Incremental Evaluation of Dynamic Datalog Programs as a Higher-order DBSP Program

Bruno Rucy Carneiro Alves de Lima[1,*], Merlin Kramer[2], Kalmer Apinis[1] and Kristopher Micinski[3]

[1]University of Tartu, Tartu, Estonia
[2]Independent Researcher, Wuppertal, Germany
[3]Syracuse University, Syracuse, USA

Abstract

We show that evaluating positive Datalog programs in a bottom-up manner can be modelled as a "circuit" of facts and rules where both can be freely added and retracted during runtime. This is realised by writing a definitional interpreter as a higher order program that defines bottom-up evaluation in terms of Database Stream Processing Theory (DBSP) operators that have incremental semantics. Our approach is compared against Soufflé, and DDLog. Up to almost 10 times faster performance is exhibited when handling incremental updates compared to DDLog, and 2 times when compared to Soufflé.

Keywords

Datalog, DBSP, Bottom-up Evaluation, Incremental View Maintenance

1. Introduction

Bottom-up Datalog garners significant interest among research and industry use cases that need logic programming at scale, as the most universally adopted evaluation mechanism, semi-naïve evaluation[1], is an efficient incremental algorithm.

We call it *incremental* as the work necessary to progress the computation is proportional to the changes induced in previous iterations. This has motivated much research[2, 3, 4, 5] attempting to pitch Datalog as the go-to venue for large-scale graph-like computations that are not well suited to SQL.

The core reasoning operation that enables reasoning at scale is called materialization, which is the act of incorporating the results of evaluating a Datalog program over some data, with the data itself.

As the referenced reasoners are bottom-up, materialisation is the computation of all entailments that follow from the input data with respect to the program. This is an attractive technique for read-intensive reasoning over large amounts of data because it completely eliminates the need for reasoning during runtime, reducing queries to lookups.

Efficiently maintaining this computation in face of both additions and deletions of data is a difficult challenge that is often tackled by combining different methods, such as using semi-naïve evaluation for additions and DRED[6, 7] or counting[8] for deletions.

Handling additions and deletions differently could result in uneven performance characteristics, potentially causing severe biases in runtime up to the point where adjusting a materialization could be more expensive than recomputing from scratch.

Moreover, maintaining the materialization while the *program* itself changes has been significantly less explored than the static scenario. This is especially important in situations where restarting the program might be too costly, but that temporarily adding or retracting rules is nonetheless a time critical requirement.

Datalog 2.0 2024: 5th International Workshop on the Resurgence of Datalog in Academia and Industry, October 11, 2024, Dallas, Texas, USA

*Corresponding author.
bruno98@ut.ee (B. R. C. A. d. Lima); merlin.kramer@d10.pw (M. Kramer); kalmer.apinis@ut.ee (K. Apinis); kkmicins@syr.edu (K. Micinski)
0000-0002-2679-6639 (B. R. C. A. d. Lima); 0009-0000-1413-0680 (M. Kramer); 0009-0006-2395-6584 (K. Apinis); 0000-0002-8650-0991 (K. Micinski)

We posit that a simple and potentially efficient way to tackle the materialization of *dynamic* programs is to define Datalog interpretation with an expressive language that has transparent incremental semantics, such as the recent DBSP[9] or its ancestor Differential Dataflow[10].

The value that these two bring is in allowing one to define non-incremental recursive programs that inherit their incremental semantics by exclusively using expressive declarative operators. We further argue that there is *significant* practical value in doing so, as both languages have very efficient interpreters[11, 12] written in the Rust programming language.

Contributions. Our contribution is in attempting to provide a practical and unified solution to handling incremental evaluation of programs where rules and facts change during runtime. This is realized over three tangible outputs:

- *A novel formulation of incremental interpretation as a higher-order streaming computation.* We formulate incremental evaluation as a DBSP circuit and then empirically evaluate whether the interpreter indeed inherited the incremental semantics of the host language by comparing it both with the DDLog[5] incremental reasoner, and with Soufflé as the recompute-from-scratch baseline.
- *A high-performance dynamic Datalog interpreter in Rust.* The implementation used for the benchmarks is fully open-source and lives on github[13]. We have thorough tests and examples that demonstrate how to use the library.
- *An accessible DBSP and dynamic Datalog interpreter in Python.* As DBSP's canonical implementation is a Rust program, it severely limits the ease of conducting research for as not only is it a novel theory, but much of its implementation intertwines high-performance computing code with its logic, often needing advanced Rust knowledge to use it. As we believe that there is much value in DBSP, we wrote a Python implementation from scratch[14] that also contains the dynamic Datalog interpretation circuit presented in this paper.

2. Related works

We consider Datalog engines to be relevant to this work if they are both bottom-up and their core purpose is either incremental materialisation, or in having had been notable ancestors of those. Soufflé being a well known engine is relevant because it is useful to gauge the overhead of supporting incremental updates in the evaluation section. A incremental reasoner will not provide value if it cannot compute updates faster than some other non-incremental reasoner can compute from scratch.

Reasoner	Kind	Fact Add	Fact Del	Rule Add	Rule Remove
Soufflé[15]	Both	No	No	No	No
BigDatalog[2]	Compiler	No	No	No	No
Cog[3]	Compiler	No	No	No	No
Nexus[4]	Compiler	Yes	No	No	No
DDLog[5]	Compiler	Yes	Yes	No	No
DBSP*	Compiler	Yes	Yes	No	No
DYRE	Interpreter	Yes	Yes	Yes	Yes

Table 1
Overview of the incremental behavior of relevant Datalog reasoners.

With Table 1 we situate the reasoning system described in this paper, the Dynamic Datalog Reasoner (DYRE), among other relevant reasoners.

BigDatalog is an extension of the Apache Spark[16] distributed batch computation framework. It provides a Datalog front-end that supports both recursive and non-recursive aggregation. While BigDatalog did satisfactorily handle large amounts of data, it required having to fork its underlying platform, as it not support any form of iteration, let alone cycles in its logical computational graph.

Cog is BigDatalog's successor project. It swaps Apache Spark for Apache Flink[17], a distributed streaming computation framework with first-class support for iteration and cyclic dataflows. The

highlight of this reasoner is that it showed how Flink's efficient delta iteration feature paved room for naturally implementing semi-naïve evaluation.

Both of these engines, while notable, do not support maintaining the materialization. Nexus is Cog's incremental follow-up, and it fully supports efficiently adjusting the computation to new data that is being streamed, including when the program contains recursive aggregates. It uses a provenance technique inspired by the counting method to keep track of all derivation paths. This is similar to how DBSP maintains computations.

The first Datalog engine that supports both incremental addition and deletion at scale is DDlog. The DD stands for Differential Dataflow[10], the language that inspired DBSP. Its ethos is the same, which is to provide a language for writing non-incremental programs that have incremental semantics.

The key component of both program provenance and incremental materialization is time. While in DD the time component can be an arbitrary lattice, DBSP restricts it to either being a total or product order, for use cases that require nested timestamps such as iteration. In our experiments we found that we did not need time to be a lattice other than product order.

DDLog compiles an expressive dialect of Datalog to non-incremental recursive relational algebra formulated with DD. In the seminal DBSP article[9] there is a section named "Implementing Recursive Datalog". In there, a blueprint is laid out for a system, referred to as DBSP* in the table, that is similar to DDLog in that it *compiles* a relational representation of the evaluation of a static Datalog program into a DBSP circuit.

Every reasoner referred to in the table compiles a *static* Datalog program to either some dialect of SQL or to a intermediate representation that models relational algebra. While this approach has proven itself to be successful, all of the underlying frameworks, sans DBSP and Differential Dataflow, were not developed with efficiently supporting iterative workloads in mind, as it is typical of Datalog.

Moreover, the compilation step is a significant bottleneck that all of these reasoners suffer from. For instance, DDLog takes minutes to compile a trivial transitive closure program. This makes it unusable for scenarios that require incrementally adding or removing rules, as that is impossible altogether.

This is not the case for the system proposed in this paper, where Datalog reasoning in itself is implemented as a circuit with both rules and facts being dynamic.

3. Revisiting datalog as a rewriting system

We model Datalog evaluation as a rewriting system in a similar vein to the expositions found in [18, 19, 20], and restrict it to the semantics of its positive fragment[1].

Syntax. A program is a finite set of horn clauses syntactically represented as implications, with definite clauses referred to as rules, and those with no positive literal as ground atoms.

$$\begin{aligned} \tau &::= \text{Const} \mid \text{Var} \\ a &::= P(\boldsymbol{\tau}) \\ g &::= P(\mathbf{Const}) \\ \pi &::= a \leftarrow \boldsymbol{a} \\ \Pi &::= \{\boldsymbol{\pi}\} \\ E &::= \{\mathbf{g}\} \end{aligned} \tag{1}$$

Equation 1 discerns predicates P, terms τ, rules π, atoms a, ground atoms g, programs Π, and finite fact stores E. Bold symbols represent ordered lists of symbols, that if keyed represent sets. Constants are quoted names, and variables are unquoted. The left-hand side of a rule is called *head*, and the right-hand *body*.

Evaluation. A substitution is a function σ that maps a single variable to a constant. A rewrite Σ is a set of substitutions such that no $\sigma \in \Sigma$ share a variable. We write $\mathbf{dom}(\Sigma)$ as the set of variables among all $\sigma \in \Sigma$ and $\Sigma[x]$ as the constant that the some variable x maps to.

Example 3.1 (Rewriting) *Applying the rewrite* $\Sigma : \{x \mapsto$ *"a"*, $y \mapsto$ *"b"*, $z \mapsto$ *"c"*$\}$ *to* $P(x, y)$ *grounds it to* $P($*"a"*, *"c"*$)$

If there is at least one rewrite that has one substitution for each variable term in some atom a, applying it to a will ground it.

Definition 3.1 *The rewrite set monoid is a triple* $\langle \mathbb{\Sigma}, \oplus, \Sigma_0 \rangle$ *where:*

- $\mathbb{\Sigma}$ *is a set of rewrites* Σ
- *The binary operation* $\Sigma_a \oplus \Sigma_b = \Sigma_c$ *for any* $\Sigma_a, \Sigma_b \in \mathbb{\Sigma}$*, with* $\textbf{\textit{dom}}(\Sigma_c) = \textbf{\textit{dom}}(\Sigma_a) \cup \textbf{\textit{dom}}(\Sigma_b)$ *and each* $x \in \textbf{\textit{dom}}(\Sigma_c)$ *maps to* $\Sigma_a[x]$ *if* $x \in \textbf{\textit{dom}}(\Sigma_a)$ *or* $\Sigma_b[x]$ *otherwise*
- *The identity element is* Σ_0*, the empty rewrite*

Grounding the head of a rule is done by iteratively building at least one Σ that satisfies the constraints of the body. Let $\mathbf{x}$ be the list of terms x of some atom a with predicate P. We use the syntax $E.P$ to refer to the set of ground atoms in E with a's predicate.

Definition 3.2 *The rewrite product* $\mathbb{\Sigma} \otimes a = \mathbb{\Sigma}'$ *of a rewrite set* $\mathbb{\Sigma}$ *and some atom* a*, with the number of variable terms in* a *being* k*, is the set of rewrites such that for each* $\Sigma \in \mathbb{\Sigma}$*, up to one new rewrite* $\Sigma' = \Sigma \oplus \Sigma_f$ *is created for each* $f \in E.P$*, with potentially up to* k *new* σ*.*

Example 3.2 (Rewrite product)

Given:

$a = P(x, y)$

$E.P = \{($*"a"*, *"b"*$), ($*"b"*, *"c"*$), ($*"c"*, *"d"*$)\}$

$\mathbb{\Sigma} = \{\{x \mapsto$ *"a"*$\}, \{x \mapsto$ *"b"*$\}\}$

Then:

$\mathbb{\Sigma} \oplus a = \{\{x \mapsto$ *"a"*$, y \mapsto$ *"b"*$\}, \{x \mapsto$ *"b"*$, y \mapsto$ *"c"*$\}\}$

There are two steps to the product. The first is *unification*, where each $f \in E.P$ is unified against all $\Sigma \in \mathbb{\Sigma}$ applied to a. f is said to *unify* with a if there is no constant term c in a that occupies a position in f where there is some other term c' such that $c \neq c'$.

The second is *extension*. For all $\langle \Sigma, a, f \rangle$ triples where $f \in E.P$ unifies with some Σ applied to some a, there is one new rewrite Σ_f where each variable x maps to some constant c in f such that $x \notin \Sigma$, $x \in a$, and x occupies the same position as c. The output of the product is $\Sigma \oplus \Sigma_f$ for each triple.

Let $\mathbb{\Sigma}_0$ be the empty rewrite set. It contains the empty rewrite Σ_0. The immediate consequence $T(\pi, E)$ of some rule π with a ground atom set E is computed by applying each rewrite from the final rewrite product $\mathbb{\Sigma}_0 \otimes a_0 \otimes ... \otimes a_n$ to a, with a being the head, and all other atoms members of the body.

The immediate consequence $\mathcal{T}(\Pi, E)$ of the program Π with $\pi_i \in \Pi$ is then $\bigcup_{i=0} T(\pi_i, E)$. Program evaluation is then defined as the least fixed point (LFP) of the immediate consequence of the program.

4. Datalog interpretation as a DBSP circuit

While we do not assume the reader to be familiar with DBSP as we introduce its core tenets alongside our contributions, we expect for there to be some familiarity with incremental view maintenance[21], as that is what DBSP formalizes at a high level of expressivity. For further clarification, we recommend the seminal DBSP paper[9].

The central element of DBSP is the Stream. Streams S are maps $\mathbb{N} \mapsto A$ with the domain representing time, and the codomain a value from an Abelian group A. The restriction imposed on A makes it unsuitable for Datalog evaluation as presented, as set union, what ensures that at some point the fixed

point is reached, does not form an Abelian group. We address this by modelling E and Π as $\mathbb{Z}$-sets[22] instead.

Definition 4.1 *$\mathbb{Z}$-sets are maps with each domain element having an associated integer count w. For any two $\mathbb{Z}$-sets $\mathbb{Z}_K$ and $\mathbb{Z}'_K$ of the same underlying set K, the following hold:*

- $\mathbb{Z}''_K = \mathbb{Z}_K + \mathbb{Z}'_K$ *where* $i \in dom(\mathbb{Z}''_K)$ *iff* $\mathbb{Z}_K[i] + \mathbb{Z}'_K[i] \neq 0$ *and* $\mathbb{Z}''_K[i] = \mathbb{Z}_K[i] + \mathbb{Z}'_K[i]$
- $\mathbb{Z}'_K = -\mathbb{Z}_K$ *where* $\forall i \in dom(\mathbb{Z}_K), \mathbb{Z}'_K[i] = -\mathbb{Z}_K[i]$
- *If* $\forall i \in dom(\mathbb{Z}_K), \mathbb{Z}_K[i] = 1$, *then* $\mathbb{Z}_K$ *is bijective to some set.*

We name $\mathbb{Z}$-sets where all values in their codomain are either 1 or -1 as $\Delta\mathbb{Z}$ and those that are bijective to sets as Z, and make use of the two auxiliary functions toset : $\mathbb{Z}_A \mapsto Z_A$, and todiff : $\mathbb{Z}_A \mapsto \Delta\mathbb{Z}_A$ that ensure this conversion.

Functions are maps $A_0 \times ... \times A_n \mapsto B$ where A and B are Abelian groups. An *operator* is also a map $S_{A_0} \times ... \times S_{A_n} \mapsto S_B$ where each S_A and S_B are streams. *Lift*-ing some function $f : A \mapsto B$ yields an operator $\uparrow F : S_A \mapsto S_B$ where $\forall t, S_B[t] = f(S_A[t])$. We use $\uparrow$ to denote lifted operators.

An operator is said to be *causal* if its output at time t only ever relies on all inputs with time t' such that $t' \leq t$, with it being *strict* if $t' < t$. All lifted operators are causal, but not necessarily strict. All strict operators are guaranteed[9] to have a unique solution to their fixpoint. We do not utilise any non-causal operators. Lifting $\mathcal{T}$ yields a causal operator.

The two primitive building blocks of circuits are lifting and delaying. The delay operator $z_A^{-1} : S_A \mapsto S'_A$ produces a stream S'_A such that $S'_A[0]$ is the Abelian identity element, and $S'_A[t] = S_A[t-1]$ otherwise. As per [9, Lemma 2.11], feeding the output of some causal operator $R : S_A \times S_B \mapsto S_B$ back to itself with z_B^{-1} yields a strict operator.

Example 4.1 (Delay, *toset* and *todiff*)

Given:

$S_{Z_E} = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1\}, 1 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 2\}, 2 \mapsto \{P(\text{"c"}, \text{"d"}) \mapsto 1\}]$

$S'_{Z_E} = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto -1\}, 1 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 2\}, 2 \mapsto \{P(\text{"c"}, \text{"d"}) \mapsto -1\}]$

$S''_{Z_E} = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1\}, 1 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 2\}]$

Then:

$\uparrow toset(S_{Z_E}) = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1\}, 1 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 1\}, 2 \mapsto \{P(\text{"c"}, \text{"d"}) \mapsto 1\}]$

$\uparrow todiff(S'_{Z_E}) = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto -1\}, 1 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 1\}, 2 \mapsto \{P(\text{"c"}, \text{"d"}) \mapsto -1\}]$

$z_E^{-1}(S''_{Z_E}) = [0 \mapsto \{\}, 1 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1\}, 2 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 2\}]$

4.1. naïve dynamic interpretation

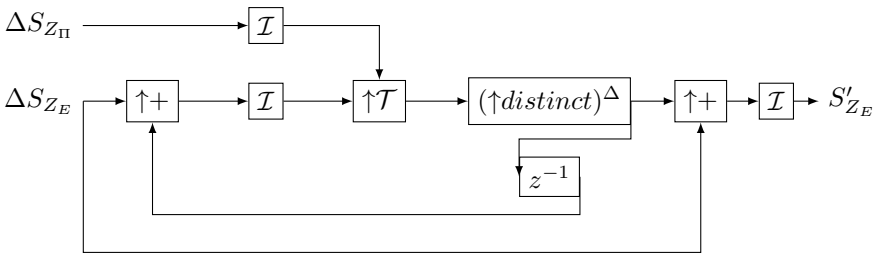

Figure 1: naïve materialization of Dynamic Datalog programs as a DBSP circuit

Let $\mathcal{I} : S_A \mapsto S'_A$ be the stream integration operator: $\forall t, S'_A[t] = \sum_{i=-1}^{t} S_A[i]$, and $(\uparrow \textit{distinct})^\Delta : S_{\mathbb{Z}_A} \mapsto S'_{\Delta\mathbb{Z}_A}$ the stateful $\mathbb{Z}$-set distinct operator: $\forall t, S'_{Z_A}[t] = \textit{todiff}(S_{Z_A}[t] - z^{-1}(\mathcal{I}(S_{Z_A}))[t])$.

With S_{Z_Π} as a stream of bijective program $\mathbb{Z}$-sets and S_{Z_E} of ground atoms, we define the lifted immediate consequence stream operator $\uparrow\mathcal{T} : S_{Z_\Pi} \times S_{Z_E} \mapsto S'_{\mathbb{Z}_E}$ as $\forall t, S'_{\mathbb{Z}_E}[t] = \mathcal{T}(S_{Z_\Pi}[t], S_{Z_E}[t])$.

Figure 1 showcases the circuit of naïve dynamic Datalog interpretation. The value of $S'_{Z_E}[t]$ for any t will be the addition of $\sum_{i=0}^{t} S_{Z_E}[i]$ to the sum of all inferred distinct ground facts by then.

The fixed point of the circuit, after either a new fact or program $\mathbb{Z}$-set is received, will happen at some timestamp t where the z^{-1} operator returns the identity of Z_E, since then no new distinct atoms can be emitted.

The value of $S'_{Z_E}[t]$ will be the full materialization. As the input S_{Z_E} and the final integration are bijective to sets, the materialization will too be bijective, therefore emitting sets and matching regular Datalog semantics.

Example 4.2 (Integration and non-incremental dynamic interpretation)

Given:

$\Delta S_{Z_E} = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1\}, 1 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 1\}, 2 \mapsto \{P(\text{"c"}, \text{"d"}) \mapsto 1\}]$

$\Delta S_{Z_\Pi}[0] = \{R(x, y) \leftarrow P(x, y) \mapsto 1\}$

$\Delta S_{Z_\Pi}[1] = \{R(x, z) \leftarrow R(x, y), P(y, z) \mapsto 1\}$

$\Delta S_{Z_\Pi}[2] = \{R(x, y) \leftarrow P(x, y) \mapsto -1\}]$

Then:

$\mathcal{I}(S_{Z_E})[0] = \{P(\text{"a"}, \text{"b"}) \mapsto 1\}$

$\mathcal{I}(S_{Z_E})[1] = \{P(\text{"a"}, \text{"b"}) \mapsto 1, P(\text{"b"}, \text{"c"}) \mapsto 2\}$

$\mathcal{I}(S_{Z_E})[2] = \{P(\text{"a"}, \text{"b"}) \mapsto 1, P(\text{"b"}, \text{"c"}) \mapsto 2, P(\text{"c"}, \text{"d"}) \mapsto 1\}$

$S'_{Z_E}[0] = \{P(\text{"a"}, \text{"b"}) \mapsto 1, R(\text{"a"}, \text{"b"}) \mapsto 1\}$

$S'_{Z_E}[1] = S'_{Z_E}[0] + \{P(\text{"b"}, \text{"c"}) \mapsto 1, R(\text{"b"}, \text{"c"}) \mapsto 1, R(\text{"a"}, \text{"c"}) \mapsto 1\}$

$S'_{Z_E}[2] = S'_{Z_E}[1] + \{R(\text{"a"}, \text{"b"}) \mapsto -1, R(\text{"b"}, \text{"c"}) \mapsto -1, R(\text{"a"}, \text{"c"}) \mapsto -1, P(\text{"c"}, \text{"d"}) \mapsto 1\}$

4.2. Incremental dynamic interpretation

To derive incremental evaluation it is necessary to reason about the linearity of functions. A function $f : A \mapsto B$ is *linear* if for any $a, b \in A$, $f(a + b) = f(a) + f(b)$. This extends to functions with more than one argument e.g $f : A \times B \mapsto C$ is bilinear if it is linear in each of its arguments.

While operators obtained by lifting linear functions are linear, $\mathcal{T}$ is not bilinear hence the general formula for their incrementalization[9, Theorem 1.4] when applied to it yields:

$$\forall t, (\uparrow\mathcal{T})^\Delta[t] = \mathcal{T}(\Delta S_{Z_\Pi}[t], \Delta S_{Z_E}[t]) + \mathcal{T}(z^{-1}(\mathcal{I}(\Delta S_{Z_\Pi}))[t], \Delta S_{Z_E}[t]) + \mathcal{T}(\Delta S_{Z_\Pi}[t], z^{-1}(\mathcal{I}(\Delta S_{Z_E}))[t])$$

Will not satisfy the invariant $\forall t, \uparrow\mathcal{T}(\mathcal{I}(\Delta S_{Z_\Pi}), \mathcal{I}(\Delta S_{Z_E}))[t] = \mathcal{I}((\uparrow\mathcal{T})^\Delta(\Delta S_{Z_\Pi}, \Delta S_{Z_E}))[t]$ therefore being unsound.

Example 4.3 (Non-linearity of $\uparrow\mathcal{T}$)

Given:

$\Delta S_{Z_E} = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1\}, 1 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 1\}\}]$

$\Delta S_{Z_\Pi} = [0 \mapsto \{R(x, y) \leftarrow P(x, y) \mapsto 1\}, 1 \mapsto \{R(x, z) \leftarrow R(x, y), P(y, z) \mapsto 1\}]$

$\uparrow\mathcal{T}(\Delta S_{Z_\Pi}, \Delta S_{Z_E})[0] = \{R(\text{"a"}, \text{"b"}) \mapsto 1\}$

$\uparrow\mathcal{T}(\Delta S_{Z_\Pi}, \Delta S_{Z_E})[1] = \{\}$
$\uparrow\mathcal{T}(\mathcal{I}(\Delta S_{Z_\Pi}), \mathcal{I}(\Delta S_{Z_E}))[0] = \{R(\text{"a"}, \text{"b"}) \mapsto 1\}$
$\uparrow\mathcal{T}(\mathcal{I}(\Delta S_{Z_\Pi}), \mathcal{I}(\Delta S_{Z_E}))[1] = \{R(\text{"a"}, \text{"b"}) \mapsto 1, R(\text{"b"}, \text{"c"}) \mapsto 1, R(\text{"a"}, \text{"c"}) \mapsto 1\}$
Then:
$(\uparrow\mathcal{T})^\Delta(\Delta S_{Z_\Pi}, \Delta S_{Z_E})[0] = \{R(\text{"a"}, \text{"b"}) \mapsto 1\} + \{\} + \{\}$
$(\uparrow\mathcal{T})^\Delta(\Delta S_{Z_\Pi}, \Delta S_{Z_E})[1] = \{\} + \{R(\text{"b"}, \text{"c"}) \mapsto 1\} + \{\}$
Therefore:
$\mathcal{I}((\uparrow\mathcal{T})^\Delta(\Delta S_{Z_\Pi}, \Delta S_{Z_E}))[1] \neq \uparrow\mathcal{T}(\mathcal{I}(\Delta S_{Z_\Pi}), \mathcal{I}(\Delta S_{Z_E}))[1]$

We aim to achieve incremental evaluation by creating a circuit that implements the previously introduced rewriting system exclusively with linear and bilinear operators.

In order to iteratively compute rewrite products, there have to be notions of *direction* and *end*, as rewrites need to be both propagated forward to meet other atoms and ultimately signalled to be ready for grounding, to generate new facts.

Definition 4.2 *The expected provenance $\mathcal{P} : \pi \times a \mapsto \mathbb{N}$ of the j-th atom a_j with respect to some rule π_i is $\sum_{k=0}^{j} \mathcal{H}(a_k)$, where $\mathcal{H}(a_k)$ is some function that returns an integer such that if two a_x, a_y atoms have the same predicate and terms, then $\mathcal{H}(a_x) = \mathcal{H}(a_y)$*

Definition 4.3 *The direction $\mathbb{Z}$-set $\mathbb{Z}_\mathcal{D}$ of some rule π with i atoms and weight k is $\{\langle\mathcal{P}(\pi, a_{j-0}), \mathcal{P}(\pi, a_j)\rangle \mapsto k \mid \forall j < i \wedge j \geq 1\} + \{\langle 0, \mathcal{P}(\pi, a_0)\rangle \mapsto k\}$. The function dir : $\pi \mapsto \mathbb{Z}_\mathcal{D}$ computes the direction set for some rule π.*

Definition 4.4 *The grounding signal $\mathcal{S}$ of some rule π with i atoms is $\langle\mathcal{P}(\pi, a_i), \pi.h\rangle$ with a_i being the last atom, and $\pi.h$ the rule head. The function sig : $\pi \mapsto \mathcal{S}$ computes the grounding signal for some rule π*

Definition 4.5 *The grounding kit $\mathbb{K}$ of some program Π and its π_i rules with k_i respective weights is a tuple $\langle\mathbb{Z}_\mathcal{D}, \mathbb{Z}_\mathcal{S}\rangle$ where:*

- *$\mathbb{Z}_D$ is $\sum_{j=0}^{j}$ dir(π_j)*
- *$\mathbb{Z}_S$ is $\{sig(\pi_j) \mapsto k_j \mid \forall j < i\}$*

*The function kit : $\Pi \mapsto \mathbb{K}$ computes the grounding kit for some program Π, with **dir** : $\Pi \mapsto \mathbb{Z}_\mathcal{D}$ and **sig** : $\Pi \mapsto \mathcal{S}$ returning, respectively, $\mathbb{Z}_D$ and $\mathbb{Z}_S$ of some program's $\mathbb{K}$.*

Example 4.4 (The grounding kit of some program)

Given:
$\pi = R(x, y) \leftarrow P(x, y)$
$\pi' = R(x, z) \leftarrow P(x, y), R(y, z)$
$Z_\Pi = \{\pi \mapsto 1, \pi' \mapsto 1\}$
Then:
$\mathcal{P}(\pi, P(x, y)) = \mathcal{H}(P(x, y))$
$\mathcal{P}(\pi', P(x, y)) = \mathcal{H}(P(x, y))$
$\mathcal{P}(\pi', R(y, z)) = \mathcal{H}(P(x, y)) + \mathcal{H}(R(y, z))$
$\textbf{\textit{dir}}(Z_\Pi) = \{\mathcal{H}(P(x, y)) \mapsto 2, \mathcal{P}(\pi', R(y, z)) \mapsto 1\}$
$\textbf{\textit{sig}}(Z_\Pi) = \{\langle\mathcal{H}(P(x, y)), R(x, y)\rangle \mapsto 1, \langle\mathcal{P}(\pi', R(y, z)), R(x, z)\rangle \mapsto 1\}$
$kit(Z_\Pi) = \langle\textbf{\textit{dir}}(Z_\Pi), \textbf{\textit{sig}}(Z_\Pi)\rangle$

We leverage the grounding kit to guide interpretation and implement the rewrite product as three incremental joins.

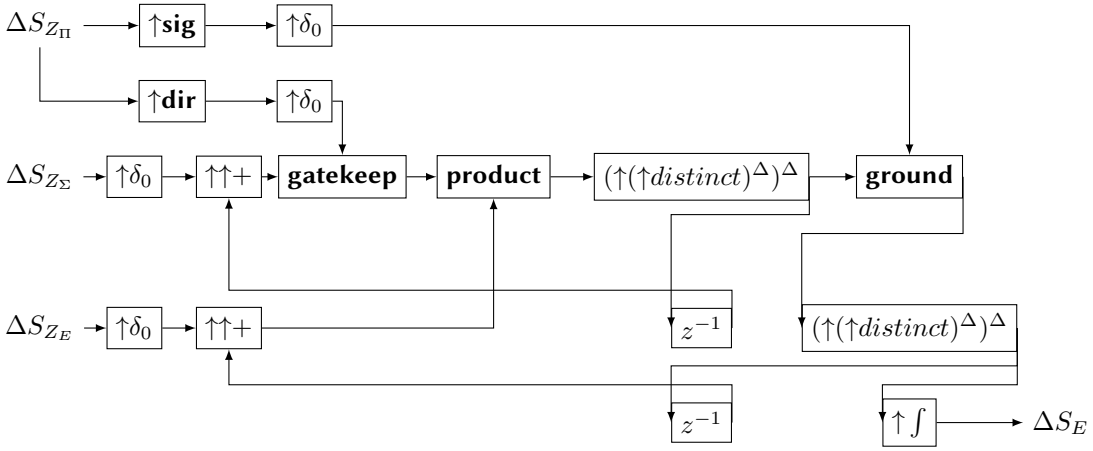

Figure 2: Incremental interpretation over streams of facts and programs

Figure 2 depicts the full circuit of the term rewriting system. To incrementally maintain the fixed point computation once updates arrive, it is necessary to use nested streams.

A nested stream S_{S_A} is a stream where at each timestamp t there is another stream. Doubly-lifting some function, or lifting an operator, yields an operator over streams of streams. Let $f : A \mapsto B$ be some scalar, $\uparrow\uparrow f : S_{S_A} \mapsto S_{S_B}, \forall t, t', (S_{S_B}[t])[t'] = f((S_{S_A}[t])[t'])$

Streams themselves are Abelian through the lifting of their group operations. The $\uparrow\delta_0$ operator yields a stream of streams from a stream.

Let $\delta_0 : A \mapsto S_A, S_A[0] = A, \forall t, S_A[t] = 0$ be the stream introduction operator, and $\int : S_A \mapsto A, A = \sum_{i=0}^{t} S_A[i]$ where t is the first timestamp such that $S_A = A_0$. We use the $\uparrow\delta_0$ operator on both input streams ΔS_{Z_Π} and ΔS_{Z_E}, and in the rewrite stream ΔS_{Z_Σ}. The rewrite stream never receives any updates, and only contains the $\mathbb{Z}_0$ set.

Example 4.5 (Nested streams and stream introduction and elimination)

Given:

$\Delta S_{Z_E} = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1\}, 1 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 1\}]$

Then:

$\delta_0(\Delta S_{Z_E}[0]) = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1\}]$

$\uparrow\delta_0(\Delta S_{Z_E}) = [0 \mapsto [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1\}], 1 \mapsto [0 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 1\}]]$

$\int(\Delta S_{Z_E}) = \{P(\text{"a"}, \text{"b"}) \mapsto 1, P(\text{"b"}, \text{"c"}) \mapsto 1\}$

$\uparrow\int(\uparrow\delta_0(\Delta S_{Z_E})) = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1\}, 1 \mapsto \{P(\text{"b"}, \text{"c"}) \mapsto 1\}\}]$

$\int(\uparrow\delta_0(\Delta S_{Z_E})) = [0 \mapsto \{P(\text{"a"}, \text{"b"}) \mapsto 1, P(\text{"b"}, \text{"c"}) \mapsto 1\}]$

We refer to t as the *outer* timestamp of some stream of streams S_{S_A} and t' as the inner one. t can be interpreted as the time of arrival of some update, and t' as the fixed point iteration. As the materialization nonetheless expects the immediate consequence to be a single $\mathbb{Z}$-set, the $\uparrow\int$ operator is used to "flatten" it into one $\mathbb{Z}$-set.

gatekeep, *product* and *ground* are the three nested stream operators that implement the rewrite product. All of them have the following form:

$$op : S_{S_{\mathbb{Z}_A}} \times S'_{S_{\mathbb{Z}_B}} \mapsto S''_{S_{\mathbb{Z}_C}}, \forall t, S''_{S_{\mathbb{Z}_C}}[t] = \uparrow\uparrow\pi_f((\uparrow(\uparrow \bowtie_p)^\Delta)^\Delta(S_{S_{\mathbb{Z}_A}}, S'_{S_{\mathbb{Z}_B}}))[t]$$

Where π_f is the $\mathbb{Z}$-set projection with scalar function f, and $\bowtie_p$ the $\mathbb{Z}$-set join with boolean join comparison scalar function p. As π_f is linear, by definition it already is incremental. For some stream S_{S_A}, the time complexity of $\uparrow\uparrow\pi_f(S_{S_A})$ at some t, t' is $O(f(((S_{S_A})[t])[t']))$[9].

The relational $\bowtie$ operator is bilinear, with the time complexity of its incremental form $(\uparrow(\uparrow \bowtie_p)^{\Delta})^{\Delta}$ with respect to its inputs at some t, t' being $O(\|\uparrow\mathcal{I}(S_{S_{\mathbb{Z}_A}})[t][t']\| \times \|\mathcal{I}(S_{S_{\mathbb{Z}_B}})[t][t']\|)$.

Example 4.6 (A parallel with semi-naïve evaluation of static programs) *With t' denoting the fixed point iteration number, at each new data arrival t a $\uparrow\mathcal{I}$ operator applied to the left side of the incremental join would emit a stream where the value at t' is $\sum_{i=0}^{t'}((S_{S_{\mathbb{Z}_A}})[t])[i]$. For the right side on the other hand, it would be $\sum_{i=0}^{t}((S_{S_{\mathbb{Z}_B}})[i])[t']$ instead as $\mathcal{I}$ is not lifted. A parallel can be made with the regular non-incremental semi-naïve evaluation of a static program, where the left side takes as input the most recently inferred facts and the right side all.*

$$\textit{gatekeep} : S_{S_{\mathbb{Z}_{\langle \mathbb{N},\Sigma\rangle}}} \times S'_{S_{\mathbb{Z}_{\mathcal{D}}}} \mapsto S''_{S_{\mathbb{Z}_{\langle a,\Sigma,\mathbb{N}\rangle}}}, \forall t, S''_{S_{\mathbb{Z}_{\langle a,\Sigma,\mathbb{N}\rangle}}}[t] = \uparrow\uparrow\pi_f((\uparrow(\uparrow \bowtie_p)^{\Delta})^{\Delta}(S_{S_{\mathbb{Z}_{\langle \mathbb{N},\Sigma\rangle}}}, S'_{S_{\mathbb{Z}_{\mathcal{D}}}}))[t]$$

Each iteration is initiated through operator *gatekeep*. Its inputs are a delayed stream of streams of $\mathbb{Z}$-sets that contain tuples where the first element is the $\mathbb{Z}$-set of *expected provenances* of some atom alongside a rewrite set $\mathbb{\Sigma}$ and a stream of directions.

The output is the Join of the delayed provenance-indexed rewrites with directions. The join predicate p is the equality between the provenance of the left side with the "previous" provenance of matching directions.

The projection function f outputs tuples that in push Σ as candidate rewrites for the next iteration. This is achieved by emitting tuples where candidate rewrites have their "next" provenance b alongside some atom a such that $\mathcal{P}(\pi, a) = b$ and π is any rule. This operator is proportional to the product of the previous iteration rewrites and all directions.

$$\textit{product} : S_{S_{\mathbb{Z}_{\langle a,\Sigma,\mathbb{N}\rangle}}} \times S'_{S_{\mathbb{Z}_E}} \mapsto S''_{S_{\mathbb{Z}_{\langle \mathbb{N},\Sigma\rangle}}}, \forall t, S''_{S_{\mathbb{Z}_{\langle \mathbb{N},\Sigma\rangle}}}[t] = \uparrow\uparrow\pi((\uparrow(\uparrow \bowtie)^{\Delta})^{\Delta}(S_{S_{\mathbb{Z}_{\langle a,\Sigma,\mathbb{N}\rangle}}}, S'_{S_{\mathbb{Z}_E}}))[t] \quad (2)$$

Operator *product* implements the core of the rewrite product, with its inputs being the output of *gatekeep* and the delayed stream of ground atoms. Let $l : \langle a, \Sigma, \mathbb{N}\rangle$ be the left side and g the right, p is true if Σ applied to a unifies with g returning Σ_f. The projection f then simply yields $\Sigma \oplus \Sigma_f$ and $\mathbb{N}$. This operator is then proportional to the product of the rewrites propagated during this iteration with all ground atoms.

$$\textit{ground} : S_{S_{\mathbb{Z}_{\langle \mathbb{N},\Sigma\rangle}}} \times S'_{S_{\mathbb{Z}_{\mathcal{S}}}} \mapsto S''_{S_{\mathbb{Z}_E}}, \forall t, S''_{S_{\mathbb{Z}_E}}[t] = \uparrow\uparrow\pi_f((\uparrow(\uparrow \bowtie_p)^{\Delta})^{\Delta}(S_{S_{\mathbb{Z}_{\langle \mathbb{N},\Sigma\rangle}}}, S'_{S_{\mathbb{Z}_{\mathcal{S}}}}))[t] \quad (3)$$

The last operation is grounding itself, where the possibly-final rewrites stemming from *product* are joined with grounding signals. The output of this operator will be all rewrites that have been propagated beyond the last body atom, hence being applicable to the head of their respective rules to generate new ground atoms. p is true if the provenance of the left side matches some found in a grounding signal. f is the application of the rewrite to the head atom contained in the signal tuple.

4.3. A simple indexing scheme to improve performance

While the proposed circuit has been demonstrated to be incremental in both programs and ground atoms, operator *product* is inefficient because in each iteration it needlessly attempts to unify every single rewrite with all facts that share the same predicate. This implies that both the best and worst case complexities of its join with respect to ground atoms are the same.

To make our circuit practical, we devise a way to make its join key more fine-grained by also taking into account constant term combinations. We implemented a simplified version of the ground atom indexing method that Souffle has pioneered[23], but with incremental DBSP operators. This means that this scheme will respond to change in the same way as the rest of the circuit.

Definition 4.6 *The naïve join order $\mathbb{Z}$-set $\mathbb{Z}_J$ of some rule π with weight k and i atoms a with predicates $P.a$ in its body is a $\mathbb{Z}$-set of tuples $\{\langle P.a_0, m_0\rangle \mapsto k, ..., \langle P.a_{i-1}, m_{i-1}\rangle \mapsto k\}$ where m_j for $j < i$ are terms in a_j that are in the same place as some other terms in a_{j+1}. For simplicity of exposition, we assume that the respective body atoms are ordered in a way that ensures that every a_j is before some other a_{j+1} with variable terms that overlap. The function jorder : $\pi \mapsto \mathbb{Z}_J$ computes the naïve join order for some rule π.*

Example 4.7 (The naïve join order of some rule) *Given $\pi = T(x, y) \leftarrow E(x, y), T(y, z)$ with $k = 1$, then jorder(π) $= \{\langle E, ()\rangle \mapsto 1, \langle T, 0\rangle \mapsto 1\}$. The first tuple indicates that for the predicate E there should be no index. The second on the other hand makes it explicit that ground atoms in T ought to be indexed on their zeroth term.*

Definition 4.7 *The extended grounding kit $\mathbb{Q}$ of some program Π and its i rules π is a triple $\langle \mathbb{Z}_\mathcal{D}, \mathbb{Z}_\mathcal{S}, \mathbb{Z}_\mathcal{J}\rangle$ where: $\mathbb{Z}_D$ and $\mathbb{Z}_S$ are the same as in $\mathbb{K}$, and $\mathbb{Z}_\mathcal{J}$ is $\sum_{j=0}^{i}$ jorder(π_j). The function ekit : $\Pi \mapsto \mathbb{Q}$ computes the extended grounding kit for some program Π, and **jorder** : $\Pi \mapsto \mathbb{Z}_\mathcal{J}$ returns some Π's join order $\mathbb{Z}$-set.*

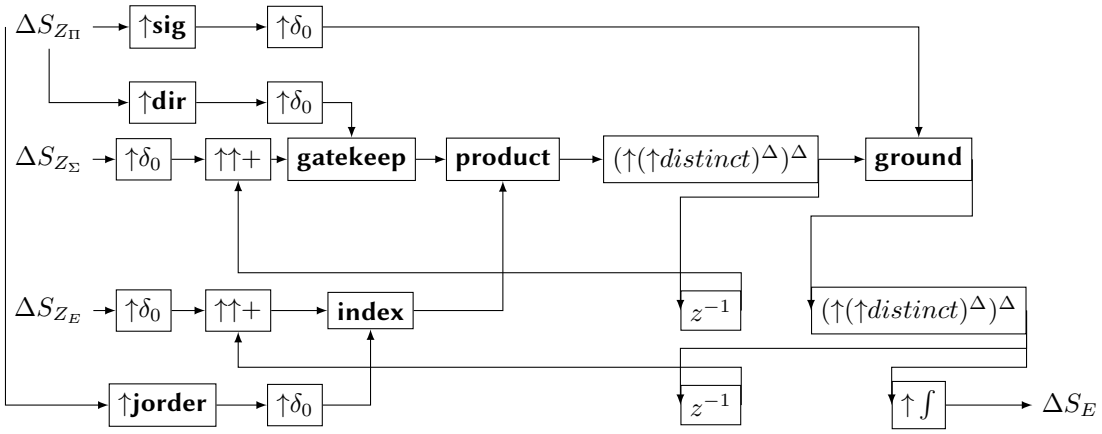

Figure 3: Incremental interpretation over streams of indexed facts and programs

As Figure 3 shows, it is only necessary to make a few changes to the circuit depicted on Figure 2 to extend it with this incremental indexing mechanism.

First, one new operator must be applied to $\Delta S_{\mathbb{Z}_\Pi}$, ↑**jorder**, and *product*'s predicate needs to be shifted to join over predicates and constant terms i.e indexed facts. This is enabled by adding a new operator after the distinct operator that follows the sum of recently arrived ground atoms and *ground*.

$$\textit{index} : S_{S_{\mathbb{Z}_E}} \times S'_{S_{\mathbb{Z}_\mathcal{J}}} \mapsto S''_{S_{\mathbb{Z}_{\langle E,E\rangle}}}, \forall t, S''_{S_{\mathbb{Z}_{\langle E,E\rangle}}}[t] = \uparrow\uparrow\pi_f((\uparrow(\uparrow \bowtie_p)^\Delta)^\Delta(S_{S_{\mathbb{Z}_E}}, S'_{S_{\mathbb{Z}_\mathcal{J}}}))[t] \quad (4)$$

The *index* operator receives as input ground atoms and join orders. The join key function p is true if some atom's predicate matches some order's predicate. Given some matching order j and atom a, f will output a tuple $\langle a', a\rangle$ containing partial atom a' according to j and a.

This will ensure that while the right side is still proportional to all ground terms of some predicate k, if there is only one match according to the order j, then the best-case complexity decreases to being a scan over only that single match. Over the next section we demonstrate how this leads to enormous performance gains, at a seemingly minimal increase in memory usage.

5. Evaluation

There are three parts to this section. The first one pertains to comparing the materialisation performance of the circuit with indexing, versus without. To give a point of reference, Soufflé was put through the same benchmark in interpreted mode.

The second part aims to compare DYRE's incremental behavior with the reasoner that is the closest to it, DDLog. This is the most important empirical result of the paper, as this would indicate that the incremental semantics were indeed inherited, with the additional benefit of the program being dynamic.

As incremental systems must keep track of more data than non-incremental ones, the last part is dedicated to providing an overview of the peak memory usage witnessed across all benchmarks.

Datasets. We choose to strictly use three well-known synthetic datasets for performance evaluation. The first dataset is **LUBM**[24], a standard benchmark for the entailment of description logic related formalisms such as RDFS[25] and OWL2RL[26]. For all of these, the data has two components, the ontology, that determines with which predicates can individuals be described, and the descriptions of the individuals themselves.

$$
\begin{aligned}
&T(y, \text{"rdf:type"}, x) \leftarrow T(a, \text{"rdfs:domain"}, x), T(y, a, z) \\
&T(z, \text{"rdf:type"}, x) \leftarrow T(a, \text{"rdfs:range"}, x), T(y, a, z) \\
&T(x, \text{"rdfs:subPropertyOf"}, z) \leftarrow T(x, \text{"rdfs:subPropertyOf"}, y), T(y, \text{"rdfs:subPropertyOf"}, z) \\
&T(x, \text{"rdfs:subClassOf"}, z) \leftarrow T(x, \text{"rdfs:subClassOf"}, y), T(y, \text{"rdfs:subClassOf"}, z) \\
&T(z, \text{"rdf:type"}, y) \leftarrow T(x, \text{"rdfs:subClassOf"}, y), T(z, \text{"rdf:type"}, x) \\
&T(x, b, y) \leftarrow T(a, \text{"rdfs:subPropertyOf"}, b), T(x, a, y)
\end{aligned}
\tag{5}
$$

The core of RDFS entailment can be modelled as the program of approximately 6 mutually recursive rules contained in equation 5.

$$
\begin{aligned}
&\text{degreeFrom}(x, y) \leftarrow \text{hasAlumnus}(y, x) \\
&\ldots \\
&\text{Faculty}(x) \leftarrow \text{PostDoc}(x) \\
&\ldots \\
&\text{Employee}(x) \leftarrow \text{Person}(x), \text{worksFor}(x, y), \text{Organization}(y) \\
&\ldots\text{(95 more rules)}
\end{aligned}
\tag{6}
$$

The OWL2RL entailment program, in equation 6, is done by converting the description logic entailments of LUBM's ontology to Datalog according to [27]. The final program has almost 100 rules.

The second and third datasets are graphs generated with the GT[28] tool. The second, **RAND1K**, is a dense graph of one thousand edges. Each edge has around 1% chance of being connected to each other. The third graph, **RMAT1K**, has ten thousand edges and one thousand nodes. It is a sparse graph that follows an inverse power-law distribution, with most nodes being only connected to a handful of other nodes.

$$
\begin{aligned}
&T(x, y) \leftarrow E(x, y) \\
&T(x, y) \leftarrow E(x, y), T(y, z)
\end{aligned}
\tag{7}
$$

We only run one program over these two datasets, the ubiquitous transitive closure one in equation 7.

All benchmarks used the same AWS-provisioned M1 Mac Pro machine. No other processes were running during the evaluation. Each measurement was either taken fifty times, or up to until there wasn't noticeable variance. Runtime performance was recorded with each reasoner's own profiler, while peak memory usage was tracked with GNU Time.

5.1. The impact of indexing

The first benchmark set is shown on Figure 4. The X axis shows the percentage of the total data that is being materialized according to the respective program and dataset, indicated by the title of the facet of each subgraph. The Y axis shows the time in milliseconds taken to materialise the program. DYRE^I refers to DYRE with indexing, and its counterpart is referred to without the superscript.

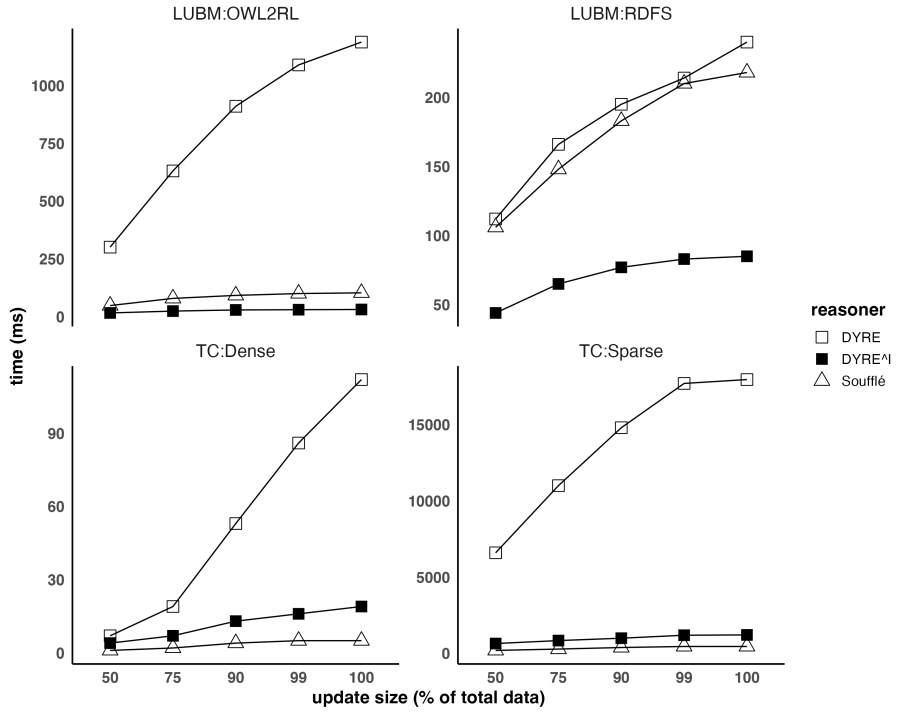

Figure 4: The performance of DYRE, its indexed variant, and Soufflé, under increasingly larger amounts of non-incremental updates

In two out of the four graphs, Soufflé is faster than both versions of DYRE. In TC:Dense, Soufflé is three times faster than DYRE with indexing, and twenty times otherwise. The same ratio holds true in the sparse graph. As both of these scenarios require imply fixpoint iterations, they offer a good proxy to evaluate the impact of DBSP's provenance, which is considerable in both variants. As the indexing scheme restricts the rewrite product to only occur between fresh atoms that share constants with facts, it decreases the amount of data that is retained.

LUBM:OWL2RL and LUBM:RDFS do not require multiple iterations to converge, with the latter only needing a handful. In both of these situations, DYRE with indexing is at least twice as fast as Soufflé in every single measurement. Given these results, we posit that our simple indexing scheme brings overwhelmingly better performance, to the point that it rivals a state of the art reasoner that is not incremental.

5.2. Incremental performance

With DYRE showing adequate single-batch performance when compared with the well-known reasoner Soufflé, we turn to comparing it with DDLog, a Soufflé-inpsired incremental reasoner that compiles non-incremental Datalog programs to differential dataflow programs that are incremental. On the benchmark shown on Figure 5, the X axis contains three sequential steps: The first step **1 - mat** indicates the initial materialisation, with the following ones representing incremental updates, with **2 - update** being a incremental addition, followed by a incremental deletion **3 - remove**.

The amount of data that is added is in the reasoner name. DDLog:99 means that the initial materialisation consisted of 99% of all available data, with the incremental addition containing the remaining 1%, and the deletion removing the same amount. DDLog:50 on the other hand, had a initial materialisation size of 50%, with subsequent updates each adding and retracting the remaining 50%.

We note that a update batch size that is equivalent to the initial materialisation is not realistic, as are all depicted measurements where the batch size is 50%. It being 1% however, is akin to highly dynamic scenarios. The expected behavior of a incremental system is to take less time to adjust to an update than to restart the computation from scratch.

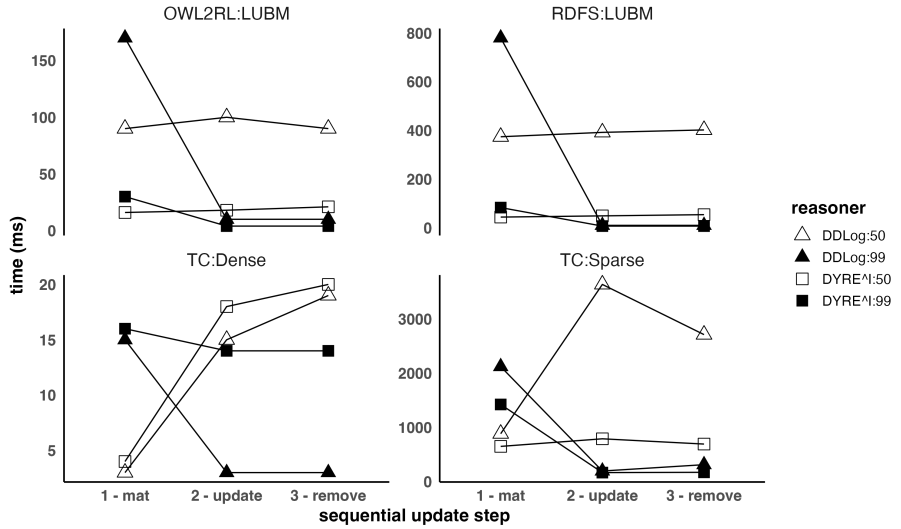

Figure 5: The performance of DYRE with indexing and DDLog under incremental updates

For the programs that are run on LUBM, both DYRE and DDLog exhibit incremental behavior. It takes roughly as much time for both to initialise the materialisation, than to update it with both additions and deletions. When the batch size is 50% the line is flat, as each update takes roughly the same time, as expected.

When the update is much smaller than the initial materialisation, as is the case when the batch size is 1%, there should be a dramatic elbow from step **1 - mat** to **2 - update**. This is witnessed in both graphs, for both reasoners. DYRE however is over ten times faster than DDLog to initially materialise the data in RDFS, and twice to process updates. In OWL2RL it is respectively is four times and two times faster.

While DYRE is significantly faster, DDLog's ratio of initial materialisation over update processing is better. For RDFS DDLog took 780 ms on average to materialise 99% of the data, and then only 10 ms for each update. DYRE on the other hand took 83 ms and 7ms. This implies that DYRE has an almost ten times bigger overhead to process updates.

The performance of DDLog in the benchmarks with the transitive closure program is close to DYRE. For the dense graph, DYRE did not manage to attain incremental behavior, as in the case where the batch size is 1%, updates took the same time as materialisation. DDLog however did not. In the sparse graph, DYRE performed better than DDLog but not by much, with both the initial materialisation and the updates being only 30% faster. DDLog struggled with the large batch size, where it took almost four times as long to update than the initial materialisation.

5.3. Maximum memory resident set size

To provide further insight into the cost of supporting incremental reasoning, Figure 6 shows the peak memory usage that was recorded from each reasoner across all experiments. Soufflé used at least one order of magnitude less memory than every single reasoner. In the most extreme case, transitive closure over the Sparse graph, DYRE used 100 times more memory than Soufflé. In all scenarios DDLog consistently consumed four times less memory than DYRE.

The last notable observation is that the indexing scheme implied only a small increase in memory usage, in spite of leading to a constant ten times higher performance.

6. Conclusion

In this paper we put forward a novel take on the incremental interpretation of dynamic Datalog programs by entirely delegating that problem to DBSP. DBSP is a formal language for incremental

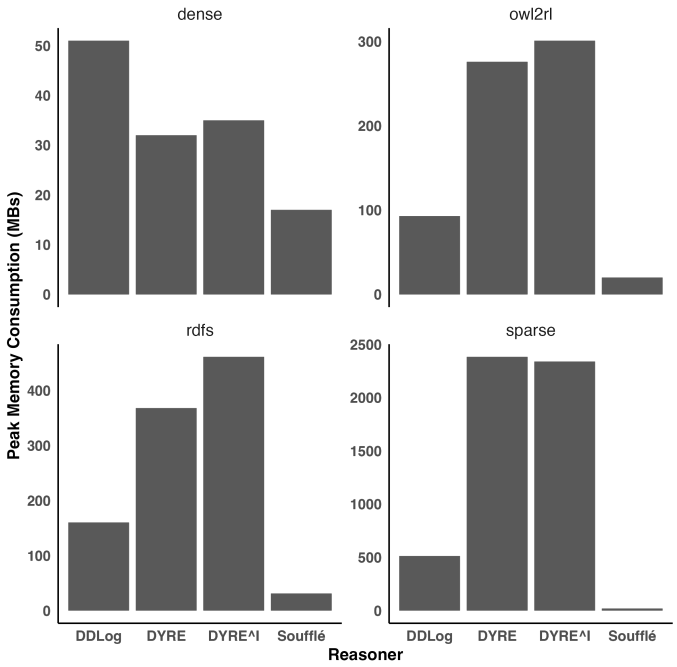

Figure 6: Peak memory usage among all reasoners throughout all benchmarks

iterative streaming computation that allows for highly expressive query languages to be expressed in.

DBSP's primary benefit however isn't in providing a suitable theoretical playground, but in also having a very efficient interpreter written in Rust. While Datalog has been recently seen almost exclusively as a problem of efficiently compiling horn clauses into relational algebra, we revisit its framing as a term rewriting system and directly model it as a incremental DBSP circuit that is then extended with a simple indexing scheme.

We evaluated our reasoner DYRE, **Dy**namic Datalog **Re**asoner, against two state of the art reasoners, Soufflé, that is not incremental, and DDLog. We attained competitive benchmark results, surpassing the former running in interpreted mode in materialization runtime in two occasions, and the latter in most benchmarks, at the cost of four times higher memory consumption. Furthermore, our proposed incremental indexing scheme was shown to be the key element that made its performance competitive.

While DYRE is an interpreter, DDLog is a compiler. The total number of lines of code of DYRE's DBSP circuit sits at 120. Each DDLog differential dataflow program had hundreds, up to thousands for OWL2RL, of lines of code and took minutes to compile.

We then conclude that DBSP is an interesting venue for describing and writing incremental and non-incremental term writing systems whose primary goal is in attaining high performance. We see multiple ways of extending this work, such as by extending the interpreter to support stratified negation or other Datalog flavors, and in improving the indexing scheme by replicating Soufflé's.

References

[1] S. Ceri, G. Gottlob, L. Tanca, et al., What you always wanted to know about datalog(and never dared to ask), IEEE transactions on knowledge and data engineering 1 (1989) 146–166.

[2] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, C. Zaniolo, Big data analytics with datalog queries on spark, in: Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 1135–1149.

[3] M. Imran, G. E. Gévay, V. Markl, Distributed graph analytics with datalog queries in flink, 2020, pp. 70–83.

[4] M. Imran, G. E. Gévay, J.-A. Quiané-Ruiz, V. Markl, Fast datalog evaluation for batch and stream graph processing, World Wide Web 25 (2022) 971–1003.
[5] L. Ryzhyk, M. Budiu, Differential datalog., volume 2, 2019, pp. 4–5.
[6] A. Gupta, I. S. Mumick, Incremental maintenance of recursive views: A survey, MIT Press, 1999.
[7] B. Motik, Y. Nenov, R. Piro, I. Horrocks, Incremental update of datalog materialisation: the backward/forward algorithm, in: AAAI Conference on Artificial Intelligence, 2015.
[8] B. Motik, Y. Nenov, R. Piro, I. Horrocks, Maintenance of datalog materialisations revisited, Artif. Intell. 269 (2019) 76–136.
[9] M. Budiu, F. McSherry, L. Ryzhyk, V. Tannen, Dbsp: Automatic incremental view maintenance for rich query languages, Proc. VLDB Endow. 16 (2022) 1601–1614.
[10] M. Abadi, F. McSherry, G. D. Plotkin, Foundations of differential dataflow, in: Foundations of Software Science and Computation Structures: 18th International Conference, FOSSACS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 18, Springer, 2015, pp. 71–83.
[11] Dbsp implementation github repository, 2024. URL: https://github.com/feldera/feldera/tree/main/crates/dbsp.
[12] Differential dataflow implementation github repository, 2024. URL: https://github.com/TimelyDataflow/differential-dataflow.
[13] M. K. Bruno Rucy Carneiro Alves de Lima, ‘materialized-view‘ System Source Code, https://github.com/brurucy/materialized-view, 2024.
[14] B. R. C. A. de Lima, Pydbsp implementation github repository, 2024. URL: https://github.com/brurucy/pydbsp.
[15] B. Scholz, H. Jordan, P. Subotić, T. Westmann, On fast large-scale program analysis in datalog (2016) 196–206.
[16] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al., Spark sql: Relational data processing in spark, in: Proceedings of the 2015 ACM SIGMOD international conference on management of data, 2015, pp. 1383–1394.
[17] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, The Bulletin of the Technical Committee on Data Engineering 38 (2015).
[18] P. Hu, J. Urbani, B. Motik, I. Horrocks, Datalog reasoning over compressed rdf knowledge bases, 28th ACM International Conference on Information and Knowledge Management (2019).
[19] Y. Zhou, Y. Nenov, B. C. Grau, I. Horrocks, Pay-as-you-go ontology query answering using a datalog reasoner, in: Description Logics, 2015.
[20] B. Motik, Y. Nenov, R. Piro, I. Horrocks, Incremental update of datalog materialisation: the backward/forward algorithm, in: AAAI Conference on Artificial Intelligence, 2015.
[21] J. Kotowski, F. Bry, S. Brodt, Reasoning as axioms change - incremental view maintenance reconsidered, in: International Conference on Web Reasoning and Rule Systems, 2011.
[22] T. J. Green, G. Karvounarakis, V. Tannen, Provenance semirings, in: 26th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 2007, pp. 31–40.
[23] S. Arch, X. Hu, D. Zhao, P. Subotić, B. Scholz, Building a join optimizer for soufflé, in: International Symposium on Logic-Based Program Synthesis and Transformation, Springer, 2022, pp. 83–102.
[24] Y. Guo, Z. Pan, J. Heflin, Lubm: A benchmark for owl knowledge base systems, J. Web Semant. 3 (2005) 158–182.
[25] D. Allemang, J. Hendler, Semantic web for the working ontologist: effective modeling in rdfs and owl, Elsevier, 2011.
[26] S. T. Cao, L. A. Nguyen, A. Szałas, The web ontology rule language owl 2 rl + and its extensions, Trans. Comput. Collect. Intell. 13 (2013) 152–175.
[27] B. N. Grosof, I. Horrocks, R. Volz, S. Decker, Description logic programs: combining logic programs with description logic, in: The Web Conference, 2003.
[28] D. A. Bader, K. Madduri, Gtgraph : A synthetic graph generator suite, 2006.