples was much more consistent for CRDT when compared with Doop. For example, with updates containing 100 EDB tuples, the impact on IDB tuples ranged between 85,726 and 91,384 tuples. This may be due to the much simpler structure of the CRDT application, which contains a larger pre-processing stage followed by a very small recursive stratum.

On the other hand, Galen performed far better with DDLog for incremental evaluation. One reason for this is that Galen has a

[1]https://github.com/frankmcsherry/dynamic-datalog

simple ruleset consisting of only 6 Datalog rules, but with challenging join characteristics. For these joins, DDLog is better optimized, and is able to out-perform Soufflé for these incremental workloads. For small updates consisting of 10 EDB tuples, an incremental update takes between 0.1 and 0.2 seconds, providing far superior performance compared to a non-incremental engine. Even for medium sized updates consisting of 10,000 EDB tuples, the performance of DDLog's incremental update is generally faster than non-incremental Soufflé. Only when we consider larger updates of 40,000, 70,000, and 100,000 EDB tuples, or 4%, 7%, and 10% respectively, does the performance of incremental evaluation slow down considerably compared to non-incremental Soufflé. The impact of these larger updates on the IDB is up to 53M tuples, which is almost double the original IDB size. This impact indicates that not only are most of the IDB tuples affected, but they are even affected in multiple iterations. Given this large impact, it is no surprise that the runtime for such an incremental update is slower than simply re-computing the result from scratch, and in fact, DDLog performs well on this benchmark compared to non-incremental Soufflé. For Galen, the Soufflé incremental strategies do not perform as well. Soufflé-counting is generally an order of magnitude slower for updates than DDLog, as a result of unfavorable join orderings. Update fares even worse, timing out for the larger updates above 10,000 EDB tuples. This is a result of these updates impacting tuples across multiple iterations, which the sparsification of the elastic strategy does not handle well.

These results indicate that state-of-the-art single strategy incremental evaluation algorithms perform well on small impact updates. However, they may be out-performed by a standard non-incremental Datalog engine for more complex applications or high impact changes. Overall, **we demonstrate Claim I by highlighting the unpredictability and tendency for degraded performance of single strategy evaluations on large impact updates compared to non-incremental Soufflé.**

5.2 Elastic Incremental Evaluation

In this section, we evaluate the performance of our elastic incremental evaluation strategy, that is **we evaluate the combination of the Update algorithm with Bootstrap.** We use an empirically determined switching parameter of 20% to determine when to use the Update, and when to switch to Bootstrap. That is, if the update time is more than **20**% of the previous bootstrap time, we restart using Bootstrap.

For this experiment, we use example workloads for incremental evaluation, which consists of 13 epochs. The first epoch is the initial evaluation, then the following 6 epochs are small updates, with alternating deletion and insertions. These are followed by one large update in epoch 7, then followed by another 4 small updates, with a large update as the final epoch. We note that these patterns may appear in all three of these benchmarks. For Doop, there is a common pattern of software updates consisting of a large refactor, followed by a number of smaller commits addressing minor comments. For CRDT, which is an application commonly used for collaborative online text editing, a large update may result from a large portion of text being moved around, while a smaller update may result from smaller additions or deletions from the text. For Galen, which is a medical ontology application associated with patient diagnosis, a large update may result from a medical test result being updated, while a smaller update may result from a minor symptom change.

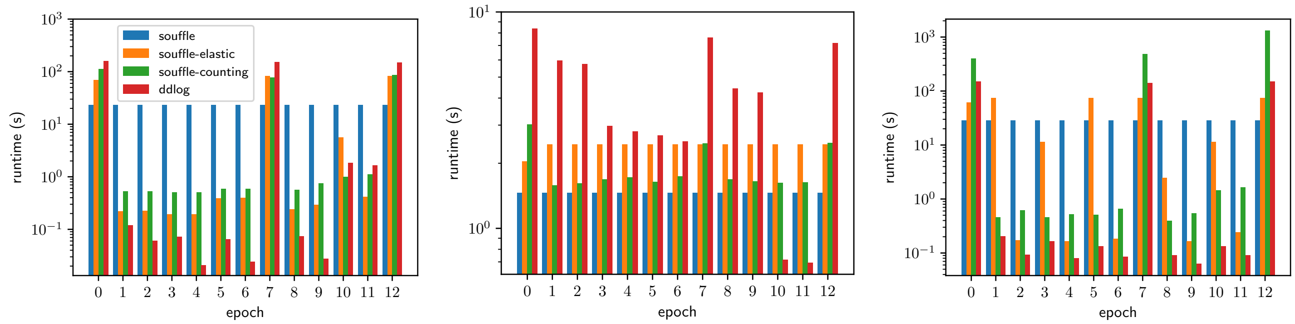
For Doop, in Fig. 6a, all of the incremental evaluation strategies are able to effectively incrementalize for the small updates. However, the main differences across the full workload are a result of the bootstrap strategy, with both the initial evaluation and the large updates being faster or on-par with the state-of-the-art counting strategy. As a result, the elastic incremental strategy is able to complete this workload in 245 seconds, compared to 284 seconds for Soufflé-counting, and 467 seconds for DDLog. In comparison, non-incremental Soufflé, which evaluates each epoch from scratch, achieves 304 seconds for this workload. This use case demonstrates that **an elastic incremental evaluation is effective for the complex Doop benchmark. We demonstrate an overall amortized net gain compared to non-incremental Soufflé as well as single strategy evaluations.**

For CRDT, in Fig. 6b, none of the incremental evaluation strategies are effective, even for the small updates. Here, the elastic strategy hits the 20% heuristic threshold for all updates, despite the update strategy actually being slightly faster than bootstrap, if it were allowed to run to completion. For this workload, non-incremental Soufflé completes all epochs in 19 seconds, followed by 25 seconds for the Soufflé-counting, 31 seconds for Soufflé-elastic, and 56 seconds for DDLog. **For this particular application, we conclude that incremental evaluation in general is ineffective.**

For Galen, in Fig. 6c, the incremental evaluation strategies were able to perform reasonably well. For epochs 1 and 5, the elastic update strategy reached the 20% heuristic threshold, thus triggering a bootstrap. If this threshold was not in place, the elastic update would have been faster for these small updates. Despite this, Soufflé-elastic is still highly competitive compared to the other incremental evaluation strategies, being able to finish the workload in 384 seconds, compared to 445 seconds for DDLog. Soufflé-counting was ineffective for the large updates for Galen, In comparison, non-incremental Soufflé required 370 seconds for this workload. **The results demonstrate that our elastic evaluation is competitive for the Galen use case**.

Overall, **the experimental evaluation has validated Claim II by showing a large performance improvement compared to single strategy approaches.** The limited overhead of our Bootstrap evaluation makes up for any cost induced by the Update evaluation. We believe with improved heuristics and tuning, this improvement can be further maximized.

Along with runtime, another aspect of performance is memory usage. For example in large program analysis use cases memory has been shown to be a limiting factor [20]. Table 2 shows the minimum, average and maximum memory usage across all the update sets for each benchmark. These results show that non-incremental Soufflé uses the least memory by far, since it does not need to keep the extra state that incremental evaluation requires. Among the incremental engines, Soufflé-elastic performs best, since it only keeps the counts for one iteration for each tuple. On the other hand, the counting algorithm, both in Soufflé and in DDLog, require to keep the count of each tuple for *every* iteration it is generated in, thus using extra memory to maintain this extra state.



(a) Doop (b) CRDT (c) Galen

Figure 6: Runtimes for an elastic workload. For each benchmark, the first epoch is an initial evaluation, followed by 6 epochs of small updates, then one large update, then 4 epochs of small updates, then one large update.

Table 2: Memory usage for each engine, showing the minimum, average, and maximum memory usage across all of the update sets

Bench.	Engine	Min (MB)	Avg (MB)	Max (MB)
Doop	Soufflé	1,759	1,762	1,764
	Soufflé-elastic	7,473	7,492	7,505
	Soufflé-counting	9,106	9,449	11,387
	DDLog	17,381	23,352	27,851
CRDT	Soufflé	42	42	42
	Soufflé-elastic	335	346	352
	Soufflé-counting	328	337	344
	DDLog	786	829	858
Galen	Soufflé	901	931	960
	Soufflé-elastic	5,641	5,672	5,698
	Soufflé-counting	14,588	17,974	21,034
	DDLog	15,333	20,862	26,461

6 RELATED WORK

There is a large corpus of incremental algorithms in related fields including Databases [5], Logic-programming [33], Compilers [31], Model-checking [36] and SAT solving [22]. In this section we focus exclusively on Datalog evaluation. The main body of work in incremental Datalog evaluation is related to the Delete-Rederive (DRed) algorithm [13]. The main weakness of this approach concerns *over-deletion*. This is resolved by re-deriving tuples that are over-deleted. The Counting algorithm presented in [13] is applicable only for non-recursive Datalog programs. For this approach, each tuple is associated with a *count* of the number of different derivations that exist for that tuple. When removing or inserting a new tuple, that count is decremented or incremented respectively, and a tuple may be removed if the count reaches 0. However, with recursive Datalog programs, deleting tuples may cause the recursive decrement of the count, thus again leading to over-deletion. More recent developments include the Backward/Forward algorithm [26] and DRed^c [15], which are both optimizations of the DRed algorithm. The aim of these approaches is to reduce the approximation induced by the over-deletion step. Backward/Forward uses a form of *backwards evaluation* to eagerly check if over-deleted tuples still have a proof from the remaining input, while DRed^c maintains separate recursive and non-recursive counters to track the number of derivations of each tuple. While these approaches indeed reduce the over-deletion of DRed, they are still approximations and worst-case scenarios may exhibit large run-time overheads. Techniques like [12] use provenance information in the form of Boolean formulae for each tuple to determine if a deleted tuple has proof support. The Differential Dataflow (DDF) system [23] implements incremental evaluation for Dataflow programming. The approach is similar to the counting algorithm, with each tuple being associated with a count for the number of derivations for that tuple. However, DDF permits recursive programs by storing a count *per iteration* of recursive evaluation. Its advantage is that it computes a precise result for an incremental update. However, the setting of Dataflow programming is different from Datalog, and more similar to stream programming, where programs tend to be less complex with smaller updates. Differential Datalog [32] is a Datalog engine built on top of DDF. Other systems, such as RDFox [25], and Ladder [39] implement variations of the DDF algorithm, specialized to their respective domains. In comparison with existing approaches, our elastic evaluation is unique in that it has two evaluation phases, recognising the importance of specializing the Bootstrap phase to initialize the computation. Our algorithms form a sparsified variation of the counting algorithm, allowing for the efficient Bootstrap phase, and lowering the space overhead per tuple.

7 CONCLUSION

In this paper, we have demonstrated the pitfalls of existing incremental evaluation algorithms for use cases with varying sized updates. We have proposed the use of an elastic approach for incremental evaluation. We switch between a low overhead Bootstrap strategy that targets large impact updates and an Update strategy that targets low impact updates. We propose a simple heuristic for switching between the two strategies. Using this setup we have shown that the elastic approach is effective in use cases where single strategy incremental evaluation struggles to perform adequately compared to regular Datalog evaluation.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley Publishing Company.
[2] Nicholas Allen, Bernhard Scholz, and Padmanabhan Krishnan. 2015. *Staged Points-to Analysis for Large Code Bases.* Springer Berlin Heidelberg, 131–150. https://doi.org/10.1007/978-3-662-46663-6_7

[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[4] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark A. Stalzer, Preethi Srinivasan, Pavle Subotic, Carsten Varming, and Blake Whaley. 2019. Reachability Analysis for AWS-Based Networks. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. 231–241.

[5] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. 2004. Incremental Validation of XML Documents. *ACM Trans. Database Syst.* 29, 4 (Dec. 2004), 710–751. https://doi.org/10.1145/1042046.1042050

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. *SIGPLAN Not.* 44, 10 (2009), 243–262. https://doi.org/10.1145/1639949.1640108

[7] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2020. Code-Level Model Checking in the Software Development Workflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/3377813.3381347

[8] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70.

[9] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *Proceedings of the 41th International Conference on Software Engineering, ICSE 2019*. ACM, Montreal, QC, Canada, (to appear).

[10] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1176–1186. https://doi.org/10.1109/ICSE.2019.00120

[11] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. In *SPLASH 2018 OOPSLA*.

[12] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. 2007. Update Exchange with Mappings and Provenance. In *In Very Large Data Bases (VLDB*. 675–686.

[13] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

[14] Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura. 2011. μZ– An Efficient Engine for Fixed Points with Constraints. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 457–462.

[15] Pan Hu, Boris Motik, and Ian Horrocks. 2018. Optimised maintenance of datalog materialisations. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[16] Pan Hu, Boris Motik, and Ian Horrocks. 2019. Modular Materialisation of Datalog Programs. (2019).

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

[18] Muhammad Imran, Gábor E Gévay, and Volker Markl. 2020. Distributed Graph Analytics with Datalog Queries in Flink. In *Software Foundations for Data Interoperability and Large Scale Graph Data Analytics*. Springer, 70–83.

[19] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. *Proceedings of Computer Aided Verification* 28 (2016), 422–430.

[20] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*. Springer, 422–430.

[21] Grigoris Karvounarakis, Todd J. Green, Zachary G. Ives, and Val Tannen. 2013. Collaborative Data Sharing via Update Exchange and Provenance. *ACM Trans. Database Syst.* 38, 3, Article 19 (Sept. 2013), 42 pages.

[22] Yousef Kilani, Mohammad Bsoul, Ayoub Alsarhan, and Ahmad Al-Khasawneh. 2013. A Survey of the Satisfiability-Problems Solving Algorithms. *Int. J. Adv. Intell. Paradigms* 5, 3 (Sept. 2013), 233–256. https://doi.org/10.1504/IJAIP.2013.056447

[23] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow.. In *CIDR*.

[24] Hongyuan Mei, Guanghui Qin, Minjie Xu, and Jason Eisner. 2020. Neural Datalog Through Time: Informed Temporal Modeling via Logical Specification. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 6808–6819.

[25] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2019. Maintenance of datalog materialisations revisited. *Artificial Intelligence* 269 (2019), 76–136.

[26] Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. 2015. Incremental update of datalog materialisation: the backward/forward algorithm. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

[27] Derek Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (proceedings of the 24th acm symposium on operating systems principles (sosp) ed.). ACM.

[28] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* 65, 3 (2018), 1–40.

[29] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. 2005. MulVAL: A Logic-based Network Security Analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Baltimore, MD) *(SSYM'05)*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=1251398.1251406

[30] Alan L Rector, Jeremy E Rogers, and Pam Pole. 1996. The GALEN high level ontology. In *Medical Informatics Europe'96*. IOS Press, 174–178.

[31] Thomas Reps. 1982. Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 169–176. https://doi.org/10.1145/582153.582172

[32] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. *Datalog* 2 (2019), 4–5.

[33] Diptikalyan Saha and C. R. Ramakrishnan. 2006. Incremental Evaluation of Tabled Prolog: Beyond Pure Logic Programs. In *Practical Aspects of Declarative Languages*, Pascal Van Hentenryck (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 215–229.

[34] Jiwon Seo, Stephen Guo, and Monica S Lam. 2013. Socialite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 278–289.

[35] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*. 1135–1149.

[36] Oleg V. Sokolsky and Scott A. Smolka. 1994. Incremental Model Checking in the Modal Mu-Calculus. In *IN CAV, VOLUME 818 OF LNCS*. Springer-Verlag, 351–363.

[37] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 59–76. https://doi.org/10.1145/1094811.1094817

[38] Pavle Subotić. 2020. Concise Explanations in Static Analysis Driven Code Reviews. (2020). https://www.youtube.com/watch?v=FPCZ2TIxrpg&t=8888s Infer Practitioners 2020.

[39] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental Whole-Program Analysis in Datalog with Lattices. (2021).

[40] Todd L Veldhuizen. 2012. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481* (2012).

[41] David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 2 (2020), 1–35.

[42] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. 2010. Efficient Querying and Maintenance of Network Provenance at Internet-Scale. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), 615–626.

A PROOFS FOR THEOREMS

A.1 Proof of Lemma 3.2

Proof. This proof is by induction over the iterations k. For $k = 1$, the semi-naïve algorithm takes Δ_0, while the bootstrap algorithm takes $\mathrm{Supp}(\mathcal{B}_0^{\#})$. Both of these sets are defined to be E, so are equal. For the induction hypothesis, assume that all I_{i-1} of bootstrap equal I_{i-1} of semi-naïve. Then, $\mathrm{Supp}(\mathcal{B}_i^{\#}) = \Delta_i$ by Lemma 3.1. Therefore, adding $\mathrm{Supp}(B_i^{\#})$ or Δ_i to the union results in the same set. □

A.2 Proof of Lemma 3.3

Proof. This proof is by induction over the iterations. For $k = 0$, I_0^- is defined as E^- and I_0^+ is defined as E^+, so properties (1) and (2) hold by definition.

The induction hypothesis is that for iteration $k-1$, we have $I_{k-1}^- \subseteq I_{k-1}^o$, $I_{k-1}^- \cap I_{k-1} = \emptyset$, $I_{k-1}^+ \cap I_{k-1}^o = \emptyset$, and $I_{k-1}^+ \subseteq I_{k-1}$.

For property (1), we show that $I_k^- \subseteq I_k^o$. Consider line 11 of Algorithm 3, where I_k^- takes the value of $(I_{k-1}^- \setminus N_k) \cup (N_k^o \setminus I_k)$. In the first part of this union, $I_{k-1}^- \subseteq I_{k-1}^o$ by the induction hypothesis. Therefore, also $I_{k-1}^- \subseteq I_k^o$, since I_{k-1}^o is monotonically growing. In the second part of the union, $N_k^o \subseteq I_k^o$ by definition of I_k^o. Therefore, $I_k^- \subseteq I_k^o$.

To show that $I_k^- \cap I_k = \emptyset$, consider the same line. In the first part of the union, $I_{k-1}^- \cap I_{k-1} = \emptyset$ by the induction hypothesis. We then exclude N_k, and since $I_k = I_{k-1} \cup N_k$ by definition, then $(I_{k-1}^- \setminus N_k) \cap I_k = \emptyset$. In the second part of the union, we exclude I_k. Therefore, $I_k^- \cap I_k = \emptyset$.

Property (2) holds by similar arguments on line 12. □

A.3 Proof of Lemma 3.4

Proof. To prove this, we first show that $I_k^o \setminus I_k = I_k^-$ by showing both directions of inclusion. The reverse direction, i.e., that $I_k^- \subseteq I_k^o \setminus I_k$ is a direct corollary of Lemma 3.3, that $I_k^- \subseteq I_k^o$ and $I_k^- \cap I_k = \emptyset$. For the forward direction, consider some tuple $t \in I_k^o \setminus I_k$. Then, t must be in some $N_i^o \setminus I_k$ for some $i \le k$. Since $I_i \subseteq I_k$, t is also in $N_i^o \setminus I_i$. Therefore, $t \in I_i^-$. Also, t cannot be removed from I^- in a later iteration, since $t \notin I_k$, and therefore, $t \in I_k^-$.

We have shown both directions of inclusion, and therefore, $I_k^o \setminus I_k = I_k^-$. By a similar argument, $I_k \setminus I_k^o = I_k^+$. From these equalities, we have:

$$\begin{aligned} I_k^o \setminus I_k^- \cup I_k^+ &= I_k^o \setminus (I_k^o \setminus I_k) \cup (I_k \setminus I_k^o) \\ &= (I_k^o \cap I_k) \cup (I_k \setminus I_k^o) \\ &= I_k \end{aligned}$$

□

A.4 Proof of Theorem 3.5

Proof. For this proof, we mainly consider the underlying sets of derivations rather than the counting multisets, since the counting multisets do not distinguish between different derivations. We introduce some new notation to convert between derivations and tuples: $\phi((t :- t_1, \ldots, t_n)) := t$ takes the head tuple of a derivation.

The proof is an induction over the iterations. The initial step, where $k = 0$, is true since both $\mathcal{B}_0^\#$ and $\mathcal{N}_0^\#$ take on the value of $E \setminus E^- \cup E^+$, where every tuple has a count of 1.

The induction hypothesis is that for all $0 \le i < k$, we have $\mathcal{B}_i^\# = \mathcal{N}_i^\#$. We consider each of the four terms in lines 9 and 10. We first need to show that the sets of derivations computed by these lines are disjoint, so that the algorithm does not double count.

- For the deletion term (we label it (1)), we have the derivations $\{d \in \Pi^D[I_{k-1}^o \mid N_{k-1}^o \mid I_{k-1}^-] \mid \phi(d) \notin I_{k-1}^o\}$.
- For the insertion term (labelled (2)), we have derivations $\{d \in \Pi^D[I_{k-1} \mid N_{k-1} \mid I_{k-1}^+] \mid \phi(d) \notin I_{k-1}\}$. Since $I_{k-1}^+ \cap I_{k-1}^o = \emptyset$ (from Corollary 3.4), then $(2) \cap (1) = \emptyset$, since (1) takes derivations only from I_{k-1}^o.
- For the re-discovery term (labelled (3)), we have derivations $\{d \in \Pi^D[I_{k-1}^o \cap I_{k-1} \mid N_{k-1}] \mid \phi(d) \in I_{k-1}^- \text{ and } \phi(d) \notin I_{k-1}\}$. Since this takes derivations from I_{k-1}^o, and $I_{k-1}^o \cap I_{k-1}^+ = \emptyset$, then $(3) \cap (2) = \emptyset$. Also, since $I_{k-1}^- \subseteq I_{k-1}^o$ (from Corollary 3.4), we have $(3) \cap (1) = \emptyset$, since (1) excludes tuples from I_{k-1}^o.
- For the sparsification term (labelled (4)), we have $\mathcal{N}_k^\# \ominus (\mathcal{N}_k^\# \cap I_{k-1}^+)$. However, note that this term is processed after the three other terms. Therefore, it naturally excludes (1), and so $(4) \cap (1) = \emptyset$. Moreover, we have $I_{k-1}^+ \subseteq I_{k-1}$ (from Corollary 3.4), and so $(4) \cap (3) = \emptyset$ and $(4) \cap (2) = \emptyset$, since both (3) and (2) exclude I_{k-1}.

Since all 4 terms produce disjoint derivations, the algorithm does not double count when adding or removing any derivations. Next, we need to prove that for any derivation in $\mathcal{N}_k^{Do}$ and not in $\mathcal{N}_k^D$, it is removed by one of the four terms, and vice versa.

Consider a derivation $d \in \mathcal{N}_k^{Do} \setminus \mathcal{N}_k^D$. By definition, $d \in \{d \in \Pi^D[I_{k-1}^o \mid N_{k-1}^o] \mid \phi(d) \notin I_{k-1}^o\} \setminus \{d \in \Pi^D[I_{k-1} \mid N_{k-1}] \mid \phi(d) \notin I_{k-1}\}$. Then, there are two cases. The first case is that $\phi(d) \in I_{k-1}$. In this case, also $\phi(d) \notin I_{k-1}^o$, by our assumption, and so $\phi(d) \in I_{k-1}^+$ (by Corollary 3.4). Therefore, d would be removed by the sparsification term which removes all tuples that are in I_{k-1}^+. The second case is that $d \notin \Pi^D[I_{k-1} \mid N_{k-1}]$. In this case, one of the body tuples of d is in $I_{k-1}^o \setminus I_{k-1}$ (or in $N_{k-1}^o \setminus N_{k-1}$, which implies also that it is in $I_{k-1}^o \setminus I_{k-1}$), which equals I_{k-1}^- (by Corollary 3.4). Therefore, $d \in \Pi^D[I_{k-1}^o \mid N_{k-1}^o \mid I_{k-1}^-]$, and since $\phi(d) \notin I_{k-1}^o$ by assumption, it would be removed by the deletion term.

Now, for the opposite case, consider a derivation $d \in \mathcal{N}_k^D \setminus \mathcal{N}_k^{Do}$. We want to show that this derivation is inserted by one of the four terms. By definition, $d \in \{d \in (\Pi^D[I_{k-1} \mid N_{k-1}] \mid \phi(d) \notin I_{k-1}\} \setminus \{d \in \Pi^D[I_{k-1}^o \mid N_{k-1}^o] \mid \phi(d) \notin I_{k-1}^o\}$. Like the deletion case, there are two cases. The first is that at least one of the body tuples of d are in $I_{k-1} \setminus I_{k-1}^o$. Then, this tuple is in I_{k-1}^+, and therefore, $d \in \Pi^D[I_{k-1} \mid N_{k-1} \mid I_{k-1}^+]$. Since $\phi(d) \notin I_{k-1}$ by assumption, then d will be inserted by the insertion term. The second case is if $\phi(d) \in I_{k-1}^o$. Then, since $\phi(d) \notin I_{k-1}$, $\phi(d) \in I_{k-1}^o \setminus I_{k-1} = I_{k-1}^-$. If the first case doesn't hold, we know that all of the body tuples are not in $I_{k-1} \setminus I_{k-1}^o$, and therefore, they must all be in I_{k-1}^o. Therefore, $d \in \Pi^D[I_{k-1}^o \cap I_{k-1} \mid N_{k-1}]$. Since $\phi(d) \notin I_{k-1}$ by assumption, d would be inserted by the re-discovery term. □

B ADDITIONAL EXPERIMENTAL DATA

Benchmark	Engine	Updates	Epoch 0 (sec)	Epoch 1 (-) (sec)	Epoch 2 (+) (sec)	Memory (MB)
doop	Soufflé-elastic(update)	10	64.53	min 0.20 max 5.62	min 0.20 max 0.42	7486.5
		100	66.22	min 4.63 max 13624.27	min 0.38 max 2.98	7497.4
		400	-	-	-	-
		700	-	-	-	-
		1000	-	-	-	-
	Soufflé-counting	10	113.64	min 0.50 max 1.00	min 0.50 max 1.11	9116.2
		100	116.00	min 0.78 max 9.05	min 0.77 max 9.49	9131.4
		400	114.00	min 10.20 max 632.30	min 10.90 max 695.00	9540.6
		700	113.81	min 21.26 max 129.37	min 22.60 max 147.66	9539.8
		1000	116.92	min 14.40 max 428.29	min 16.70 max 500.99	9917.3
	DDLog	10	164.53	min 0.07 max 1.85	min 0.02 max 1.67	17495.4
		100	167.83	min 2.57 max 102.51	min 2.30 max 110.29	20682.4
		400	169.45	min 90.65 max 145.37	min 101.38 max 146.70	25002.1
		700	164.86	min 115.14 max 205.84	min 129.22 max 185.37	26350.2
		1000	167.35	min 149.71 max 232.32	min 148.30 max 205.08	27230.5
crdt	Soufflé-elastic(update)	10	1.99	min 1.64 max 1.74	min 1.45 max 1.51	338.7
		40	1.96	min 1.93 max 2.10	min 1.67 max 1.78	345.7
		70	1.96	min 2.32 max 2.68	min 2.01 max 2.58	349.3
		100	2.00	min 2.51 max 2.90	min 2.05 max 2.65	351.8
	Soufflé-counting	10	2.98	min 1.58 max 1.69	min 1.62 max 1.74	331.4
		40	3.02	min 1.83 max 1.99	min 1.80 max 2.00	336.6
		70	3.10	min 2.15 max 2.32	min 2.11 max 2.34	338.6
		100	2.92	min 2.21 max 2.47	min 2.19 max 2.49	341.6
	DDLog	10	8.52	min 0.71 max 5.97	min 0.69 max 5.74	804.5
		40	8.61	min 4.69 max 6.56	min 4.47 max 6.72	825.4
		70	8.59	min 7.02 max 7.36	min 6.78 max 7.33	833.1
		100	8.50	min 7.29 max 7.62	min 7.08 max 7.28	851.8
galen	Soufflé-elastic(update)	10	60.69	min 2.48 max 20.29	min 0.17 max 0.24	5666.9
		10000	59.64	min 37.16 max 94.22	min 0.29 max 0.62	5676.2
		40000	-	-	-	-
		70000	-	-	-	-
		100000	-	-	-	-
	Soufflé-counting	10	415.63	min 0.40 max 1.44	min 0.53 max 1.64	14595.4
		10000	415.05	min 147.14 max 203.44	min 146.88 max 239.68	15799.4
		40000	422.35	min 568.20 max 902.38	min 612.21 max 977.07	18776.6
		70000	411.63	min 996.40 max 1130.46	min 1025.11 max 1216.50	20027.0
		100000	414.45	min 1131.57 max 1602.21	min 21.03 max 1377.78	20671.6
	DDLog	10	152.09	min 0.09 max 0.20	min 0.06 max 0.09	15602.4
		10000	154.10	min 19.69 max 28.02	min 20.78 max 27.14	16771.1
		40000	152.70	min 74.72 max 96.72	min 77.23 max 99.10	22027.8
		70000	154.10	min 107.76 max 130.03	min 110.19 max 132.05	24195.4
		100000	157.96	min 135.58 max 160.32	min 137.96 max 168.43	25712.8

Table 3: Running times and memory usage for Dynamic Datalog Benchmarks, each min and max value denotes the minimum and maximum runtimes over 5 different datasets of the corresponding update size, - denotes timeout, and variations in Epoch 0 runtime are due to re-runs of the experiment