Incremental Inference for Probabilistic Datalog

Xuyang Li[1], Weiyi Chen[1], Isil Dillig[2], and Jingbo Wang[1(✉)]

[1] Purdue University, West Lafayette, IN, USA
{li5274,chen5332,wang6203}@purdue.edu
[2] The University of Texas at Austin, Austin, TX, USA
isil@cs.utexas.edu

Abstract. Several extensions of Datalog perform probabilistic inference by allowing users to annotate input facts and rules with probabilities. While extremely useful in many domains (e.g., quantitative program analysis), existing systems typically do not support *incremental inference*, meaning that even small changes trigger costly recomputation from scratch. This paper presents `PINQ`[3], the first incremental solving framework for probabilistic Datalog. Given a previously solved program and a set of changes, `PINQ` updates query probabilities by reusing the old derivation graphs and compiled decision diagrams. The key idea is to translate structural changes into parametric updates whenever sound to avoid redundant recomputation. Our framework combines an incremental derivation graph construction algorithm with an adaptive BDD construction technique that safely reuses existing BDDs via weight calibration whenever possible. Experimentally, `PINQ` achieves an average speedup of 17× over recomputation from scratch on a representative set of program analysis benchmarks.

1 Introduction

Datalog is widely used as a foundation for program analysis due to its declarative, solver-backed mechanism for expressing fixed-point computations. Over the past two decades, Datalog encodings have powered analyses ranging from alias [2,19,26] and taint analysis [37,40] to data-race detection [20,22,39], security policy enforcement [16], and side-channel vulnerability detection [36,35]. Recently, this foundation has been extended with *probabilistic* variants of Datalog, which associate probabilities with rules [7,8,34,17] to enable *quantitative* program analyses that go beyond binary alarms [20,36,35].

In practice, however, the programs subject to analysis are rarely static: they evolve through frequent, small edits as developers fix bugs or add features. In this context, re-running a Datalog analysis from scratch after each change is wasteful because most intermediate results remain valid across versions. This observation has motivated a substantial body of work on *incremental* Datalog, which reuses prior fixed-point results to obtain significant empirical speedups [9,32,30,31,41].

[3] Stands for Probabilistic INcremental Querying

However, existing incremental techniques target traditional (deterministic) Datalog as opposed to its probabilistic variants. The latter setting poses significant challenges for incremental solving because a query result's probability is induced by the *full space* of derivations, rather than just the derived facts. Thus, in the probabilistic setting, even a small edit can change the shared derivation structure in ways that propagate widely, making it challenging to update probabilities without full recomputation.

This paper aims to address this gap by presenting the first end-to-end incremental probabilistic Datalog engine. From a technical standpoint, this involves two core contributions. First, we present an incremental algorithm for maintaining a *complete derivation graph* under edits. The key idea is to maintain a *delta derivation graph* that records which derivations are introduced or invalidated by an edit and propagates these updates through derivation dependencies. Because each *individual* derivation impacts the computed probabilities, incremental maintenance must preserve exact provenance in the probabilistic setting. Accordingly, our method tracks not only which derived atoms remain valid, but also incrementally maintains *the full set of derivations* supporting each fact.

The second technical contribution of this paper is an incremental probability computation method that avoids recompiling the Boolean representation used for inference after each edit. Traditionally, query probabilities are computed by compiling the space of derivation alternatives into a *decision diagram* (e.g., a BDD or SDD) and then evaluating it via weighted model counting under a weight map. Thus, a naive way to perform incremental inference is to regenerate the compiled knowledge representation of each fact whose derivation is affected. However, since BDD construction is often a key bottleneck, reconstructing the BDD for each affected artifact can be prohibitive. Our key insight is to reuse as much of the previous compilation as possible by separating *structural* from *parametric* changes. In particular, the effect of many edits can be captured by recalibrating some of the weights while reusing the prior decision-diagram structure. Our method identifies when such reuse is sound and computes the corresponding weight updates needed to make it correct.

We implemented the proposed method in a tool called `PINQ` and evaluated it on 285 probabilistic Datalog benchmarks derived from side-channel vulnerability detection [34]. Across all benchmarks and update regimes, `PINQ` reduces end-to-end update latency by over an order of magnitude, achieving an average speedup of 17× and up to 55× relative to full recomputation.

To summarize, this paper makes the following key contributions:

- We present an algorithm for maintaining a *complete* derivation graph for probabilistic Datalog under edits, with soundness and completeness guarantees relative to the updated program.
- We present an incremental probability computation method that reuses compiled Boolean knowledge representations as much as possible by re-calibrating the weights used for weighted model counting.

– We show that our method achieves over an order-of-magnitude reduction in end-to-end latency (average 17×) in side channel detection tasks spanning 19 unique programs and 15 different update regimes to these base programs.

2 Background on Probabilistic Datalog

This section introduces probabilistic Datalog programs and their semantics.

Definition 1 (Probabilistic Datalog). *A* probabilistic Datalog program *is a pair* $\langle \mathcal{R}, \mathcal{Q} \rangle$ *where* $\mathcal{R}$ *is a finite set of* probabilistic rules *and* $\mathcal{Q}$ *is a finite set of (ground) query atoms. Each rule* $r \in \mathcal{R}$ *has the form*

$$p \;::\; H(\mathbf{x}) \coloneq \odot_1 B_1(\mathbf{y}_1), \ldots, \odot_n B_n(\mathbf{y}_n),$$

where $\odot_i \in \{\,\cdot\,, \neg\}$ *indicates whether the* i*th body atom appears positively or negated, and* $p \in (0, 1]$ *is the probability annotation of* r*. A rule with an empty body is a probabilistic input fact.*

We assume that probabilistic input facts and rule applications are mutually independent, with marginals given by their annotations. As standard [21], we also assume that program P is *stratified*, meaning that there exists a mapping σ from predicate symbols to strata such that for every rule $H \coloneq \odot_1 B_1, \ldots, \odot_n B_n$, we have $\sigma(H) \geq \sigma(B_i)$ for all i, and $\sigma(H) > \sigma(B_i)$ whenever $\odot_i = \neg$.

Semantics. Datalog solvers generate a finite set of ground rule instances by grounding rules $\mathcal{R}$ over the Herbrand universe [18]. Each ground instance e corresponds to an independent Boolean random event indicating whether that instance "fires". Let X denote the set of these event variables, with one variable X_e per ground instance e, and let $W : X \to (0, 1]$ map X_e to the probability annotation of the underlying rule. A *world* is an assignment $\omega : X \to \{0, 1\}$ selecting which ground rule instances occur. Given ω, let $\mathsf{lfp}(\omega)$ denote the least Herbrand model of the deterministic Datalog program induced by ω. A query q succeeds in ω, denoted $\omega \models q$, iff $q \in \mathsf{lfp}(\omega)$. Under the assumption that the event variables are mutually independent, ω has probability:

$$\Pr[\omega] = \prod_{X_e \in X \,:\, \omega(X_e) = 1} W(X_e) \cdot \prod_{X_e \in X \,:\, \omega(X_e) = 0} (1 - W(X_e)).$$

Then, the *success probability* of query atom q is given by $\mathsf{Pr}[q] = \sum_{\omega \models q} \Pr[\omega]$.

Solving view. Figure 1 illustrates how ProbLog-style systems [7] compute $\mathsf{Pr}[q]$: they first construct a derivation graph representing the space of grounded rule applications, then compile derivability into a Boolean representation (e.g., BDD), and finally evaluate a weighted count using that representation. We next introduce the artifacts produced by this process.

Definition 2 (Derivation graph). *A* derivation graph *is a directed, edge-labeled hypergraph* $\mathcal{G} = (N, E, \mathcal{L}, W)$ *where:*

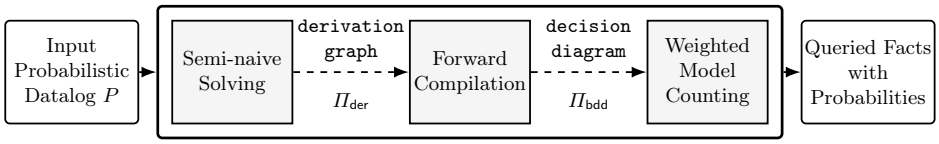

Fig. 1. Overview of probabilistic Datalog solving via derivation graph construction, forward compilation, and weighted model counting.

- N *is a finite set of nodes, each corresponding to a ground atom.*
- E *is a finite set of hyperedges. Each* $e \in E$ *has the form* $(B^+, B^-) \to H$*, where* B^+ *and* B^- *are finite sets of ground atoms (positive and negative body atoms respectively) and* H *is a ground atom (the head).*
- $\mathcal{L} : E \to X$ *assigns each hyperedge* $e \in E$ *the Boolean event variable* X_e*.*
- $W : X \to (0, 1]$ *assigns each* X_e *a probability.*

In other words, the derivation graph represents the space of all possible derivations for each ground atom. For a node $n \in N$, each incoming hyperedge $(B^+, B^-) \to n$ corresponds to one grounded rule instance that can derive n, where B^+ (resp. B^-) is the set of ground atoms that appear positively (resp. negated) in the body of that instance. In the remainder of this paper we assume and maintain a *positive-support* derivation graph: E contains exactly those grounded instances whose positive premises hold, i.e., we include $(B^+, B^-) \to n$ whenever all atoms in B^+ hold, independent of whether the atoms in B^- hold. This choice makes the derivation graph monotone in the positive dependency structure and simplifies incremental maintenance.

Forward compilation. Given a derivation graph $\mathcal{G}$, *forward compilation* translates each node $n \in N$ to an *objective* formula $\phi(n)$ over the event variables X such that, for every world $\omega : X \to \{0, 1\}$, $\phi(n)$ evaluates to true under ω iff $n \in \mathsf{lfp}(\omega)$. Intuitively, each incoming hyperedge $e = ((B^+, B^-) \to n)$ represents one grounded rule instance that may derive n. In a world ω, this instance can justify n exactly when its event variable is active ($\omega(X_e) = 1$), all positive premises hold, and all negated premises fail. Accordingly, the edge e contributes the disjunct

$$X_e \wedge \bigwedge_{a \in B^+} \phi(a) \wedge \bigwedge_{b \in B^-} \neg\phi(b)$$

and $\phi(n)$ is the disjunction over all such incoming hyperedges. When $\mathcal{G}$ contains cycles, forward compilation produces a well-defined formula by eliminating cyclic dependencies so that $\phi(n)$ depends only on the independent events in X [8]. Given the objective formula $\phi(n_q)$ for a query atom q, we compute $\mathsf{Pr}[q]$ as the weighted model count of $\phi(n_q)$ under the weights induced by W.

Definition 3 (Weighted model counting). *Given a weight mapping* $W : X \to (0, 1]$ *and a Boolean formula* φ *over* X*, the* weighted model count *of* φ *is*

$$\mathsf{WMC}(\varphi, W) := \sum_{\omega \models \varphi} \Big(\prod_{X_e \in X :\, \omega(X_e) = 1} W(X_e) \Big) \Big(\prod_{X_e \in X :\, \omega(X_e) = 0} (1 - W(X_e)) \Big).$$

Algorithm 1: Top-level Incremental Solving Procedure

Input: Base program P with $\texttt{Solve}(P) = \langle \Psi, \Pi \rangle$, change set Δ
Output: Updated query probability mapping Ψ' and updated provenance Π'

```
1 begin
2     (Π_der, Π_bdd) ← Π
3     Δ_der ← IncDeriv(Π_der, Δ, P) ;                          // Section 4
4     (Δ_bdd, W', Δ_fact) ← IncBddGen(Π_der, Δ_der, Π_bdd) ;   // Section 5
5     Δ_Ψ ← IncWMC(Δ_fact, Π_bdd, Δ_bdd, W')
6     return (Ψ ⊎ Δ_Ψ, (Π_der ⊎ Δ_der, Π_bdd ⊎ Δ_bdd))
```

Thus, for a query atom $q \in \mathcal{Q}$ with corresponding node $n_q \in N$, we can compute $\mathsf{Pr}[q]$ as $\mathsf{WMC}(\phi(n_q), W)$.

BDD representation. ProbLog-style systems [7] evaluate $\mathsf{WMC}(\phi(n_q), W)$ via knowledge compilation, and this paper uses binary decision diagrams (BDDs) [3] for this purpose. Let $D(n)$ denote a BDD representing $\phi(n)$. A bottom-up evaluation computes $\mathsf{WMC}(\phi(n), W)$ in time linear in $|D(n)|$ using the recurrence:

$$\mathsf{val}(v) = \begin{cases} c & \text{if } v \text{ is terminal } c \in \{0, 1\} \\ W(X) \cdot \mathsf{val}(\mathsf{hi}(v)) + (1 - W(X)) \cdot \mathsf{val}(\mathsf{lo}(v)) & \text{if } v \text{ tests variable } X \end{cases}$$

where $\mathsf{hi}(v)$ and $\mathsf{lo}(v)$ are the high and low children of v, respectively.

Solving output. In addition to the computed probabilities for each query fact, we assume that the solver also returns the derivation graph and the resulting BDD representation. We formalize this as *provenance*:

Definition 4 (Provenance). *Given program $P = \langle \mathcal{R}, \mathcal{Q} \rangle$, the* provenance *returned by solving P is a pair $\Pi = (\Pi_{\mathsf{der}}, \Pi_{\mathsf{bdd}})$ where:*

- *Π_{der} is the complete derivation graph $\mathcal{G} = (N, E, \mathcal{L}, W)$ induced by P (Def 2), with event variables $X = \mathcal{L}(E)$.*
- *Π_{bdd} is a map that associates each ground fact q with a BDD $B(q)$ over X, representing the compiled objective formula $\phi(q)$.*

Definition 5 ($\texttt{Solve}$). *Given program $P = \langle \mathcal{R}, \mathcal{Q} \rangle$, $\texttt{Solve}(P)$ yields a pair $\langle \Psi, \Pi \rangle$, where $\Psi : \mathcal{Q} \to [0, 1]$ maps each query atom q to its probability $\mathsf{Pr}[q]$ and Π is the provenance (Def 4).*

3 Top-Level Incremental Inference Algorithm

The solving pipeline discussed in the previous section constructs the derivation graph and the BDD representation of each node from scratch. The key goal of this paper is to make solving incremental with respect to the previous inference result and a set of changes to the input Datalog program. To this end, we represent an *updated* program as a *change set* Δ applied to a base program.

Definition 6 (Change Set). *A* change set Δ *is a pair* (Δ^+, Δ^-) *containing rules to be inserted and deleted respectively. Applying* Δ *to a probabilistic Datalog program* $P = \langle \mathcal{R}, \mathcal{Q} \rangle$ *yields the updated program:*

$$P' = P \oplus \Delta := \langle (\mathcal{R} \setminus \Delta^-) \cup \Delta^+,\ \mathcal{Q} \rangle.$$

We represent rule modifications as the deletion of the old rule followed by the insertion of a new one. We also assume that no rule appears in both Δ^+ and Δ^-, and that the rules in Δ are input facts (i.e., rules with empty bodies).

Our incremental solving procedure is summarized in Algorithm 1 and consists of three key phases. First, IncDeriv (line 3) computes a *delta derivation graph* Δ_{der} that records exactly the grounded rule applications whose status changes in $P \oplus \Delta$ relative to P (Section 4). Second, given Δ_{der}, IncBddGen (line 4) incrementally updates the BDD representation used for weighted model counting (Section 5). The key idea is to reuse existing BDDs as much as possible by rebuilding only a selected recompilation region R and capturing the impact of the rebuilt portion via weight calibration, yielding an updated weight map W' and a map Δ_{bdd} of rebuilt BDDs. Finally, IncWMC (line 5) uses these artifacts to update query probabilities by performing weighted model counting only for impacted atoms in Δ_{fact}. Concretely, let $G' \triangleq (\Pi_{\mathsf{der}} \oplus \Delta_{\mathsf{der}})$ denote the updated derivation graph produced by applying the change set. Then, IncWMC computes Δ_Ψ by evaluating each affected atom v as follows:

$$\Pr[v] \;=\; \begin{cases} \mathrm{WMC}\big(\Delta_{\mathsf{bdd}}(v),\, W_{G'}\big) & \text{if } v \in \mathsf{R} \text{ (i.e., } v \in \mathsf{dom}(\Delta_{\mathsf{bdd}})), \\ \mathrm{WMC}\big(\Pi_{\mathsf{bdd}}(v),\, W'\big) & \text{if } v \in \Delta_{\mathsf{fact}} \setminus \mathsf{R}. \end{cases}$$

That is, atoms whose BDDs were rebuilt are evaluated using their updated BDDs under the full-recompilation weight map for G', whereas atoms outside the recompilation region (R) reuse their old BDDs under the calibrated map W'.

Theorem 1 (Correctness). *For any probabilistic Datalog program* P *and any change set* Δ*, let* $\texttt{Solve}(P) = \langle \Psi, \Pi \rangle$*. Then the incremental solver* `PINQ` *is semantically equivalent to batch recomputation:*

$$\texttt{PINQ}(P, \Psi, \Pi, \Delta) \;=\; \texttt{Solve}(P \oplus \Delta).$$

4 Incremental Derivation Graph Generation

This section describes the IncDeriv procedure for incrementally constructing the updated derivation graph. This algorithm proceeds in two phases. The first phase propagates edits *locally* to capture their immediate consequences, i.e., which ground rule instances become newly enabled or disabled. The second phase then accounts for the *non-local* effect of deletions. In a cyclic derivation graph, local propagation can leave derivations in place even after their last external justification has been removed. Thus, the second phase of IncDeriv performs "cycle repair" to eliminate such globally unsupported (spurious) derivations.

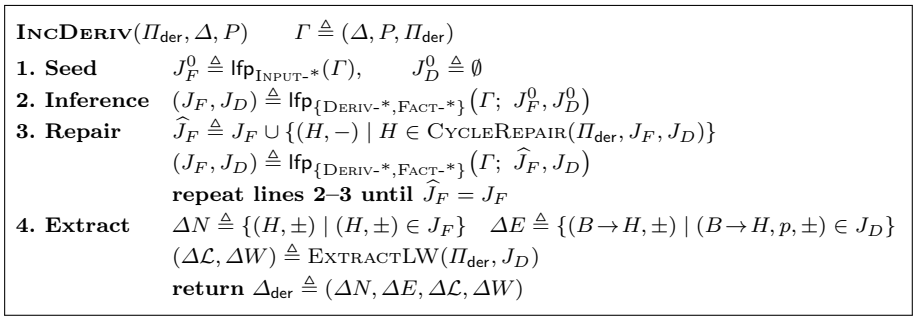

Fig. 2. IncDeriv overview. lfp_R saturates inference rules R (see Sec. 4.1); cycle repair is discussed in Sec. 4.2. ExtractLW assigns fresh label X_e with weight p to each inserted edge $(e, p, +) \in J_D$, and removes the label/weight of deleted edge $(e, p, -) \in J_D$.

The IncDeriv procedure is summarized in Figure 2 and maintains two working sets of updates: J_F collects atom insertions and deletions (e.g., $(H, +)$ or $(H, -)$), and J_D collects derivation-edge insertions and deletions (e.g., $(B \to H, p, +)$ or $(B \to H, p, -)$). Starting from the base edits in Δ, IncDeriv first seeds J_F (Step 1) and then saturates a set of inference rules (explained later) to compute a least fixed point over (J_F, J_D) (Step 2). This local saturation captures the immediate impact of the edit on enabled ground rule instances. In particular, it derives edge updates in J_D when a ground body becomes newly satisfied or falsified, and propagates these edge updates back to atom updates in J_F.

The second phase of IncDeriv addresses a key subtlety in cyclic derivation graphs. After Step 2, the fixed point for (J_F, J_D) may be too large despite being stable. In particular, within an SCC, atoms can remain mutually supported even after every derivation entering the SCC from outside has been removed, resulting in stale edges in J_D. To address this issue, Step 3 invokes CycleRepair to infer additional atoms that should be deleted. These newly inferred deletions are then added to J_F and the inference rules are re-saturated. As shown in Step 3 of Figure 2, IncDeriv alternates cycle repair and saturation until J_F stabilizes. Finally, Step 4 extracts $\Delta_{\mathsf{der}} = (\Delta N, \Delta E, \Delta\mathcal{L}, \Delta W)$ from the final sets.

Theorem 2 (Correctness of IncDeriv). *Let $P = \langle \mathcal{R}, \mathcal{Q} \rangle$ be a program and let Δ be an edit set of base facts. Let Π_{der} be the old derivation graph for P, and let Π'_{der} be the new derivation graph obtained by batch reconstruction for $P \oplus \Delta$. Let $\Delta_{\mathsf{der}} \triangleq \text{IncDeriv}(\Pi_{\mathsf{der}}, \Delta, P)$. Then, we have $\Pi_{\mathsf{der}} \oplus \Delta_{\mathsf{der}} = \Pi'_{\mathsf{der}}$.*

4.1 Inference Rules for Updating Derivation Graph

Figure 3 defines the inference system used in Steps 1–2 of the IncDeriv procedure. The system derives two kinds of updates:

- *Fact updates* of the form $\Gamma \models (H, \pm)$, asserting that atom H must be inserted into $(+)$ or deleted from $(-)$ the node set; and

$$\frac{(\texttt{insert}\ p :: I) \in \Delta}{\Gamma \models (I, +)}\ \textsc{Input-Add} \qquad \frac{(\texttt{delete}\ p :: I) \in \Delta}{\Gamma \models (I, -)}\ \textsc{Input-Delete}$$

$$\frac{\begin{array}{c} P = \langle \mathcal{R}, \mathcal{Q} \rangle \qquad \Pi_{\mathsf{der}} = (N, E, \mathcal{L}, W) \qquad p :: H :- \odot\, b_1, \ldots, \odot b_n \in \mathsf{Gnd}(\mathcal{R}) \\ B^+ \triangleq \{\, b_i \mid \odot \neq \neg \,\} \qquad B^- \triangleq \{\, b_i \mid \odot = \neg \,\} \qquad B \triangleq (B^+, B^-) \\ S^+ \triangleq \{\, b \mid \Gamma \models (b, +) \,\} \qquad S^- \triangleq \{\, b \mid \Gamma \models (b, -) \,\} \\ B^+ \subseteq \big((N \cup S^+) \setminus S^-\big) \qquad (B \to H) \notin E \end{array}}{\Gamma \models (B \to H, p, +)}\ \textsc{Deriv-Add}$$

$$\frac{\begin{array}{c} P = \langle \mathcal{R}, \mathcal{Q} \rangle \qquad \Pi_{\mathsf{der}} = (N, E, \mathcal{L}, W) \qquad p :: H :- \odot\, b_1, \ldots, \odot b_n \in \mathsf{Gnd}(\mathcal{R}) \\ B^+ \triangleq \{\, b_i \mid \odot \neq \neg \,\} \qquad B^- \triangleq \{\, b_i \mid \odot = \neg \,\} \qquad B \triangleq (B^+, B^-) \\ S^- \triangleq \{\, b \mid \Gamma \models (b, -) \,\} \qquad B^+ \cap S^- \neq \emptyset \qquad (B \to H) \in E \end{array}}{\Gamma \models (B \to H, p, -)}\ \textsc{Deriv-Del}$$

$$\frac{\Pi_{\mathsf{der}} = (N, E, \mathcal{L}, W) \qquad H \notin N \qquad \exists B, p.\ \Gamma \models (B \to H, p, +)}{\Gamma \models (H, +)}\ \textsc{Fact-Add}$$

$$\frac{\begin{array}{c} \Pi_{\mathsf{der}} = (N, E, \mathcal{L}, W) \qquad H \in N \qquad S_H \triangleq \{\, e \mid e = (B \to H) \in E \,\} \\ \forall e \in S_H.\ \Gamma \models (e, \mathcal{L}(e), -) \qquad \neg \exists B, p.\ \Gamma \models (B \to H, p, +) \end{array}}{\Gamma \models (H, -)}\ \textsc{Fact-Del}$$

Fig. 3. Inference rules for derivation graph updates, where $\Gamma \triangleq (\Delta, P, \Pi_{\mathsf{der}})$ and $\mathsf{Gnd}(\mathcal{R})$ ranges over all ground instances of rules in $\mathcal{R}$.

– *Edge updates* of the form $\Gamma \models (B \to H, p, \pm)$, asserting that the edge $B \to H$ must be inserted $(+)$ or deleted $(-)$ in the updated derivation graph.

The inference system propagates edits back and forth between facts and derivation edges until saturation. We now unpack the individual rules in Figure 3.

Initial fact updates. Rules Input-Add/Delete lift edits in Δ to seed the fact-update set. These are the only rules that inspect Δ; all remaining ones propagate their effect.

Edge updates. Rule Deriv-Add adds a ground edge $B \to H$ only if it is absent from E and its *positive* body becomes satisfied after applying the derived fact updates. Concretely, the side condition $B^+ \subseteq (N \cup S^+) \setminus S^-$ checks the *net* effect of simultaneous inserts/deletes on the body. It is worth noting that this rule does not test B^- because Π_{der} is maintained as a *positive-support* derivation graph: an edge is present whenever its positive prerequisites hold, independent of the negated literals. Negative literals are enforced later, when the edge is translated into the Boolean constraint for H. Dually, Deriv-Del removes an existing edge as soon as it loses positive support, i.e., when $(B \to H) \in E$ and $B^+ \cap S^- \neq \emptyset$. Here, B^- is again irrelevant for edge maintenance because changes to negated literals do not affect whether the edge is present in Π_{der}.

Fact updates. Rule Fact-Add derives an insertion of a head atom H when some derivation edge for H is inserted and H was not already present in the old node set N. Dually, Fact-Del derives a deletion of a head atom H when

$$\textsc{CycleRepair}(\Pi_{\mathsf{der}}, J_F, J_D): \; D_{\mathsf{cycle}} \qquad \Pi_{\mathsf{der}} = (N, E, \mathcal{L}, W)$$

$N' \triangleq \big(N \setminus \{H \mid (H,-) \in J_F\}\big) \;\cup\; \{H \mid (H,+) \in J_F\}.$
$E' \triangleq \big(E \setminus \{(B \to H) \mid \exists\ (B \to H, p, -) \in J_D\}\big) \;\cup\; \{(B \to H) \mid \exists (B \to H, p, +) \in J_D\}.$
$\mathsf{derive}_C \triangleq \mathsf{lfp}(\Phi_C)$, where
$\Phi_C(D) \triangleq \{\, H \in C \mid \exists (B \to H) \in E'.\ (B \subseteq N') \;\wedge\; (B \cap C = \emptyset) \,\}$
$\qquad\qquad \cup \{\, H \in C \mid \exists (B \to H) \in E'.\ (B \subseteq N') \;\wedge\; (B \cap C \subseteq D) \,\}.$
$\mathsf{DelCycle}(C) \triangleq \{\, H \in C \mid \exists (B \to H) \in E' \;\wedge\; H \notin \mathsf{derive}_C \,\}.$
$D_{\mathsf{cycle}} \triangleq \bigcup \{\, \mathsf{DelCycle}(C) \mid C \in \mathsf{SCC}(N', E') \}.$

Fig. 4. Cycle repair phase.

every old supporting edge for H is deleted and no new supporting edge for H is inserted. In other words, H disappears exactly when it loses all derivations.

4.2 Cycle Repair

The inference rules of Section 4.1 are sufficient on acyclic derivation graphs, but they can *under-delete* in the presence of cycles. In particular, because Π_{der} represents support edge-by-edge, an SCC can remain internally self-supporting after its last incoming justification is removed. As a result, local propagation may reach a fixed point that is stable but too large, as the next example shows.

Example 1. Consider the rules $A \coloneq I$, $A \coloneq B, I_1$, $B \coloneq A, I_2$. Assume I, I_1, I_2 hold in the old graph, so A is derived from I and B from A, enabling the cyclic edges $\{B, I_1\} \to A$ and $\{A, I_2\} \to B$. Now, if we delete I, neither A nor B should remain derivable, since the only remaining support is circular. However, local saturation deletes only $\{I\} \to A$ but does not remove the two cyclic edges.

The CycleRepair procedure (Figure 4) eliminates such spurious self-support by first constructing the new derivation graph (N', E') (first two lines) and then checking whether atoms in an SCC C still have non-circular support. To do so, it computes the least fixed point $\mathsf{derive}_C = \mathsf{lfp}(\Phi_C)$, where Φ_C adds $H \in C$ whenever there is a surviving edge $(B \to H) \in E'$ whose positive prerequisites hold in N' and whose prerequisites inside C are already established. Finally, any atom $H \in C$ with an incoming edge in E' but $H \notin \mathsf{derive}_C$ is marked for deletion.

5 Incremental BDD Generation

This section describes the incremental forward compilation procedure (Algorithm 2) for constructing the updated BDD for each query atom. The key idea is to reuse existing BDDs whenever possible, subject to weight updates. At a high level, the algorithm consists of four conceptual steps:

Algorithm 2: IncBddGen: Incremental BDD construction

Input: Old provenance Π_{der}; delta derivations Δ_{der}; old BDDs Π_{bdd}
Output: Delta Δ_{bdd}, updated weight map W', affected atoms Δ_{fact}

begin
  $\Pi'_{\mathsf{bdd}} \leftarrow \Pi_{\mathsf{bdd}}$; $\Delta_{\mathsf{bdd}} \leftarrow \emptyset$; $W' \leftarrow \mathsf{Weights}(\Pi_{\mathsf{der}} \oplus \Delta_{\mathsf{der}})$
  // Step 1: compute dependency closure to identify affected atoms
  $\Delta_{\mathsf{fact}} \leftarrow \mathsf{DepAnalysis}(\Pi_{\mathsf{der}}, \Delta_{\mathsf{der}})$
  // Step 2: identify regions where BDD structure must change
  $(\mathsf{R}, \mathsf{Anchor}) \leftarrow$ SelectRegion$(\Pi_{\mathsf{der}}, \Delta_{\mathsf{der}}, \Delta_{\mathsf{fact}})$
  // Step 3: rebuild BDDs inside R
  foreach $u \in \mathsf{R}$ **do**
    $\Pi'_{\mathsf{bdd}}(u) \leftarrow \mathsf{RebuildNodeBDD}(u,\ \Pi_{\mathsf{der}} \oplus \Delta_{\mathsf{der}},\ \Pi'_{\mathsf{bdd}})$
    $\Delta_{\mathsf{bdd}} \leftarrow \Delta_{\mathsf{bdd}} \cup \{(u \mapsto \Pi'_{\mathsf{bdd}}(u))\}$
  // Step 4: weight calibration to allow reuse outside R
  foreach $v \in \mathrm{dom}(\mathsf{Anchor})$ **do**
    $W' \leftarrow$ CalibrateWeight$\big(v,\ \mathsf{Anchor}(v),\ \Pi_{\mathsf{bdd}}(v),\ \Pi'_{\mathsf{bdd}}(v),\ W'\big)$
  return $(\Delta_{\mathsf{bdd}}, W', \Delta_{\mathsf{fact}})$

Step 1: dependency closure. The algorithm first identifies impacted atoms Δ_{fact} (via DepAnalysis) by taking the forward reachability closure in the derivation graph, seeded by atoms incident to Δ_{der}. Atoms outside Δ_{fact} are not affected by any edits, so their compiled BDDs (and probabilities) can be reused.

Step 2: region identification. The next step (SelectRegion) refines the affected set Δ_{fact} into a recompilation region $\mathsf{R} \subseteq \Delta_{\mathsf{fact}}$. This comprises atoms whose objective formulas must be rebuilt because Δ_{der} changes their Boolean structure. For atoms outside of the recompilation region, the solver can reuse their existing BDD subject to weight updates. To support reuse, SelectRegion also returns an *anchor map* Anchor for affected atoms outside R: for each such v, $\mathsf{Anchor}(v)$ is a variable already present in v's reused BDD whose weight is updated to summarize the contribution of derivations that pass through R.

Step 3: local rebuilding. Next, for each atom in R, the algorithm rebuilds its BDD using the updated derivation graph $\Pi_{\mathsf{der}} \oplus \Delta_{\mathsf{der}}$. The resulting changes are recorded in Δ_{bdd}; all other atoms retain their previous BDDs.

Step 4: weight calibration. Finally, the algorithm performs *weight calibration* so that reused BDDs outside R evaluate to the same probabilities they would under full recompilation. Intuitively, calibration retunes the weights of a small number of existing variables selected in Step 2, so they summarize the contribution of derivations that pass through R.

In the remainder of this section, we describe the two key ingredients behind IncBddGen, namely SelectRegion and CalibrateWeight.

5.1 Region Identification

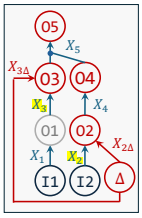

Fig. 5. G'

Region identification decides where the update forces a *structural* change in the compilation, i.e., which atoms must be rebuilt rather than handled by weight updates alone. The following example gives intuition for when such structural changes are unavoidable:

Example 2. Consider Figure 5, where the two *red* edges represent new derivation edges introduced by change set Δ. Each derivation edge is labeled by a Boolean variable X; for example, X_5 denotes the edge variable associated with the hyperedge $(O_3 \wedge O_4) \rightarrow O_5$. We consider two variants of Δ to illustrate when the BDD of O_5 can be reused and when it must be rebuilt.

Case 1: Single affected derivation. Suppose Δ introduces a new derivation for O_2 but *not* for O_3 (i.e., ignore the edge from Δ to O_3 in Figure 5). In this case, only O_2 requires BDD reconstruction, with O_4, O_5 reusing their old BDDs subject to weight changes. To see why, consider the old objective formula for O_5:

$$\phi_{\mathsf{old}}(O_5) \;=\; X_5 \;\wedge\; \underline{\phi_{\mathsf{old}}(O_3)} \;\wedge\; \underline{X_4 \wedge \phi_{\mathsf{old}}(O_2)},$$

and its corresponding probability $\Pr\big(\phi_{\mathsf{old}}(O_5)\big)$ which factorizes as $\Pr(X_5) \cdot \underline{\Pr\big(\phi_{\mathsf{old}}(O_3)\big)} \cdot \underline{\Pr(X_4) \cdot \Pr\big(\phi_{\mathsf{old}}(O_2)\big)}$. Since all input facts and edge variables are assumed mutually independent, any two formulas that mention *disjoint* sets of these primitives are also independent. Thus, we have $\phi_{\mathsf{old}}(O_3) \perp\!\!\!\perp \phi_{\mathsf{old}}(O_2)$, which justifies the product decomposition above.

After applying Δ, the objective formula of O_2 becomes $\phi_{\mathsf{new}}(O_2) \;=\; \phi_{\mathsf{old}}(O_2) \vee (X_{2\Delta} \wedge \Delta)$, and the updated formula for O_5 is

$$\phi_{\mathsf{new}}(O_5) \;=\; X_5 \;\wedge\; \underline{\phi_{\mathsf{old}}(O_3)} \;\wedge\; \underline{X_4 \wedge \phi_{\mathsf{new}}(O_2)}.$$

Importantly, the independence relation $\phi_{\mathsf{old}}(O_3) \perp\!\!\!\perp \phi_{\mathsf{new}}(O_2)$ still holds, so the effect of Δ can be fully encapsulated in the updated probability of O_2. Thus, $\Pi_{\mathsf{bdd}}(O_5)$ can be re-used after calibrating the weight for edge X_2.

Case 2: Correlated derivations. Now assume that Δ introduces new derivations for *both* O_2 and O_3, as shown by the red edges in Figure 5. In this case, $\phi_{\mathsf{new}}(O_3) \;=\; \phi_{\mathsf{old}}(O_3) \;\vee\; (X_{3\Delta} \wedge \Delta)$, and the new objective formula of O_5 is:

$$\phi_{\mathsf{new}}(O_5) \;=\; X_5 \;\wedge\; \underline{\phi_{\mathsf{new}}(O_3)} \;\wedge\; \underline{X_4 \wedge \phi_{\mathsf{new}}(O_2)}.$$

In contrast to case 1, $\phi_{\mathsf{new}}(O_2)$ and $\phi_{\mathsf{new}}(O_3)$ now share the common dependency Δ, meaning they are no longer statistically independent and the correlation must be explicitly represented in the Boolean structure. Thus, O_5 is included in the recompilation region R, and its BDD must be rebuilt.

As this example illustrates, the soundness of a recompilation region depends on how derivations interact, where soundness is defined as follows:

Definition 7 (Region soundness). *Let* $G' \triangleq (\Pi_{\mathsf{der}} \oplus \Delta_{\mathsf{der}})$ *be the updated derivation graph, with* Π_{bdd} *mapping atoms to their* old *BDDs and* W *the old weight map. A recompilation region* $\mathsf{R} \subseteq \Delta_{\mathsf{fact}}$ *is* sound *if there exists a weight map* W' *such that:*

(S1) W' *differs from* W *only on variables labeling edges in* Δ_{der} *and on variables that already occur in some reused BDD* $\Pi_{\mathsf{bdd}}(u)$ *for* $u \notin \mathsf{R}$*; and*

(S2) *For every atom* $u \notin \mathsf{R}$*, reusing* $\Pi_{\mathsf{bdd}}(u)$ *and evaluating it under* W' *yields the correct probability under* G'*:*

$$\forall u \notin \mathsf{R}. \quad \mathsf{WMC}\big(\Pi_{\mathsf{bdd}}(u),\, W'\big) \;=\; \mathsf{Pr}[u \text{ holds under } G'] \,.$$

Definition 7 defines soundness with respect to a semantic condition (S2), but directly checking this condition would require computing $\mathsf{Pr}[u$ holds under $G']$, which would entirely defeat the point of incremental computation. Thus, we require an alternative graph-theoretic criterion that can be checked locally. To this end, we introduce a representation that summarizes atom-to-atom influence:

Definition 8 (Flattened derivation graph). *Let* $G' \triangleq (\Pi_{\mathsf{der}} \oplus \Delta_{\mathsf{der}})$ *be the updated derivation graph. The* flattened derivation graph $\mathcal{F}(G')$ *is the graph whose nodes are atoms and that contains an arc* $u \to h$ *iff there exists an edge* $e \in G'$ *with* $\mathsf{Head}(e) = h$ *and* $u \in \mathsf{Body}(e)$*. For atom* u*, its* dependency set $\mathsf{Dep}(u)$ *is the set of atoms* v *such that there exists a directed path from* v *to* u *in* $\mathcal{F}(G')$*.*

That is, flattening represents hyperedges as a set of edges, and the dependencies of an atom u include all atoms that may transitively influence u. Next, to capture where changes *originate*, we introduce the concept of source atoms.

Definition 9 (Source atoms). *For a given* Δ_{der}*, the* source atoms S *are:*

$$S \triangleq \{\, a \mid \exists e \in \Delta_{\mathsf{der}}.\ a \in \mathsf{Head}(e) \cup \mathsf{Body}(e) \,\}.$$

That is, a source is any atom that appears in the diff Δ_{der}. Next, we introduce *summary nodes* to capture when the effect of a change reaches an atom through a *single* interface point. Intuitively, reuse at a target atom t becomes difficult when influence from a source can arrive along multiple paths that enter t through different immediate predecessors. Summary nodes ensure that this cannot happen by finding a single interface point.

Definition 10 (Summary node; anchor variable). *Let* G' *be the new derivation graph,* $s \in S$ *a source atom, and* t *an atom reachable from* s *in* $\mathcal{F}(G')$*. A* summary node *for* (s, t) *is an atom* $\mathsf{Summ}(s, t) = t'$ *(if it exists) such that:*

(C1) *The arc* $t' \to t$ *is in* $\mathcal{F}(G')$ *and is the* unique last hop on s–t paths*: every path from* s *to* t *in* $\mathcal{F}(G')$ *ends with* $t' \to t$*. Moreover, for every* $v \in \mathsf{Dep}(t')$*, every path from* v *to* t *in* $\mathcal{F}(G')$ *also ends with* $t' \to t$*.*

(C2) *There exists an old incoming edge* $e^* = (B \to t') \in \Pi_{\mathsf{der}}$ *such that* $\mathcal{L}(e^*)$ *occurs in* $\Pi_{\mathsf{bdd}}(t)$ *and*

$$\Big(\bigwedge_{u \in B} u\Big) \wedge \neg\Big(\bigvee_{\substack{e'=(B' \to t') \in \Pi_{\mathsf{der}} \\ e' \neq e^*}} \big(\bigwedge_{u \in B'} u\big)\Big) \not\equiv \bot.$$

Any witness e^* *for* (C2) *is an* anchor edge *for* (s,t)*, and* $\mathcal{L}(e^*)$ *is referred to as the corresponding* anchor variable*. If no such* t' *exists, we write* $\mathsf{Summ}(s,t) = \bot$*.*

Intuitively, a summary node certifies that the impact of an update originating at s can be absorbed at t by reweighting a variable that already appears in $\Pi_{\mathsf{bdd}}(t)$. Condition (C1) is the key structural requirement: it makes t' the *unique last hop* into t for all relevant influence from the source s, so the update reaches t through a single interface rather than through multiple immediate predecessors (ruling out the correlation failure mode of Example 2). Condition (C2) then supplies a concrete parameter to reweight: it requires an old incoming edge e^* for t' whose label already occurs in $\Pi_{\mathsf{bdd}}(t)$ and is non-redundant, so changing its weight can actually change the contribution of the $t' \to t$ branch without rebuilding the Boolean structure of $\Pi_{\mathsf{bdd}}(t)$. We next use summary nodes to determine whether the BDD for an existing atom is reusable:

Definition 11 (Reusability). *Given a source* $s \in S$*, we say that an atom* v *is* reusable w.r.t. s*, written* $\mathsf{Reuse}_s(v)$*, if either* $s \notin \mathsf{Dep}(v)$*, or both of the following conditions are satisfied:*

(R1) *Every incoming edge to* v *in the updated derivation graph* G' *exists in* Π_{der}*;*
(R2) *For every incoming edge* $(B \to v)$ *in* $\mathcal{F}(G')$ *and every* $b \in B$*, either* $s \notin \mathsf{Dep}(b)$ *or* $b = \mathsf{Summ}(s,v)$*.*

An atom v *is* reusable*, denoted* $\mathsf{Reuse}(v)$*, if either* $v \notin \Delta_{\mathsf{fact}}$ *or* $\mathsf{Reuse}_s(v)$ *holds for every source* $s \in S$*.*

As the following theorem states, we can now define the recompilation region to consist of all atoms that are reusable according to the above definition.

Theorem 3. *Given new derivation graph* G' *and affected atoms* Δ_{fact}*, let* R *be* $\{\, v \in \Delta_{\mathsf{fact}} \mid \neg\mathsf{Reuse}(v) \}$*. Then* R *is sound in the sense of Definition 7.*

5.2 Weight Calibration

In the previous subsection, we identified the region R where BDD structure must be rebuilt and, for each reusable atom $t \notin \mathsf{R}$ and source $s \in S$ that can influence t, we extracted a summary node $u = \mathsf{Summ}(s,t)$ together with an anchor edge $e^* = (B \to u) \in \Pi_{\mathsf{der}}$ whose label $X \triangleq \mathcal{L}(e^*)$ already occurs in the reused BDDs. Rebuilding the region and performing weighted model yields the new probabilities of $u \in \mathsf{R}$ under the updated derivation graph $G' \triangleq (\Pi_{\mathsf{der}} \oplus \Delta_{\mathsf{der}})$. However, *outside* R, we keep the *old* BDD structure, so these reused BDDs

mention variables that were already present in the previous compilation (labels of edges from Π_{der}), rather than any fresh labels introduced by Δ_{der}. Weight calibration bridges this gap by adjusting the weights of a small set of such pre-existing *anchor* variables, so the reused BDDs reflect the updated probabilities without rebuilding their Boolean structure.

Concretely, consider a pair (s,t) and let $u = \mathsf{Summ}(s,t)$ be its summary node with anchor variable X. Since $u \in \mathsf{R}$, we rebuild its BDD under G', obtaining $\Pi'_{\mathsf{bdd}}(u)$. This rebuilt BDD fixes the probability of u after the update:

$$p_u \triangleq \mathsf{WMC}\big(\Pi'_{\mathsf{bdd}}(u),\, W_{G'}\big),$$

where $W_{G'}$ is the weight map in G'. To make this update visible to atoms outside R without rebuilding their BDDs, we transfer the change in u's probability onto the anchor variable X, which already appears in the reused compilation. Intuitively, X is the single weight parameter through which the contribution of u is exposed to reused BDDs downstream of the $u \to t$ interface.

Calibration sets a new weight $W'(X)$ so that the *old* BDD for u evaluates to the target value p_u under the updated weights. Concretely, we view the weight of X as an unknown $\omega \in [0,1]$ and keep all other variable weights fixed as in the current map W'. Conditioning $\Pi_{\mathsf{bdd}}(u)$ on the value of X yields two constants:

$$p_u^+ \triangleq \mathsf{WMC}\big(\Pi_{\mathsf{bdd}}(u)\,|_{X=\mathsf{true}},\, W'\big), \qquad p_u^- \triangleq \mathsf{WMC}\big(\Pi_{\mathsf{bdd}}(u)\,|_{X=\mathsf{false}},\, W'\big).$$

Since X is fixed in the conditioned BDDs, these do not depend on ω; hence:

$$\mathsf{WMC}\big(\Pi_{\mathsf{bdd}}(u),\, W'[X \mapsto \omega]\big) \;=\; \omega \cdot p_u^+ + (1-\omega)\cdot p_u^-. \tag{1}$$

Calibration chooses ω so that Equation (1) equals the target probability p_u. Thus, we can choose the new weight of X as follows:

$$W'(X) \;\triangleq\; \frac{p_u - p_u^-}{p_u^+ - p_u^-}.$$

The non-subsumption requirement used when selecting e^* ensures $p_u^+ \neq p_u^-$, so the denominator is nonzero and the update is well-defined. Instantiating this equation starting from $W' \leftarrow W$ and updating one anchor variable at a time yields a calibrated weight map W'.

6 Implementation and Evaluation

We have implemented the proposed approach in a new tool called `PINQ` written in `C++`. `PINQ` instruments and incrementalizes the semi-naive solving procedure of `Soufflé` [25] and uses `CUDD` [27] as its underlying BDD library. In the remainder of this section, we describe an experimental evaluation of `PINQ` that is designed to answer the following research questions:
RQ1: What is the end-to-end benefit of incremental solving compared to recomputation from scratch? **RQ2:** Where does `PINQ` spend its running time, and how

does the runtime breakdown change across update regimes? **RQ3:** What is the relative contribution of each key algorithmic component of `PINQ` to end-to-end performance?

All experiments are conducted on a machine equipped with an AMD Ryzen 7 7800X3D 8-core processor and 32 GB of RAM, running Ubuntu 22.04.5 LTS.

Base Datalog programs. To answer these research questions, we evaluate `PINQ` on a suite of probabilistic Datalog instances derived from a classic program-analysis task: *side-channel detection* [34,35,36]. We choose this application domain because side-channel risk assessment depends on estimating the likelihood of information leakage rather than establishing a purely Boolean property [4,24,33]. Furthermore, recent work has shown that probabilistic Datalog provides a natural modeling framework for such analyses [34]. Thus, we conduct our evaluation using the same program analysis formulation from [34] and analyze implementations of well-known cryptographic primitives such as AES [6], SHA3, and MAC-Keccak [1,23]. Overall, this yields 19 different base Datalog programs that differ in their input facts and their probabilities. Table 1 summarizes key statistics of this benchmark suite, with #nodes and #edges reporting the number of nodes and hyperedges in the corresponding derivation graphs.

Table 1. Benchmark stats.

	#nodes	**#edges**
Med	9k	6k
Avg	43k	33k
Max	200k	173k

Program updates. To model iterative development, we apply edit sets of size $\Delta \in \{0.5\%, 1\%, 1.5\%\}$ to the input fact base, with absolute caps of 150, 300, and 450 facts, respectively, for larger benchmarks. For each update, we vary the mix of deletions and insertions within the edit set. Concretely, we choose a *deletion fraction* $\alpha \in [0, 1]$, delete $\alpha\Delta$ of the current input facts, and insert $(1 - \alpha)\Delta$ new input facts. Thus, $\alpha = 1$ corresponds to deletion-only updates, $\alpha = 0$ corresponds to insertion-only updates, and intermediate values model mixed edits. New facts are drawn from a precomputed pool of syntactically valid candidate input facts. To construct this pool, we randomly synthesize additional program statements whose data dependencies reference variables appearing in the original source program, and, for each synthesized statement, we emit the corresponding collection of input facts required by the analysis.

Experimental setup. Each of our experimental benchmarks can be represented as a triple (P, Δ, α), where P is the base probabilistic Datalog program, Δ specifies the edit-set size, and α is the deletion fraction that determines the split between deletions and insertions. We evaluate all (Δ, α) pairs from the Cartesian product $\Delta \in \{0.5\%, 1\%, 1.5\%\}$ and $\alpha \in \{0, 0.25, 0.5, 0.75, 1\}$, yielding 15 update regimes per base program. Using this methodology, we obtain a total of 285 benchmarks.

Baseline. We use our Soufflé-based implementation of ProbLog-style inference as the full-recomputation baseline. We do not use ProbLog [7,8] itself because, in our experiments, even the latest version completed fewer than 30% of the benchmarks and timed out on most of the remaining instances, making the

incremental-vs.-recomputation comparison unreliable. Our Soufflé-based implementation completes all benchmarks within the timeout and therefore provides a practical full-recomputation baseline.

6.1 End-to-end Performance Improvement Evaluation

Table 2 summarizes the end-to-end speedups of PINQ on all 285 benchmarks, where speedup is measured relative to full recomputation. We report the arithmetic mean, geometric mean, maximum speedup, and the fraction of instances with speedup greater than 1. Overall, PINQ achieves an arithmetic-mean speedup of 17.1× and a maximum speedup of 55.2×. Moreover, every benchmark instance achieves speedup greater than 1.

Table 2. Results.

Statistic	Speedup
Arith. mean	17.1×
Geom. mean	14.8×
Maximum	55.2×
% faster (> 1)	100%

To provide a finer-grained view of these results, Figure 6 visualizes the variation in end-to-end speedup across different update regimes using a heat map. Each cell aggregates results from all benchmark instances under a fixed (Δ, α) configuration, and reports the average end-to-end speedup across those benchmarks. Each cell shows the speedup $T_{\text{full}}/T_{\text{inc}}$ for a given update magnitude Δ (rows) and deletion fraction α (columns). Warmer (redder) colors indicate larger speedups; cooler colors indicate smaller speedups.

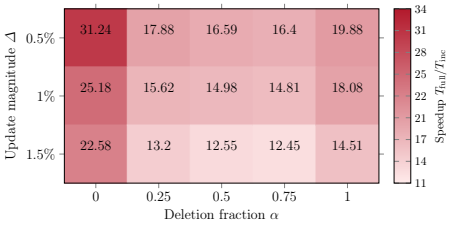

Fig. 6. End-to-end speedup heat maps.

We observe two dominant patterns in Figure 6. First, as expected, speedups decrease monotonically as the update magnitude Δ grows. For example, for insertion-only updates ($\alpha = 0$), the average speedup drops from 31.24× at $\Delta = 0.5\%$ to 25.18× at $\Delta = 1\%$ and 22.58× at $\Delta = 1.5\%$. Second, pure insertions ($\alpha = 0$) consistently yield the largest speedups across all Δ, while *mixed* updates tend to yield increasingly smaller speed-ups as α grows. However, deletion-only updates ($\alpha = 1$) are often slightly faster than mixed edits, which arises from our implementation strategy, which propagates all deletions before processing insertions; mixed updates therefore incur additional initialization costs for the data structures used to represent provenance.

6.2 Runtime Breakdown

Next, to address **RQ2**, we evaluate which stages dominate end-to-end runtime in practice, and how this profile changes under incrementality. Recall that probabilistic Datalog solving consists of three conceptual stages (for both incremental and non-incremental versions): (i) derivation-graph construction, (ii) compilation to the

Component	Full (%)	PINQ (%)	Δ (pp)
Derivation	5.6	35.4	+29.8
Compilation	93.6	62.0	-31.6
WMC	0.7	2.6	+1.9

Fig. 7. Runtime breakdown.

underlying knowledge representation (in our case, BDD construction), and (iii) weighted model counting (WMC). Figure 7 reports the fraction of end-to-end time spent in each stage, averaged over all benchmarks, both for full recomputation and for PINQ.

As is evident from Figure 7, full recomputation is dominated by compilation: BDD construction accounts for 93.6% of total runtime on average, while derivation construction contributes 5.6% and WMC is negligible (0.7%). Under PINQ, compilation remains the largest component, but its share drops to 62.0% because updates reuse most compiled structure and only rebuild what is impacted by the edit. The apparent increase in the share of derivation maintenance (to 35.4%) is therefore primarily a denominator effect: once compilation becomes cheaper, the fixed cost of updating the derivation graph constitutes a much larger fraction of the remaining end-to-end time.

6.3 Ablation Studies

To answer RQ3, we evaluate ablations that disable individual components of PINQ and measure the resulting end-to-end performance. Concretely, we compare the complete system PINQ (**Ours**) against two variants. The first variant, **NoDer**, disables incremental derivation maintenance (i.e., it rebuilds the complete derivation graph from scratch), and the second variant, **NoBDD**, disables incremental BDD construction.

Table 3. Ablation results.

Variant	Arith. mean	Geom. mean	% faster	Max
PINQ	17.1×	14.8×	100%	55.2×
NoDer	9.2×	8.6×	100%	14.2×
NoBdd	1.1×	1.1×	100%	1.1×

Table 3 reports the resulting end-to-end speedups relative to full recomputation. As expected, disabling incremental BDD updates largely eliminates the benefit of incrementality, with NoBdd achieving only marginal improvement over full recomputation (average 1.1×), indicating that reuse in the compilation stage is the primary driver of PINQ's overall speedups. In contrast, NoDer still provides non-trivial gains but reduces the average speedup substantially compared to PINQ, showing that incremental derivation maintenance is also important.

To gain further intuition about incremental compilation, we ablate this component more finely. We consider two incremental compilation strategies. The first, **IncRegionalBdd** (our default), only rebuilds BDDs inside the identified recompilation and then performs weight calibration. The second, **IncNaiveBdd**, rebuilds BDDs for all atoms in Δ_{fact} (i.e., the set produced by dependency closure). To quantify the benefit of region-based updates, we report

Regime	Arith. mean	Geom. mean	% faster	Max
$\alpha = 0.00$	3.8×	2.2×	100%	12×
$\alpha = 0.25$	2.4×	1.7×	100%	8.8×
$\alpha = 0.50$	1.8×	1.4×	100%	6.7×
$\alpha = 0.75$	1.1×	1.1×	100%	1.1×
$\alpha = 1.00$	1.0×	1.0×	88%	1.1×
Overall	**2.0×**	**1.4×**	**98%**	**12×**

Fig. 8. Speedup of region-based BDD updates over a naive incremental baseline.

the speedup of IncRegionalBdd over IncNaiveBdd, computed as the runtime of IncNaiveBdd divided by the runtime of IncRegionalBdd. Values above 1 indicate that region-based updates are faster.

The results of this fine-grained ablation are reported in Figure 8. Overall, IncRegionalBdd is faster than IncNaiveBdd in nearly all cases (98% of instances), yielding an average speedup of 2.0×. The magnitude of the improvement depends strongly on the deletion fraction α: region-based updates are most effective for insertion-heavy edits and the advantage shrinks as updates become more deletion-heavy. A small number of updates exhibit substantially larger gains (up to 12×). These outliers arise when the naive baseline rebuilds enough structure to trigger expensive BDD variable reordering in CUDD, whereas region-based updates rebuild less structure and are therefore less likely to incur this overhead.

7 Related Work

Datalog underlies many program analysis techniques, with applications spanning data-race detection [20,22,39], side-channel analysis [36,35], points-to analysis [2,19,26,37,40], program differencing [29], and thread-modular analysis [11,12]. Probabilistic variants such as ProbLog [7,8], Bingo [22], Praline [34], and Scallop [17] extend this foundation with quantitative semantics and enable analyses that rank, prioritize warnings or estimate risk of a vulnerability. This paper studies how to *incrementally compile* such probabilistic analyses so that small edits can be reflected in updated probabilities without rerunning the entire pipeline.

There is extensive work on incremental evaluation of *deterministic* Datalog, including DRed [9], IncA [32], and later systems [30,31,41]. These methods aim to maintain the fixed-point *set of derived facts* under insertions and deletions. However, that interface is too weak for exact probabilistic inference because probabilities depend on the full space of derivations, not just which facts are ultimately true. Moreover, these prior techniques do not address how BDD representations can be reused for weighted model counting.

The Drake [10] system targets evolving Datalog-style program analyses and builds a provenance object that is *differential* across versions. At a high level, Drake constructs a change-focused derivation graph that captures how an edit affects the derivations of reported alarms, and then performs Bayesian inference over this graph to prioritize alarms that are most likely to be introduced by the change. In contrast to Drake which aims to explain what alarms are caused by a change, this work aims to incrementally update query probabilities under edits.

Finally, prior work studies incremental (weighted) model counting for evolving Boolean constraints, often by compiling a base formula to a tractable circuit (e.g., Decision-DNNF/CCDD) and updating counts as the formula is edited over time [38,15,28,5,13,14]. This line of work takes the Boolean formula (e.g., CNF) as the primary object and designs update mechanisms around small syntactic deltas to that formula. Our setting differs in the *source* of change: edits occur at the level of probabilistic Datalog rules and facts, which induce changes in the

derivation graph and indirectly affect the compiled Boolean objective. The goal here is therefore not incremental counting for a user-supplied evolving formula, but incremental compilation that identifies where the compiled representation must be re-built vs where it can be re-used via weight updates.

8 Conclusion

This paper presents `PINQ`, an end-to-end framework for incremental inference in probabilistic Datalog that preserves exact semantics under both insertions and deletions while avoiding full recomputation. `PINQ` incrementally updates only the affected portions of the derivation graph and subsequently updates the compiled BDD representation, favoring parametric (as opposed to structural) changes whenever possible. We evaluate `PINQ` on probabilistic program-analysis workloads for detecting side-channel information leaks under diverse source-program update regimes. Our results demonstrate that incremental solving can substantially reduce end-to-end latency for program analysis applications.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data-Availability Statement

The artifact supporting the experimental results of this paper is archived on Zenodo at `https://doi.org/10.5281/zenodo.19644611`. It contains the implementation, benchmark inputs, scripts, and instructions for reproducing the main experimental results. Exact runtime measurements may vary depending on the hardware and system environment.

References

1. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.A., Grégoire, B., Strub, P.Y.: Verified proofs of higher-order masking. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 457–485. Springer (2015). `https://doi.org/10.1007/978-3-662-46800-5_18`
2. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. pp. 243–262 (2009). `https://doi.org/10.1145/1639949.1640108`
3. Bryant, R.E.: Binary decision diagrams. In: Handbook of model checking, pp. 191–217. Springer (2018). `https://doi.org/10.1007/978-3-319-10575-8_7`
4. Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 875–890. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). `https://doi.org/10.1145/3133956.3134058`

5. Cheng, C., Luo, Y., Jiang, J.H.R.: Knowledge compilation for incremental and checkable stochastic boolean satisfiability. In: Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence. pp. 1862–1872 (2024). https://doi.org/10.24963/ijcai.2024/206
6. Coron, J.S., Prouff, E., Rivain, M., Roche, T.: Higher-order side channel security and mask refreshing. In: International Workshop on Fast Software Encryption. pp. 410–424. Springer (2013). https://doi.org/10.1007/978-3-662-43933-3_21
7. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: A probabilistic prolog and its application in link discovery. In: IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence. pp. 2462–2467. IJCAI-INT JOINT CONF ARTIF INTELL (2007), https://dl.acm.org/doi/10.5555/1625275.1625673
8. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted boolean formulas. Theory and Practice of Logic Programming **15**(3), 358–401 (2015). https://doi.org/10.1017/S1471068414000076
9. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. ACM SIGMOD Record **22**(2), 157–166 (1993). https://doi.org/10.1145/170036.170066
10. Heo, K., Raghothaman, M., Si, X., Naik, M.: Continuously reasoning about programs using differential bayesian inference. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 561–575 (2019). https://doi.org/10.1145/3314221.3314616
11. Kusano, M., Wang, C.: Flow-sensitive composition of thread-modular abstract interpretation. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 799–809 (2016). https://doi.org/10.1145/2950290.2950291
12. Kusano, M., Wang, C.: Thread-modular static analysis for relaxed memory models. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 337–348 (2017). https://doi.org/10.1145/3106237.3106243
13. Lagniez, J.M., Marquis, P.: An improved decision-dnnf compiler. In: IJCAI. vol. 17, pp. 667–673 (2017). https://doi.org/10.24963/ijcai.2017/93
14. Lai, Y., Meel, K.S., Yap, R.H.: The power of literal equivalence in model counting. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 3851–3859 (2021). https://doi.org/10.1609/aaai.v35i5.16503
15. Lai, Y., Meel, K.S., Yap, R.H.: Panini: an efficient and flexible knowledge compiler. In: International Conference on Computer Aided Verification. pp. 92–105. Springer (2025). https://doi.org/10.1007/978-3-031-98682-6_6
16. Li, T., Zhang, X.: Combining formal and informal information in bayesian program analysis via soft evidences. Proceedings of the ACM on Programming Languages **9**(OOPSLA1), 1774–1801 (2025). https://doi.org/10.1145/3720508
17. Li, Z., Huang, J., Naik, M.: Scallop: A language for neurosymbolic programming. Proceedings of the ACM on Programming Languages **7**(PLDI), 1463–1487 (2023). https://doi.org/10.1145/3591280
18. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag, Berlin, Heidelberg, 2nd, extended edn. (1987)
19. Madsen, M., Yee, M.H., Lhoták, O.: From datalog to flix: A declarative language for fixed points on lattices. ACM SIGPLAN Notices **51**(6), 194–208 (2016). https://doi.org/10.1145/2908080.2908096
20. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design

and Implementation. pp. 308–319 (2006). https://doi.org/10.1145/1133981.1134018
21. Przymusinski, T.C.: On the declarative semantics of deductive databases and logic programs. In: Foundations of deductive databases and logic programming, pp. 193–216. Elsevier (1988). https://doi.org/10.1016/B978-0-934613-40-8.50009-9
22. Raghothaman, M., Kulkarni, S., Heo, K., Naik, M.: User-guided program reasoning using bayesian inference. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 722–735 (2018). https://doi.org/10.1145/3192366.3192417
23. Rivain, M., Prouff, E.: Provably secure higher-order masking of aes. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 413–427. Springer (2010). https://doi.org/10.1007/978-3-642-15031-9_28
24. Saha, S., Ghentiyala, S., Lu, S., Bang, L., Bultan, T.: Obtaining information leakage bounds via approximate model counting. Proc. ACM Program. Lang. **7**(PLDI) (Jun 2023). https://doi.org/10.1145/3591281
25. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in datalog. In: Proceedings of the 25th International Conference on Compiler Construction. p. 196–206. CC '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2892208.2892226
26. Smaragdakis, Y., Kastrinis, G., Balatsouras, G.: Introspective analysis: context-sensitivity, across the board. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 485–495 (2014). https://doi.org/10.1145/2594291.2594320
27. Somenzi, F.: CUDD: CU decision diagram package, release 3.0.0. Software library (2015), https://github.com/cuddorg/cudd
28. Sundermann, C., Raab, H., Heß, T., Thüm, T., Schaefer, I.: Exploiting d-dnnfs for repetitive counting queries on feature models. arXiv preprint arXiv:2303.12383 (2023)
29. Sung, C., Lahiri, S.K., Enea, C., Wang, C.: Datalog-based scalable semantic diffing of concurrent programs. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. p. 656–666. ASE 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3238147.3238211
30. Szabó, T., Bergmann, G., Erdweg, S., Voelter, M.: Incrementalizing lattice-based program analyses in datalog. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 1–29 (2018). https://doi.org/10.1145/3276509
31. Szabó, T., Erdweg, S., Bergmann, G.: Incremental whole-program analysis in datalog with lattices. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 1–15 (2021). https://doi.org/10.1145/3453483.3454026
32. Szabó, T., Erdweg, S., Voelter, M.: Inca: a dsl for the definition of incremental program analyses. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 320–331 (2016). https://doi.org/10.1145/2970276.2970298
33. Tizpaz-Niari, S., Černý, P., Trivedi, A.: Quantitative mitigation of timing side channels. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 140–160. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_8
34. Wang, J., Halalingaiah, S., Chen, W., Wang, C., Dillig, I.: Probabilistic inference for datalog with correlated inputs. Proceedings of the ACM on Programming Languages **9**(OOPSLA2), 220–247 (2025). https://doi.org/10.1145/3763058

35. Wang, J., Sung, C., Raghothaman, M., Wang, C.: Data-driven synthesis of provably sound side channel analyses. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 810–822. IEEE (2021). https://doi.org/10.1109/ICSE43902.2021.00079
36. Wang, J., Sung, C., Wang, C.: Mitigating power side channels during compilation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 590–601 (2019). https://doi.org/10.1145/3338906.3338913
37. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using datalog with binary decision diagrams for program analysis. In: Asian Symposium on Programming Languages and Systems. pp. 97–118. Springer (2005). https://doi.org/10.1007/11575467_8
38. Yang, S., Meel, K.S.: Towards projected and incremental pseudo-boolean model counting. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 39, pp. 11399–11407 (2025). https://doi.org/10.1609/aaai.v39i11.33240
39. Zhang, X., Grigore, R., Si, X., Naik, M.: Effective interactive resolution of static analysis alarms. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 1–30 (2017). https://doi.org/10.1145/3133881
40. Zhang, X., Mangal, R., Grigore, R., Naik, M., Yang, H.: On abstraction refinement for program analyses in datalog. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 239–248 (2014). https://doi.org/10.1145/2594291.2594327
41. Zhao, D., Subotic, P., Raghothaman, M., Scholz, B.: Towards elastic incrementalization for datalog. In: Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming. pp. 1–16 (2021). https://doi.org/10.1145/3479394.3479415