A Back-to-basics Empirical Study of Datalog Materialization

BRUNO RUCY CARNEIRO ALVES DE LIMA[*] and G.K.M. TOBIN[*], Institute for Clarity in Documentation, USA
LARS THØRVÄLD, The Thørväld Group, Iceland
VALERIE BÉRANGER, Inria Paris-Rocquencourt, France
APARNA PATEL, Rajiv Gandhi University, India
HUIFEN CHAN, Tsinghua University, China
CHARLES PALMER, Palmer Research Laboratories, USA
JOHN SMITH, The Thørväld Group, Iceland
JULIUS P. KUMQUAT, The Kumquat Consortium, USA

Fig. 1. Seattle Mariners at Spring Training, 2010.

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Bruno Rucy Carneiro Alves de Lima, G.K.M. Tobin, Lars Thørväld, Valerie Béranger, Aparna Patel, Huifen Chan, Charles Palmer, John Smith, and Julius P. Kumquat. 2018. A Back-to-basics Empirical Study of Datalog Materialization. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

[*]Both authors contributed equally to this research.

1 INTRODUCTION

Motivation. SQL[1] has been the *de facto* universal relational database interface for querying and management since its inception, with all other alternatives having had experienced disproportionately little interest. The reasons for this are many, out of which only one is relevant to this narrative; performance.

The runner-ups of popularity were languages used for machine reasoning, a subset of the Artificial Intelligence field that attempts to attain intelligence through the usage of rules over knowledge. The canonical language for reasoning over relational databases is datalog[**?**]. Similarly to SQL, it also is declarative, however, its main semantics' difference is in the support for recursion while still ensuring termination irrespective of the program being run.

A notable issue with respect to real-world adoption of datalog is in performance. The combined complexity of evaluating a program is EXPTIME[**?**], while SQL queries are AC^0. It was not until recently[**?**] that scalable implementations were developed.

Digital analytics has been one of the main drivers of the recent datalog renaissance, with virtually all big-data oriented datalog implementations having had either been built on top of the most mainstream industry oriented frameworks[**? ? ?**] or with the aid of the most high-profile technology companies[**? ? ?**].

Another strong source of research interest has been from the knowledge graph community. A knowledge graph *KG* is a regular relational database *I* that contains *ground truths*, and rules.

The most important datalog operation is called *materialization*, the derivation of all truths that follow from the application of rules over the relational database, with the most straightforward goal being to ensure queries do not need any reasoning.

Problem. The maintenance of materialization is possibly the most important issue that plagues the real-world usage of datalog, and has been a source of much research interest, most commonly under the guise of incremental view maintenance.

The handling of additions is efficiently done with the semi-naive evaluation method[**?**], that can be notoriously efficient for a special class of programs that are popular in practice. Deletions on the other hand, are possibly much less efficient, since the retraction of a *ground truth* naively implies the deletion of all data derived from it.

The quintessential algorithm for computing the materialization adjustment with respect to a deletion is the delete-rederive[**?**] method, that generates new datalog programs to compute the deletions necessary to ensure correctness, doing so by first computing all possible deletions, and then all possible alternative derivations, with the adjusment being the difference between both of these sets.

The fact that two different algorithms are used implies that there are two different performance characteristics for additions and deletions, in spite of dred utilizing semi-naive evaluation as well, therefore there being possibly-severe biases in performance. To this date, there hasn't been a study that evaluated a datalog engine performance in properly dynamic scenarios.

A promising computation framework that could be well-suited to this problem is differential dataflow[**?**]. Timely dataflow is a high-throughput, low-latency streaming-and-distributed computational model that extends the dataflow paradigm. The timely in its name refers to it thoroughly establishing the semantics of each aspect of communication notification, taking the parallelization blueprint and making its execution transparent to the user.

A notification sent between operators is assigned a timestamp, that needs only to be partially ordered, with no assumptions being made with respect to insertion. The goal is to use these to correctly reason about the data with respect to the local operator states' of computation completion, in order to cleverly know with certainty when not only whether should work be done asynchronously or with sharding, but whether there is data yet to be processed or not.

Differential Dataflow, a generalization of incremental processing, is an extension to the Timely Dataflow model. While the shape of the data that flows in the latter is in the form of `(data, timestamp)`, the former's is `(data, timestamp, multiplicity)`, therefore restricting its bag-of-data semantics to multisets, with the multiplicity element representing a difference, for instance, +1 representing the addition of 1 datapoint arriving at a specific timestamp, or −1, a deletion.

The key difference between both models is that similarly to how timely makes distribution transparent, differential does the same with incremental computation, by providing to the user the possibility of not only adding to a stream, the only possible action in timely dataflow, but allow it to respond to arbitrary additions and retractions at any timestamp. There is one reasoner that uses differential dataflow[**?**], however, it has no storage layer, instead it compiles fixed datalog programs into fixed differential dataflow programs, that can only run in one computer.

The lack of a canonical open-source implementation of datalog makes attempts at making empirical statements about performance-impacting theoretical developments, such as the usage of differential dataflow, brittle and difficult, since there is no point of reference to compare and validate, and comparisons against commercial implementations are not reliable, since optimizations might be trade secrets.

A notorious exploration that highlights this issue is the COST, Configuration That Outperforms a Single Thread, article[**?**], in which the author posits that multiple published graph-processing engines are likely never able to outperform simple single-threaded implementations. Some high-profile datalog implementations were built upon systems mentioned on that article. Later on, the author made multiple pieces of informal writing in which the most performant datalog engines were investigated for COST[**? ?**], with results that showed a very different picture than the ones depicted by the original articles.

Methodology. To address the aforementioned problem, we conduct a back-to-basics empirical study of datalog program materialization, revisiting and measuring the core assumptions that go into the implementation of a reasoner. We coalesce this knowledge into a framework that provides multiple shared components for reasoning research. Straightforward single-threaded, parallel and distributed implementations, with the last being done with differential dataflow, of both substitution-based and relational-algebra interpretations are realized alongside common well-known optimizations.

Contributions. In this article we make several contributions to clarifying, benchmarking and easing pushing the boundaries of datalog evaluation engineering research further by providing performant and open-source implementations of single-threaded, parallel and distributed evaluators.

- **Techniques and Guidelines.** We study the challenge of building a reasoner from scratch, with no underlying framework, and ponder over all decisions necessary in order to bring that to fruition, alongside with relevant recent literature.
- **Differential Dataflow.** We accurately investigate the suitability of differential dataflow for datalog, and showcase how does it fare against the ubiquitous DRED and semi-naive evaluation based reasoners with the same language, base and with the same data format.
- **Framework.** All code outputs of this article are coalesced in a rust library named shapiro, consisted of a datalog to relational algebra rewriter, a relational algebra evaluator, and two datalog engines, one that is parallel-capable and supports both substitution-based and relational algebra methods, and other that relies on the state-of-the-art differential dataflow[**?**] distribution computation framework. The main expected outcome of this library is to

provide well-understood-and-reasoned-about baseline implementations from where future research can take advantage of, and reliable COST configurations can be attained.

- **Benchmarking.** We perform two thorough benchmark suites. One evaluates the performance of the developed relational reasoner with multiple different index data structures, and another that compares the performance of the distributed reasoner against four state-of-the-art distributed reasoners. The selected datasets are either from the program analysis, heavy users of not-distributed datalog, or from the semantic web community , which has multiple popular infinitely-scalable benchmarks, and are the main proponents of existential datalog.

2 RELATED WORK

Datalog engines. There are two kinds of recent relevant datalog engines. The first encompasses those that push the performance boundary, with the biggest proponents being RDFox[?], that proposes the to-date, according to their benchmarks, most scalable parallelisation routine, RecStep[?], that builds on top of a highly efficient relational engine, and DCDatalog[?], that builds upon an influential query optimizer, DeALS[?] 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[?], that originally used the query optimizer DeALs, and was built on top of the very popular Spark[?] distribution framework. Soon after, a prototypical implementation[?] over Flink[?], 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[?], that is also built on Flink, and makes use of its most advanced feature, incremental stream processing. To date, it is the fastest distributed implementation.

Data structures used in datalog engines. The core of each datalog engine is consisted of possibly two main data structures: one to hold the data itself, and another for indexes. Surprisingly little regard is given to this, compared to algorithms themselves, despite potentially being one of the most detrimental factors for performance . Binary-decision diagrams[?], hash sets[?] and B-Trees[?] are often used as either one or both main structures. An important highlight of the importance of data structure implementation is how in [?] Subotic et al, managed to attain an almost 50 times higher performance in certain benchmarks than other implementations of the same data structure.

3 DATALOG EVALUATION

In this section we review the basics of all concepts related to datalog evaluation, as it is done in the current time.

3.1 Datalog

Datalog[?] is a declarative programming language. A program P is a set of rules r, with each r being a restricted first-order formula of the following form:

$$\bigwedge_{i=1}^{k} B_i(x_1, ..., x_j) \rightarrow \exists(y_1, ..., y_j)H(x_1, ..., x_j, y_1, ..., y_j)$$

with k, j as finite integers, x and y as terms, and each B_i and H as predicates. A term can belong either to the set of variables, or constants, however, it is to be noted that all y are existentially quantified. The set of all B_i is called the *body*, and H the *head*.

A rule r is said to be datalog, if the set of all y is empty, and no predicate is negated, conversely, a datalog program is one in which all rules are datalog.

Example 3.1. Datalog Program

$$P = \left\{ \text{ SubClassOf}(?x, ?y) \land \text{SubClassOf}(?y, ?z) \longrightarrow \text{SubClassOf}(?x, ?z) \right\}$$

Example 3.1 shows a simple valid recursive program. The only rule denotes that *for all x, y, z, if x is in a SubClassOf relation with y, and y is in a SubClassOf relation with z, then it follows that x is in a subClassOf relation with z.*

The meaning of a datalog program is often[**?**] defined through a *Herbrand Interpretation*. The first step to attain it is the *Herbrand Universe* $\mathfrak{U}$, the set of all constant, commonly referred to as *ground*, terms.

Example 3.2. Herbrand Universe

$$S = \left\{ \begin{array}{l} \text{SubClassOf(professor, employee)} \\ \text{SubClassOf(employee, taxPayer)} \\ \text{SubClassOf(employee, employed)} \\ \text{SubClassOf(employed, employee)} \end{array} \right\}$$

$$\mathfrak{U} = \left\{ \text{ professor, employee, employed, taxPayer } \right\}$$

From the shown universe on example 3.2, it is possible to build The *Herbrand Base*, the set of all possible truths, from *facts*, assertions that are true, as represented by the actual constituents of the SubClassOf set.

Example 3.3. Herbrand Base

$$\mathfrak{B} = S \cup \left\{ \begin{array}{l} \text{SubClassOf(professor}, professor) \\ \text{SubClassOf(employee}, employee) \\ \text{SubClassOf(employed}, employed) \\ \text{SubClassOf(taxPayer}, taxPayer) \\ \text{SubClassOf(professor}, taxPayer) \\ \text{SubClassOf(taxPayer}, professor) \\ \text{SubClassOf(employee}, professor) \\ \text{SubClassOf(taxPayer}, employee) \end{array} \right\}$$

On example 3.3, all facts are indeed *possible*, but not necessarily *derivable* from the actual data and program. An interpretation I is a subset of $\mathfrak{B}$, and a *model* is an interpretation such that all rules are satisfied. A rule is satisfied if either the head is true, or if the body is not true.

Example 3.4. Models

$$I_1 = S \cup \left\{ \begin{array}{l} \text{SubClassOf(professor}, taxPayer) \\ \text{SubClassOf(employee}, employee) \end{array} \right\}$$

$$I_2 = S \cup \left\{ \begin{array}{l} \text{SubClassOf(professor}, taxPayer) \\ \text{SubClassOf(employee}, employee) \\ \text{SubClassOf(employed}, employed) \end{array} \right\}$$

$$I_3 = S \cup \left\{ \begin{array}{l} \text{SubClassOf(professor}, taxPayer) \\ \text{SubClassOf(employee}, employee) \\ \text{SubClassOf(employed}, employed) \\ \text{SubClassOf(professor, professor)} \end{array} \right\}$$

The first interpretation, I_1, from example 3.4, is not a model, since SubClassOf(employed, employed) is satisfied and present. Despite both I_2 and I_3 being models, I_2 is the *minimal* model, which is the definition of the meaning of the program over the data. The input data, the database, is named as the *Extensional Database EDB*, and the output of the program is the *Intensional Database IDB*.

Let an $DB = EDB \cup IDB$, 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, and 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.

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 $I_c(DB)$[**?**], and happens to be the *minimal* model.

Example 3.5. Repeated application of I_c

$$\begin{aligned} P = \{Edge(?x, ?y) \longrightarrow TC(?x, ?y), TC(?x, ?y), TC(?y, ?z) \longrightarrow TC(?x, ?z)\} \\ EDB = \{Edge(1, 2), Edge(2, 3), Edge(3, 4)\} \\ DB = EDB \\ DB = I_C(DB) \\ DB == EDB \cup \{TC(1, 2), TC(2, 3), TC(3, 4)\} \\ DB = I_C(DB) \\ DB == EDB \cup \{TC(1, 2), TC(2, 3), TC(3, 4), TC(1, 3), TC(2, 4)\} \\ DB = I_C(DB) \\ DB == EDB \cup \{TC(1, 2), TC(2, 3), TC(3, 4), TC(1, 3), TC(2, 4), TC(1, 4)\} \\ DB = I_C(DB) \\ DB == EDB \cup \{TC(1, 2), TC(2, 3), TC(3, 4), TC(1, 3), TC(2, 4), TC(1, 4)\} \\ IDB = DB \setminus EDB \end{aligned}$$

The introduced form of evaluation, with a walkthrough given on example 3.5, is called *naive*, meanwhile, the ubiquitous evaluation mechanism, as of the date of writing this paper, is the *semi-naive* one. The only difference is that *semi-naive* does not repeatedly union the EDB with the entire IDB, but does so only with the difference of the previous immediate consequence with the IDB. This can be hinted from the example, where each next application of I_c only renders new facts from the previous newly derived ones.

3.2 Infer

The most relevant performance-oriented aspect of both 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, and executing it.

The Infer[**?**] algorithm is the simplest example of the former, and relies on substitutions. A substitution S is a homomorphism $[x_1 \longrightarrow y_1, ..., x_i \longrightarrow y_i]$, 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 $[?x \longrightarrow 1]$ to it will yield the ground fact $TC(1, 4)$.

Infer relies on attempting to build and extend substitutions for each fact in each rule body over every single DB fact. Once all substitutions are made, they are applied to the heads of each rule. Every result of this application that is ground belongs to the immediate consequence.

3.3 Relational Algebra

Relational Algebra[**?**] is an imperative language, that 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[**? ? ?**] and supporting relational algebra as the SQL compilation target.

Let R and T be relations with arity r and t, θ be a binary operation with a boolean output, $R(i)$ be the i-th column in R, and $R[h, ..., k]$ be the subset of R such that only the columns $h, ..., k$ remain, and Const the set of all constant terms. The following are the most relevant relational algebra operators and their semantics:

- Selection by column $\sigma_{i=j)}(R) = \{a \in R | a(i) == a(j)\}$
- Selection by value $\sigma_{i=k}(R) = \{a \in R | a(i) == k\}$
- Projection $\pi_{h,...,k}(R) = \{(R(i), ..., R(j), \overrightarrow{C}) | i, j >= 1 \land i, j <= r \ \land \forall c \in C.c \in \text{Const}$
- Product $\times(R, T) = \{(a, b) | a \in R \land b \in T\}$
- Join $\bowtie_{i=j} = \{(a, b) | a \in R \land b \in T \land a(i) == b(j)\}$

Rewriting datalog into some form of relational algebra has been the most successful strategy employed by the vast majority of all current state-of-the-art reasoners[**? ? ? ? ? ?**] mostly due to the extensive industrial and academic research into developing data processing frameworks that process very large amounts of data, and the techniques that have arisen from these.

In spite of this, there is no open-source library that provides a stand-alone datalog to relational algebra translator, therefore every single datalog evaluator has to repeat this effort. Moreover, datalog rules translate to a specific form of relational algebra expressions, the select-project-join $\mathcal{SPJ}$ form.

A relational algebra expression is in the $\mathcal{SPJ}$ form if it consists solely of select, project and join operators. This form is very often seen in practice, being equivalent to `SELECT ... FROM ... WHERE ...` SQL queries, and highly benefits from being equivalent to conjunctive queries, that are equivalent to single-rule and non-recursive datalog programs.

We propose a straightforward not-recursive pseudocode algorithm to translate a datalog rule into a SPJ expression tree, in which relational operators are nodes and leaves are relations. The value proposition of the algorithm is for the resulting tree to be ready to be recursively executed, alongside having two essential relational optimizations, selection by value pushdown, and melding selection by column with products into joins, the most important relational operator. The canonical algorithms for translating datalog to relational algebra[**? ?**] are recursive, complex, and do not assume the output to be a tree, instead being mostly symbolic.

4 SHAPIRO

In order to coalesce all contributions in this paper, an extensible reasoning system was designed, from scratch. No frameworks were used, nor any code related to any other reasoner has been reutilized, furthermore, a modern, performant and safe programming language was used, Rust[**?**].

The main rationale for this choice is twofold: garbage-collected languages such as java are hard to benchmark, and tuning performance, alongside reasoning about it, often requires the developer to either learn highly specific minutiae relating to the compiler or interpreter, and, or, the garbage collector.

All most performant shared-memory reasoners are unsurpisingly all implemented in C++[**? ? ?**], since it provides manual memory management and is the *de facto* language for high-performance computing. Rust is a recent language, approximately 10 years old, that exhibits similar or faster performance than C++, with its *raison d'etre* being that, unlike C++, it is memory-safe and thread-safe[**? ? ?**]; both of these guarantees are of incredible immediate benefit to writing a reasoner.

Shapiro, the developed system, is fully modular and offers a heap of independent components and interfaces.

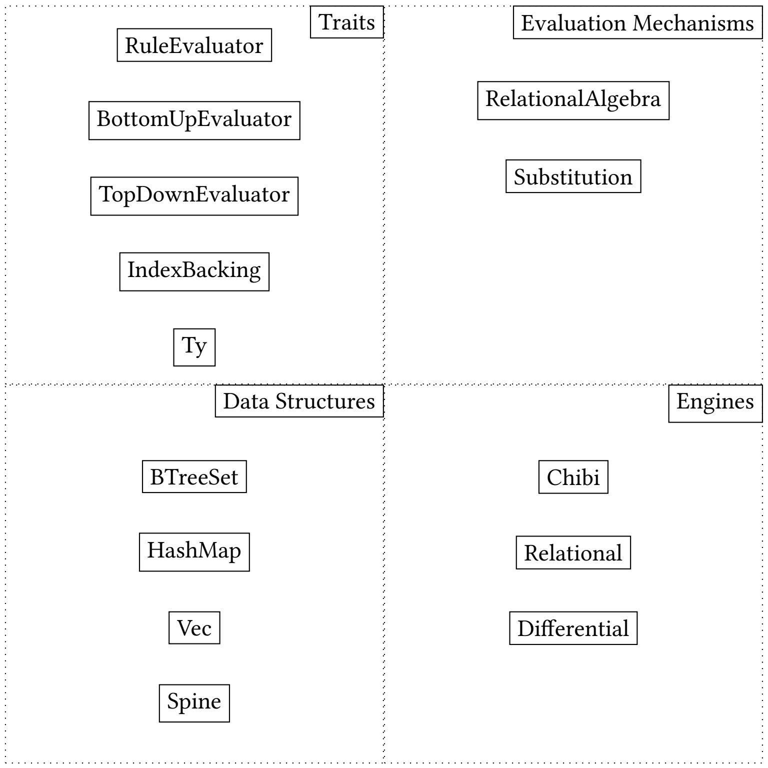

Fig. 2. Shapiro Components

On figure 2 we can see the most relevant modules. The northwestern quadrant, Traits, refers to Rust Traits, constructs that define *behavior*, and that can be passed to functions. It is important to take note that Rust favors *composition* over inheritance, unlike Java. Nevertheless, all other quadrants are built upon these traits, and are therefore generic over any type that implements them.

A `Rule Evaluator` represents $\mathbf{T}_p$, requiring only that types implement a function `evaluate`, that accepts an instance, and produces another instance. The evaluation mechanisms `Relational Algebra` and `Substitution` both implement this trait, evaluating datalog rules by either using the aforegiven translation algorithm, or using Infer. We also provide naive parallel evaluation of rules, in which each rule, in both mechanisms, are sent to be executed in a threadpool. We leverage the highly efficient, and easy to use, requiring only one additional line of code, rayon[**?**] library. This should suffice as a transparent baseline from which other parallelization strategies could be compared to.

The concept of fixpoint evaluation is a Struct, that consists of the ground fact database, *EDB*, a `Rule Evaluator`, and an array of instances. It is assumed that the datalog program is in the instance itself. Semi-naive evaluation is a method of fixpoint evaluation that takes no arguments, and instead uses two additional instances, one to keep the current delta, and the previous one. This is succintly implemented in rust in the same manner as this description.

`BottomUpEvaluator` and `TopDownEvaluator` both respectively denote the two kinds of evaluation, respectively, the explicit materialization of the program, and the answering to a query with respect to a program. The former requires the implementation of a function that takes in only a program as an argument, and the latter, a program and a goal.

The three engines, `Chibi`, `Relational` and `Differential` all implement only `BottomUpEvaluator`. In order to be able to evaluate a program, it is required to have access to an instance. An instance is defined in code as it is in literature, a set of relations. Taking inspiration from the highly successful open-source project DataScript[**?**], we implement relations as hashmaps, with generic hashing functions. The point of using a hashmap is that it allows for one to easily disregard duplicates altogether, further simplifying the implementation.

Indexes on the other hand, are a vital optimization aspect. A considerable amount of recent datalog research has dealt with speeding up relational joins with respect to datalog, such as in attempting to specialize data structures to it[**?**], and rminimizing the number of necessary joins to evaluate a *SPJ* expression[**?**]. Thus, a trait `IndexBacking` is defined, whose requirements are two methods, one that takes in a row for insertion, and another that requires a join implementation.

We implement the `IndexBacking` trait for implementations of all most used data structures for datalog indexes, such as the Rust standard library BTree[**?**] and HashMap, persistent implementations of both them, a regular vector, that naively sorts itself before a join, and the Spine, an experimental data structure, that empirically shows good performance characteristics in datalog workflows.

4.1 The Spine

The Spine is a simple two-level pointer-free data structure comprised of an array of arrays, and an index. Sorted Arrays are the fastest data structure for binary search lookups. Compared to search trees, they are much more cache efficient, since every single piece of data is in one contiguous region in memory, and no pointer indirection is needed in every binary search iteration.

Insertion in sorted arrays is commonly implemented throgh an emulated binary insertion sort, where the position of the to-be-inserted element is first found through a binary search, and then all elements to the right of its supposed position are shifted. The complexity of this operation is $O(n)$, therefore being exponentially slower than search trees, rendering it unusuable in dynamic cases, such as in datalog evaluation, in which possibly costly bulk insertions occur at a very fast pace.

In modern CPU architectures, there are multiple levels of memory, going from CPU registers, L-caches, up to disk. Accessing one element cached in low-level memory can be hundreds of thousands of times faster than doing so in the disk, therefore it is possible, in practice, for asymptotic linear-time access to be faster than logarithmic.

The Spine is a naive attempt to take advantage of that, by keeping a totally ordered set of fixed-capacity B sorted arrays, ordered by their maximum, with B being determined empirically to be the value such that the performance ratio of insertion and search is as desired. Moreover, given that the most efficient data structure for datalog indexing, BTree, also relies on sorted data, it often occurs that operations benefit from both spatial and temporal cache locality.

Souffle's datalog-tailored B-Tree[**?**] has a simple yet effective *hint* mechanism for speeding up searches and insertions, by taking advantage of locality. Whenever a new row is inserted, its traversal is kept as a *hint*, therefore if the next row to be inserted falls within the same *range* as the previous, the *range* itself can be searched, possibly entirely avoiding a traversal, and only searching in the child node. This same mechanism is implemented in the Spine, with the help of its index.

Fenwick Trees[**?**] are highly specialized data structures used for dynamic cumulative sum. Given a fixed-length array of counts, the fenwick tree allows for calculating the prefix sum in $O(\log_2 n)$ time, while also providing adjustments to counts in the same bounds. We leverage the [**?**] in order to maintain an index that will keep track of the size of each internal array. The cost of updating the fenwick tree is minimal, in the context of the Spine, taking $O(\log_2(B))$ time with $B \ll n$.

In order to ensure that all sorted sub-arrays have at most B elements, a *maintenance* operation must occur. Whenever a sub-array *overflows*, it is split in half, with all arrays to the right of the overflown one shifted one place, and a new array being created, moving all elements that belonged to the upper half. The cost of this operation, alongside the respawning of the Fenwick Tree, is $O(B)$, which is, in practice, negligible to most workloads.

The Spine leverages the Fenwick Tree in order to support $O(\log_2(B))$-time accesses by position. In order to locate the sorted array in which the element in the global i-th position among the union of all sorted arrays is, naively, one can run a binary search over the fenwick tree, calling prefix sum on each iteration. Given that, as previously mentioned, prefix sum takes $O(\log_2(N))$ time, a binary search that calls prefix sum on each iteration would take $O(\log_2^2(N))$. It is possible to leverage the structure of the fenwick tree, and calculate that, in a very efficient manner, taking only $O(\log_2 N)$ time.

The algorithm to do so is as follows:

Being logarithmic with respect to B access time, we can have an improved version of hints. In the original proposal, the cost of hint is to retain a pointer to the leaf node that it is in. Let there be a search, with a query point k and a hint h. When a search starts, in case the element isn't in the leaf child of the hint pointer, then a traversal starts from zero, this implies the worst-case cost will be $O(\log(N))$. On the Spine in the other hand, first k is compared with h, and then a binary search is started. if $k > h$, then h is the lower bound, if else, then k is the upper bound. This provides significantly more value than the simple child lookup in souffle's B-Tree.

4.2 Differential Dataflow

The `Differential` engine is an attempt at implementing a substitution-based engine on top of the distribution framework that addresses datalog's evaluation inefficiencies in a manner that no other currently available framework does, differential dataflow[**?**]. Most of the distributed datalog engines are built on top of either graph-processing or map-reduce frameworks, both kinds of projects that were not specifically made with expressive query answering in mind, but more with efficiently handling complex non-recursive queries over large amounts of data.

One of the biggest challenges in efficiently evaluating datalog is in the deletion of data. Incremental bottom-up processing of additions is already the manner in which semi-naive evaluation works, and is relatively efficient. The other direction is significantly more complex. In order to delete a fact, one has to take into account that it might have influenced the inferrence of other facts, which imply having to look over the whole data, while the opposite direction does not require that, therefore incurring possibly very large performance differences between both directions.

Similarly to how semi-naive evaluation is the *de facto* method for incremental bottom-up evaluation, the Delete-Rederive[**?**] algorithm holds the same regard with respect to the incremental maintenance of bottom-up evaluation with respect to deletions. There are two stages to it, the first, Deletion, naively overdeletes all facts that could have been derived from it, but at the same time, keeps track of possible facts that could have had alternative derivations. The second stage, rederive, restarts the evaluation process.

Differential dataflow directly addresses this issue by evaluating both additions and deletions in the same manner, while at the same time efficiently parallelizing semi-naive evaluation, however, it is first necessary to digress over Timely Dataflow [**?**], the underlying networked computation system in which differential is built upon.

Timely is a high-throughput, low-latency streaming-and-distributed model that extends dataflow. The timely in its name refers to it thoroughly establishing the semantics of each aspect of communication notification, taking the parallelization blueprint and making its execution transparent to the user. A notification sent between operators is assigned a timestamp, that needs only to be partially ordered, with no assumptions being made with respect to insertion. The goal is to use them to correctly reason about the data with respect to the local operator states' of computation completion, in order to cleverly know with certainty when not only whether should work be done asynchronously or with sharding, but whether there is data yet to be processed or not.

Differential Dataflow, a generalization of incremental processing, is an extension to the Timely Dataflow model. While the shape of the data that flows in the latter is in the form of `(data, timestamp)`, the former's is `(data, timestamp, multiplicity)`, therefore restricting its bag-of-data semantics to multisets, with the multiplicity element representing a difference, for instance, +1 representing the addition of 1 datapoint arriving at a specific timestamp, or -1, a deletion. The key difference between both models is that similarly to how timely makes distribution transparent, differential does the same with incremental computation, by providing to the user the possibility of not only adding to a stream, the only possible action in timely dataflow, but allow it to respond to arbitrary additions and retractions at any timestamp.

Similarly to how timestamps in timely are the key element to the efficient parallelization of iterative dataflows, they can also be used in incremental scenarios in order to overcome their inherently sequential nature. This is of paramount importance to the performance of datalog evaluation.

Let's assume that there is some dataflow that computes the transitive closure of some graph d_0, and one update consisted of four edge differences, labeled as δ, arrives, with the resulting updated graph being d_1. In a regular incremental system, each triple would increment the iteration counter by 1, and even though the relational operations inside the dataflow might happen in parallel, δ_1, δ_2, δ_3 and δ_4 will only be evaluated after δ_0 and each of its sequent difference's iteration finishes.

If that dataflow were to be executed with differential dataflow, inside the iteration scope there would be a product timestamp $\langle a, b \rangle$, with a representing the time in which some initial triple was fed into the computation, and b denoting the iteration count, therefore tracking the transitive chain length, respecting the following partial order:

$$\langle a_i, b_1 \rangle \leq \langle a_j, b_j \rangle \iff a_i \leq a_j \land b_i \leq b_j$$

If we take that δ_0, δ_3, δ_1 and δ_2 are differences with the following respective timestamps: $\langle 0, 1 \rangle$, $\langle 0, 2 \rangle$, $\langle 1, 1 \rangle$, $\langle 1, 2 \rangle$ it is noticiable that δ_0 and δ_3 are comparable, but both of them, and vice-versa, are incomparable with respect to δ_1 and δ_2. This, in turn, means that, as it can these pairs of differences, despite notifying change on the same iteration operator, could safely be executed in parallel in differential dataflow, but not in a regular incremental system that uses totally-ordered timestamps.

The most notable difference between iterative and incremental processing is that in the latter computation advances by, ideally, making adjustments proportional to the newly received data, referred to as difference. Naturally, this could result in massively reduced latency, however, in order to support this in the first place, incremental systems have to maintain indexes of all updates, be it an addition or a retraction, that could impact the calculation.

The differential dataflow implementation, built on top of timely dataflow's rust one, utilizes a novel method to maintain indexes such that they are not restricted to each individual operator, and could also be shared between multiple readers. This method utilizes shared arrangements, out of which its core concept, collection trace, is a structure that represents the state of a collection since the latest frontier, as explained in the timely dataflow section, as an append-only ordered sequence of batches of updates, with each batch possibly being merged with other batches, that make up an LSM-Tree [**?**], therefore benefitting from its compaction mechanism to ensure that only a logarithmic number of batches exist.

5 EXPERIMENTS

I actually did some of the experiments. results were promising, trust me :D this will revolutise the field!

REFERENCES

[1] John Doe and Jane Doe. 2015. A super interesting Article. *Journal of unreproducible Results* (2015).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

Algorithm 1: An algorithm to translate a datalog rule into relational algebra

```
Input: A datalog rule $\mathcal{R}$
Result: A relational algebra expression $\mathcal{R}_a$
Function toIncompleteExpression($r$ : Datalog Rule) is
    let $t$ be a fresh tree
    For For every fact $a_i$ in the rule body $b$:
        create a relation node $r_i$ with the same arity as $a_i$, and its terms representing columns. variable terms are column identifiers, and constant terms are
          temporary-lived proxies for selections. if $i < len(b) - 1$ then
            add a product node $p_i$ to $t$. set $r_i$ as the left child of $p_i$.
        else
            if $len(b) > 1$ then
                set $r_i$ as the right child of $p_{i-1}$
        if $i > 0$ then
            set $p_i$ as the right child of $p_{i-1}$
    return $t$
Function constantToSelection($e$ : Expression) is
    let $t$ be a copy of $e$
    let $C$ be a map $C$ : Const $\longrightarrow$ Var
    For every relation $r_i$ in $t$:
        For every constant $c_j$ in $r_i$:
            add a selection by value node $s_j$ to $t$ with column index $j$ and value $c_j$
            set $s_j$'s parent to $r_i$'s, and $r_i$ as its left child
            if $\neg(c_j \in C)$ then
                create a fresh variable term $v_j$ and store it in $C$ with $c_j$ as the key
            else
                replace $c_j$ for the value in $C$ under $c_j$
    return $t$
Function equalityToSelection($e$ : Expression) is
    let $t$ be a copy of $e$
    let $V$ be a map $V$ : Var $\longrightarrow \mathbb{Z}$
    let $t_p$ be a pre-order traversal of $t$
    For every relation $r_i$ in $t_p$:
        For every variable $v_j$ in $r_i$:
            if $\neg(v_j \in V)$ then
                add $v_j$ to $V$ with $j$ as the value
            else
                let $k$ be the value of $v_j$ in $V$ let $p_i$ be the first product to the left of $r_i$ in $t_p$
                add a selection by column node $s_j$ to $t$ with left column index $k$ and right column index $j$
                set $s_j$'s parent to $p_i$'s, and $p_i$ as its left child
    return $t$
Function projectHead($e$ : Expression, $r$ : Datalog Rule) is
    let $n$ be 0
    let $t$ be a copy of $e$
    let $h$ be the head of $r$
    let $t_p$ be a pre-order traversal of $t$
    let $V$ be a map $V$ : Var $\longrightarrow \mathbb{Z}$
    For every relation $r_i$ in $t_p$:
        For every term $x$ in $r_i$:
            if $x$ is a variable then
                add $x$ to $V$ with $n$ as value
            else
                continue
            $n$ += 1
            add projection node $z$ to $t$ and set it as root with an empty list
            For every term $x$ in $h$:
                if $x$ is a constant then
                    push $x$ into $z$
                else
                    let $k$ be the value of $x$ in $V$
                    push $k$ into $z$
    return $t$
Function productToJoin($e$ : Expression) is
    let $t$ be a copy of $e$
    let $t_p$ be a pre-order traversal of $t$
    For every selection by column $s_i$ in $t_p$:
        find the first product $p_j$ after $s_i$ in $t_p$
        remove $s_i$ from $t$, swap $p_j$ for a join $g_j$ in $t$ with left and right column indexes by those of $s_i$
    return t
$\mathcal{R}_a$ = toIncompleteExpression($\mathcal{R}$)
$\mathcal{R}_a$ = constantToSelection(expression, $\mathcal{R}$)
$\mathcal{R}_a$ = equalityToSelection(expression, $\mathcal{R}$)
$\mathcal{R}_a$ = projectHead(expression, $\mathcal{R}$)
$\mathcal{R}_a$ = productToJoin(expression, $\mathcal{R}$)
return $\mathcal{R}_a$
```

Algorithm 2: An algorithm to binary search a fenwick tree in logarithmic time

```
Input: a fenwick tree $\mathcal{F}$, a desired position $i$-th
Result: the count inside $F$ in which $i$-th would be in
1  let length = $\mathcal{F}$.length()
2  let prefix_sum = $i$
3  let mut idx = 0;
4  let mut $x$ = most significant bit of length * 2
5  While $x \geq 0$:
6      let lsb = least significant bit of x
7      if $x \leq$ length, $\mathcal{F}[x-1] \leq$ prefix_sum then
8          idx = x
9          prefix_sum = prefix_sum - $\mathcal{F}[x-1]$
10         x = x + lsb / 2;
11     else
12         if lsb is even then
13             break
14         x = lsb / 2 - lsb
15 return idx
```