Large-Scale Provenance for Soufflé

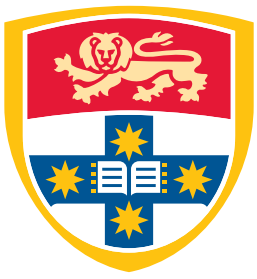
DAVID ZHAO
SID: 440153399

Supervisor: Dr. Bernhard Scholz

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Information Technology (Honours)

School of Information Technologies
The University of Sydney
Australia

7 November 2017



THE UNIVERSITY OF
SYDNEY

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: David Zhao

Signature: **Date**:

Abstract

Logic programming languages, such as Datalog, have seen a rise in popularity in recent years, now being widely used to answer questions about real world problems. This popularity stems from the wide applicability of logic as a domain specific language for a variety of problems including static program analysis, declarative networking and security analysis. The use of logic reduces the complexity of implementing applications as programs are written in a declarative fashion. In other words, instead of describing computational steps imperatively, computations are concisely specified by their intended result. As a consequence, however, the lack of computational steps makes debugging very challenging, and there is no clear debugging strategy in logic programming. In the field of database research, *provenance* has been introduced as a way of explaining the output of relations, and can also be applied to Datalog. The big challenge with large real-world problems, however, is that the relations may contain billions of tuples, rendering existing provenance approaches infeasible. Hence, there is still a gap to design and implement scalable and reusable provenance systems for high-performance open-source Datalog engines, such as Soufflé.

In this thesis, we develop the theory of provenance in the form of a *proof tree* for a given tuple. However, the construction of the proof tree in a naïve fashion is too expensive for large datasets. To ensure scalability, we push computation from logic evaluation time to proof construction time, achieving an efficient space/time trade-off. This lazy evaluation approach produces an annotated intensional database, which can be queried after evaluation by an arbitrary number of provenance queries without the need for recomputing the intensional database. We conduct experiments with complex industrial-strength benchmarks, including DOOP with DaCapo, which produce hundreds of millions of output tuples. We demonstrate that our novel provenance approach incurs a runtime and memory consumption overhead of 1.5× on average. Thus, it can cope with large datasets, where existing techniques and a naïve implementation become infeasible.

Acknowledgements

First of all, a gigantic thanks to my wonderful supervisor, Bernhard. Thank you for all your patience and helpfulness, and all your great ideas. It was a fun year with a great topic, and I've learnt a lot. Also a massive thanks to my co-supervisor, Paul. Even from across the other side of the world, you've been an invaluable help to me throughout this year.

To the rest of the programming languages research group, thanks for accepting me as a member and teaching me a lot of background. The Friday meetings have all been fun and informative, and I'm glad to have taken part in the group.

I'd also like to thank Veronica, Abdul, Vincey, Sampson and Lexi for helping me proofread this thesis, and I hope it was an interesting read!

To my family, my girlfriend Veronica, and my other friends Abdul, Cecilia, Hisham, Vincey, Nancy, Judy, Erny, and others, thanks for supporting me throughout this year. You've all been great fun to hang out with, and it's made this year all the more enjoyable.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

Logic programming languages including Datalog-like languages have seen a rise in popularity in recent years, being widely used to answer questions about real world problems such as program analysis (Jordan et al., 2016; Allen et al., 2015), declarative networking (Zhou et al., 2010; Huang et al., 2011), security analysis (Ou et al., 2005) and business applications (Aref et al., 2015). Logic programming provides declarative semantics for programs, resulting in succinct program representations and rapid-prototyping capabilities for these real-world applications. Rather than specifying the computational steps imperatively, logic programs specify the intended result logically, and thus are able to express computation in a more concise manner. For instance, logic programming has gained traction in the area of program analysis due to its flexibility in building custom program analysers (Jordan et al., 2016), and has been used in practice to build analysers for Java programs (Bravenboer and Smaragdakis, 2009).

However, the declarative semantics of Datalog result in a challenge for debugging. Strategies employed in debugging imperative programs, such as inspecting variables at certain points in time, do not translate to declarative programming. Logic programs lack the notions of variables or time, and instead only allow views of full relations without any explanation of the origin or derivation of data. Thus, these outputs are inconclusive in a debugging context.

In the database research community, the concept of data provenance has been introduced as a way of providing explanations of the origins of data (Buneman et al., 2001; Cheney et al., 2009). Data provenance is described as a way to connect output data to its derivation from input data. In Datalog programs, this relates closely to the idea of *proof trees*. A proof tree for a tuple provides a complete explanation of how the tuple is derived from input facts and rules. In this thesis, we focus on methods to generate proof trees for tuples of a Datalog program. These proof trees can be used as means for debugging, as they provide an explanation of how output data is derived. However, there is a considerable challenge with the large-scale data of real-world problems, such as DOOP, which may produce hundreds of millions of

output tuples for certain analyses. Existing approaches to provenance become infeasible at these large sizes.

Datalog provides two major notions of logic evaluation: (1) top-down evaluation and (2) bottom-up evaluation. Top-down evaluation asserts a goal by applying rules until eventually reaching facts, and intrinsically produces a proof tree as a consequence of evaluation. Bottom-up evaluation asserts a goal starting from facts and applying rules. Its evaluation is related to fix-point semantics and does not require an explicit notion of proof trees. Due to the large scale data involved in real-world problems, modern Datalog engines (Jordan et al., 2016; Aref et al., 2015; Whaley et al., 2005) use bottom-up evaluation as a default strategy. However, while providing better run-time performance, unlike top-down methods, bottom-up evaluation does not compute proof trees. As a consequence, existing approaches for computing data provenance (Köhler et al., 2012; Deutch et al., 2015; Lee et al., 2017) must store extra provenance information during evaluation, resulting in high runtime and memory overhead. To overcome this "provenance bottleneck" for large scale data in real-world problems, a new approach is required.

In this thesis, we introduce "guided SLD", a novel hybrid approach for computing provenance information for Datalog by producing proof trees with minimal height. Our lazy evaluation approach ensures scalability by pushing computation from logic evaluation time to proof construction time, utilising a hybrid of bottom-up and top-down evaluation strategies. The initial logic evaluation is performed using a bottom-up strategy, and results in an intensional database (IDB) annotated with rule numbers and proof tree heights, which can then be queried by any number of arbitrary provenance queries, without any need to recompute the IDB. The provenance queries are then answered by constructing proof trees in a top-down fashion utilising the annotations in the IDB. A brief description of our provenance system is provided in Figure 1.1.

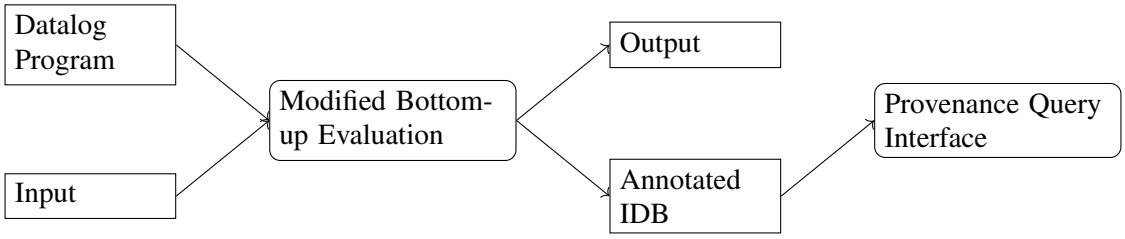


FIGURE 1.1: Diagram of guided SLD provenance system

1.1 Motivation and Running Example

The main motivation for this work is to build a high-performance debugging system for Soufflé. Soufflé is a high-performance open-source Datalog engine that synthesises highly parallel C++ programs from logic programs. It has been deployed for several large-scale applications including security analysis (Scholz et al., 2015), DOOP (Antoniadis et al., 2017), parallelising compilers, declarative networking, and smart-contract analysis.

The use case of such a debugging system can be demonstrated by the following example. If a Datalog program contains a logic error resulting in some unwanted tuple being outputted, then a provenance system could tell us which input and which rules were involved in generating that tuple, and thus pointing the programmer towards the error.

We demonstrate the usage of our system with a running concrete example. Consider the following Datalog program, which computes the transitive closure of a graph:

$$r_1 : path(x, y) \leftarrow edge(x, y)$$
$$r_2 : path(x, z) \leftarrow edge(x, y), path(y, z)$$

FIGURE 1.2: Datalog program for transitive closure

This program computes the relation *path*, using the two rules shown. The first rule implies that any tuple in the *edge* relation is also in the *path* relation. The second rule implies the transitivity property, i.e., if there is a tuple (x, y) in the *edge* relation and there is a tuple (y, z) in the *path* relation, then there is the tuple (x, z) in the *path* relation.

Let's assume that following facts are provided as input:

$$edge(1, 2),\ edge(2, 3),\ edge(3, 1)$$

that denote edges in the graph. The graph is depicted in Figure 1.3.

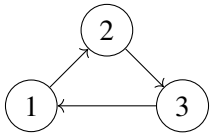


FIGURE 1.3: Graph of example program

For this example, the tuple $path(1, 3)$ would be generated. Our provenance system can then be used to query the provenance of this tuple. The result would be a valid proof tree as shown in Figure 1.4.

$$\dfrac{edge(1,2) \qquad \dfrac{edge(2,3)}{path(2,3)}\,(r_1)}{path(1,3)}\,(r_2)$$

FIGURE 1.4: Proof tree for $path(1, 3)$

1.2 Contributions

In this thesis, we present the following contributions:

- We have developed a theoretical framework that connects proof trees and tuples via a homomorphism. This framework is used to show two main results: (1) the correctness of our provenance system, and (2) that it produces proof trees of minimal height.
- We have designed a novel provenance system for Datalog, focusing on scalability and reusability. For this new provenance system, we use a hybrid of bottom-up and top-down evaluation techniques. The key idea of our approach is to shift computation from logic evaluation time to proof construction time. An initial bottom-up evaluation of Datalog produces an annotated IDB which can then be efficiently queried for provenance multiple times. Thus, our approach is reusable for multiple provenance queries without needing to recompute the IDB, a major advantage over current systems (Deutch et al., 2015; Lee et al., 2017) which require re-evaluation of Datalog for each provenance query.
- We have implemented our provenance system in Soufflé. The implementation required an extension to the synthesis procedure of Soufflé consisting of the production of annotations for the IDB, and the use of these annotations in answering provenance queries. This current implementation consists of 2,500 lines of code, and has been contributed to the open-source implementation of Soufflé, publicly accessible here (gui, 2017). In addition, we implemented a naïve approach for provenance in order to establish a baseline for performance, which was achieved by a pure Datalog-to-Datalog transformation.
- We have conducted extensive experiments using large-scale problems including DOOP and other complex benchmarks. We measure overheads in runtime and memory, demonstrating a

cost of 1.5× overhead for storing provenance information. We also compare our system with a naïve approach and the state-of-the-art.

1.3 Organisation

This thesis is organised as follows: In Chapter 2, we provide background definitions and survey related work. In Chapter 3, we present the theoretical framework of proof trees, and delve into the details of both a naïve approach and guided SLD. In Chapter 4, we detail the implementation of our approaches in Soufflé. In Chapter 5, we present empirical experiments conducted to demonstrate the feasibility of guided SLD, and discuss their results. We draw our conclusions and discuss future work in Chapter 6.

CHAPTER 2

Background

In this chapter, we review the syntax and semantics of Datalog, as well as past and current research in the area of data provenance. We look at the definitions of provenance, and the practical developments that rely on these definitions. We include some applications of the research into provenance.

2.1 Datalog

Datalog is a declarative programming language based on the logic programming paradigm. It is widely used in database query systems, offering a big advantage over relational calculus and relational algebra in the form of recursion (Greco and Molinaro, 2015).

2.1.1 Syntax and Semantics of Datalog

A Datalog program P consists of a finite set of *rules*. A rule is a Horn clause of the form

$$r_i : R(X) :- R_1(X_1), \ldots, R_n(X_n).$$

Each R_j is a *relation name*, and each X_j is a sequence of variables and constants of correct arity. Each $R_j(X_j)$ is a *predicate*. $R(X)$ is the *head* of the rule r_i, and $R_1(X_1), \ldots, R_n(X_n)$ is the *body* of r_i.

The semantic meaning of such a rule is as follows: if each predicate in the body of the rule holds, then the head of the rule holds. Negated predicates in the body are also allowed, and in this thesis, we use the semantics of *semi-positive* Datalog. This means that negated predicates must have an EDB relation, and a negated predicate holds if the positive version does not hold.

A concrete instantiation of a predicate $R(X)$, with variables replaced by appropriate constants, is denoted as a *fact* or a *tuple*, $R(u)$. We say that a relation is extensional (or in the EDB) if no predicates

with that relation occur in the head of any rule, or intensional (in the IDB) otherwise. A tuple is EDB or IDB if the associated relation is EDB or IDB respectively.

An *instance* of a Datalog program P is a set of tuples with relations from P, and is denoted I. An instance is EDB if all tuples contained are EDB, or IDB otherwise.

2.1.2 Evaluation of Datalog

There are two main approaches that are widely used in evaluating a Datalog program: bottom-up and top-down evaluation. Bottom-up evaluation is used in modern Datalog systems such as Soufflé (Jordan et al., 2016) and LogicBlox (Aref et al., 2015), while top-down evaluation is employed by older systems such as XSB (Sagonas et al., 1993).

2.1.2.1 Bottom-Up Evaluation

Bottom-up evaluation of a Datalog program P describes a general method of starting from the EDB tuples to generate all IDB tuples. The process starts from an instance I of P, consisting only of EDB facts. Then, we define an *immediate consequence* of I to be a fact $R(u)$ such that either $R(u) \in I$, or $R(u) \leftarrow R_1(u_1), \ldots, R_n(u_n)$ is a valid instantiation of a rule with each $R_i(u_i) \in I$. We then define the *immediate consequence operator*, Γ_P, to I as follows:

$$\Gamma_P : inst(P) \rightarrow inst(P)$$

$$\Gamma_P(I) = \{A : A \text{ is an immediate consequence of } I\}$$

It can be seen that Γ_P is monotone, and using *Tarski's Fixpoint Theorem* (Tarski, 1955), we can show that there exists a minimum fixpoint of Γ_P (Abiteboul et al., 1995). Thus, we can apply Γ_P to the input instance repeatedly, until we find a fixpoint. The resulting fixpoint is denoted the *model* of P given I, or $P(I)$, and is the final result of bottom-up evaluation.

An example of bottom-up evaluation is given in Figure 2.1

$$
\begin{aligned}
I &: edge(1,2), edge(2,3) \\
\Gamma_P(I) &: edge(1,2), edge(2,3), path(1,2), path(2,3) \\
\Gamma_P^2(I) &: edge(1,2), edge(2,3), path(1,2), path(2,3), path(1,3)
\end{aligned}
$$

FIGURE 2.1: Example bottom-up evaluation of transitive closure program (Figure 1.2), where I is the input instance

This process can therefore be seen as constructing an instance of the program, starting from the EDB and applying rules to compute new facts until no more new facts can be computed. A simple implementation of this process is known as *naïve evaluation*, and an improved version is called *seminaïve evaluation*.

Bottom-up evaluation is presented in more detail in (Abiteboul et al., 1995; Greco and Molinaro, 2015; Ceri et al., 1989).

2.1.2.2 Top-Down Evaluation

Top-down evaluation describes the opposite method of starting from a query, and checking in the program whether there are rules and facts that make the query satisfiable. The query takes the form of a *goal clause*, which is a sequence of predicates:

$$\leftarrow A_1, \ldots, A_n$$

We consider each predicate A_i in the goal clause, and search for some rule with A_i as the head. We may need to substitute constants for variables so they match (a process known as unification), then replace A_i in the goal clause with the body of that rule. We can apply this step repeatedly until either we reach EDB facts, in which case we show that the original goal clause holds, or we fail to find a valid instantiation at some point, in which case the original goal clause does not hold.

An example of top-down evaluation is given in Figure 2.2

$$\leftarrow path(1,3)$$
$$\leftarrow edge(1,y), path(y,3)$$
$$\leftarrow edge(1,2), path(2,3)$$
$$\leftarrow edge(1,2), edge(2,3)$$
$$\leftarrow \square$$

FIGURE 2.2: Example top-down evaluation of transitive closure program (Figure 1.2) showing $path(1,3)$ holds

However, top-down evaluation requires the notion of backtracking since it is unspecified which rule will be selected. In the case where it select a rule which cannot generate an instantiation of A_i, backtracking may be needed for it to try a different rule. In addition, top-down evaluation has no guarantees for termination in its most basic form. This can be illustrated by the case where the program contains a rule $A \leftarrow A$, where top-down evaluation may try to replace a predicate A with itself and thus never terminating.

An example of top-down evaluation is SLD resolution, which defines a mechanical method of performing top-down evaluation. In SLD resolution, the replacement step is repeatedly applied to some goal clause, until it is either shown to hold or all attempts lead to failure, in which case it does not hold.

This top-down process is detailed further in (Abiteboul et al., 1995; Greco and Molinaro, 2015; Ceri et al., 1989).

2.1.2.3 Magic Sets

In addition to these two main approaches, there exists methods of evaluating Datalog which combine bottom-up and top-down evaluation. One such method is magic sets, as presented in (Beeri and Ramakrishnan, 1987). The general concept is to optimise bottom-up evaluation for certain queries by rewriting rules based on *adornments*. These adornments provide labels for free or bound variables, and are generated in a top-down fashion. Ullman (Ullman, 1989) showed that by using a magic set transformation, we can guarantee that bottom-up evaluation never performs worse than top-down. The advantage of this appears with queries with many constants, where bottom-up must still compute full relations, while top-down can answer the query by only computing a subset.

This approach of combining bottom-up and top-down evaluation inspires our hybrid approach to computing provenance information.

2.2 Definition and Semantics of Provenance

Data provenance describes the path that data takes through a system. Provenance information allows us to trace the origins and history of data, and such information is useful for many data management tasks. Cheney et al. (Cheney et al., 2009) argue that the necessity of provenance systems is driven by the ease of creating and storing data, and provenance helps ensure the integrity of such data. In particular, large database systems such as data warehouses benefit from the capabilities of a provenance system. In this database domain, provenance is defined in (Deutch et al., 2015) as "an explanation of the ways [facts] are derived". This provenance information can be used to provide valuable information for debugging, as described in (Köhler et al., 2012). In the Datalog world, which this thesis is focused on, provenance is closely related to *proof trees*, an approach used in practice by (Arora et al., 1993).

However, there are different approaches to defining the semantics of provenance. (Cheney et al., 2009) presents three of the possible semantics: *why*, *how*, and *where*-provenance.

2.2.1 *Why*-provenance

Why-provenance describes a relationship between the output tuples of a Datalog program with input tuples. Intuitively, why-provenance links an output tuple with the input tuples that produce it.

More precisely, (Cheney et al., 2009) defines the why-provenance of an output tuple t to be a set of proofs called the *witness basis*. Each proof is a set of input tuples I', called a *witness*, such that the program will produce t if given I' as input. (Buneman et al., 2001) shows that the set of tuples in the witness basis corresponds to the leaves of a proof tree for t.

2.2.2 *How*-provenance

How-provenance can be seen as a generalisation of why-provenance. In order to describe its semantics, (Green et al., 2007) applies the concept of the abstract algebraic construction of semirings [1]. (Green et al., 2007) takes the approach of labeling each tuple by an element in the semiring.

This definition of provenance is further refined and compared to why-provenance in (Cheney et al., 2009), and a practical provenance system based on these semantics is described in (Deutch et al., 2014).

Formally, this approach defines a semiring $(K, +, \cdot, 0, 1)$, and a function mapping from a set of tuples to K:

$$f : \{R(u)\} \to K$$

This function provides the annotation of a tuple by a semiring element. Then, the provenance of an output tuple $R(u)$ is defined as the sum of the product of the leaves of each possible derivation tree deriving $R(u)$, where the leaves are the annotations of input tuples.

$$\sum_{\text{proof trees } \tau \text{ deriving } R(u)} \left(\prod_{\text{tuples } R(u)' \in \text{leaves of } \tau} f(R(u)) \right)$$

This structure shows a correspondence between provenance infomation and proof trees, where the provenance polynomial describes the leaves of the proof trees and how the proof trees are related. However, it does not describe the internal structure of a proof tree, and thus there is not enough information to reconstruct a full proof for a tuple.

By relating tuples to elements of an algebraic structure, (Green et al., 2007) shows a correspondence between database provenance and other database systems, such as probablistic databases and bag semantics. Here, tuples are labelled by elements in some semiring (in probablistic databases, this is the set $[0, 1]$, and in bag semantics, this is the set $\mathbb{N}$). However, while such semantics make sense in a relational algebra system, the recursion of Datalog presents an issue for using finite polynomials to represent provenance. As a solution, (Green et al., 2007) defines provenance for recursive queries using formal power series, which are useful in theory but cumbersome in practice.

[1] A semiring is a mathematical object $(K, +, \cdot, 0, 1)$ defined as a set of elements K with two operators, $+$ and $\cdot$. 0 and 1 are identity elements for $+$ and $\cdot$ respectively.

Other applications for these provenance semantics include using boolean circuits to store the elements of the semiring in a more space efficient manner, as presented by (Deutch et al., 2014).

We can see that how-provenance is more descriptive than why-provenance, in that if one is given the how-provenance of a program, it is easy to construct the why-provenance (Cheney et al., 2009). In this thesis, we focus on a more general form of how-provenance, namely we deal with proof trees directly.

2.2.3 *Where*-provenance

This definition takes a different approach, describing provenance for each element of a tuple rather than the full tuple itself. (Buneman et al., 2001) states that it is closely connected to why-provenance, in that the where-provenance of any value in an output tuple is a subset of the witnesses of that tuple. (Buneman et al., 2001) defines where-provenance as a *derivation basis*, which contains all the pieces of input data contributing to the output.

(Buneman et al., 2001) provides a constructive definition for where-provenance for relational queries in normal form. (Cheney et al., 2009) extends this definition to general relational queries, providing a direct definition in terms of the relational calculus operations that make up the query.

Where-provenance provides interesting semantics for data provenance, however its use in a debugging context may be limited. While the provenance for a full tuple can easily be linked to its derivation from program rules, the provenance of single elements of tuples may be more difficult to describe in terms of these rules. Thus, where-provenance semantics are used more prominently in annotation management systems, such as (Bhagwat et al., 2004)

Examples of applications of where-provenance for relational systems include *Perm* (Glavic and Alonso, 2009; Glavic et al., 2013) and *pSQL* (Bhagwat et al., 2004).

2.3 Datalog Provenance Representations

In this section, we focus on practical provenance systems for Datalog. There are multiple approaches to achieve this. One such method is by rewriting the Datalog program to capture provenance information, as in (Köhler et al., 2012; Deutch et al., 2015). Another method is to use a secondary data structure in order to store the provenance information (Deutch et al., 2014).

2.3.1 Datalog rewriting

By rewriting Datalog, we can capture information such as rule firings and a provenance graph (Köhler et al., 2012). One such method is to instrument a Datalog program to store a "provenance graph". Their method of doing so, however, ventures into the world of second order logic, utilising Statelog (Lausen et al., 1998) in order to encode provenance information by using states. Köler et al. also show an equivalent method by only using Datalog, however the disadvantage is a more complex encoding.

The weakness of this approach is that it stores the full provenance information, which may be prohibitively large for large Datalog instances. This is reflected in the experiments of (Köhler et al., 2012), in that the largest experiment presented was a transitive closure example with 1710 nodes and 3936 edges, containing 304,000 paths. This suggests that the full provenance storage approach does not scale well to very large databases containing millions of output tuples.

(Köhler et al., 2012), however, demonstrates how this provenance system can be used for profiling logic programs, in addition to debugging. They are able to use their full provenance system in order to compare various alternative rewritings of the transitive closure program, in terms of the number of intermediate tuples, rule firings and other statistics. Thus, while full provenance encodings are prohibitive for debugging large programs, they may have other uses in profiling. On the other hand, Datalog engines such as Soufflé (Jordan et al., 2016) contain profiling features without a need to store full provenance information.

In order to avoid having to store full provenance information, Deutch et al. (Deutch et al., 2015) introduce the notion of selective provenance. In this approach, the concept of a *derivation tree pattern* is introduced as a specification language for what provenance information to produce. A derivation tree pattern describes a specification which any generated proof trees must match. This is used to instrument the Datalog program to store relevant information for these patterns, such that proof trees generated from the extra information match the derivation tree pattern.

The main disadvantage of this strategy is that the program needs to be rewritten and reevaluated for each new provenance query. However, the flexibility of derivation tree patterns means that the instrumented Datalog itself can be more efficient for certain small patterns. In experiments in (Deutch et al., 2015), it is shown that selective provenance information can be computed in reasonable time for databases with 1.7M output tuples. However, full provenance information for this instance is still prohibitively costly, taking over 6.5 hours on a standard desktop computer.

2.3.2 External data structures

A memory efficient method of annotating tuples to record provenance information can be achieved by using *boolean circuits*, as shown in (Deutch et al., 2014). A boolean circuit essentially represents a boolean formula in a compact manner by being able to compress repetitive parts of the formula together. This property is utilised in (Deutch et al., 2014) to produce a concise representation of Datalog provenance using boolean circuits.

This representation improves on previous methods of representing Datalog provenance, which were shown to "incur a super-polynomial size blowup in data complexity" (Deutch et al., 2014). The circuit representation presented by Deutch et al., however, leads to a representation polynomial in the size of the input database.

Being closely related to boolean formulae, these circuit representations can also be used to compute semiring annotations, as in how-provenance. We can construct a semiring using the operations of boolean formulae, $(K, \vee, \wedge, false, true)$, demonstrating the relationship between circuit representations and how-provenance. However, these circuit data structures are rather complex and difficult to understand, and thus, no implementation or query mechanism has been presented.

2.3.3 Network provenance

An application of data provenance is network provenance. In this context, the aim is to trace data through a network graph, rather than through a relational query. This problem relates closely to data provenance, since we can view a query as a graph of operations. In the context of network provenance, a network-oriented variant of Datalog, called *NDlog*, is used to specify the networks.

(Zhou et al., 2010) proposes a scheme for storing a network provenance graph in a distributed data structure. In doing so, numerous optimisations for maintaining and querying the distributed data are investigated, such as caching query results and compressing the provenance store itself.

While data provenance isn't traditionally associated with a distributed setting, the caching and compression of provenance information may be helpful in optimising data provenance systems.

2.4 Efficient provenance storage

We have seen that provenance in a database system corresponds closely with the proof trees of an output tuples. A common theme with storage of provenance information involves optimising the structure of these trees to make storage more efficient.

A proof tree may contain a number of redundant structures in neighbouring nodes, allowing them to be combined into a single rule. Bao et al. presents two rules in (Bao et al., 2012) to accomplish this. In addition, the structure of a proof tree in a relational system is highly correlated with the structure of the query itself. Thus, Bao et al. considers the trade off of storing either the query operation, or the actual tuples in each node of the proof tree. A dynamic programming algorithm is presented to 'materialize' the nodes from query operations to tuples in the most space efficient manner.

This work, however, omits the trade off of querying the provenance information, which may be more costly with the reductions proposed.

Proof trees may also contain properties of repeated similar patterns, leading to the transformations in (Chapman et al., 2008) to reduce the impact of these similar patterns. Chapman et al. introduce the ideas of factorization and inheritance to prevent similar patterns from occuring more than once, and instead simply using pointers to the same copy of the pattern wherever needed. After these transformations are applied to the provenance store, Chapman et al. show that querying of provenance information is still efficient.

These schemes for reducing the size of storage for provenance information are presented in the context of relational queries. Hence, they make no mention of the recursive structures possible in Datalog. In addition, these schemes to compress full provenance data do not explore the possibility of storing partial provenance data in the first place.

2.5 Selective Provenance

There are a few approaches when it comes to querying provenance information. One such approach is to store full provenance information, and query this after evaluation. Another is to only compute and store provenance information depending on a query given before evaluation.

2.5.1 Storing full provenance

One method of querying provenance is to store all possible provenance information during evaluation, then provenance queries simply return some subset of that information.

Karvounarakis et al. introduce a new provenance query language called *ProQL* in (Karvounarakis et al., 2010). The result of a query in ProQL is a subgraph of the provenance graph, matching a pattern defined by the syntax of ProQL. To facilitate efficient querying, schemes for storage, processing and indexing of provenance data are also proposed.

An extension to ProQL is the notion of ranking for output tuples. Ives et al. propose a system of ranking output tuples of a query based on their provenance in (Ives et al., 2012). This is a similar problem to link analysis for web pages, and algorithms such as Google's PageRank. This application of provenance information would be useful for platforms with sharing and recommendations.

ProQL, however, was built for relational systems, and so no notion of recursion is considered.

2.5.2 Storing queried provenance

The other method of querying provenance is to only store a subset of the provenance information. In this scheme, the provenance query is given to the system prior to evaluation of the actual query itself, and only the provenance information relevant to the provenance query is stored.

One such system is called *Perm*, introduced by Glavic et al. in (Glavic and Alonso, 2009; Glavic et al., 2013). In this system, relational queries are transformed to store an extra relation which records provenance information. It can be specified which parts of the query to transform and store provenance information for, leading to only storing a subset of provenance relevant to the query.

However, this system is purely based on relational databases, with the provenance information itself being stored as a relational table.

A similar approach can be taken for Datalog provenance, including recursion. This is shown in (Deutch et al., 2015). The notion of derivation tree patterns are introduced, and the rewriting of Datalog is done in terms of the derivation tree pattern. Thus, the querying portion is performed before computation of the original Datalog program, meaning that only relevant provenance information is stored. In addition

to this, a notion of ranking the output provenance trees in terms of relevance is introduced, and Deutch et al. present an algorithm to return the top-k derivation trees for some output tuple.

CHAPTER 3

Provenance Construction

In this chapter, we detail our approach to computing proof trees for Datalog programs. We begin with background and definitions of proof trees, and show an equivalence between them and tuples in the context of Datalog evaluation. We then present a naïve method of computing proof trees, which we compare to an improved guided SLD method.

3.1 Proof Trees

We begin by formalising the theory of proof trees and how they relate to bottom-up Datalog evaluation. Evaluating a Datalog program in a bottom-up fashion results in a set of IDB tuples. However, this output is given without any explanation for how the tuples are derived. An explanation for the derivation of a tuple may take the form of a *proof tree*. A proof tree for an IDB tuple $R(u)$ formally describes the derivation for $R(u)$ in terms of the EDB tuples and rules. A formal definition follows.

DEFINITION (Proof Tree). *A proof tree $T_{R(u)}$ for a tuple $R(u)$ is a tree where each node n is annotated with a tuple $R_n(u_n)$. The leaves are annotated with EDB facts, and each internal node is annotated with an IDB tuple. The root node is annotated with $R(u)$. Then, the children $n_1, \ldots, n_k$ of each node n are ordered such that $R_n(u_n) :- R_{n_1}(u_{n_1}), \ldots, R_{n_k}(u_{n_k})$ is a valid instantiation of some rule in P.*

We aim to demonstrate an equivalence between proof trees and tuples, and so we require a notion of associating a proof tree for a tuple with the tuple itself. Thus, we define an operator t as

$$t(T_{R(u)}) = R(u) \text{ such that } R(u) \text{ is at the root of } T_{R(u)}$$

Intuitively, t maps a proof tree $T_{R(u)}$ to $R(u)$ itself.

3.1.1 Proof Tree Model of Datalog

In this thesis, we introduce a method of computing a proof tree for some tuple $R(u)$. Firstly, we need to ensure that this problem is well defined, and that a proof tree always exists for any tuple. To do this, we develop a bottom-up semantics for Datalog based on proof trees. The aim is to show an equivalence between proof trees and tuples, which we illustrate in Figure 3.1.

$$path(1,2) \qquad \frac{edge(1,2)}{path(1,2)}\,(r_1)$$

$$path(2,3) \qquad \frac{edge(2,3)}{path(2,3)}\,(r_1)$$

$$path(1,3) \qquad \frac{edge(1,2) \qquad \dfrac{edge(2,3)}{path(2,3)}\,(r_1)}{path(1,3)}\,(r_2)$$

FIGURE 3.1: Some tuples and their proof trees for the Datalog example in Figure 1.2

We base our definitions on the standard fixpoint semantics for Datalog (Abiteboul et al., 1995; Greco and Molinaro, 2015). Using these standard fixpoint semantics, an input instance I is a set of tuples, and the output of a Datalog program P given I is defined to be the model $P(I)$. This model is usually defined as a fixpoint of the immediate consequence operator, Γ_P (see Section 2.1.2.1 for more details).

However, we introduce a new semantics for Datalog by defining a *tree instance* and a *tree model*, which deal with *proof trees* rather than tuples. A tree instance is intuitively a set of proof trees for tuples computed by P, and a tree model is the output of P given an input tree instance. Formally, we define a tree instance as follows:

DEFINITION. *A tree instance, I^T is a set of proof trees for tuples computed by P. I^T is input if for all $T_{R(u)} \in I^T$, the corresponding tuple $R(u)$ is EDB.*

We say that a tuple $R(u) \in I^T$ if there exists some $T_{R(u)} \in I^T$ such that $t(T_{R(u)}) = R(u)$.

We can also define the set of all tree instances of a program, $treeinst(P)$, which forms a lattice under the ordering given by subsets.

In order to define a tree model, we need an analogue of the immediate consequence operator, Γ_P, but for tree instances. Given a tree consequence operator, we can define a tree model to be a fixpoint of this operator:

DEFINITION. *Let P be a Datalog program, I^T be a tree instance of P. Define the tree consequence operator, $\mathcal{T}_P$ as follows:*

- *If $T_{R(u)} \in I^T$, then $T_{R(u)} \in \mathcal{T}_P(I^T)$*
- *Assume $R(u) :- R_1(u_1), \ldots, R_k(u_k)$ is an instantiation of a rule in P where for each positive $R_i(u_i)$, there exists a proof tree for $R_i(u_i)$, $T_{R_i(u_i)} \in I^T$, and for each negative $R_i(u_i)$, the associated relation is EDB and $R_i(u_i) \notin I$. Then $T_{R(u)} \in \mathcal{T}_P(I^T)$, where $T_{R(u)}$ is the proof tree formed by having $R(u)$ at the root, and each $T_{R_i(u_i)}$ as a subtree.*

Note that this definition assumes the negation semantics of semi-positive Datalog, in other words, negated predicates must have an EDB relation. Also note that there may be multiple proof trees for a single tuple, if there are multiple different instantiations for rules which produce that tuple. For example, in Figure 3.2, a tuple $path(1,2)$ may have two different proof trees.

$$\frac{edge(1,2)}{path(1,2)}\,(r_1) \qquad\qquad \frac{edge(1,2) \qquad \dfrac{\cdots}{path(2,2)}}{path(1,2)}\,(r_2)$$

FIGURE 3.2: Possible proof trees for $path(1,2)$

We see that $\mathcal{T}_P$ is a lattice function, which maps a tree instance to another tree instance. In order to define a tree model, we must show that a fixpoint for the tree consequence operator exists, which we can do by applying Tarski's fixpoint theorem (Tarski, 1955). However, a requirement of this theorem is that $\mathcal{T}_P$ is monotone. So, we show that this is the case, under the assumption that the input instance of the program is constant. In other words, we show that if $I_1^T \subseteq I_2^T$, then $\mathcal{T}_P(I_1^T) \subseteq \mathcal{T}_P(I_2^T)$

PROOF. Let P be a Datalog program, and I_1^T and I_2^T be tree instances of P with the same EDB tuples, such that $I_1^T \subseteq I_2^T$. Then, if $T_{R(u)} \in \mathcal{T}_P(I_1^T)$, then either:

- $T_{R(u)} \in I_1^T$, in which case it must also be in I_2^T, so it must be in $\mathcal{T}_P(I_2^T)$, or

- There exists an instantiation of a rule with all positive body atoms in I_1^T, and all negative body atoms not in the EDB. Then, all positive body atoms must be in I_2^T, and since the EDB tuples are equal, the resulting $T_{R(u)} \in \mathcal{T}_P(I_2^T)$.

□

Now, we have that $\mathcal{T}_P$ is a monotone function on the lattice of tree instances. Thus, we can apply Tarski's fixpoint theorem in order to show that there exists a unique minimal fixpoint of $\mathcal{T}_P$. We define the tree model $V_P(I^T)$ to be this minimal fixpoint.

However, it is crucial to show that the result of our tree model is equivalent to the original semantics of the program. So, we must show that the tree model of a Datalog program is equivalent to the standard fixpoint semantics, in other words, that every tuple in $P(I)$ has a proof tree in $V_P(I^T)$ and vice versa.

We show that $V_P(I^T)$ is equivalent to the fixpoint semantics, $P(I)$, by showing a homomorphism between the set of tree instances $treeinst(P)$ and instances $inst(P)$. Define an operator α which maps from $treeinst(P)$ to $inst(P)$, defined by

$$\alpha(I^T) = \left\{ t(T_{R(u)}) \mid T_{R(u)} \in I^T \right\}$$

We show that α is a homomorphism. Let I^T be a tree instance. We want to show that $\alpha(\mathcal{T}_P(I^T)) = \Gamma_P(\alpha(I^T))$.

PROOF. We prove this by a double inclusion:

- $\alpha(\mathcal{T}_P(I^T)) \subseteq \Gamma_P(\alpha(I^T))$: Let $R(u)$ be a tuple in $\alpha(\mathcal{T}_P(I^T))$. Then, there must be a proof tree for $R(u)$, denoted $T_{R(u)} \in \mathcal{T}_P(I^T)$. There are two cases:
  - $T_{R(u)} \in I^T$, in which case $t(T_{R(u)}) = R(u) \in \alpha(I^T)$, which implies $R(u) \in \Gamma_P(\alpha(I^T))$.
  - $R(u) :- R_1(u_1), \ldots, R_n(u_n)$ is an instantiation of a rule in P, with each positive $R_i(u_i)$ having a proof tree $T_{R_i(u_i)} \in I^T$. Then, each positive $R_i(u_i) \in \alpha(I^T)$, and so $R(u) \in \Gamma_P(\alpha(I^T))$, by definition of Γ_P.

  Hence, $\alpha(\mathcal{T}_P(I^T)) \subseteq \Gamma_P(\alpha(I^T))$
- $\alpha(\mathcal{T}_P(I^T)) \supseteq \Gamma_P(\alpha(I^T))$: Let $R(u)$ be a tuple in $\Gamma_P(\alpha(I^T))$. Then, we have two cases
  - $R(u) \in \alpha(I^T)$, in which case there exists a proof tree for $R(u)$, denoted $T_{R(u)} \in I^T$, which implies $T_{R(u)} \in \mathcal{T}_P(I^T)$, and so $R(u) \in \alpha(\mathcal{T}_P(I^T))$

– $R(u) :- R_1(u_1), \ldots, R_n(u_n)$ is an instantiation of a rule in P, with each positive $R_i(u_i) \in \alpha(I^T)$. Then, for each positive $R_i(u_i)$, there exists a proof tree $T_{R_i(u_i)} \in I^T$, and so there exists a $T_{R(u)} \in \mathcal{T}_P(I^T)$. Thus, $t(T_{R(u)}) = R(u) \in \alpha(\mathcal{T}_P(I^T))$

This proves that α is a homomorphism. □

We illustrate this homomorphism structure with the following commutative diagram

$$\begin{array}{ccc} I^T & \xrightarrow{\alpha} & I \\ {\scriptstyle \mathcal{T}_P}\downarrow & & \downarrow{\scriptstyle \Gamma_P} \\ \mathcal{T}_P(I^T) & \xrightarrow{\alpha} & \Gamma_P(I) \end{array}$$

It remains to be proved that the tree model and the model defined by the immediate consequence operator are equivalent. In other words, we want to show that the minimal fixpoints of $\mathcal{T}_P$ and Γ_P are equivalent.

PROOF. Let $P(I)$ be the model of program P given instance I. Let I^T be such that $\alpha(I^T) = P(I)$. Then, we see that

$$\begin{aligned} \alpha(\mathcal{T}_P(I^T)) &= \Gamma_P(\alpha(I^T)) \\ &= \alpha(I^T) \\ &= P(I) \end{aligned}$$

since $P(I)$ is a fixpoint of Γ_P. If $\mathcal{T}_P(I^T)$ is not a fixpoint of $\mathcal{T}_P$, we can repeatedly apply $\mathcal{T}_P$, since $\alpha(\mathcal{T}_P^k(I^T)) = P(I)$ for all $k \geq 1$ by the above argument. There must be some l such that $\mathcal{T}_P^l(I^T)$ is a minimal fixpoint for $\mathcal{T}_P$, and $\alpha(\mathcal{T}_P^l(I^T)) = P(I)$. □

The result of this is that we introduce an alternative semantics for Datalog based on proof trees, and we show that it is equivalent to the standard fixpoint semantics. These preliminaries will become crucial in the next chapters, where we use these constructive definitions in order to design algorithms to construct proof trees.

3.2 Problem Statement for Proof Tree Construction

Given a Datalog program P, an input instance I, and a provenance query tuple $R(u) \in P(I)$, we want to find any valid $T_{R(u)}$ for $R(u)$.

We can find such a tree recursively as follows. We have defined a tree model of Datalog as a fixpoint of a tree consequence operator. In each application of this operator, if we find a new tuple $R(u)$, we also construct a new proof tree $T_{R(u)}$. This tree is rooted at $R(u)$ with the proof trees for $R_1(u_1), \ldots, R_k(u_k)$ being subtrees of the root, where $R_1(u_1), \ldots, R_k(u_k)$ generate $R(u)$ via some rule in P. If we can find for any given $R(u)$, the tuples $R_1(u_1), \ldots, R_k(u_k)$ used to produce $R(u)$, and do this recursively for each $R_i(u_i)$, then we can generate a valid proof tree $T_{R(u)}$.

Thus, the original problem reduces to the following. Given a Datalog program P, an input instance I, and a tuple $R(u) \in P(I)$, we want to find a rule r and a set of tuples $\{R_1(u_1), \ldots, R_k(u_k)\}$ such that they generate $R(u)$ via r. We call the set $sub(R(u)) = \{R_1(u_1), \ldots, R_k(u_k)\}$ a subproof for $R(u)$.

3.3 Naïve Encoding

The naïve encoding method is a simple approach to generating provenance information, based on storing subproofs directly. We present this approach as a baseline to use for comparison with other improved approaches.

In the naïve encoding method, for each tuple $R(u)$, we directly record a subproof for $R(u)$. More specifically, we store for each tuple $R(u)$ an associated rule $r_{R(u)}$ and a set of tuples $sub(R(u))$. We achieve this by instrumenting the Datalog program to store the required information.

3.3.1 Datalog Instrumentation

Our Datalog instrumentation algorithm produces a Datalog program which directly stores a subproof for each tuple, and is as follows:

Algorithm 1 Naïve Encoding Instrumentation

Input: Datalog program P

1: Let P' be the modified program
2: **for** rule $r_i \in P$ **do**
3: Let $R(X)$ be head of r
4: Let $R_1(X_1), \ldots, R_k(X_k)$ be body of r
5: Create a new relation R^i with arity equal to the arity of R plus the sum of the arities of each R_i
6: Create a new rule $R^i(X, X_1, \ldots, X_k) :- R_1(X_1), \ldots, R_k(X_k)$, and add it to P
7: Create a new rule $R(X) :- R^i(X, \_, \ldots, \_)$
8: **end for**

The result of this algorithm is a set of new relations R^i which store a subproof for each tuple of R. These relations R^i store the original tuple X, along with its subproof $X_1, \ldots, X_k$, providing the direct storage of subproofs for each tuple, which we utilise during proof tree construction. We then project the original tuple X back into the relation R, which can then be used in other rules.

For example, the result of running Algorithm 1 on the example program in Figure 1.2 is:

$$r_1 : path_1(x, y, x, y) :- edge(x, y)$$

$$r_2 : path_2(x, z, x, y, y, z) :- edge(x, y), path(y, z)$$

$$r_3 : path(x, y) :- path_1(x, y, \_, \_)$$

$$r_4 : path(x, y) :- path_2(x, y, \_, \_, \_, \_)$$

We must show the correctness of this algorithm. The resulting Datalog program correctly stores subproofs for each tuple, ensured by the semantics of Datalog. In addition, the instrumented program will terminate, since it has the same semantics as the original Datalog program.

3.3.2 Proof Tree Construction

Given the result of Algorithm 1, we can form a proof tree given a provenance query $R(u)$ as shown

Algorithm 2 Building Proof Tree via Naïve Encoding

Input: Instrumented Datalog program P', instance I' of P', tuple $R(u)$

```
1: return CREATETREE(R(u))
2: function CREATETREE(R(u))
3:     Let T_{R(u)} be a tree
4:     Let R(u) be the root of T_{R(u)}
5:     if R(u) is EDB then
6:         return T_{R(u)}
7:     end if
8:     Let R^i(u, u_1, ..., u_k) be a tuple of R^i corresponding to R(u)
9:     for each u_j do
10:        Let R_j be the corresponding relation
11:        Add CREATETREE(R_j(u_j)) as a child of R(u) in T_{R(u)}
12:    end for
13:    return T_{R(u)}
14: end function
```

We can see that we get a valid proof tree for $R(u)$, as the instrumented Datalog program correctly generates subproofs for each tuple. However, this algorithm is not necessarily optimal, as we have no guarantee about the size of the generated proof tree. As seen in Figure 3.2, there may be multiple different proof trees for a single tuple. In the algorithms above, it is unspecified which derivation will be produced, and in some cases, the proof tree may even be infinite in height.

In addition, the instrumented Datalog for naïve encoding must store a large amount of extra information. Due to the fact that we record every possible derivation of a tuple, each tuple of relation R in the original program P will now be stored up to $k + l$ times, where R is the head of k rules in P, and R appears in the body of l rules in P. This leads to a linear overhead in the size of the model computed for P.

Hence, the size of data generated by the instrumented program is a factor of $O(ks)$ larger than the data generated by the original program, where k is the number of rules, and s is the size of each rule. This is a prohibitive overhead for large Datalog instances, as demonstrated in the experiments in Chapter 5.

3.4 Guided SLD

Our guided SLD approach is a more efficient method of generating provenance information. Rather than directly recording a subproof for each tuple, we store a constant amount of information for each tuple, which allows us to construct a proof tree lazily. Namely, we store the rule that generates the tuple, and the height of a proof tree for the tuple. The main idea is to push computation from Datalog evaluation

time to proof tree construction time, when answers for provenance queries are generated. This allows for faster evaluation of Datalog, and on demand construction of proof trees, without the need to re-evaluate the Datalog program.

In particular, for each tuple $R(u)$, we associate it with the rule used to produce it, as well as a number representing the height of a proof tree for $R(u)$. To do this, we instrument the Datalog program to store the extra information, as in Algorithm 3.

3.4.1 Datalog Instrumentation

We present our algorithm to instrument Datalog to store a rule number and proof tree height for each tuple $R(u)$.

Algorithm 3 Guided SLD Instrumentation

Input: Datalog program P
Output: Modified Datalog program P'

for rule $r_i \in P$ **do**
  if $r_i = R(X)$ is a fact **then**
    Change r_i to become $R(X, r_0, 0)$
  else
    Let r_i be $R(X) :- R_1(X_1), \ldots, R_k(X_k)$
    Change head $R(X)$ to become $R(X, r_i, \max(l_1, \ldots, l_k) + 1)$ where $l_i = 0$ if $R_i(X_i)$ is negative
    for body predicate $R_j(X_j)$ **do**
      if $R_j(X_j)$ is negative **then**
        Change $R_j(X_j)$ to become $R_j(X_j, \_, \_)$
      else
        Change $R_j(X_j)$ to become $R_j(X_j, \_, l_j)$
      end if
    end for
  end if
end for

In this algorithm, the r_i in each predicate denotes the unique rule number, and the l_j denotes the height of a proof tree for that tuple. We alternatively call l_j the level in a proof tree for the tuple. We can see that all EDB facts are defined as having height 0, and an IDB tuple $R(u)$ has height equal to the maximum of the heights of the tuples generating $R(u)$, plus 1. Intuitively, this is because a proof tree for $R(u)$ is formed by $R(u)$ at the root, with the proof trees for the tuples generating $R(u)$ as subtrees, hence the total height being the maximum height of these plus 1.

Path tuple	Rule number	Proof tree height
$(1, 2)$	r_1	1
$(2, 3)$	r_1	1
$(3, 1)$	r_1	1
$(1, 3)$	r_2	2
$(2, 1)$	r_2	2
$(3, 2)$	r_2	2
$(1, 1)$	r_2	3
$(2, 2)$	r_2	3
$(3, 3)$	r_2	3

TABLE 3.1: The resulting annotated IDB of guided SLD provenance encoding with Datalog program from Figure 1.2

For example, given the example program in Figure 1.2, the modified program is:

$$r_1 : path(x, y, r_1, \max(l_1) + 1) :- edge(x, y, \_, l_1)$$

$$r_2 : path(x, z, r_2, \max(l_1, l_2) + 1) :- edge(x, y, \_, l_1), path(y, z, \_, l_2)$$

However, if this modified program P' is executed directly, it may not terminate, since each tuple may have an infinite number of possible proof heights. For example, consider the program in Figure 1.2 modelling the paths in a graph. If we have a cyclic input graph, a single path may have infinitely many possible proof trees, each with a different height, as illustrated in Figure 3.2. P' would try to store each of these different heights as a different tuple. This means that each application of the immediate consequence operator would always generate new tuples, and thus evaluation of P' is not guaranteed to terminate.

To avoid this issue, we must also modify the semantics of Datalog execution, such that the set enforcement only considers the original tuple. This ensures that each tuple $R(u)$ is added only the first time it is seen during execution and subsequently generating the same tuple will not cause a new tuple to be added, regardless of whether the rule number or level number is different.

The result of evaluating the instrumented program using these modified Datalog semantics is an IDB annotated with rule numbers and proof tree heights, illustrated in Table 3.1.

3.4.2 Proof Tree Construction

As with the naïve encoding method, we build a proof tree by recursively finding subproofs for each tuple. The algorithm is as follows:

Algorithm 4 Building Proof Tree via Guided SLD

Input: Instance I', program P', tuple $R(u)$

Find a tuple $R(u, i, l) \in I'$ where u matches $R(u)$
return BUILDTREE($R(u)$, i, l)
function BUILDTREE(tuple $R(u)$, rule num i, level l)
    Let r_i be $R(X, i, \max(l_1, \ldots, l_k) + 1) :- R_1(X_1, \_, l_1), \ldots, R_k(X_k, \_, l_k)$
    Let $T_{R(u)}$ be a proof tree
    Let $R(u)$ be the root of $T_{R(u)}$
    if $R(u)$ is EDB **then**
        return $T_{R(u)}$
    end if
    for negative body predicates $R_i(X_i, \_, l_i)$ **do**
        Add the negative tuple corresponding to $R_i(X_i)$ as a child of $R(u)$ in $T_{R(u)}$
    end for
    for $1 \leq j \leq k$ **do**
        Find a tuple $R_j(u_j, r_{u_j}, l_{u_j}) \in I'$ matching predicate $R_j(X_j, \_, l_j)$, such that $l_{u_j} < l$
        Add BUILDTREE($R_j(u_j)$, r_{u_j}, l_{u_j}) as a child of $R(u)$ in $T_{R(u)}$
    end for
    return $T_{R(u)}$
end function

As an example, say we want to find the proof tree for $path(1, 3)$. Then, we begin by finding the correct rule and level numbers, resulting in the full tuple $path(1, 3, r_2, 2)$. Recall that the rule r_2 is

$$path(x, z) :- edge(x, y), path(y, z).$$

We then search for tuples matching the body of the rule, with strictly lower level. This finds the tuples $edge(1, 2, r_0, 0)$ and $path(2, 3, r_1, 1)$. This results in the first level of the proof tree being

$$\frac{edge(1,2) \qquad path(2,3)}{path(1,3)}\,(r_2)$$

Since $edge(1, 2, r_0, 0)$ is associated with rule r_0, we know it is EDB and so it remains as a leaf. We then search for a subproof for $path(2, 3, r_1, 1)$, finding $edge(2, 3, r_0, 0)$. This results in the full proof tree:

$$\frac{edge(1,2) \qquad \dfrac{edge(2,3)}{path(2,3)}\,(r_1)}{path(1,3)}\,(r_2)$$

It is important to show that this algorithm terminates. Since each recursive call results in a strictly decreasing level l, and the base case is when $l = 0$, there must be a finite number of recursive calls, and hence the algorithm must terminate.

The efficiency of Algorithm 4 depends on which data structures are used. In lines 1 and 14 we attempt to find a tuple matching some conditions. This depends on the data structures used to store the instance of the Datalog program, and hence the efficiency of proof tree construction depends on these data structures.

3.4.3 Minimal Proof Tree Height

This algorithm, as before, returns a valid proof tree since we simulate the tree consequence operator using recursion. However, an important result the proof tree returned by guided SLD is always a minimal height proof tree for $R(u)$.

First we show that if we have a tuple $R(u, i, l) \in I'$, then l equals the number of applications of the immediate consequence operator Γ_P to generate $R(u, i, l)$.

PROOF. The proof is by induction on the level l:

- $l = 0$ if and only if the tuple is EDB, if and only if it is produced by 0 applications of Γ_P
- $l > 0$. Assume true for all $k < l$. Now, $R(u, i, l)$ must be IDB, since otherwise it would have level 0. So, $R(u, i, l) :- R_1(u_1, i_1, l_1), \ldots, R_k(u_k, i_k, l_k)$ is a valid instantiation of some rule, where $l = \max(l_1, \ldots, l_k)+1$. Assume without loss of generality that $l_1 = \max(l_1, \ldots, l_k) = l-1$. Then, each of the body tuples are generated in at most l_1 applications of Γ_P, and applying Γ_P generates $R(u, i, l)$.

□

We now show that l is the minimal height for tuple $R(u, i, l)$, in other words there is no $l' < l$ such that $R(u, i, l') \in P'(I)$.

PROOF. The proof is by contradiction. Assume $R(u, i, l) \in P'(I)$, and there is an $l' < l$ such that $R(u, i, l') \in P'(I)$. Then, $R(u, i, l')$ is produced in fewer applications than $R(u, i, l)$, and so will have been added to the model earlier in evaluation. Since our semantics for evaluation specify that a tuple will

be added only when $R(u)$ is first encountered, then $R(u, i, l') \in P'(I)$ and $R(u, i, l) \notin P'(I)$. Hence, a contradiction, so there is no such $l' < l$. □

It is clear to see that the proof tree produced for a tuple $R(u, i, l)$ has height at most l, since each step in Algorithm 4 searches for a subproof with height strictly less than the tuple currently being considered. Thus, the recursion depth is at most l, and each recursive step produces one level of the proof tree.

Therefore, given some tuple $R(u, i, l)$, we produce a proof tree for $R(u)$ with minimal height. This is an important result, since we can guarantee optimality of our solution.

Note, however, that the proof tree for a tuple $R(u)$ produced by guided SLD may not contain the same tuples used to derive $R(u)$ during bottom-up evaluation. The result will still be a valid, minimal height proof tree, but due to the approach of searching through the generated instance, it may not result in the same construction as bottom-up evaluation.

3.4.4 Efficiency and Advantage of Guided SLD

The name guided SLD comes from SLD resolution, the top-down Datalog evaluation strategy. The proof tree algorithm for guided SLD is similar to SLD resolution, in that the main idea is that we start from a provenance query tuple, and repeatedly attempt to find subproofs for this tuple. However, guided SLD makes use of an already populated Datalog instance as well as rule and height information. The main advantage of this is that the already populated Datalog instance guarantees the existence of subproofs for every tuple, thus eliminating the need for backtracking as in standard SLD resolution. The extra information forms the 'guided' part of the name, and allows this approach to be far more efficient than standard top-down evaluation.

Thus, guided SLD is a hybrid approach, with the guiding information generated by bottom-up evaluation, and proof tree construction a top-down approach using the guided information. This hybrid approach is novel to the best of our knowledge, and is the basis for superior performance compared to existing methods.

In terms of the storage overhead, each tuple produced in the original program now stores two extra values: the rule number and a level number. This amounts to a constant factor overhead. Hence, the size of data generated by the instrumented program is a factor of $O(1)$ larger than that of the original program.

CHAPTER 4

Implementation in Soufflé

Both provenance approaches were implemented as proof of concepts using Soufflé as the Datalog evaluation engine. Soufflé is an open-source, compilation based Datalog engine (sou, 2017), which synthesises high-performance parallel C++ code from Datalog. It utilises a bottom-up evaluation strategy, and techniques such as Futamura projections, partial evaluation, staged compilation and automated parallelisation in order to maximise performance. In this chapter, we present details of implementing our provenance approaches as extensions to Soufflé. These provenance systems can be accessed via a flag in Soufflé, namely `-T` for naïve provenance, and `-t` for guided SLD.

4.1 Naïve Encoding

Our implementation of the Datalog instrumentation for naïve encoding, Algorithm 1, is a Datalog transformation in Soufflé, taking the original Datalog program as input and producing the instrumented Datalog program as output. The implementation included 1800 lines of code, and is publicly available at (nai, 2017).

In implementing this transformation, we utilise records to store subproofs. These records are a convenient structure in Soufflé which allow us to easily provide unique labels for subproofs. A record is a sequence of values, for example $path([1, 2])$ denotes a path relation using records rather than individual values. This allows easier storage of subproofs, simplifying the implementation of the naïve approach, with a slight performance cost.

Path tuple	Subproof	Rule
$[1, 2]$	$[1, 2]$	r_1
$[2, 3]$	$[2, 3]$	r_1
$[3, 1]$	$[3, 1]$	r_1
$[1, 3]$	$[1, 2], [2, 3]$	r_2
$[2, 1]$	$[2, 3], [3, 1]$	r_2
$[3, 2]$	$[3, 1], [1, 2]$	r_2
$[1, 1]$	$[1, 2], [2, 1]$	r_2
$[2, 2]$	$[2, 3], [3, 2]$	r_2
$[3, 3]$	$[3, 1], [1, 3]$	r_2

TABLE 4.1: The result of naïve provenance encoding

For the path example in Figure 1.2, the result of this Datalog transformation is the instrumented program

$$r_1 : path_1([x, y], [x, y]) \coloneq edge([x, y])$$

$$r_2 : path_2([x, z], [x, y], [y, z]) \coloneq edge([x, y]), path([y, z])$$

$$r_3 : path([x, y]) \coloneq path_1([x, y], \_)$$

$$r_4 : path([x, y]) \coloneq path_2([x, y], \_, \_)$$

For each new relation $path_i$, the first record stores the equivalent of the original tuple, while the other records store the subproof. By storing these together, we allow easier computation of subproofs, since we can directly find which tuples contribute to a subproof.

However, by storing subproofs directly, we change the arity of the original relations. Thus, in order to maintain the ability to use the relation in the body of other rules, we must project the original tuple back into the original relation, in rules r_3 and r_4.

The result of evaluating this instrumented Datalog on the running example Figure 1.3 is a set of tuples associated with their subproofs, shown in Table 4.1

The proof tree construction algorithm was implemented separately from the Datalog evaluation. This system stores the IDB produced by evaluation as a `std::map` in C++11, mapping the record identifier of each tuple to the record identifiers of its subproof. When given a provenance query (which consists of a single tuple), the system translates the query tuple to the matching record identifier, then accesses the map to obtain identifiers for its subproof. The full proof tree is constructed by recursively finding subproofs for each tuple, following Algorithm 2.

4.2 Guided SLD

The guided SLD approach is also implemented as a Datalog transformation in Soufflé. The implementation includes 2500 lines of code, and is publicly available at (gui, 2017).

The preliminaries that were implemented first were the notion of unique rule numbers, and a `max` functor producing the maximum out of two values. Using these, we implemented the Datalog transformation as detailed in Algorithm 3.

For the path example in Figure 1.2, this algorithm results in the program

$$r_1 : path(x, y, 1, l_1 + 1) :- edge(x, y, , l_1)$$

$$r_2 : path(x, z, 2, \max(l_1, l_2) + 1) :- edge(x, y, , l_1), path(y, z, , l_2)$$

In addition to this Datalog transformation, we implemented a modified semantics for Datalog evaluation. This is necessary due to the set enforcement semantics of Datalog. In other words, standard Datalog evaluation will only add a tuple to an instance if it does not already exist in the instance, ensuring that every tuple is unique. However, by adding the rule and level numbers, we no longer have the same behaviour. For example, a tuple $path(1, 3)$ may now exist with two different level numbers, $path(1, 3, 2, 2)$ and $path(1, 3, 2, 3)$, despite the original tuple still being identical. With standard Datalog set enforcement semantics, the tuple $path(1, 3)$ would now be duplicated.

In order to prevent this issue, we modified the set enforcement semantics of Soufflé, such that it checks only the original tuple elements before inserting a new tuple. This was implemented by performing an additional partial existence check in the internal B-Tree of Soufflé before inserting a new tuple. However, this causes a slight performance overhead, and can be improved in the future by integrating it more closely with the data structure.

The resulting instrumented Datalog, when evaluated using these modified set enforcement semantics on the input in Figure 1.3 is shown in Table 3.1

The proof tree construction, Algorithm 4 is implemented using the existing machinery in Soufflé. In Soufflé, interactions with the internal data structure are done through *loop nests*, which contain nested conditions or operations to be executed on the data structure, the B-Tree in this case. However, by default these run in order to simulate bottom-up Datalog evaluation, without any interaction. In order to

execute the proof tree construction algorithm, we need to be able to search the internal data structure for a specific tuple matching certain conditions. Thus, we want to reuse the concept of loop nests, but with some interaction of specifying which tuples to search for.

To do this, we introduced the notion of subroutines. A subroutine contains a single loop nest in addition to subroutine arguments and return values. Thus, the subproof searching part of the algorithm was implemented as a loop nest, with the constraints for the search given by the subroutine arguments, and the result of the search being returned from the subroutine. For example, on line 14 of Algorithm 4, we must find a tuple with lower proof tree level than the current tuple, and so we can pass the current tuple's level as an argument to the subroutine.

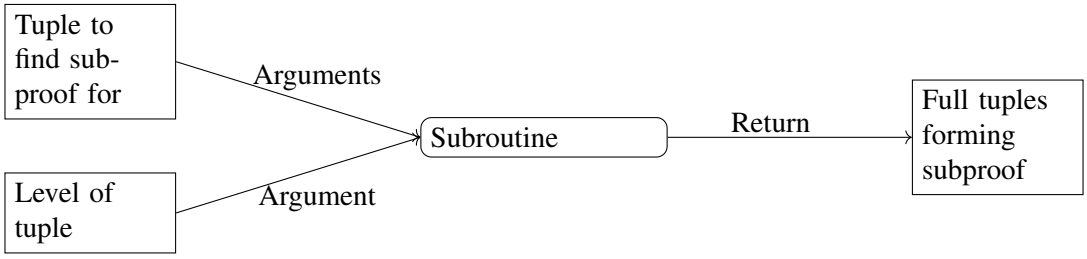


FIGURE 4.1: Usage of a subroutine for finding subproofs

The usage of subroutines in the context of proof tree construction is illustrated in Figure 4.1. A subroutine takes the tuple for which to find a subproof and its level, and returns the set of tuples which form the subproof for the query tuple.

As an illustrative example, the intermediate code for the subroutines generated for the Datalog program in Figure 1.2 are shown in Figure 4.2

```
SUBROUTINE path_1_subproof
INSERT
        SCAN edge AS t0 WHERE t0.x = argument(0) and t0.y = argument(1) and t0.@level_number < argument(2)
                RETURN (t0.x, t0.y, t0.@rule_number, t0.@level_number)
SUBROUTINE path_2_subproof
INSERT
        SCAN edge AS t0 WHERE t0.x = argument(0) and t0.@level_number < argument(2)
                SEARCH path AS t1 ON INDEX t1.x=t0.y and t1.y=argument(1) WHERE t1.@level_number < argument(2)
                        RETURN (t0.x, t0.y, t0.@rule_number, t0.@level_number, t0.y, t1.y, t1.@rule_number, t1.@level_number)
```

FIGURE 4.2: Subroutines for path program

Consider the first subroutine in Figure 4.2, labelled `SUBROUTINE path_1_subproof`. The loop nest is specified by the `INSERT` keyword, and the nested statements form the body of the loop nest. In this subroutine, the `SCAN` keyword specifies that we want to find a tuple of the relation `edge`, matching

the arguments in the subroutine, with level number strictly lower than that given in the arguments. The result of the search is then returned. Similarly, the second subroutine searches in a similar manner, but needs a second statement since the second rule contains two predicates in the body, and thus a subproof must contain two tuples.

These examples demonstrate that the search for subroutines is heavily guided by the level number annotations produced by the instrumented Datalog. It is this guidance that allows the proof tree construction algorithm to guarantee termination, since each subproof must have strictly lower level. In addition, the rule number annotation enforces which rule produces a particular tuple, and hence which subroutine to use when searching for a subproof.

In order to optimise the performance of this system, when used with Soufflé in compiled mode, these subroutines are compiled as part of the synthesised C++ code. The subroutines can be called by the provenance query user interface, allowing high performance accesses of the internal data structures.

4.3 Provenance Query User Interface

A prototype interface for provenance queries was also implemented. This command line interface allows the user to:

- `explain` a tuple, producing the proof tree for that tuple, limited to a certain number of levels
- `set depth`, changing the limit for the number of levels of a proof tree
- `subproof`, producing the proof tree for sections omitted due to the height limit

We illustrate the use of this system by an example use case. Consider the running example in Figure 1.2, which computes paths in a graph. Say there is a tuple $path(1, 8)$ which the user wants to produce the proof tree for.

```
Enter command > explain path(1, 8)
                      edge(3, 4) subproof path(0)
                      -----------------------(R1)
           edge(2, 3)           path(3, 8)
           ------------------------------------(R1)
edge(1, 2)                 path(2, 8)
-----------------------------------------------(R1)
                     path(1, 8)
```

FIGURE 4.3: Explaining a tuple

Figure 4.3 demonstrates the usage of the `explain` command. The user can query for an explanation for a tuple, resulting in a partial proof tree for that tuple. In Figure 4.3, the user queries for an explanation for the tuple $path(1, 8)$, and the result is depicted.

At the top is a section marked `subproof path(0)` denoting a subproof omitted due to the height limit. The omitted subproof is given a label, `path(0)` in this case, which can be used to query for the subproof later. This subproof query is depicted in Figure 4.4, producing the proof tree for the omitted subproof. The height limit was implemented to improve usability of the system when used with large data, where full proof trees may be prohibitively large for display (see Section 5.4).

```
Enter command > subproof path(0)
            edge(5, 8)
            -------(R2)
edge(4, 5) path(5, 8)
-------------------(R1)
      path(4, 8)
```

FIGURE 4.4: Explaining a subproof

The combination of the height limit and subproof exploration allows the user to explore a large proof tree in an on-demand fashion, overcoming usability limitations involved with these large Datalog programs. In addition, the user may change the height limit, in order to display more or less of the proof tree in a single provenance query, as demonstrated in Figure 4.5.

```
Enter command > setdepth 6
Depth is now 6
Enter command > explain path(1, 8)
                                          edge(5, 8)
                                          -------(R2)
                                edge(4, 5) path(5, 8)
                                -------------------(R1)
                      edge(3, 4)       path(4, 8)
                      -------------------------------(R1)
            edge(2, 3)             path(3, 8)
            ----------------------------------------(R1)
edge(1, 2)                   path(2, 8)
-------------------------------------------------------(R1)
                        path(1, 8)
```

FIGURE 4.5: Changing the height limit for proof trees

The provenance query user interface was implemented in C++, separately from the main Soufflé system. In order to interact with Soufflé, an interface was developed for Soufflé in both interpreted and compiled modes. This interface allows outside programs to access the IDB computed by running Soufflé. Thus, by using this interface, we were able to implement a user interface system which could be used by both the interpreter and the compiler.

The main difficulty in implementing the provenance query system was the need to access the rules in the original Datalog program. By default, the search for subproofs only has access to the Datalog instance resulting from evaluation. To overcome this, extra relations were added to the Datalog programs to store the rules of the original program needed to compute proof trees.

CHAPTER 5

Experiments and Results

The empirical evaluation of our provenance approaches is important to demonstrate viability in the face of large scale data. In this chapter, we first detail the aims for the experiments, and our experimental design. Then we answer these aims by demonstrating and discussing empirical results. We then present the results of these experiments, and discuss the implications of these results. The aims of these experiments is to answer the following experimental questions:

(Q1) How large is the size of the real-world benchmarks that we use, in particular, how many relations, rules, and tuples are involved?
(Q2) What is the runtime and memory overhead of guided SLD for Datalog evaluation?
(Q3) How do these overheads compare to other provenance approaches?
(Q4) How large are the proof trees, and how feasible is the construction of proof trees using guided SLD?

It is important to compare our guided SLD approach to other approaches in order to demonstrate an improvement over existing systems. In our experiments, we assess other approaches including no provenance, naïve provenance, and the state-of-the-art approach of (Deutch et al., 2015). The different approaches are summarized below:

- **Guided SLD.** The prototype implementation of guided SLD in Soufflé is evaluated and compared to the performance of naïve encoding.
- **Standard Datalog (with no provenance).** This is evaluated by running Soufflé without any provenance encodings.
- **Naïve encoding.** We use our prototype implementation of naïve encoding in Soufflé. This method is evaluated to provide a baseline performance for provenance approaches.
- **Top-k** (Deutch et al., 2015). To measure the performance of the top-k approach presented in (Deutch et al., 2015), we implemented the Datalog instrumentation algorithm for a specific

provenance query, for transitive closure and same generation benchmarks. The query that we use generates the proof tree for a single output tuple for each benchmark. Thus, we obtain an instrumented Datalog program which is able to answer this particular provenance query. We note that this instrumented Datalog program only provides provenance for a specific query, compared to the guided SLD instrumentation result which can be used for any provenance query. The top-k instrumented Datalog program is run using standard Soufflé.

The aforementioned approaches are representative of the current state of research for provenance in Datalog. As far as we are aware, the top-k approach is the state-of-the-art for provenance in Datalog (Deutch et al., 2015). Thus, it is sufficient to compare guided SLD with these approaches to demonstrate its performance advantages.

As a test machine we use a Core i7-7700K 4.2GHz computer with 64 GB memory that runs Ubuntu 16.04 as an operating system. For generating Soufflé's executables we use GCC 5.4. Soufflé by default selects between a Trie and a B-Tree as the internal data structure, depending on the input program. The guided SLD implementation, however, relies on the B-Tree data structure, and thus Soufflé was modified to always select the B-Tree regardless of whether provenance was enabled or not. This removes the variable of data structure in the experiments, so differences in Datalog evaluation time are purely a result of the provenance encoding.

5.1 (Q1) Benchmarks

It is important to use appropriate benchmarks in order to demonstrate the performance of guided SLD. We use a combination of real-world program analysis benchmarks and synthetic benchmarks.

- **DOOP** (Bravenboer and Smaragdakis, 2009) is an industrial strength framework for points-to analysis of Java programs. The included DaCapo set of Java programs used as benchmarks in this thesis. This set of benchmarks produces Datalog programs with large input, and large output of up to 100 million tuples. Thus, this is an appropriate benchmark to demonstrate the performance of guided SLD on large scale real-world datasets.

  We use the *context-insensitive* and *1-object-sensitive, 1-heap* (1-obj, 1-heap) analyses. Context-insensitive provides a less accurate and less data intensive static program analysis, while 1-obj, 1-heap provides a more accurate and more data intensive analysis. However, both

Benchmark	# relations	# rules	# EDB tuples	# IDB tuples
DOOP				
context-insensitive	274	847	See Table 5.2	
1-obj, 1-heap	274	851	See Table 5.2	
Soufflé				
cellular	10	17	12	2,009,000
dominance	12	16	32,766	1,075,150,848
josephus	3	4	3000	4,498,500
sg	2	4	2046	1,398,101
tc	2	3	4000	16,000,000
topological	6	7	4,498,597	4,501,502
turing	5	24	66	204,006
Transitive closure & Same generation				
tc	2	3	20,000	1,000,000
sg	2	4	2000	634,650

TABLE 5.1: Statistics for benchmarks

analyses provide a large and demanding benchmark, with the largest instances involving over 200 relations, 800 rules and producing over 90M output tuples.

- **Soufflé test suite.** Soufflé (sou, 2017) includes an extensive test suite with various Datalog programs, collected over a period of 4 years from various contributors to Soufflé. We select a number of these and in some cases generate synthetic data designed to be a worst case scenario. In particular, for the dominance, same-generation, transitive-closure, and topological benchmarks, we use our synthetically generated data as a benchmark. These benchmarks with synthetic data contain small programs and input, however produce output data at least a quadratic factor larger than the input data. Thus, they test the scalability of guided SLD when faced with these worst case scenarios. We expect this to result in a larger overhead than real-world test cases, as we introduce extra annotations for each output tuple.
- **Transitive closure and same generation.** These are standard benchmarks for many Datalog applications. They contain a small set of simple Datalog rules, however recursion means that the output data can be large. In our experiments, we evaluate using a transitive closure instance with 1000 nodes and 20,000 randomly generated edges, producing 1M output paths, and a same generation instance with 1000 nodes and 2000 random edges, producing approximately 630,000 output tuples.

The full statistics for the benchmarks are presented in Tables 5.1 and 5.2.

Benchmark	context-insensitive		1-obj, 1-heap	
	# EDB tuples	# IDB tuples	# EDB tuples	# IDB tuples
antlr	8,255,336	28,696,167	8,255,343	38,271,445
bloat	4,428,034	24,521,158	4,428,034	91,442,534
chart	8,675,836	25,299,934	8,675,869	30,206,018
eclipse	4,360,525	18,907,089	4,360,525	26,830,303
fop	8,705,960	24,001,594	8,705,924	26,631,375
hsqldb	8,929,951	25,422,344	8,929,951	28,442,958
jython	5,187,075	74,625,219	n/a	n/a
luindex	4,362,347	14,235,201	4,362,347	16,830,712
lusearch	4,362,370	14,843,746	4,362,342	17,541,158
pmd	8,325,264	25,299,170	8,325,264	29,781,734
xalan	8,602,591	28,206,129	8,602,591	43,644,018

TABLE 5.2: Statistics for DOOP benchmarks

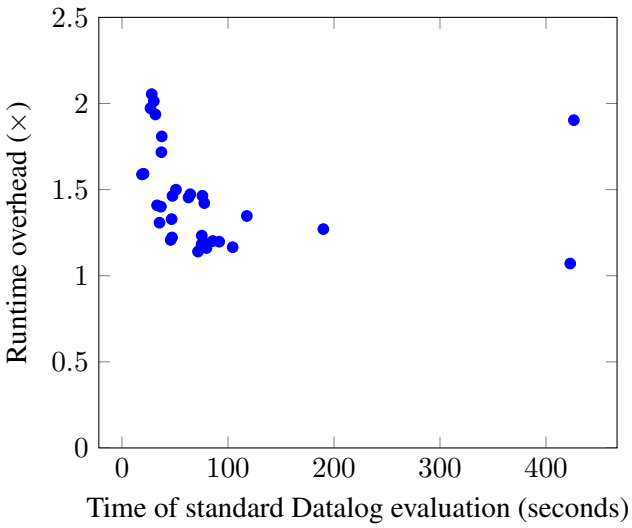


FIGURE 5.1: Runtime overhead of evaluation time for guided SLD provenance encoding

These benchmarks cover numerous aspects of Soufflé's performance including join performance, fixed-point performance for recursive relations, arithmetic performance, and record/constructor semantics. We also assess the performance of our provenance approach using two different modes of operation in Soufflé, the compiler and the interpreter. The Soufflé compiler generates high-performance C++ code from Datalog, while the interpreter evaluates Datalog directly. By using these two different modes, we ensure that our results are independent of any changes made to the underlying Soufflé system.

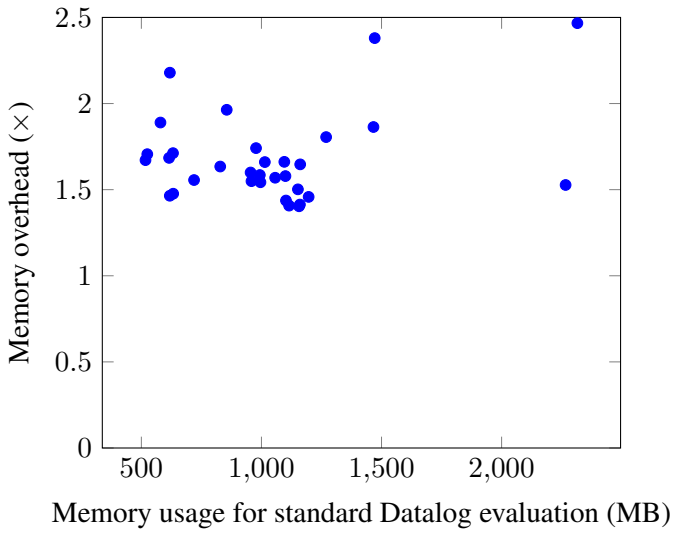


FIGURE 5.2: Memory usage overhead of evaluation time for guided SLD provenance encoding

5.2 (Q2) Runtime and Memory Overhead

We begin with a summary of the runtime overhead of guided SLD. Figure 5.1 shows the factor of runtime overhead that using guided SLD causes compared to standard Datalog evaluation without provenance encoding. In other words, we plot

$$\frac{\text{Time for Soufflé with guided SLD}}{\text{Time for standard Soufflé}} \text{ vs. Time for standard Soufflé}$$

Figure 5.2 depicts similar results for the memory usage overhead of guided SLD compared to standard Datalog.

Each data point in these plots is a single Datalog program in the set of DOOP DaCapo benchmarks. This summary combines the context-insensitive and 1-obj, 1-heap analyses for DOOP DaCapo benchmarks. In Figure 5.1, the evaluation runtime for standard Datalog is an approximate baseline measurement of the size of the Datalog program, and similarly the memory usage for evaluation of standard Datalog is used as a baseline in Figure 5.2. The result demonstrates that the overhead for Datalog evaluation with guided SLD remains approximately constant in terms of both runtime and memory usage, regardless of the size of the Datalog program. More specifically, we show a constant factor overhead of around 1.5 times for guided SLD, thus indicating its viability for computing provenance information even for large scale data.

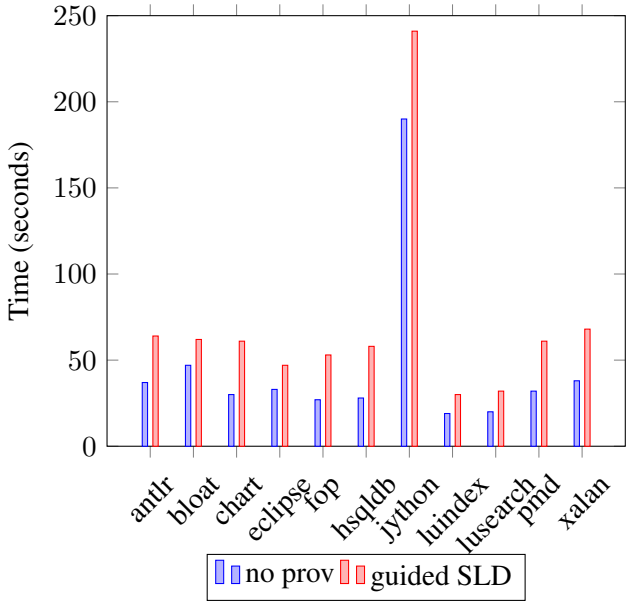


FIGURE 5.3: Results of Datalog evaluation time on DOOP Dacapo context-insensitive benchmarks

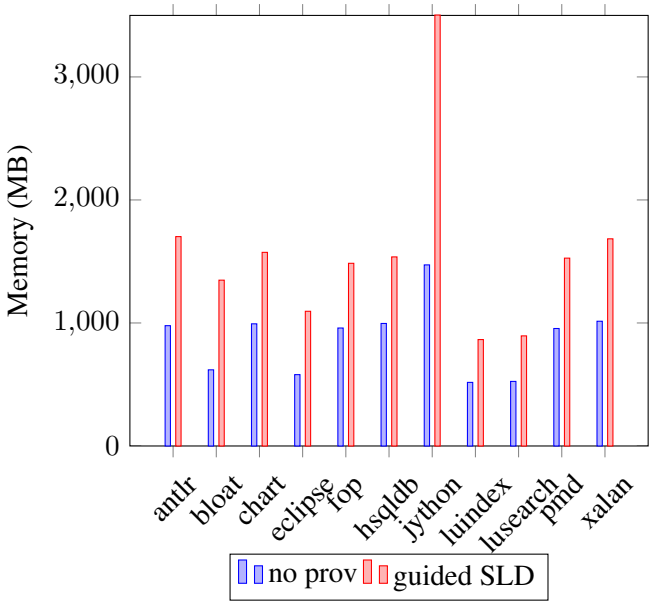


FIGURE 5.4: Results of Datalog evaluation memory usage on DOOP Dacapo context-insensitive benchmarks

The runtime and memory usage of guided SLD and standard Datalog on the DaCapo benchmarks are detailed in Figures 5.3 and 5.4. In these figures, DOOP was running in context-insensitive mode, which provides the least accurate program analysis and thus the smallest benchmark size. Despite this, however, the evaluation time of standard Datalog is still in excess of 20 seconds in most benchmarks, showing

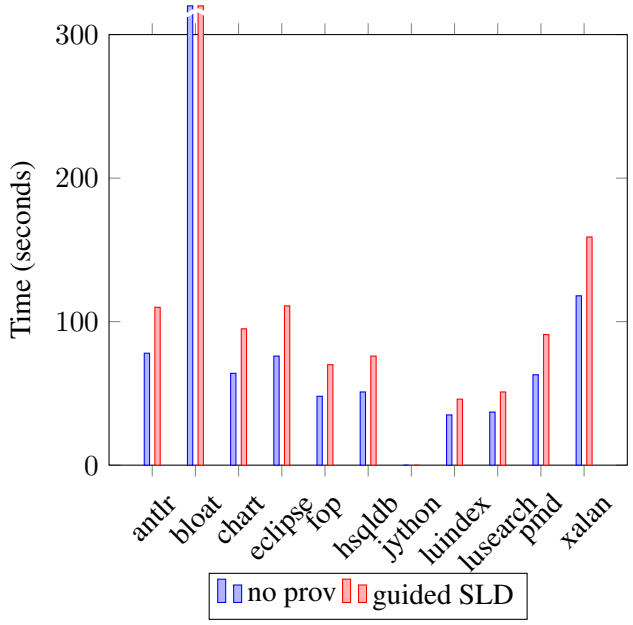


FIGURE 5.5: Results of Datalog evaluation time on DOOP Dacapo 1-obj, 1-heap benchmarks

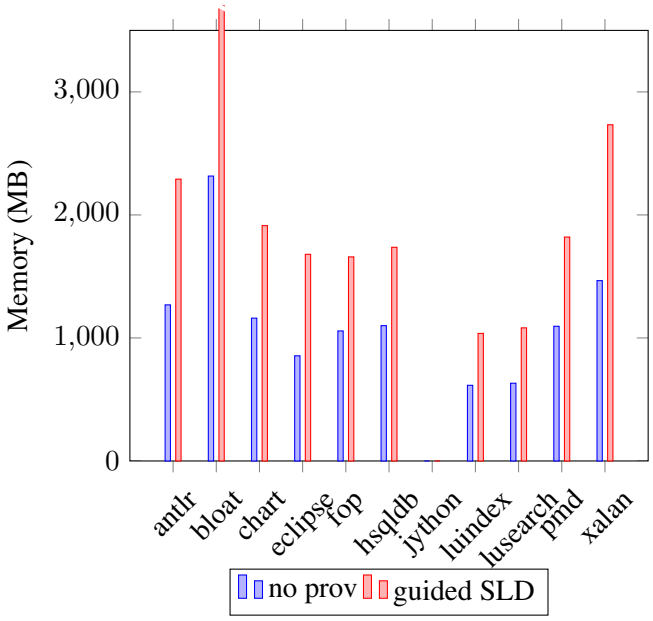


FIGURE 5.6: Results of Datalog evaluation memory usage on DOOP Dacapo 1-obj, 1-heap benchmarks

that it is indeed a demanding workload. The overhead is approximately equivalent for all benchmarks, for both evaluation time and memory consumption, reflecting the summary in figure 5.1.

In Figures 5.5 and 5.6 we present the evaluation time and memory usage of guided SLD vs standard Datalog, using DOOP in 1-obj, 1-heap mode. This mode provides a more accurate program analysis,

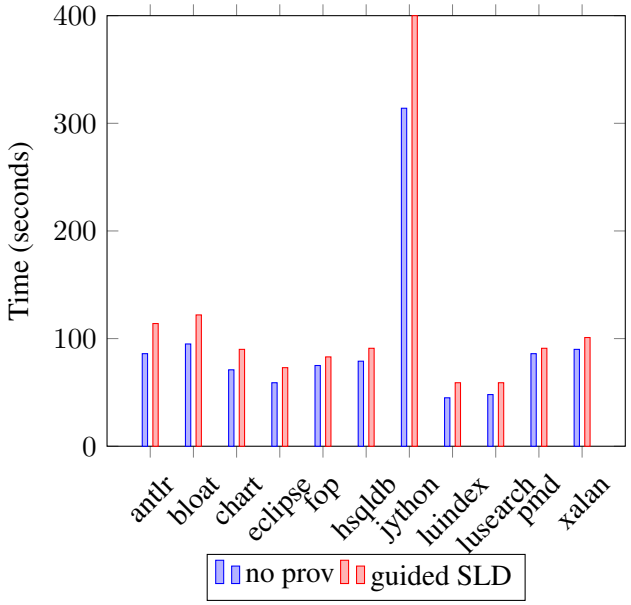


FIGURE 5.7: Results of Datalog evaluation time on DOOP Dacapo context-insensitive benchmarks with Soufflé in interpreted mode

therefore resulting in Datalog programs with larger input and output data. Due to this increased size, both runtime and memory construction are higher compared to the results of context-insensitive mode in Figures 5.3 and 5.4. The missing results for the Jython benchmark in Figures 5.5 and 5.6 are a result of it timing out after 1 hour, with both standard Datalog as well as guided SLD. Of all the benchmarks that completed running, however, we've showed that the overheads remain constant even with larger data sizes.

The results in Figures 5.3, 5.4, 5.5, and 5.6 were assessed using Soufflé in compiled mode, as this provides the best performance for these large scale datasets.

These same DOOP DaCapo benchmarks were run using Soufflé in interpreted mode to demonstrate that the runtime and memory overheads remain similar, regardless of the mode of operation. Only the context-insensitive analyses were assessed, as the slower performance of interpreted mode is prohibitive for more complex analyses. These results are shown in Figures 5.7 and 5.8. This demonstrates that the Datalog evaluation overhead of guided SLD is purely a result of the instrumented Datalog, rather than any internal changes to the compilation mode of Soufflé.

The results using the DOOP DaCapo benchmarks demonstrate that guided SLD performs well in terms of both runtime and memory overhead for large scale real-world problems.

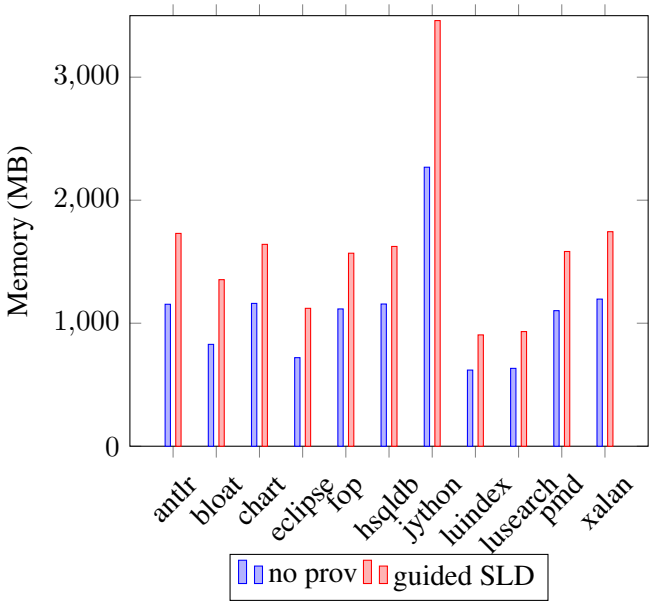


FIGURE 5.8: Results of Datalog evaluation memory usage on DOOP Dacapo context-insensitive benchmarks with Soufflé in interpreted mode

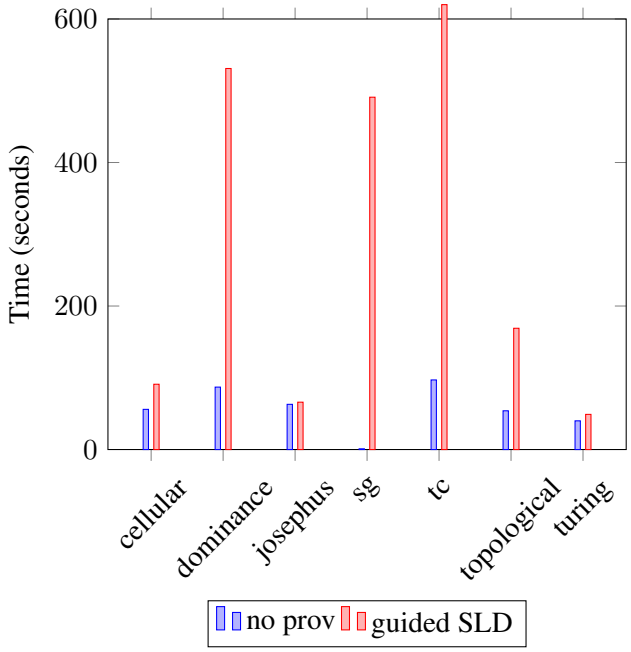


FIGURE 5.9: Results of Datalog evaluation time on Soufflé benchmarks

Synthetically generated benchmarks were also tested as part of the Soufflé benchmarks shown in Figures 5.9 and 5.10. These benchmarks demonstrate that guided SLD exhibits poor performance when faced with a worst case scenario. In particular, the same generation benchmark exhibited a runtime

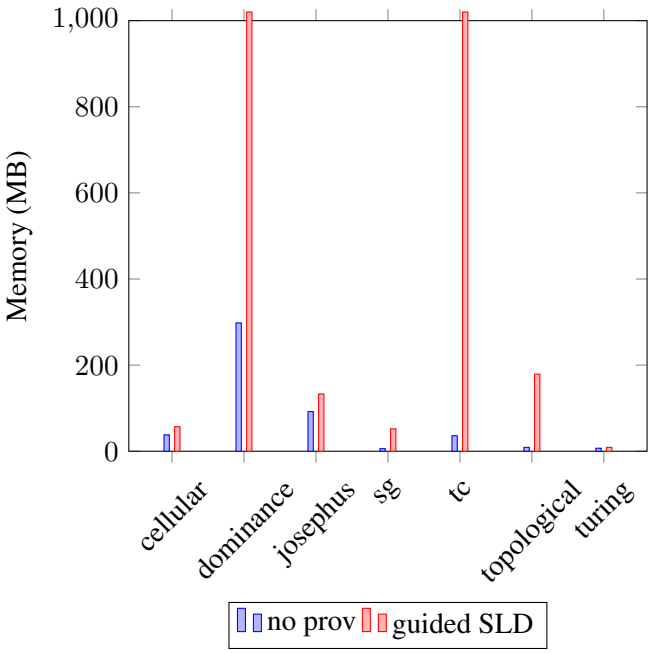


FIGURE 5.10: Results of Datalog evaluation memory usage on Soufflé benchmarks

overhead of over 1000×. This benchmark was implemented to simulate a complete binary tree and hence the output size is significantly larger than the input size, resulting in a large number of annotations that must be computed by guided SLD.

The other synthetic benchmarks result in a quadratically larger output than input, thus also exhibiting worse performance than the real-world DOOP benchmarks. We notice, however, that the Soufflé benchmarks without synthetic data, namely cellular, josephus and turing exhibit a runtime and memory overhead in line with that of the real-world DOOP benchmarks.

In summary, we conclude that the Datalog instrumentation algorithm of guided SLD produces an instrumented Datalog program which can be evaluated with reasonable runtime and memory usage overhead of approximately 1.5× on real-world data. Thus it is feasible to produce an annotated IDB with partial provenance information even for large Datalog programs using the guided SLD approach.

5.3 (Q3) Comparison with other approaches

In Figures 5.11 and 5.12, we compare the evaluation time and memory usage of standard Datalog, naïve encoding and guided SLD, as well as the top-k approach (Deutch et al., 2015). The instance of transitive closure was a graph with 1000 nodes, and 20,000 randomly generated edges, which produces an output

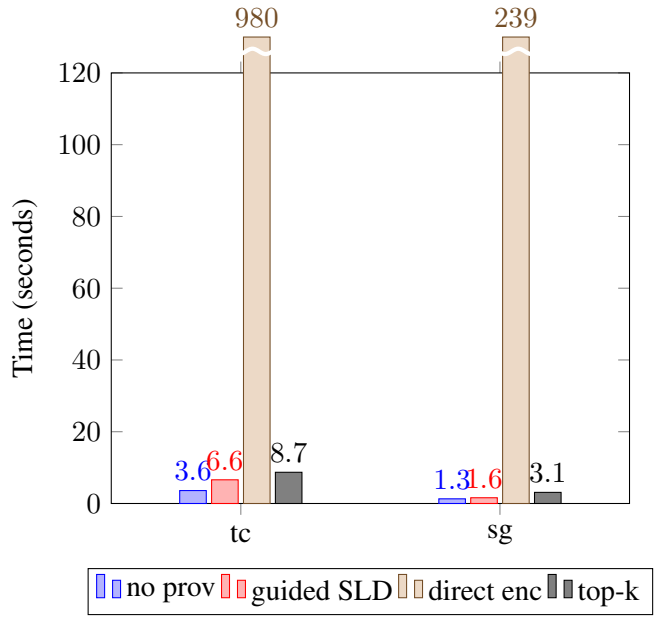


FIGURE 5.11: Results of Datalog evaluation time on transitive closure and same generation

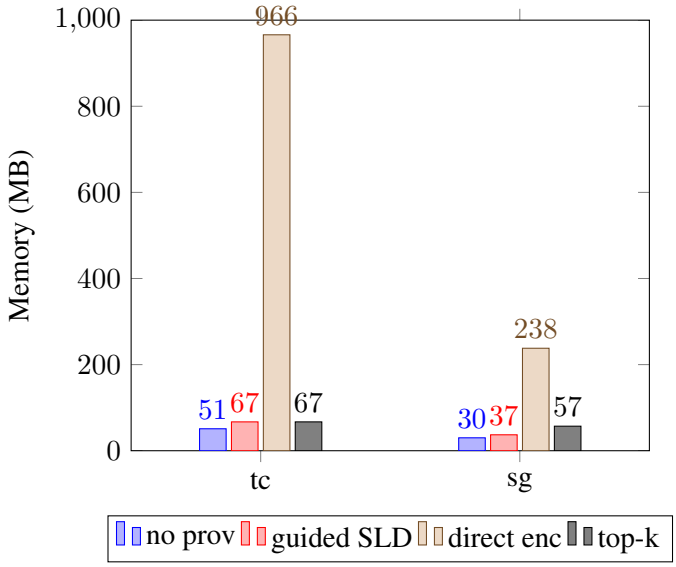


FIGURE 5.12: Results of Datalog evaluation memory usage on transitive closure and same generation

database containing 1M paths. The instance of same generation contained 1000 nodes with 2000 edges, yielding approximately 630,000 output tuples.

These results show that naïve encoding has a prohibitively large overhead, even when dealing with relatively small Datalog programs. Top-k performs similarly to guided SLD, but only with a single

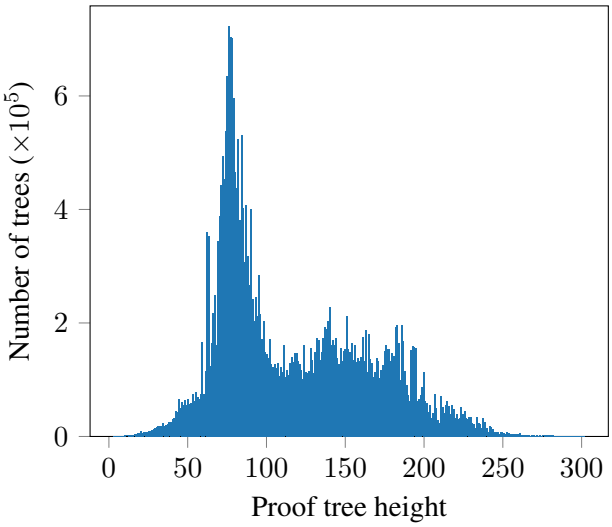


FIGURE 5.13: Heights of proof trees for DaCapo benchmarks

provenance query. In (Deutch et al., 2015), the empirical experiments show that full provenance tracking for transitive closure with a database containing 1.7M output tuples required over 6.5 hours, however this result used a different Datalog evaluation engine to Soufflé. As a result of this high runtime for full provenance tracking, we conclude that guided SLD performs better than both naïve encoding and top-k in the general case. We also highlight the additional flexibility and reusability of guided SLD compared to top-k, since the instrumented Datalog only needs to be evaluated once then can be queried by any number of arbitrary provenance queries.

5.4 (Q4) Proof Tree Construction

We also present results demonstrating the difficulty of proof tree construction for Datalog programs at large scale. Figure 5.13 shows the distribution of heights of proof trees for the DaCapo benchmarks, and these heights can be in excess of 300. While this may not seem prohibitive, the expected number of nodes in the proof tree is exponential in the height. A non-linear regression run on the sizes of actual proof trees suggests that the number of nodes in a proof tree of the DaCapo benchmarks is approximately 1.466^h, where h is the height of the proof tree. This value explodes dramatically for larger proof trees, and thus presents a technical and usability challenge in producing a meaningful explanation for a tuple.

In Figure 5.14, we show the time for proof tree construction for the first 30 levels of proof trees for randomly sampled output tuples of DaCapo benchmarks, plotted against the number of nodes in the

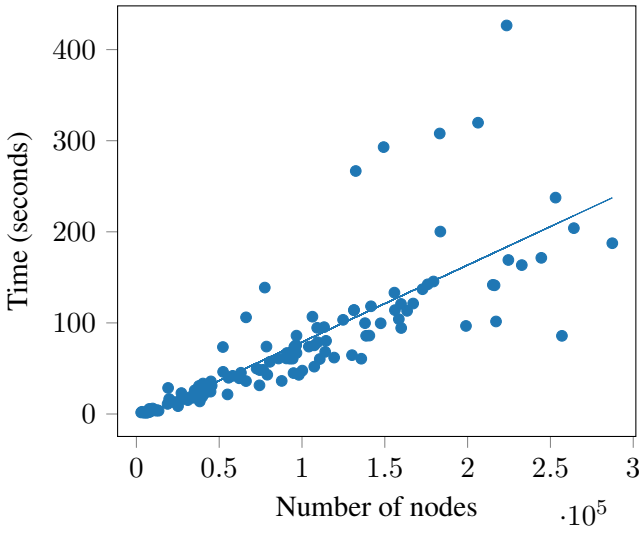


FIGURE 5.14: Proof tree construction time for first 30 levels of proof tree

partial proof trees. Even for the first 30 levels, the proof trees contain in excess of 300,000, suggesting that the full proof trees would be prohibitively large.

While these larger proof trees may be prohibitive in terms of both construction effort and usability, we show that proof tree construction is approximately linear in the size of the tree. This demonstrates that guided SLD scales well to large sizes for proof tree construction, and that reasonably sized proof trees can be be constructed in a feasible amount of time.

CHAPTER 6

Conclusion and Future Work

The debugging of logic programs is a challenge with many implications for real world problems. Previous works have suggested data provenance as an approach to provide debugging information, however existing approaches to computing provenance fall over when faced with the large scale data of real problems.

In this thesis, we have presented the problem of debugging for large scale logic problems, and outlined existing solutions to this problem. We have identified the challenge in building solutions that scale well to large datasets.

In response to this problem, we have introduced a novel hybrid approach to generating provenance information for Datalog programs. We have first explored the theoretical background of this problem, and shown an equivalence between proof trees and the program output itself, thus demonstrating that the problem is well defined. We then present two approaches of computing proof trees in practice, and implemented prototypes of both approaches.

The first is a naïve approach to the problem, which is feasible for small scale data, but demonstrates the difficulty of this problem for larger data sizes.

The second is guided SLD, a novel hybrid approach, combining bottom-up and top-down Datalog evaluation in order to minimise the overhead of computing provenance information. This is the main contribution of our research, demonstrating the possibility of practically computing provenance for Datalog programs with large data sizes.

Through empirical experiments, we have demonstrated the feasibility of guided SLD on real-world static program analysis datasets, with a constant Datalog evaluation overhead of approximately 1.5x, and proof tree construction being a few minutes for reasonably sized explanations. Simultaneously, this demonstrates that naïve approaches to provenance fall flat in practice.

6.1 Future Work

There are many future avenues of research that can be considered, both regarding the theoretical and practical components of this thesis.

6.1.1 Provenance for Negation

A major issue with current approaches to provenance, including this research, is that they provide meaningful explanations only for positive tuples. Namely, we are able to answer the question "why does this tuple exist in the output?" However, the opposite question, "why does this tuple *not* exist in the output?" presents a major theoretical challenge.

The reason is that semantics for negation in Datalog deal with enumerating all possible tuples, and a negated tuple holds if the tuple does not appear in the enumeration. Thus, current approaches to explaining negated tuples (Lee et al., 2017) explicitly provide this enumeration. With the large sizes of data for real world problems, however, this enumeration is a rather clumsy approach to provenance.

Therefore, the theoretical possibility of summarising these explanations in some meaningful way needs to be explored. If this can be done, and practical approaches to provenance for negation can be developed, then we could feasibly explain why a tuple doesn't exist in the output of a program. Such a system could lead towards automated repair of Datalog databases, and other massive implications for logic programming.

6.1.2 Query Scheduling for Guided SLD

Query scheduling is a possible avenue for optimisation of the top-down section of guided SLD. Currently, the implementation naïvely utilises the same schedule for bottom-up evaluation as top-down proof construction. However, the aim for scheduling for bottom-up evaluation is to discover as large a number of new tuples as possible, in order to reduce the number of iterations taken to reach a fixpoint. On the other hand, top-down proof tree construction optimally searches through the smallest number of tuples, in order to reduce the effort in finding a subproof.

Thus, in many cases reusing the query scheduling may cause suboptimal performance, and improving this scheduling for top-down proof tree construction may drastically reduce the time taken to compute proof trees.

6.1.3 Data Structure Optimisations for Guided SLD

The current prototype implementation of guided SLD utilises the existing data structures in Soufflé. Thus, the modification to set enforcement semantics requires an explicit extra existence check in each iteration for Datalog evaluation. This can be integrated into the data structure, potentially resulting in faster Datalog evaluation with guided SLD.

Bibliography

2017. Alternative guided sld by taipan-snake - pull request #459 - souffle-lang/souffle. Accessed: 31-10-2017.

2017. Naive implementation of provenance in souffle by taipan-snake - pull request #418 - souffle-lang/souffle. Accessed: 31-10-2017.

2017. souffle-lang/souffle: Soufflé is a variant of datalog for tool designers crafting analyses in horn clauses. soufflé synthesizes a native parallel c++ program from a logic specification. more information can be found here: http://souffle-lang.github.io/. Accessed: 19-10-2017.

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley Publishing Company.

Nicholas Allen, Bernhard Scholz, and Padmanabhan Krishnan. 2015. Staged points-to analysis for large code bases. In *Proceedings of Compiler Construction: 24th International Conference, CC 2015*, pages 131–150. Springer Berlin Heidelberg.

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. 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*, SOAP 2017, pages 25–30. ACM, New York, NY, USA.

Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1371–1382. ACM, New York, NY, USA.

Tarun Arora, Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, and Divesh Srivastava". 1993. Explaining program execution in deductive systems. In *Proceedings of Deductive and Object-Oriented Databases: Third International Conference*, pages 101–119.

Zhifeng Bao, Henning Köhler, Liwei Wang, Xiaofang Zhou, and Shazia Sadiq. 2012. Efficient provenance storage for relational queries. In *Proceedings of the ACM international conference on Information and knowledge management*, volume 21, pages 1352–1361.

Catriel Beeri and Raghu Ramakrishnan. 1987. On the power of magic. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '87, pages 269–284. ACM, New York, NY, USA.

Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. 2004. An annotation management system for relational databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 900–911. VLDB Endowment.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262.

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2001. Why and where: A characterization of data provenance. In *Proceedings of the International Conference on Database Theory*, volume 1973, pages 316–330.

Stefan Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions Knowledge and Data Engineering*, 1(1):146–166.

Adriane P. Chapman, H.V. Jagadish, and Prakash Ramanan. 2008. Efficient provenance storage. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 993–1006.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1:379–474.

Daniel Deutch, Amir Gilad, and Yuval Moskovitch. 2015. Selective provenance for datalog programs using top-k queries. In *Proceedings of the VLDB Endowment*, volume 8, pages 1394–1405.

Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. 2014. Circuits for datalog provenance. *Conference on Database Theory*, 17:201–212.

Boris Glavic and Gustavo Alonso. 2009. Perm: Processing provenance and data on the same data model through query rewriting. *IEEE International Conference on Data Engineering*, 25:174–185.

Boris Glavic, Renée J. Miller, and Gustavo Alonso. 2013. Using sql for efficient generation and querying of provenance information. *Lecture Notes in Computer Science*, 8000:291–320.

Sergio Greco and Cristian Molinaro. 2015. *Datalog and Logic Databases*. Morgan & Claypool Publishers.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 675–686.

Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1213–1216. ACM.

Zachary G. Ives, Andreas Haeberlen, Tao Feng, and Wolfgang Gatterbauer. 2012. Querying provenance for ranking and recommending. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*, pages 9–9.

Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On synthesis of program analyzers. In *Proceedings of Computer Aided Verification*, volume 28, pages 422–430.

Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. 2010. Querying data provenance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 951–962.

Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. 2012. Declarative datalog debugging for mere mortals. *Lecture Notes in Computer Science*, 7494:111–122.

Georg Lausen, Bertram Luascher, and Wolfgang May. 1998. On active deductive databases: The statelog approach. *Lecture Notes in Computer Science*, 1472:69–106.

Seokki Lee, Sven Köhler, Bertram Ludäscher, and Boris Glavic. 2017. Efficiently computing provenance graphs for queries with negation. *CoRR*, abs/1701.05699.

Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. 2005. Mulval: A logic-based network security analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 8–8. USENIX Association, Berkeley, CA, USA.

Konstantinos Sagonas, Terrance Swift, and David S. Warren. 1993. Xsb: An overview of its use and implementation. *SUNY Stony Brook*, pages 11794–4400.

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

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309.

Jeffrey D. Ullman. 1989. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, pages 140–149. ACM, New York, NY, USA.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. *Using Datalog with Binary Decision Diagrams for Program Analysis*, pages 97–118. Springer Berlin Heidelberg, Berlin, Heidelberg.

Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. 2010. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 615–626.