# A Differential Datalog Interpreter

Matthew Stephenson
Stanford University
Menlo Park, USA
msteph@uw.edu



## ABSTRACT

The core reasoning task for datalog engines is materialization, the evaluation of a datalog program over a database alongside its physical incorporation into the database itself. The de-facto method of computing it, is through the recursive application of inference rules. Due to it being a costly operation, it is a must for datalog engines to provide incremental materialization, that is, to adjust the computation to new data, instead of restarting from scratch. One of the major caveats, is that deleting data is notoriously more involved than adding, since one has to take into account all possible data that has been entailed from what is being deleted. Differential Dataflow is a computational model that provides efficient incremental maintenance, notoriously with equal performance between additions and deletions, and work distribution, of iterative dataflows. In this paper we investigate the performance of materialization with three reference datalog implementations, out of which one is built on top of a lightweight relational engine, and the two others are differential-dataflow and non-differential versions of the same rewrite algorithm, with the same optimizations.

## KEYWORDS

datalog, incremental view maintenance, differential dataflow

## 1 INTRODUCTION

Datalog[9], the canonical language for reasoning over relational databases, ground fact stores, is a declarative language used to evaluate sets of possibly-recursive restricted horn clauses, programs, while remaining not Turing complete. Evaluating a program entails computing implicit consequences over a fact store, yielding new facts.

Materialization, or the physical storage of a program's consequences, eliminates the need for reasoning during query answering. Maintaining this computation is essential for modern Datalog use-cases, as it relates to the broader problem of incremental view maintenance.

While the semi-naive evaluation method[9] efficiently handles additions, deletions are often less efficient, as retracting a fact may naively imply deleting all data derived from it. The delete-rederive[18] method addresses this issue by computing the materialization adjustment through the generation of new Datalog programs, first calculating all possible deletions, and then determining alternative derivations. The difference between these sets represents the actual facts to be deleted.

Using two distinct algorithms for additions and deletions results in different performance characteristics, potentially causing severe biases. For example, when a large portion of ground facts are deleted, such as more than ten percent, a not very realistic value, delete-rederive could be significantly more expensive than recomputing from scratch, since computing all overdeletions, and alternative derivations, might take longer than re-materialization in itself, for as there can be cases where a small portion of ground facts have a high impact on the number of inferred facts.

A promising way to tackle incremental maintenance in a more uniform manner is to use differential dataflow, a programming model that efficiently processes and maintains large-scale possibly recursive dataflow computations. Central to it is the notion of fine-grained tracking, with partially-ordered timestamps, and processing differences between collections of data, rather than entire collections themselves. This approach facilitates efficient updates in response to changes in the underlying data[1].

In the context of datalog, differential dataflow (DD) presents an opportunity to address the performance challenges arising from handling additions and deletions. Contrary to traditional methods, such as semi-naive evaluation for additions and delete-rederive for deletions, differential dataflow provides a unified and efficient approach to incremental view maintenance.

The utilization of partially ordered timestamps and arrangements allows DD to precisely identify affected parts of the computation and to recompute only the necessary components. This leads to a more efficient handling of incremental updates in Datalog evaluation, as the system can focus on affected sub-computations rather than re-evaluating the entire program. Furthermore, there also is first-class support for both automatic parallelism and distributed computing, contributing to enhanced performance and scalability.

DDLog[26] has been the only attempt at building a datalog engine that utilized DD. Similarly to the high-profile reasoner Souffle[27], it is a compiler, in which a datalog program becomes an executable low-level language program, C++ in Souffle's case, and Rust for DDLog. The rationale for the language choice, is that DD's canonical implementation lives as a heavily optimized map-reduce-like framework written in Rust.

Notably, given that DDLog is a compiler, it is not suited for situations where either the program is expected to be dynamic, with rules being added or removed, or where new programs ought to be evaluated during run time, therefore restricting its use case to the specific scenarios where such drawbacks are acceptable.

There has been no study evaluating the isolated benefit of DD to datalog evaluation. Therefore the suitability of DD in this context remains unclear, emphasizing the importance of further research on its potential benefits and limitations in incremental view maintenance.

**Contributions.** In this work, we directly address the posited research question by developing a datalog interpreter utilizing DD. We then compare our implementation with other prototypical datalog interpreters, created from scratch, that share as many components as it is reasonable, in order to isolate the effect of DD in both runtime performance and memory efficiency. This allows us

to more accurately empirically assess how does DD in itself fare against more traditional approaches.

Unlike DDLog, which compiles a datalog program into its evaluation as a fixed DD program, our approach involves writing a single DD program capable of evaluating any datalog program. This eliminates the need for compilation and provides the additional benefit of incremental maintenance for both rule removals and additions.

**Structure of the paper.**

- **Background.** A brief recapitulation of the general background, with datalog, its evaluation methods, and the delete-rederive method being formally introduced.
- **Differential Evaluation.** DD, and the translation of datalog evaluation to a dataflow is showcased and explained.
- **System.** The developed interpreters are described, alongside with all optimizations and benchmark-relevant information.
- **Evaluation.** An empirical evaluation of all reasoners, over multiple different programs and datasets is undertaken.

## 2 RELATED WORKS

**DD Applications and Related Projects.** There are two relevant DD projects that are worth mentioning. One of them is Graspan, a parallel graph processing system that uses DD for efficient incremental computation of static program analyses over large codebases.

Graspan models the program analysis problem as a reachability problem on a graph, where nodes represent program elements and edges represent the relationships between these elements. It leverages DD to incrementally update the analysis results in response to changes in the input graph, which can be due to code modifications or updates to the analysis rules. Graspan has demonstrated its ability to scale to large codebases and provide low-latency updates for various static analyses, including points-to analysis, control-flow analysis, and data-flow analysis.

Another project of interest is DBSP[8], a recent development, that started from the need for a more concise theoretical definition of DD. All of DBSP operators are based on DD's, however, its computational model is less powerful as it does not allow updates to past values in a stream, and it is also assumed that inputs arrive in time order. DBSP can express both incremental and non-incremental computations, with the former not being possible in DD.

**Datalog engines.** There are two kinds of datalog engines. The first encompasses those that compile a datalog program to usually a systems-level programming language, and the second are interpreters, able to evaluate any datalog program.

Soufflé is a prominent example of a datalog compiler that translates datalog programs into high-performance C++ code. It incorporates several optimization techniques, such as parallel execution with highly specialized data structures[21], and nearly optimal join ordering[3]. Notably, its development has been an unparalleled source of articles on the engineering of reasoners.

DDLog As previously mentioned, compiles datalog to DD, achieving efficient differential data updates for datalog programs. It demonstrates the applicability of DD in the context of declarative logic programming and incremental view maintenance.

The majority of reasoners recently developed have been mostly interpreters, further split into distributed or shared memory systems. Out of the shared memory ones, the most notable are RDFox[23], a highly specialized and performant reasoner that is tailored to the semantic web needs, RecStep[31], that builds on top of a highly efficient relational engine, and DCDatalog[30], that builds upon the query optimizer DeALS[29] and extends a work that establishes how some linear datalog programs could be evaluated in a lock-free manner, to general positive programs.

One of the most high-profile datalog papers of interest has been BigDatalog[28], that originally used the query optimizer DeALs, and was built on top of the very popular Spark[4] distribution framework. Soon after, a prototypical implementation[20] over Flink[24], a distribution framework that supports streaming, Cog, followed. Flink, unlike Spark, supports iteration, so implementing reasoning did not need to extend the core of the underlying framework. The most successful attempt at creating a distributed implemention has been Nexus[19], that is also built on Flink, and makes use of its most advanced feature, incremental stream processing.

## 3 BACKGROUND

*Datalog*[9] is a declarative programming language. A program $P$ is a set of rules $r$, with each $r$ being a restriction of tuple-generating dependencies:

$$H(x_1, ..., x_j) \leftarrow \bigwedge_{i=1}^{k} B_i(x_1, ..., x_j)$$

with $k$, $j$ as finite integers, $x$ as terms, and each $B_i$ and $H$ as predicates. A term can belong either to the set of variables, or constants. The set of all $B_i$ is called the *body*, and $H$ the *head*.

A rule $r$ is said to be datalog, if no predicate is negated, and all variables in the head appear somewhere in the body, thereby not there being the possibility for existential variables to exist, conversely, a datalog program is one in which all rules are datalog.

**Example 3.1.** Datalog Program

$$P = \{TC(?x, ?y) \leftarrow Edge(?x, ?y)$$
$$TC(?x, ?z) \leftarrow TC(?x, ?y), TC(?y, ?z)\}$$

Example 3.1 shows a simple valid recursive program. The first rule denotes that *for all x and y, if x is in a Edge relation with y, then it follows that x is in a TC relation with y*, and the second *for all x, y, z, if x is in a TC relation with y, and y is in a TC relation with z, then it follows that x is in a TC relation with z.*

Programs denote implications over a store of ground facts. This store is called the extensional database, *EDB*, and the result of evaluating a program over some *EDB* is the *IDB*, the intensional database.

Let $DB = EDB \cup IDB$, and for there to be a program $P$. We define the *immediate consequence* of $P$ over $DB$ as all facts that are either in $DB$, or stem from the result of applying the rules in $P$

to $DB$. The *immediate consequence operator* $\mathbf{I}_C(DB)$ is the union of $DB$ and its immediate consequence. The $IDB$, at the moment of the application of $\mathbf{I}_C(DB)$, is the difference of the union of all previous $DB$ with the $EDB$, therefore consisting only of the inferred facts.

It is trivial to see that $I_C(DB)$ is monotone, and given that both the $EDB$ and $P$ are finite sets, and that $IDB = \emptyset$ at the start, at some point $I_C(DB) = DB$, since there won't be new facts to be inferred. This point is the *least fixed point* of $I_c(DB)$[9]. Computing the *least fixed point* as described, recursively applying the immediate consequence operator, is called naive evaluation, which is not often used in practice, since in every iteration not only does it infer new facts, but also recomputes all previously inferred ones.

## 3.1 Semi-Naive Evaluation

The semi-naive evaluation algorithm [9] is a widely-used technique for improving naive evaluation, that directly addresses, but does not solve entirely, its major inefficiency, redundant recomputations of previously inferred facts. Given a Datalog program $P$ and an $EDB$, the algorithm iteratively computes the $IDB$ in the same manner as naive evaluation, with the addition of maintaining a set of new delta facts $\Delta$ that are generated in each iteration.

Given a program $P$ with rules $r_0, ..., r_n$, with bodies $b(r) = \{b_0, ..., b_k\}$ and heads $H(r)$, the delta program will generate one new $\Delta$rule for each $IDB$ relation $b_j$ in each rule body $b(r_i)$, in order to represent that only facts that have been recently inferred are to be taken into account for subsequent iterations.

**Example 3.2.** Semi-naive Evaluation Delta Program

$$
\begin{aligned}
r_0 &= TC(?x, ?y) \leftarrow Edge(?x, ?y) \\
\Delta r_1 &= TC(?x, ?z) \leftarrow \Delta TC(?x, ?y), TC(?y, ?z) \\
\Delta r_2 &= TC(?x, ?z) \leftarrow TC(?x, ?y), \Delta TC(?y, ?z)
\end{aligned}
$$

With example 3.1 as the baseline, 3.2 is its resulting delta program. While semi-naive evaluation indeed reduces the number of inferred redundant facts, it is particularly efficient for a certain class of simple Datalog programs that are common in practice, namely linear programs, which are those in which each rule has at most one IDB relation in its body, therefore generating only one delta rule per rule, instead of multiple, as in the example.

In spite of being asymptotically better than naive evaluation, there are substantial implementation challenges that need to be addressed in order to ensure that the overhead is not larger than possible performance gains, since it requires multiple indexes, each delta relation, and efficient set operations to keep track of the most recently inferred facts. This is of utmost importance when using semi-naive evaluation as a method to incrementally handle additions to the $EDB$.

It often occurs that a materialization needs to be adjusted, either to additions or retractions of ground facts. Both semi-naive and naive evaluation are iterative, thus additions can be dealt with by simply having their computations restarted, with the former having the entire $IDB$ as the initial set of delta facts, instead of the empty set. The major goal of continuing the computation is such that it will be more efficient than restarting the materialization altogether.

## 3.2 Delete-Rederive

While both aforementioned evaluation methods provide mechanisms to incrementally adjust materialization to new ground facts, neither support retraction of ground facts, a problem that is significantly more involved, since a single fact might have multiple possible derivations.

The most used method is a bottom-up algorithm[18] that relies on evaluating two new programs, one that computes all possible deletions that could stem from the deletion of the facts being retracted, and then another that attempts to find alternative derivations to the overdeleted ones.

Given a program $P$ with rules $r_0, ..., r_n$, with bodies $b(r) = \{b_0, ..., b_k\}$ and heads $h(r)$, the overdeletion program will generate one new $-$rule for each $b_j$ in each rule body $b(r_i)$, in order to represent that if such fact were to be deleted, then $h(r_i)$ would not hold true.

**Example 3.3.** DRED Overdeletion Program

$$
\begin{aligned}
-r_0 &= -TC(?x, ?y) \leftarrow - Edge(?x, ?y) \\
-r_1 &= -TC(?x, ?z) \leftarrow - Edge(?x, ?y), TC(?y, ?z) \\
-r_2 &= -TC(?x, ?z) \leftarrow Edge(?x, ?y), -TC(?y, ?z)
\end{aligned}
$$

On example 3.3 negative predicates represent overdeletion targets for example 3.1. For instance, if `Edge(2, 3)` is being deleted, then `TC(2, 3)` will be deleted, and any other inferred fact that depends on it. Given that it is a regular datalog program, it can be efficiently evaluated with semi-naive evaluation, or any other evaluation algorithm.

The next step is to compute the alternative derivations of the deleted facts, since some overdeleted facts might still hold true. The alternative derivation program will generate one new +rule for each $r_i$ in $P$, with one extra $-$ head predicate per body, representing an overdeleted fact. The + program requires the overdeleted facts to already not be present.

**Example 3.4.** DRED Alternative Derivation Program

$$
\begin{aligned}
r_0 &= +TC(?x, ?y) \leftarrow - TC(?x, ?y), Edge(?x, ?y) \\
r_1 &= +TC(?x, ?z) \leftarrow - TC(?x, ?z), Edge(?x, ?y), TC(?y, ?z)
\end{aligned}
$$

The output relations from example 3.4 represent the data that has to be put back into the materialization for example 3.1. The rationale for alternative derivations is that, for $r_1$, for instance, if the edge `TC(3, 4)` was overdeleted, because of there being `Edge(1, 2)` and `TC(2, 3)`, if `Edge(3, 4)` was not deleted, by rule $r_0$, then there is an alternative derivation for `TC(3, 4)`.

As it can be seen, computing the maintenance of the materialization implies evaluating a program bigger than the materialization itself, however, due to the fact that it is evaluated with semi-naive evaluation, the asymptotic complexity remains the same. Nonetheless, in practice, deletion is often much slower than addition, as it can be trivially seen by the worst-possible scenario, in which all facts are deleted, whereby while materialization would be free, DRED would inquire an expensive fact-by-fact deletion operation.

### 3.3 Substitution-based evaluation

The most impactful aspect of all of the introduced evaluation mechanisms is the implementation of $I_c$ itself. The two most high-profile methods to do so are either purely evaluating the rules, or rewriting them in some other imperative formalism, such as relational algebra, and executing it.

The substitution-based[9] method is the simplest example of the former. A substitution $\sigma$ is a homomorphism $[x_1 \rightarrow y_1, ..., x_i \rightarrow y_i]$, such that $x_i$ is a variable, and $y_i$ is a constant. Given a not-ground fact, such as $TC(?x, 4)$, *applying* the substitution $[?x \rightarrow 1]$ to it will yield the ground fact $TC(1, 4)$.

Let $r$ be a Datalog rule of the form $h \leftarrow b_1, b_2, \ldots, b_m$, where $h$ is the head atom and $b_i$ are the body atoms. Let $EDB$ be the set of ground facts for the input relations.

The substitution-based method computes the immediate consequence of the rule $r$ as follows:

Define the initial set of substitutions as $\Sigma_0 = \{\sigma_0\}$, where $\sigma_0$ is an empty substitution. For each body atom $b_m$, find the set of ground facts $F_j \subseteq F$ that match $b_m$. Algorithm 1 is the formal specification of the substitution-based method. There is a noteworthy performance issue that arises due to the interaction between it and DRED. During the alternate derivation phase, the new program has one more body atom. This can be prohibitively more expensive to evaluate than the original program, since one extra body atom implies one extra iteration, which could generate a polynomial number of substitutions, due to the cartesian product nature of each step.

**Algorithm 1:** Substitution-based Immediate Consequence

**Input :** $\Sigma_0$, set of ground facts $F$, head atom $h$, body atom list $B$
**Output :** Immediate Consequence $I$

**for** *each* $i = 1, 2, \ldots, m$ **do**
  **for** *each fact* $f \in F$ *and each partial substitution* $\sigma_{i-1} \in \Sigma_{i-1}$ **do**
    Generate an extension $\sigma'_{i-1}$ of $\sigma_{i-1}$ with the constant-to-variable homomorphisms from $f$ that are consistent with the current body atom $b_m$;
    **if** $\sigma'_{i-1}$ *is a valid substitution* **then**
      $\Sigma_i = \Sigma_{i-1} \cup \sigma'_{i-1}$;
    **end**
  **end**
  **for** *each final substitution* $\sigma_m \in \Sigma_m$ **do**
    $I = I \cup \sigma_m H$
  **end**
**end**

### 3.4 Relational algebra rewriting method

The de-facto datalog evaluation method, that virtually all recent reasoners[19, 20, 27, 28, 30, 31] abide by, is to rewrite datalog rules into relational algebra, a well-known technique, to efficiently compute their evaluation, due to the extensive industrial and academic research poured into developing data processing frameworks that handle very large amounts of data, and the techniques that have arisen from those.

Relational Algebra[11] explicitly denotes operations over sets of tuples with fixed arity, relations. It is the most popular database formalism that there is, with virtually every single major database system adhering to the relational model[10, 12**?** ] and using SQL as a declarative syntax.

DD either implements, or makes it trivial to do so, all relevant-to-datalog relational algebra operators, therefore providing convenient tools to manually specify the evaluation of a datalog program as a dataflow. It nonetheless does not directly make writing the interpreter more convenient, only a compiler.

## 4 DIFFERENTIAL EVALUATION

**Differential Dataflow** is a computational framework that generalizes incremental processing to times that are possibly partially ordered, and specifically operates over generalized multisets.

Let $C$ be a multiset, referred to as a collection, with $C_t$ being its value at a partially ordered time $t$, and $C_t(b)$ being the monoid representing the multiplicity of some record $b \in C_t$. We establish that the difference of some collection $C$ at time $t$, named $\delta C_t$, is defined as:

$$\delta C_t = C_t - C_{t-1}$$

It also therefore holds that the value of $C_t$ can be reconstructed by the following equivalence:

$$C_t = \sum_{i \leq t} \delta C_i$$

We utilize plain multiset semantics with signed integers as multiplicity.

Let $A$ and $B$ be collections, and $\mathcal{OP}$ be some operator that maps a collection to some other collection, or itself. Assuming $B$ to be the output of $\mathcal{OP}$ applied over $A$, computations in DD follow the following:

$$B_t = \mathcal{OP}(A_t) = \mathcal{OP}(\sum_{i \leq t} \delta A_i) = \sum_{i \leq t} \mathcal{OP}(\delta A_i)$$

with $\mathcal{OP}$ being proportional to $\delta A_t$, and not $A_t$. Stateful Operators, such as the relational join, require more involved differentiation steps.

A core premise of the canonical DD implementation, is in cleverly, and efficiently, maintaining $\delta B$ and $\delta A$, specifically in the context of iterative dataflows, due to $t$ being partially ordered.

Let's assume that a datalog program is being evaluated, and five fact updates, labeled as $\alpha_t$ arrive. In regular semi-naive evaluation, even though rule application might happen in parallel, $\alpha_{t+1}$ will only be evaluated after $\alpha_t$'s evaluation has finished, and the data used to compute each will always consist of all extensional and intensional(previously inferred) facts.

In contrast, program evaluation could be written as a DD dataflow with a (partially ordered) product order timestamp $\langle t, a \rangle$ with $t$ being the time of arrival of the update, and $a$ keeping track of iteration. Product order is defined as:

$$\langle t_i, a_j \rangle \leq \langle c_k, d_l \rangle \iff t_i \leq c_k \wedge a_j \leq d_l$$

If we treat $\alpha_0$, $\alpha_1$, $\alpha_2$, $\alpha_3$, $\alpha_4$ as differences with the following re-

**Table 1: Product Order Truth Table**

| $\leq$ | $\langle 0,0\rangle$ | $\langle 0,1\rangle$ | $\langle 0,2\rangle$ | $\langle 1,1\rangle$ | $\langle 2,1\rangle$ |
|---|---|---|---|---|---|
| $\langle 0,0\rangle$ | 1 | 1 | 1 | 1 | 1 |
| $\langle 0,1\rangle$ | 0 | 1 | 1 | 1 | 1 |
| $\langle 0,2\rangle$ | 0 | 0 | 1 | 0 | 0 |
| $\langle 1,1\rangle$ | 0 | 0 | 0 | 1 | 1 |
| $\langle 2,1\rangle$ | 0 | 0 | 0 | 0 | 1 |

spective timestamps:

$$\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 1,1\rangle, \langle 2,1\rangle$$

it is noticeable, from table 1, that neither $\alpha_2$ is visible from $\alpha_3$, nor that $\alpha_3$ is visible from $\alpha_2$. This, in turn, has an important consequence in differential dataflow, that the computation of both $\alpha_3$ and $\alpha_2$ happened independent of each other, meaning both may be computed in parallel:

$$\alpha_2 = \delta A_{0,2} = A_{0,2} - (\delta A_{0,0} + \delta A_{0,1})$$
$$\alpha_3 = \delta A_{1,1} = A_{1,1} - (\delta A_{0,0} + \delta A_{0,1} + \delta A_{1,0})$$

Within the context of datalog, the aforementioned evaluation semantics provide a full alternative to the way incremental datalog evaluation is currently done, most specifically, the practical advantage of differential dataflow, is that instead of using semi-naive evaluation and DRED, one can just describe the evaluation process as a dataflow, and have both additions and retractions handled in the same way, with efficient parallelism and symmetric handling of updates.

## 4.1 Differential Substitution-based Method

We now present a translation of algorithm 1 to DD, by emulating sequentially iterating over each rule's body with relational joins, notably, all relational algebra operators are available through a map- reduce-like API. Figure 1 depicts the substitution-based method

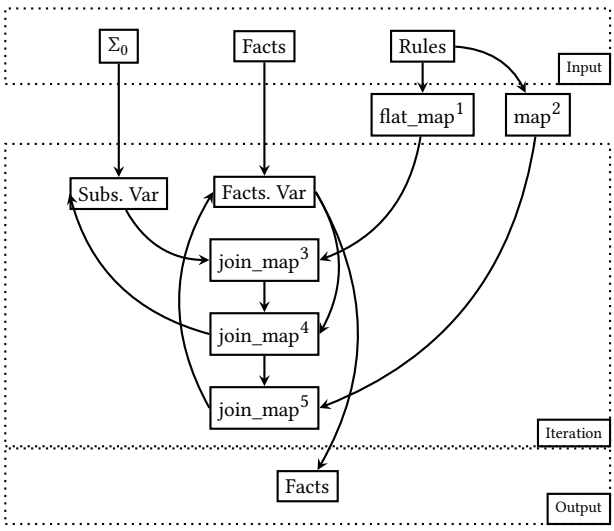


**Figure 1: Substitution method dataflow**

as a dataflow. Superscripts denote points of the dataflow that require further explanation. Furthermore, for clarity, we establish the shape of the data, and the meaning of the `Var` suffix, that both facts and substitutions eventually take up. a Variable is used to express recursive or iterative computations. It allows one to define iterative operations and data dependencies in the dataflow graph, enabling the system to track and propagate changes across iterations efficiently, with product timestamps. Each node either represents an operation, such as `join_map`, that joins indexed collections and then applies a mapping function to the join output, or `flat_map`, that given a function that outputs an iterable, applies it over a collection, and flattens each element's output to be part of a single collection.

We also note that this is a summarized description, where certain trivial, or too-implementation-specific parts have been omitted. $\Sigma_0$ is the stream of empty substitutions indexed per rule identifier, which is pre-populated with one empty substitution per rule. We assume that rules have an unique identifier. Facts is the relation-indexed stream of facts, and rules is the stream of rules, with two indexes, created with the operations with superscripts 1 and 2.

(1) The first rule index indexes rules first by their identifier, and then by each of its body atoms, enumerating them sequentially, imposing an order of evaluation as the original algorithm.
(2) The second rule index indexes by identifier and body size, being necessary to ensure that only the substitutions which have been exhaustively expanded ought to be considered for application to the rule head.
(3) In the first join, the function that is applied, is one that applies substitutions to the input atoms, therefore either creating new atoms, with less variables as terms, or the very same ones. This is equivalent to the necessary setup for step 1 of Algorithm 1 to occur, making use of index 1.
(4) The next join creates new substitutions, based on the newly minted atoms. All current substitutions are attempted to be expanded further, with the successful ones being emitted from the join.
(5) This is the last step of the algorithm, where all final substitutions are applied to the head of each rule, index 2, to then create new ground facts.

With the dataflow being specified, over the next section we elaborate on the commonalities and differences with the other implementations.

# 5 SYSTEM

In this section we provide a technical overview of the implemented reasoners, and what is shared between them, alongside a novel indexing technique for the substitution-based method, that at the cost of increased memory usage, can significantly decrease the number of times the operation that occurs the most frequently, substitution extension, occurs.

The reasoner that uses the substitution-based method without DD is named Chibi, differential is the one that does. Both of these reasoners share the implementation of the three core elements: unification, substitution application, and in asserting that a fact is ground. All of the aforementioned operations are trivial, and each

do not require more than ten or so lines of code. Unification is a computationally cheap operation, given an atom, and a ground fact, the output is a new substitution that maps the variables of the right to the constants of the left one. All others are self descriptive, with substitution application merely substituting an atom's variables for the mapped variables in a substitution. Checking if a fact is ground is done by ensuring that no terms are variables.

Chibi, Differential and Relational all share the same memory layout for the core elements of datalog and storage. In Rust terms, it is to be assumed that all referred data structures are standard library implementations unless stated otherwise. Furthermore, a step of rule application is always done in parallel.

- Constant: an enumeration of boolean, 64-bit integer or string, respectively named typed values
- Variable: an 8 bit integer, hence imposing a bound on the number of variables that a rule can have
- Term: an enumeration of constant and variable
- Atom: A struct with a vector of terms, and a symbol, that can be either a 64-bit integer or a string
- Rule: A struct with an atom representing the head, and a vector of atoms as the body
- Storage: A Hash map of hash sets, with keys representing relation names, or id, and their respective hash sets containing vectors of typed terms, ground facts

Relational reasoner has one extra data structure, a btree index, that is used for sort-merge joins. Relational relies on naively translating datalog rules into relational algebra, without any further optimizations whatsoever, aside from inserting all data that is to be joined in its index, right before actually doing it. All relational operations and their evaluator were implemented from scratch. The point of this reasoner is to evaluate how performant the popular relational algebra evaluation can be in isolation, compared to the often forgotten substitution-based method.

Rule application until the least fixpoint is reached is done with semi-naive evaluation[2], with a program transformation. DRED is implemented as described in [18], in two steps, with both the overdeletion and alternative derivation program being executed with semi-naive evaluation too. Both Chibi and Relational use the same function for this, with differential evidently not using semi-naive evaluation nor DRED, given that it has its own iteration mechanism, heavily inspired by semi-naive evaluation, which already handles retractions.

## 5.1 Demand-driven Multiple-column-based Indexing

There is a possibly very large performance cost of the substitution-method, that can be exemplified in the specific scenario of DRED, that could render it unable to be used in practice. As it was introduced, substitutions are both incrementally expanded, and built anew, by iterating over every single body atom.

In the second step of DRED, an alternate derivation program is created. This program has one extra body atom, representing overdeletions of the head's relation. This implies that this step could be prohibitively more expensive to evaluate than even evaluating the program, due to the cartesian nature of the unification step, that implies iterating over the knowledge base once, for every atom. This inefficiency can be demonstrated with the following example, in which the rule could be seen as the alternate derivation step of some rule: $R(?x, ?z) < -T(?x, ?y), T(?y, ?z)$, with $-R$ representing the overdeletion estimation from the previous step.

Let $P = \{+R(?x, ?z) \longleftarrow -R(?x, ?z), T(?x, ?y), T(?y, ?z)\}$, and $EDB = \{T(a, b), T(b, c), T(c, d), -R(a, c), -R(b, d)\}$ Algorithm 1 will have three iterations:

(1) (a) Current body atom: $-R(?x, ?z)$, $\Sigma_0$: $[\{\}]$
  (b) Fresh atoms - Applying all $\sigma : \Sigma_0$ to $-R(?x, ?z)$ yields $-R(?x, ?z)$
  (c) Substitution extension:
    (i) unification: -R(?x, ?z) ∪ -R(a, c) = $\{?x \rightarrow a, ?z \rightarrow c\}$
    (ii) unification: -R(?x, ?z) ∪ -R(b, d) = $\{?x \rightarrow b, ?z \rightarrow d\}$

(2) (a) Current body atom: $T(?x, ?y)$, $\Sigma_1$: $[\{?x \rightarrow a, ?z \rightarrow c\}, \{?x \rightarrow b, ?z \rightarrow d\}]$
  (b) Fresh atoms - Applying all $\sigma : \Sigma_1$ to $T(?x, ?y)$ yields $T(a, ?z), T(b, ?z)$
  (c) Substitution extension:
    (i) unification: T(a, ?y) ∪ T(a, b) = $\{?x \rightarrow a, ?y \rightarrow b, ?z \rightarrow c\}$
    (ii) unification: T(a, ?y) ∪ T(b, c) = none
    (iii) unification: T(a, ?y) ∪ T(c, d) = none
    (iv) unification: T(b, ?y) ∪ T(a, b) = none
    (v) unification: T(b, ?y) ∪ T(b, c) = $\{?x \rightarrow b, ?y \rightarrow c, ?z \rightarrow d\}$
    (vi) unification: T(b, ?y) ∪ T(c, d) = none

(3) (a) Current body atom: $T(?y, ?z)$, $\Sigma_2$: $[\{?x \rightarrow a, ?y \rightarrow b, ?z \rightarrow c\}, \{?x \rightarrow b, ?y \rightarrow c, ?z \rightarrow d\}]$
  (b) Fresh atoms - Applying all $\sigma : \Sigma_2$ to $T(?y, ?z)$ yields $T(b, c), T(c, d)$
  (c) Substitution extension:
    (i) unification: T(b, c) ∪ T(a, b) = none
    (ii) unification: T(b, c) ∪ T(b, c) = $\{?x \rightarrow a, ?y \rightarrow b, ?z \rightarrow c\}$
    (iii) unification: T(c, d) ∪ T(c, d) = none
    (iv) unification: T(c, d) ∪ T(a, b) = none
    (v) unification: T(c, d) ∪ T(b, c) = none
    (vi) unification: T(c, d) ∪ T(c, d) = $\{?x \rightarrow b, ?y \rightarrow c, ?z \rightarrow d\}$

With the final substitutions being: $[\{?x \rightarrow a, ?y \rightarrow b, ?z \rightarrow c\}, \{?x \rightarrow b, ?y \rightarrow c, ?z \rightarrow d\}]$, therefore inferring two atoms: $+R(a, c)$ and $+R(b, d)$. The major source of inefficiency are calls to unification attempt, that yield no new substitution. The number of unification attempts could grow quadratically with each next body atom. The solution to this issue is straightforward; to avoid the cartesian product. We devise a novel indexing technique specifically tailored to be portable to DD.

Returning to the example, it is trivial to see that wasteful unification attempts can be prevented by joining on bindings; If $T(a, ?y)$ is the left-hand side of unification, and $T(a, b)$, $T(b, c)$ are the candidates, no candidate that does not already match all constants in $T(a, ?y)$ would produce a substitution extension.

We name our approach Demand-driven Multiple-column-based Indexing, because indexes are built on-demand to address the need of indices for joining substitutions, that can be over multiple constants, therefore spanning over multiple columns, in each iteration. For each rule we determine the column combinations that will be used in such a join, and maintain one globally shared index for each unique column combination. First, we demonstrate the technique over the same example, and then provide a new version of Algorithm 1.

(1) (a) Current body atom: $-R(?x, ?z)$, $\Sigma_0$: [{}]
  (b) Fresh atoms - Applying all $\sigma : \Sigma_0$ to $-R(?x, ?z)$ yields $-R(?x, ?z)$
  (c) Index 1 - Index all fresh atoms with the positions of their constant terms as keys: {[] : [[?x, ?z]]}
  (d) Index 2 - Index $-R$ based on all distinct values of the column keys of index 1 : {[] : [[a, c] : [[]], [b, d] : [[]]]}
  (e) Index 4 - Join Index 1 with Index 2:
    (i) ([?x, ?z], [[a, c] : [[]]])
    (ii) ([?x, ?z], [[b, d] : [[]]])
  (f) Attempt to unify:
    (i) unification: -R(?x, ?z) U -R(a, c) = $\{?x \to a, ?z \to c\}$
    (ii) unification: -R(?x, ?z) U -R(b, d) = $\{?x \to b, ?z \to d\}$

(2) (a) Current body atom: $T(?x, ?y)$, $\Sigma_1$: [$\{?x \to a, ?z \to c\}$, $\{?x \to b, ?z \to d\}$]
  (b) Fresh atoms - Applying all $\sigma : \Sigma_1$ to $T(?x, ?y)$ yields $T(a, ?y), T(b, ?y)$
  (c) Index 1 - Index all fresh atoms with the positions of their constant terms as keys: {[0] : [[a, ?y], [b, ?y]]}
  (d) Index 2 - Index $T$ based on all distinct values of the column keys of index 1 {[0] : {[a] : [[b]], [b] : [[c]], [c] : [[d]]}}
  (e) Index 4 - Join Index 1 with Index 2:
    (i) ([a, ?y], [[a] : [[b]]])
    (ii) ([b, ?y], [[b] : [[c]]])
  (f) Attempt to unify:
    (i) unification: T(a, ?y) U T(a, b) = $\{?x \to a, ?y \to b, ?z \to c\}$
    (ii) unification: T(b, ?y) U T(b, c) = $\{?x \to b, ?y \to c, ?z \to d\}$

(3) (a) Current body atom: $T(?y, ?z)$, $\Sigma_2$: [$\{?x \to a, ?y \to b, ?z \to c\}$, $\{?x \to b, ?y \to c, ?z \to d\}$]
  (b) Fresh atoms - Applying all $\sigma : \Sigma_2$ to $T(?y, ?z)$ yields $T(b, c), T(c, d)$
  (c) Index 1 - Index all fresh atoms with the positions of their constant terms as keys: {[0, 1] : [[b, c], [c, d]]}
  (d) Index 2 - Index $T$ based on all distinct values of the column keys of index 1 : {[0, 1] : {[a, b] : [[]], [b, c] : [[]], [c, d] : [[]]}}
  (e) Index 4 - Join Index 1 with Index 2:
    (i) ([b, c], [[b, c] : [[]]])
    (ii) ([c, d], [[c, d] : [[]]])
  (f) Attempt to unify:
    (i) unification: T(b, c) U T(b, c) = $\{?x \to a, ?y \to b, ?z \to c\}$
    (ii) unification: T(c, d) U T(c, d) = $\{?x \to b, ?y \to c, ?z \to d\}$

From this new example, it can be seen that the indexing scheme is relatively simple, relying on creating new indices that would allow unification to never wastefully occur. We now structure it as Algorithm **??**.

Let $P : a \to [\mathbb{N}]$ be a function mapping an atom to an array of integers representing the positions of constants within the atom's terms, and $R : ([\mathbb{N}], a) \to C$ another function, that maps an array of integers and an atom, to a subset of the atom's terms $c$ denoted by $C$.

The algorithm relies on two main indexes:

(1) $I_1 : P(a) \to 1^F$, where $1^F$ is the subset of $F$ such that all $a$ have the same $P(a)$ value.
(2) $I_2 : P(a) \to I_3$, where $I_3 : R(F) \to 1^F$ is a nested index, and $1^F$ is the subset of $F$ such that all $a$ have the same $R(P(a), a)$.

**Algorithm 2:** Substitution-based Immediate Consequence with Demand-driven Multiple-column-based Indexing

**Input :** $\Sigma_0$, set of ground facts $F$, head atom $H$, body atom list $B$
**Output :** Immediate Consequence $I$

**for** *each* $i = 1, 2, \ldots, m$ **do**
  Apply $\Sigma_{i-1}$ to the current body atom to obtain fresh atoms $\mathfrak{A}$;
  Create the first index $I_1 : P(a) \to 1^{\mathfrak{A}}$;
  Create the second index $I_2 : P(a) \to I_3$;
  Create the last index by joining Index 1 and Index 2: $I_4(a) = (I_1(a), I_2(a))$ for each $a \in \mathfrak{A}$;
  **for** $(S_1, S_2) \in I_4$ **do**
    Unify each element in $S_1$ with each element in $S_2$;
    Generate an extension $\sigma'_{i-1}$ of $\sigma_{i-1}$ with the constant-to-variable homomorphisms from $f$ that are consistent with the current body atom $b_m$;
    **if** $\sigma'_{i-1}$ *is a valid substitution* **then**
      $\Sigma_i = \Sigma_{i-1} \cup \sigma'_{i-1}$;
    **end**
  **end**
  **for** *each final substitution* $\sigma_m \in \Sigma_m$ **do**
    $I = I \cup \sigma_m H$
  **end**
**end**
**??**

All indexing steps are $O(|F|)$ in time and data complexity, save for index two, that has worst-case data complexity of $O(|F| \cdot 2^{|a|})$, with $2^{|a|}$ representing the powerset of the number of terms in some atom $a$, and $F$ such that it has only atoms $a$. The product with the powerset arises due to how indexing occurs by mapping all unique combinations of constant terms of fresh atoms, which in the worst-case could be exponential to the arity. Figure 2 displays the DD

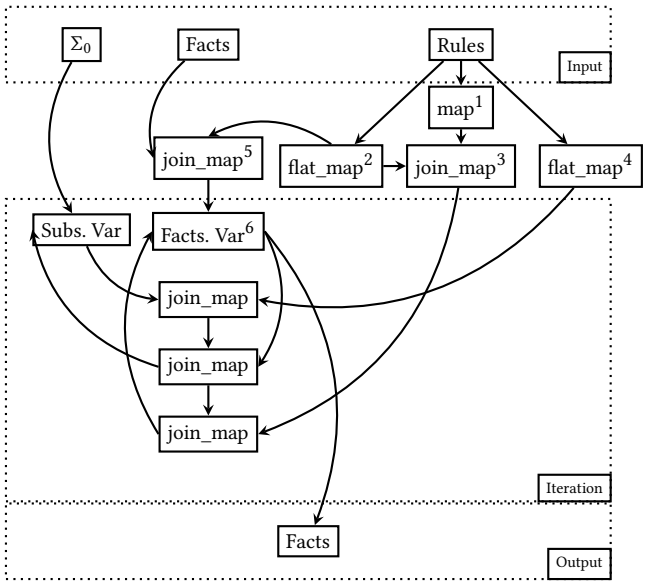


**Figure 2: Substitution method with indexing dataflow**

version of Algorithm **??**, that mostly remains exactly the same, save for new operations happening during the phase before iteration. We now clarify the points of interest in the new dataflow. There were no differences in the steps inside iteration, aside from joins happening through the vector of constant positions and relation symbols, instead of only relation symbols.

(1) The first `map` operator remains the same, indexing rules by their identifier and body size, used to ensure that only fully expanded substitutions will be applied to rule heads. The same as superscript 2 in 1.
(2) The unique column combinations of the input ruleset are computed by this operator.
(3) This step joins the rule identifiers with the unique column combinations. This is only used at the very last join during iteration, to ensure that the output fact is indexed by the correct column combination.
(4) Equivalent to superscript 1 in 1.
(5) With superscript 2, the input fact stream can be immediately indexed by the necessary constant position combinations. This is done by a join on relation symbol, that will index each fact by all column combinations.
(6) Facts. var, unlike in Algorithm 1's dataflow, which was only indexed by relation, is now indexed by each unique column combination.

This dataflow is possibly much more efficient. An arrangement in DD is a pre-computed, indexed representation of a collection that allows for efficient querying and manipulation of the data. These arrangements play a crucial role in the performance of joins. By carefully choosing which arrangements to create and maintain, it is possible to keep joins efficient, without unnecessarily wasting memory.

**Table 2: Dataset Overview**

| Dataset | Area of Interest | Programs |
|---|---|---|
| LUBM | semantic web | RhoDFS, RhoDFS-s, OWL2RL |
| RMAT1K | synthetic | tc |
| RAND1K | synthetic | tc |

Most specifically, arrangements dictate the level of join efficiency. The fact that the join operator indexes the data by a more fine-grained key than relation symbol, such as relation symbol and positions occupied by constant values, allows is to be much more restrictive than cartesian product.

## 6 EVALUATION

Three thorough experiments were conducted in order to showcase relative performance, scalability, and memory usage, of all reasoners, with the intent being twofold: to evaluate the performance characteristics of DD, in isolation of virtually all other elements, and to establish as to whether general algorithmic improvements, such as the demand-driven indexing scheme, are portable to DD.

**Setup.** The experiments were run on a google-cloud-provisioned x86 machine of type e2-standard-16, with 16 intel skylake cores and 64 gigabytes of RAM. Each benchmark measurement was taken 70 times, with the 20 measurements of most variance removed, and averaged out. All datasets, datalog programs and reasoner implementations are available online[25].

**Datasets.** On table 2 all datasets and program names, or acronyms, are shown. There are two areas of interest. The semantic web has very specific use-cases for datalog, and are the leading source of research in extending the datalog mathematical formalism, and in providing improvements to decades-old algorithms, such as DRED, with the backward-forward algorithm[22]. Seeking ways to introduce tuple-generating dependencies to programs, with evaluation remaining tractable, has been one of the most active research directions, with highly-influential papers establishing new families of datalog languages[14] and thoroughly exploring their complexity classes alongside even further extensions[6, 13, 15]. These advancements have been somewhat tested in practice, albeit with no full reference implementation having been specified. The most comprehensive, and recent, is closed-source[7]. The leading datalog engine in general, is also closed-source[23], and is tailored specifically to the semantic web.

The second area of interest is of purely mathematical synthetic graph benchmarks, that allow for generating infinitely-scalable specific graph structures. all datasets however, including LUBM[17], are synthetic, with the difference being that there are multiple specific programs for RhoDFS.

- **LUBM** is a classic inference benchmark dataset for both RhoDFS and OWL2RL rulesets. The data is divided in two parts, the TBox, terminological box, that holds an ontology able to describe universities, and the ABox, assertional box, that asserts facts about universities using the terminology in the TBox. The RhoDFS ruleset, depicted on A.1, is relatively simple, but complex, there being only a single relation that is mutually recursive in every single rule.

RhoDFS-s A.2 is an improved version of RhoDFS, that creates new relations for every single constant combination in the original program, avoiding every body atom implying a full dataset, mimicking the relational selection. The last ruleset, OWL2RL, has over 100 rules and is by far the most complex, representing the lower bound of OWL2RL implications, specific of the LUBM Tbox. More information on converting description logic entailments to datalog can be found on[16].

- **RMAT1k**. is a graph generated by the `rmat` profile of the GT[5] graph generator, used to benchmark various other reasoners[31][28]. The dataset is a graph with ten times the number of edges as vertices, that follows an inverse power-law distribution.
- **RAND1k** is also a graph generated with the `rand` profile of GT. The dataset is comprised of a graph that has one thousand edges, with each having 0.01 probability of being connected to every other. In spite of having a small number of nodes, it is incredibly dense, with the output of the transitive closure program having almost a hundred times more edges than the initial graph.

## 6.1 Runtime comparison

Table 3 pictures the main benchmark, in which three measurements, Mat, +, and -, for every batch size, are recorded. All measurements are in **seconds**. If the batch size is 75%, then Mat is the amount of time taken to materialize 75% of the data, using regular semi-naive evaluation, + is how much incremental materialization, of 25% of the data, the remaining amount, also using semi-naive evaluation, took, and lastly, - is how much time DRED has taken to delete the 25% that has been added. This provides a comprehensive and thorough overview of the performance of DRED and semi-naive evaluation, compared to differential dataflow, which offers an alternative to both.

Notably, the selection of facts in + and - can dramatically influence the performance of both DRED and DD. However, conducting extensive performance estimations by running the algorithms on numerous random subsets of the data is impractical due to the extensive duration required to run the entire benchmark, coupled with the factorial number of possible permutations. Thus, we chose to just select random subsets of the data that contained 50%, 25%, 10%, 1%, and 0.1% of its original size, as update sizes.

We discuss the table over each dataset and its respective programs. First, for LUBM under the rdfs program, all differential reasoners exhibit a clear trend of decreasing update computation times, as the batch size increases, with diff$^I$ performing much better in general, up until updates get very small, possibly indicating that at this level, indexing starts to have too big of an overhead. In the case of all other reasoners, the trend is very different, with all update times, curiously save for chibi, which is orders of magnitude slower than all other reasoners, not decreasing. This is unsurprising, due to the very strong degree of recursiveness of the program, therefore showcasing that neither DRED nor semi-naive evaluation provide significant speedups over rematerialization, with the best result being for chibi$^I$, in which updates and deletions, in spite of being constant, are up to 40% faster.

All reasoners perform significantly better on rdfs-s, indicating the importance of the program. Chibi's pathological performance issue is entirely gone with the new program, and its performance discrepancy with chibi$^I$ is almost eliminated, save for deletions, which remain several times slower than rematerialization.

In the most complex program, owl2rl, both chibi and diff are not able to finish materialization, with the former having had taken more than 1000 seconds, and the latter exceeding 64 gigabytes of RAM. Differential performs in the same manner as the previous programs, with decreasing update times, and symmetry between additions and deletions. Both chibi$^I$ and rel exhibit decreasing deletion reasoning times in aggressive cliffs, with little decrease for additions.

The transitive closure program is simple, and linear, therefore being embarassingly simple to incrementalize. For the RAND-1k dataset, differential reasoners once again perform in the same manner, with incremental behavior scaling linearly with the size of the data. The same behavior is shown for all other reasoners, with a caveat, that DRED only starts to be competitive once the update size is less than 10% of the original data. For RMAT-1k, reasoning times are much longer, showcasing a significantly more complex dataset, with all non-differential reasoners struggling to provide proportional update times save for update sizes of less than 1%.

In sum, diff and diff$^I$ performed predictably irrespective of the dataset and program being run, always being faster, and having proportionally decreasing reasoning times for updates, while at the same time being symmetric. All other reasoners did not show the expected incremental behavior, neither for semi-naive evaluation nor DRED unless the update size was small, which is not necessarily a hindrance in practice, since rarely if ever a system will receive an update that is bigger than 10% of the original size of the data.

## 6.2 Peak memory usage comparison.

The results of the previous subsection cannot be seen in an entirely positive light without there being consideration for memory usage. DD relies on multiple in-memory indexes to keep track of all changes, and as it was seen, it entirely failed a benchmark due to running out of memory, thus, in this section we analyze the results of measuring peak memory usage over the previous experiments. Table 4 presents the peak memory usage for each of the methods and programs across different datasets. Memory usage is presented in megabytes. LUBM1 occupies 20 megabytes of disk space, RAND-1k and RMAT-1k, respectively, 100 kilobytes.

For LUBM1, with the 'rdfs' and 'rdfs-s' programs, all reasoners performed comparably with each other, with respect to memory usage, however, as seen on the previous table, there are major differences in runtime performance between them, with the most extreme example being for chibi and diff$^I$, in which the former is over 1000x times slower, while using almost 50% more memory. Interestingly, diff performed significantly better for the owl2rl program, consuming 100 times less memory than chibi and rel. It is likely that this is due to the aforementioned aggressive compaction mechanism by the in-memory LSM trees. Notably, the indexed version of diff, diff$^I$, ran out of memory (OOM) for this program, indicating possible limitations of the indexing method for handling complex queries in large datasets, which conversely is not true in the case

**Table 3: Runtime Experimental Results**

| Dataset | Program | Batch | diff | | | diff$^I$ | | | chibi | | | chibi$^I$ | | | rel | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Mat | + | - | Mat | + | - | Mat | + | - | Mat | + | - | Mat | + | - |
| LUBM1 | rdfs | 50% | 1.47 | 1.43 | 1.40 | 0.47 | 0.48 | 0.49 | 124 | 530 | 584 | 0.84 | 1.13 | 1.62 | 0.71 | 1.02 | 1.58 |
| | | 75% | 2.15 | 0.74 | 0.73 | 0.67 | 0.29 | 0.25 | 276 | 559 | 369 | 1.10 | 1.01 | 1.38 | 1.01 | 0.97 | 1.42 |
| | | 90% | 2.58 | 0.33 | 0.34 | 0.84 | 0.14 | 0.13 | 397 | 573 | 168 | 1.40 | 1.02 | 1.22 | 1.26 | 1.03 | 1.42 |
| | | 99% | 2.91 | 0.05 | 0.05 | 0.95 | 0.05 | 0.03 | 486 | 584 | 23 | 1.54 | 1.00 | 0.97 | 1.41 | 0.97 | 1.23 |
| | | 99.9% | 2.94 | 0.03 | 0.01 | 0.97 | 0.03 | 0.02 | 487 | 586 | 5.5 | 1.60 | 1.00 | 1.23 | 1.38 | 0.94 | 1.45 |
| | | 100% | 2.89 | 0 | 0 | 0.99 | 0 | 0 | 487 | 0 | 0 | 1.34 | 0 | 0 | 1.20 | 0 | 0 |
| | rdfs-s | 50% | 0.84 | 1.11 | 0.92 | 0.27 | 0.29 | 0.35 | 0.72 | 1.2 | 126 | 0.65 | 1.11 | 1.67 | 0.63 | 1.25 | 1.72 |
| | | 75% | 1.31 | 0.46 | 0.49 | 0.35 | 0.17 | 0.16 | 1.11 | 1.04 | 103 | 1.11 | 1.21 | 1.3 | 0.94 | 1.03 | 1.41 |
| | | 90% | 1.67 | 0.24 | 0.23 | 0.40 | 0.09 | 0.09 | 1.3 | 1.10 | 54 | 1.12 | 1.08 | 1.2 | 1.26 | 1.16 | 1.32 |
| | | 99% | 1.72 | 0.05 | 0.05 | 0.44 | 0.05 | 0.02 | 1.5 | 1.1 | 9.5 | 1.48 | 1.09 | 1.1 | 1.28 | 1.20 | 1.58 |
| | | 99.9% | 1.65 | 0.03 | 0.02 | 0.45 | 0.03 | 0.02 | 1.5 | 1.0 | 2.9 | 1.39 | 1.10 | 1.0 | 1.46 | 1.32 | 1.52 |
| | | 100% | 1.77 | 0 | 0 | 0.45 | 0 | 0 | 1.2 | 0 | 0 | 1.38 | 0 | 0 | 1.12 | 0 | 0 |
| | owl2rl | 50% | 3.16 | 8.48 | 9.19 | OOM | OOM | OOM | OOT | OOT | OOT | 31.1 | 85.7 | 55.9 | 32.0 | 88.1 | 16.3 |
| | | 75% | 6.59 | 4.91 | 5.00 | OOM | OOM | OOM | OOT | OOT | OOT | 66.8 | 71.7 | 36.4 | 85.1 | 81.3 | 16.1 |
| | | 90% | 9.50 | 2.42 | 2.29 | OOM | OOM | OOM | OOT | OOT | OOT | 114 | 63.5 | 15.1 | 130 | 70 | 16.3 |
| | | 99% | 11.2 | 0.04 | 0.03 | OOM | OOM | OOM | OOT | OOT | OOT | 114 | 60.2 | 2.52 | 156 | 34 | 0.60 |
| | | 99.9% | 11.3 | 0.03 | 0.02 | OOM | OOM | OOM | OOT | OOT | OOT | 117 | 73.3 | 1.3 | 161 | 34 | 0.61 |
| | | 100% | 11.2 | 0 | 0 | OOM | 0 | 0 | OOT | 0 | 0 | 138 | 0 | 0 | 162 | 0 | 0 |
| RAND-1k | tc | 50% | 0.06 | 1.07 | 1.02 | 0.03 | 0.08 | 0.10 | 0.03 | 0.48 | 1.08 | 0.01 | 0.13 | 0.17 | 0.01 | 0.13 | 0.13 |
| | | 75% | 0.23 | 0.94 | 0.91 | 0.05 | 0.07 | 0.07 | 0.14 | 0.42 | 2.25 | 0.02 | 0.12 | 0.23 | 0.02 | 0.13 | 0.16 |
| | | 90% | 0.64 | 0.56 | 0.56 | 0.07 | 0.06 | 0.05 | 0.45 | 0.48 | 5.96 | 0.08 | 0.15 | 0.70 | 0.07 | 0.15 | 0.26 |
| | | 99% | 1.05 | 0.17 | 0.17 | 0.08 | 0.03 | 0.03 | 0.77 | 0.52 | 0.72 | 0.12 | 0.16 | 0.15 | 0.11 | 0.16 | 0.16 |
| | | 99.9% | 1.13 | 0.03 | 0.03 | 0.09 | 0.01 | 0.01 | 0.85 | 0.43 | 0.11 | 0.16 | 0.07 | 0.06 | 0.14 | 0.05 | 0.05 |
| | | 100% | 1.15 | 0 | 0 | 0.10 | 0 | 0 | 0.86 | 0 | 0 | 0.16 | 0 | 0 | 0.14 | 0 | 0 |
| RMAT-1k | tc | 50% | 1.30 | 13.0 | 11.2 | 0.63 | 2.51 | 3.83 | 0.99 | 5.01 | 7.70 | 0.12 | 1.40 | 2.03 | 0.20 | 1.36 | 1.72 |
| | | 75% | 5.29 | 9.22 | 8.59 | 1.51 | 2.13 | 2.57 | 3.71 | 4.52 | 8.84 | 0.57 | 1.67 | 2.06 | 0.61 | 1.54 | 1.84 |
| | | 90% | 8.88 | 4.09 | 3.91 | 2.11 | 1.08 | 0.95 | 6.17 | 5.25 | 9.48 | 0.89 | 1.72 | 2.11 | 0.89 | 1.67 | 2.01 |
| | | 99% | 12.0 | 0.76 | 0.59 | 2.40 | 0.06 | 0.06 | 8.32 | 5.51 | 10.2 | 1.12 | 1.68 | 2.68 | 1.20 | 1.55 | 2.28 |
| | | 99.9% | 12.7 | 0.04 | 0.04 | 2.36 | 0.01 | 0.01 | 8.79 | 4.63 | 0.55 | 1.25 | 0.90 | 0.69 | 1.31 | 0.58 | 0.78 |
| | | 100% | 12.8 | 0 | 0 | 2.31 | 0 | 0 | 8.78 | 0 | 0 | 1.26 | 0 | 0 | 1.30 | 0 | 0 |

**Table 4: Memory usage experimental results**

| Dataset | Program | diff | diff$^I$ | chibi | chibi$^I$ | rel |
|---|---|---|---|---|---|---|
| LUBM1 | rdfs | 488 | 466 | 631 | 941 | 722 |
| | rdfs-s | 495 | 383 | 573 | 665 | 579 |
| | owl2rl | 446 | OOM | 42190 | 29269 | 25450 |
| RAND-1k | tc | 90 | 85 | 41 | 47 | 31 |
| RMAT-1k | tc | 434 | 521 | 265 | 285 | 258 |

of chibi$^I$, therefore being an issue with the DD implementation in itself.

In both the RAND-1k and RMAT-1k datasets, all differential reasoners consume at least twice as much memory as all other reasoners, while performing similarly for initial materialization runtime. This posits an interesting counterpoint to the dominance in both memory usage and runtime shown with more complex programs. The reason for this discrepancy, is that the TC program has a very large number of iterations, therefore causing a significantly greater flux in the dataflow, and since each iteration implies a new difference being stored, memory usage can grow at a fast pace.

While there are major differences in runtime among all reasoners, with some being orders of magnitude faster, the same cannot be said about memory usage, which save for a very large program, there are no clear winners, implying that the memory requirements for DD in itself are not greater than regular reasoners, save for highly-iterative dataflows, and remains proportional to the computation. The starkest example of this is for the owl2rl program, which in spite of containing over a hundred rules, does not output much more data than rdfs/rdfs-s.

## 7 CONCLUSION

In this article we introduced a novel datalog reasoner, with two different algorithms, whose core value proposition is in it using the promising, but relatively obscure, DD model of computation, and evaluated it against two other reference implementations that shared as many components as reasonable. We also described an indexing method that significantly sped up a often overlooked method of implementing reasoning, the substitution method, that was shown

to have solved many pathological performance issues in benchmarks, at very little cost of extra memory. In all experiments, all DD based reasoners implemented bested their non differential counterparts, showing unparalleled scalability over increasing update sizes, alongside virtually no performance differences between additions and retraction, while remaining competitive in memory usage. There are multiple ways in which the work could be expanded in the future, such as in porting it over to support negation and more expressive variants of datalog, and most importantly, making it distributed, which DD provides out of the box.

## REFERENCES


[1] Martín Abadi, Frank McSherry, and Gordon Plotkin. 2015. Foundations of Differential Dataflow. 71–83. https://doi.org/10.1007/978-3-662-46678-0_5

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1994. Foundations of Databases.

[3] Samuel Arch, Xiaowen Hu, David Zhao, Pavle Subotic, and Bernhard Scholz. 2022. Building a Join Optimizer for Soufflé. In *International Workshop/Symposium on Logic-based Program Synthesis and Transformation.*

[4] Michael Armbrust, Ali Ghodsi, Matei Zaharia, Reynold Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph Bradley, Xiangrui Meng, Tomer Kaftan, and Michael Franklin. 2015. Spark SQL. 1383–1394. https://doi.org/10.1145/2723372.2742797

[5] David A. Bader and Kamesh Madduri. 2006. GTgraph : A Synthetic Graph Generator Suite.

[6] Teodoro Baldazzi, Luigi Bellomarini, Emanuel Sallinger, and Paolo Atzeni. 2021. *Eliminating Harmful Joins in Warded Datalog+/.* 267–275. https://doi.org/10.1007/978-3-030-91167-6_18

[7] Luigi Bellomarini, Davide Benedetto, Georg Gottlob, and Emanuel Sallinger. 2020. Vadalog: A modern architecture for automated reasoning with large knowledge graphs. *Information Systems* 105 (05 2020), 101528. https://doi.org/10.1016/j.is.2020.101528

[8] Mihai Budiu, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2022. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16 (2022), 1601–1614.

[9] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *Knowledge and Data Engineering, IEEE Transactions on* 1 (04 1989), 146 – 166. https://doi.org/10.1109/69.43410

[10] Binildas A. Christudas. 2019. MySQL. *Practical Microservices Architectural Patterns* (2019).

[11] E. F. Codd. 1970. A Relational Model for Large Shared Data Banks. *Communications of The ACM* (1970).

[12] Lutz Fröhlich. 2022. PostgreSQL.

[13] Georg Gottlob and Christoph Koch. 2003. Monadic Datalog and the Expressive Power of. (11 2003).

[14] Georg Gottlob, Thomas Lukasiewicz, and Andreas Pieris. 2014. Datalog+/-: Questions and Answers. In *KR.*

[15] Georg Gottlob and Andreas Pieris. 2012. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence* 193 (12 2012), 87–128. https://doi.org/10.1016/j.artint.2012.08.002

[16] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and S. Decker. 2003. Description logic programs: combining logic programs with description logic. In *The Web Conference.*

[17] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.* 3 (2005), 158–182.

[18] Ashish Kumar Gupta and Inderpal Singh Mumick. 1999. Incremental Maintenance of Recursive Views: A Survey.

[19] Muhammad Imran, Gábor E. Gévay, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Fast datalog evaluation for batch and stream graph processing. *World Wide Web* 25 (2022), 971–1003.

[20] Muhammad Imran, Gábor Gévay, and Volker Markl. 2020. *Distributed Graph Analytics with Datalog Queries in Flink.* 70–83. https://doi.org/10.1007/978-3-030-61133-0_6

[21] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. A specialized B-tree for concurrent datalog evaluation. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (2019).

[22] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2015. Incremental Update of Datalog Materialisation: the Backward/Forward Algorithm. In *AAAI Conference on Artificial Intelligence.*

[23] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A Highly-Scalable RDF Store. 3–20. https://doi.org/10.1007/978-3-319-25010-6_1

[24] Tilmann Rabl, Jonas Traub, Asterios Katsifodimos, and Volker Markl. 2016. Apache Flink in current research. *it - Information Technology* 58 (01 2016). https://doi.org/10.1515/itit-2016-0005

[25] Bruno Rucy and Merlin Kramer. 2023. https://github.com/brurucy/shapiro. Accessed: 2023-05-31.

[26] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog.*

[27] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. *Proceedings of the 25th International Conference on Compiler Construction* (2016).

[28] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. *Proceedings. ACM-Sigmod International Conference on Management of Data* 2016, 1135–1149. https://doi.org/10.1145/2882903.2915229

[29] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. *Proceedings - International Conference on Data Engineering* 2015 (05 2015), 867–878. https://doi.org/10.1109/ICDE.2015.7113340

[30] Jiacheng Wu, Jin Wang, and Carlo Zaniolo. 2022. Optimizing Parallel Recursive Datalog Evaluation on Multicore Machines. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22).* Association for Computing Machinery, New York, NY, USA, 1433–1446. https://doi.org/10.1145/3514221.3517853

[31] Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh Patel. 2018. Scaling-Up In-Memory Datalog Processing: Observations and Techniques. (12 2018).


## A PROGRAMS

**Program A.1.** RhoDFS inference rules

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

**Program A.2.** RhoDFS-s inference rules

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