BUILDING DATALOG SYSTEMS FOR EFFICIENT AND SCALABLE DATA ANALYTICS

by

Zhiwei Fan

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2022

Date of final oral examination: 18 August, 2022

ACKNOWLEDGMENTS

I am forever grateful to my advisor, Paris Koutris. The dissertation would not have been possible without his kindness, patience, and endless support in my research, life, and career. Despite Paris's own interests in theory, he has unconditionally supported me to do system research and offered me full freedom to explore my own ideas that were outside of his core interests. Paris's incredible wisdom and empathy have helped me countless times in tough situations that I would not have been able to walk out of and keep my passion for research if it were not for his wise and kind words, which have become my invaluable life lessons that I believe will continue to guide my road of future. Mostly attributed to Paris' kindness and his open-mindedness, I am very fortunate to be able to feel the joy of research and continue to be passionate about doing research work. No words can fully express my infinite gratitude to Paris.

I am deeply grateful to Jignesh Patel and Aws Albarghouthi for their collaborations and serving on my committee. Jignesh's remarkable grasp on grounding research of database management that always connects to practical relevance has inspired and educated me in many ways, and during the limited interactions with him out of his busy and packed schedule, I have greatly benefited from his tough, critical, but really insightful questions on my work, which has helped improve and sharpen my way of problem thinking in research. Jignesh has also offered numerous valuable comments and suggestions to help improve this dissertation. Without taking a single class in programming languages, I have almost gained all the relevant knowledge needed in my research from Aws through many of the educational conversations. I really thank Aws for opening my eyes to the charm of programming language, including the very important area, program analysis. I cannot appreciate him more for his kindness and understanding during those down moments; he has always been open for conversations, the help to my research and psychological support from which cannot be expressed or measured in words. I thank Richard for serving on my committee and for his insightful feedback that helps to connect my work to applications in education research.

I am also deeply grateful to Jeff Naughton, who was a Professor of Computer Sciences at the University of Wisconsin–Madison when I was an undergraduate there, and Arun Kumar, who was Jeff's student at the time and is now a Professor at the University of California, San Diego. Taking Jeff's courses on database management and learning to do research under Jeff and Arun have literally changed my life, leading me to start my Ph.D. journey. Jeff's fun teaching style and those research anecdotes being told during his lectures have sparked my interest in database research. His kindness and wisdom felt during our conversations have helped clear up many of my confusions during that period of time. Arun has always been kind, patient and understanding when guiding me on the research work. He has provided tremendous help during my graduate school application process, including giving valuable suggestions and helping revise my personal statement. I will forever remember and be grateful for the help and positive influence that Jeff and Arun have given me.

I thank Jerry Zhu, Jinyi Cai, Anhai Doan, Remzi Arpaci-Dusseau, Shuchi Chawla, Aditya Akella, and Theo Rekatsinas, the classes of whom I have taken that have benefited my research and this dissertation in various ways. Over the years, I have taken several internships in the industry and have had the opportunity to meet and work with many talented and brilliant people during this journey. I would like to thank Nikolaos Vasiloglou, my mentor of my first internship, for giving me the opportunity to work on an interesting research problem related to the interaction between database management and machine learning. I am very lucky to get to know my colleague Mikhail Yurochkin during the internship and become friends with him. Mikhail is definitely one of the smartest, most hard-working, nicest people I know. During many of our interactions and collaborations, I have learned much statistical machine learning knowledge from Mikhail. His brilliant way of communicating has made me start to appreciate math more than ever. What we discussed outside of work has also inspired me many times, reconstructing my perspectives on many things in life. I was very lucky to work with Wentao Wu and Phil Bernstein during my internship in the DMX Group at Microsoft Research Redmond. The streaming project on which we have worked has given me inspiration for the work in this dissertation. During the internship, Wentao and Phil have offered numerous help during the project idea discussion and have given life advice. I am very grateful to get to know people who have not only provided help in research and technical matters, but also taught me life lessons, leaving me with long-term benefits. There are many other people with whom I have interacted during internships that play important roles in this journey. The incomplete list includes Nantia Makrynioti, Anshuman Dutt, Yeye He, Anrd Christian König, Yao Lu, Lukas Mass, Yue Wang, Chi Wang, Qizhen Zhang, Tarique Siddiqui, Walter Cai, Christina Pavlopoulou and Vasilis Zois. I apologize that I could not make a complete list here, which would otherwise make this paragraph unbelievably long.

This journey would have been much less enjoyable without my interactions with other fellow students in the computer sciences department. Throughout the years, I have made many friends during the journey. An incomplete list includes Hongyi Wang, Yuzhe Ma, Xuzhou Zhang, Xin Jin, Yunfeng Li, Xiahe Liu, Kan Wu, Han Li, Jianqiao Zhu, Haojun Zhang, Zuyu Zhang, Xiating Ouyang, Shaleen Deep, Hangdong Zhao, Kevin Gaffney. They have not only enriched my Ph.D. life, but have also educated me on different research topics such as machine learning, computational complexity, and deep system understanding.

Finally, I am beyond measure indebted to my parents for their unconditional and endless love and support in my life. My father Zhongxiang Fan and my mother Cuiyun Zheng have always tried everything they can to give me the best education and living conditions, and I would not have become who I am today without their unconditional giving and sacrifice. I therefore dedicate this dissertation to them. Last but definitely not least, I am grateful to my wife Xinyu for her love and support during the crucial final stages of my dissertation and my job arrangement. We have gone through many difficult times since we got married, and I believe that these experiences have only made us better understand each other and I look forward to the next chapter of my life and career with her.

TABLE OF CONTENTS

ABSTRACT

The ability to perform advanced data analytics efficiently is becoming increasingly important for a wide spectrum of data-driven applications in the big data era. The efficiency of data analysis is generally considered from two aspects: (i) the ability to quickly prototype and express the corresponding analysis tasks (here referred to as *development efficiency*) and (ii) the ability to process a large volume of data involved in the analysis with high performance and good scalability (here referred to as *computational efficiency*). Datalog as a declarative programming language is seeing a resurgence of interest in recent years and has found new applications in multiple domains such as data integration, graph analytics, security, program analysis, networking, and decision-making, largely attributed to its development efficiency. To seek better support for computational efficiency in using Datalog as the language for a wide variety of data-driven tasks, especially taking advantage of its superior ability to express applications involving recursive computations concisely, several research efforts, across multiple communities, have explored techniques to build efficient Datalog systems. However, our experience with the corresponding resulting systems indicates that their performance does 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). Furthermore, the lack of understanding of the property of varying Datalog workloads makes it challenging to analyze the performance difference observed on different systems, further impedes the progress in improving existing systems and building more efficient new systems.

In this dissertation, we explore techniques for building a general-purpose Datalog system for scalable and efficient data analytics. The exploration has led to two prototype Datalog systems, *RecStep*, and *FlowLog*, which are implemented on top of a parallel single-node relational system and a modern stream processor, respectively.

We first show that by leveraging multiple years of efforts in the advancement of database techniques such as query optimization and efficient parallel query execution, *RecStep* is able to outperform a few state-of-the-art specialized Datalog engines on complex and large-scale Datalog evaluation. Next, we present the important profiling components of a general-purpose recursive computation profiling framework, which provide insights regarding the performance behavior of different systems on varying workloads, guiding our design and implementation of *FlowLog*. Then, we present the prototype system *FlowLog* in detail, discussing the philosophy behind its design and its implementation, and showing the high performance it delivers. Finally, we show how we can leverage the development efficiency provided by Datalog to concisely express better algorithms for a specific application called *consistent query answering* (CQA) and how *FlowLog* efficiently evaluates the corresponding Datalog programs, often matching and sometimes surpassing the state-of-the-art performance numbers while other existing Datalog systems cannot achieve this.

Chapter 1

Introduction

To support data analytics over ever-growing volume of data is becoming increasingly important in today's data-driven world. As a result, the past years have witnessed numerous efforts to invent new systems and improve existing systems to fit the needs seen in the big data era. Examples of such systems include Microsoft SQL Server [Mic19], Hive [TSJ+09], Spark [ZXW+16], Presto [STS+19], Flink [CKE+15] and Kafka [H+18]. To achieve both development (i.e., implementation of analysis tasks) and computational (i.e., analyzing and generating the results) efficiencies, most of these systems provide SQL, the standardized declarative query language for structured data processing, for easy portability while achieving scalability via parallel computation across multiple CPU cores/computation nodes.

However, expressing complex data analysis tasks that typically involve recursion with SQL has been historically proven difficult and discussed in [KKN03, SYI+16, Yan17]. Example attempts to extend SQL to support advanced analytics tasks include user-defined functions (UDF) [Mic22a] and recursive common table expressions [Mic22b], which present limited flexibility and come at the cost of poorer performance and readability. Datalog as a declarative language that is arguably more expressive than SQL, has experienced a recent resurgence, as a result of finding its role in many modern application domains, and can be seen as a promising alternative for advanced data analytics that can possibly provide both succinct expressibility and high performance.

The motivating question of this thesis is the following: *How can we design and build a general-purpose Datalog system for efficient and scalable advanced data analytics?* In this dissertation, we first show that it is possible to effectively use a modern parallel relational database management system (RDBMS) as a backend for Datalog evaluation by carefully considering the underlying issues of the system. We then point out the limitations of this approach and discuss how such limitations are understood more fully by looking at the *recursive computation profiling* components, which characterize the recursive Datalog workloads, the runtime performance, and the usage of computation resources in Datalog systems. Based on our observations and insights gained from the recursive computation profiling, we show

how we can build a general-purpose asynchronous high-performance Datalog system, named *FlowLog*. Examing the non-recursive Datalog rules that express the first-order rewriting of consistent query answering (CQA) for a given self-join-free first-order rewritable conjunctive query, we observe that the evaluation of *non-recursive* Datalog program with *negation* can largely benefit from *asynchronous execution*, which has been surprisingly dismissed in recent works focusing on Datalog evaluation techniques.

1.1 Motivation

To motivate the use of Datalog, we start with one of the simplest yet most common and important computation task that involves recursion *transitive closure* (TC).

Example 1. *To compute the transitive closure (TC) of a directed graph, we represent the directed graph using a binary relation* `edge(X, Y)`*: this means that there is a directed edge from vertex* `X` *to vertex* `Y`*. TC can be expressed through the following Datalog program consisting of two rules:*

$$\textit{Rule 1}: \texttt{reachable}(\texttt{X},\texttt{Y}) \text{ :- } \texttt{edge}(\texttt{X},\texttt{Y}).$$

$$\textit{Rule 2}: \texttt{reachable}(\texttt{X},\texttt{Y}) \text{ :- } \texttt{reachable}(\texttt{X},\texttt{Z}), \texttt{edge}(\texttt{Z},\texttt{Y}).$$

In the above program, `edge` *is an* EDB *relation (input), and* `reachable` *is an* IDB *relation (output). The program can be interpreted as follows.* ***Rule 1*** *(called the* base rule*) initializes the transitive closure by adding to the initially empty relation all the edges of the graph.* ***Rule 2*** *is a* recursive rule*, and produces new facts iteratively: a new fact* `reachable(a,b)` *is added whenever there exists some constant* `c` *such that* `reachable(a,c)` *is already in the relation (from the previous iterations), and* `edge(c,b)` *is an edge in the graph.*

Compared to the two simple and succinct Datalog rules expressing TC in Example 1, the C++ code snippet that implements TC as shown in Figure 1.1a looks much more complicated even after omitting the detailed logic of reading the input data (i.e., `readEdge()`). The C++ implementation requires making choices about the proper data structures to use and handling the actual computation logic, which could be both time-consuming and error-prone. In addition, it is also much harder to understand the C++ implementation since the higher complexity leads to poorer readability. Expressing TC as a SQL statement with recursion support makes it relatively easier compared to the C++ implementation, since SQL is declarative. However, as shown in Figure 1.1b, the logic of the SQL statement is still less straightforward to understand compared to the Datalog rules, and the SQL query will soon become harder to write and less readable when considering more complex tasks. Furthermore, SQL with recursion provides only simple linear-recursion support, suggesting

```cpp
using Tuple = std::array<int, 2>;
using Relation = std::set<Tuple>;

Relation edge, tc;
// Populate facts into edge relation
edge = readEdges();
// reachable(X, Y) :- edge(X, Y).
reachable = edge;
// reachable(X, Y) :- reachable(X, Z), edge(Z, Y)
auto delta = reachable;
while(!delta.empty()) {
    // Compute the delta produced in the current iteration
    Relation newDelta;
    // Compute the new facts derived in reachable
    for (auto t1 : delta) {
        for (auto ) {
            Tuple t = {t1[0], t2[1]};
            if (!reachable.contains(t)) {
                newDelta.insert(t);
            }
        }
    }
    // Update the reachable relation
    reachable.insert(newDelta.begin(), newDelta.end());
    // Update the delta
    delta.swap(newDelta);
}
```

(a) C++ code snippet

```sql
WITH RECURSIVE
iterations(x) AS (
    SELECT COUNT(*)
    FROM edge
),
reachable(src, dest) AS (
    SELECT src, dest FROM edge
    UNION ALL
    SELECT reachable.src, edge.dest FROM reachable
    JOIN edge on reachable.dest = edge.src
    LIMIT (SELECT (SELECT * FROM iterations) *
                  (SELECT * FROM iterations)
          )
)
SELECT DISTINCT src, dest
FROM reachable
```

(b) SQL statement with recursion

Figure 1.1: Transitive Closure Implementations in C++ and SQL

that applications involving more complex recursion logic cannot be expressed by SQL. In contrast, the declarative abstraction Datalog provides lets users focus on a wide variety of tasks (*what*) instead of the low-level details (*how*).

Despite the superior ability of Datalog and its language extension to express applications that involve recursive computations succinctly, a general purpose, high-performance and scalable Datalog system is lacking. The recent revival of recursive query processing has motivated a line of work to build efficient Datalog systems. Our experience with these systems indicates that their performance does not translate across different application domains (Chapter 3) or even different Datalog workloads in the same domain (Chapter 4) - e.g., a system designed for large-scale graph analytics does not exhibit the same performance on program analysis tasks and vice versa. Most current existing Datalog systems rely heavily on synchronous batch processing that introduces synchronization overhead between different computation steps (e.g., different rules and iterations) in recursive computation, and such overhead could translate into poor performance for the Datalog workload consisting of many computation stages, each of which involves only a small amount of work.

1.2 Contribution

In this dissertation, we present a series of explorations and studies, covering novel Datalog evaluation techniques, applications, system design and implementations. In summary, we make the following contributions.

RecStep In this project, we perform an extensive comparison of four state-of-the-art Datalog and Datalog-like systems on a multi-core machine. We consider benchmarks from two application domains: graph analytics and program analysis using both synthetic and real-world datasets. Based on our observations, we study the challenges of building a recursive query processing engine on top of a RDBMS and consider a spectrum of techniques that resolve them, systematically measuring the effect of each technique on performance improvement. The project has led to a Datalog system prototype implementation, namely RecStep, which is built on top of Quickstep [PDZ+18], a single-node in-memory parallel RDBMS. Our experimental results show that RecStep can efficiently perform large-scale tasks in different application domains using a single-node multi-core machine with large memory, demonstrating that it is feasible to build a fast general-purpose Datalog engine using RDBMS as the backend, which interestingly contradicts empirical and anecdotal evidence [SVKW15, JSS16].

Profiling To further understand the performance difference observed in different systems and the limitations presented in RecStep when performing experiments on varying Datalog workloads, we propose *general recursive computation profiling*. We show that by leveraging the visualizations generated from the profiling, we are able to further analyze the causes behind the inefficient executions observed on varying systems, including costly index construction on certain input datasets, accumulative overhead caused by repeated work without maintained index reuse, and ineffective CPU utilization caused by synchronous computation steps when each computation step involves only a small amount of work. The insights gained provide guidance on how to design more efficient systems.

FlowLog Based on our experience building RecStep, evaluating different systems on varying Datalog workloads, and our observations on the corresponding recursive computation profiles, we rethink the evaluation of Datalog program, designing and implementing a high-performance asynchronous Datalog system FlowLog on top of a modern stream processor *Differential Dataflow* [MMII13, AMP15, Mcs22a]. Combining with techniques such as index sharing and incremental computation, we show that FlowLog significantly outperforms existing Datalog systems on varying workloads.

LinCQA Most data analytical pipelines often face the problem of querying inconsistent data that violate predetermined integrity constraints. Data cleaning is an extensively studied paradigm that singles out a consistent repair of the inconsistent data. Consistent query answering (CQA) is an alternative approach to data cleaning that asks for all tuples guaranteed to be returned by a given query on all (in most cases, exponentially many) repairs of the inconsistent data. In this project, we identify a class of acyclic select-project-join (SPJ) queries for which CQA can be solved via algorithms with a linear-time guarantee.

Our approach can be seen as a generalization of Yannakakis's algorithm for acyclic joins to the inconsistent setting. We present LinCQA, a system that can output rewritings in both SQL and non-recursive Datalog rules for every query in this class. We show that LinCQA often outperforms existing CQA systems on both synthetic and real-world workloads, and in some cases by orders of magnitude. We further show that by executing the Datalog rewriting that consists of non-recursive rules with negation in an efficient asynchronous Datalog system, we are able to surpass the state-of-the-art performance numbers that other existing Datalog engines are unable to. The evaluation of non-recursive Datalog program has been often dismissed in recent works targeting efficient Datalog computation, due to their heavy focus on the recursive query processing.

1.3 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we first provide the background for the Datalog language, its extensions, and the algorithms used for its evaluation, along with the related notations and terminology; we then briefly introduce the systems studied and the frameworks used to build the two prototype Datalog systems.

Next, we present the design, implementation and evaluation of *RecStep* in Chapter 3. In Chapter 4, we show how simple visualizations can be used to better characterize recursive computation tasks and help understand the performance difference observed in different systems. In Chapter 5, we introduce *FlowLog*, a high-performance asynchronous Datalog prototype system that aims to be easy to install and use, maintainable, and extensible on different platforms.

We look at an important problem called *consistent query answering* (CQA) that queries inconsistent data in Chapter 6 and show how we use Datalog to help analyze the problem and come up with an efficient solution. Finally, we conclude this dissertation in Chapter 7.

Chapter 2

Background

In this chapter, we present the necessary background and notation for the reader to follow this dissertation. We provide background on the syntax and evaluation strategies of the Datalog language and its extensions that we consider in this dissertation. We then discuss the existing Datalog systems, some of which are studied in this dissertation.

2.1 Datalog

Datalog Basics

A Datalog program P is a finite set of rules. A *rule* is an expression of the form:

$$h \text{ :- } p_1, p_2, \ldots, p_k.$$

The expressions $h, p_1, \ldots, p_k$ are *atoms*, i.e., formulas of the form $R(t_1, \ldots, t_\ell)$, where R is a table/relation name (predicate) and each t_i is a *term* that can be a constant or a variable. An atom is a *fact* (or *tuple*) when all t_i are constants. The atom h is called the *head* of the rule, and the atoms $p_1, \ldots, p_k$ are called the *body* of the rule. A rule can be interpreted as a logical implication: if the predicates $p_1, \ldots, p_k$ are true, then so is the head h. We assume that rules are always safe: this means that all variables in the head occur in at least one atom in the body. We will use the upper case $X, Y, Z, \ldots$ to denote variables and lower case $a, b, c, \ldots$ for constants.

The relations in a Datalog program are of two types: IDB and EDB relations. A relation that represents a base table (input) is called EDB (extensional database); EDB relations can occur only in the body of a rule. A relation that represents a derived relation is called IDB (intentional database) and must appear in the head of at least one rule. The *transitive closure* presented in Example 1 illustrates these concepts concretely.

Stratification. Given a Datalog program P, we construct its *dependency graph* G_P as follows: every rule is a vertex and a directional edge $r \to r'$ exists in G_P whenever the head of the rule r appears in the body of rule r', meaning r' *depends on* r. A rule is *recursive* if it belongs in a directed cycle, otherwise it is called *non-recursive.* A Datalog program is recursive if it contains at least one recursive rule. For instance, the dependency graph of the program in Example 1 contains two vertices r_1 (Rule 1) and r_2 (Rule 2), and two directed edges: $r_1 \rightarrow r_2$ and $r_2 \rightarrow r_2$. Since the graph has a self-loop, the program is recursive. A *stratification* of P is a partition of rules into strata, where each stratum contains the rules that are in the same strongly connected component of G_P. The topological ordering of the strongly connected components also defines an ordering in the strata. In example 1, there exist two strata, $\{r_1\}, \{r_2\}$.

Datalog Evaluation

Datalog is a declarative query language, and hence there are different algorithms that can be applied to evaluate a Datalog program. Most implementations of Datalog use *bottom-up* evaluation techniques, which start from the input (EDB) tables, and then iteratively apply the rules until no more new tuples (i.e., facts) can be added to the IDB relations, reaching a *fixpoint.* At each iteration of the *naive evaluation* strategy, the rules are applied using all the facts produced so far. For our running example, we would initialize the IDB relation `reachable` with $\texttt{reachable}^0 \leftarrow \texttt{edge}$. To calculate the $(i+1)$ iteration, we compute $\texttt{reachable}^{i+1} \leftarrow \pi_{X,Y}(\texttt{reachable}^i \bowtie \texttt{edge}) \cup \texttt{reachable}^i$. The evaluation ends when $\texttt{reachable}^{i+1} = \texttt{reachable}^i$. The naive evaluation strategy has the drawback that the same facts might be produced multiple times throughout the evaluation, which could lead to redundant computation.

In the *semi-naïve evaluation* strategy, at every iteration the algorithm uses only the *new* facts from the previous iteration to generate tuples in the current iteration. For instance, in the motivating example, at every iteration i, we maintain together with $\texttt{reachable}^i$ the facts that are generated only in the i^{th} iteration (and not in previous iterations), denoted by $\Delta\texttt{reachable}^i = \texttt{reachable}^i - \texttt{reachable}^{i-1}$. Then, we compute `reachable` of the $(i+1)^{th}$ iteration as $\texttt{reachable}^{i+1} \leftarrow \pi_{X,Y}(\Delta\texttt{reachable}^i \bowtie \texttt{arc}) \cup \texttt{reachable}^i$. The running example is an instance of *linear recursion*, where each recursive rule contains at most one atom with an IDB. However, many Datalog programs, especially in the context of program analysis, contain non-linear recursion, where the body of a rule contains multiple IDB atoms. In this case, the Δ relations are computed by taking the union of multiple subqueries (for more details see [AHV95]).

Semi-naïve evaluation can be further sped up by exploiting the stratification of a Datalog program: the strata are ordered from lower to higher according to the topological order of the rule dependency graph G_P, and then each stratum is evaluated sequentially, considering the IDB relations of the prior strata as EDB relations (input tables) in the current stratum.

In our implementation of Datalog, we use the semi-naïve evaluation strategy in combination with stratification.

Negation and Aggregation

In order to enhance the expressiveness of Datalog for use in modern applications, we consider two syntactic extensions: *negation* and *aggregation*.

Negation. Datalog is a monotone language, which means that it cannot express tasks where the output can become smaller as the input increases. However, many tasks are inherently non-monotone. To express these tasks, we extend Datalog with a simple form of negation, called *stratified negation*. In this extension, negation is expressed by adding the symbol $\neg$ in front of an atom. However, the use of $\neg$ is restricted syntactically, such that an atom $R(t_1, \dots, t_\ell)$ can be negated in a rule if (*i*) R is an EDB, or (*ii*) any rule where R occurs in the head is in a strictly lower stratum.

Example 2. *Suppose we want to compute the complement of transitive closure, in other words, the vertex pairs that do not belong in the closure. This task can be expressed by the following Datalog program with stratified negation:*

$$\begin{aligned}
\texttt{reachable(X,Y)} &\ \text{:-}\ \texttt{edge(X,Y)}.\\
\texttt{reachable(X,Y)} &\ \text{:-}\ \texttt{reachable(X,Z)},\texttt{edge(Z,Y)}.\\
\texttt{node(X)} &\ \text{:-}\ \texttt{edge(X,Y)}.\\
\texttt{node(Y)} &\ \text{:-}\ \texttt{edge(X,Y)}.\\
\texttt{unreachable(X,Y)} &\ \text{:-}\ \texttt{node(X)},\texttt{node(Y)},\neg\texttt{reachable(X,Y)}.
\end{aligned}$$

Aggregation. We further extend Datalog with aggregation operators. To support aggregation, we allow the terms in the head of the rule to be of the form $\texttt{AGG}(X, Y, \dots)$, where $X, Y, \dots$ are variables in the body of the rule, and `AGG` is an aggregation operator that can be `MIN`, `MAX`, `SUM`, `COUNT`, or `AVG`. We allow aggregation not only in non-recursive rules, but inside recursion as well, as studied in [Lef92]. In the latter case, one must be careful that the semantics of the Datalog program lead to convergence to a fixpoint; in this paper, we assume that the program given as input always converges ([ZYI$^+$18] studies how to test this property). As an example of the use of aggregation, suppose that we want to compute for each vertex the number of vertices that are reachable from this vertex. To achieve this, we can simply add to the transitive closure Datalog program of Example 1 the following rule:

$$r_3 : \texttt{count\_reachable(X,COUNT(Y))}\ \text{:-}\ \texttt{reachable(X,Y)}.$$

We should note here that it is straightforward to incorporate negation and aggregation in the standard semi-naïve evaluation strategy. Since negation can only be applied when there is no recursion, it can easily be encoded in a SQL query using the difference operator. Aggregation can be similarly encoded as group-by plus aggregation.

2.2 Existing Datalog Systems

Over the years, many works have been published that focus on efficient evaluation Datalog. Here, we briefly discuss the systems and related techniques that emerged as the results of these works. We note here that the list is incomplete, and we mainly focus on the recent works that have close relevance to this dissertation.

Distributed Datalog Engines. Over the past few years, there have been several efforts to develop scalable evaluation engines for Datalog. Seo et al. [SPSL13] presented a distributed engine for a Datalog variant for social network analysis called Socialite. Socialite employs a number of techniques to enable distribution and scalability, including delta stepping, approximation of results, and data sharding. The notable limitation is Socialite's reliance on user-provided annotations to determine how to decompose data on different machines. Wang et al. [WBH15] implement a variant of Datalog in the Myria system [HdAC+14], focusing mainly on *asynchronous evaluation* and *fault tolerance*. The BigDatalog system [SYI+16] is a distributed Datalog engine built on a modified version of Apache Spark. A key enabler of BigDatalog is a modified version of RDDs in Apache Spark, enabling fast set-semantic implementation. The BigDatalog work has shown superior results to the previously proposed systems that we discussed above, Myria and Socialite. Therefore, in our study, we focus on BigDatalog for comparison with distributed implementations. Cog [IGM20] and Nexus [IGQRM22] are philosophically similar to FlowLog as they exploit the cyclic dataflow model supported by Flink [CKE+15], which provides the capability of incremental computation and asynchronous iteration. Nexus is an extension to Cog with the added support of recursive aggregation and use in streaming scenarios. However, it is not clear whether Nexus allows index sharing between different Datalog rules as the underlying system Flink does not support indexed state sharing, and Nexus lacks support for mutual recursion. The task of parallelizing Datalog has also been studied in the context of the popular MapReduce framework [ABC+11, AU12, SKHS12]. Motik et al. [MNP+14] provide an implementation of parallel Datalog in multicore main memory systems.

Datalog Solvers in Program Analysis. Static program analysis traditionally is the problem of overapproximating runtime program behaviors. Since the problem is generally undecidable, program analyses resort to overapproximations of runtime facts of a program. A large and powerful class of program analyses, formulated as *context-free language reachability*, has been shown to be equivalent to Datalog evaluation. Thus, multiple Datalog engines have been built and optimized specifically for program analysis tasks. The BDDBDDB Datalog solver [WACL05] pioneered the use of Datalog in program analysis by employing binary decision diagrams (BDD) to compactly represent the results of program analysis. The idea is that there is a lot of redundancy in the results of a program analysis, due to overapproximation, which BDDS help to obtain exponential savings. However, BDDBDDB does not support parallel computation and operations, such as aggregation. Furthermore, BDD is very sensitive to the ordering of variables used in binary encoding, and figuring the optimal order of is NP-complete. Several Datalog solvers have recently been used for program analysis that employ tabular representations of data. These include the Souffle solver [SJSW16] and the LogicBlox solver [GAK12]. Souffle (which has been shown to outperform LogicBlox [ATS17]) compiles Datalog programs into native parallel C++ code, which is then compiled and optimized for a specific platform. Datalog solver mainly built for static program analysis, exploiting techniques such as automatic index selection and indexing structures that allow efficient parallel operations. Differential Datalog [RB19] is an in-memory Datalog engine that is built for incremental computation, under the hood of which is differential dataflow [MMII13]. All of these solvers do not employ a deep form of parallelism, which our work exhibits by utilizing a parallel in-memory database and a computation framework. The Graspan engine [WHZ$^+$17] takes a context-free grammar representation and is thus restricted to binary relations—graphs. Graspan employs a worklist-based algorithm to parallelize the computation of the fix point on a multicore machine. However, as we show experimentally, our systems RecStep and FlowLog as well as other studied Datalog systems, can significantly outperform Graspan on its own benchmark set.

Other Graph Engines and Graph Query Languages. By now, there are numerous distributed graph processing systems, such as Pregel [MAB$^+$10] and Giraph [HD15]. These systems espouse the *think-like-a-vertex* programming model, where one writes operations per graph vertex. These are restricted to binary relations (graphs); Datalog, by definition, is more general in that it captures computation over hypergraphs. The native graph database such as Neo4j [Mil13], using Cypher [FGG$^+$18] as its declarative query language for property graphs, in which each vertex and edge of the graph consists of a list of properties such as a unique identifier, a collection of key-value pairs. For graph queries, different thinking ways are required when using Datalog compared to other graph-native query languages such as Cypher, mainly due to differences in the underlying data models.

Chapter 3

RecStep: Datalog Evaluation by RDBMS

In this chapter, we introduce our own designed and implemented general-purpose Datalog engine *RecStep*, which is built on top of a parallel single-node relational system. We discuss in detail the techniques applied to RecStep, as well as the contribution of each technique to the overall performance. Using RecStep as a baseline, we demonstrate that it generally outperforms state-of-the-art parallel Datalog engines on complex and large-scale Datalog evaluation, by a 4-6X margin. Our work on building and evaluating RecStep additionally suggests that it is possible to build a high-performance Datalog system on top of a relational engine, an idea that has been dismissed in previous work.

Datalog language and its syntactic extensions have experienced a recent resurgence, as a result of the needs for recursive query processing found in many modern application domains, including but not limited to data integration [FKMP03, Len02], graph analytics [SGL13, SPSL13], program analysis [WL04] [SJSW16] [RB19], networking [LCG+06]. Due to the large volumes of data being processed, several research efforts across multiple communities have explored how to scale up recursive queries in Datalog or Datalog-like language. The development of Datalog solvers (or engines) has been a subject of study in both the database community and the programming language community. The database community independently developed its own tools to evaluate general Datalog programs, both in centralized and distributed settings. These include the LogicBlox solver [GAK12], as well as distributed and cloud-based engines such as BigDatalog [SYI+16], Myria [WBH15], and Socialite [SPSL13]. In the programming language (PL) community, it has been observed that a rich class of fundamental static program analyses can be written equivalently as Datalog programs [Rep97, WL04]. The PL community has extensively implemented solvers that target all (or a subset) of Datalog. This line of research has resulted in several Datalog-based tools for program analysis, including BDDBDDB [WACL05], Souffle [SJSW16], and more recently Graspan [WHZ+17].

Our experience with the corresponding tools produced indicates that their performance does not translate across domains—e.g., a system designed for large-scale graph analytics does not exhibit the same performance on program-analysis tasks, and vice versa. Starting from the above observation, we tested a number of state-of-the-art Datalog systems developed for a wide spectrum of graph analytics and program-analysis tasks, summarizing the pros and cons of existing techniques. Tools such as LogicBlox and BDDBDDB were unable to scale well with large input datasets prevalent in other domains. Even Souffle, the best-performing tool for program analysis tasks, is not well-suited for tasks outside program analysis, such as graph analytics (which also require the language support for aggregation). BigDatalog, Myria, and Socialite, on the other hand, can only handle simple Datalog programs with limited recursion capabilities (*linear and non-mutual recursion*) and does not support or only partially support more complex computation structures such as *non-linear recursion, mutual recursion, and recursive aggregation*, which present in the majority of program analyses expressed in Datalog programs.

To address this divide, we ask two questions: (*i*) *what are the performance characteristics of existing parallel Datalog engines on tasks from different application domains?* (*ii*) *can we design and implement an efficient parallel general-purpose engine that can support a wide spectrum of Datalog programs?* To answer these questions, we perform a detailed experimental evaluation of different Datalog engines across tasks from graph analytics and program analysis, and compare their performance with our own in-memory parallel Datalog engine, which uses a relational data management system as a backend. We systematically examine the techniques and optimizations necessary to transform a naive Datalog solver into a highly optimized one. As a consequence of this work, we also show that – contrary to anecdotal and empirical evidence [SVKW15, JSS16]—it is possible to effectively use a RDBMS as a backend for Datalog evaluation through careful consideration of the underlying system issues.

Our Contribution. In summary, the project makes the following contributions:

1. *Benchmarking.* We perform an extensive comparison of four state-of-the-art Datalog engines on a multi-core machine. We consider benchmarks from the domains of graph analytics (e.g., *transitive closure*, *reachability*, *connected components*) and program analysis (e.g., *points-to* and *dataflow analyses*) using both synthetic and real-world datasets. We compare these systems across both runtime and memory usage. Our findings are summarized in Table 3.1.

2. *Techniques and Guidelines.* We study the challenges of building a recursive query processing engine on top of a parallel RDBMS, and consider a spectrum of techniques that solve them. Key techniques include (*i*) a lightweight way to enable query re-optimization at every recursive step, (*ii*) careful scheduling of the queries issued to the RDBMS in order to maximize resource utilization, and (*iii*) the design of specialized high-performance algorithms that target the bottleneck operators of recursive query

	Graspan	**Bddbddb**	**BigDatalog**	**Souffle**	**RecStep**
Scale-Up	*yes*	*no*	*yes*	*yes*	*yes*
Scale-Out	*no*	*no*	*yes*	*no*	*no*
Memory Usage	*low*	*low*	*high*	*medium*	*low*
CPU Utilization	*medium*	*poor*	*high*	*medium*	*high*
CPU Efficiency	*low*	-	*medium*	*high*	*high*
Parameters Tuning	*light*	*complex*	*moderate*	*no*	*no*
Mutual Recursion	*yes*	*yes*	*no*	*yes*	*yes*
Non-Recursive AGG	*no*	*no*	*yes*	*yes*	*yes*
Recursive AGG	*no*	*no*	*yes*	*no*	*yes*

Table 3.1: **Summary of Comparison Between Selected Systems.** For a given workload (i.e., a Datalog program and an input dataset), *CPU efficiency* is defined as the reciprocal of the product of the overall performance (runtime) of the system supporting multi-core computation and the number of CPU cores given for computation - the greater number suggests higher CPU efficiency. Table 3.2 shows the CPU efficiency of different systems on the selected Datalog workloads. **AGG** stands for *Aggregation*.

	Graspan	**BigDatalog**	**Souffle**	**RecStep**
TC (`G20K`)	-	2.75e-04	2.92e-04	**1.12e-03**
SG (`G10K`))	-	7.18e-05	5.41e-04	**2.45e-03**
REACH (`orkut`)	-	1.92e-04	3.52e-04	**1.32e-03**
CC (`orkut`)	-	2.17e-04	-	**5.81e-04**
SSSP (`orkut`)	-	1.81e-04	-	**1.00e-03**
AA (`dataset 7`)	-	2.20e-04	5.65e-05	**7.65e-04**
CSDA (`linux`)	2.22e-06	1.29e-04	**2.05e-04**	5.81e-05
CSPA (`linux`)	4.56e-05	-	2.03e-04	**4.10e-04**

Table 3.2: CPU Efficiency of Different Systems on Selected Datalog Programs and Datasets

processing (*set difference*, *deduplication*). We also propose a specialized technique for graph analytics that can compress the intermediate data through the use of a bit matrix to reduce memory usage. We systematically measure the effect of each technique on performance.

3. *Implementation.* We implement our techniques as part of RecStep, a Datalog engine built on top of QuickStep [PDZ$^+$18], which is a single-node in-memory parallel RDBMS. RecStep supports a language extension of pure Datalog with both stratified negation

and aggregation, a language fragment that can express a wide variety of data processing tasks.

4. *Evaluation.* We experimentally show that RecStep can efficiently solve large-scale problems in different domains using a single-node multi-core machine with large memory, and also can scale well when given more cores. RecStep outperforms the other systems in almost all cases , sometimes even by a factor of 8. In addition, the single-node implementation of RecStep compares favorably to cluster-based engines, such as BigDatalog, which use far more resources (more processing power and memory). Our results show that (*i*) it is feasible to build a fast general-purpose Datalog engine using an RDBMS as a backend, and (*ii*) with the trend towards powerful (multi-core and large main memory) servers, single-node systems may be sufficient for a large class of Datalog workloads.

Organization. In Section 3.1, we give a brief introduction of Quickstep, the in-memory RDBMS used as the backend of RecStep. We then give a summary of the architecture design of RecStep in Section 3.2. We discuss in detail the techniques and optimizations that lead to RecStep's high performance for Datalog evaluation in Section 3.3 followed by a comprehensive experimental evaluation in Section 3.4.

3.1 QuickStep

Quickstep [PDZ$^+$18] is a single-node parallel RDBMS that focuses on in-memory query processing. To fully exploit the large amount of parallelism that is packed inside the multi-core servers today, Quickstep builds on a wide spectrum of mechanisms and carefully designed components such as *block-layout storage manager*, which is further leveraged by the query execution paradigm that allows for *high intra-operator parallelism*, in which *multiple work orders* are produced and *processed independently* at the *block level.* Besides, Quickstep also comes up with a scheduler that allows for elastic resource allocation, the mechanisms to quickly drop irrelevant data as early as possible during the query processing, and many other techniques focusing on efficient in-memory processing.

For one of the central computation tasks when processing relational data *join processing*, QuickStep implements a hash join algorithm consists of *join phase* and *build phase.* The latch-free concurrent hash table is used to operate on multiple blocks by multiple work orders in parallel and the joined tuples are materialized into in-memory blocks.

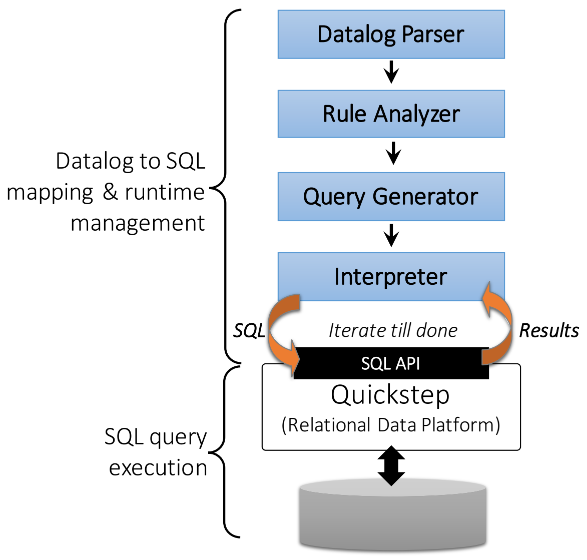


Figure 3.1: Architectural overview of RecStep

3.2 Architecture

In this section, we present the architecture of RecStep. The core design choice of RecStep is that, in contrast to other existing Datalog engines, it is built on top of an existing parallel in-memory RDBMS (QuickStep). This design enables the use of existing techniques (e.g., indexing, memory management, optimized operator implementations) that provide high-performance query execution in a multi-core environment. Further, it allows us to improve performance by focusing on characteristics specific to Datalog evaluation.

Overview. The architecture of our system is summarized in Figure 3.1. The Datalog program is read from a `.Datalog` file, which, along with the rules of the Datalog program, specifies the schemas of IDB and EDB relations. The parsed program is first given as input to the *rule analyzer*. The job of the rule analyzer is to preprocess the program: identify the IDB and EDB relations, verify the syntactic correctness of the program, construct the dependency graph and stratification, and build the necessary mapping information (e.g., for joins). Next, the *query generator* takes the output of the rule analyzer and produces the necessary SQL code to evaluate each stratum of the Datalog program using semi-naïve evaluation. Finally, the *interpreter* is responsible for the evaluation of the program. It starts the RDBMS server, creates the IDB and EDB tables in the database, and takes care of the loop control for the semi-naïve evaluation in each stratum. It also controls the communication and flow of information between the RDBMS server.

Execution. We now describe how the interpreter executes a Datalog program, as outlined in Algorithm 1.

Function	**Description**
$\mathsf{idb}(s)$	returns relations that are heads in stratum s
$\mathsf{rules}(R, s)$	returns rules of stratum s with R as head
$\mathsf{uieval}(r)$	evaluates all the rules in the set r
$\mathsf{analyze}(R)$	call to the RDBMS to collect statistics for R
$\mathsf{dedup}(R)$	deduplicates R

Table 3.3: Notation used in Algorithm 1

Algorithm 1: Execution Strategy for Datalog program P

```
1:  for each IDB $R$
2:      $R \leftarrow \emptyset$
3:  // $\mathcal{S}$ is a stratification of $P$
4:  for each stratum $s \in \mathcal{S}$
5:      repeat
6:          for each $R \in \mathsf{idb}(s)$
7:              $R_t \leftarrow \mathsf{uieval}(\mathsf{rules}(R, s))$
8:              $\mathsf{analyze}(R_t)$
9:              $R_\delta \leftarrow \mathsf{dedup}(R_t)$
10:             $\mathsf{analyze}(R_\delta, R)$
11:             $\Delta R \leftarrow R_\delta - R$
12:             $R \leftarrow R \uplus \Delta R$
13:         if $s$ is non-recursive
14:             break
15:     until $\forall R \in \mathsf{idb}(s)$, $\Delta R = 0$
```

The Datalog rules are evaluated in groups and in the order given by the stratification. The IDB relations are initialized so that they are empty (line 2). For each stratum, the interpreter enters the control loop for semi-naïve evaluation. Note that in the case where the stratum is non-recursive (i.e., all the rules are non-recursive), the loop exits after a single iteration (line 13). In each iteration of the loop, two temporary tables are created for each IDB R in the stratum: ΔR, which stores only the new facts produced in the current iteration, and a table that stores the result at the end of the previous iteration. These tables are deleted immediately after the evaluation of the next iteration is complete.

The function uieval executes the SQL query that is generated from the query generator based on the rules in the stratum where the relation appears in the head (more details on the next section). We should note here that deduplication does not occur within uieval, but in

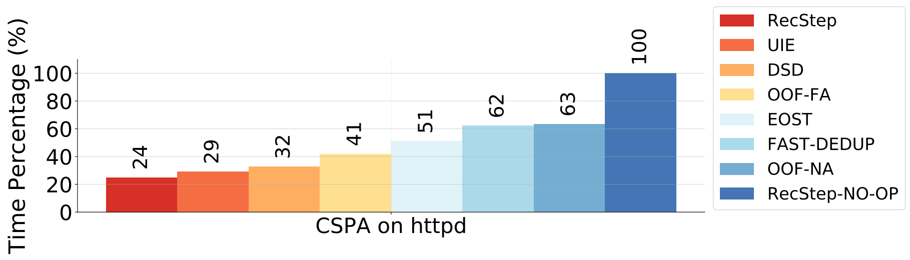


Figure 3.2: **The effect of different optimizations techniques on the CSPA analysis on `httpd` dataset**: the figure shows the effect of *turning off* each optimization on runtime, depicted as a percentage over the runtime of RecStep with all optimizations turned off (RecStep-NO-OP).

a separate call (dedup). This is achieved in practice by using `UNION ALL` (simply appending data) instead of `UNION`.

Finally, we should remark that the interpreter calls the function analyze() during execution, which tells the backend explicitly to collect statistics on the specified table. As we will see in the next section, analyze() is a necessary feature to achieve a lightweight and dynamic query optimization.

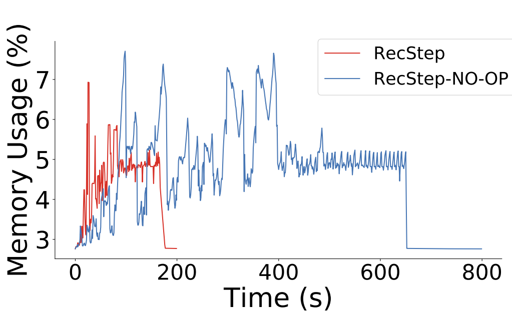
3.3 Optimizations

This section presents the key optimizations that we have implemented in RecStep to speed up performance and efficiently utilize system resources (memory and cores). We consider optimizations at two levels: Datalog-to-SQL level and system level.

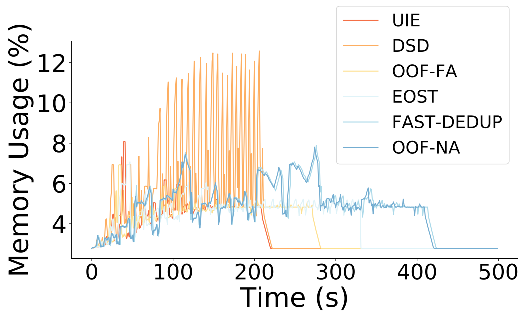
For Datalog-to-SQL-level optimizations, we study the translation of Datalog rules to a set of SQL queries. An effective translation minimizes the overhead of catalog updates, selects the optimal algorithms and query plans, avoids redundant computations, and fully utilizes the available parallelism. In terms of system-level optimizations, we focus on bottlenecks that cannot be resolved by the translation-level optimizations, and modify the back-end system by introducing new specialized data structures, implementing efficient algorithms, and revising the rules in the query optimizer. We summarize our optimizations as follows:

1. *Unified* IDB *Evaluation* (UIE): different rules and different subqueries inside each recursive rule evaluating the same IDB relation are issued as a single query.

2. *Optimization On the Fly* (OOF): the same set of SQL queries are re-optimized at each iteration considering the change of IDB tables and intermediate results.



(a) RecStep



(b) Optimizations

Figure 3.3: The effect of *turning off* each optimization of CSPA analysis on `httpd` on memory usage. The memory consumption with all optimization turned on/off is shown separately in Figure 3.3a for comparison purposes.

3. *Dynamic Set Difference* (DSD): for each IDB table, the algorithm to perform the set difference to compute Δ is dynamically chosen at each iteration, by considering the size of IDB tables and intermediate results.

4. *Evaluation as A Single Transaction* (EOST): the evaluation of a whole Datalog program is regarded as a single transaction and nothing is committed until the end.

5. *Fast Deduplication* (FAST-DEDUP): a memory-efficient implementation for high-performance deduplication.

Apart from the above list of optimizations, we also provide a specialized technique that can speed up performance on Datalog programs that operate on dense graphs called PBME. This technique represents the relation as a bit-matrix, with the goal of minimizing the memory footprint of the algorithm. Next, we detail each optimization; their effect on runtime and memory is visualized in Figure 3.2 and Figure 3.3.

Datalog-to-SQL-level optimizations

Unified IDB Evaluation (UIE). For each IDB relation R, there can be several rules where R appears in the head of the rule. In addition, for a nonlinear recursive rule, in which the rule body contains more than one IDB relation, the IDB relation is evaluated by multiple subqueries. In this case, instead of producing a separate SQL query for each rule and then computing the union of the intermediate results, the query generator produces a *single* SQL query using the `UNION ALL` construct. We call this method *unified* IDB *evaluation* (UIE). Figure 3.4 provides an example of the two different choices for the case of Andersen analysis.

The idea underlying UIE is to fully utilize all the available resources, i.e., all the cores in a multi-core machine. QuickStep does not allow the concurrent execution of SQL queries, and

Individual IDB Evaluation:

```
// Evaluate and write results separately
INSERT INTO pointsTo_tmp_mDelta0
    SELECT a0.y AS y, p1.x AS x FROM …
INSERT INTO pointsTo_tmp_mDelta1
    SELECT l0.y AS y, p2.x AS x FROM …;
  …
INSERT INTO pointsTo_tmp_mDelta6
    SELECT p1.x AS y, p2.x AS x FROM …;
// Merge results from seperate queries
INSERT INTO pointsTo_mDelta
    SELECT * FROM pointsTo_tmp_mDelta0
        UNION ALL
    SELECT * FROM pointsTo_tmp_mDelta1
        UNION ALL
      …
    SELECT * FROM pointsTo_tmp_mDelta6;
```

Unified IDB Evaluation:

```
// Evaluate and write results as a whole
INSERT INTO pointsTo_mDelta
    SELECT a0.y AS y, p1.x AS x FROM …
        UNION ALL
    SELECT l0.y AS y, p2.x AS x FROM …
        UNION ALL
      …
    SELECT p1.x AS y, p2.x AS x FROM …;
```

Figure 3.4: Example UIE in Andersen analysis.

hence by grouping the subqueries into a single query, we maximize the number of tasks that can be executed in parallel without explicitly considering concurrent multi-task coordination.

In addition, UIE mitigates the overhead incurred by each individual query call, and enables the query optimizer to jointly optimize the subqueries (e.g., enable cache sharing, pipelining instead of materializing intermediate results). The latter point is not specific to QuickStep, but generally applicable to any RDBMS backend (even ones that support concurrent query processing).

Optimization On the Fly (OOF). In Datalog evaluation, even though the set of queries is fixed across iterations, the input data to the queries changes, since the IDB relations and the corresponding Δ-relations change at every iteration. This means that the optimal query plan for each query may be different across different iterations. For example, in some Datalog programs, the size of ΔR (Algorithm 1) produced in the first few iterations might be much larger than the joining EDB table, and thus the hash table should be preferably built on the EDB when performing a join. However, as the ΔR produced in later iterations tends to become smaller, the build side of the hash table should be switched later.

In order to achieve optimal performance, it is necessary to re-optimize each query at every iteration (lines 8, 10 in Algorithm 1) by using the latest table statistics from the previous iteration. However, collecting the statistical data (e.g., size, avg, min, max) of the whole database at every iteration can cause a large overhead, since it may be necessary to perform a full scan of all tables. To mitigate this issue, our solution is to control precisely at which point which statistical data we collect for the query optimizer, depending on the type of the query. For instance, before joining two tables, only the size of the two tables is necessary for the optimizer to choose the right side to build the hash table on (the smaller table), as illustrated in the previous example.

In particular, we collect the following statistics:

- For *deduplication*, the size of the hash table needs to be estimated in order to pre-allocate memory. we use a conservative approximation that takes the size of the table.
- For *join processing*, we collect only the number of tuples and the tuple size of the joining tables, if any of the tables is updated or newly created.
- For *aggregation*, we collect statistics regarding the min, max, sum and avg of the tables.

The effect of OOF can be seen in Figure 3.4. Without updating the statistics across the iterations, the running time percentage jumps from 24% to 63% (OOF-NA). On the other hand, if we update the full set of statistics, the running time percentage increases to 41% (OOF-FA).

Dynamic Set-Difference (DSD). In semi-naïve evaluation, the execution engine must compute the *set difference* between the newly evaluated results (R_δ) and the entire recursive relation (R) at the end of every iteration, to generate the new $\Delta R \leftarrow R_\delta - R$ (line 12 in Algorithm 1). Since set difference is executed at every iteration for every IDB in the stratum, it is a computational bottleneck that must be highly optimized. There exist two different ways we can translate set difference as a SQL query.

The first approach *One-Phase Set Difference* (OPSD) simply runs the set difference as a single SQL query. The default strategy that QuickStep uses for set difference is to first build a hash table on R, and then R_δ probes the hash table to output the tuples of R_δ that do not match with any tuple in the hash table. Since the size of R grows at each iteration (recall that Datalog is monotone), this suggests that the cost of building the hash table on R will constantly increase for the set difference computation under OPSD.

An alternative approach is to use a translation that we call *Two-Phase Set Difference* (TPSD). This approach involves two queries: the first query computes the intersection of the two relations, $r \leftarrow R \cap R_\delta$. The second query performs set difference, but now between R and r (instead of R_δ). Although this approach requires more relational operators, it avoids building a hash table on R.

We observe that none of the two approaches always dominates the other, since the size of R and R_δ changes at different iterations. Hence, we need to *dynamically* choose the best translation at every iteration. We guide this choice using a simple cost model, presented in full detail in [FZZ$^+$18].

System-level Optimizations

Evaluation as One Single Transaction (EOST). By default, QuickStep (as well as other RDBMSs) view each query that changes the state of database as a separate transaction. Keeping the default transaction semantics in QuickStep during evaluation incurs I/O overhead in each iteration due to the frequent insertion happening to IDB tables, and the creation of tables storing intermediate results. Such frequent I/O actions are unnecessary, since we only need to commit the final results at the end of the evaluation. To avoid this overhead, we use the *evaluation as one single transaction* (EOST) semantics. Under these semantics, the data is kept in memory until the fixpoint is reached (when there is enough main memory), and only the final results are written to persistent storage at the end of evaluation.

To achieve EOST, we slightly modify the kernel code in QuickStep to pend the I/O actions until the fixpoint is reached (by default, if some pages of the table are found *dirty* after a query execution, the pages are written back to the disk). At the end of the evaluation, a signal is sent to QuickStep and the data is written to disk.

For other popular RDBMSs (e.g., PostgreSQL, MySQL, SQL Server), the start and the end of a transaction can be explicitly specified, but this approach is only feasible for a set of queries that are pre-determined.[1]

However, in recursive query processing the issued queries are dynamically generated, and the number of iterations is not known until the fixpoint is reached, which means that similar changes need to be made in these systems to apply EOST.

Fast Deduplication. In Datalog evaluation, *deduplication* of the evaluated facts is not only necessary for conforming to the set semantics, but also helps to avoid redundant computation. Deduplication is also a computational bottleneck, since it occurs at every iteration for every IDB in the stratum (line 10 in Algorithm 1); hence, it is necessary to optimize its parallel execution.

To achieve this, we use a specialized *Global Separate Chaining Hash Table* implementation that uses a *Compact Concatenated Key* (CK), which we call CCK-GSCHT. CCK-GSCHT is a global latch-free hash table built using a compact representation of $\langle key, value \rangle$ pairs, in which tuples from each data partition/block can be inserted *in parallel*. Figure 3.5 illustrates the deduplication algorithm using an example in which CCK-GSCHT is applied on a table with two integer attributes (src int, dest int).

Based on the approximated number of distinct elements from the query optimizer, RecStep *pre-allocates* a list of buckets, where each bucket contains only a pointer. An important point here is that the number of pre-allocated buckets will be as large as possible when there is

[1]To fully achieve EOST, transactional databases also need to turn off features such as checkpoint, logging for recovery, etc.

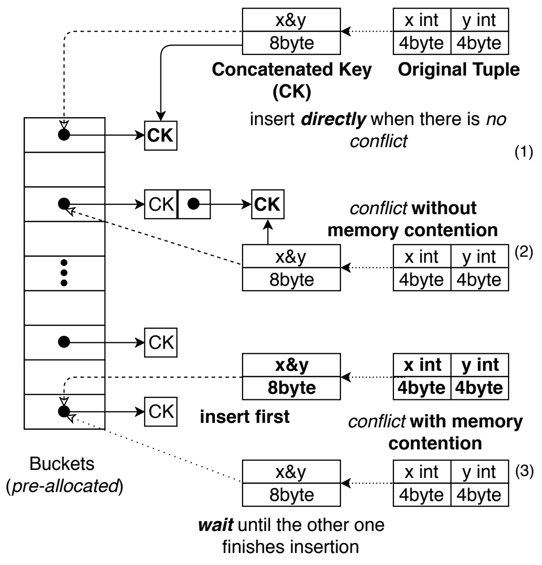

x&y
8byte
x int
y int
4byte
4byte
Concatenated Key (CK)
Original Tuple
insert *directly* when there is *no conflict*
(1)
CK
conflict without memory contention
(2)
conflict with memory contention
(3)
insert first
wait until the other one finishes insertion
Buckets (*pre-allocated*)

Figure 3.5: Example of applying fast deduplication algorithm on table with two integer attributes in RecStep

enough memory, for the purpose of minimizing conflicts in the same bucket, and preventing memory contention. Tuples are assigned to each thread in a round-robin fashion and are inserted in parallel. Knowing the length of each attribute in the table, a compact CK of fixed size [2] is constructed for each tuple (8 bytes for two integer attributes as shown in Figure 3.5). The compact CK itself contains all information of the original tuple, eliminating the need for explicit $\langle key, value\rangle$ pair representation. Additionally, the key itself is used as the hash value, which saves computational cost and memory space.

Parallel Bit-Matrix Evaluation

In our experimental evaluation, we observed that the usage of memory increased drastically during evaluation over dense relations. By default, QuickStep uses hash tables for joins between tables, aggregation and deduplication. When the intermediate result becomes very large, the use of hash tables for join processing becomes memory-costly. In the extreme, the intermediate results are too big to fit in main memory, and are forced to disk, incurring additional I/O overhead, or even out-of-memory errors. This phenomenon was observed in both graph analytics and program analysis. In both cases, a Datalog program starts with sparse input relations with a relatively small active domain, but end up with large and dense output relations. A typical example of this behavior is transitive closure on graphs.

Inspired by this observation, we exploit a specialized data structure, called *bit-matrix*, that

[2]The inputs of Datalog programs are usually integers transformed by mapping the *active domain* of the original data (if not integers). Thus the technique can also applied to data where the original type has varied length.

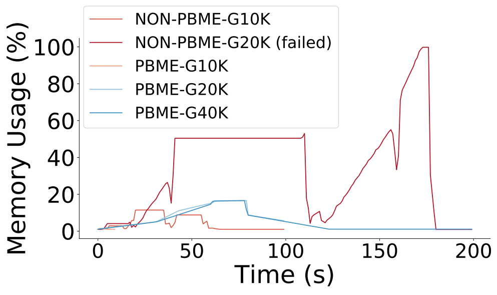
Algorithm 2: Parallel Bit-Matrix Evaluation of TC

```
 1: Input: $edge(x, y)$ - EDB relation, number of threads $k$
 2: Output: $reachable(x, y)$ - IDB relation
 3: // $M_{edge}$: virtual bit-matrix of $\texttt{edge(x, y)}$
 4: Construct bit-matrix $M_{reachable}$ of $\texttt{reachable(x, y)}$
 5: $M_{reachable} \leftarrow M_{edge}$
 6: Partition the rows of $M_{reachable}$ into $k$ partitions
 7: // the $k$ threads evaluate $k$ partitions in parallel
 8: for each row $i$ in partition $p$
 9:     $\delta \leftarrow \{u \mid M_{edge}[i, u] = 1\}$
10:     while $\delta \neq \emptyset$
11:         $\delta_n \leftarrow \emptyset$
12:         for each $t \in \delta$
13:             for each $j$ s.t. $M_{edge}[t, j] = 1$
14:                 if $M_{reachable}[i, j] = 0$
15:                     $\delta_n \leftarrow \delta_n \cup \{j\}$
16:                     $M_{reachable}[i, j] \leftarrow 1$
17:         $\delta \leftarrow \delta_n$
```

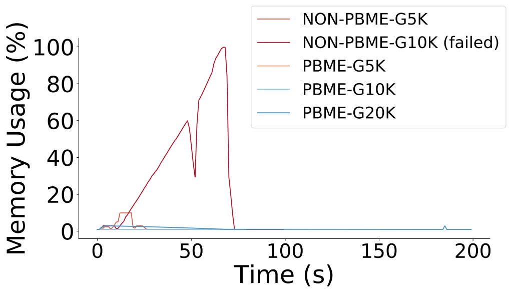
replaces a hash map during join and deduplication in the case when the graph representing the data is *dense* and has relatively small number of vertices. This data structure represents the join results in a much more compact way under certain conditions, greatly reducing the memory cost compared to a hash table. In this paper, we only describe the bit matrix for binary relations, but the technique can be extended to relations of higher arity. At the same time, we implement new operators directly operating on the bit-matrix, naturally merging the join and deduplication into *one single operation* and thus avoid the materialization cost of the intermediate results. We call this technique *Parallel Bit-Matrix Evaluation* (PBME). Our experiments (Figure 3.6) show that PBME improved performance for transitive closure (TC) and same generation (SG). The latter is expressed through the following program:

$$\texttt{sg(x, y) :- arc(p, x), arc(p, y), x} \neq \texttt{y.}$$
$$\texttt{sg(x, y) :- arc(a, x), sg(a, b), arc(b, y).}$$

We next describe the bit-matrix technique and show how to apply it for *TC* (Algorithm 2). We discuss the algorithm for SG in [FZZ+18]. Note that we construct a matrix only for each IDB, but for convenience of illustration, we use matrix notation for the EDBs as well (line 3 in Algorithm 2).



(a) Transitive Closure



(b) Same Generation

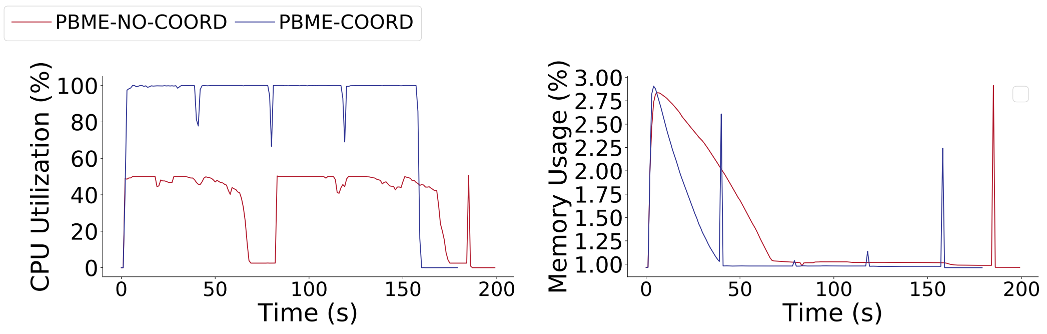
Figure 3.6: Memory Saving of PBME on TC and SG

The Bit-Matrix Data Structure. Let $R(x, y)$ be a binary IDB relation, with active domain $\{1, 2, \ldots, n\}$ for both attributes. Instead of representing R as a set of tuples, we represent it as an $n \times n$ bit matrix denoted M_R. If $R(a, b)$ is a tuple in R, the bit at the a-th row and b-th column, denoted $M_R[a, b]$ is set to 1, otherwise it is 0. The relation is updated during recursion by setting bits from 0 to 1. We decide to build the bit-matrix data structure only if the memory available can fit both the bit matrix, as well as any additional index data structures used during evaluation.

One of the key features of PBME is *zero-coordination*: each thread is only responsible for the partition of the data assigned to it and there is almost no coordination between different threads. Algorithm 2 outlines PBME for transitive closure. For the evaluation of TC (Algorithm 2), the rows of the IDB bit-matrix $M_{reachable}$ are firstly partitioned in a round-robin fashion (line 6). For each row i assigned to each thread, the set δ stores the new bits (paths starting from i) produced at every iteration (line 12-16). For each new bit t produced, the thread searches for all the bits at row t of M_{edge}, and computes the new δ (line 13-16).

The Effect of Skew. While unnoticeable in TC, data skew across different threads was observed in SG. To analyze the effect of data skew, we implement a variant of PBME with coordination (PBME-COORD) and compare it with PBME without coordination. PBME-COORD mitigates the data skew by rebalancing the workloads between threads. In the case where there is no skew, coordination only incurs unnecessary overhead. We refer readers to [FZZ+18] for more detail.

The comparison of the two algorithms is shown in Figure 3.7. The CPU utilization of PBME with coordination is almost 100% throughout the entire SG evaluation, and it takes less time to finish compared to PBME without coordination. This demonstrates that skew can indeed affect performance.



(a) CPU Utilization (b) Memory Usage

Figure 3.7: PBME with Coordination v.s Non-Coordination on SG on the `G20K` dataset.

3.4 Experiments

In this section, we evaluate the performance of RecStep. Our experimental evaluation focuses on answering the following two questions:

1. How does our proposed system scale with increased computation power (cores) and data size?

2. How does our proposed system perform compared to other parallel Datalog evaluation engines?

To answer these two questions, we performed experiments using several benchmark Datalog programs from the literature: both from traditional *graph analytics* tasks (e.g., reachability, shortest path, connected components), as well as *program analysis* tasks (e.g., pointer static analysis). We compare RecStep against a variety of state-of-the-art Datalog engines, as well as a recent single-machine, multi-core engine (Graspan), which can express only a subset of Datalog.

Experimental Setup

We briefly describe here the setup for our experiments.

System Configuration. Our experiments are conducted on a bare-metal server in Cloudlab [Clo18], a large cloud infrastructure. The server runs Ubuntu 14.04 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. The server has 160GB memory, and each NUMA node is directly attached to 80GB of memory.

Table 3.4: Summary of Datalog Programs and Datasets in Performance Evaluation

Graph Analytics	**Datasets**	**Reference**
Transitive Closure (TC)	[`G`n-p]	[SYI+16]
Same Generation (SG)	[`G`n-p]	[SYI+16]
Reachability (REACH)	`social-network graphs`, [`RMAT`]	[SYI+16]
Connected Components (CC)	`social-network graphs`, [`RMAT`]	[SYI+16]
Single Source Shortest Path (SSSP)	`social-network graphs`, [`RMAT`]	[SYI+16]
Program Analysis	**Datasets**	**Reference**
Andersen's Analysis (AA)	7 synthetic datasets	-
Context-sensitive Dataflow Analysis (CSDA)	[`linux`, `postgresql`, `httpd`]	[WHZ+17]
Context-sensitive Points-to Analysis (CSPA)	[`linux`, `postgresql`, `httpd`]	[WHZ+17]

Other Datalog Engines. We compare the performance of RecStep with several state-of-the-art systems that perform either general Datalog evaluation, or evaluate only a fragment of Datalog for domain-specific tasks.

1. BigDatalog [SYI+16] is a general-purpose distributed Datalog system implemented on top of Apache Spark.[3]
2. Souffle [SJSW16] is a parallel Datalog evaluation tool that compiles Datalog to a native C++ program. It focuses on evaluating Datalog programs for the domain of static program analysis.[4]
3. BDDBDDB [WACL05] is a single-thread Datalog solver designed for static program analysis. Its key feature is the representation of relations using binary decision diagrams (BDDS).
4. Graspan [WHZ+17] is a single-machine disk-based parallel graph system, used mainly for interprocedural static analysis of large-scale system code.

Benchmark Programs and Datasets

We conduct our experiments using Datalog programs that arise from two different domains: *graph analytics* and *static program analysis*. The graph analytics benchmarks are

[3]BigDatalog exhibits significant performance improvements over Myria and Socialite, and therefore we do not compare against them.

[4]Recent work has shown that Souffle outperforms LogicBlox [ATS17]. Indeed, our early attempts using LogicBlox confirm that its performance is not comparable to other parallel Datalog systems. Thus, we exclude LogixBlox from our experimental evaluation.

those used for evaluating BigDatalog [SYI+16]. Below, we present them in detail (with the exception of TC and SG, which are described earlier in the paper).

Reachability **(REACH)**

```
reach(y) :- id(y).
reach(y) :- reach(x), arc(x, y).
```

Connected Components **(CC)**

```
cc3(x, MIN(x)) :- arc(x, _).
cc3(y, MIN(z)) :- cc3(x, z), arc(x, y).
cc2(x, MIN(y)) :- cc3(x, y).
         cc(x) :- cc2(_, x).
```

Single Source Shortest Path **(SSSP)**

```
     sssp2(y, MIN(0)) :- id(y).
sssp2(y, MIN(d1 + d2)) :- sssp2(x, d1), arc(x, y, d2).
      sssp(x, MIN(d)) :- sssp2(x, d).
```

The static analysis benchmarks include analyses on which Graspan was evaluated [WHZ+17], as well as a classic static analysis called Andersen's analysis [And94].

Andersen's Analysis (AA)

```
pointsTo(y, x) :- addressOf(y, x).
pointsTo(y, x) :- assign(y, z), pointsTo(z, x).
pointsTo(y, w) :- load(y, x), pointsTo(x, z), pointsTo(z, w).
pointsTo(z, w) :- store(y, x), pointsTo(y, z), pointsTo(x, w).
```

***Context-sensitive Points-to Analysis* (CSPA)**[5]

$$
\begin{aligned}
\texttt{valueFlow}(\texttt{y},\texttt{x}) &\text{ :- } \texttt{assign}(\texttt{y},\texttt{x}).\\
\texttt{valueFlow}(\texttt{x},\texttt{x}) &\text{ :- } \texttt{assign}(\texttt{x},\texttt{y}).\\
\texttt{valueFlow}(\texttt{x},\texttt{x}) &\text{ :- } \texttt{assign}(\texttt{y},\texttt{x}).\\
\texttt{memoryAlias}(\texttt{x},\texttt{x}) &\text{ :- } \texttt{assign}(\texttt{y},\texttt{x}).\\
\texttt{memoryAlias}(\texttt{x},\texttt{x}) &\text{ :- } \texttt{assign}(\texttt{x},\texttt{y}).\\
\texttt{valueFlow}(\texttt{x},\texttt{y}) &\text{ :- } \texttt{assign}(\texttt{x},\texttt{z}), \texttt{memoryAlias}(\texttt{z},\texttt{y}).\\
\texttt{valueFlow}(\texttt{x},\texttt{y}) &\text{ :- } \texttt{valueFlow}(\texttt{x},\texttt{z}), \texttt{valueFlow}(\texttt{z},\texttt{y}).\\
\texttt{memoryAlias}(\texttt{x},\texttt{w}) &\text{ :- } \texttt{dereference}(\texttt{y},\texttt{x}), \texttt{valueAlias}(\texttt{y},\texttt{z}),\\
&\phantom{\text{ :- }} \texttt{dereference}(\texttt{z},\texttt{w}).\\
\texttt{valueAlias}(\texttt{x},\texttt{y}) &\text{ :- } \texttt{valueFlow}(\texttt{z},\texttt{x}), \texttt{valueFlow}(\texttt{z},\texttt{y}).\\
\texttt{valueAlias}(\texttt{x},\texttt{y}) &\text{ :- } \texttt{valueFlow}(\texttt{z},\texttt{x}), \texttt{memoryAlias}(\texttt{z},\texttt{w}),\\
&\phantom{\text{ :- }} \texttt{valueFlow}(\texttt{w},\texttt{y}).
\end{aligned}
$$

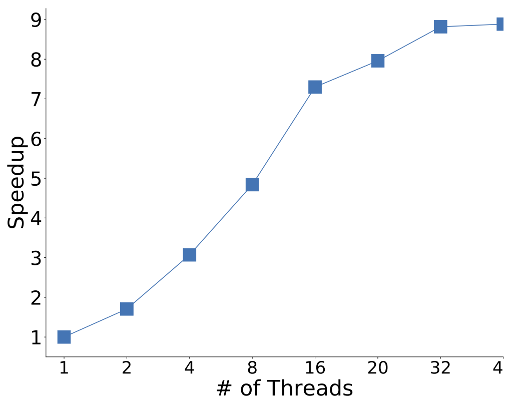
***Context-sensitive Dataflow Analysis* (CSDA)**

$$
\begin{aligned}
\texttt{null}(\texttt{x},\texttt{y}) &\text{ :- } \texttt{nullEdge}(\texttt{x},\texttt{y}).\\
\texttt{null}(\texttt{x},\texttt{y}) &\text{ :- } \texttt{null}(\texttt{x},\texttt{w}), \texttt{arc}(\texttt{w},\texttt{y}).
\end{aligned}
$$

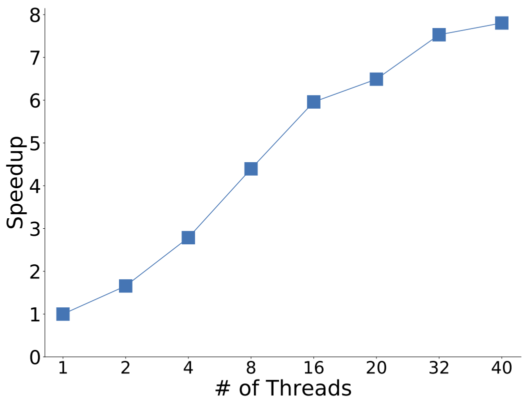
To evaluate the benchmark programs, we use a combination of synthetic and real-world datasets, which are summarized in Table 3.4. To give a better view of the performance evaluation, we briefly summarize some of the datasets and corresponding Datalog programs here. For more details, readers can go to the reference in Table 3.4.

Gn-p graphs are graphs generated by the GTgraph synthetic graph generator [GTg], where n represents the number of total vertices of the graph in which each pair of vertices is connected by probability p. Each pair of vertices in Gn omitting p is connected with probability 0.001. All Gn-p graphs are very dense considering their relatively small number of vertices. SG and TC generate very large results when evaluation is performed on Gn-p, (a few orders of magnitude larger than the number of vertices). `RMAT` graphs are graphs generated by the RMAT graph generator [GTg], with the same specification in [SYI+16], `RMAT-n` represents the graph that has n vertices and $10n$ directed edges ($n \in \{1\text{M}, 2\text{M}, 4\text{M}, 8\text{M}, 16\text{M}, 32\text{M}, 64\text{M}, 128\text{M}\}$ in our evaluation experiments). `livejournal`, `orkut`, `arabic`, `twitter` are all large-scale real-world social-network graphs that have tens of millions of vertices and edges. For the Andersen's analysis, seven datasets are generated ranging

[5]Graspan's analysis is context-sensitive via method cloning [WL04]—therefore, calling context does not appear in the rules, but in the data.



(a) CSPA on `httpd`



(b) CC on `livejournal`

Figure 3.8: Scaling-up on Cores

from small size to large size based on the characteristics of a tiny real dataset available at hand, numbered from 1 to 7. The graph representations of the datasets are small and produce moderate number of tuples. `linux`, `postgresql`, `httpd` are all real system programs used for CSDA and CSPA experiments in [WHZ+17].

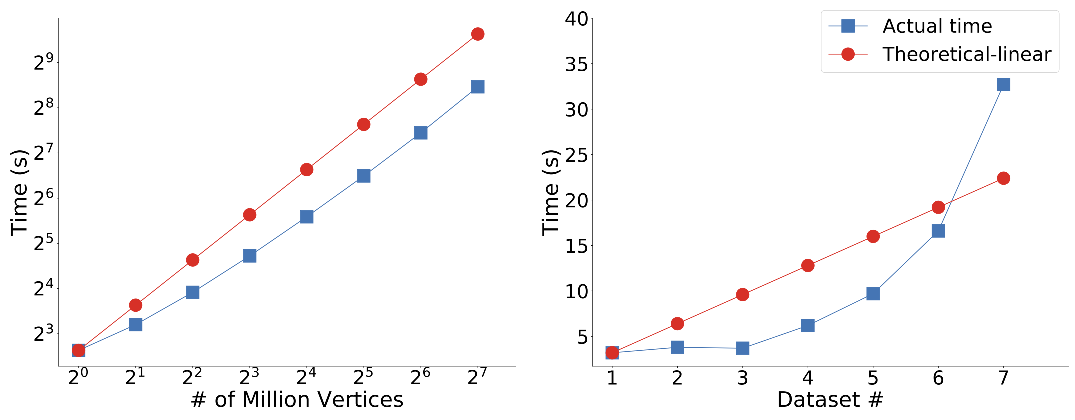
Experimental Results

We first evaluate the scalability of RecStep, and then move to a comparison with other systems.

Scalability

Scaling-up Cores. To evaluate the speedup of RecStep, we run the CC benchmark on `livejournal` graph, and the CSPA benchmark on the `httpd` dataset. We vary the number of threads from 2 to 40. Figure 3.8 demonstrates that for both cases, RecStep scales really well using up to 16 threads, and after that point achieves a much smaller speedup. This drop on speedup occurs because of the synchronization/scheduling primitive around the common shared hash table that is accessed from all the threads. Recall that 20 is the total number of physical cores.

Scaling-up Data. We perform experiments on both a graph analytics program (CC on `RMAT`) and a program analysis task (AA on the synthetic datasets) using all 20 physical cores (40 hyperthreads) to observe how our system scales over datasets of increasing sizes. From Figure 3.9a, we observe that the time increases nearly proportionally to the increasing size of the datasets. In Figure 3.9b, we observe that for datasets numbered from 1 to 3, the



(a) CC on `RMAT` (b) AA on synthetic dataset

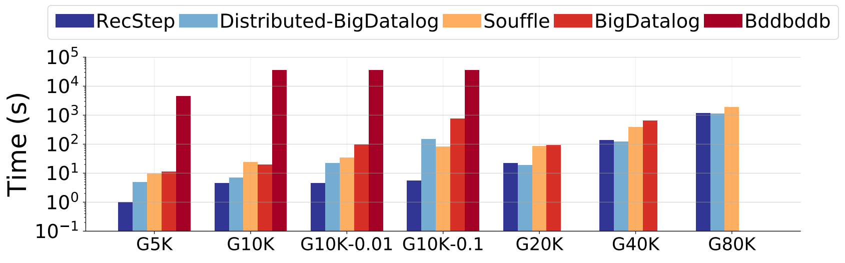
Figure 3.9: Scaling-up on Datasets: the x-axis of 3.9a represents the number of vertices of the corresponding `RMAT` graph datasets (`RMAT-1M` to `RMAT-128M`); the synthetic datasets are numbered from 1 to 7 and the x-axis of 3.9b suggests the corresponding dataset number.

evaluation times on these three datasets are roughly the same. The corresponding graphs representing the input for these three datasets are relatively sparse and the total size of the data (input and intermediate results) during evaluation is small, and the cores/threads are underutilized; thus, when the data increases, the stale threads will take over the extra processing, and runtime will not increase. With the increasing size of datasets (4 to 7), we observe a similar trend as seen in Figure 3.9a since the size of the input data and the intermediate results produced increases. All cores are fully utilized, so more data will cause increase in runtime.

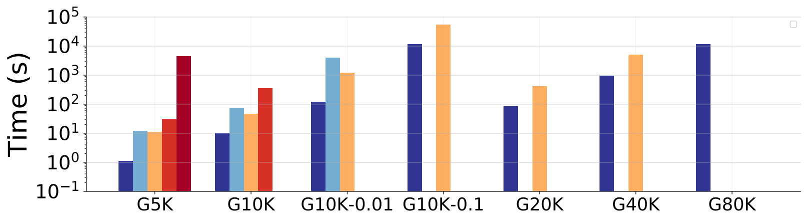
Comparison With Other Systems

In this section, we report experimental results on our benchmarks for several other Datalog systems and Graspan.

For each Datalog program and dataset shown in the comparison results, we run the evaluation four times (with the exception of BDDBDDB, since its runtime is substantially longer than all other systems across the workloads), we discard the first run and report the average time of the last three runs. For each system, we report the total execution time, including the time to load data from the disk and write data back to the disk. For BigDatalog, since its evaluation is *parameter workload dependent* based on the available resources provided (e.g., memory), and its performance depends on both of the supplied parameters, datasets and the programs, we tried different combinations of parameters (e.g., different join types) and report the best runtime numbers. For comparison purposes, we also display the results



(a) Transitive Closure



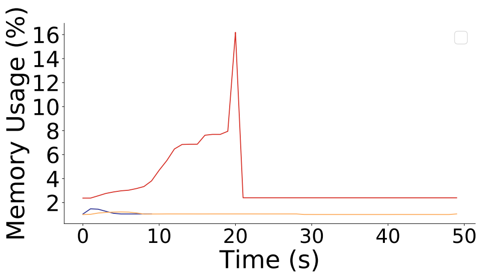
(b) Same Generation

Figure 3.10: Performance Comparison of TC and SG

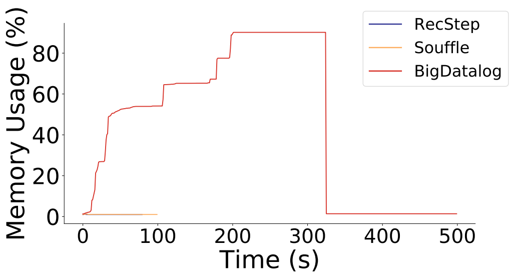
of BigDatalog that runs on the full cluster from [SYI$^+$16] (Distributed-BigDatalog in Figure 3.10, 3.12, 3.13, which has 15 worker nodes with 120 CPU cores and 450GB memory in total.) As we will see next, the experiments show that our system can efficiently evaluate Datalog programs for both large-scale graph analytics and program analyses, by being able to efficiently utilize the available resources on a single node equipped with powerful modern hardware (multi-core processors and large memory). Specific runtime numbers are not shown in Figure 3.10, 3.15a due to the space limit.

TC and SG Experiments. For TC and SG, our system uses PBME as discussed in Section 5. Since the `G`n-p graphs are very dense, in each iteration intermediate results of large sizes are produced. Hence, the original QuickStep operators run out of memory due to the high materialization cost and demand for memory. By using a *bit-matrix* data structure, the evaluation naturally fuses the join and deduplication into a single operation, avoiding the materialization cost and heavy memory usage. Our system is the only one that completes the evaluation for TC and SG on all Gn-p graphs (the runtime bar is not shown if the system fails to finish the evaluation due to OOM or timeout ($> 10h$) is observed). The evaluation time of all four systems is shown in Figure 3.10. Figure 3.11 shows the memory consumption of each system as percentage over the total memory given.

For TC, except for Distributed-BigDatalog, our system outperforms all other systems on all `G`n-k graphs (Distributed-BigDatalog is only slightly faster than RecStep on `G20K`, `G40K` and `G80K`). For `G5K`, `G10K`, `G10K-0.01`, and `G10K-0.1`, RecStep even outperforms Distributed-BigDatalog, which uses 120 cores and 450GB memory. For graphs that have more vertices,



(a) Transitive Closure



(b) Same Generation

Figure 3.11: Memory Usage of TC and SG (`G10k`)

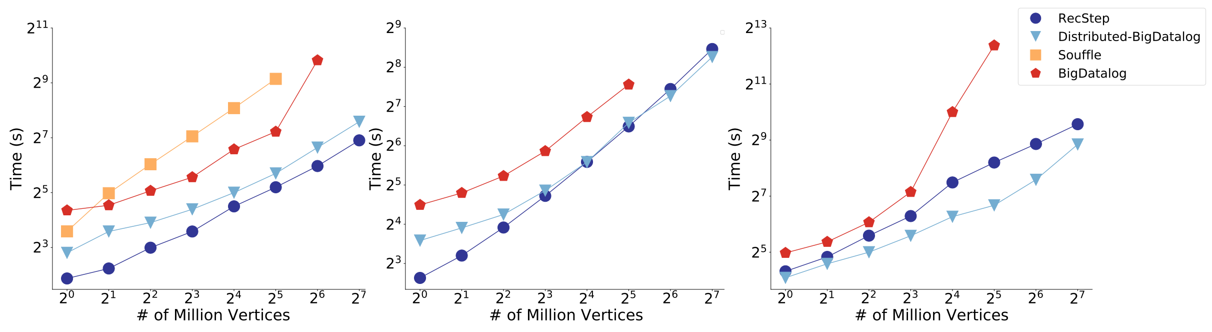
Distributed-BigDatalog slightly outperforms RecStep due to the additional CPU cores and memory it uses for evaluation. Due to the use of BDDS, BDDBDDB can only efficiently evaluate graph analytics expressed in Datalog when the graph has a relatively small number of vertices and when the proper variable ordering is given.[6] When the evaluation violates either of these two conditions, BDDBDDB is a few orders of magnitude slower than other systems as shown in graphs `G5K`, `G10K`, `G10K-0.01`, `G10K-0.1`. For graphs `G20K`, `G40K`, `G80K`, BDDBDDB runs out of time ($> 10h$). Souffle runs out of memory when evaluating TC on `G80K`.

Compared to TC, the evaluation of SG is more memory demanding and computationally expensive as observed in Figure 3.10b and 3.11. Except for our system, all other systems either use up the memory before the completion of the evaluation of SG or run into timeout ($> 15h$) on some of the `G`n`-`k graphs. Unlike TC, we observe that RecStep on SG evaluation is much more sensitive to the graph density (e.g., `G10K`, `G10K-0.01`, `G10K-0.1`).

Experiments of Other Graph Analytics Besides TC and SG, we also perform experiments running REACH, CC and SSSP on both the `RMAT` graphs and the real world graphs (Table 3.4), comparing the execution time and memory consumption (on `livejournal`) of our system with Souffle and BigDatalog (Figure 3.13, 3.14). Since Soufflle does not support recursive aggregation (which shows in CC and SSSP), we only show the execution time results of our system and BigDatalog for CC and SSSP. BDDBDDB is excluded, since all graphs have a very large number of vertices that BDDBDDB is not able to handle efficiently.

As mentioned in [SYI+16], the size of the intermediate results produced during the evaluation of REACH, CC, SSSP is $O(m)$, $O(dm)$ and $O(nm)$, where n is the number of vertices, m is the number of edges and d is the diameter of the graph. For convenience of comparison, we follow the way in which [SYI+16] presents the experimental results: for REACH and SSSP, we report the average run time over ten randomly selected vertices. We only consider an evaluation *complete* if the system is able to finish the evaluation on *all ten vertices* for

[6]The size of BDD is highly sensitive to the variable ordering used in the binary encoding; finding the best ordering is NP-complete. We let BDDBDDB pick the ordering.



(a) REACH (b) CC (c) SSSP

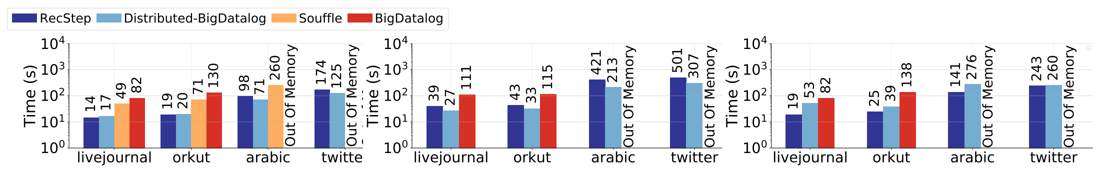
Figure 3.12: Performance Comparison on `RMAT` Graphs: The x-axis represents `RMAT-1M` to `RMAT-128M`.

REACH and SSSP, otherwise the evaluation is seen as *failed* (due to OOM). The corresponding point of failed evaluation is not reported in Figure 3.12 (on `RMAT`) and is shown as *Out of Memory* in Figure 3.13 (on real world graphs).

Besides Distributed-BigDatalog, RecStep is the *only* system that completes the evaluation for REACH, CC, SSSP on all `RMAT` graphs and real-world graphs, and is 3-6X faster than other systems using scale-up approach on all the workloads that other systems manage to finish (as shown in Figures 3.12 and 3.13); compared to Distributed-BigDatalog, RecStep shows comparable performance using far less computational resources. Both BigDatalog and Souffle fail to finish evaluating some of the workloads due to OOM. As shown in Figure 3.12, the execution time of our system increases nearly proportionally to the increasing size of the dataset on all three graph analytics tasks. In contrast, Souffle's parallel behavior is workload dependent though it efficiently evaluates dataflow and points-to analysis (Fig 3.15b, 3.15c), it does not fully utilize all the CPU cores when evaluating REACH (Fig3.13a, 3.12a) and Andersen's analysis (Fig 3.15a) . The CPU utilization of different systems evaluating Andersen's analysis, CSPA is visualized in Figure 3.16.

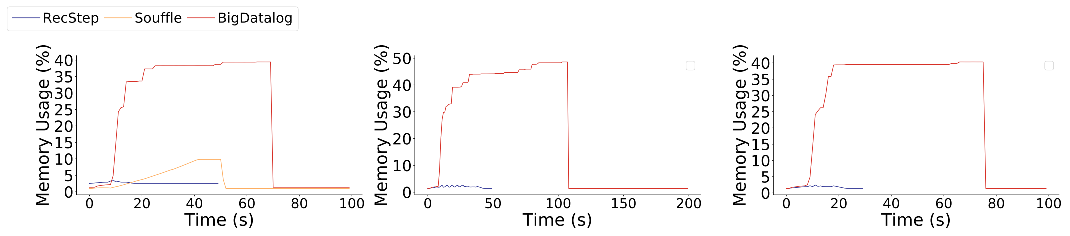
Program Analysis. We perform experiments on Andersen's analysis using the synthetic datasets (generated based on a real-world dataset). Besides, we also conduct experiments comparing the execution time of CSPA and CSDA on the real system programs in [WHZ$^+$17]. Nonlinear recursive rules are commonly observed in Datalog programs for program analysis, and the results help us understand the behavior of our system and other systems when evaluating Datalog programs with(out) involving complex recursive rules.

For Andersen's analysis, the number of variables (the size of active domains of each EDB relation) increases from `dataset 1` to `dataset 7`. Our system outperforms all other systems on every dataset. The performance of BDDBDDB is comparable to other systems when the number of variables being considered is small (`dataset 1` and `dataset 2`), but the runtime



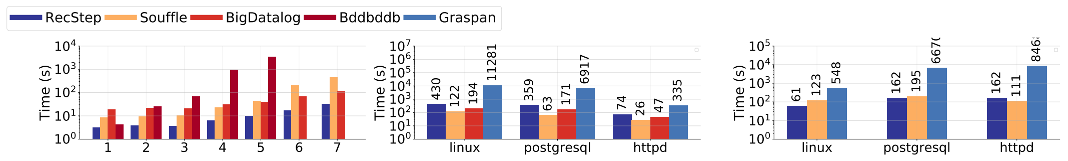
(a) REACH (b) CC (c) SSSP

Figure 3.13: Performance Comparison on Real-World Graphs.



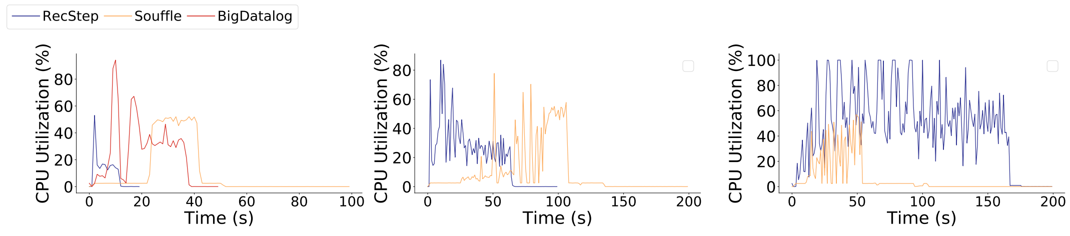
(a) REACH (b) CC (c) SSSP

Figure 3.14: Memory Consumption on `livejournal`.



(a) Evaltion of AA (b) Evaluation of CSDA (c) evaluation of CSPA

Figure 3.15: Performance Comparison on Program Analyses



(a) AA on `dataset 5` (b) CSPA on `linux` (c) CSPA on `httpd`

Figure 3.16: CPU Utilization on Program Analyses

increases a lot when the number of variables grows, due to its large overhead of building the BDD. BigDatalog outperforms Souffle on large datasets, since Souffle does not parallelize the computation as effectively.

All three systems significantly outperform Graspan on both CSPA and CSDA, as shown in Figure 3.15b and Figure 3.15c. Since BigDatalog does not support mutual recursion, it is not present in Figure 3.15c). The inefficiency of Graspan is mainly due to its frequent use of sorting, coordination during the computation and relatively poor utilization of multi-cores for parallel computation.

CSDA is the only Datalog program on which RecStep is outperformed by Souffle and BigDatalog. The reasons are two-fold. First, the evaluation of CSDA on all three datasets requires many iterations (~ 1000), and thus the overhead of triggering each query accumulates. There is also an additional overhead from the `analyze` calls and the materialization cost of the intermediate results. Compared to this overhead, the cost of the actual computation is much lower. The second reason is that the rules in CSDA are simple and linear. Since the input data and the intermediate results produced in each iteration are small in size, the RDBMS cannot fully utilize the available cores. In contrast, CSPA has more rules and involves *nonlinear recursion*, producing large Δ and intermediate results at each iteration. This enables RecStep to exploit both data-level and multiquery-level parallelism. Figure 3.15c shows the evaluation time for CSPA: while Souffle slightly outperforms our system on the `httpd` dataset, RecStep outperforms Souffle and Graspan on the other two datasets.

3.5 Summary

In this chapter, we presented the design and implementation of RecStep, a general-purpose, parallel, in-memory Datalog solver, along with the experimental comparison results of existing techniques. Specifically, we demonstrated how to implement an efficient, parallel Datalog solver atop a relational database. To achieve high efficiency, we presented a series of algorithms, data structures, and optimizations, at the level of Datalog compilation to SQL and at the level of the underlying RDBMS. Our results demonstrate the scalability of our approach, its applicability to a range of application domains, and its competitiveness with highly optimized and specialized Datalog solvers.

Most of the content of this chapter is from our paper titled *Scale-Up In-Memory Datalog Processing: Observations and Techniques* [FZZ+18] that appeared in VLDB 2019 conference. The code of RecStep is open source and is available at `https://github.com/Hacker0912/RecStep`.
In next chapter, we are going to present and discuss the tools that can help us gain a better understanding of the performance difference observed in different systems, as well as the limitations found in RecStep and how such limitations should be addressed.

Chapter 4

Recursive Computation Profiling

In this chapter, we discuss the general profiling of 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 a few existing Datalog systems, which further provides guidance on making improvements over the existing system and designing/building more efficient new systems.

Datalog is seeing a resurgence of interest in recent years and has found new applications in multiple application 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 support high-performance and scalable computation, multiple research efforts [FZZ$^+$18, RB19, SJSW16, SYI$^+$16, ZAC$^+$19] have explored ways to develop Datalog systems that are capable of efficiently handling recursive computation, some of which have been discussed in the last chapter. The resulting systems from these works often focus on a particular application domain. For example, Souffle [SJSW16] is designed and built primarily for static program analysis, and DDlog [RB19] is a Datalog implementation built on top of Differential Dataflow [MMII13] 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 (i.e., program and data), little or 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 the previous chapter, 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 between different systems and workloads. In turn, this makes choosing the best system for applications of interest (for users) and improving existing systems (for system developers) challenging. For example, when trying to build a new Datalog system, we need to answer questions such as *what techniques can we leverage in existing systems? 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 above issues, we argue that a general-purpose recursive computation profiling framework that can provide insight *across systems and workloads* is needed. In this chapter, we present and discuss four important profiling components as the first step towards building such a profiling framework: *recursion profile, runtime, CPU utilization* and *memory utilization*. We first describe these four components and briefly discuss their importance. We then present case studies based on profiling visualizations of Datalog workloads from two application domains: graph analytics and program analysis. As shown in Section 4.2, there is no single system that is always 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. 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 so the system limitations.

In this chapter, we focus mainly on three recently published well-documented Datalog systems that are publicly available, RecStep [FZZ+18], Souffle [SJSW16], and DDlog [RB19]. Souffle is a recent high-performance Datalog system that is primarily 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, as introduced in the last chapter, is a Datalog engine built on top of an efficient single-node in-memory database called Quickstep [PDZ+18], 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 to the corresponding Differential Dataflow [MMII13] program that allows incremental computation. For other systems mentioned in the last chapter, BDDBDDB [WACL05] does not support multicore computation; BigDatalog [SYI+16] suffers from noticeable memory inefficiency and scheduling overhead inherited from its backbone system Spark [ZXW+16], leading to observed less competitive performance on a single-node multi-core machine, and does not support mutual recursion; Graspan [WHZ+17] is a disk-based system that is much slower than other systems; thus we exclude these two systems in our case studies.

We use the profiling functionalities embedded in RecStep [FZZ+18], which are able to provide general profiling information of different systems on the Datalog workload evaluation that is not tied to a specific system. Although the profile components presented are fairly simple, they already give meaningful insights that can aid further system analysis and improvement, which is not possible by looking solely at the performance numbers.

4.1 Recursive Computation Profiling

Most existing systems evaluate Datalog programs using a type of bottom-up evaluation called *semi-naïve evaluation* (SN) [?] 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 4.2, the sizes of these three different types of facts in different iterations serve as the primary *fingerprint* of the Datalog workload and help to 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 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 $GF_{size} \geq UGF_{size} \geq NF_{size}$. Intuitively, a large gap between GF_{size} and UGF_{size} indicates

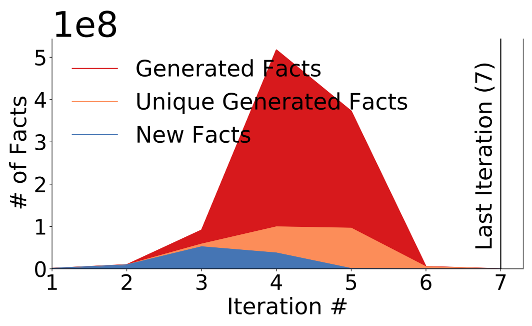


Figure 4.1: Recursion Profile of TC-`G10k`

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 4.1 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 4.2.

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 [FZZ+18, RB19, SJSW16, YSZ17] 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 Usage Many recent works focus on building in-memory Datalog systems [**?**,RB19, SJSW16, SYI+16]. However, most of them either ignore the evaluation of memory utilization [SYI+16] or miss the comparison with other existing Datalog systems [SJSW16, RB19]. Since most of the evaluations presented in these works are standalone (i.e., a system only evaluates one workload at a time on a server without interference), the lack of understanding of the memory footprint makes it hard to choose the appropriate 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).

4.2 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 [Clo18], 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. 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. The information of the input and output is summarized in Table 4.1.

Simple Linear Recursion

Transitive Closure (**TC**):

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

Same Generation **(SG)**:

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

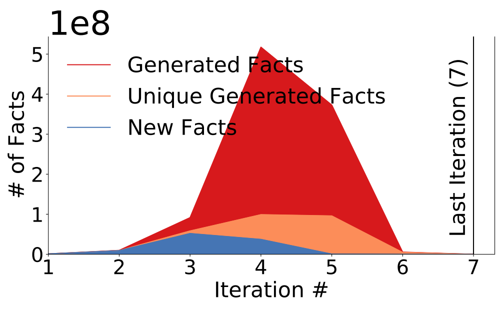
Figure 4.2 and Figure 4.3 show the recursive computation profiles of TC and SG of three systems on `G10k` and `G5k` respectively. The `G10k` and `G5k` datasets are random graphs of $10k$ ($\sim$ $100k$ edges) and $5k$ ($\sim$ $25k$ edges) vertices generated based on the Erdős-Rényi model [BM06], in which 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 4.2a and Figure 4.3a 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

Program	**Dataset**	**EDB Tuple #**	**IDB Tuple #**	**Reference**
SG	`G5k`	arc: 9.98e4	tc: 1.00e8	[FZZ+18]
TC	`G10k`	arc: 2.50e4	sg: 2.47e7	[FZZ+18]
REACH	`livejournal`	arc: 6.90e7 id: 100	reach: 4.40e6	[FZZ+18]
	`orkut`	arc: 1.17e8 id: 100	reach: 2.90e6	[FZZ+18]
	`arabic`	arc: 6.40e8 id: 100	reach: 2.62e6	[FZZ+18]
	`twitter`	arc: 1.47e9 id: 100	reach: 2.24e7	[FZZ+18]
AA	D7	assign: 1.00e7 load: 3.30e7 store: 2.10e7 addressOf: 4.00e6	pointsTo: 5.30e6	[FZZ+18]
CSPA	linux	assign: 1.98e6 dereference: 7.50e6	valueFlow: 5.50e6 valueAlias: 3.09e7 memoryAlias: 1.37e7	[FZZ+18]
	postgresql	assign: 1.20e6 dereference: 3.46e6	valueFlow: 3.71e6 valueAlias: 2.23e8 memoryAlias: 8.94e7	[FZZ+18]
	httpd	assign: 3.62e5 dereference: 1.14e6	valueFlow: 1.36e6 valueAlias: 2.34e8 memoryAlias: 8.89e7	[FZZ+18]
CSDA	linux	arc: 4.34e7 nullEdge: 5.89e5	null: 5.57e7	[FZZ+18]
	postgresql	arc: 3.45e7 nullEdge: 2.17e5	null: 2.15e7	[FZZ+18]
	httpd	arc: 9.90e6 nullEdge: 1.38e5	null: 9.39e6	[FZZ+18]

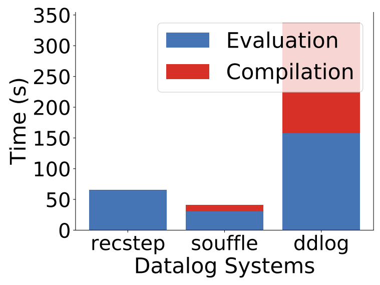
Table 4.1: Summary of EDB/Input and IDB/Output in Different Workloads

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.

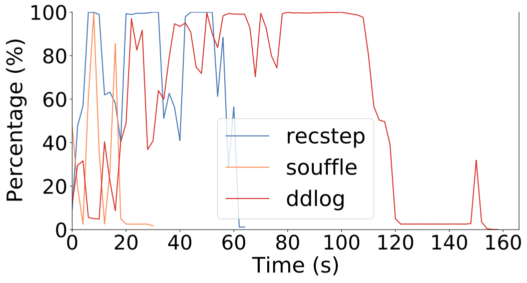
For TC and SG, while all three systems have relatively high CPU utilization throughout the evaluation (Figures 4.2c, 4.3c), the runtime varies. Besides the relatively long compilation time of DDlog ($\sim 200s$), DDlog's evaluation time is about $2-3X$ longer than that



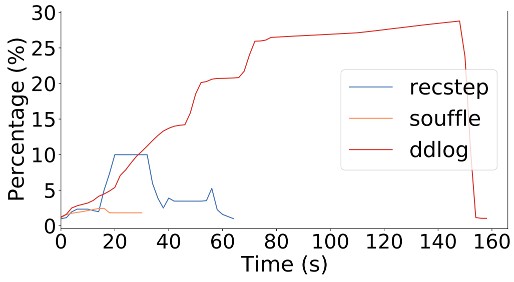
(a) Recursion Profile



(b) Runtime



(c) CPU Utilization

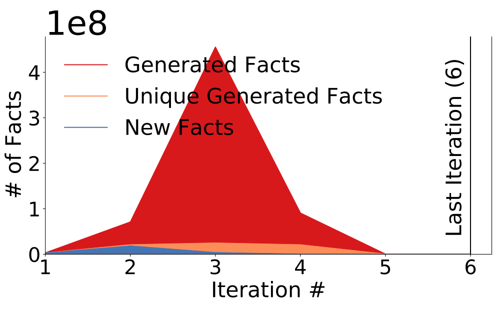


(d) Memory Usage

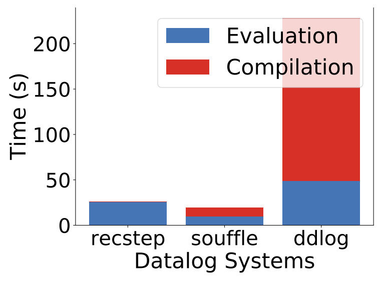
Figure 4.2: Transitive Closure on `G10k`

of the other two competitors (i.e., Souffle and RecStep), using a large amount of memory (Figures 4.2c, 4.3c). The performance number of DDlog is significantly worse than that of its runtime primitive Differential Dataflow [MMII13] as reported in [MLSR18]. 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), trading off the performance for batch processing.

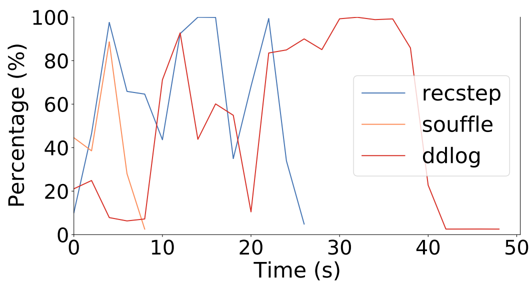
RecStep uses significantly more memory (Figures 4.2d, 4.3d) on the small input datasets (i.e., `G10k` and `G5k`) compared to other workloads (Figures 4.5d-4.14d), the sizes of input datasets of which vary from 22MB to 1.7GB. This is because RecStep performs *deduplication* as a separate step, relying on its backend in-memory relational database QuickStep [PDZ+18], 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 where *large duplicated results* are observed, RecStep quickly 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 with small vertices. Although PBME [FZZ+18] is an efficient technique to address this issue, it is specifically designed for graphs with a relatively small number of vertices, and its generality



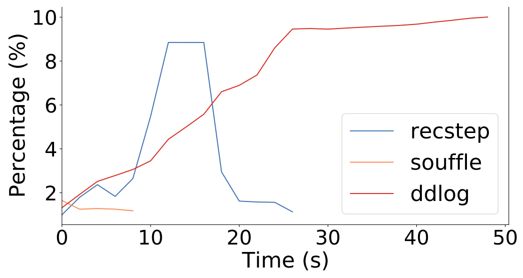
(a) Recursion Profile



(b) Runtime



(c) CPU Utilization



(d) Memory Usage

Figure 4.3: Same Generation on `G5k`

is limited.

In contrast, Souffle has acceptable overhead from code generation and compilation ($\sim 10s$), showing the overall best performance on TC-`G10k` and SG-`G5k` with a small memory footprint mainly due to its specialized parallel data structure Brie [JSZS19] for relation storage and indexing, which provides good compression capability for high-density relations with a large data volume and efficient parallel operations (e.g., insertion, lookup).

Reachability (**REACH**):

$$\texttt{reach}(\texttt{Y}) \coloneq \texttt{id}(\texttt{Y}).$$

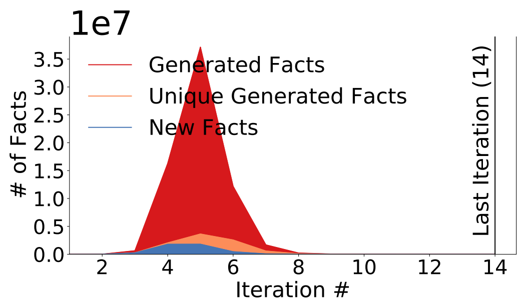
$$\texttt{reach}(\texttt{Y}) \coloneq \texttt{reach}(\texttt{X}), \texttt{arc}(\texttt{X}, \texttt{Y}).$$

Context-Sensitive Dataflow Analysis (**CSDA**):

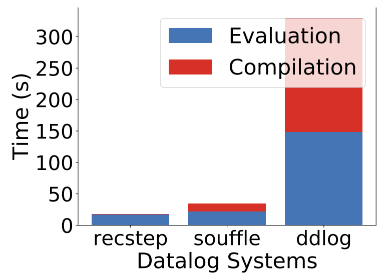
$$\texttt{null}(\texttt{X}, \texttt{Y}) \coloneq \texttt{nullEdge}(\texttt{X}, \texttt{Y}).$$

$$\texttt{null}(\texttt{X}, \texttt{Y}) \coloneq \texttt{null}(\texttt{X}, \texttt{W}), \texttt{arc}(\texttt{W}, \texttt{Y}).$$

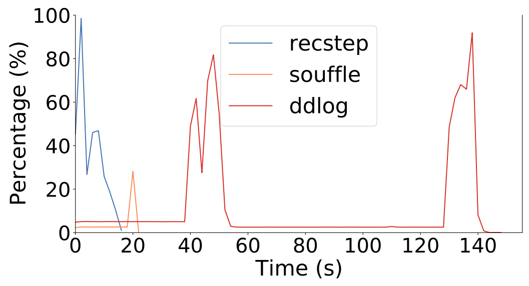
We run REACH on `livejournal`, `orkut`, `arabic` and `twitter`, four relatively large real-world social network datasets in which the friendship of users is represented as edges. REACH finds friends of a given set of user ids. Figure 4.5 is the recursive computation profile of



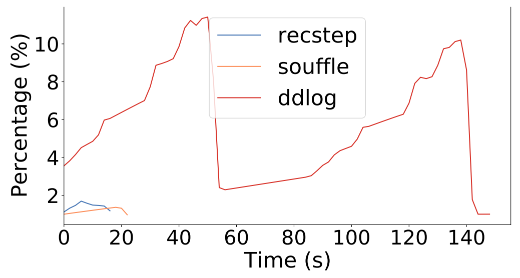
(a) Recursion Profile



(b) Runtime

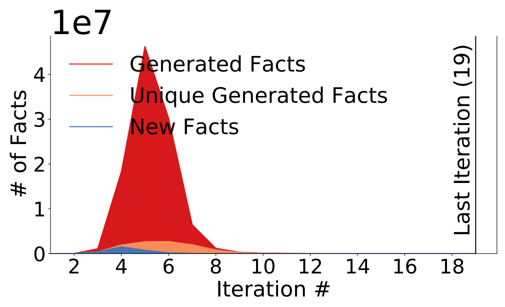


(c) CPU Utilization

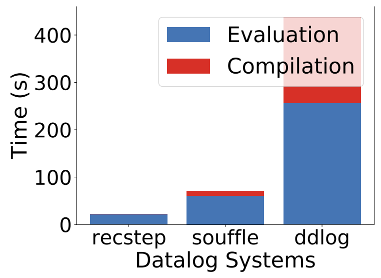


(d) Memory Usage

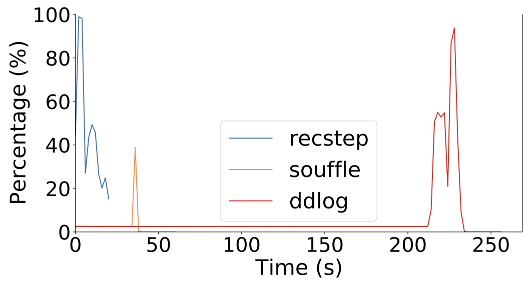
Figure 4.4: Reach on `livejournal`



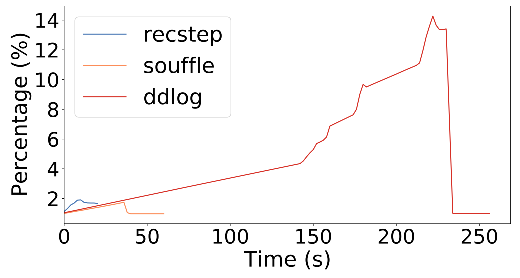
(a) Recursion Profile



(b) Runtime



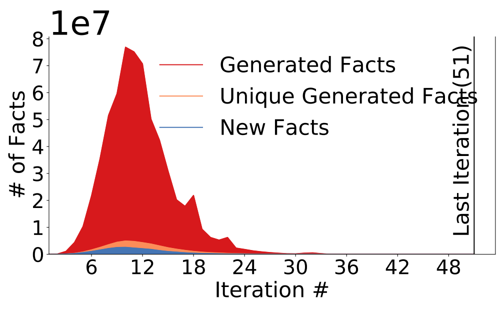
(c) CPU Utilization



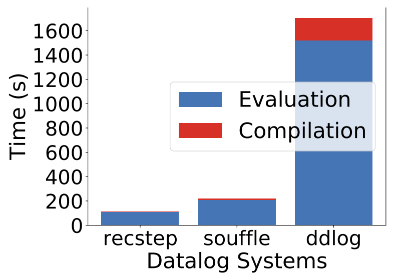
(d) Memory Usage

Figure 4.5: Reach on `orkut`

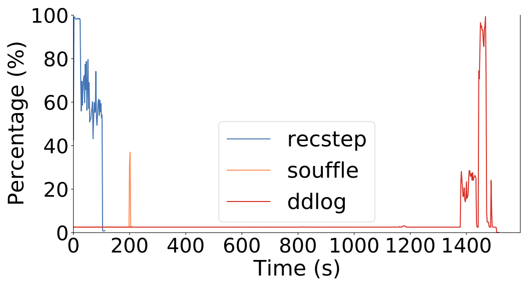
REACH-`orkut` and we observe that the relative performance of the system looks quite different from what is observed on TC-`G10k` and SG-`G5k`. RecStep significantly outperforms



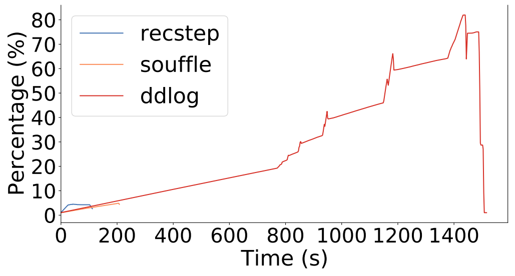
(a) Recursion Profile



(b) Runtime

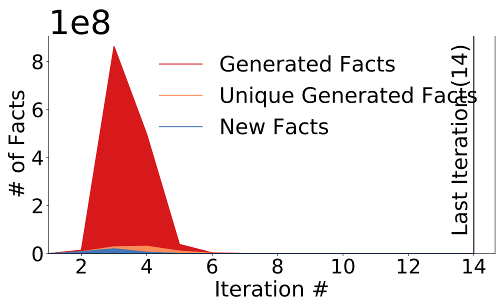


(c) CPU Utilization

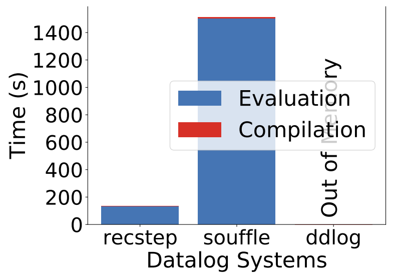


(d) Memory Usage

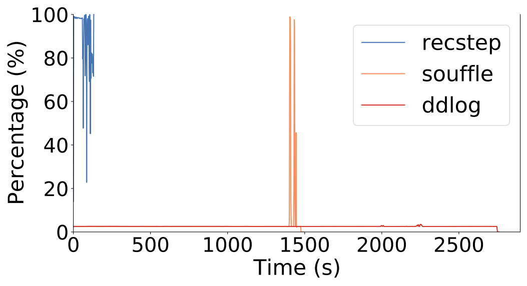
Figure 4.6: Reach on `arabic`



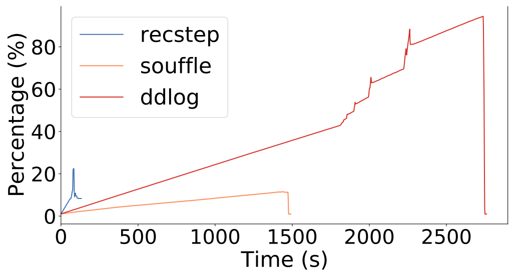
(a) Recursion Profile



(b) Runtime



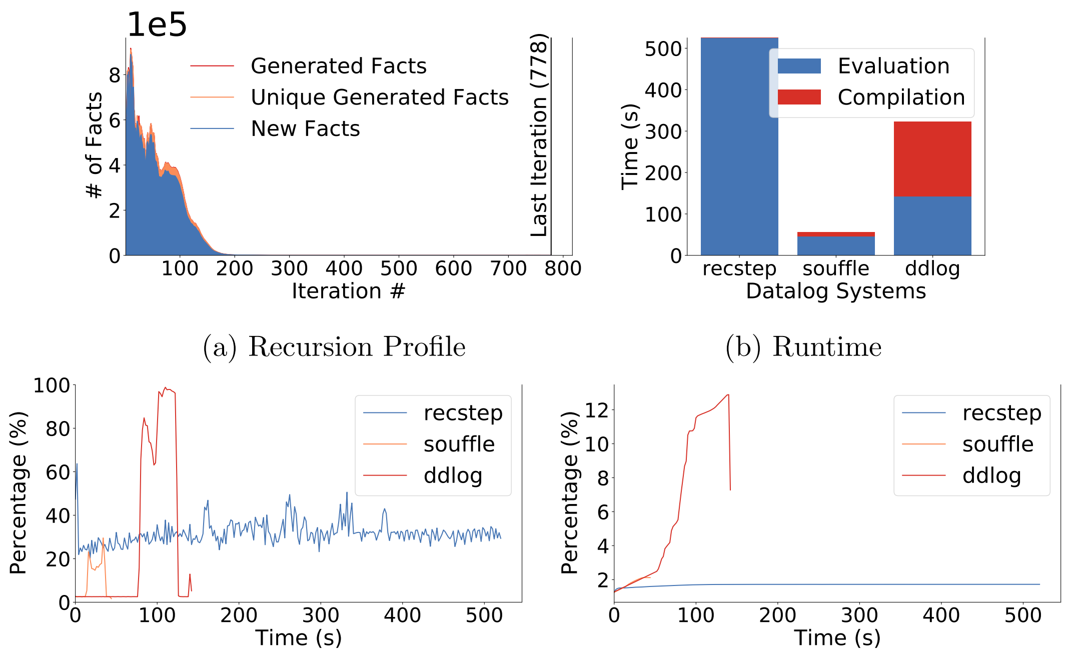
(c) CPU Utilization



(d) Memory Usage

Figure 4.7: Reach on `twitter`

Souffle and DDlog and the reasons are two-fold: (i) GF_{size} is relatively small across a relatively small 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. The recursive profiles of REACH-`livejournal`, REACH-`arabic` and REACH-`twitter` and the relative performance of the system are similar to that of REACH-`orkut`. It should be noted that the warm-up time of Souffle and DDlog *increases as the size of the input increases.* DDlog runs out of memory on `livejournal` before starting the evaluation.



(a) Recursion Profile

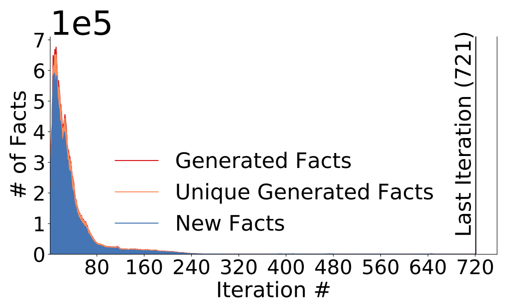
(b) Runtime

(c) CPU Utilization

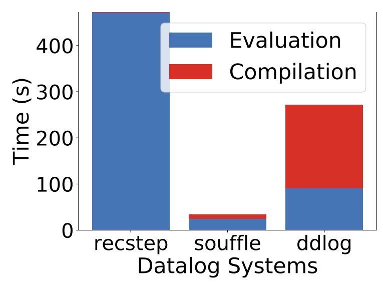
(d) Memory Usage

Figure 4.8: Context-Sensitive Dataflow Analysis Linux

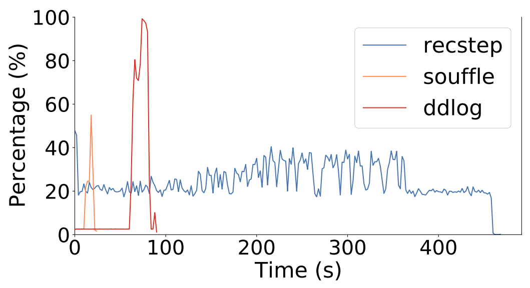
CSDA can be seen as a variant of transitive closure: first a set of *null edges* is given to initialize the non-recursive rule, and then the linear-recursive rule looks the same as that of TC. However, the recursive computation profiles of the CSDA and TC workloads are very different. We observe that RecStep performs significantly worse compared to Souffle and DDlog (Figures 4.8b, 4.9b, 4.10b) while its CPU utilization over time is lower compared to Souffle and DDlog (Figures 4.8c, 4.9c, 4.10c). 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 4.8a, 4.9a, 4.10a): 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 the hash tables and repeatedly



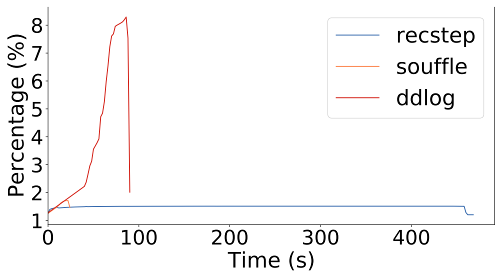
(a) Recursion Profile



(b) Runtime

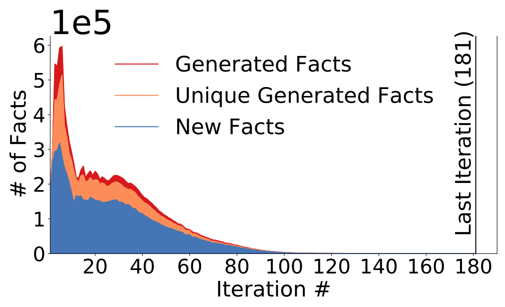


(c) CPU Utilization

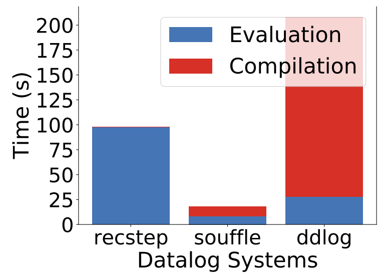


(d) Memory Usage

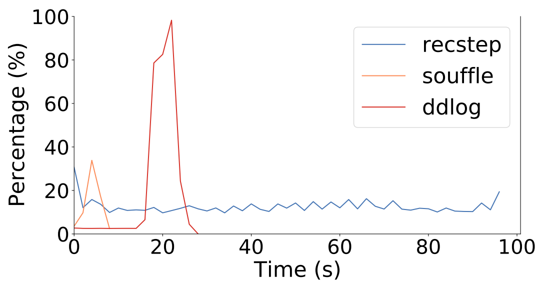
Figure 4.9: Context-Sensitive Dataflow Analysis Postgresql



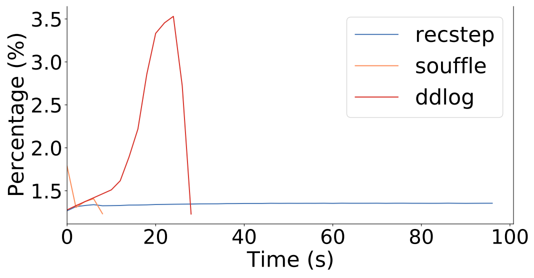
(a) Recursion Profile



(b) Runtime



(c) CPU Utilization



(d) Memory Usage

Figure 4.10: Context-Sensitive Dataflow Analysis Httpd

scan the base table (for probing) for the processing of the joins in every iteration regardless of the size of the joining table. 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 execution of the recursive Datalog program in these cases.

Non-linear Recursion

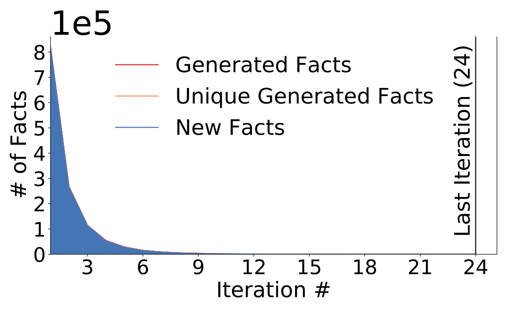
Andersen's Analysis (**AA**):

```
pointsTo(Y, X) :- addressOf(Y, X).
pointsTo(Y, X) :- assign(Y, Z), pointsTo(Z, X).
pointsTo(Y, W) :- load(Y, X), pointsTo(X, Z), pointsTo(Z, W).
pointsTo(Z, W) :- store(Y, X), pointsTo(Y, Z), pointsTo(X, W).
```

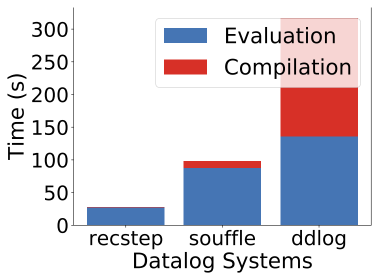
Context-Sensitive Points-To Analysis (**CSPA**):

```
  valueFlow(Y, X) :- assign(Y, X).
  valueFlow(X, X) :- assign(X, Y).
  valueFlow(X, X) :- assign(Y, X).
  valueFlow(X, Y) :- assign(X, Z), memoryAlias(Z, Y).
  valueFlow(X, Y) :- valueFlow(X, Z), valueFlow(Z, Y).
 valueAlias(X, Y) :- valueFlow(Z, X), valueFlow(Z, Y).
 valueAlias(X, Y) :- valueFlow(Z, X), memoryAlias(Z, W), valueFlow(W, Y).
memoryAlias(X, X) :- assign(Y, X).
memoryAlias(X, X) :- assign(X, Y).
memoryAlias(X, W) :- dereference(Y, X), valueAlias(Y, Z), dereference(Z, W).
```

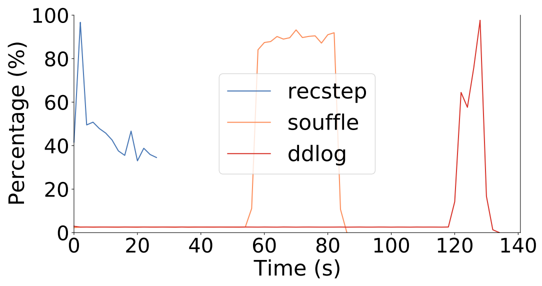
Switching from linear recursion to non-linear recursion, we observe that RecStep significantly outperforms Souffle and DDlog (Figure 4.11b) for Andersen's Analysis evaluated on the largest input dataset ($\sim$ 1.2G) used in [FZZ^{+}18]. Figure 4.11c shows that both Souffle and DDlog have a long *warm-up time* in which the CPU utilization is very low. One possible reason could be that there are four EDB/input relations in AA, which distinguishes AA from other Datalog programs studied here and could possibly result in more preprocessing work for Souffle and DDlog (e.g., index construction) - similar to what is observed on REACH workloads.



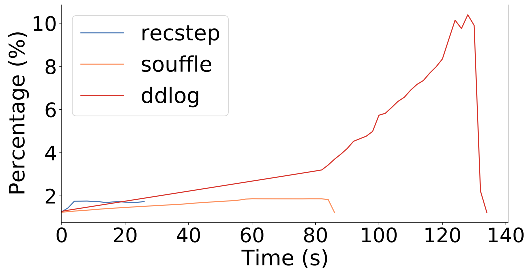
(a) Recursion Profile



(b) Runtime



(c) CPU Utilization

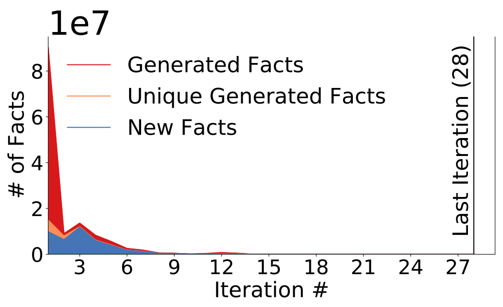


(d) Memory Usage

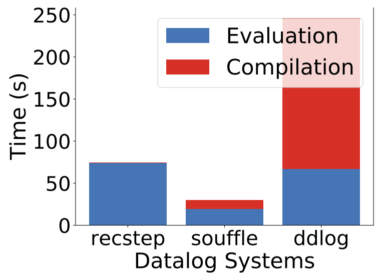
Figure 4.11: Andersen's Analysis

Since the GF$_{size}$ is relatively small across different iterations, RecStep evaluates AA efficiently while using only a small amount of memory (Figure 4.11d). Additionally, AA has *nonlinear-recursive* rules, all of which 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.

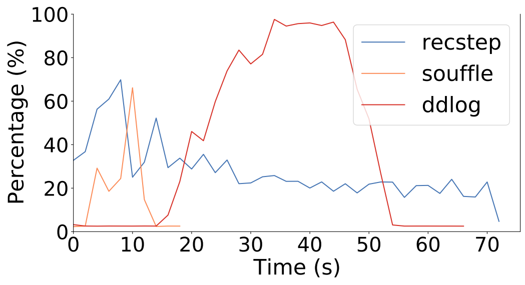
The Datalog program itself alone is insufficient to characterize the recursive computation. For CSPA, the recursive computation profiles of three systems on `linux` dataset look very different from the ones on `postgresql` and `httpd` datasets. Interestingly, we can see that similar recursive profiles (Figure 4.13a and Figure 4.14a) come along with similar profiling information on runtime (Figure 4.13b and Figure 4.14b), CPU (Figure 4.13c and Figure 4.14c) and memory (Figure 4.13d and Figure 4.14d). DDlog's performance seems to be very sensitive to the sizes of the intermediate results: when there are a large number of facts being generated in several iterations, DDlog turns out to require a great amount of memory to maintain the intermediate states (Figure 4.13d and Figure 4.14d) and the corresponding overhead also affects the overall performance greatly (i.e., DDlog is outperformed by RecStep and Souffle as shown in Figure 4.13b and Figure 4.14b). Such inference can be further strengthened by looking at Figure 4.12, in which Figure 4.12a shows that fewer facts are generated during the iterative evaluation and DDlog shows a better relative performance over RecStep in Figure 4.12b while using considerably less memory as shown in Figure 4.12d.



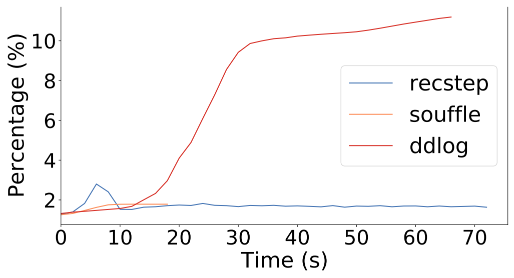
(a) Recursion Profile



(b) Runtime

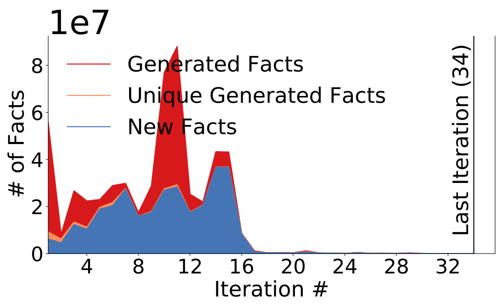


(c) CPU Utilization

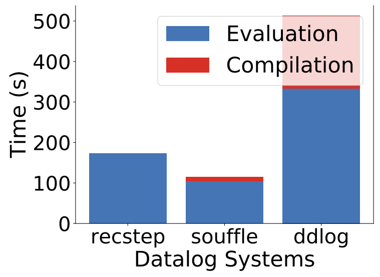


(d) Memory Usage

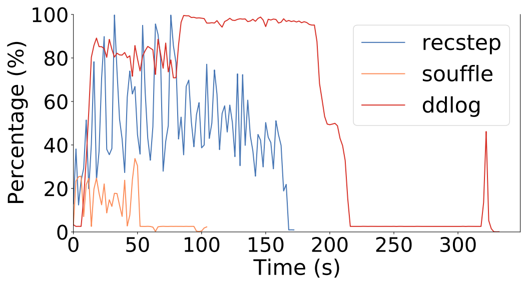
Figure 4.12: Context-Sensitive Points-to Analysis Linux



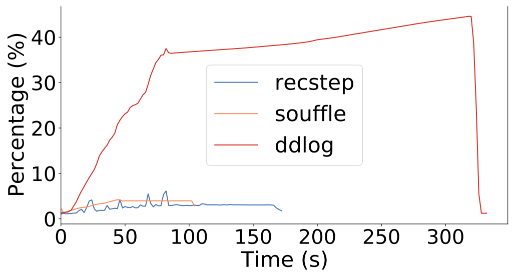
(a) Recursion Profile



(b) Runtime

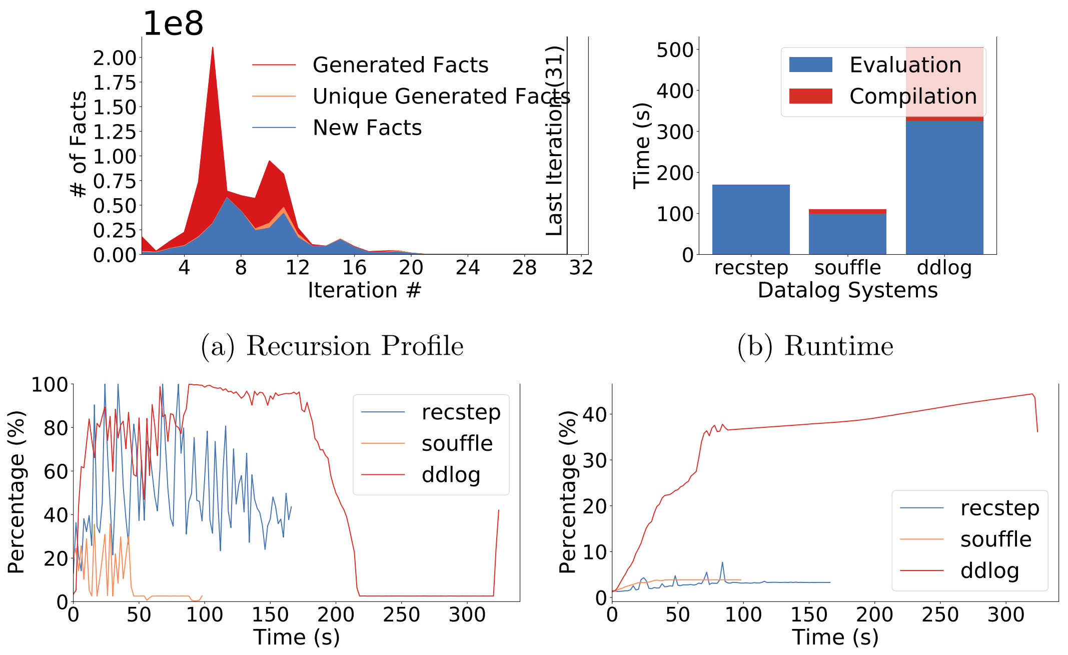


(c) CPU Utilization



(d) Memory Usage

Figure 4.13: Context-Sensitive Points-to Analysis Postgresql



(a) Recursion Profile

(b) Runtime

(c) CPU Utilization

(d) Memory Usage

Figure 4.14: Context-Sensitive Points-to Analysis Httpd

4.3 Summary

Recently, we have observed 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. The recursive computation profiling components we present in this paper are the first step toward this goal. Also, a standard benchmark is needed that covers different aspects of Datalog workloads. This benchmark should be in analogy to the online analytical processing benchmarks [BBF15] that have been used for more than two decades for the 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).

Most of the contents in this chapter is from our paper titled *Towards Better Understanding of the Performance and Design of Datalog Systems* appeared in Datalog 2.0 2022 [FMK22]. Next In Chapter 5, we introduce the new high-performance Datalog system that we are building based on recursive computation profiling. We will discuss how we design and implement the new system based on the lessons learned in Chapter 3 and this chapter. Before that, in the next chapter, we show how we can leverage the succinct syntax of Datalog and its expressive power to concisely design, analyze and express new, better algorithms of a specific application *consistent query answering* (CQA), the corresponding Datalog program of which can be efficiently evaluated by our high-performance Datalog system to largely surpass the state-of-the-art performance numbers.

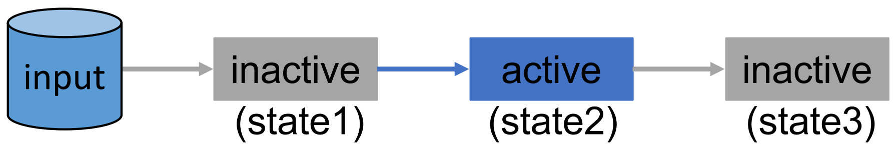
Chapter 5

FlowLog: Asynchronous Datalog Evaluation

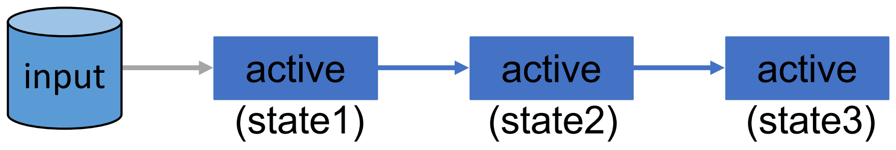
In this chapter, we introduce FlowLog, an asynchronous dataflow-based Datalog system we are building that aims to address the inefficiency observed in RecStep. Furthermore, FlowLog is designed and built to be easily maintained and expandable by using *differential datalfow* [MMII13], a computational framework designed to perform computations efficiently on large amounts of data and maintain computations as the data change (i.e., incremental computation). Differential dataflow is built on top of *timely dataflow* that is introduced in [MMI$^+$13], a system for implementing distributed streaming computation that allows one to organize computation in general. Compared to the best performers discussed in Chapter 3, namely RecStep [FZZ$^+$18] and Souffle [SJSW16], FlowLog is generally faster than both systems by 2-10X, proving its superior efficiency in evaluating the workload in which each computation stage performs a small amount of work and there are opportunities for index sharing, two characteristics that are commonly observed in real-world applications.

State-of-the-art Datalog systems such as RecStep and Souffle have shown superior performance in evaluating Datalog workloads for application domains such as program analysis and graph analytics. However, both RecStep and Souffle are *batch processing* systems, in which synchronization between different computation stages is required, as illustrated in Figure 5.1a. Such synchronization could happen at both *macro-level* (e.g., different Datalog rules) and *micro-level* (e.g., different operators). For recursive computation, iterations make use of synchronization in batch processing systems at the cost of latency. This latency could significantly affect the efficiency of the Datalog evaluation of the program for some workloads, as discussed in Chapter 4.

To address the performance issue observed in batch processing Datalog systems, we rethink the way how Datalog program is evaluated and explore *asynchronous dataflow-based* Datalog evaluation, implementing a system prototype named FlowLog. FlowLog exploits the asynchronous dataflow model, as well as the incremental computational capability in different iterations of recursive computation. Furthermore, FlowLog allows for sharing the indexed operator state within the same Datalog rule and among different Datalog rules at



(a) Batch Processing



(b) Asynchronous Dataflow Processing

Figure 5.1: Figure 5.1a depicts how *batch processing* works: only one state (i.e., state2) is active at a time while other states are inactive either due to the task completion (i.e., state1) or waiting for the active state to finish the current task (i.e., state3). In contrast, *every state is active* in *asynchronous dataflow processing* (Figure 5.1b). Considering the context of Datalog program evaluation, each state here could map to a Datalog rule, an iteration in recursive rule evaluation or an single operation (e.g., join, deduplication, set-difference).

very low cost, which leads to approximately additional 2X speedup in some of the workloads we have evaluated.

Our Contribution. In summary, we make the following contributions:

1. We point out the inefficiency observed in batch processing Datalog systems such as Souffle and RecStep, analyzing the root causes behind, and rethinking the evaluation of Datalog programs.

2. We present FlowLog, an asynchronous dataflow-based Datalog system for batch processing implemented on top of differential dataflow [MMII13]. FlowLog supports both linear and nonlinear recursion (including mutual recursion). Negation and recursive aggregation are allowed and performed in an efficient incremental manner.

3. We consider the indexing-sharing technique adopted from the advancements of relational database management systems and streaming processing.

4. By evaluating FlowLog against RecStep and Souffle on varying Datalog workloads in Chapter 3 and Chapter 4, we demonstrate the efficiency of our rethinking of the evaluation of Datalog.

Organization. In Section 5.1, we briefly introduce *differential dataflow*, the backbone system used to build FlowLog. We then give a summary of the architecture design of FlowLog in Section 5.2, after which we discuss the optimizations imposed on FlowLog that lead to its high performance for Datalog evaluation in Section 5.3. Section 5.4 presents the results of the experimental evaluation and we summarize this Chapter in Section 5.5.

5.1 Differential Dataflow

Differential dataflow [MMII13] is a computational framework built on top of *timely dataflow*, a model designed for the execution of data-parallel data flows that is used and introduced by Naiad [MMI+13]. Timely dataflow allows for *intra-operator parallelism* by sharding each operator across all workers (i.e. threads in a single node or different machines in a distributed setting). Data exchange happens between different workers when needed, and each data carries a logical timestamp that is used to reason correctness and progress tracking. In addition to the efficiency provided by the timely dataflow, the differential dataflow operators provide the ability for *incremental computation* when the input changes.

One main characteristic that distinguishes differential dataflow (DD) from other frameworks that support traditional incremental computation is that DD allows arbitrarily nested iteration. DD varies the state of computation based on a *partially ordered* set of versions and retains and indexes the updates required to reconstruct the states of different versions.

5.2 Architecture

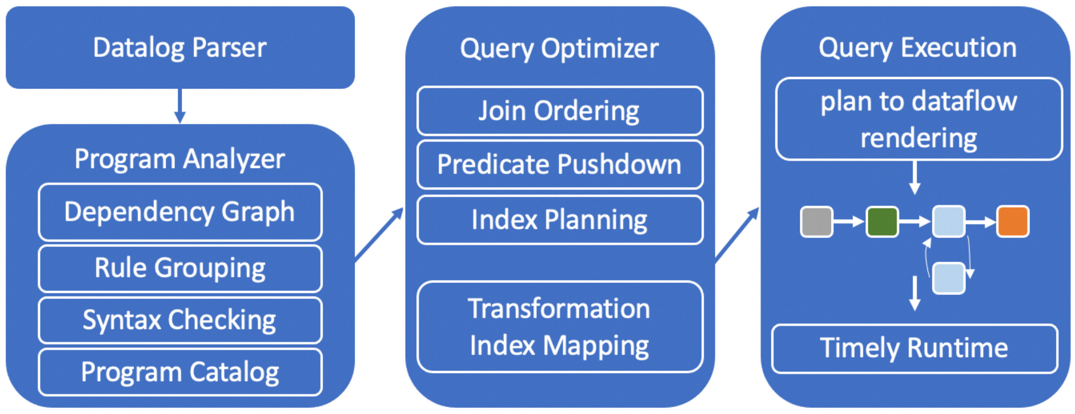


Figure 5.2: Architecture overview of FlowLog

In this section, we present and briefly discuss the architecture of FlowLog. FlowLog is built on top of the differential dataflow [Mcs22a] API and the timely dataflow [Mcs22b] runtime. Timely dataflow is an asynchronous dataflow model that allows for cyclic dataflow in the dataflow graph, providing natural support for asynchronous iterative computation.

Differential dataflow is a computational framework that is built on top of timely dataflow, in which operators (e.g., map, join, filter, etc) are able to incrementally maintain the computations as the data change. Low-level details, such as moving data around and parallelizing the computation, are fully handled by the timely dataflow runtime so that we can focus on the Datalog evaluation strategy itself. Compared to RecStep that lets RDBMS figure out the query plan for generated SQL statements from the rules Datalog, in FlowLog we have full flexibility to plan the execution of the rule, such as making the decision of the order of operators being placed in the dataflow graph (e.g., query plan) and deciding the data arrangement (e.g., indexing).

Overview. The overall architecture of FlowLog is shown in Figure 5.2. Starting from the *parser*, the input Datalog program is validated, parsed, and the rules and relations are stored in the proper data structures. The *program analyzer* then analyzes the Datalog rules, constructs the rule evaluation dependency graph (for stratification), groups rules belonging in the same connected component of the dependency graph, checks the correctness of syntax, and processes and stores the attribute mapping information (e.g., joining attributes, constant constraints, negation, etc) in the program catalog.

The *query optimizer* then takes the program catalog to construct the query plan for each rule. FlowLog exploits the rule-based approach for query optimization, which has been adopted in RDBMS such as QuickStep [PDZ+18]. Predicate pushdown is performed to eliminate redundant data as early as possible during dataflow processing. The joining order is searched so that the potential Cartesian product is avoided, and indexes of minimal number needed to be built and could be shared by different operations. Negation is also considered for predicate pushdown and is performed as early as possible. The output query plan is an ordered list of transformation mappings. Each transformation mapping consists of the input and output logic collections (as shown in Figure 5.3a) along with the corresponding transformation indexes (as shown in Figure 5.3b).

Taken the transformation mappings from the query optimizer, the *query execution engine* renders each mapping into the corresponding operator and logic, constructing the *dataflow graph* that can be executed by the timely dataflow runtime, which then reads the input data and drives the computation.

5.3 Optimizations

Thinking of Datalog evaluation as asynchronous dataflow execution itself brings great performance benefits by avoiding the expensive synchronization cost between different computational stages and better utilizing the multicore computation resource. Furthermore, leveraging the incremental computation capability of differential dataflow, FlowLog gets

```
pub enum Transformation {
    RowToRow {
        input: Collection,
        output: Collection,
        indexes: TransformationLocalIndex,
    },
    RowToKeyValue {
        input: Collection,
        output: Collection,
        indexes: TransformationLocalIndex,
    },
    KeyValueToKeyValue {
        input: Collection,
        output: Collection,
        indexes: TransformationLocalIndex,
    },
    Join {
        input: (Collection, Collection),
        output: Collection,
        indexes: TransformationLocalIndex,
    },
    AntiJoin {
        input: (Collection, Collection),
        output: Collection,
    },
    Projection {
        input: Collection,
        output: Collection,
        indexes: TransformationLocalIndex,
    },
    Filter {
        input: Collection,
        output: Collection,
        indexes: TransformationLocalIndex,
    },
    Dummy {
        input: Collection,
    },
}
```

(a) Transformation

```
pub enum TransformationLocalIndex {
    // Row(usize)
    MultiConstraintsFiltering {
        constant_equality_constraints_indexes: Vec<(CollectionSignatureIndex,
            Constant)>,
        attribute_equality_constraints_indexes:
            Vec<(CollectionSignatureIndex, CollectionSignatureIndex)>,
        inequality_constant_constraints_indexes:
            Vec<(CollectionSignatureIndex, ComparisonOperator, Constant)>,
        inequality_variable_constraints_indexes: Vec<(
            CollectionSignatureIndex,
            ComparisonOperator,
            CollectionSignatureIndex,
        )>,
    },
    EmptyRowFiltering,
    // Row(usize)
    RowToRow(Vec<CollectionSignatureIndex>),
    //  Row(usize)
    RowToKeyValue {
        key: Vec<CollectionSignatureIndex>,
        value: Vec<CollectionSignatureIndex>,
    },
    // KeyValue((usize, usize))
    KeyValueToKeyValue {
        key: Vec<CollectionSignatureIndex>,
        value: Vec<CollectionSignatureIndex>,
    },
    // CollectionKeyValue((usize, usize, usize))
    Join {
        join_output_indexes: Vec<CollectionSignatureIndex>,
        inequality_variable_constraints_indexes: Vec<(
            CollectionSignatureIndex,
            ComparisonOperator,
            CollectionSignatureIndex,
        )>,
    },
    // Row(usize), KeyValue((usize, usize))
    Projection(Vec<CollectionSignatureIndex>),
}
```

(b) Transformation Index

Figure 5.3: FlowLog's code snippet for transformation mapping

semi-naïve evaluation for free without performing different steps explicitly (e.g., deduplication and set-difference calculation) during the iterative computation, since the new facts generated in each iteration are automatically maintained (i.e., as differences) and used for incremental computation during the next iteration. For recursive aggregation, properly *abusing* the monotone property, redundant computation can be avoided. Optimizations such as predicate pushdown and join ordering mentioned in Section 5.2 are commonly exploited in RDBMS, we next discuss the index-sharing and related optimizations we imposed on FlowLog in detail.

Andersen's Analysis:

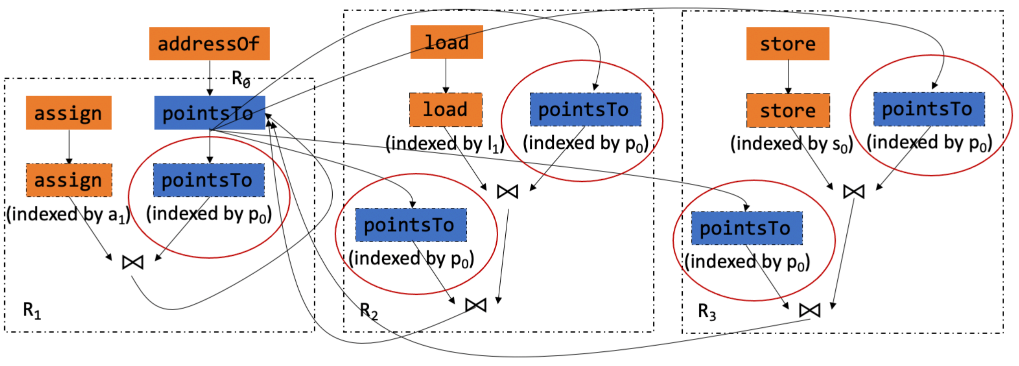
R_1 : `pointsTo(Y, X) :- addressOf(Y, X).`

R_2 : `pointsTo(Y, X) :- assign(Y, Z), pointsTo(Z, X).`

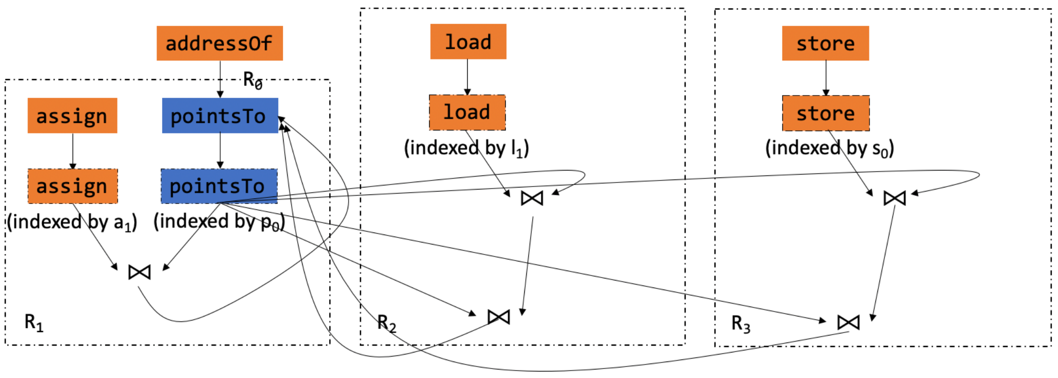
R_3 : `pointsTo(Y, W) :- load(Y, X), pointsTo(X, Z), pointsTo(Z, W).`

R_4 : `pointsTo(Z, W) :- store(Y, X), pointsTo(Y, Z), pointsTo(X, W).`

Index Planning & Sharing When performing operations such as joining two relations in differential dataflow, *arrangements* needed to be built firstly for both relations, in which the data of each relation are *arranged* by the joining attributes (i.e., key). Then efficient access to different data groups indexed by different keys (e.g., point lookup) can be achieved. Such arrangements or indexes are *continuously maintained* throughout the dataflow execution.



(a) No Index Sharing



(b) Index Sharing

Figure 5.4: Dataflow - No Index Sharing v.s Index Sharing on Andersen's Analysis

However, redundant computation can occur in cases where multiple operators (e.g., joins) in the dataflow graph build the same arrangements without actually sharing. Such a non-wareness of sharing opportunities could lead to repeated work (i.e., waste of CPU) and extra space (i.e., waste of memory). Shared arrangement has been recently introduced and described in [MLSR18], mainly advocating its efficiency in the streaming processing scenario where new queries may come and are able to reuse the indexed states of existing running queries. We exploit shared arrangement in FlowLog, in which the indexed state could be shared by subgraphs corresponding to different Datalog rules where there are sharing opportunities. The arrangement can even be shared within the same rule, where the same relation appears multiple times and joins with other relations on the same attributes. Thus, when considering the joining ordering of a Datalog rule, a joining order is selected such that the maximal sharing is able to be achieved.

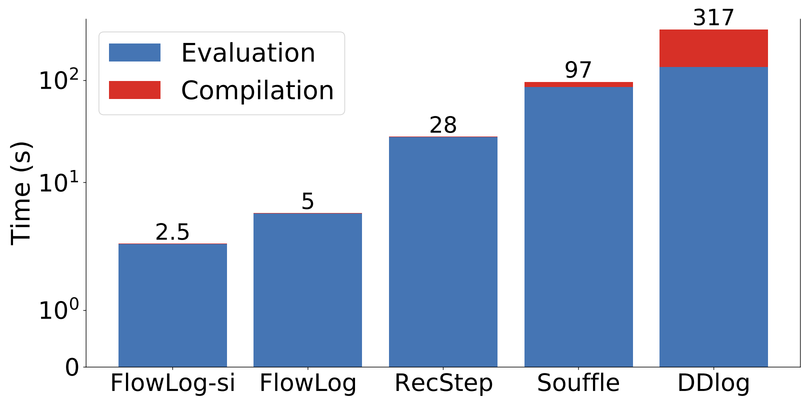


Figure 5.5: Performance of different systems evaluating **Andersen's analysis**

The comparison between the simplified dataflow graphs with and without arrangement sharing of `pointsTo` is presented in Figure 5.4. Without arrangement sharing, five arragements on `pointsTo` are built and maintained separately (Figure 5.4a while only one arrangement is needed and shared between different rules (i.e., R_2, R_3, R_4) and inside the same rule (i.e., $R3$ and R_4), as depicted in Figure 5.4b. Figure 5.5 shows the performance comparison between different systems on Andersen's analysis, a program analysis task that is discussed in both Chapter 3 and Chapter 4. We observe that FlowLog without index sharing between different operators already significantly outperforms other systems by 6-50X, including RecStep, the best performer of Andersen's analysis in the performance evaluation presented in Chapter 3 and Chapter 4. By sharing the indexes, FlowLog is able to gain another 2X speedup, as shown in Figure 5.5 (FlowLog-si).

Plan Optimization for Aggregation When considering a Datalog program that involves recursive aggregation, a more efficient data flow graph can be constructed for aggregation operators such as `MIN` and `MAX`. Since for `MIN` and `MAX`, *each group key* should have *only one value at a time*, it is guaranteed that there will be no duplicates in the output in differential dataflow, thus a separate deduplication step can be completely removed. Compared with using the general strategy to evaluate Datalog programs without aggregation, this strategy saves both memory (i.e. reduce the number of states that need to be maintained) and evaluation time (i.e. less work to do).

5.4 Experiments

In this section, we evaluate the performance of FlowLog, experimentally proving that by thinking of Datalog evaluation as asynchronous dataflow execution addresses the inefficiency observed in RecStep. Additionally, we aim to understand the existing limitation of FlowLog for iterative batch processing tasks, which can be addressed later. The Datalog workloads used for the evaluation are the same as those presented in Chapter 3 and Chapter 4.

System Configuration. We use the same system configuration as used in Chapter 4 (4.2). In summary, the experiments are conducted on a single-node server with 20 physical cores and 40 hyper-threading hardware threads. The total size of available memory is 160GB.

Other Datalog Systems. We compare the performance of FlowLog with the best-performing systems studied in Chapter 3 and Chapter 4, namely RecStep and Souffle. We note that the Souffle used for evaluation in Chapter 4 and this section is an improved version over the one studied in Chapter 3. Differential Datalog (DDlog), is included for comparison purposes since it is built on top of the same computational framework that is used by FlowLog, differential dataflow.

Benchmark Programs and Datasets. Since we are using the same benchmark programs and datasets as used in Chapter 4 and Chapter 3, here we omit the details and refer the readers who are interested to Section 3.4 and Section 4.2 of the previous chapters.

System	*livejournal*	*orkut*	*arabic*	*twitter*
FlowLog	**5s** (0s)	**8s** (0s)	**50s** (0s)	139s (0s)
RecStep	17s (0s)	21 (0s)	112 (0s)	**133** (0s)
Souffle	22 (10s)	60 (10s)	208 (10s)	1503 (10s)
ddlog	148 (181s)	256 (181s)	1522 (182s)	2760 (181s)

Table 5.1: Runtime (compile time) of different systems evaluating **REACH**

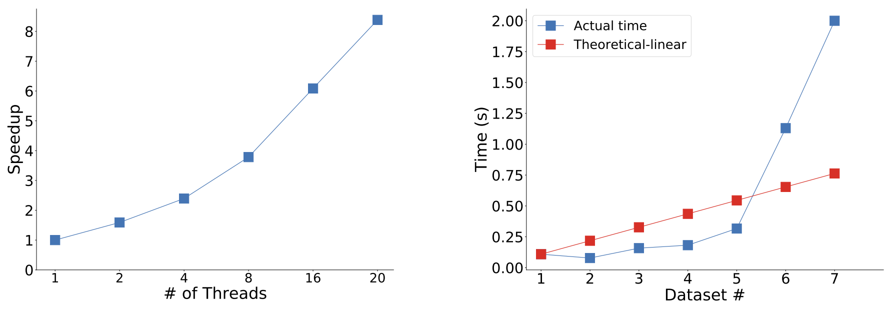
Scalability Figure 5.6 shows how FlowLog scales up in the cores (5.6a) on CSPA and how FlowLog scales up in the sizes of the data sets on AA (5.6). The trends of both are similar to those for RecStep presented in Chapter 3, however, with better performance numbers.

System	*livejournal*	*orkut*	*arabic*	*twitter*
FlowLog	**17s** (0s)	**25s** (0s)	**197s** (0s)	OOM
RecStep	39s (0s)	43s (0s)	421s (0s)	**501s** (0s)
DDlog	184s (210s)	273s (210s)	OOM	OOM

Table 5.2: Runtime (compile time) of different systems evaluating **CC**

System	*livejournal*	*orkut*	*arabic*	*twitter*
FlowLog	**3s (0s)**	**6s** (0s)	**29s** (0s)	**73s** (0s)
RecStep	14s (0s)	30s (0s)	113s (0s)	120s (0s)
DDlog	205 (193s)	244s (193s)	3088s (193s)	OOM

Table 5.3: Runtime (compile time) of different systems evaluating **SSSP**



(a) Scaling-up on Cores (CSPA)   (b) Scaling-up on Datasets (AA)

Figure 5.6: Scaling-up on Cores and Datasets

Iterative graph batch processing. We evaluate FlowLog on standard batch iterative graph analytics on four real-world social network graphs: *livejournal*, *orkut*, *arabic* and *twitter*. For reachability (REACH) and single-source-shortest-path (SSSP), instead of choosing one single vertex as the starting ID, we randomly sampled 100 vertices from each graph and then fed them as the starting points. Thus, now the task REACH is interpreted as *find all vertices that are reachable from the given 100 vertices* and SSSP is interpreted as *find the lengths of the shortest paths from the given 100 vertices to other vertices of the graph that are reachable.*

We observe that FlowLog outperforms other systems on REACH most of the time in Table 5.1. The only exception is *twitter*, on which RecStep slightly outperforms FlowLog. Looking at Figure 4.7a, we observe that the total number of iterations is small and there is a relatively large amount of work to be done (i.e., represented by the size of the generated facts). Thus, the overhead RecStep suffers from synchronization is negligible and is capable of fully using the CPU cores for efficient computation in such a case. Interestingly, when the size of the dataset increases , the relative performance difference between FlowLog and RecStep also becomes smaller, as a batch-processing system works well in processing large amounts of data in a small number of iterations when multicores can be exploited effectively.

Being able to perform the computation incrementally in each iteration of the recursive computation not only leads to *automatic semi-naïve* evaluation in the normal monotone Datalog program, but also helps avoid unnecessary work to be performed during *recursive aggregation* such as `MIN` and `MAX`. Taking the program Datalog that calculates connected components (CC) as an example, to evaluate the rule

$$\texttt{cc3}(\texttt{y}, \texttt{MIN}(\texttt{z})) \coloneq \texttt{cc3}(\texttt{x}, \texttt{z}), \texttt{arc}(\texttt{x}, \texttt{y}).$$

RecStep has to compare the newly propagated labels for each node y with all existing labels (i.e., other nodes belonging to the same connected components stored in `cc3`) that the node has seen so far, *adding* those strictly smaller than in $\{x|\texttt{cc3}(\texttt{y}, \texttt{x})\}$. In contrast, for each node y, FlowLog retains *only one value* in each iteration, which is only updated if a smaller label has been propagated to the node y. Thus, at each iteration, only one value for each y must be considered when comparing with the newly propagated labels. Since there is only one value for each node y (e.g., key at a time), a separate deduplication is not necessary. For CC FlowLog outperforms RecStep and DDlog on *livejournal*, *orkut*, and *arabic*. However, FlowLog fails at the *twitter* dataset due to an out-of-memory error. The cause behind the failure is that FlowLog maintains the arrangements of both the IDB relation `arc` and the output relation `cc3` including timestamps and count for each different data point, and the total size required surpassing the available memory. For SSSP, FlowLog does not encounter the issue since the IDB relation `sssp2` only needs to maintain a relatively small number of subgraph vertices that are reachable from the given vertices (that is, `id`).

System	*linux*	*postgresql*	*httpd*
FlowLog-si	**6s** (0s)	**31s** (0s)	**35s** (0s)
FlowLog	11s (0s)	52s (0s)	59s (0s)
RecStep	74s (0s)	173s (0s)	169s (0s)
Souffle	19s (10s)	105s (10s)	99s (10s)
DDlog	67s (179s)	332s (179s)	325s (179s)

Table 5.4: Runtime (compile time) of different systems evaluating **CSPA**

Program analysis. In Table 5.4 and Table 5.5, we report the performance numbers of different systems on two program analysis tasks, CSPA and CSDA, in which Souffle outperforms RecStep as observed in Chapter 4. For CSPA, FlowLog significantly outperforms Souffle on all datasets and shows further improvement by enabling index/arrangement

System	*linux*	*postgresql*	*httpd*
FlowLog	**25s (0s)**	**10s** (0s)	**2s** (0s)
RecStep	525 (0s)	472s (0s)	98s (0s)
Souffle	45s (10s)	25s (10s)	8s (10s)
DDlog	143s (180s)	91s (180s)	28s (180s)

Table 5.5: Runtime (compile time) of different systems evaluating **CSDA**

sharing (FlowLog-si). Using asynchronous data flow execution, FlowLog shows its superior performance on CSDA (Table 5.5) without feeling the pain of synchronization between a large number of iterations of its predecessor RecStep suffers.

5.5 Summary

In this chapter, we present our exploration of rethinking the Datalog evaluation in an asynchronous incremental dataflow framework *differential dataflow*. By building our prototype system FlowLog and performing extensive experiments comparing against existing systems, including RecStep introduced in Chapter 3, we show that this new way of thinking of Datalog evaluation could efficiently utilize computational resources by leveraging asynchronous execution of different computation stages and techniques such as maintained index sharing. The performance benefits of FlowLog over a batch processing system such as RecStep are significant when the amount of work done by each computation stage is small and when there are opportunities to share the index.

Chapter 6

Consistent Query Answering by Datalog

The semantic succinctness of Datalog allows easier analysis of complicated applications and design of effective techniques to solve the corresponding problems. In this Chapter, we present an efficient algorithm designed for managing inconsistent data, called consistent query answering, by leveraging the development efficiency of Datalog.

A database is *inconsistent* if it violates one or more integrity constraints that are supposed to be satisfied. Database inconsistency can naturally occur when the dataset results from an integration of heterogeneous sources, or because of noise during data collection.

Data cleaning [RD00] is the most widely used approach to manage inconsistent data in practice. It first *repairs* the inconsistent database by removing or modifying the inconsistent records so as to obey the integrity constraints. Then, users can run queries on a *clean* database. There has been a long line of research on data cleaning. Several frameworks have been proposed [GGM+21, ROA+21, GMPS13, AK09, GGZ03], using techniques such as knowledge bases and machine learning [RCIR17, CIKW16, BKL13b, HCG+18, EEI+13, LRB+21, BMNT15, CIP13, TCZ+14, KWW+16]. Data cleaning has also been studied in different contexts [KL21, CCX08, BKL13a, KIJ+15, BFG+07, PSC+15]. However, the process of data cleaning is often ad-hoc, and arbitrary choices are frequently made regarding which data to keep in order to restore database consistency. This comes at the price of losing important information since the number of cleaned versions of the database can be exponential in the size of the database. Moreover, data cleaning is commonly seen as a laborious and time-intensive process in data analysis. There have been efforts to accelerate the data cleaning process [RCIR17, CIP13, CMI+15, ROA+21], but in most cases, users need to wait until the data is clean before being able to query the database. *Consistent query answering* (CQA) is an alternative approach to data cleaning for managing inconsistent data [ABC99] that has recently received more attention [Wij19, Ber19]. Instead of singling out the *best repair*, CQA considers *all* possible repairs of the inconsistent database, returning the intersection of the query answers over all repairs, called the *consistent answers*. CQA serves as a viable complementary procedure to data cleaning for multiple reasons. First, it deals with inconsistent

System	Target Query Class	Intermediate Output	Backend
EQUIP [KPT13a]	SPJ	Big Integer Program	DBMS, BIP solver
CAvSAT [DK21, DK19]	SPJ	SAT formula	DBMS, SAT solver
Conquer [FFM05]	$\mathcal{C}_{\mathsf{forest}}$	SQL	DBMS
Conquesto [KJL+20]	SJF SPJ in **FO**	Datalog	Datalog engine
LinCQA (our system)	PPJT	SQL/Datalog	DBMS/Datalog engine

Table 6.1: A summary of systems for consistent query answering

data at query time without the need for an expensive offline cleaning process during which the users cannot query the database. Thus, users can quickly perform preliminary data analysis to obtain the consistent answers while waiting for the cleaned version of the database. Second, consistent answers can also be returned alongside the answers obtained after data cleaning by marking which answers are certainly/reliably correct and which are not. This information may provide additional guidance on critical decision-making data analysis tasks. Third, CQA can be used to design more efficient data cleaning algorithms [KLW+20].

In this chapter, we will focus on CQA for the most common kind of integrity constraint: *primary keys*. A primary-key constraint enforces that no two distinct tuples in the same table agree on all primary-key attributes. CQA under primary-key constraints has been extensively studied over the last two decades.

From a theoretical perspective, CQA for select-project-join (SPJ) queries is computationally hard, as it potentially requires inspecting an exponential number of repairs. However, for some SPJ queries, the consistent answers can be computed in polynomial time, and for some other SPJ queries, CQA is *first-order rewritable* (**FO**-rewritable): we can construct another query such that executing it directly on the inconsistent database will return the consistent answers of the original query. After a long line of research [KP12,KS14,KW15,KW19,KW21], it was proven that given any self-join-free SPJ query, the problem is either **FO**-rewritable, polynomial-time solvable but not **FO**-rewritable or **coNP**-complete [KW17].

From a system point of view, most CQA systems fall into two categories (summarized in Table 6.1): (1) systems that can compute the consistent answers of join queries with arbitrary denial constraints but require solvers for computationally hard problems (e.g., EQUIP [KPT13a] relies on Integer Programming solvers, and CAvSAT [DK21, DK19] requires SAT solvers), and (2) systems that output the **FO**-rewriting of the input query, but only target a specific class of queries that occurs frequently in practice. Fuxman and Miller [FFM05] identified a class of **FO**-rewritable queries called $\mathcal{C}_{\mathsf{forest}}$ and implemented their rewriting in ConQuer, which outputs a single SQL query. Conquesto [KJL+20] is the most recent system that targets **FO**-rewritable join queries by producing the rewriting in Datalog.

We identify several drawbacks with all the above systems. Both EQUIP and CAvSAT rely on solvers for **NP**-complete problems, which does not guarantee efficient termination, even if the input query is **FO**-rewritable. Although $\mathcal{C}_{\mathsf{forest}}$ captures many join queries seen in practice, it excludes queries that involve *(i)* joining with only part of a composite primary key, often appearing in snowflake schemas, and *(ii)* joining two tables on both primary-key and non-primary-key attributes, which commonly occur in settings such as entity matching and cross-comparison scenarios. Conquesto, on the other hand, implements the generic **FO**-rewriting algorithm without strong performance guarantees. Moreover, neither ConQuer nor Conquesto has theoretical guarantees on the running time of their produced rewritings.

Contributions. To address the above observed issues, we make the following contributions:

1. *Theory & Algorithms.* We identify a subclass of acyclic Boolean join queries that captures a wide range of queries commonly seen in practice for which we can produce **FO**-rewritings with a *linear running time guarantee* (Section 6.3). This class subsumes all acyclic Boolean queries in $\mathcal{C}_{\mathsf{forest}}$. For consistent databases, Yannakakis's algorithm [BFMY83] evaluates acyclic Boolean join queries in linear time in the size of the database. Our algorithm shows that even when inconsistency is introduced w.r.t. primary key constraints, the consistent answers of many acyclic Boolean join queries can still be computed in linear time, exhibiting no overhead to Yannakakis's algorithm. Our technical treatment follows Yannakakis' algorithm by considering a rooted join tree with an additional annotation of the **FO**-rewritability property, called a *pair-pruning join tree (PPJT)*. Our algorithm follows the pair-pruning join tree to compute the consistent answers and degenerates to Yannakakis's algorithm if the database has no inconsistencies.

2. *Implementation.* We implement our algorithm in LinCQA ($\underline{\text{Lin}}$ear $\underline{\text{C}}$onsistent $\underline{\text{Q}}$uery $\underline{\text{A}}$nswering), a system prototype that produces an efficient and optimized rewriting in both SQL and non-recursive Datalog rules with negation (Section 6.4).

3. *Evaluation.* We perform an extensive experimental evaluation comparing LinCQA to the other state-of-the-art CQA systems. Our findings show that *(i)* a properly implemented rewriting can significantly outperform a generic CQA system (e.g., CAvSAT); *(ii)* LinCQA achieves the best overall performance throughout all our experiments under different inconsistency scenarios; and *(iii)* the strong theoretical guarantees of LinCQA translate to a significant performance gap for worst-case database instances. LinCQA often outperforms other CQA systems, in several cases by orders of magnitude on both synthetic and real-world workloads. We also demonstrate that CQA can be an effective approach even for real-world datasets of very large scale ($\sim$ 400GB), which,

to the best of our knowledge, have not been tested before. We further present a showcase that by executing the Datalog rewriting that consists of non-recursive rules with negation in an efficient asynchronous Datalog system, we are able to surpass the state-of-the-art performance numbers. The evaluation of non-recursive Datalog program has been often dismissed in recent works targeting efficient Datalog program evaluation, due to their heavy focus on the recursive query processing.

6.1 Related Work

Inconsistency in databases has been studied in different contexts [CM05,KKD$^+$20,KDPV10, BF17,BF15,LB07,RBM13,CCP21]. The notion of Consistent Query Answering (CQA) was introduced in the seminal work by Arenas, Bertossi, and Chomicki [ABC99]. After twenty years, their contribution was recognized in a *Gems of PODS session* [Ber19]. An overview of complexity classification results in CQA appeared recently in the *Database Principles* column of SIGMOD Record [Wij19].

The term $\mathsf{CERTAINTY}(q)$ was coined in [Wij10] to refer to CQA for Boolean queries q on databases that violate primary keys, one per relation, which are fixed by q's schema. The complexity classification of $\mathsf{CERTAINTY}(q)$ for the class of self-join-free Boolean conjunctive queries started with the work by Fuxman and Miller [FM07], and was further pursued in [KP12,KS14,KW15,KW17,KW19,KW21], which eventually revealed that the complexity of $\mathsf{CERTAINTY}(q)$ for self-join-free (SJF) conjunctive queries displays a trichotomy between **FO**, **L**-complete, and **coNP**-complete. A recent result also extends the complexity classification of $\mathsf{CERTAINTY}(q)$ to path queries that may contain self-joins [KOW21]. The complexity of $\mathsf{CERTAINTY}(q)$ for self-join-free Boolean conjunctive queries with negated atoms was studied in [KW18]. For self-join-free Boolean conjunctive queries w.r.t. multiple keys, it remains decidable whether or not $\mathsf{CERTAINTY}(q)$ is in **FO** [KW20].

Several systems for CQA that are used for comparison in our study have already been introduced: ConQuer [FFM05], Conquesto [KJL$^+$20], CAvSAT [DK21,DK19], and EQUIP [KPT13a]. Most early systems for CQA rely on efficient solvers for Disjunctive Logic Programming and Answer Set Programming (ASP) [CMS04,GGZ03,MRT15,ABC03,MB05,LB07].

Similar notions to CQA are also emerging in machine learning with the aim of computing the consistent classification result of certain machine learning models over inconsistent training data [KLW$^+$20].

6.2 Preliminaries

In this section, we provide additional background and examples to help the reader better follow this chapter. We use the example **Company** database shown in Figure 6.1 to illustrate our constructs, where the primary-key attribute of each table is highlighted in **bold**.

Employee		
eid	office_city	wfh_city
0011	Boston	Boston
0011	Chicago	New York
0011	Chicago	Chicago
0022	New York	New York
0022	Chicago	Chicago
0034	Boston	New York

Manager		
office_city	mid	start_year
Boston	0011	2020
Boston	0011	2021
Chicago	0022	2020
LA	0034	2020
LA	0037	2020
New York	0022	2020

Contact	
office_city	cid
Boston	0011
Boston	0022
Chicago	0022
LA	0034
LA	0037
New York	0022

Figure 6.1: An inconsistent database (**Company**). **eid** is the primary key of Employee; *mid* and *cid* are the corresponding foreign keys in Manager and Contact. A repair of this database is highlighted in gray.

Inconsistent databases and integrity constraints. A database is inconsistent if it violates one or more *integrity constraints* that are supposed to be satisfied. Database inconsistency can naturally occur when the dataset results from an integration of heterogeneous sources, or because of noise during data collection. For this dissertation, we will focus on the most common kind of integrity constraint: *primary keys.* A primary-key constraint enforces that no two distinct tuples in the same table agree on all primary-key attributes.

Database instances, blocks, and repairs. A *database schema* is a finite set of table names. Each table name is associated with a finite sequence of *attributes*, and the length of that sequence is called the *arity* of that table. Some of these attributes are declared as *primary-key attributes*, forming together the *primary key.* A *database instance* **db** associates with each table name a finite set of *tuples* that agree on the arity of the table, called a *relation.* A relation is *consistent* if it does not contain two distinct tuples that agree on all primary-key attributes. A *block* of a relation is a maximal set of tuples that agree on all primary-key attributes. Thus, a relation is consistent if and only if it has no block with two or more tuples. A *repair* of a (possibly inconsistent) relation is obtained by selecting exactly one tuple from each block. Clearly, a relation with n blocks of size 2 each has 2^n repairs, an exponential number. A database instance **db** is consistent if all relations in it are consistent. A repair of a (possibly inconsistent) database instance is obtained by selecting one repair for each relation. For technical treatment, it will be convenient to view a database instance **db**

as a set of facts: if the relation associated with the table name R contains a tuple $\vec{t}$, then we say that $R(\vec{t})$ is a fact of **db**.

Example 3. *The **Company** database in Figure 6.1 is inconsistent with respect to the primary-key constraints. For example, in the* Employee *table there are 3 distinct tuples sharing the same primary key* eid 0011. *The blocks in the **Company** database are represented as rectangles. An example repair of the **Company** database can be obtained by* choosing exactly one tuple from each block*, and there are in total* $96 = 3 \times 2^5$ *distinct repairs.*

Atoms and key-equal facts. Let $\vec{x}$ be a sequence of variables and constants. We write $\mathsf{vars}(\vec{x})$ for the set of variables that appear in $\vec{x}$. An *atom* F with relation name R takes the form $R(\underline{\vec{x}}, \vec{y})$, where the primary key is underlined; we denote $\mathsf{key}(F) = \mathsf{vars}(\vec{x})$. Whenever a database instance **db** is understood, we write $R(\underline{\vec{c}}, *)$ for the block containing all tuples with primary-key value $\vec{c}$ in relation R.

Example 4. *In **Company**, we can have atoms* $\mathsf{Employee}(\underline{x}, y, y)$, $\mathsf{Manager}(\underline{u}, v, \mathsf{2020})$, *and* $\mathsf{Contact}(\underline{\mathsf{LA}}, \mathsf{2020})$. *The block* $\mathsf{Manager}(\underline{\mathsf{Boston}}, *)$ *contains two facts:* Manager($\underline{\text{Boston}}$, 0011, 2020) *and* Manager($\underline{\text{Boston}}$, 0011, 2021).

Conjunctive Queries. For select-project-join (SPJ) queries, we will also use the term *conjunctive queries (CQ)*. Each CQ q can be represented as a succinct rule of the following form:

$$q(\vec{u}) \text{ :- } R_1(\underline{\vec{x_1}}, \vec{y_1}), \ldots, R_n(\underline{\vec{x_n}}, \vec{y_n}) \tag{6.1}$$

where each $R_i(\underline{\vec{x_i}}, \vec{y_i})$ is an atom for $1 \leq i \leq n$. We denote by $\mathsf{vars}(q)$ the set of variables that occur in q and $\vec{u}$ is said to be the *free variables* of q. The atom $q(\vec{u})$ is the *head* of the rule, and the remaining atoms are called the *body* of the rule $\mathsf{body}(q)$. A CQ q is Boolean (BCQ) if it has no free variables, and it is *full* if all its variables are free. We say that q has a *self-join* if some relation name occurs more than once in q. A CQ without self-joins is called *self-join-free* (SJF). If a self-join-free query q is understood, an atom $R(\underline{\vec{x}}, \vec{y})$ in q can be denoted by R. If the body of a CQ of the form (6.1) can be partitioned into two non-empty parts that have no variable in common, then we say that the query is *disconnected*; otherwise it is *connected.* For a CQ q, let $\vec{x} = \langle x_1, \ldots, x_\ell \rangle$ be a sequence of distinct variables that occur in q and $\vec{a} = \langle a_1, \ldots, a_\ell \rangle$ be a sequence of constants, then $q_{[\vec{x} \to \vec{a}]}$ denotes the query obtained from q by replacing all occurrences of x_i with a_i for all $1 \leq i \leq \ell$.

Example 5. ***Conjunctive Queries*** *6.2: Consider the query over the **Company** database that* returns the id's of all employees who work in some office city with a manager who started in year 2020. *It can be expressed by the following SQL query:*

```
SELECT E.eid
FROM Employee E, Manager M
WHERE E.office_city=M.office_city AND M.start_year=2020
```

and the following CQ:

$$q(x) \text{ :- } \mathsf{Employee}(\underline{x}, y, z), \mathsf{Manager}(\underline{y}, w, \mathit{2020}).$$

The following CQ q' is a Boolean conjunctive query (BCQ), since it merely asks whether 0011 *is such an eid satisfying the conditions in q:*

$$q'() \text{ :- } \mathsf{Employee}(\underline{0011}, y, z), \mathsf{Manager}(\underline{y}, w, \mathit{2020}).$$

It is easy to see that q' is equivalent to $q_{[x \to 0011]}$.

Consistent Query Answering For every CQ q, given an input database instance **db**, the problem $\mathsf{CERTAINTY}(q)$ asks for the intersection of query outputs over all repairs of **db**. If q is Boolean, the problem $\mathsf{CERTAINTY}(q)$ then asks whether q is satisfied by every repair of the input database instance **db**. The problem $\mathsf{CERTAINTY}(q)$ has a first-order rewriting (**FO**-rewriting) if there is another first-order query q' (which, in most cases, uses the difference operator and, hence, is not an SPJ query) such that evaluating q' on the input database **db** would return the answers of $\mathsf{CERTAINTY}(q)$. In other words, executing q' directly on the inconsistent database *simulates* computing the original query q over all possible repairs.

Example 6. *Recall that in Example 5, the query q returns* $\{0011, 0022, 0034\}$ *on the inconsistent database* ***Company****. For $\mathsf{CERTAINTY}(q)$ however, the only output is* 0022*: for any repair that contains the tuples* Employee(<u>0011</u>, Boston, Boston) *and* Manager(<u>Boston</u>, 0011, 2021)*, neither* 0011 *nor* 0034 *would be returned by q; and in any repair,* 0022 *is returned by q with the following crucial observation:* Regardless of which tuple in $\mathsf{Employee}(\underline{0022}, *)$ the repair contains, both offices are present in the Manager table and both managers in Chicago and New York offices started in 2020.

Based on the observation, it is sufficient to solve $\mathsf{CERTAINTY}(q)$ by running the following single SQL query, called an **FO***-rewriting of $\mathsf{CERTAINTY}(q)$.*

```
SELECT E.employee_id
FROM Employee E
EXCEPT
    SELECT E.employee_id FROM Employee E
    WHERE E.office_city NOT IN (
      SELECT M.office_city
      FROM Manager EXCEPT
SELECT M.office_city
FROM Manager
WHERE M.start_year <> 2020)
```

Acyclic queries and join trees Let q be a CQ. A *join tree* of q is an undirected tree whose nodes are the atoms of q such that for every two distinct atoms R and S, their common variables occur in all atoms on the unique path from R to S in the tree. A CQ q is *acyclic*[1] if it has a join tree. If τ is a subtree of a join tree of a query q, we will denote by q_τ the query whose atoms are the nodes of τ. Whenever R is a node in an undirected tree τ, then (τ, R) denotes the rooted tree obtained by choosing R as the root of the tree.

Example 7. *The join tree of the query q in Example 5 has a single edge between $\mathsf{Employee}(\underline{x}, y, z)$ and $\mathsf{Manager}(\underline{y}, w, \mathit{2020})$.*

Attack graphs. Let q be an acyclic, self-join-free BCQ with join tree τ. For every atom $F \in q$, we define $F^{+,q}$ as the set of all variables in q that are functionally determined by $\mathsf{key}(F)$ with respect to all functional dependencies of the form $\mathsf{key}(G) \rightarrow \mathsf{vars}(G)$ with $G \in q \setminus \{F\}$.

Following [Wij12], the *attack graph* of q is a directed graph whose vertices are the atoms of q. There is a directed edge, called *attack*, from F to G ($F \neq G$), if on the unique path between F and G in τ, every two adjacent atoms share a variable not in $F^{+,q}$. An atom without incoming edges in the attack graph is called *unattacked*. The attack graph of q is used to determine the data complexity of $\mathsf{CERTAINTY}(q)$: the attack graph of q is acyclic if and only if $\mathsf{CERTAINTY}(q)$ is in **FO** [KW17].

Example 8. *For the query q in Example 5, $\mathsf{Employee}^{+,q} = \{x\}$ and $\mathsf{Manager}^{+,q} = \{y\}$. It follows that $\mathsf{Employee}$ attacks $\mathsf{Manager}$ because the variable y is shared between atoms $\mathsf{Employee}$ and $\mathsf{Manager}$ and $y \notin \mathsf{Employee}^{+,q}$. However, $\mathsf{Manager}$ does not attack $\mathsf{Employee}$ since the only shared variable y is in $\mathsf{Manager}^{+,q}$.*

*The attack graph of q is acyclic since it only contains one attack from $\mathsf{Employee}$ to $\mathsf{Manager}$. It follows that $\mathsf{CERTAINTY}(q)$ is in **FO**, as witnessed by the **FO**-rewriting in Example 6.*

[1]Throughout this chapter, whenever we say that a CQ is acyclic, we mean acyclicity as defined in [BFMY83], a notion that today is also known as α-acyclicity, to distinguish it from other notions of acyclicity.

6.3 A Linear-Time Rewriting

Before presenting our linear-time rewriting for $\mathsf{CERTAINTY}(q)$, we first provide a motivating example. Consider the following query on the **Company** database shown in Figure 6.1:

> *Is there an office whose contact person works for the office and, moreover, manages the office since 2020?*

This query can be expressed by the following CQ:

$$q^{\mathsf{ex}}() \text{ :- } \mathsf{Employee}(\underline{x}, y, z), \mathsf{Manager}(\underline{y}, x, 2020), \mathsf{Contact}(\underline{y}, x).$$

To the best of our knowledge, the most efficient running time for $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ guaranteed by existing systems is quadratic in the input database size, denoted N. The problem $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ admits an **FO**-rewriting according to the classification theorem in [KOW21]. However, the non-recursive Datalog rewriting of $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ produced by Conquesto contains cartesian products between two tables, meaning that it runs in $\Omega(N^2)$ time in the worst case. Also, since q^{ex} is not in $\mathcal{C}_{\mathsf{forest}}$, ConQuer cannot produce an **FO**-rewriting. EQUIP and CAvSAT solve the problem through Integer Programming or SAT solvers, which can take exponential time. One key observation is that q^{ex} requires a primary-key to primary-key join and a non-key to non-key join at the same time. As will become clear in our technical treatment in Section 6.3.2, this property allows us to solve $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ in $O(N)$ time, while existing CQA systems will run in more than linear time.

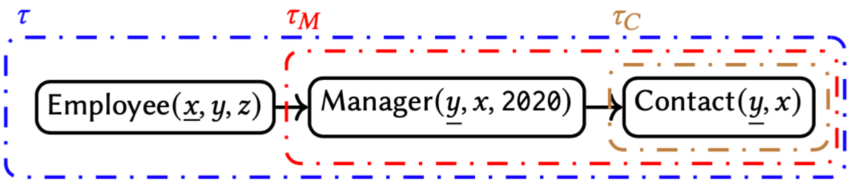
The remainder of this section is organized as follows. In Section 6.3.1, we introduce the pair-pruning join tree (PPJT). In Section 6.3.2, we consider every Boolean query q having a PPJT and present a novel linear-time non-recursive Datalog program for $\mathsf{CERTAINTY}(q)$ (Theorem 1). Finally, we extend our result to all acyclic self-join-free CQs in Section 6.3.3 (Theorem 2) .

6.3.1 Pair-pruning Join Tree

Here we introduce the notion of a *pair-pruning join tree* (PPJT). We first assume that the query q is connected, and then discuss how to handle disconnected queries. Recall that an atom in a self-join-free query can be uniquely denoted by its relation name. For example, we may use $\mathsf{Employee}$ as a shorthand for the atom $\mathsf{Employee}(\underline{x}, y, z)$ in q^{ex}.

Definition 1 (PPJT). *Let q be an acyclic self-join-free BCQ. Let τ be a join tree of q and R a node in τ. The tree (τ, R) is a pair-pruning join tree (PPJT) of q if for any rooted subtree (τ', R') of (τ, R), the atom R' is unattacked in $q_{\tau'}$.*

Example 9. *For the join tree τ in Figure 6.2, the rooted tree $(\tau, \mathsf{Employee})$ is a PPJT for q^{ex}. The atom $\mathsf{Employee}(\underline{x}, y, z)$ is unattacked in q. For the child subtree $(\tau_M, \mathsf{Manager})$ of*



τ
τ_M
τ_C
Employee($\underline{x}$, y, z)
Manager($\underline{y}$, x, 2020)
Contact($\underline{y}$, x)

Figure 6.2: A pair-pruning join tree (PPJT) of the query q^{ex}.

$(\tau, \mathit{Employee})$, *the atom* $\mathit{Manager}(\underline{y}, x, 2020)$ *is also unattacked in the following subquery*

$$q^{\mathsf{ex}}_{\tau_M}() \text{ :- } \mathit{Manager}(\underline{y}, x, 2020), \mathit{Contact}(\underline{y}, x).$$

Finally, for the subtree $(\tau_C, \mathit{Contact})$, *the atom* $\mathit{Contact}(\underline{y}, x)$ *is also unattacked in the corresponding subquery* $q^{\mathsf{ex}}_{\tau_C}() \text{ :- } \mathit{Contact}(\underline{y}, x)$. *Hence* $(\tau, \mathit{Employee})$ *is a PPJT of* q^{ex}.

Which queries admit a PPJT? As we show next, having a PPJT is a sufficient condition for the existence of an **FO**-rewriting.

Proposition 1. *Let* q *be an acyclic self-join-free BCQ. If* q *has a PPJT, then* $\mathsf{CERTAINTY}(q)$ *admits an* **FO***-rewriting.*

Proposition 1 is proved in the appendix of the technical report [FKOW22], in which we show that if q has a PPJT, then the attack graph of q must be acyclic. We note that not all acyclic self-join-free BCQs with an acyclic attack graph have a PPJT, as demonstrated in the next example.

Example 10. *Let* $q() \text{ :- } R(\underline{x, w}, y), S(\underline{y, w}, z), T(\underline{w}, z)$. *The attack graph of* q *is acyclic. The only join tree* τ *of* q *is the path* $R - S - T$. *However, neither* (τ, R) *nor* (τ, S) *is a PPJT for* q *since* R *and* S *are attacked in* q*; and* (τ, T) *is not a PPJT since in its subtree* (τ', S), S *is attacked in the subquery that contains* R *and* S.

Fuxman and Miller [FM07] identified a large class of self-join-free CQs, called $\mathcal{C}_{\mathsf{forest}}$, that includes most queries with primary-key-foreign-key joins, path queries, and queries on a star schema, such as found in SSB and TPC-H [OOCR09, PF00]. This class covers many of the SPJ queries seen in practical settings. In view of this, the following proposition is of practical significance.

Proposition 2. *Every acyclic BCQ in* $\mathcal{C}_{\mathsf{forest}}$ *has a PPJT.*

Furthermore, it is easy to verify that, unlike $\mathcal{C}_{\mathsf{forest}}$, PPJT captures *all* **FO**-rewritable self-join-free SPJ queries on two tables, a.k.a. binary joins. For example, the binary join q_5 in Section 6.5 admits a PPJT but is not in $\mathcal{C}_{\mathsf{forest}}$. Proposition 2 is proved in the appendix of the technical report [FKOW22].

How to find a PPJT. For any acyclic self-join-free BCQ q, we can check whether q admits a PPJT via a brute-force search over all possible join trees and roots. If q involves n relations, then there are at most n^{n-1} candidate rooted join trees for PPJT (n^{n-2} join trees and for each join tree, n choices for the root). For the data complexity of $\mathsf{CERTAINTY}(q)$, this exhaustive search runs in constant time since we assume n is a constant. In practice, the search cost is acceptable for most join queries that do not involve too many tables.

Appendix of the technical report [FKOW22] shows that the foregoing brute-force search for q can be optimized to run in polynomial time when q has an acyclic attack graph and, when expressed as a rule, does not contain two distinct body atoms $R(\underline{\vec{x}}, \vec{y})$ and $S(\underline{\vec{u}}, \vec{w})$ such that every variable occurring in $\vec{x}$ also occurs in $\vec{u}$. Most queries we observe and used in our experiments fall under this category.

Main Result. We previously showed that the existence of a PPJT implies an **FO**-rewriting that computes the consistent answers. Our main result shows that it also leads to an efficient algorithm that runs in linear time.

Theorem 1. *Let q be an acyclic self-join-free BCQ that admits a PPJT, and* **db** *be a database instance of size N. Then, there exists an algorithm for* $\mathsf{CERTAINTY}(q)$ *that runs in time $O(N)$.*

It is worth contrasting our result with Yannakakis' algorithm, which computes the result of any acyclic BCQ also in linear time $O(N)$ [Yan81]. Hence, the existence of a PPJT implies that computing $\mathsf{CERTAINTY}(q)$ will have the same asymptotic complexity.

Disconnected CQs. Every disconnected BCQ q can be written as $q = q_1, q_2, \ldots, q_n$ where $\mathsf{vars}(q_i) \cap \mathsf{vars}(q_j) = \emptyset$ for $1 \leq i < j \leq n$ and each q_i is connected. If each q_i has a PPJT, then $\mathsf{CERTAINTY}(q)$ can be solved by checking whether the input database is a "yes"-instance for each $\mathsf{CERTAINTY}(q_i)$, by Lemma B.1 of [KOW21].

6.3.2 The Rewriting Rules

We now show how to produce an efficient rewriting in Datalog and prove Theorem 1. In Section 6.4, we will discuss how to translate the Datalog program to SQL. Let q be an acyclic self-join-free BCQ with a PPJT (τ, R) and **db** an instance for the problem $\mathsf{CERTAINTY}(q)$: does the query q evaluate to true on every repair of **db**?

Let us first revisit Yannakakis' algorithm for evaluating q on a database **db** in linear time. Given a rooted join tree (τ, R) of q, Yannakakis' algorithm visits all nodes in a bottom-up fashion. For every internal node S of (τ, R), it keeps the tuples in table S that join with every child of S in (τ, R), where each such child has been visited recursively. In the end, the algorithm returns whether the root table R is empty or not. Equivalently, Yannakakis' algorithm evaluates q on **db** by removing tuples from each table that cannot contribute to an answer in **db** at each recursive step.

Our algorithm for CQA proceeds like Yannakakis' algorithm in a bottom-up fashion. At each step, we remove tuples from each table that cannot contribute to an answer to q in at least one repair of **db**. Informally, if a tuple cannot contribute to an answer in at least one repair of **db** containing it, then it cannot contribute to a consistent answer to q on **db**. Specifically, given a PPJT (τ, R) of q, to compute all tuples of each internal node S of (τ, R) that may contribute to a consistent answer, we need to *prune* the blocks of S in which there is some tuple that violates either the local selection condition on table S, or the joining condition with some child table of S in (τ, R). The term *pair-pruning* is motivated by the latter process, where we consider only one pair of tables at a time. This idea is formalized in Algorithm 3, where the procedures SELF-PRUNING and PAIR-PRUNING prune, respectively, the blocks that violate the local selection condition and the joining condition.

To ease the exposition of the rewriting, we now present both procedures in Datalog syntax. We will use two predicates for every atom S in the tree (let T be the unique parent of S in τ):

- the predicate S_{fkey} has arity equal to $|\mathsf{key}(S)|$ and collects the primary-key values of the S-table that cannot contribute to a consistent answer for q [2]; and

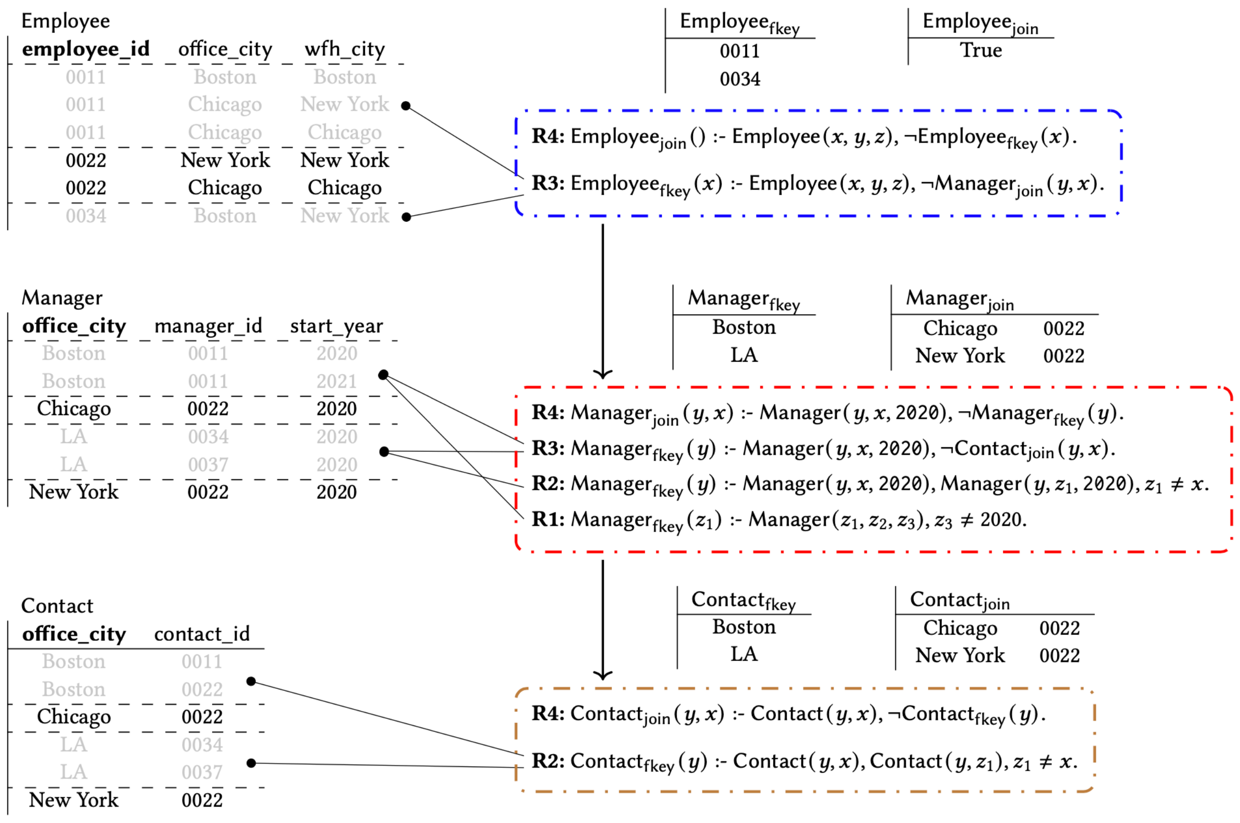
- the predicate S_{join} has arity equal to $|\mathsf{vars}(S) \cap \mathsf{vars}(T)|$ and collects the values for these variables in the S-table that may contribute to a consistent answer.

Algorithm 3: PPJT-REWRITING(τ, R)

```
Input: PPJT $(\tau, R)$ of $q$
Output: a Datalog program $P$ deciding $\mathsf{CERTAINTY}(q)$
1 $P := \emptyset$
2 $P := P \cup$ SELF-PRUNING$(R)$
3 foreach child node $S$ of $R$ in $\tau$ do
4     $P := P \cup$ PPJT-REWRITING$(\tau, S)$
5     $P := P \cup$ PAIR-PRUNING$(R, S)$
6 $P := P \cup$ EXIT-RULE$(R)$
7 return $P$
```

Figure 6.3 depicts how each step generates the rewriting rules for q^{ex}. We now describe how each step is implemented in detail.

[2]The f in fkey is for "<u>f</u>alse key".



Employee

employee_id	office_city	wfh_city
0011	Boston	Boston
0011	Chicago	New York
0011	Chicago	Chicago
0022	New York	New York
0022	Chicago	Chicago
0034	Boston	New York

$\text{Employee}_{\text{fkey}}$
0011
0034

$\text{Employee}_{\text{join}}$
True

R4: $\text{Employee}_{\text{join}}() \text{ :- } \text{Employee}(x, y, z), \neg\text{Employee}_{\text{fkey}}(x)$.

R3: $\text{Employee}_{\text{fkey}}(x) \text{ :- } \text{Employee}(x, y, z), \neg\text{Manager}_{\text{join}}(y, x)$.

Manager

office_city	manager_id	start_year
Boston	0011	2020
Boston	0011	2021
Chicago	0022	2020
LA	0034	2020
LA	0037	2020
New York	0022	2020

$\text{Manager}_{\text{fkey}}$
Boston
LA

$\text{Manager}_{\text{join}}$	
Chicago	0022
New York	0022

R4: $\text{Manager}_{\text{join}}(y, x) \text{ :- } \text{Manager}(y, x, 2020), \neg\text{Manager}_{\text{fkey}}(y)$.

R3: $\text{Manager}_{\text{fkey}}(y) \text{ :- } \text{Manager}(y, x, 2020), \neg\text{Contact}_{\text{join}}(y, x)$.

R2: $\text{Manager}_{\text{fkey}}(y) \text{ :- } \text{Manager}(y, x, 2020), \text{Manager}(y, z_1, 2020), z_1 \neq x$.

R1: $\text{Manager}_{\text{fkey}}(z_1) \text{ :- } \text{Manager}(z_1, z_2, z_3), z_3 \neq 2020$.

Contact

office_city	contact_id
Boston	0011
Boston	0022
Chicago	0022
LA	0034
LA	0037
New York	0022

$\text{Contact}_{\text{fkey}}$
Boston
LA

$\text{Contact}_{\text{join}}$	
Chicago	0022
New York	0022

R4: $\text{Contact}_{\text{join}}(y, x) \text{ :- } \text{Contact}(y, x), \neg\text{Contact}_{\text{fkey}}(y)$.

R2: $\text{Contact}_{\text{fkey}}(y) \text{ :- } \text{Contact}(y, x), \text{Contact}(y, z_1), z_1 \neq x$.

Figure 6.3: The non-recursive Datalog program for evaluating $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ together with an example execution on the **Company** database in Figure 6.1. The faded-out rows denote blocks that are removed since they do not contribute to any consistent answer. The arrows denote which rules remove which blocks (some blocks can be removed by multiple rules).

<u>SELF-PRUNING(R)</u>: Let $R(\underline{x_1, \ldots x_k}, x_{k+1}, \ldots, x_n)$, where x_i can be a variable or a constant. The first rule finds the primary-key values of the R-table that can be pruned because some tuple with that primary-key violates the local selection conditions imposed on R.

Rule 1. *If $x_i = c$ for some constant c, we add the rule*

$$R_{\mathsf{fkey}}(z_1, \ldots, z_k) \text{ :- } R(z_1, \ldots, z_n), z_i \neq c.$$

If for some variable x_i there exists $j < i$ with $x_i = x_j$, we add the rule

$$R_{\mathsf{fkey}}(z_1, \ldots, z_k) \text{ :- } R(z_1, \ldots, z_n), z_i \neq z_j.$$

Here, $z_1, \ldots, z_n$ are fresh distinct variables.

The second rule finds the primary-key values of the R-table that can be pruned because R joins with its parent T in the tree. The underlying intuition is that if some R-block of the input database contains two tuples that disagree on a non-key position that is used in an equality-join with T, then for every given T-tuple t, we can pick an R-tuple in that block that does not join with t. Therefore, that R-block cannot contribute to a consistent answer.

Rule 2. *For each variable x_i with $i > k$ (so in a non-key position) such that $x_i \in \mathsf{vars}(T)$, we produce a rule*

$$\begin{aligned} R_{\mathsf{fkey}}(x_1, \dots, x_k) \text{ :- } & R(x_1, \dots, x_k, x_{k+1}, \dots, x_n), \\ & R(x_1, \dots, x_k, z_{k+1}, \dots, z_k), z_i \neq x_i. \end{aligned}$$

where $z_{k+1}, \dots, z_n$ are fresh variables.

Example 11. *The self-pruning phase on$(\tau_M, \mathsf{Manager})$ produces one rule using Rule 1. When executed on the* **Company** *database, the key Boston is added to $\mathsf{Manager}_{\mathsf{fkey}}$, since the tuple (Boston, 0011, 2021) has $\mathsf{start\_year} \neq 2020$. Finally, the self-pruning phase on the PPJT $(\tau_C, \mathsf{Contact})$ produces one rule using Rule 2 (here x is the non-key join variable). Hence, the keys Boston and LA will be added to $\mathsf{Contact}_{\mathsf{fkey}}$.*

<u>PAIR-PRUNING(R, S)</u>: Suppose that q contains the atoms $R(\underline{\vec{x}}, \vec{y})$ and $S(\underline{\vec{u}}, \vec{v})$, where the S-atom is a child of the R-atom in the PPJT. Let $\vec{w}$ be a sequence of distinct variables containing all (and only) variables in $\mathsf{vars}(R) \cap \mathsf{vars}(S)$. The third rule prunes all R-blocks containing some tuple that cannot join with any S-tuple to contribute to a consistent answer.

Rule 3. *Add the rule*

$$R_{\mathsf{fkey}}(\vec{x}) \text{ :- } R(\vec{x}, \vec{y}), \neg S_{\mathsf{join}}(\vec{w}),$$

where the rules for S_{join} will be defined in Rule 4.

The rule is safe because every variable in $\vec{w}$ occurs in $R(\vec{x}, \vec{y})$.

Example 12. *Figure 6.3 shows the two pair-pruning rules generated (in general, there will be one pair-pruning rule for each parent-child edge in the PPJT. In both cases, the join variables are $\{y, x\}$. For the table $\mathsf{Employee}$, the rule prunes the two blocks with keys $0011, 0034$ and adds them to $\mathsf{Employee}_{\mathsf{fkey}}$.*

<u>EXIT-RULE(R)</u>: Suppose that q contains $R(\underline{\vec{x}}, \vec{y})$. If R is an internal node, let $\vec{w}$ be a sequence of distinct variables containing all (and only) the join variables of R and its parent node in

τ. If R is the root node, let $\vec{w}$ be the empty vector. The exit rule removes the pruned blocks of R and projects on the variables in $\vec{w}$. If R is an internal node, the resulting tuples in the projection could contribute to a consistent answer, and will be later used for pair pruning; if R is the root, the projection returns the final result.

Rule 4. *If R_{fkey} exists in the head of a rule, we produce the rule*

$$R_{\mathsf{join}}(\vec{w}) \text{ :- } R(\vec{x}, \vec{y}), \neg R_{\mathsf{fkey}}(\vec{x}).$$

Otherwise, we produce the rule

$$R_{\mathsf{join}}(\vec{w}) \text{ :- } R(\vec{x}, \vec{y}).$$

Example 13. *Figure 6.3 shows the three exit rules for q^{ex}—one rule for each node in the PPJT. The boolean predicate $\mathsf{Employee}_{\mathsf{join}}$ determines whether* `True` *is the consistent answer to the query.*

Runtime Analysis It is easy to see that **Rule 1, 3, and 4** can be evaluated in linear time. We now argue how to evaluate **Rule 2** in linear time as well. Indeed, instead of performing the self-join on the key, it suffices to create a hash table using the primary key as the hash key (which can be constructed in linear time). Then, for every value of the key, we can easily check whether all tuples in the block have the same value at the i-th attribute.

Sketch of Correctness Let q be an acyclic self-join-free BCQ with a PPJT (τ, R) and **db** an instance for $\mathsf{CERTAINTY}(q)$. The easier property to show is the *soundness* of our rewriting **Rules 1, 2, 3, 4**: if the predicate R_{join} is nonempty when our rewriting is executed on **db**, then every repair of **db** must necessarily satisfy q. The argumentation uses a straightforward bottom-up induction on the PPJT: for every rooted subtree (τ', S) of (τ, R), the tuples in S_{join} are consistent answers to the corresponding subquery $q_{\tau'}$ projected on the join variables with the parent of S (i.e., on the variables $\vec{w}$ in **Rule 4**).

The more difficult property to show is the *completeness* of our rewriting rules: if every repair of **db** satisfies q, then the predicate R_{join} must be nonempty after executing the rules on **db**. The crux here is a known result (see, for example, Lemma 4.4 in [KW17]) which states that for every unattacked atom R in a self-join-free BCQ q, the following holds true:

> *if every repair of* **db** *satisfies q, then there is a nonempty block* **b** *of R such that in each repair of* **db**, *the query q can be made true by using the (unique) tuple of* **b** *in that repair.*

Our recursive construction of a PPJT (τ, R) ensures that for each rooted subtree (τ', S) of (τ, R), S is unattacked in $q_{\tau'}$. Therefore, it suffices to compute the blocks in S that could contribute to a consistent answer to $q_{\tau'}$ at each recursive step in a bottom-up fashion, eventually returning the consistent answer to q in $\mathbf{db}$.

The soundness and completeness arguments taken together imply that our rewriting rules return only and all consistent answers. The formal correctness proof is in the appendix of the technical report [FKOW22].

6.3.3 Extension to Non-Boolean Queries

Let $q(\vec{u})$ be an acyclic self-join-free CQ with free variables $\vec{u}$, and $\mathbf{db}$ be a database instance. If $\vec{c}$ is a sequence of constants of the same length as $\vec{u}$, we say that $\vec{c}$ is a *consistent answer* to q on $\mathbf{db}$ if $\vec{c} \in q(I)$ in every repair I of $\mathbf{db}$. Furthermore, we say that $\vec{c}$ is a *possible answer* to q on $\mathbf{db}$ if $\vec{c} \in q(\mathbf{db})$. It can be easily seen that for CQs every consistent answer is a possible answer.

Lemma 1 reduces computing the consistent answers of non-Boolean queries to that of Boolean queries.

Lemma 1. *Let q be a CQ with free variables $\vec{u}$, and let $\vec{c}$ be a sequence of constants of the same length as $\vec{u}$. Let $\mathbf{db}$ be an database instance. Then $\vec{c}$ is a consistent answer to q on $\mathbf{db}$ if and only if $\mathbf{db}$ is a* yes-*instance for* $\mathsf{CERTAINTY}(q_{[\vec{u}\to\vec{c}]})$.

If q has free variables $\vec{u} = (u_1, u_2, \ldots, u_n)$, we say that q admits a PPJT if the Boolean query $q_{[\vec{u}\to\vec{c}]}$ admits a PPJT, where $\vec{c} = (c_1, c_2, \ldots, c_n)$ is a sequence of distinct constants. We can now state our main result for non-Boolean CQs.

Theorem 2. *Let q be an acyclic self-join-free Conjunctive Query that admits a PPJT, and $\mathbf{db}$ be a database instance of size N. Let OUT_p be the set of possible answers to q on $\mathbf{db}$, and OUT_c the set of consistent answers to q on $\mathbf{db}$. Then:*

1. *the set of consistent answers can be computed in time $O(N \cdot |\mathsf{OUT}_p|)$; and*
2. *moreover, if q is full, the set of consistent answers can be computed in time $O(N + |\mathsf{OUT}_c|)$.*

To contrast this with Yannakakis result, for acyclic full CQs we have a running time of $O(N + |\mathsf{OUT}|)$, and a running time of $O(N \cdot |\mathsf{OUT}|)$ for general CQs.

Proof Sketch. Our algorithm first evaluates q on $\mathbf{db}$ to yield a set S of size $|\mathsf{OUT}_p|$ in time $O(N \cdot |\mathsf{OUT}_p|)$. We then return all answers $\vec{c} \in S$ such that $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{u}\to\vec{c}]})$, which runs in $O(N)$ by Theorem 1. This approach gives an algorithm with running time $O(N \cdot |\mathsf{OUT}_p|)$.

If q is full, we proceed by (i) removing all blocks with at least two tuples from $\mathbf{db}$ to yield

$\mathbf{db}^c$ and (ii) evaluating q on $\mathbf{db}^c$. In our algorithm, step (i) runs in $O(N)$ and since q is full, step (ii) runs in time $O(N + |\mathsf{OUT}_c|)$. The correctness proof of the algorithm is in the appendix of the technical report [FKOW22]. $\square$

Rewriting for non-Boolean Queries Let $\vec{c} = (c_1, c_2, \dots, c_n)$ be a sequence of fresh, distinct constants. If $q_{[\vec{u} \to \vec{c}]}$ has a PPJT, the Datalog rewriting for $\mathsf{CERTAINTY}(q)$ can be obtained as follows:

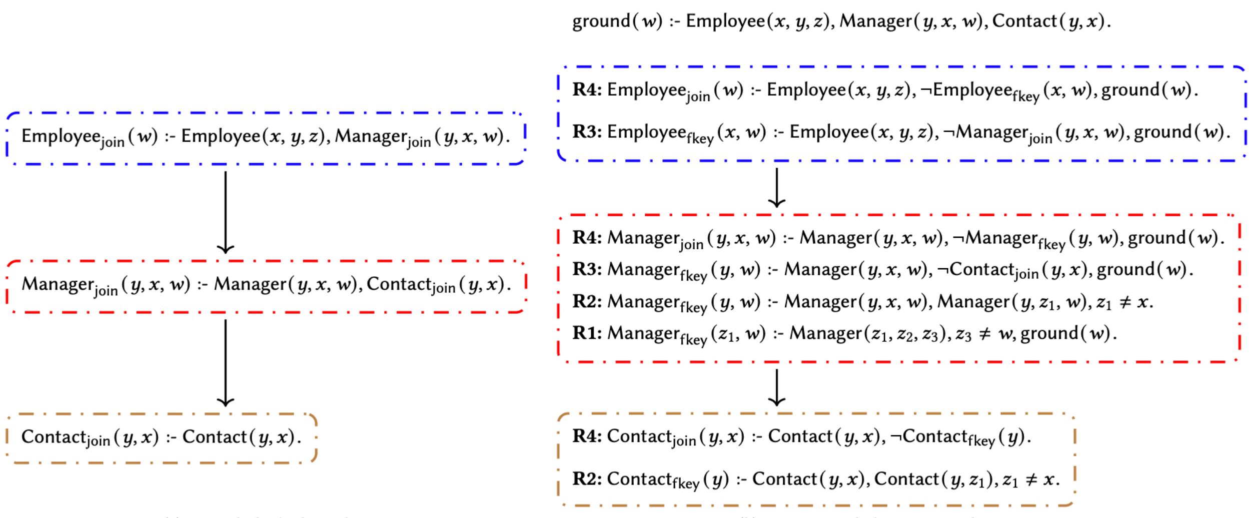
1. Produce the program P for $\mathsf{CERTAINTY}(q_{[\vec{u} \to \vec{c}]})$ using the rewriting algorithm for Boolean queries (Subsection 6.3.2).

2. Replace each occurrence of the constant c_i in P with the free variable u_i.

3. Add the rule: $\mathsf{ground}(\vec{u})$:- $\mathsf{body}(q)$.

4. For a relation T, let $\vec{u}_T$ be a sequence of all free variables that occur in the subtree rooted at T. Then, append $\vec{u}_T$ to every occurrence of T_{join} and T_{fkey}.

5. For any rule of P that has a free variable u_i that is unsafe, add the atom $\mathsf{ground}(\vec{u})$ to the rule.


ground(w) :- Employee(x, y, z), Manager(y, x, w), Contact(y, x).

Employee$_{\mathsf{join}}$(w) :- Employee(x, y, z), Manager$_{\mathsf{join}}$(y, x, w).

Manager$_{\mathsf{join}}$(y, x, w) :- Manager(y, x, w), Contact$_{\mathsf{join}}$(y, x).

Contact$_{\mathsf{join}}$(y, x) :- Contact(y, x).

R4: Employee$_{\mathsf{join}}$(w) :- Employee(x, y, z), ¬Employee$_{\mathsf{fkey}}$(x, w), ground(w).

R3: Employee$_{\mathsf{fkey}}$(x, w) :- Employee(x, y, z), ¬Manager$_{\mathsf{join}}$(y, x, w), ground(w).

R4: Manager$_{\mathsf{join}}$(y, x, w) :- Manager(y, x, w), ¬Manager$_{\mathsf{fkey}}$(y, w), ground(w).

R3: Manager$_{\mathsf{fkey}}$(y, w) :- Manager(y, x, w), ¬Contact$_{\mathsf{join}}$(y, x), ground(w).

R2: Manager$_{\mathsf{fkey}}$(y, w) :- Manager(y, x, w), Manager(y, z_1, w), $z_1 \neq x$.

R1: Manager$_{\mathsf{fkey}}$(z_1, w) :- Manager(z_1, z_2, z_3), $z_3 \neq w$, ground(w).

R4: Contact$_{\mathsf{join}}$(y, x) :- Contact(y, x), ¬Contact$_{\mathsf{fkey}}$(y).

R2: Contact$_{\mathsf{fkey}}$(y) :- Contact(y, x), Contact(y, z_1), $z_1 \neq x$.

(a) Yannakakis' Algorithm **(b) PPJT extended to non-Boolean queries**

Figure 4: The non-recursive Datalog program for q^{nex} and $\mathsf{CERTAINTY}(q^{\mathsf{nex}})$.

Figure 6.4: The non-recursive Datalog program for q^{nex} and $\mathsf{CERTAINTY}(q^{\mathsf{nex}})$.

Example 14. *Consider the non-Boolean query*

$$q^{\mathsf{nex}}(w) \text{ :- } \mathsf{Employee}(\underline{x}, y, z), \mathsf{Manager}(\underline{y}, x, w), \mathsf{Contact}(\underline{y}, x).$$

Note that the constant 2020 in q^{ex} is replaced by the free variable w in q^{nex}. Hence, the program P for $\mathsf{CERTAINTY}(q^{\mathsf{nex}}_{[w \to c]})$ *is the same as Figure 6.3, with the only difference that 2020 is replaced by the constant c. The ground rule produced is:*

$$\mathsf{ground}(w) \text{ :- } \mathsf{Employee}(x, y, z), \mathsf{Manager}(y, x, w), \mathsf{Contact}(y, x),$$

and Figure 5.4a shows how Yannakakis' algorithm evaluates q^{nex}.

To see how the rule of P would change for the non-Boolean case, consider the self-pruning rule for Contact*. This rule would remain as is, because it contains no free variable and the predicate $\mathsf{Contact}_{\mathsf{fkey}}$ remains unchanged. In contrast, consider the first self-pruning rule for* Manager*, which in P would be:*

$$\mathsf{Manager}_{\mathsf{fkey}}(y_1) \text{ :- } \mathsf{Manager}(y_1, y_2, y_3), y_3 \neq w.$$

Here, w is unsafe, so we need to add the atom $\mathsf{ground}(w)$*. Additionally, w is now a free variable in the subtree rooted at Manager, so the predicate $\mathsf{Manager}_{\mathsf{fkey}}(y_1)$ becomes $\mathsf{Manager}_{\mathsf{fkey}}(y_1, w)$. The transformed rule will be:*

$$\mathsf{Manager}_{\mathsf{fkey}}(y_1, w) \text{ :- } \mathsf{Manager}(y_1, y_2, y_3), y_3 \neq w, \mathsf{ground}(w).$$

The full rewriting for q^{nex} can be seen in Figure 5.4b.

The above rewriting process may introduce cartesian products in the rules. In the next section, we will see how we can tweak the rules in order to avoid this inefficiency.

6.4 Implementation

In this section, we first present LinCQA, a system that produces the consistent **FO**-rewriting of a query q in both Datalog and SQL formats if q has a PPJT. Having a rewriting in both formats allows us to use both Datalog and SQL engines as a backend. We then briefly discuss how we address the flaws of Conquer and Conquesto that impair their actual runtime performance.

LinCQA: Rewriting in Datalog/SQL

Our implementation takes as input a self-join-free CQ q written in either Datalog or SQL. LinCQA first checks whether the query q admits a PPJT, and if so, it proceeds to produce the consistent **FO**-rewriting of q in either Datalog or SQL, or it terminates otherwise.

Datalog rewriting. LinCQA implements all rules introduced in Subsection 6.3.2, with one modification to the ground rule atom when cartesian products are introduced. Let the input query be

$$q(\vec{u}) \text{ :- } R_1(\underline{\vec{x_1}}, \vec{y_1}), R_2(\underline{\vec{x_2}}, \vec{y_2}), \ldots, R_k(\underline{\vec{x_k}}, \vec{y_k}).$$

In Subsection 6.3.3, the head of the ground rule is $\mathsf{ground}(\vec{u})$. In the implementation, we replace that rule with

$$\mathsf{ground}^*(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_k, \vec{u}) \text{ :- } \mathsf{body}(q),$$

keeping the key variables of all atoms. For each unsafe rule with head $R_{i,\mathsf{label}}$ where $\mathsf{label} \in \{\mathsf{fkey}, \mathsf{join}\}$, let $\vec{v}$ be the key in the occurrence of R_i in the body of the rule (if the unsafe rule is produced by **Rule 2**, both occurrences of R_i share the same key). Then, we add to the rule body the atom

$$\mathsf{ground}^*(\vec{z}_1, \ldots, \vec{z}_{i-1}, \vec{v}, \vec{z}_{i+1}, \ldots, \vec{z}_k, \vec{u})$$

where $\vec{z}_i$ is a sequence of fresh variables of the same length as $\vec{x}_i$.

The rationale is that appending $\mathsf{ground}(\vec{u})$ to all unsafe rules could potentially introduce a Cartesian product between $\mathsf{ground}(\vec{u})$ and some existing atom $R(\vec{v}, \vec{w})$ in the rule. The Cartesian product has size $O(N \cdot |\mathsf{OUT}_p|)$ and would take $\Omega(N \cdot |\mathsf{OUT}_p|)$ time to compute, often resulting in inefficient evaluations or even out-of-memory errors. On the other hand, adding ground^* guarantees a join with an existing atom in the rule. Hence the revised rules would take $O(N + |\mathsf{ground}^*|)$ time to compute. Note that the size of ground^* can be as large as $N^k \cdot |\mathsf{OUT}_p|$ in the worst case; but as we observe in the experiments, the size of ground^* is small in practice.

SQL rewriting We now describe how to translate the Datalog rules in Subsection 6.3.2 to SQL queries. Given a query q, we first denote the following:

1. **KeyAttri**(R): the primary key attributes of relation R;
2. **JoinAttri**(R, T): the attributes of R that join with T;
3. **Comp**(R): the conjunction of comparison predicates imposed entirely on R, excluding all join predicates (e.g., $\mathsf{R}.A = 42$ and $\mathsf{R}.A = \mathsf{R}.B$); and
4. **NegComp**(R): the negation of **Comp**(R) (e.g., $\mathsf{R}.A \neq 42$ or $\mathsf{R}.A \neq \mathsf{R}.B$).

Translation of Rule 1. We translate **Rule 1** of Subsection 6.3.2 into the following SQL query computing the keys of R.

```
SELECT KeyAttri(R)
FROM R
WHERE NegComp(R)
```

Translation of Rule 2. First we produce the projection on all key attributes and the joining attributes of R with its parent T (if it exists), and then compute all blocks containing at least two facts that disagree on the joining attributes. This can be implemented effectively in SQL with `GROUP BY` and `HAVING`.

```
SELECT KeyAttri(R)
FROM
     (SELECT DISTINCT KeyAttri(R), JoinAttri(R, T)
      FROM R) t
GROUP BY KeyAttri(R)
HAVING COUNT(*) > 1
```

Translation of Rule 3. For **Rule 3** in the pair-pruning phase, we need to compute all blocks of R containing some fact that does not join with some fact in S_{join} for some child node S of R. This can be achieved through a *left outer join* between R and each of its child node $\mathsf{S}^1_{\mathsf{join}}$, $\mathsf{S}^2_{\mathsf{join}}$, $\ldots$, $\mathsf{S}^k_{\mathsf{join}}$, which are readily computed in the recursive steps. For each $1 \le i \le k$, let the attributes of S^i be B^i_1, B^i_2, $\ldots$, $B^i_{m_i}$, joining with attributes $A_{\alpha^i_1}$, $A_{\alpha^i_2}$, $\ldots$, $A_{\alpha^i_{m_i}}$ in R respectively. We produce the following rule:

```
SELECT KeyAttri(R)
FROM R
LEFT OUTER JOIN $\mathsf{S}^1_{\mathsf{join}}$ ON
   R.$A_{\alpha^1_1}$ = $\mathsf{S}^1_{\mathsf{join}}.B^1_1$ AND ... AND R.$A_{\alpha^1_{m_1}}$ = $\mathsf{S}^1_{\mathsf{join}}.B^1_{m_1}$
...
LEFT OUTER JOIN $\mathsf{S}^k_{\mathsf{join}}$ ON
   R.$A_{\alpha^k_1}$ = $\mathsf{S}^k_{\mathsf{join}}.B^k_1$ AND ... AND R.$A_{\alpha^k_{m_k}}$ = $\mathsf{S}^k_{\mathsf{join}}.B^k_{m_k}$
WHERE $\mathsf{S}^1_{\mathsf{join}}.B^1_1$ IS NULL OR ... $\mathsf{S}^1_{\mathsf{join}}.B^1_{m_1}$ IS NULL OR
      $\mathsf{S}^2_{\mathsf{join}}.B^2_1$ IS NULL OR ... $\mathsf{S}^2_{\mathsf{join}}.B^2_{m_2}$ IS NULL OR
      ...
      $\mathsf{S}^k_{\mathsf{join}}.B^k_1$ IS NULL OR ... $\mathsf{S}^k_{\mathsf{join}}.B^k_{m_k}$ IS NULL
```

The *inconsistent blocks* represented by the keys found by the above three queries are *combined* using `UNION ALL` (e.g., R_{fkey} in **Rule 1, 2, 3**).

Translation of Rule 4. Finally, we translate **Rule 4** computing the values on join attributes between *good blocks* in R and its unique parent T if it exists. Let $A_1, A_2, \ldots, A_k$ be the key attributes of R.

```
SELECT JoinAttri(R, T)
FROM R
WHERE NOT EXISTS (
SELECT *
FROM  R_fkey
WHERE R.A_1 = R_fkey.A_1 AND ... AND R.A_k = R_fkey.A_k)
```

If R is the root relation of the PPJT, we replace **JoinAttri**(R, T) with `DISTINCT 1` (i.e. a Boolean query). Otherwise, the results returned from the above query are stored as $\mathsf{R}_{\mathsf{join}}$ and the recursive process continues as described in Algorithm 3.

Extension to non-Boolean queries. Let q be a non-Boolean query. We use **ProjAttri**(q) to denote a sequence of attributes of q to be projected and let **CompPredicate**(q) be the comparison expression in the `WHERE` clause of q. We first produce the SQL query that computes the facts of ground^*.

```
SELECT KeyAttri(R_1), ..., KeyAttri(R_k), ProjAttri(q)
FROM R_1, R_2, ..., R_k
WHERE CompPredicate(q)
```

We then modify each SQL statement as follows. Consider a SQL statement whose corresponding Datalog rule is unsafe and let $T(\vec{v}, \vec{w})$ be an atom in the rule body. Let $\vec{u}_T$ be a sequence of free variables in q_{τ_T} and let **FreeAttri**(T) be a sequence of attributes in q_{τ_T} to be projected (i.e., corresponding to the variables in $\vec{u}_T$). Recall that $T_{\mathsf{join}}(\vec{v})$ and $T_{\mathsf{fkey}}(\vec{v})$ would be replaced with $T_{\mathsf{join}}(\vec{v}, \vec{u}_T)$ and $T_{\mathsf{fkey}}(\vec{v}, \vec{u}_T)$ respectively, we thus first append **FreeAttri**(T) to the `SELECT` clause and then add a `JOIN` between table T and ground on all attributes in **KeyAttri**(T). Finally, for a rule that has some negated IDB containing a free variable corresponding to some attribute in ground (i.e., $\mathsf{ground}.A$),

1. if the rule is produced by **Rule 3,** in each `LEFT OUTER JOIN` with S^i_{join} we add the expression `ground.A = `$\mathsf{S}^i_{\mathsf{join}}.B$ connected by the `AND` operator, where B is an attribute to be projected in $\mathsf{S}^i_{\mathsf{join}}$. In the `WHERE` clause we also add an expression `ground.A IS NULL`, connected by the `OR` operator.

2. if the rule is produced by **Rule 4**, in the `WHERE` clause of the subquery we add an expression `ground.A = `$\mathsf{R}_{\mathsf{fkey}}$`.A`.

6.4.1 Improvements upon existing CQA systems

ConQuer [FFM05] and Conquesto [KJL$^+$20] are two other CQA systems that target their own subclasses of **FO**-rewritable queries, both with noticeable performance issues. For a fair comparison with LinCQA, we implemented our own optimized version of both systems. Specifically, we complement Conquer presented in [FFM05] which was only able to handle tree queries (a subclass of $\mathcal{C}_{\mathsf{forest}}$), allowing us to handle all queries in $\mathcal{C}_{\mathsf{forest}}$. Additionally, we optimized Conquesto [KJL$^+$20] to eliminate unnecessarily repeated computation and undesired cartesian products produced due to its original formulation. The optimized system has significant performance gains over the original implementation and is named FastFO. Readers interested in the details can refer to the appendix of the technical report [FKOW22].

6.5 Experiments

Our experimental evaluation addresses the following questions:

1. How do first-order rewriting techniques perform compared to generic state-of-the-art CQA systems (e.g., CAvSAT)?
2. How does LinCQA perform compared to other existing CQA techniques?
3. How do different CQA techniques behave on inconsistent databases with different properties (e.g., varying inconsistent block sizes, inconsistency)?
4. Are there instances where we can observe the worst-case guarantee of LinCQA that other CQA techniques lack?
5. Can our optimized asynchronous Datalog system outperform state-of-the-art RDBMS - SQL Server on CQA tasks?

To answer these questions, we performed experiments using synthetic benchmarks used in previous work and a large real-world dataset of 400GB. We compare LinCQA against several state-of-the-art CQA systems with improvements. To the best of our knowledge, this is the most comprehensive performance evaluation of existing CQA techniques, and we are the first to evaluate different CQA techniques on a real-world dataset of this large scale.

Experimental Setup

Next, we briefly describe the setup of our experiments.

System configuration. All of our experiments are carried out on a bare metal server in Cloudlab [Clo18], a large cloud infrastructure. The server runs Ubuntu 18.04.1 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. The server has a SATA SSD with 440GB space available, 160GB memory, and each NUMA node is directly attached to 80GB of memory. We run Microsoft SQL Server 2019 Developer Edition (64-bit) on Linux as the relational backend for all CQA systems for SQL-based rewritings. For CAvSAT, MaxHS v3.2.1 [DB11] is used as the solver for the output WPMaxSAT instances.

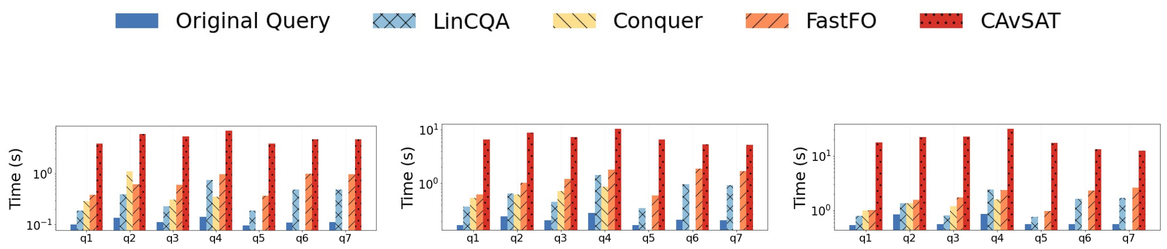
Other CQA systems. We compare the performance of LinCQA with several state-of-the-art CQA methods.

ConQuer: a CQA system that outputs a SQL rewriting for queries that are in $\mathcal{C}_{\mathsf{forest}}$ [FFM05]. We implement the complete version of ConQuer as described in Section 6.4.1.

FastFO: our own implementation of the general method that can handle any query for which CQA is **FO**-rewritable. It improves upon Conquesto [KJL+20] by addressing a few of its inefficiencies as described in Section 6.4.1.

CAvSAT: a recent SAT-based system. It reduces the complement of CQA with arbitrary denial constraints to a SAT problem, which is solved with an efficient SAT solver [DK19].

For LinCQA, ConQuer and FastFO, we only report execution time of **FO**-rewritings, since the rewritings can be produced within 1ms for all our queries. We report the performance of each **FO**-rewriting using the best query plan. The preprocessing time required by CAvSAT *prior* to computing the consistent answers is not reported. For each rewriting and database shown in the experimental results, we run the evaluation five times (unless timed out), discard the first run and report the average time of the last four runs.

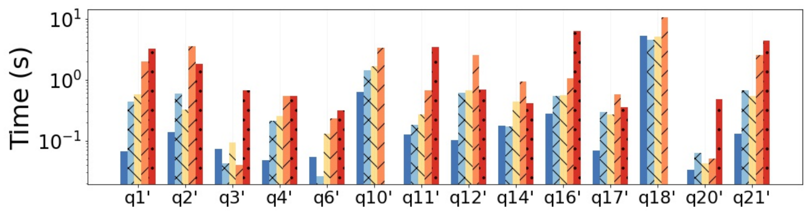


(a) rSize = 500K (b) rSize = 1M (c) rSize = 5M

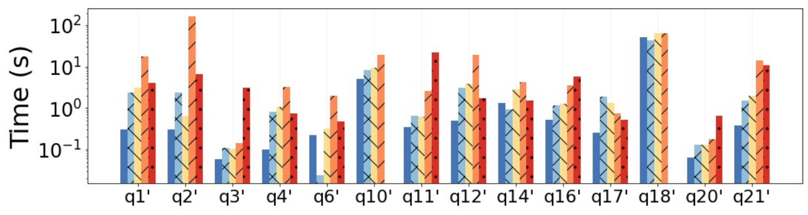
Figure 6.5: Performance comparison of different CQA systems on a synthetic workload with varying relation sizes.

Databases and Queries

Synthetic workload. We consider the synthetic workload used in previous work [KPT13b, DK19, Dix21]. Specifically, we take the seven queries that are consistent first-order rewritable in [DK19, KPT13b, Dix21]. These queries feature joins between primary-key attributes and



(a) SF $= 1$



(b) SF $= 10$

Figure 6.6: Performance comparison of different CQA systems on the TPC-H benchmark with varying scale factor (SF).

foreign-key attributes, as well as projections on non-key attributes:

$$\begin{aligned}
q_1(z) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_3(\underline{y}, v, w). \\
q_2(z, w) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_2(\underline{y}, v, w). \\
q_3(z) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_2(\underline{y}, v, w), \mathsf{R}_7(\underline{v}, u, d). \\
q_4(z, d) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_2(\underline{y}, v, w), \mathsf{R}_7(\underline{v}, u, d). \\
q_5(z) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_8(\underline{y, v}, w). \\
q_6(z) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_6(\underline{t}, y, w), \mathsf{R}_9(\underline{x}, y, d). \\
q_7(z) &\text{ :- } \mathsf{R}_3(\underline{x}, y, z), \mathsf{R}_4(\underline{y}, x, w), \mathsf{R}_{10}(\underline{x}, y, d).
\end{aligned}$$

The synthetic instances are generated in two phases. In the first phase, we generate the consistent instance, followed by inconsistency injection in the second phase. We use the following parameters for data generation: (i) rSize: the number of tuples per relation, (ii) inRatio: the ratio of the number of tuples that violate primary key constraints (i.e., number of tuples that are in inconsistent blocks) to the total number of tuples of the database, and (iii) bSize: the number of inconsistent tuples in each inconsistent block.

Consistent data generation. Each relation in the consistent database has the same number of tuples, so that injecting inconsistency with the specified bSize and inRatio makes the total number of tuples in the relation equal to rSize. The data generation is *query-specific*: for each of the seven queries, the data is generated in a way to ensure the output size of the original query on the consistent database is reasonably large. To achieve this purpose, when generating the database instance for one of the seven queries, we ensure that for any two relations that join on some attributes, the number of matching tuples in each relation is approximately 25%; for the third attribute in each ternary relation that does not participate in a join, but is sometimes present in the final projection, the values are chosen uniformly from the range $[1, \mathsf{rSize}/10]$.

Inconsistency injection. In each relation, we first select a number of primary keys (or number of inconsistent blocks $\mathsf{inBlockNum}$) from the generated consistent instance. Then, for each selected primary key, the inconsistency is injected by inserting the *same number of additional tuples* ($\mathsf{bSize} - 1$) into each block. The parameter $\mathsf{inBlockNum}$ is calculated by the given $\mathsf{rSize}, \mathsf{inRatio}$ and bSize: $\mathsf{inBlockNum} = (\mathsf{inRatio} \cdot \mathsf{rSize})/\mathsf{bSize}$.

TPC-H benchmark. We also altered the 22 queries from the original TPC-H benchmark [PF00] by removing aggregation, nested subqueries and selection predicates other than constant constraints, yielding 14 simplified conjunctive queries, namely queries q'_1, q'_2, q'_3, q'_4, q'_6, q'_{10}, q'_{11}, q'_{12}, q'_{14}, q'_{16}, q'_{17}, q'_{18}, q'_{20}, q'_{21}. All of the 14 queries are in $\mathcal{C}_{\mathsf{forest}}$ and hence each query has a PPJT, meaning that they can be handled by both ConQuer and LinCQA.

We generate the inconsistent instances by injecting inconsistency into the TPC-H databases of scale factor (SF) 1 and 10 in the same way as described for the synthetic data. The only difference is that for a given consistent database instance, instead of fixing rSize for the inconsistent database, we determine the number of inconsistent tuples to be injected based on the size of the consistent database instance, the specified $\mathsf{inRatio}$ and bSize.

Table	# of rows (rSize)	inRatio	max. bSize	# of Attributes
Users	14,839,627	0%	1	14
Posts	53,086,328	0%	1	20
PostHistory	141,277,451	0.001%	4	9
Badges	40,338,942	0.58%	941	4
Votes	213,555,899	30.9%	1441	6

Table 6.2: A summary of the Stackoverflow Dataset

Stackoverflow Dataset. We obtained the `stackoverflow.com` metadata as of 02/2021 from the Stack Exchange Data Dump, with 551,271,294 rows taking up 400GB. [34] The database tables used are summarized in Table 6.2. We remark that the *Id* attributes in PostHistory, Comments, Badges, and Votes are surrogate keys and therefore not imposed as natural primary keys; instead, we properly choose composite keys as primary keys (or quasi-keys). Table 6.3 shows the five queries used in our CQA evaluation, where the number of tables joined together increases from 2 in Q_1 to 4 in Q_5.

[3]`https://archive.org/details/stackexchange`
[4]https://sedeschema.github.io/

Table 6.3: StackOverflow queries

Q_1	`SELECT DISTINCT P.id, P.title FROM Posts P, Votes V WHERE P.Id = V.PostId AND P.OwnerUserId = V.UserId AND BountyAmount > 100`
Q_2	`SELECT DISTINCT U.Id, U.DisplayName FROM Users U, Badges B WHERE U.Id = B.UserId AND B.name = "Illuminator"`
Q_3	`SELECT DISTINCT U.DisplayName FROM Users U, Posts P WHERE U.Id = P.OwnerUserId AND P.Tags LIKE "<c++>"`
Q_4	`SELECT DISTINCT U.Id, U.DisplayName FROM Users U, Posts P, Comments C WHERE C.UserId = U.Id AND C.PostId = P.Id AND P.Tags LIKE "%SQL%" AND C.Score > 5`
Q_5	`SELECT DISTINCT P.Id, P.Title FROM Posts P, PostHistory PH, Votes V, Comments C WHERE P.id = V.PostId AND P.id = PH.PostId AND P.id = C.PostId AND P.Tags LIKE "%SQL%" AND V.BountyAmount > 100 AND PH.PostHistoryTypeId = 2 AND C.score = 0`

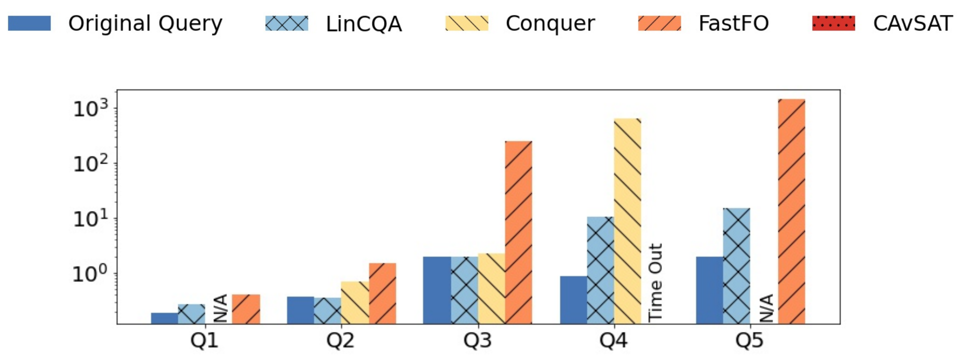


Figure 6.7: Runtime Comparison on StackOverflow

Experimental Results

In this section, we report the evaluation of LinCQA and the other CQA systems on synthetic workloads and the StackOverflow dataset. Table 4 in the technical report [FKOW22] summarizes the number of consistent and possible answers for each query in the selected datasets.

Fixed inconsistency with varying relation sizes. To compare LinCQA with other CQA systems, we evaluate all systems using both the synthetic workload and the altered TPC-H benchmark with fixed inconsistency (inRatio $= 10\%$, bSize $= 2$) as in previous works [KPT13b, DK19, Dix21]. We vary the size of each relation (rSize $\in \{500K, 1M, 5M\}$) in the synthetic data (Figure 6.5) and we evaluate on TPC-H database instances of scale factors 1 and 10 (Figure 6.6). Both figures include the time for running the original query on the inconsistent database (which returns the possible answers).

In the synthetic dataset, all three systems based on **FO**-rewriting techniques outperform

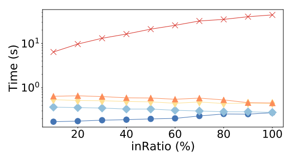
CAvSAT, often by an order of magnitude. This observation shows that if $\mathsf{CERTAINTY}(q)$ is **FO**-rewritable, a properly implemented rewriting is more efficient than the generic algorithm in practice, refuting some observations in [DK19,KPT13b]. Compared to ConQuer, LinCQA performs better or comparably on q_1 through q_4. LinCQA is also more efficient than ConQuer for q_1, q_2 and q_3. As the size of the database increases, the relative performance gap between LinCQA and ConQuer decreases for q_4. ConQuer cannot produce the SQL rewritings for queries q_5, q_6 and q_7 since they are not in $\mathcal{C}_{\mathsf{forest}}$. In summary, LinCQA is more efficient and at worst competitive to ConQuer on relatively small databases with less than $5M$ tuples, and is applicable to a wider class of acyclic queries.

In the TPC-H benchmark, the CQA systems are much closer in terms of performance. In this experiment, we observe that LinCQA almost always produces the fastest rewriting, and even when it is not, its performance is comparable to the other baselines. It is also worth noting that for most queries in the TPC-H benchmark, the overhead over running the SQL query directly is much smaller when compared to the synthetic benchmark. Note that CAvSAT times out after 1 hour for queries q'_{10} and q'_{18} for both scale 1 and 10, while the systems based on **FO**-rewriting techniques terminate. We also remark that for Boolean queries, CAvSAT will terminate at an early stage without processing the inconsistent part of the database using SAT solvers if the consistent part of the database already satisfies the query (e.g., q'_6, q'_{14}, q'_{17} in TPC-H). Overall, both LinCQA and ConQuer perform better than FastFO, since they both are better at exploiting the structure of the join tree. We also note that ConQuer and LinCQA exhibit comparable performances on most queries in TPC-H. To compute the consistent answers for a certain query, we note that the actual runtime performance heavily depends on the query plan chosen by the query optimizer in addition to the SQL rewriting given, thus we focus on the overall performance of different CQA systems rather than a few cases in which the performance difference between different systems is relatively small.

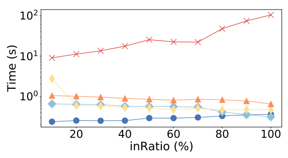
Fixed relation size with varying inconsistency. We perform experiments to observe how different CQA systems react when the inconsistency of the instance changes. Using synthetic data, we first fix rSize = 1M, $\mathsf{bSize} = 2$ and run all CQA systems on databases instances of varying inconsistency ratio from $\mathsf{inRatio} = 10\%$ to $\mathsf{inRatio} = 100\%$. The results are depicted in Figure 6.8. We observe that the running time of CAvSAT increases when the inconsistency ratio of the database instance becomes larger. This happens because the SAT formula grows with larger inconsistency, and hence the SAT solver becomes slower. In contrast, the running time of all **FO**-rewriting techniques is relatively stable across database instances of different inconsistency ratios. More interestingly, the running time of LinCQA decreases when the inconsistency ratio becomes larger. This behavior occurs because of the early pruning on the relations at lower levels of the PPJT, which shrinks the size of candidate space being considered at higher levels of the PPJT and thus reduces the overall computation time. The overall performance trends of different systems are similar for all queries and thus we present only figures of q_1, q_3, q_5, q_7 here due to the space limit.

In our next experiment, we fix the database instance size with $\mathsf{rSize} = 1\text{M}$ and inconsistency ratio with $\mathsf{inRatio} = 10\%$, running all CQA systems on databases of varying inconsistent block size bSize from 2 to 10. The results are shown in Figure 6.8. We observe that the performance of all CQA systems is not very sensitive to the change of inconsistent block sizes as shown in Figure 6.9

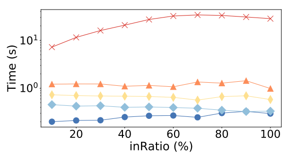
Original Query
LinCQA
Conquer
FastFO
CAvSAT



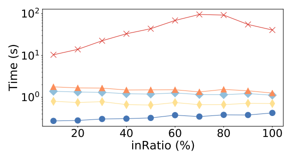
(a) q_1



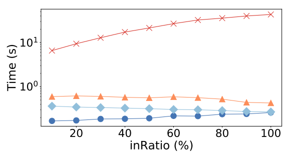
(b) q_2



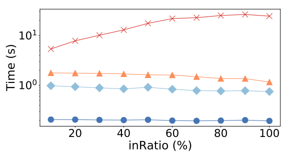
(c) q_3



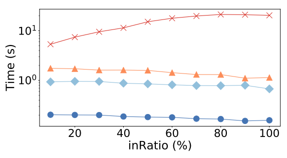
(d) q_4



(e) q_5

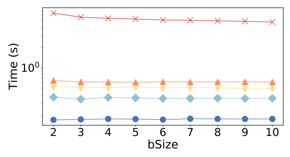


(f) q_6

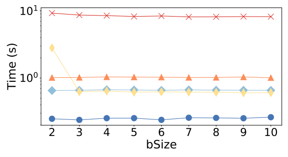


(g) q_7

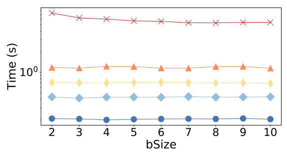
Figure 6.8: Performance of different systems on inconsistent databases with varying inconsistency ratio



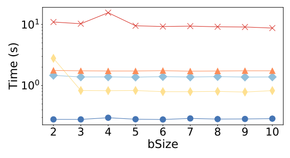
(a) q_1



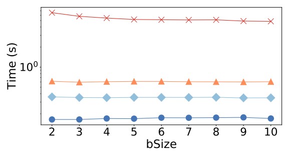
(b) q_2



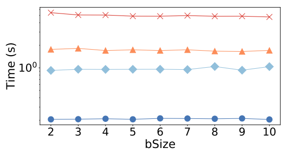
(c) q_3



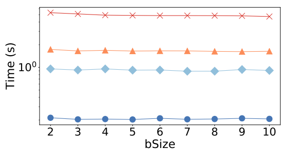
(d) q_4



(e) q_5



(f) q_6

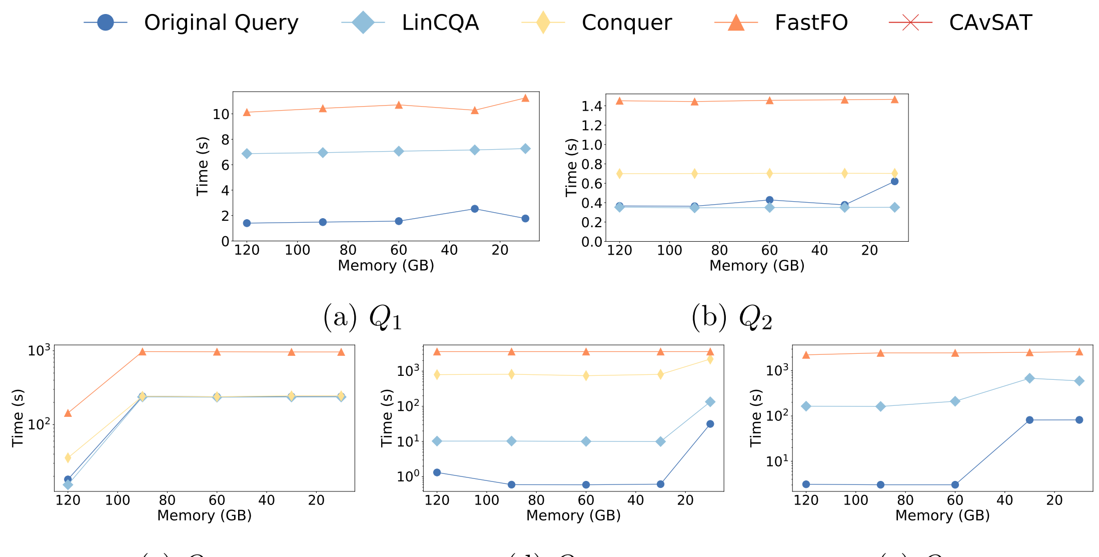


(g) q_7

Figure 6.9: Performance of different systems on inconsistent database of varying block size

StackOverflow Dataset We use a 400GB StackOverflow dataset to evaluate the performance of different systems on large-scale real-world datasets. Another motivation to use such a large dataset is that LinCQA and ConQuer exhibit comparable performance on the medium-sized synthetic and TPC-H datasets. CAvSAT is excluded since it requires extra storage for preprocessing, which is beyond the limit of the available disk space. Since Q_1 and Q_5 are not in $\mathcal{C}_{\mathsf{forest}}$, ConQuer cannot handle them and their execution times are marked as "N/A". Query executions that do not finish within an hour are marked as "Time Out". We observe that on all five queries, LinCQA significantly outperforms other competitors. In particular, when the database size is very large, LinCQA is much more scalable than ConQuer due to its more efficient strategy. We intentionally select queries with small possible answer sizes for ease of experiments and presentation. Some queries with possible answer size up to 1M would require hours to be executed and it is prohibitive to measure the performances of our baseline systems. For queries that ConQuer (Q_4) and FastFO (Q_3, Q_5) take a long time to compute, LinCQA manages to finish execution quickly due to its efficient self-pruning and pair-pruning steps.

To see the performance change of different systems when executing in small available memory, we run the experiments on a SQL server with the maximum allowed memory of 120GB, 90GB, 60GB, 30GB, and 10GB respectively. Figure 6.10 shows that, despite the memory reduction, LinCQA is still the best performer on all five queries given different amounts of available memory. No obvious performance regression is observed on Q_1 and Q_2 when reducing memory since both queries access only two tables.



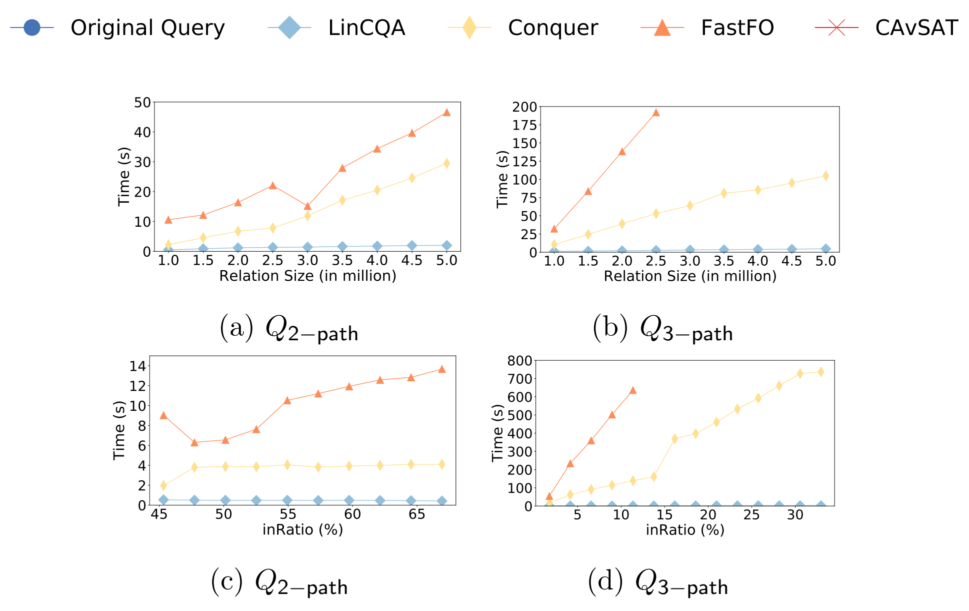
(a) Q_1 (b) Q_2

(c) Q_3 (d) Q_4 (e) Q_5

Figure 6.10: Performance of StackOverflow queries with varying amount of available memory

Summary Our experiments show that both LinCQA and ConQuer outperform FastFO and CAvSAT, systems that produce generic **FO**-rewritings and reduce to SAT respectively.

Despite LinCQA and ConQuer showing a similar performance on most queries in our experiments, we observe that LinCQA is (1) applicable to a wider class of acyclic queries than ConQuer and (2) more scalable than ConQuer when the database size increases significantly.



(a) $Q_{2-\mathsf{path}}$ (b) $Q_{3-\mathsf{path}}$

(c) $Q_{2-\mathsf{path}}$ (d) $Q_{3-\mathsf{path}}$

Figure 6.11: Performance comparison between different systems on varying relation sizes/inconsistency ratios

6.5.1 Worst-Case Study

To demonstrate the robustness and efficiency of LinCQA due to its theoretical guarantees, we generate synthetic *worst-case* inconsistent database instances for the 2-path query $Q_{2-\mathsf{path}}$ and the 3-path query $Q_{3-\mathsf{path}}$:

$$Q_{2-\mathsf{path}}(x) \text{ :- } \mathsf{R}(\underline{x}, y), \mathsf{S}(\underline{y}, z).$$
$$Q_{3-\mathsf{path}}(x) \text{ :- } \mathsf{R}(\underline{x}, y), \mathsf{S}(\underline{y}, z), \mathsf{T}(\underline{z}, w).$$

We compare the performance of LinCQA with ConQuer and FastFO on both queries. CAvSAT does not finish its execution on any instance within one hour, due to the long time required to solve the SAT formula. Thus, we do not report the time of CAvSAT. We define a generic binary relation $\mathcal{D}(x, y, N)$ as

$$\mathcal{D}(x, y, N) = ([x] \times [y]) \cup \{(u, u) \mid xy + 1 \leq u \leq N, u \in \mathbb{Z}^+\},$$

where $x, y, N \in \mathbb{Z}^+$, $[n] = \{1, 2, \ldots, n\}$ and $[a] \times [b]$ denotes the cartesian product between $[a]$ and $[b]$. To generate the input instances for $Q_{2-\mathsf{path}}$, we generate relations $\mathsf{R} = \mathcal{D}(a, b, N)$ and $\mathsf{S} = \mathcal{D}(b, c, N)$ with integer parameters a, b, c and N. For $Q_{3-\mathsf{path}}$, we additionally generate the relation $\mathsf{T} = \mathcal{D}(c, d, N)$. Intuitively, for R, $[a] \times [b]$ is the set of inconsistent tuples and $\{(u, u) \mid ab + 1 \leq u \leq N, u \in \mathbb{Z}^+\}$ is the set of consistent tuples. The values of a and b control both the number of inconsistent tuples (i.e. ab) and the size of inconsistent blocks (i.e. b). We note that $[a] \times [b]$ and $\{(u, u) \mid ab + 1 \leq u \leq N, u \in \mathbb{Z}^+\}$ are disjoint.

Fixed database inconsistency with varying size. We perform experiments to see how robust different CQA systems are when running queries on an instance of increasing size. For $Q_{2-\mathsf{path}}$, we fix $b = c = 800$, and for each $k = 0, 1, \ldots, 8$, we construct a database instance with $a = 120 + 460k$ and $N = (1 + k/2) \cdot 10^6$. By construction, each database instance has inconsistent block size $\mathsf{bSize} = b = c = 800$ in both relations R and S, and $\mathsf{inRatio} = (ab + bc)/2N = 36.8\%$, with varying relation size $\mathsf{rSize} = N$ ranging from 1M to 5M. Similarly for $Q_{3-\mathsf{path}}$, we fix $b = c = d = 120$, and for each $k = 0, 1, \ldots, 8$, we construct a database instance with $a = 120 + 180k$ and $N = (1 + k/2) \cdot 10^6$. Here, the constructed database instances have $\mathsf{inRatio} = (ab + bc + cd)/3N = 1.44\%$. As shown in Figures 6.11a and 6.11b, the performance of LinCQA is much less sensitive to changes in relation sizes compared to other CQA systems. We omit reporting the running time of FastFO for $Q_{3-\mathsf{path}}$ on relatively larger database instances in Figure 6.11b for better contrast with ConQuer and LinCQA.

Fixed database sizes with varying inconsistency. Next, we experiment on instances of varying inconsistency ratio $\mathsf{inRatio}$ in which the joining occurs mainly between inconsistent blocks of different relations. For $Q_{2-\mathsf{path}}$, we fix $b = c = 800$ and $N = 10^6$ and generate database instances for each $a = 100, 190, 280, \ldots, 1000$. All generated database instances have inconsistent block size $\mathsf{bSize} = b = c = 800$ for both relations R and S, and the size of each relation $\mathsf{rSize} = N = 10^6$ by construction. The inconsistency ratio $\mathsf{inRatio}$ varies from 36% to 72%. For $Q_{3-\mathsf{path}}$, we fix $b = c = d = 120$ and $N = 10^6$ and generate database instances with $a = 200, 800, 1400, \ldots, 8000$. The inconsistency ratio of the generated database instances varies from 1.76% to 32.96%. Figures 6.11c and 6.11d show that LinCQA is the only system whose performance is agnostic to the change in inconsistency ratio. The running time of FastFO and Conquer increases when the inconsistency of the input database increases. Similarly to the experiments varying relation sizes, the running times of FastFO for $Q_{3-\mathsf{path}}$ are omitted on relatively larger database instances in Figure 6.11d for better contrast with ConQuer and LinCQA.

6.5.2 A Case Study of LinCQA Execution on Datalog systems

Here we present a case study on the performance comparison of different Datalog systems executing LinCQA using the synthetic workload with the largest database, as shown in Figure 6.12. LinCQA-Cold represents the first run of LinCQA on SQL Server. As we can

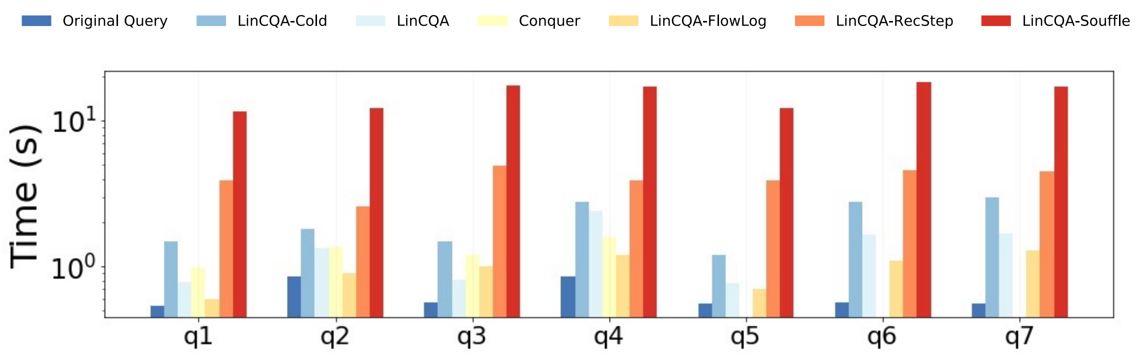


Figure 6.12: Performance comparison of different CQA systems and LinCQA execution on different Datalog systems on a synthetic workload with rSize = 5M

see, RecStep and Souffle are much less competitive compared to SQL Server, even compared to the cold run LinCQA-Cold. On the contrary, FlowLog not only matches the performance of the best performers, but also surpasses the best performance numbers in most cases, except for q_3. This observation demonstrates the efficiency of the design and implementation of FlowLog, which also shows competitive performance in the evaluation of non-recursive Datalog programs.

6.6 Summary

We introduce the notion of a pair-pruning join tree (PPJT) in this project, showing that if a BCQ has a PPJT, then $\mathsf{CERTAINTY}(q)$ is in **FO** and solvable in linear time in the size of the inconsistent database. We analyze and design efficient algorithms based on this idea expressed in Datalog rules. We further implement the corresponding system called LinCQA that is able to produce both rewriting in SQL query and Datalog rules to compute the consistent answer of q. In our experimental evaluation, we show that LinCQA produces efficient rewritings, is scalable, and robust on worst-case instances, often outperforming or matching the performance of state-of-the-art techniques for consistent query answering. By executing LinCQA in FlowLog, we show that our asynchronous dataflow-based Datalog system is able to match and even surpass the performance of the state-of-the-art relational system SQL-Server, while other existing Datalog systems are unable to achieve.

Chapter 7

Conclusion and Future Work

In this dissertation, we examine existing techniques that evaluate the Datalog programs by looking at a wide spectrum of tasks that appear in different application domains. Leveraging the expressiveness and succinct syntax of Datalog, we devise efficient algorithms for consistent query answering that outperforms the existing state-of-the-art techniques named LinCQA. To further improve the efficiency of Datalog evaluation, we present two different ways of evaluating Datalog programs: *Datalog evaluation by* RDBMS, in which a parallel single-node relational system is used combining with a spectrum of techniques for performance improvement, and *Datalog evaluation as an asynchronous dataflow execution*, in which we show that by rendering the Datalog program as the corresponding dataflow graph and executing the dataflow graph asynchronously, the unpleasant cost of synchronization observed in batch-processing systems.

Learning from our experience in building RecStep, we propose simple yet intuitive profiling components that help to better understand the behaviors of different systems in varying workloads Datalog. This better understanding guides the design and implementation of FlowLog, an asynchronous high-performance data flow-based Datalog system, which significantly outperforms existing systems in most cases. Our work motivates one to rethink the way the Datalog could be used and the way the Datalog program is evaluated and opens up new related research questions.

Future Work Related to Consistent Query Answering

An open question that is raised in our implementation and evaluation of LinCQA is whether Datalog evaluation techniques can help to perform consistent query answering (CQA) more efficiently. Complex first-order CQA rewritings give very complicated translated SQL statements consisting of a large number of nested sub-queries, which lead to difficulty of finding the optimal query plan by for backend RDBMS and result in unstable performance as observed in our study. Analyzing the performance bottleneck and performing potential optimizations could be much easier in Datalog rules since the syntax is much more succinct.

Future Work Related to FlowLog

Memory Management. While FlowLog shows better performance in most of the Datalog workloads compared to other existing Datalog engines, it fails to evaluate CC on `Twitter` due to the large memory used by the maintained intermediate states. It remains to answer whether such a limitation can be addressed by compressing the intermediate states or dropping partial states or devising a new economical way of representing the intermediate states for such cases. More generally, it is worth considering whether memory usage can be further reduced when applying FlowLog for batch processing tasks, either by constructing a more memory efficient dataflow graph or writing specialized operators.

Data Partitioning. FlowLog has not yet considered the data partition strategy seriously, the good one of which could potentially help reduce the communication overhead between dataflows residing in different workers.

Hybrid Computation. Since the underlying framework of FlowLog differential dataflow was originally designed for efficient incremental computation, it is interesting to explore the way to extend FlowLog to accommodate both batch processing and incremental computation effectively.

Distributed Computation The study of FlowLog presented in this dissertation focuses on a single-node evaluation, and different challenges could be found when considering the distributed setting - exploring efficient distributed computation of FlowLog will allow using the resource of the computation cluster consisting of a large number of nodes, which could allow the computation that FlowLog cannot currently perform in a single node due to resource restriction, such as CC evaluation on `twitter`.

Future Work Related to Profiling

Collecting and organizing a set of Datalog workloads that covers comprehensively recursive computation profiles is critical to perform a holistic evaluation of the performance of a Datalog system. One can only gain a partial view by looking at the performance numbers on a few workloads or workloads of very similar behaviors. Such a partial view could be misleading and sometimes even leads to wrong insights.

On the other hand, we see that there is still no *all-winner* Datalog system today. That is, a system (e.g. FlowLog) that outperforms other existing competitors most of the time might still fail on some other workloads. Then choices need to be made between different systems depending on the recursive computation profiles of the workloads of interest, which there is no existing way to know or estimate before actually evaluating the Datalog program. Devising a way to perform such an estimate accurately, if possible, could be very useful.

More General Future Work

The language extensions of Datalog seen recently, such as recursive aggregation, have greatly empowered Datalog to perform advanced data analytics. It would be meaningful to consider a wide spectrum of real-world data analysis tasks across different application domains and attempt to extend the semantics of Datalog to support these applications. Along with the new semantics, the corresponding techniques will be in need to perform the new operations correctly and efficiently.

LIST OF REFERENCES

[ABC99] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79. ACM Press, 1999.

[ABC03] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Answer sets for consistent query answering in inconsistent databases. *Theory Pract. Log. Program.*, 3(4-5):393–424, 2003.

[ABC$^+$11] Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Map-reduce extensions and recursive queries. In *EDBT '11*, pages 1–8, 2011.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[AK09] Arvind Arasu and Raghav Kaushik. A grammar-based entity representation framework for data cleaning. In *SIGMOD Conference*, pages 233–244. ACM, 2009.

[AMP15] Martín Abadi, Frank McSherry, and Gordon D Plotkin. Foundations of differential dataflow. In *International Conference on Foundations of Software Science and Computation Structures*, pages 71–83. Springer, 2015.

[And94] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[ATS17] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. Porting doop to soufflé: a tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 25–30. ACM, 2017.

[AU12] Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *EDBT '12*, pages 132–143, 2012.

[BBF15] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. An overview of decision support benchmarks: Tpc-ds, tpc-h and ssb. *New Contributions in Information Systems and Technologies*, pages 619–628, 2015.

[Ber19] Leopoldo E. Bertossi. Database repairs and consistent query answering: Origins and further developments. In *PODS*, pages 48–58. ACM, 2019.

[BF15] Pablo Barceló and Gaëlle Fontaine. On the data complexity of consistent query answering over graph databases. In *ICDT*, volume 31 of *LIPIcs*, pages 380–397. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[BF17] Pablo Barceló and Gaëlle Fontaine. On the data complexity of consistent query answering over graph databases. *J. Comput. Syst. Sci.*, 88:164–194, 2017.

[BFG$^+$07] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755. IEEE Computer Society, 2007.

[BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.

[BKL13a] Leopoldo Bertossi, Solmaz Kolahi, and Laks VS Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory of Computing Systems*, 52(3):441–482, 2013.

[BKL13b] Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory Comput. Syst.*, 52(3):441–482, 2013.

[BM06] David A Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February*, 38, 2006.

[BMNT15] Moria Bergman, Tova Milo, Slava Novgorodov, and Wang Chiew Tan. Query-oriented data cleaning with oracles. In *SIGMOD Conference*, pages 1199–1214. ACM, 2015.

[CCP21] Marco Calautti, Marco Console, and Andreas Pieris. Benchmarking approximate consistent query answering. In *PODS*, pages 233–246. ACM, 2021.

[CCX08] Reynold Cheng, Jinchuan Chen, and Xike Xie. Cleaning uncertain data with quality guarantees. *Proc. VLDB Endow.*, 1(1):722–735, 2008.

[CIKW16] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *SIGMOD Conference*, pages 2201–2206. ACM, 2016.

[CIP13] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469. IEEE Computer Society, 2013.

[CKE$^+$15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[Clo18] `https://www.cloudlab.us/`, 2018.

[CM05] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 197(1-2):90–121, 2005.

[CMI$^+$15] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD Conference*, pages 1247–1261. ACM, 2015.

[CMS04] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. Hippo: A system for computing consistent answers to a class of SQL queries. In *EDBT*, volume 2992 of *Lecture Notes in Computer Science*, pages 841–844. Springer, 2004.

[DB11] Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *CP*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2011.

[Dix21] Akhil Anand Dixit. *Answering Queries Over Inconsistent Databases Using SAT Solvers*. PhD thesis, UC Santa Cruz, 2021.

[DK19] Akhil A. Dixit and Phokion G. Kolaitis. A sat-based system for consistent query answering. In *SAT*, volume 11628 of *Lecture Notes in Computer Science*, pages 117–135. Springer, 2019.

[DK21] Akhil A. Dixit and Phokion G. Kolaitis. Consistent answers of aggregation queries using SAT solvers. *CoRR*, abs/2103.03314, 2021.

[EEI$^+$13] Amr Ebaid, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. NADEEF: A generalized data cleaning system. *Proc. VLDB Endow.*, 6(12):1218–1221, 2013.

[FFM05] Ariel Fuxman, Elham Fazli, and Renée J Miller. Conquer: Efficient management of inconsistent databases. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 155–166, 2005.

[FGG$^+$18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445, 2018.

[FKMP03] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, pages 207–224, 2003.

[FKOW22] Zhiwei Fan, Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Lincqa: Faster consistent query answering with linear time guarantees. *arXiv preprint arXiv:2208.12339*, 2022.

[FM07] Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci.*, 73(4):610–635, 2007.

[FMK22] Zhiwei Fan, Sunil Mallireddy, and Paraschos Koutris. Towards better understanding of the performance and design of datalog systems. 2022.

[FZZ+18] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh Patel. Scaling-Up In-Memory Datalog Processing: Observations and Techniques. *arXiv e-prints*, page arXiv:1812.03975, December 2018.

[GAK12] Todd J Green, Molham Aref, and Grigoris Karvounarakis. Logicblox, platform and language: A tutorial. In *Datalog in Academia and Industry*, pages 1–8. Springer, 2012.

[GGM+21] Congcong Ge, Yunjun Gao, Xiaoye Miao, Bin Yao, and Haobo Wang. A hybrid data cleaning framework using markov logic networks (extended abstract). In *ICDE*, pages 2344–2345. IEEE, 2021.

[GGZ03] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. Knowl. Data Eng.*, 15(6):1389–1408, 2003.

[GMPS13] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. The LLUNATIC data-cleaning framework. *Proc. VLDB Endow.*, 6(9):625–636, 2013.

[GTg] `http://www.cse.psu.edu/~kxm85/software/GTgraph`.

[H+18] Bhole Rahul Hiraman et al. A study of apache kafka in big data stream processing. In *2018 International Conference on Information, Communication, Engineering and Technology (ICICET)*, pages 1–3. IEEE, 2018.

[HCG+18] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek R. Narasayya, and Surajit Chaudhuri. Transform-data-by-example (TDE): an extensible search engine for data transformations. *Proc. VLDB Endow.*, 11(10):1165–1177, 2018.

[HD15] Minyang Han and Khuzaima Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015.

[HdAC$^+$14] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In *SIGMOD '14*, pages 881–884, 2014.

[IGM20] Muhammad Imran, Gábor E Gévay, and Volker Markl. Distributed graph analytics with datalog queries in flink. In *Software Foundations for Data Interoperability and Large Scale Graph Data Analytics*, pages 70–83. Springer, 2020.

[IGQRM22] Muhammad Imran, Gábor E Gévay, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. Fast datalog evaluation for batch and stream graph processing. *World Wide Web*, 25(2):971–1003, 2022.

[JSS16] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[JSZS19] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard 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, page 31–40, New York, NY, USA, 2019. Association for Computing Machinery.

[KDPV10] Yannis Katsis, Alin Deutsch, Yannis Papakonstantinou, and Vasilis Vassalos. Inconsistency resolution in online databases. In *ICDE*, pages 1205–1208. IEEE Computer Society, 2010.

[KIJ$^+$15] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. Bigdansing: A system for big data cleansing. In *SIGMOD Conference*, pages 1215–1230. ACM, 2015.

[KJL$^+$20] Aziz Amezian El Khalfioui, Jonathan Joertz, Dorian Labeeuw, Gaëtan Staquet, and Jef Wijsen. Optimization of answer set programs for consistent query answering by means of first-order rewriting. In *CIKM*, pages 25–34. ACM, 2020.

[KKD$^+$20] Lara A Kahale, Assem M Khamis, Batoul Diab, Yaping Chang, Luciane Cruz Lopes, Arnav Agarwal, Ling Li, Reem A Mustafa, Serge Koujanian, Reem Waziry, et al. Meta-analyses proved inconsistent in how missing data were handled across their included primary trials: A methodological survey. *Clinical Epidemiology*, 12:527–535, 2020.

[KKN03] Rajasekar Krishnamurthy, Raghav Kaushik, and Jeffrey F Naughton. Xml-to-sql query translation literature: The state of the art and open problems. In *International XML Database Symposium*, pages 1–18. Springer, 2003.

[KL21] Henning Kohler and Sebastian Link. Possibilistic data cleaning. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[KLW$^+$20] Bojan Karlas, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. Nearest neighbor classifiers over incomplete information: From certain answers to certain predictions. *Proc. VLDB Endow.*, 14(3):255–267, 2020.

[KOW21] Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Consistent query answering for primary keys on path queries. In *PODS*, pages 215–232. ACM, 2021.

[KP12] Phokion G. Kolaitis and Enela Pema. A dichotomy in the complexity of consistent query answering for queries with two atoms. *Inf. Process. Lett.*, 112(3):77–85, 2012.

[KPT13a] Phokion G. Kolaitis, Enela Pema, and Wang-Chiew Tan. Efficient querying of inconsistent databases with binary integer programming. *Proc. VLDB Endow.*, 6(6):397–408, 2013.

[KPT13b] Phokion G. Kolaitis, Enela Pema, and Wang-Chiew Tan. Efficient querying of inconsistent databases with binary integer programming. *Proc. VLDB Endow.*, 6(6):397–408, 2013.

[KS14] Paraschos Koutris and Dan Suciu. A dichotomy on the complexity of consistent query answering for atoms with simple keys. In *ICDT*, pages 165–176. OpenProceedings.org, 2014.

[KW15] Paraschos Koutris and Jef Wijsen. The data complexity of consistent query answering for self-join-free conjunctive queries under primary key constraints. In *PODS*, pages 17–29. ACM, 2015.

[KW17] Paraschos Koutris and Jef Wijsen. Consistent query answering for self-join-free conjunctive queries under primary key constraints. *ACM Trans. Database Syst.*, 42(2):9:1–9:45, 2017.

[KW18] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys and conjunctive queries with negated atoms. In *PODS*, pages 209–224. ACM, 2018.

[KW19] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys in logspace. In *ICDT*, volume 127 of *LIPIcs*, pages 23:1–23:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[KW20] Paraschos Koutris and Jef Wijsen. First-order rewritability in consistent query answering with respect to multiple keys. In *PODS*, pages 113–129. ACM, 2020.

[KW21] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys in datalog. *Theory Comput. Syst.*, 65(1):122–178, 2021.

[KWW$^+$16] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proc. VLDB Endow.*, 9(12):948–959, 2016.

[LB07] Andrei Lopatenko and Leopoldo E. Bertossi. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *ICDT*, volume 4353 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2007.

[LCG$^+$06] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 97–108, 2006.

[Lef92] Alexandre Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases. In *FGCS*, pages 915–925, 1992.

[Len02] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246, 2002.

[LRB$^+$21] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. Cleanml: A study for evaluating the impact of data cleaning on ML classification tasks. In *ICDE*, pages 13–24. IEEE, 2021.

[MAB$^+$10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[MB05] Mónica Caniupán Marileo and Leopoldo E. Bertossi. Optimizing repair programs for consistent query answering. In *SCCC*, pages 3–12. IEEE Computer Society, 2005.

[Mcs22a] Frank Mcsherry. Differential dataflow. `https://github.com/TimelyDataflow/differential-dataflow`, 2022.

[Mcs22b] Frank Mcsherry. Timely dataflow. `https://github.com/TimelyDataflow/timely-dataflow`, 2022.

[Mic19] Microsoft. SQL Server 2019. `https://www.microsoft.com/en-us/sql-server/sql-server-2019`, 2019.

[Mic22a] Microsoft. User-defined functions. `https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions?view=sql-server-ver16`, 2022.

[Mic22b] Microsoft. With common_table_expression (transact-sql) - sql server. `https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql?view=sql-server-ver16`, 2022.

[Mil13] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, volume 2324, 2013.

[MLSR18] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *arXiv preprint arXiv:1812.02639*, 2018.

[MMI$^+$13] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[MMII13] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[MNP$^+$14] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *AAAI '14*, pages 129–137, 2014.

[MRT15] Marco Manna, Francesco Ricca, and Giorgio Terracina. Taming primary key violations to query large inconsistent data via ASP. *Theory Pract. Log. Program.*, 15(4-5):696–710, 2015.

[OOCR09] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2009.

[PDZ$^+$18] Jignesh M Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A data platform based on the scaling-up approach. *Proceedings of the VLDB Endowment*, 11(6):663–676, 2018.

[PF00] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[PSC$^+$15] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J. Miller, and Divesh Srivastava. Combining quantitative and logical data cleaning. *Proc. VLDB Endow.*, 9(4):300–311, 2015.

[RB19] Leonid Ryzhyk and Mihai Budiu. Differential datalog. *Datalog*, 2:4–5, 2019.

[RBM13] M. Andrea Rodríguez, Leopoldo E. Bertossi, and Mónica Caniupán Marileo. Consistent query answering under spatial semantic constraints. *Inf. Syst.*, 38(2):244–263, 2013.

[RCIR17] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, 2017.

[RD00] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[Rep97] Thomas W. Reps. Program analysis via graph reachability. In Jan Maluszynski, editor, *ILPS*, pages 5–19. MIT Press, 1997.

[ROA$^+$21] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. Horizon: Scalable dependency-driven data cleaning. *Proc. VLDB Endow.*, 14(11):2546–2554, 2021.

[SGL13] Jiwon Seo, Stephen Guo, and Monica S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 278–289, 2013.

[SJSW16] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 196–206, New York, NY, USA, 2016. Association for Computing Machinery.

[SKHS12] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In *Datalog 2.0*, pages 165–176, 2012.

[SPSL13] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013.

[STS$^+$19] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813. IEEE, 2019.

[SVKW15] Bernhard Scholz, Kostyantyn Vorobyov, Padmanabhan Krishnan, and Till Westmann. A datalog source-to-source translator for static program analysis: An experience report. In *2015 24th Australasian Software Engineering Conference*, pages 28–37. IEEE, 2015.

[SYI$^+$16] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1135–1149, 2016.

[TCZ$^+$14] Yongxin Tong, Caleb Chen Cao, Chen Jason Zhang, Yatao Li, and Lei Chen. Crowdcleaner: Data cleaning for multi-version data on the web via crowdsourcing. In *ICDE*, pages 1182–1185. IEEE Computer Society, 2014.

[TSJ$^+$09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 97–118, Berlin, Heidelberg, 2005. Springer-Verlag.

[WBH15] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

[WHZ$^+$17] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 389–404, New York, NY, USA, 2017. ACM.

[Wij10] Jef Wijsen. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In *PODS*, pages 179–190. ACM, 2010.

[Wij12] Jef Wijsen. Certain conjunctive query answering in first-order logic. *ACM Trans. Database Syst.*, 37(2):9:1–9:35, 2012.

[Wij19] Jef Wijsen. Foundations of query answering on inconsistent databases. *SIGMOD Rec.*, 48(3):6–16, 2019.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In William Pugh and Craig Chambers, editors, *PLDI*, pages 131–144. ACM, 2004.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

[Yan17] Mohan Yang. *Declarative languages and scalable systems for graph analytics and knowledge discovery.* University of California, Los Angeles, 2017.

[YSZ17] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. Scaling up the performance of more powerful datalog systems on multicore machines. *The VLDB Journal*, 26(2):229–248, 2017.

[ZAC$^+$19] Qizhen Zhang, Akash Acharya, Hongzhi Chen, Simran Arora, Ang Chen, Vincent Liu, and Boon Thau Loo. Optimizing declarative graph queries at large scale. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1411–1428, 2019.

[ZXW$^+$16] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[ZYI$^+$18] Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. Declarative bigdata algorithms via aggregates and relational database dependencies. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018.*, 2018.