Towards Better Understanding of the Performance and Design of Datalog Systems

Zhiwei Fan[1], Sunil Mallireddy[2] and Paraschos Koutris[3]

[1]*University of Wisconsin-Madison, WI, USA*
[2]*University of Wisconsin-Madison, WI, USA*
[3]*University of Wisconsin-Madison, WI, USA*

Abstract
Recent years have seen a resurgence of interest in the Datalog language and its syntactic extensions from both the industry and academia. Such interest has motivated a line of work to build efficient Datalog systems that support expressing data analytics of different types such as program and graph analyses in a concise and simple way. However, besides the performance improvement of different systems presented in these works over existing competitors, little understanding has been gained about the property of varying Datalog workloads (i.e., program and data), and the computation resource usage of different systems (i.e., memory and CPU utilization), which are crucial for understanding the essence behind the performance difference observed in different systems. When such knowledge is gained, clear guidance could be provided for users to choose between different Datalog systems depending on their use cases and system builders to improve the existing system or build more efficient new systems. In this paper, we propose the general profiling of the Datalog program evaluation and present the corresponding visualizations. We further discuss the insights gained from the produced visualizations and how these insights shed light on the pros and cons of existing Datalog systems, which further provides guidance on making improvements over the existing system and designing/building more efficient new systems.

Keywords
Datalog Systems, Recursive Computation, Benchmarks, Profiling

1. Introduction

Datalog is seeing a resurgence of interest over the past years and has found new applications in multiple domains such as graph analytics, program analysis, data integration, security, etc. The regained popularity of Datalog is largely attributed to its superior ability to express applications involving recursive computations concisely. To provide high-performance and scalable computation, multiple research efforts [1, 2, 3, 4, 5] have explored ways to develop Datalog systems that are able to handle recursive computation efficiently, often focusing on a particular application domain. For example, Souffle[3] is mainly designed and built for static program analysis, and DDlog [2] is a Datalog implementation built on top of Differential Dataflow [6] that focuses on efficient incremental computation.

However, in most of these works, besides the better performance numbers shown for the systems being presented compared to existing competitors on the chosen workloads, little or

Datalog 2.0 2022: 4th International Workshop on the Resurgence of Datalog in Academia and Industry, September 05, 2022, Genova - Nervi, Italy
zhiwei@cs.wisc.edu (Z. Fan); sunlim@cs.wisc.edu (S. Mallireddy); paris@cs.wisc.edu (P. Koutris)

no description has been provided for the *profile* of the recursive computation. Here, by profile, we mean information such as the number of iterations, how many facts are produced in every iteration, etc. As shown in RecStep [1], the relative performance of different Datalog systems may not translate across different workloads (i.e., a system that performs well on one Datalog program and a particular dataset does not show comparable performance on the others). The lack of a closer look at the recursive computation profiles makes it difficult to analyze and explain the performance difference across different systems and workloads. In turn, this makes choosing the best system for applications of interest (for users) and improving existing systems (for system builders) challenging. For example, when attempting to build a new Datalog system, we need to answer questions such as *what techniques in existing systems can we leverage? what are the limitations of existing systems? are there Datalog workloads of different characteristics that need to be handled differently? what could be done to improve existing techniques?*

To address the aforementioned issue, we argue that a general-purpose recursive computation profiling framework that can provide insights *across systems and workloads* is needed. In this work, we make the first step towards building such a profiling framework by presenting four important profiling components: *recursion profile, runtime, CPU utilization* and *memory utilization.* First, we describe these four components and briefly discuss their importance. Then we present case studies based on the profiling visualizations of Datalog workloads from two application domains: graph analytics and program analysis. As shown in Section 3, there is no single system always being the winner across all Datalog workloads, even within the same application domain. With the help of profiling visualizations, we analyze the causes behind the inefficient executions, extracting insights regarding the proper use cases and limitations of the studied systems (Section 3.3). By analyzing high-level causes (revealed by the profiling components) of the inefficiency exposed by a system, one is able to connect these high-level observations to the specific technical components in the system (e.g., data structures, algorithms), understanding the system limitations better.

In this paper, we focus on single-node systems, mainly looking at three recently published well-documented Datalog engines that are publicly available: RecStep [1], Souffle [3], and DDlog [2]. Souffle is a new recent high-performance Datalog system that is mainly designed for program analysis and uses optimization techniques such as efficient program synthesis, specialized parallel data structures for indexing and compression, and automatic index selection. RecStep is a Datalog engine built on top of an efficient single-node in-memory database called Quickstep[7], leveraging multiple years of efforts in the advancement of database techniques such as query optimization and efficient parallel query execution. DDlog translates a set of Datalog rules into the corresponding Differential Dataflow [6] program that allows for incremental computation. Other Datalog systems such as BigDatalog [4], BDDBDDB [8] have been shown to be significantly outperformed by two of the systems (i.e., RecStep, Souffle) studied in this paper [1] with some notable system limitation and performance issues and therefore are excluded from the study. We do not consider solvers designed for answer-set programming such as clingo [9] and dlv [10] as the recent study [11] shows that they are not capable of handling Datalog workloads in our interested application domains (e.g., graph analytics, program analysis) due to inefficient use of memory. Our study focuses on positive Datalog programs (i.e., without negation) that involve recursion.

We use the profiling functionalities embedded in RecStep [1], which is able to provide general profiling information of different systems on the Datalog workload evaluation that is not tied to a specific system. Although the profiling components presented are fairly simple, they already provide meaningful insights that can aid further system analysis and improvement, which is not possible by looking solely at the performance numbers.

2. Recursive Computation Profiling

Most existing systems evaluate Datalog programs using a type of bottom-up evaluation called *semi-naïve evaluation* (SN) [12] either explicitly (e.g., RecStep, Souffle) or implicitly (e.g., DDlog).

At each iteration of the recursive computation, there are *three types of facts* that are important to consider. Facts of the first type are generated from the evaluation of each recursive rule (*generated facts*, GF). The generated facts can contain duplicates, so we also need to consider the facts after deduplication, called *unique generated facts, UGF*. Finally, the set-difference is performed between the unique generated facts and the existing facts to produce the *new facts, NF*. Some Datalog systems perform the deduplication and set-difference separately (e.g., RecStep), and other systems (e.g., Souffle, DDlog) fuse these two steps into one, often through the maintained indexes built on the IDB relations throughout the whole computation procedure.

As we will see in Section 3, the sizes of these three different types of facts in different iterations serve as the primary *fingerprint* of the Datalog workload and help better understand the behavior of various systems along with other profiling information such as runtime and resource usage. Next, we briefly discuss a few major components for recursive computation profiling. We note that these components are *not the only* ones to look at when analyzing the system performance on varying Datalog workloads. When being available, additional information such as the size of input data, and hot code paths could be useful and provide additional insights.

Recursion Profile The sizes of facts of three different types in each iteration of the recursive computation characterize the Datalog workload, which consists of a specific Datalog program (i.e., a set of Datalog rules) and a specific dataset (i.e., ground facts of the EDB/input relations). At each iteration, let GF_{size}, UGF_{size}, and NF_{size} denote the sizes of GF, UGF, and NF respectively. Note that we always have $\text{GF}_{size} \geq \text{UGF}_{size} \geq \text{NF}_{size}$. Intuitively, a large gap between GF_{size} and UGF_{size} indicates that many facts were produced multiple times in the same iteration, while a large gap between UGF_{size} and NF_{size} indicates that many facts have already been produced in previous iterations. Note that NF_{size} also indicates the amount of work to be done during the next iteration in SN. As an example, Figure 1 is the recursive profile showing the sizes of facts of three types across seven iterations during SN of transitive closure evaluated on G10k dataset, which will be analyzed in detail in Section 3.

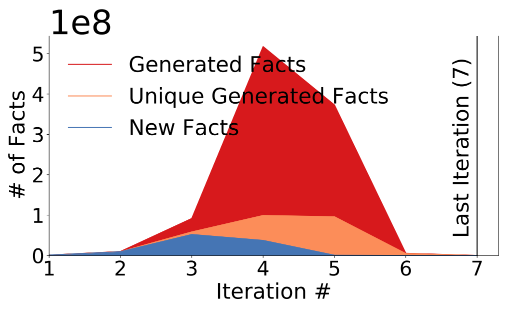


Figure 1: Recursive Profile of TC-G10k

Runtime Runtime is probably the most straightforward performance measure of different systems on a given workload. The runtime of many existing Datalog systems can be divided into compilation time and evaluation time. Systems such as Souffle and DDlog first generate the code given the input program, followed by compilation-level optimizations and executable binary generation. The overhead induced by code generation and compilation can be safely ignored, assuming that the generated executable files will be used repetitively later with different inputs. However, such an assumption might not always hold and may not be acceptable in circumstances where the overhead far exceeds the evaluation time. Thus, it is crucial to have access to a clear view of runtime breakdown when considering a specific application.

CPU Utilization Like other data-parallel compute engines, recent Datalog systems [1, 2, 3, 11] exploit the parallelism packed inside modern servers to achieve high performance and scalability. However, achieving consistent high CPU efficiency and utilization across different workloads is challenging. Low performance could occur due to either low CPU efficiency (suggesting that the system might handle more work than necessary), or low CPU utilization (meaning that the system does not utilize multiple CPU cores well).

Memory Consumption Many recent works focus on building in-memory Datalog systems [1, 2, 3, 4]. However, most of them either ignore the evaluation of memory utilization [4] or miss the comparison with other existing Datalog systems [3, 2]. Since most of the evaluations presented in these works are standalone (i.e., a system only evaluates one workload at a time in a server without interference), the lack of understanding of the memory footprint makes it hard to choose the proper hardware (e.g., a server with small or large memory), estimate the scalability of a Datalog program (e.g., the maximum dataset the system can handle), and the applicability (e.g., whether concurrent evaluation is feasible or not).

3. Case Studies

We next present the Datalog programs that arise from *graph analytics* (TC, SG, REACH) and *program analysis* (AA, CSPA, CSDA) followed by the case studies of the corresponding Datalog workloads. These Datalog programs are supported by all three systems of interest in this paper. We first look at the *linear recursive* Datalog programs (TC, SG, REACH, CSDA), in which each program consists of one *non-recursive* rule and one *linear recursive* rule (i.e., the rule body contains only one recursive IDB predicate). Then, we study the Datalog workloads of *non-linear recursive* programs (AA, CSPA) each of which contains at least one recursive rule that has more than one recursive IDB predicate in the rule body. As we will see from the recursive computation profiles of these workloads, even when two Datalog programs look very similar, the relative performance of different systems can be very different. This happens when two programs are from the same domain (e.g., TC, REACH) or different application domains (e.g., REACH, CSDA).

All experiments are conducted on a bare-metal server in Cloudlab [13], a large cloud infrastructure. The server runs Ubuntu 18.04 LTS and has two Intel Xeon E5-2660 v3 2.60 GHz (Haswell EP) processors. Each processor has 10 cores, and 20 hyper-threading hardware threads.

Table 1
Summary of EDB/Input and IDB/Output in Different Workloads

Program	Dataset	EDB Tuple #	IDB Tuple #	Reference
SG	G5k	arc: 9.98e4	tc: 1.00e8	[1]
TC	G10k	arc: 2.50e4	sg: 2.47e7	[1]
REACH	`livejournal`	arc: 6.90e7 id: 100	reach: 4.40e6	[1]
	`orkut`	arc: 1.17e8 id: 100	reach: 2.90e6	[1]
	`twitter`	arc: 1.47e9 id: 100	reach: 2.24e7	[1]
AA	D7	assign: 1.00e7 load: 3.30e7 store: 2.10e7 addressOf: 4.00e6	pointsTo: 5.30e6	[1]
CSPA	linux	assign: 1.98e6 dereference: 7.50e6	valueFlow: 5.50e6 valueAlias: 3.09e7 memoryAlias: 1.37e7	[1]
	postgresql	assign: 1.20e6 dereference: 3.46e6	valueFlow: 3.71e6 valueAlias: 2.23e8 memoryAlias: 8.94e7	[1]
	httpd	assign: 3.62e5 dereference: 1.14e6	valueFlow: 1.36e6 valueAlias: 2.34e8 memoryAlias: 8.89e7	[1]
CSDA	linux	arc: 4.34e7 nullEdge: 5.89e5	null: 5.57e7	[1]
	postgresql	arc: 3.45e7 nullEdge: 2.17e5	null: 2.15e7	[1]
	httpd	arc: 9.90e6 nullEdge: 1.38e5	null: 9.39e6	[1]

The server has 160GB memory and each NUMA node is directly attached to 80GB of memory. We only consider the CPU and memory utilization of the systems during their *actual execution period* and thus the time period used for code generation and compilation is excluded for CPU and memory profiling. Information about the input and output is summarized in Table 1.

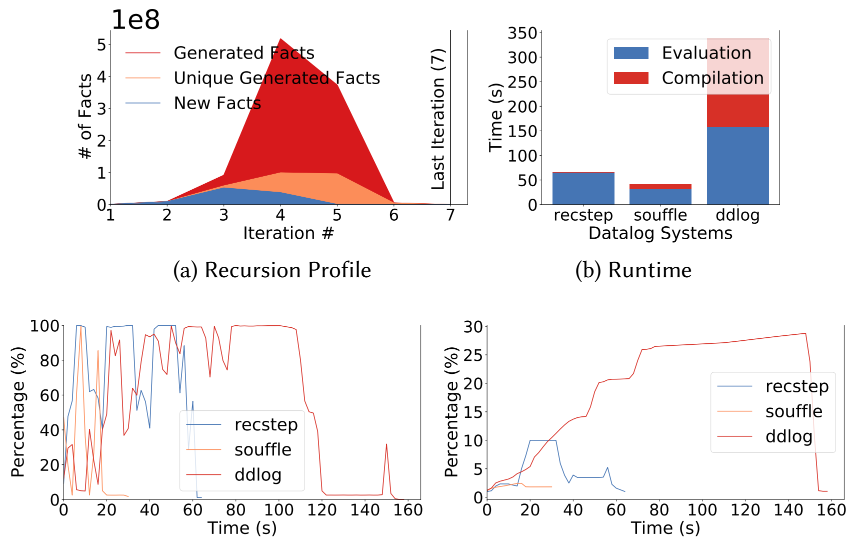
3.1. Simple Linear Recursion

Transitive Closure (TC):

$$\mathtt{tc}(\mathtt{X},\mathtt{Y}) \text{ :- } \mathtt{arc}(\mathtt{X},\mathtt{Y}).$$
$$\mathtt{tc}(\mathtt{X},\mathtt{Y}) \text{ :- } \mathtt{tc}(\mathtt{X},\mathtt{Z}), \mathtt{arc}(\mathtt{Z},\mathtt{Y}).$$

Same Generation (SG):

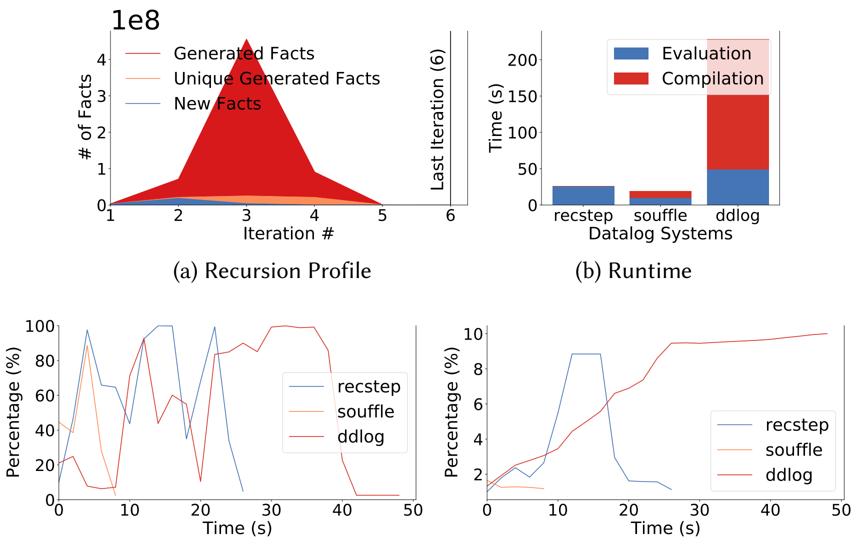
$$\mathtt{sg}(\mathtt{X},\mathtt{Y}) \text{ :- } \mathtt{arc}(\mathtt{P},\mathtt{X}), \mathtt{arc}(\mathtt{P},\mathtt{Y}), \mathtt{X} \neq \mathtt{Y}.$$



(a) Recursion Profile (b) Runtime

(c) CPU Utilization (d) Memory Consumption

Figure 2: Transitive Closure on G10k



(a) Recursion Profile (b) Runtime

(c) CPU Utilization (d) Memory Consumption

Figure 3: Same Generation on G5k

$$\mathtt{sg(X,Y)} \text{ :- } \mathtt{arc(W,X), sg(W,U), arc(U,Y).}$$

Figure 2 and Figure 3 show the recursive computation profiles of TC and SG of three systems on G10k and G5k respectively. The G10k and G5k datasets are random graphs of $10k$ ($\sim 100k$ edges) and $5k$ ($\sim 25k$ edges) vertices generated based on the Erdős-Rényi model [14], in which each edge is included with probability 0.001. Although the sizes of the input datasets are fairly small ($<$ 1MB), large intermediate results (i.e., three types of facts) as shown in Figure 2a and Figure 3a are generated. We observe the following features across all recursion profiles for the above tasks: (i) the number of iterations is relatively small, (ii) there is a large gap between GF$_{size}$ and UGF$_{size}$, which suggests efficient deduplication is critical to the overall good performance, and (iii) the gap between UGF$_{size}$ and NF$_{size}$ is small or non-existent in most cases.

For TC and SG, while all three systems have relatively high CPU utilization throughout the evaluation (Figures 2c, 3c), the runtime varies. Besides the relatively long compilation time of DDlog ($\sim 200s$), DDlog's evaluation time is about $2 - 3X$ longer than that of the other two competitors (i.e., Souffle and RecStep), using a large amount of memory (Figures 2c, 3c). The performance number of DDlog is significantly worse than that of its runtime primitive Differential Dataflow [6] as shown in [15]. The inefficiency of DDlog could be attributed to the fact that its design heavily focuses on incremental computation (e.g., maintaining intermediate states of large sizes, separate management of existing computation and monitoring new input, etc), sacrificing the performance for batch processing.

RecStep uses significantly more memory (Figures 2d, 3d) on the small input datasets (i.e., G10k and G5k) compared to other workloads (Figures 4d-12d), the sizes of input datasets of which vary from 22MB to 1.7GB. This is because RecStep performs *deduplication* as a separate step, relying on its backend in-memory relational database QuickStep [7], which pre-allocates the memory to the hash table for deduplication based on the size of the generated facts. Due to the memory inefficiency observed in such cases, RecStep soon runs out of memory when evaluating TC and SG on graphs with a large number of vertices. At the same time, the edge inclusion probability remains the same. We run RecStep using its default interpretation mode without the specialized *parallel bit-matrix evaluation* (PBME) designed for dense graphs of small vertices. While PBME [1] is an efficient technique to address this issue, it is specifically designed for graphs with a relatively small number of vertices, and its generality is limited.

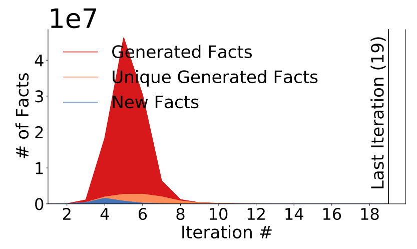
In contrast, Souffle has acceptable overhead from the code generation and compilation ($\sim 10s$), showing the overall best performance on TC-G10k and SG-G5k with a small memory footprint mainly thanks to its specialized parallel data structure Brie [16] for relation storage and indexing, which provides good compression capability for high-density relations of large sizes and support of efficient parallel operations (e.g., insertion, lookup).

Reachability (REACH):

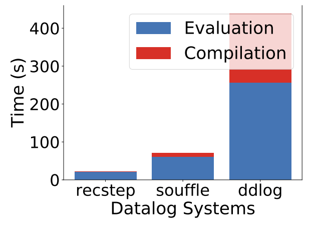
```
reach(Y) :- id(Y).
reach(Y) :- reach(X), arc(X, Y).
```

Context-Sensitive Dataflow Analysis (CSDA):

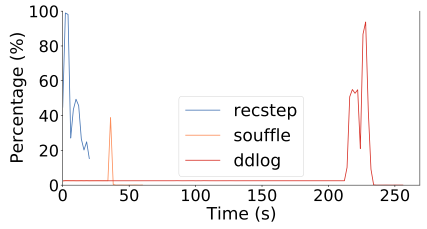
```
null(X, Y) :- nullEdge(X, Y).
null(X, Y) :- null(X, W), arc(W, Y).
```



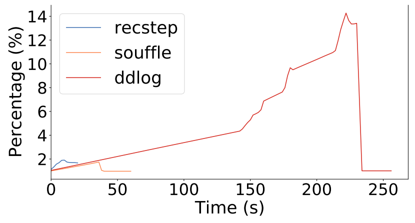
(a) Recursion Profile



(b) Runtime

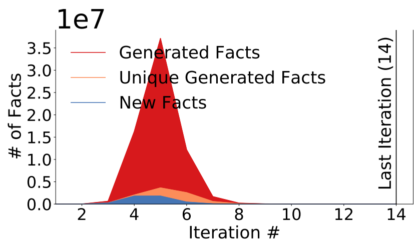


(c) CPU Utilization

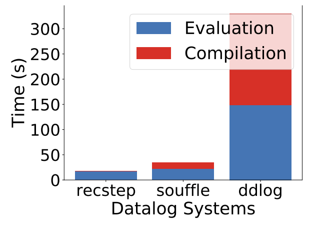


(d) Memory Consumption

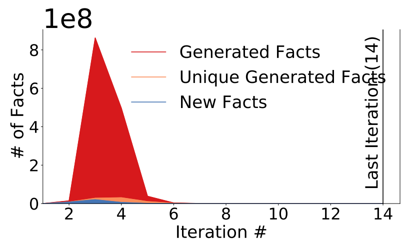
Figure 4: Reach on `orkut`



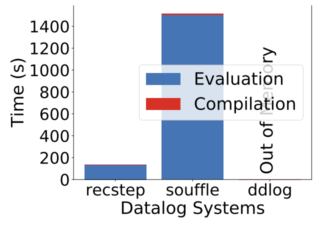
(a) Recursion Profile on `livejournal`



(b) Runtime on `livejournal`



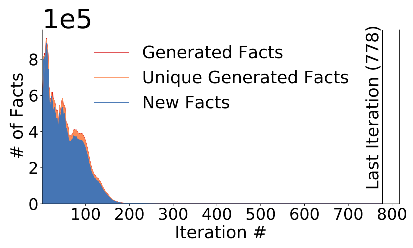
(c) Recursion Profile on `twitter`



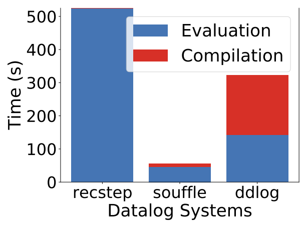
(d) Runtime on `twitter`

Figure 5: Reach on `livejournal` and `twitter`

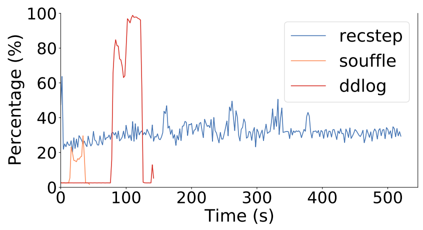
We run REACH on `orkut`, a relatively large real-world online social network dataset in which the friendship of users is represented as edges. REACH finds the friends of a given set of user ids. Figure 4 is the recursive computation profile of REACH-`orkut` and we observe that the relative system performance looks quite different from what is observed on TC-G10k and SG-G5k. RecStep significantly outperforms Souffle and DDlog and the reasons are two-fold: (i) GF$_{size}$ is relatively small across just a few number of total iterations, resulting in the negligible overhead of the separate deduplication step and overall efficiency of RecStep (ii) RecStep utilizes CPU much more efficiently compared to Souffle and DDlog, which suffer from the long warm-up phase (e.g., index building) due to the much larger EDB/input sizes. Systems such as Souffle using a specialized data structure for indexing (i.e., Brie), may suffer from much bigger index building overhead if the characteristic of indexed data is *adversarial* to the property of the indexing data structure. For REACH on relatively sparse large real-world social graphs, we observe larger indexing overhead in Souffle as the input graph size increases (e.g., `livejounral` $\rightarrow$ `orkut` $\rightarrow$ `twitter` in Table 1), which results in larger relative performance gap between Souffle and RecStep when the total number of iterations of recursive computation remains small (e.g., Figures 5a, 4a, 5c). The observation suggests that the input data itself serves as an important profiling component sometimes and should be considered together with other profiling components (e.g., recursion profile, CPU utilization).



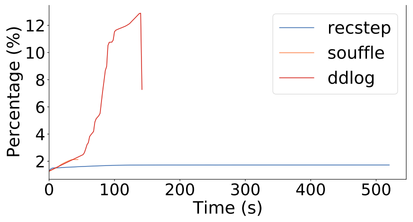
(a) Recursion Profile



(b) Runtime



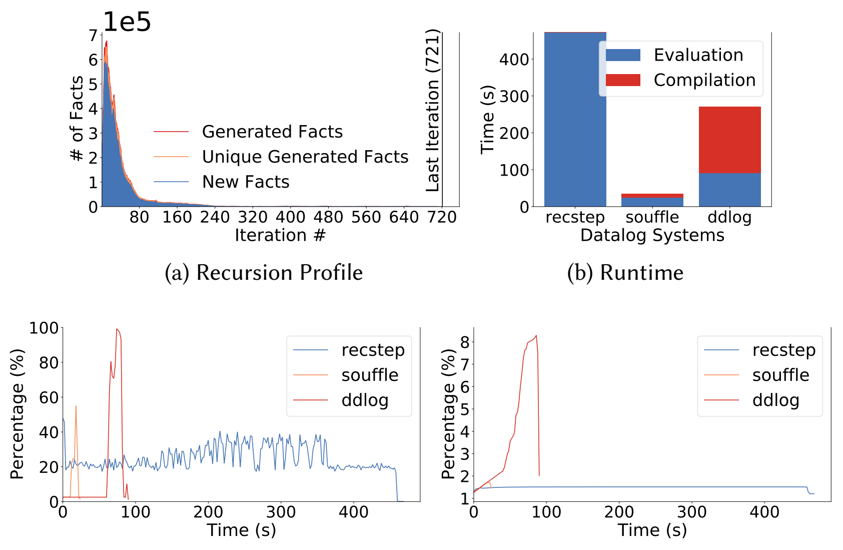
(c) CPU Utilization



(d) Memory Consumption

Figure 6: Context-Sensitive Dataflow Analysis on linux

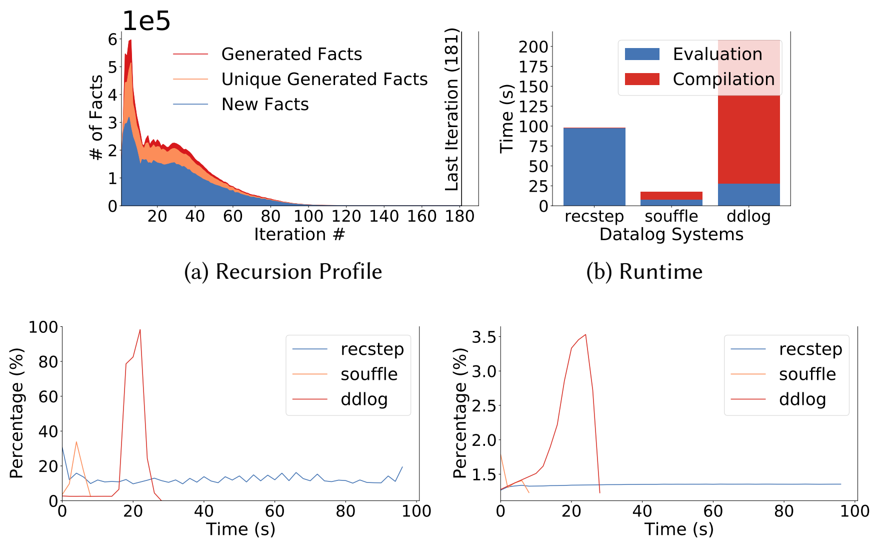
CSDA can be seen as a variant of transitive closure: first a set of *null edges* is given to initialize the non-recursive rule, and then the linear-recursive rule looks the same as that of TC. However, the recursive computation profiles of CSDA and TC workloads are very different. We observe that RecStep performs significantly worse compared to Souffle and DDlog (Figures 6b, 7b, 8b) while its CPU utilization over time is lower compared to Souffle and DDlog (Figures 6c, 7c, 8c). Comparing with TC-G5k and REACH-`orkut`, the CSDA workloads on `linux`, `postgresql` and `httpd` have a *very long tail* in their recursion profiles (Figures 6a, 7a, 8a): it takes a large number of iterations for the evaluation to reach the *fixpoint*, and most of the work is performed during the first few iterations. After further digging, we have confirmed that RecStep's poor performance is mainly due to the lack of continuously maintained indexes throughout the program evaluation that its competitors Souffle and DDlog have. This forces RecStep to reconstruct hash tables and



(a) Recursion Profile (b) Runtime

(c) CPU Utilization (d) Memory Consumption

Figure 7: Context-Sensitive Dataflow Analysis on postgresql



(a) Recursion Profile (b) Runtime

(c) CPU Utilization (d) Memory Consumption

Figure 8: Context-Sensitive Dataflow Analysis on httpd

repeatedly probe the input table (i.e., `arc`) for joins in every iteration. The resulting overhead accumulates across iterations, leading to poor CPU utilization and efficiency when the Datalog workloads have very long-tail recursion profiles. This observation shows the necessity of continuously maintained indexes for consistent efficient recursive Datalog program execution in these cases.

3.2. Non-linear Recursion

*Andersen's Analysis (**AA**):*

$$
\begin{aligned}
\texttt{pointsTo}(\texttt{Y},\texttt{X}) &\mathrel{:-} \texttt{addressOf}(\texttt{Y},\texttt{X}).\\
\texttt{pointsTo}(\texttt{Y},\texttt{X}) &\mathrel{:-} \texttt{assign}(\texttt{Y},\texttt{Z}),\texttt{pointsTo}(\texttt{Z},\texttt{X}).\\
\texttt{pointsTo}(\texttt{Y},\texttt{W}) &\mathrel{:-} \texttt{load}(\texttt{Y},\texttt{X}),\texttt{pointsTo}(\texttt{X},\texttt{Z}),\texttt{pointsTo}(\texttt{Z},\texttt{W}).\\
\texttt{pointsTo}(\texttt{Z},\texttt{W}) &\mathrel{:-} \texttt{store}(\texttt{Y},\texttt{X}),\texttt{pointsTo}(\texttt{Y},\texttt{Z}),\texttt{pointsTo}(\texttt{X},\texttt{W}).
\end{aligned}
$$

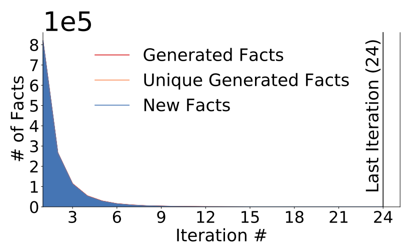
Context-Sensitive Points-To Analysis (**CSPA**):

$$
\begin{aligned}
\texttt{valueFlow}(\texttt{Y},\texttt{X}) &\mathrel{:-} \texttt{assign}(\texttt{Y},\texttt{X}).\\
\texttt{valueFlow}(\texttt{X},\texttt{X}) &\mathrel{:-} \texttt{assign}(\texttt{X},\texttt{Y}).\\
\texttt{valueFlow}(\texttt{X},\texttt{X}) &\mathrel{:-} \texttt{assign}(\texttt{Y},\texttt{X}).\\
\texttt{valueFlow}(\texttt{X},\texttt{Y}) &\mathrel{:-} \texttt{assign}(\texttt{X},\texttt{Z}),\texttt{memoryAlias}(\texttt{Z},\texttt{Y}).\\
\texttt{valueFlow}(\texttt{X},\texttt{Y}) &\mathrel{:-} \texttt{valueFlow}(\texttt{X},\texttt{Z}),\texttt{valueFlow}(\texttt{Z},\texttt{Y}).\\
\texttt{valueAlias}(\texttt{X},\texttt{Y}) &\mathrel{:-} \texttt{valueFlow}(\texttt{Z},\texttt{X}),\texttt{valueFlow}(\texttt{Z},\texttt{Y}).\\
\texttt{valueAlias}(\texttt{X},\texttt{Y}) &\mathrel{:-} \texttt{valueFlow}(\texttt{Z},\texttt{X}),\texttt{memoryAlias}(\texttt{Z},\texttt{W}),\texttt{valueFlow}(\texttt{W},\texttt{Y}).\\
\texttt{memoryAlias}(\texttt{X},\texttt{X}) &\mathrel{:-} \texttt{assign}(\texttt{Y},\texttt{X}).\\
\texttt{memoryAlias}(\texttt{X},\texttt{X}) &\mathrel{:-} \texttt{assign}(\texttt{X},\texttt{Y}).\\
\texttt{memoryAlias}(\texttt{X},\texttt{W}) &\mathrel{:-} \texttt{dereference}(\texttt{Y},\texttt{X}),\texttt{valueAlias}(\texttt{Y},\texttt{Z}),\texttt{dereference}(\texttt{Z},\texttt{W}).
\end{aligned}
$$

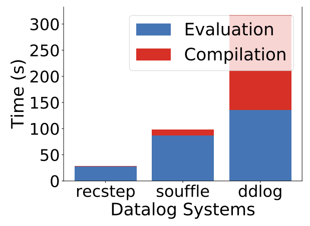
Switching from linear recursion to non-linear recursion, we observe that RecStep significantly outperforms Souffle and DDlog (Figure 9b) for Andersen's Analysis evaluated on the largest input dataset ($\sim$ 1.2G) used in [1]. Figure 9c shows that both Souffle and RecStep have a long *warm-up time* in which the CPU utilization is very low. The reason is similar to that observed in the evaluation of REACH on large graphs - since there are four EDB/input relations in AA, the total size of which is relatively large, more preprocessing work for Souffle and DDlog (e.g., index construction) is needed.

Since the size of *generated facts* is relatively small across different iterations, RecStep evaluates AA efficiently while using only a small amount of memory (Figure 9d). Additionally, AA has *nonlinear-recursive* rules, and all rules in AA derive the facts for the same relation (i.e., `pointsTo`), in which case RecStep is able to fully utilize CPU and evaluate all rules in parallel.

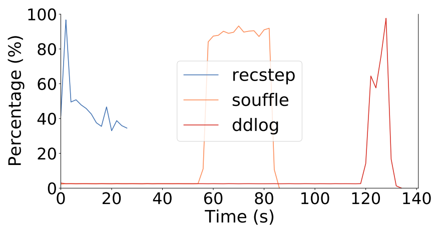
The Datalog program itself alone is insufficient to characterize the recursive computation. For CSPA, the recursive computation profiles of three systems on `linux` dataset look very different from the ones on `postgresql` and `httpd` datasets. Interestingly, we can see that similar recursive profiles (Figure 11a and Figure 12a) come along with similar profiling information on runtime (Figure 11b and Figure 12b), CPU (Figure 11c and Figure 12c) and memory (Figure 11d and Figure 12d). DDlog's performance seems to be very sensitive to the sizes of the intermediate results: when there are large number of facts being generated in several iterations, DDlog



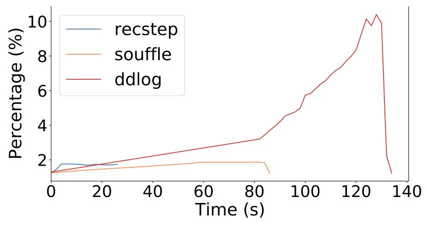
(a) Recursion Profile



(b) Runtime



(c) CPU Utilization



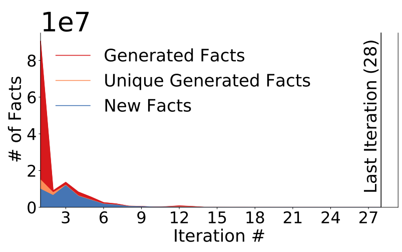
(d) Memory Consumption

Figure 9: Andersen's Analysis

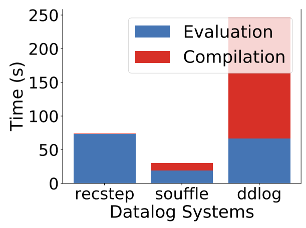
turns out to require a great amount of memory to maintain the intermediate states (Figure 11d and Figure 12d) and the corresponding overhead also affects the overall performance greatly (i.e., DDlog is outperformed by RecStep and Souffle as shown in Figure 11b and Figure 12b). Such inference can be additionally strengthened by looking at Figure 10, in which Figure 10a shows fewer facts are generated during the iterative evaluation and DDlog show better relative performance over RecStep in Figure 10b while using considerably less memory as shown in Figure 10d.

3.3. Discussion

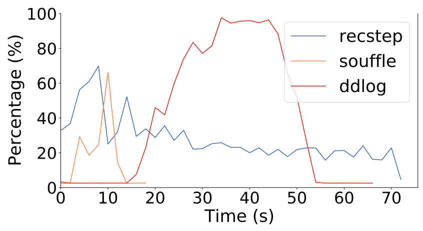
The case studies above show how we can leverage the recursive computation profiling components described in Section 2 to understand better the performance difference between different Datalog systems on a given Datalog workload. This also allows us to identify the limitation of existing systems. For example, DDlog is unsuitable for batch-processing on the workload that generates intermediate results of large sizes but might be of proper use when GF_{size} is small. Souffle performs well primarily because of the heavy optimization of indexes, but indexing could become a bottleneck in some cases (Figure 4b and Figure 9b). The large performance gap between RecStep and the other two Datalog systems observed on CSDA reveals the importance of continuously maintained indexes to good overall performance. Finally, the large memory consumption of RecStep observed on TC-G10k and SG-G5k suggests such indexes should also have good compression capability for memory efficiency.



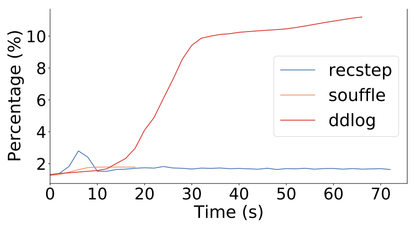
(a) Recursion Profile



(b) Runtime

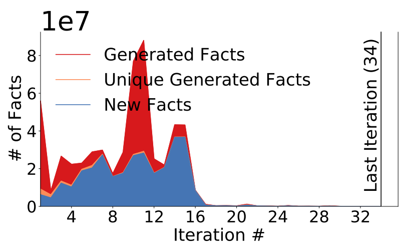


(c) CPU Utilization

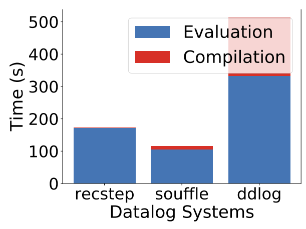


(d) Memory Consumption

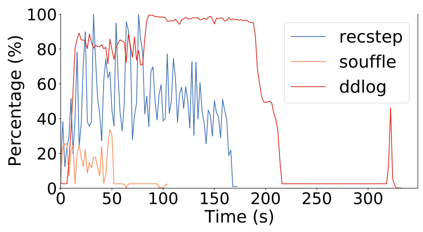
Figure 10: Context-Sensitive Points-to Analysis Linux



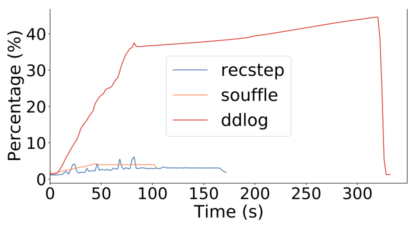
(a) Recursion Profile



(b) Runtime



(c) CPU Utilization

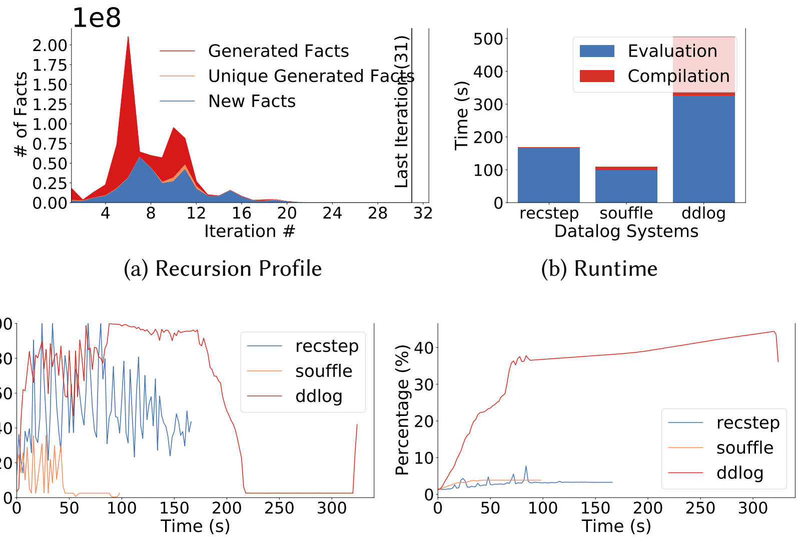


(d) Memory Consumption

Figure 11: Context-Sensitive Points-to Analysis Postgresql

4. Conclusion and Future Work

We have recently observed a renewed interest in Datalog. While recent work has significantly advanced the state-of-the-art of Datalog evaluation techniques, we believe that a systematic way to help gain a better understanding of these techniques is of great importance. Besides the recursive computation profiling we propose in the paper, a standard benchmark covering



(a) Recursion Profile (b) Runtime

(c) CPU Utilization (d) Memory Consumption

Figure 12: Context-Sensitive Points-to Analysis Httpd

different aspects of Datalog workloads is needed. This benchmark should be in analogy to the online analytical processing benchmarks [17] that have been used for more than two decades for performance validation of decision support systems. Without a benchmark suite that covers Datalog workloads with different profiles, one can only gain a partial view of the system performance, which could lead to the lack of essential factors needed during application deployment (for users) and miss of system design decisions (for system builders).

Our ongoing and future work includes improving the recursive computation profiling by adding other possible profiling components, building the new high-performance Datalog system based on the recursive computation profiling, and creating a comprehensive benchmark suite. Once such a benchmark is available and we have a better understanding of the characteristics of different recursive profiles, we wish to find ways to estimate the recursive profile of a Datalog workload without actually running it, which we believe will be helpful to think of evaluation strategies and techniques that provide consistent, efficient execution across Datalog workloads of different characteristics.

References

[1] Z. Fan, J. Zhu, Z. Zhang, A. Albarghouthi, P. Koutris, J. Patel, Scaling-up in-memory datalog processing: Observations and techniques, arXiv preprint arXiv:1812.03975 (2018).
[2] L. Ryzhyk, M. Budiu, Differential datalog., Datalog 2 (2019) 4–5.
[3] B. Scholz, H. Jordan, P. Subotić, T. Westmann, On fast large-scale program analysis in datalog, in: Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Association for Computing Machinery, New York, NY, USA, 2016, p. 196–206. URL: https://doi.org/10.1145/2892208.2892226. doi:10.1145/2892208.2892226.

[4] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, C. Zaniolo, Big data analytics with datalog queries on spark, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 1135–1149. URL: https://doi.org/10.1145/2882903.2915229. doi:10.1145/2882903.2915229.

[5] Q. Zhang, A. Acharya, H. Chen, S. Arora, A. Chen, V. Liu, B. T. Loo, Optimizing declarative graph queries at large scale, in: Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 1411–1428.

[6] F. McSherry, D. G. Murray, R. Isaacs, M. Isard, Differential dataflow., in: CIDR, 2013.

[7] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, S. Saurabh, Quickstep: A data platform based on the scaling-up approach, Proceedings of the VLDB Endowment 11 (2018) 663–676.

[8] J. Whaley, D. Avots, M. Carbin, M. S. Lam, Using datalog with binary decision diagrams for program analysis, in: Asian Symposium on Programming Languages and Systems, Springer, 2005, pp. 97–118.

[9] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Clingo= asp+ control: Preliminary report, arXiv preprint arXiv:1405.3694 (2014).

[10] M. Alviano, W. Faber, N. Leone, S. Perri, G. Pfeifer, G. Terracina, The disjunctive datalog system dlv, in: International Datalog 2.0 Workshop, Springer, 2010, pp. 282–301.

[11] M. Yang, A. Shkapsky, C. Zaniolo, Scaling up the performance of more powerful datalog systems on multicore machines, The VLDB Journal 26 (2017) 229–248.

[12] S. Abiteboul, R. Hull, V. Vianu, Foundations of databases, volume 8, Addison-Wesley Reading, 1995.

[13] CloudLab, https://www.cloudlab.us/, 2018.

[14] D. A. Bader, K. Madduri, Gtgraph: A synthetic graph generator suite, Atlanta, GA, February 38 (2006).

[15] F. McSherry, A. Lattuada, M. Schwarzkopf, T. Roscoe, Shared arrangements: practical inter-query sharing for streaming dataflows, arXiv preprint arXiv:1812.02639 (2018).

[16] H. Jordan, P. Subotić, D. Zhao, B. Scholz, Brie: A specialized trie for concurrent datalog, in: Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'19, Association for Computing Machinery, New York, NY, USA, 2019, p. 31–40. URL: https://doi.org/10.1145/3303084.3309490. doi:10.1145/3303084.3309490.

[17] M. Barata, J. Bernardino, P. Furtado, An overview of decision support benchmarks: Tpc-ds, tpc-h and ssb, New Contributions in Information Systems and Technologies (2015) 619–628.