Provenance, Incremental Evaluation, and Debugging in Datalog

David Wei Zhao

A thesis submitted to fulfil requirements for the degree of

Doctor of Philosophy

School of Computer Science

Faculty of Engineering

The University of Sydney

March 2022

Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

David Zhao

Publication List

During my candidature, I contributed to the following publications:

- [1] Hu, X., Karp, J., Zhao, D., Zreika, A., Wu, X., and Scholz, B. (2021, October). The Choice Construct in the Soufflé Language. In *Asian Symposium on Programming Languages and Systems (pp. 163-181)*. Springer, Cham. doi:10.1007/978-3-030-89051-3_10

- [2] Zhao, D., Subotic, P., Raghothaman, M., and Scholz, B. (2021, September). Towards Elastic Incrementalization for Datalog. In *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP)* (pp. 1-16). doi:10.1145/3479394.3479415

- [3] Hu, X., Zhao, D., Jordan, H. and Scholz, B. (2021, June). An efficient interpreter for Datalog by de-specializing relations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)* (pp. 681-695). doi:10.1145/3410297

- [4] Zhao, D., Subotić, P. and Scholz, B. (2020). Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. In *ACM Transactions on Programming Languages and Systems (TOPLAS), 42(2)*, (pp. 1-35). doi:10.1145/3379446

- [5] Jordan, H., Subotić, P., Zhao, D. and Scholz, B. (2020). Specializing parallel data structures for Datalog. In *Concurrency and Computation: Practice and Experience*, (p.e5643). doi:10.1002/cpe.5643

- [6] Raghothaman, M., Mendelson, J., Zhao, D., Naik, M. and Scholz, B. (2019). Provenance-guided synthesis of Datalog programs. In *Proceedings of the ACM on Programming Languages, 4 (POPL)*, (pp. 1-27). doi:10.1145/3371130

- [7] Nappa, P., Zhao, D., Subotić, P. and Scholz, B. (2019, September). Fast Parallel Equivalence Relations in a Datalog Compiler. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (pp. 82-96). IEEE. doi:10.1109/PACT.2019.00015

- [8] Jordan, H., Subotić, P., Zhao, D. and Scholz, B. (2019, February). A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*. ACM, New York, NY, USA. doi:10.1145/3293883.3295719

- [9] Jordan, H., Subotić, P., Zhao, D. and Scholz, B. (2019, February). Brie: A Specialized Trie for Concurrent Datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores* (pp. 31-40). ACM. doi:10.1145/3303084.3309490

Authorship Attribution

This thesis contains the following material submitted or accepted for publication:

- Chapter 4 is published as [4]. I designed the encoding and algorithms with the co-authors. I implemented the approach and performed the experimental evaluation.
- Chapter 5 is published as [2]. I developed the encoding and algorithms, implemented the approach, and performed the experimental evaluation.
- Chapter 6 is submitted for publication as [10]. I designed and implemented the approach, and performed the experimental evaluation.

In addition to the statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

David Zhao

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Prof. Bernhard Scholz

Acknowledgements

First of all, a tremendous thanks to my wonderful supervisor, Bernhard. These four years of research and progress have only been possible thanks to the tireless patience and guidance of Bernhard. The journey hasn't always been easy, but I am grateful for Bernhard's groundedness and steady encouragement. His connections and understanding of the wider world of research have been invaluable, and ultimately led to higher quality publications and research.

I'd also like to thank my core contributors, Paul and Mukund, who have also been with me throughout this PhD journey. Their contributions, which range from technical to experimental to writing to general support, have been a massive help, and the publications in this thesis are only possible thanks to them. I'd also like to thank my other contributors: Mayur and the group at UPenn, Xiaowen, Abdul, Sam, Herbert, Patrick, among others.

To the rest of the programming languages group, thank you for being there with support and encouragement, and for just being a fun group to hang out with. I will always cherish these friendships that I've made.

Last, but certainly not least, I'd like to thank my wonderful family and friends and my girlfriend for all of their encouragement and support over the last four years.

Abstract

The Datalog programming language has recently found increasing traction in research and industry. Driven by its clean declarative semantics, along with its conciseness and ease of use, Datalog has been adopted for a wide range of important applications, such as program analysis, graph problems, and networking. To enable this adoption, modern Datalog engines have implemented advanced language features and high-performance evaluation of Datalog programs. Unfortunately, critical infrastructure and tooling to support Datalog users and developers are still missing. For example, there are only limited tools addressing the crucial debugging problem, where developers can spend up to 30% of their time finding and fixing bugs.

This thesis addresses Datalog's tooling gaps, with the ultimate goal of improving the productivity of Datalog programmers. The first contribution is centered around the critical problem of debugging: we develop a new debugging approach that explains the execution steps taken to produce a faulty output. Crucially, our debugging method can be applied for large-scale applications without substantially sacrificing performance. The second contribution addresses the problem of incremental evaluation, which is necessary when program inputs change slightly, and results need to be recomputed. Incremental evaluation allows this recomputation to happen more efficiently, without discarding the previous results and recomputing from scratch. Finally, the last contribution provides a new incremental debugging approach that identifies the root causes of faulty outputs that occur after an incremental evaluation. Incremental debugging focuses on the relationship between input and output and can provide debugging suggestions to amend the inputs so that faults no longer occur. These techniques, in combination, form a corpus of critical infrastructure and tooling developments for Datalog, allowing developers and users to use Datalog more productively.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The *Datalog* programming language has had a tumultuous history [11, 12, 13]. Having been introduced in the 1980s, Datalog was originally designed as a database query language, allowing for more expressive queries such as those including *recursion*. However, Datalog fell out of favor in the following years, with database practitioners preferring the verbosity of SQL [14] and not needing recursion. In recent decades, Datalog has seen a resurgence in popularity, driven by a shift towards its use as a programming language.

During this resurgence, Datalog has found several important use cases, ranging from semantic web searching [15] to networking [16, 17, 18, 19] to program analysis [20, 21, 22], among others. In comparison to database querying, these new use cases are characterized by increasingly complex recursive queries and data representations. Furthermore, these complex applications have also driven Datalog language development, including more sophisticated features such as complex data types, user-defined functors, components/modules, and data structures [23, 24].

A Datalog program is specified as a set of logical rules. These rules define what the intended output should look like, in contrast to other paradigms, which specify step-by-step how to compute a program. One commonly cited example of Datalog, which showcases its succinctness, is *transitive closure*. This program computes all possible paths in a graph and is represented in just two lines of code:

```
1 path(X,Y) :- edge(X,Y).
2 path(X,Z) :- edge(X,Y), path(Y,Z).
```

This transitive closure program contains two relations: `edge` and `path`. The first rule states that if there is an edge `(X,Y)`, then there is also a path `(X,Y)`. The second rule expresses transitivity, stating that if there is an edge `(X,Y)` and a path `(Y,Z)`, then there is also a path `(X,Z)`. In contrast to the elegance of these two Datalog rules, other programming paradigms such as imperative or functional languages may require tens or even hundreds of lines to implement an algorithm that traverses the graph to discover its transitive closure.

In the early history of Datalog, research mainly focused on optimizations and efficient evaluation of Datalog programs [12]. For early database querying purposes, the ability to efficiently evaluate a Datalog program was sufficient for most applications. In these use cases, advanced infrastructure and tooling were not essential, given that most database queries written in Dat-

alog were only a handful of lines of code. However, the rise of modern Datalog applications characterized by increasingly complex programs containing hundreds of relations and rules [20, 22, 25] has necessitated the development of improved tools and infrastructure to support Datalog programmers and increase productivity.

While the clean semantics and high performance of the Datalog language are huge benefits for improving programmer productivity, tool support and infrastructure for Datalog have unfortunately been lacking. For example, consider the important problem of debugging. Studies have shown that programmers can spend up to 30% of their development time debugging [26], thus forming an essential consideration for programmer productivity. Therefore, over the decades of support for traditional imperative languages, tools such as debuggers and IDEs [27, 28] have been developed and refined to aid the debugging process. Nowadays, many such tools exist for almost all common imperative programming languages, and these tools are integrated into user-friendly development environments.

Meanwhile, Datalog and other logic programming languages have not seen the same popularity and support. Current approaches for debugging logic programs are often clunky [29], requiring users to inspect confusing execution traces manually. Moreover, the decades of research in developing user-friendly debugging techniques for imperative languages cannot be directly translated to logic languages due to the different semantics and execution models. However, despite these challenges, the clean declarative semantics of logic programming provides the potential for more usable and expressive debugging tools and even automated techniques to suggest code fixes.

Furthermore, the logic programming paradigm offers even more opportunities for automated tooling compared to traditional paradigms. For instance, one important technique is *incremental evaluation*, which allows a program to efficiently update its results given a small change to the input. Real-world tasks are increasingly taking advantage of this incrementality, such as for graph analyses [30], database querying [31], debugging [32], among many others. The declarative semantics of Datalog lends itself to allowing *automatic* incrementalization of a program, providing orders of magnitude speedups for updating computations while maintaining the same ease of succinct logic programming for users. In contrast, automatically incrementalizing imperative or functional programs can be challenging or impossible [33], and programs written in these languages must be manually adapted to implement complex incremental algorithms.

In this thesis, we aim to answer the following questions to enhance the tooling and infrastructure support for Datalog.

1. **How can we debug large-scale Datalog programs?** The Datalog language and semantics can inherently record what happened during execution in the form of *proof trees* [34] which describe the computations performed during evaluation. However, these proof trees are discarded during Datalog evaluation in the name of performance. Therefore, how can we provide an encoding with minimal overhead, which allows for efficiently reconstructing trees for debugging? Furthermore, can we provide a utility for users to explore these proof trees in a usable way?

2. **How can we effectively incrementalize Datalog programs?** Incremental evaluation

is essential for applications where inputs may change slightly between program runs (especially in debugging scenarios). While existing techniques for Datalog and similar languages can perform incremental evaluation where input updates are small, can we efficiently incrementalize Datalog programs for both large and small updates?

3. **How can we provide automated debugging utilities?** While proof trees are an effective vehicle for Datalog programmers to find explanations for tuples, they may be difficult to understand for users who are not familiar with the Datalog rules. Furthermore, under an incremental evaluation setting, faults may appear between updates. For Datalog users using incremental evaluation, can we devise a debugging framework that finds the input changes which cause faults? Furthermore, can we provide debugging suggestions such that faults can be fixed?

1.1 Thesis Structure

This thesis presents our contributions towards developing improved logic programming infrastructure to improve utility, user-friendliness, and ultimately the productivity of logic programmers. While designing and developing these systems, we find novel extensions to standard Datalog evaluation strategies, which we integrate into a production Datalog engine called Soufflé [35].

We begin our presentation by describing the well-established Datalog language. Chapter 2 provides a background of the history of Datalog before detailing the syntax and semantics of the language. Along the way, we motivate our study with a running example, describing the use of Datalog for program analysis. Finally, we briefly describe modern Datalog engines and the technical innovations that have allowed them to achieve high performance and strong scalability.

Chapter 3 discusses the major developments in the research areas spanning logic programming provenance, incremental evaluation, and automated fault localization and repair.

In Chapter 4, we present our novel approach for debugging large-scale Datalog. For this purpose, we present a *provenance* framework, which traces the origins and history of data. In the context of debugging Datalog programs, provenance in the form of proof trees can be used to discover the causes of faults that may appear as a result of bugs. To design the provenance system, we develop a novel encoding that uses two phases. In the first phase, we instrument the standard Datalog evaluation with *proof annotations* with minimal overhead, with a second phase to construct proof trees using these proof annotations. We also provide a user-friendly utility for exploring the trees in an on-demand fashion.

In Chapter 5, we describe our approach for automatic incremental evaluation in Datalog, intending to enable effective processing of incremental updates that are both small and large. We first describe the current state-of-the-art incremental evaluation techniques, then detail our adaptations to allow an *elastic* approach for incremental evaluation.

Using provenance and incremental evaluation, Chapter 6 describes our approach for incremental debugging for Datalog. The premise of incremental debugging is when a fault is introduced due to an incremental update. Our delta debugging approach uses a combination of

incremental evaluation and provenance, along with integer linear programming techniques, to automatically localize or even provide debugging suggestions for the causes of these faults.

Chapter 2

Datalog

Datalog is a declarative programming language based on the *logic programming* paradigm. Survey [11] provides a brief history of Datalog, from its origins to its gradual decline and resurgence. Originally, Datalog derived from the more expressive *Prolog* [36], with similar syntax and semantics but a different computational model. Datalog is a strict subset of Prolog, removing some features such as the non-declarative cut operator. As a result, Datalog has cleaner and simpler semantics while avoiding such problems as non-termination that plagued Prolog in its early days. At the same time, Datalog was of interest in the database community, being adopted as a query language that supports *recursion*, which was missing from the relational algebra-based languages in use at the time.

However, the popularity of Datalog declined in the late 1990s and early 2000s, primarily due to a lack of a 'killer application,' with recursion being less necessary than initially thought and more traditional languages such as SQL being favored in the database community. Furthermore, practical limitations such as memory space and the need for user interfaces and interactivity led to a rise in popularity for object-oriented languages, overshadowing the once-popular logic programming paradigm.

In recent years since the mid-2000s, Datalog has seen a revival of interest in various modern applications. For example, we present a non-exhaustive list of Datalog applications in use today:

- Semantic web [15] - a project which aims to classify webpages using logical statements, allowing for more relevant search results and recommendation systems
- Program analysis [20, 21, 22] - automatically detecting bugs or vulnerabilities in source code, using techniques such as abstract interpretation
- Declarative networking [16, 17, 18, 19] - describing networking algorithms to check properties such as connectivity, security, etc.
- Graph databases [37, 30] - expressing graph queries such as connectivity, reachability, etc.

The uptake of Datalog in these application areas and others has been driven by the succinctness of Datalog and the improving performance of modern engines. Traditionally, these applications have been developed in imperative or object-oriented languages, which are often less modular, less concise, and harder to optimize and maintain than Datalog.

The current state of Datalog is that it is commercially used in several specific application areas requiring complex recursive reasoning, with support for high performance and modern language features in current Datalog engines. While Datalog may not be a household name in programming languages, it is an essential tool in these domains.

2.1 Logic and Datalog

The Datalog programming language is a subset of first-order logic, a formalism of mathematical logic [34] that has a long and rich history dating back to early philosophers, such as Chrysippus in the 3rd century BC. Formal logic is a fundamental branch of mathematics that deals with mathematical concepts expressed using formal logical systems. Two important systems of logic are *propositional* logic and *first-order* logic. First, propositional logic allows the formal treatment of statements including components like *and*, *not*, and *if*. Then, first-order logic extends propositional logic to model modifiers like *exists*, *every*, and *only*.

Propositional Logic. Propositional logic is a formal language in which we can express declarative statements. For example, the sentence "if it is raining, or if it is nighttime, then we cannot play tennis" is a declarative statement that can be *true* or *false*. In propositional logic, we can define symbols (or *propositions*), for example r meaning "it is raining," n meaning "it is nighttime," and t meaning "we can play tennis." Then, the above sentence could be written as

$$r \vee n \implies \neg t$$

meaning that if r or n is true, then t is false, which gives the same meaning as the above English sentence.

Of course, we would also like to draw a conclusion from a given set of logical statements. In the above example, if we know that it is raining (i.e., that r is true), then we can conclude that we cannot play tennis (i.e., that t is false). Alternatively, if we are playing tennis (i.e., that t is true), then it logically follows that it is both not raining (i.e., that r is false) and that it is not nighttime (i.e., that n is false). To this end, *natural deduction* [38] is a system of proof rules which allow us to manipulate logical formulas to draw conclusions. For example, one proof rule in the natural deduction system is:

$$\frac{\phi \qquad \phi \implies \psi}{\psi}$$

This rule is named "implies elimination," meaning that if ϕ is true, and if $\phi \implies \psi$, then we can conclude that ψ is true. The natural deduction system also includes various other proof rules. In combination, these rules form a sound and complete system that can be used to draw conclusions from sets of logical formulas.

First-order Logic. While propositional logic is a fundamental framework for dealing with logical statements, it can be quite limited if we wish to consider more intricate properties of

objects. For example, consider the sentence "every parent is older than their child." Under propositional logic, it is impossible to reason about components such as *every* and *older than*. Thus, *first-order logic* (also called *predicate logic*) is an extension that allows *predicates* that define objects' properties. For the above example, we could define the predicate $\texttt{parent}(X, Y)$, which is true if X is the parent of Y, and the predicate $\texttt{older}(X, Y)$, which is true if X is older than Y. In first-order logic, we also have *quantifiers*, such as $\forall$ meaning "for all," and $\exists$ meaning "there exists." Then, the following first-order statement defines the same sentence as above:

$$\forall X\ \forall Y,\ \texttt{parent}(X, Y) \implies \texttt{older}(X, Y)$$

This statement can be read as "for all X and Y, if X is Y's parent, then X is older than Y," giving the same meaning as the above sentence.

Similar to propositional logic, we would wish to draw conclusions from a set of first-order statements, and so there are systems of deduction rules similar to natural deduction, which apply to first-order logic. However, the extra expressiveness of first-order logic means that it becomes *undecidable*. In other words, there does not exist a decision procedure that can determine whether an arbitrary first-order formula is logically valid.

Datalog as a Fragment of First-order Logic. The Datalog language is a subset of full first-order logic. In particular, Datalog programs are sets of Horn clauses, a restricted class of first-order formulas where each statement is of the form

$$(p \wedge q \wedge \ldots \wedge t) \implies u$$

containing a conjunction of predicates, implying a single predicate. Here, each p, q, etc., is a first-order predicate of the form $R(x_1, \ldots, x_n)$. This restricted Horn logic is *decidable*. Thus, in contrast to the undecidability of full first-order logic, the satisfiability of a set of Horn clauses can be computed in finite time (and, in fact, in polynomial time with respect to data size).

In Datalog, all formulas are implicitly universally quantified, i.e., a Datalog clause would be:

$$\forall x_1, \ldots, x_n :\ (R_1(x_1, \ldots) \wedge \ldots \wedge R_k(x_k, \ldots)) \implies R(x, \ldots)$$

Therefore, quantifier symbols are typically excluded when writing Datalog clauses.

Van Emden and Kowalski in 1976 [39] investigated the properties of Horn clauses when used as a programming language. This seminal work proposes a number of semantics, namely, an operational semantics, a model-theoretic semantics, and a fixpoint semantics, and demonstrates equivalences between these three semantics. Having established the utility of Horn clauses when used as a programming language, this work was the starting point of a long history of logic programming research.

2.2 Syntax of Datalog

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

$$R(X) \coloneq R_1(X_1), \ldots, R_n(X_n).$$

Here, a corresponding mathematical logic statement would be

$$\forall x_1, \ldots, x_k : \; R_1(X_1), \ldots, R_n(X_n) \implies R(X)$$

where $x_1, \ldots, x_k$ appear in $X_1, \ldots, X_n$. However, the convention in logic programming is to exclude the implicit universal quantifiers and write the implication in reverse order.

Each R_j is a *relation name*, and each X_j is a sequence of *terms* (terms are variables or constants) of correct arity. Each $R_j(X_j)$ is a *predicate*. The predicate $R(X)$ on the left of the $:-$ sign is the *head* of the rule and $R_1(X_1), \ldots, R_n(X_n)$ is the *body*.

A predicate $R(X)$ may be *instantiated*, with variables replaced by appropriate constants, to form a *fact* or *tuple*. An instantiated rule is a rule where each predicate is instantiated such that all variable substitutions are consistent between predicates. We say that a relation is extensional (or in the *extensional database*, EDB) if no predicates with that relation occur in the head of any rule or intensional (in the *intensional database*, IDB) otherwise. In other words, EDB relations can be seen as the *input* for a Datalog program, while IDB relations are derived through the execution of the program and can be intermediate or *output*. A tuple is EDB or IDB if the associated relation is EDB or IDB, respectively.

Semantically, a Datalog rule is read from right to left as a universally quantified implication: "for all rule instantiations, if every tuple in the body is derivable, then the corresponding tuple for the head is also derivable."

Constraints and Functions. Basic Datalog, as presented above, has simple syntax and semantics; however, it can be limited for practical uses. Thus, extensions are commonly added to Datalog implementations, such as constraints and functions.

Constraints such as $\leq$, $=$, etc. are added to the body of the rule and impose the meaning that an instantiated rule is only valid if the constraints are all satisfied. For example, the following rule

$$\texttt{sibling}(X, Y) :- \texttt{parent}(P, X), \texttt{parent}(P, Y), X \neq Y.$$

is interpreted as "if P is a parent of X, and P is a parent of Y, and X is not equal to Y, then X is a sibling of Y." These constraints are a powerful extension of Datalog, which allow the insertion of extra semantics based on the domains of terms.

Functions such as $+$, $-$, concat, max, etc. can be used in place of a term in a Datalog rule. For example, the following rule

$$\texttt{path}(X, Z, C_1 + C_2) :- \texttt{edge}(X, Y, C_1), \texttt{path}(Y, Z, C_2), C_1 + C_2 \leq 10.$$

computes lengths of weighted paths in a graph, up to a weight limit of 10. Note here that functions can be used inside predicates and constraints and denote the natural meaning of the function. Semantically, the use of functions extends the *expressivity* of Datalog, and usual guarantees such as termination and decidability no longer hold under Datalog with functions.

Stratification and Negation. Predicates in Datalog rules may also be negated, denoted with a ! symbol. For example, the following rule contains negated predicates $!R_{k+1}(X_{k+1}), \ldots, !R_n(X_n)$.

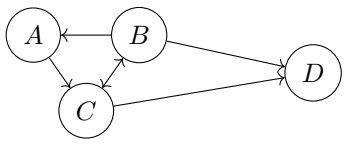
$$R(X) :- R_1(X_1), \ldots, R_k(X_k), !R_{k+1}(X_{k+1}), \ldots, !R_n(X_n)$$

A negated predicate must contain only variables that also appear in some positive predicate in the rule body, a property known as groundedness. Semantically, a negated predicate holds true if the corresponding tuple is *not* true in the database.

With the possibility of recursion in Datalog, the semantics of negation become more involved. There are several possible semantics for negation, with the most widely adopted in modern Datalog engines being *stratified negation*. Stratified negation avoids the issues associated with cyclic negations, where the semantics of a program can become ambiguous (i.e., there may be multiple correct solutions if cyclic negation is allowed). To explain the stratified negation semantics, we first explain stratification.

A Datalog program can be *stratified* by constructing a dependency graph with relation names as nodes and edges going from relation A to relation B if there is a rule with A in the body and B as the head. Then, each strongly connected component of the dependency graph is a *stratum*, with an ordering over strata determined by the topological order over the SCC graph. For example, consider the following Datalog program, shown with its dependency graph.

$$A :- B.$$
$$B :- C.$$
$$C :- A, B.$$
$$D :- B, C.$$



Here, relations A, B, and C are mutually recursive, so they would form one stratum. Then D depends only on B and C, so is in its own stratum. Therefore, a stratification of the above Datalog program is $\{A, B, C\}, \{D\}$.

In stratified negation, any negated predicates in a rule must have a relation from an earlier stratum than the head of the rule. For example,

$$A :- B.$$
$$B :- C.$$
$$C :- A, !B.$$

would be disallowed under stratified negation since C and B are in the same stratum. By enforcing this stratification, cyclic negations are avoided, which is important for the fixpoint semantics of Datalog. Each stratum is evaluated separately to evaluate a stratified program in order of the stratification. Then, any negations that appear in some stratum must be of relations in an earlier stratum. The truth value of these negations is a simple check for the existence of the corresponding tuple in the relation, which was already computed in the earlier stratum.

2.3 Running Example

One of the major applications of modern Datalog is specifying static program analysis problems. Static program analysis is the process of automatically analyzing the possible behaviors of a program. This is a crucial process in many aspects of software development, for example, finding optimization opportunities in compilers, checking for security and correctness properties such as dangling references or uninitialized memory, providing assistance tools in IDEs, ensuring business compliance requirements, among many other applications. For a more comprehensive overview, [40] gives a detailed explanation of static program analysis.

Static program analysis is concerned with checking properties that hold for *all* possible program inputs. For example, suppose a program analysis says there are no possible null pointer exceptions. In that case, this property holds regardless of program input, and therefore, the source program will never encounter a null pointer exception. Of course, depending on the property to check, verifying it for all inputs is generally impossible due to the *halting problem* [41]. Therefore, all program analyses employ conservative *abstractions* [42] of actual program semantics. For example, a sign analysis could be performed by abstracting all numbers in the program to being positive, negative, or zero. Then, operations in the program are assigned to semantics in the abstract domain, e.g., `positive`+`positive` = `positive`, and `positive`+`negative` = `unknown`. This abstraction covers all possible behaviors of numbers, and is conservative in that an `unknown` value may take on any `positive`, `negative`, or `zero` value.

One form of static program analysis is *pointer analysis*. A pointer analysis finds relationships between pointers and memory cells, namely, which pointers in a program may point to which memory cells. The abstraction used in pointer analysis is that an arbitrary set of possible memory cells is abstracted to *object creation sites*. For instance, if line 100 of a source program contains a `new` statement, then line 100 is treated as a representation of all objects created from that `new` statement. Pointer analysis is an important problem for finding relationships such as *aliasing*, where two pointers may point to the same object, which has implications for security or parallelism opportunities. In general, a simple pointer analysis handles the following kinds of program statements:

- Allocation `x = new Object()`, where `x` now points to the new created object
- Assignment `x = y`, where any memory cells pointed to by `y` may now also be pointed to by `x`
- Load `x.f = y` and store `y = x.f`, which propagates pointer information similarly to assignment
- etc., different implementations of pointer analysis may consider different kinds of statements

The result of a pointer analysis would be a relation `VariablePointsTo`. For example, if this relation contains the tuple `VariablePointsTo(sum,object1)`, then the variable `sum` may point to the object `object1` during the execution of the source program.

```
var admin = new Admin();
var sec = new AdminSession();
var ins = new InsecureSession();
var userSession;
var superuser;

admin.session = ins;

if (admin.isAdmin && admin.isAuth) {
    admin.session = sec;
    superuser = sec;
} else {
    userSession = ins;
    superuser = userSession;
}
superuser = admin.session;
```

(a) Source Program

```
new(admin,L1).
new(sec,L2).
new(ins,L3).
new(userSession,nullptr).
new(superuser,nullptr).

store(admin,session,ins).


store(admin,session,sec).
assign(superuser,sec).

assign(userSession,ins).
assign(superuser,userSession).

load(superuser,admin,session).
```

(b) EDB Tuples

```
vpt(Var, Obj) :- new(Var, Obj).     //r1

vpt(Var, Obj) :- assign(Var, Var2),
                 vpt(Var2, Obj).    //r2

vpt(Var, Obj) :- load(Var, Inter, F),
                 store(Inter2, F, Var2),
                 vpt(Inter, InterObj),
                 vpt(Inter2, InterObj),
                 vpt(Var2, Obj).    //r3

alias(Var1, Var2) :- vpt(Var1, Obj),
                     vpt(Var2, Obj),
                     Var1 != Var2,
                     Obj != nullptr. //r4

safevar(Var) :- vpt(Var, _),
                !vpt(Var, nullptr). //r5
```

(c) Datalog Points-to Analysis

Figure 2.2: Program Analysis Datalog Setup

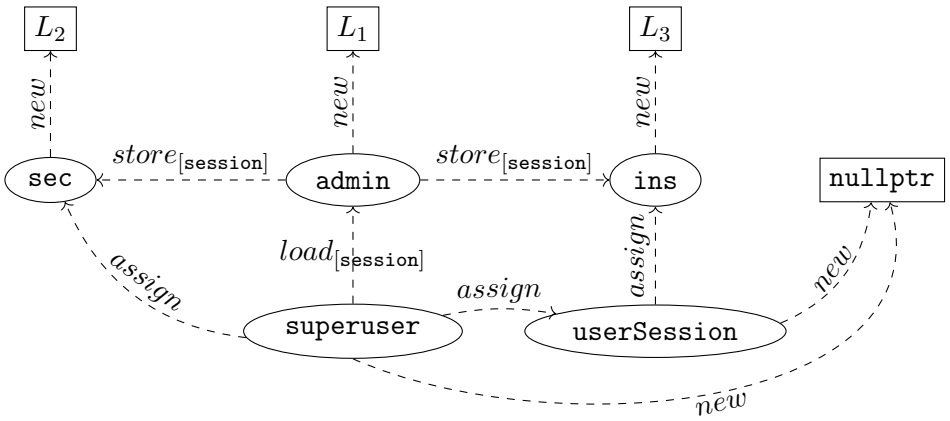


Figure 2.3: Points-to Input Diagram

2.3.1 Pointer Analysis in Datalog

We illustrate Datalog as a language for expressing pointer analyses through Figure 2.2. In this scenario, we have a source program to analyze (Figure 2.2a). To perform the analysis, the source program is encoded as a set of Datalog tuples (Figure 2.2b). In this case, we have tuples of relations `new`, `assign`, `load`, and `store`, where each tuple encodes a line of the source program. This set of tuples forms the EDB input for the Datalog-based analysis. The Datalog program (Figure 2.2c) computes the relations `vpt` (short for `VariablePointsTo`, representing variables which may point to objects), `alias` (representing pairs of variables which may point to the same object), and `safevar` (representing variables which do not point to the special `nullptr` object).

Figure 2.3 shows a diagrammatic representation of the pointer relationships in the source program. Objects are represented as rectangles, and variables are represented as ellipses. Edges represent input EDB relations, for example the edge labeled *assign* going from `userSession` to `ins` represents a tuple `assign(userSession,ins)`.

For the points-to analysis, the Datalog program (Figure 2.2c) consists of five Datalog rules. For example, the rule r_2 is

$$\texttt{vpt}(\texttt{Var},\texttt{Obj}) :- \texttt{assign}(\texttt{Var},\texttt{Var2}),\ \texttt{vpt}(\texttt{Var2},\texttt{Obj}).$$

This rule, r_2, can be interpreted as "if we have an assignment from `Var` to `Var2`, and if `Var2` may point to `Obj`, then also `Var` may point to `Obj`".

In combination, the five Datalog rules represent a *flow-insensitive* but *field-sensitive* points-to analysis. The IDB relations `vpt`, `alias`, and `safevar` represent the result of the analysis, computing variables which may point to objects, pairs of variables that may alias with each other, and variables that cannot point to `nullptr`, respectively.

2.4 Semantics and Evaluation of Datalog

There are three main approaches to defining semantics for Datalog programs: model-theoretic semantics, fixpoint semantics, and proof-theoretic semantics. Model-theoretic semantics provides

a definition for Datalog semantics, but it does not correspond to any concrete algorithm for evaluation. On the other hand, fixpoint semantics and proof-theoretic semantics provide algorithms for Datalog evaluation, corresponding to *bottom-up* and *top-down* evaluation, respectively.

Bottom-up evaluation is used in modern Datalog systems such as Soufflé [35] and LogicBlox [24] and tends to exhibit better performance for large-scale Datalog applications with large database sizes. Top-down evaluation is employed by older systems such as XSB [43] and is similar to standard Prolog evaluation. Basic top-down evaluation loses some guarantees of Datalog, such as termination, but modern implementations employ techniques such as tabling to mitigate these limitations. Both evaluation strategies, along with model-theoretic semantics, are discussed in more detail in [12, 44].

2.4.1 Model-theoretic semantics

The model-theoretic semantics of a Datalog program is purely definitional and does not provide an algorithm to compute the result of a Datalog program. The model-theoretic semantics is defined in terms of a *Herbrand base*. We refer to a Datalog program P with some database D (a database is a set of tuples) as P_D. Then, the *Herbrand universe* H_{P_D} is the set of all constants that appear in the database. The *Herbrand base* B_{P_D} is the set of all possible tuples constructed by taking predicates in P_D and substituting variables with constants from H_{P_D}.

Given these basic definitions, we begin to define a model-theoretic semantics. An *interpretation* of P_D is some subset $I \subseteq B_{P_D}$. Now, let I be an interpretation and consider a rule r in P:

$$R(X) :- R_1(X_1), \ldots, R_k(X_k)$$

We say that the interpretation I satisfies r (i.e., $I \models r$) if for every instantiation of r, when tuples corresponding to the body literals $R_1(X_1), \ldots, R_k(X_k)$ are in I, then the corresponding head tuple $R(X)$ is also in I. An interpretation I is a *model* of P_D if I satisfies *every* rule in P.

If χ is the set of all models of P_D, then $\cap\chi$ is the *minimal* model of P_D. The model-theoretic semantics of P_D is defined to be this intersection $\cap\chi$ and is minimal in the sense that it includes only the tuples necessary to form a model and no extra tuples.

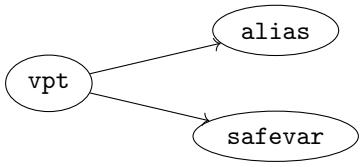
2.4.2 Bottom-Up Evaluation

Bottom-up evaluation of a Datalog program P, which corresponds to the fixpoint semantics, describes a general method of starting from the EDB tuples to generate all IDB tuples. For Datalog, the basic bottom-up evaluation strategy is termed *naïve evaluation*. The process begins from an instance I of P, containing only EDB facts. Then, an *immediate consequence* of I is a fact t such that either $t \in I$, or $t \leftarrow t_1, \ldots, t_n$ is a valid instantiation of a rule with each $t_i \in I$. Then, the main operation for naïve evaluation, the *immediate consequence operator*, Γ_P maps one instance of the program to another by applying immediate consequences:

$$\Gamma_P : inst(P) \to inst(P)$$
$$\Gamma_P(I) = I \cup \{t \mid t :\!- t_1, \ldots, t_n \text{ is an instantiated rule with all } t_i \in I\}$$

It can be seen that Γ_P is monotone (with stratified negation, evaluating stratum by stratum in order gives monotonicity in each stratum), and using *Knaster-Tarski's Fixpoint Theorem* [45], we can show that there exists a minimum fixpoint of Γ_P [12]. Thus, the naïve evaluation algorithm applies Γ_P to the input instance repeatedly, until a fixpoint is found. 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 2.4. Since this program contains a stratified negation, we show the stratification of the IDB relations:



In the following example, the initial input is denoted E, consisting of only the input tuples directly corresponding to the source program. For the first stratum, in the first iteration, the evaluation applies the non-recursive rule r_1 to compute the initial `vpt` tuples. Then, in the second iteration, the recursive rules r_2 and r_3 are applied. In the third iteration, no new `vpt` tuples can be discovered, so a fixpoint is reached.

$$E = \left\{ \begin{array}{l} \texttt{new(admin,L1)},\ \texttt{new(sec,L2)},\ \texttt{new(ins,L3)},\ \texttt{new(userSession,nullptr)}, \\ \texttt{new(superuser,nullptr)},\ \texttt{store(admin,session,ins)}, \\ \texttt{store(admin,session,sec)},\ \texttt{load(superuser,admin,session)}, \\ \texttt{assign(userSession,ins)},\ \texttt{assign(superuser,sec)} \end{array} \right\}$$
$$\Gamma_P(E) = E \cup \left\{ \begin{array}{l} \texttt{vpt(admin,L1)},\ \texttt{vpt(sec,L2)},\ \texttt{vpt(ins,L3)}, \\ \texttt{vpt(userSession,nullptr)},\ \texttt{vpt(superuser,nullptr)} \end{array} \right\}$$
$$\Gamma_P^2(E) = \Gamma_P(E) \cup \{\texttt{vpt(userSession,L3)},\ \texttt{vpt(superuser,L2)},\ \texttt{vpt(superuser,L3)}\}$$
$$\Gamma_P^3(E) = \Gamma_P^2(E)$$

Figure 2.4: Bottom-up evaluation of the `vpt` stratum of the running example (Figure 2.2c), where E is the input instance

In the subsequent strata, the evaluation separately computes `alias` and `safevar`.

$$\Gamma_P^4(E) = \Gamma_P^3(E) \cup \left\{ \begin{array}{l} \texttt{alias(userSession,ins)},\ \texttt{alias(superuser,sec)}, \\ \texttt{alias(superuser,ins)} \end{array} \right\}$$

When computing `safevar`, the negation must be satisfied. Since the `vpt` relation is already at fixpoint, the negation is equivalent to checking a constraint, which is satisfied if the corresponding

tuple is not contained in `vpt`.

$$\Gamma_P^5(E) = \Gamma_P^4(E) \cup \{\texttt{safevar(admin)},\ \texttt{safevar(sec)},\ \texttt{safevar(ins)}\}$$

Algorithm 1 Naïve(P, E)

1: $I_0 \leftarrow E$
2: **for all** $k \in \{1, 2, \ldots\}$ **do**
3: $\quad I_k \leftarrow \Gamma_P(I_{k-1})$
4: $\quad$ **if** $I_k = I_{k-1}$ **then**
5: $\quad\quad$ **return** I_{k-1}
6: $\quad$ **end if**
7: **end for**

Formally, naïve evaluation is presented in Algorithm 1 for a single stratum. The algorithm follows a simple fixpoint structure, where the immediate consequence operator is applied in each iteration until a fixpoint is reached.

Semi-naïve Evaluation. While the naïve evaluation presented above demonstrates a standard bottom-up evaluation, it is sub-optimal in practice. Naïve evaluation will repeat computations since a tuple computed in some iteration will be recomputed in every subsequent iteration. Therefore, the standard implementation of bottom-up evaluation in real systems such as [35, 46] is *semi-naïve*. Semi-naïve evaluation contains one main optimization over naïve evaluation, which we call the *new knowledge optimization*.

For the new knowledge optimization, in each stratum's fixpoint computation, the evaluation is optimized in each iteration by considering the new tuples generated in the previous iteration. A new tuple is generated in the current iteration only if it directly depends on tuples generated in the previous iteration, therefore avoiding the recomputation of tuples already computed in prior iterations.

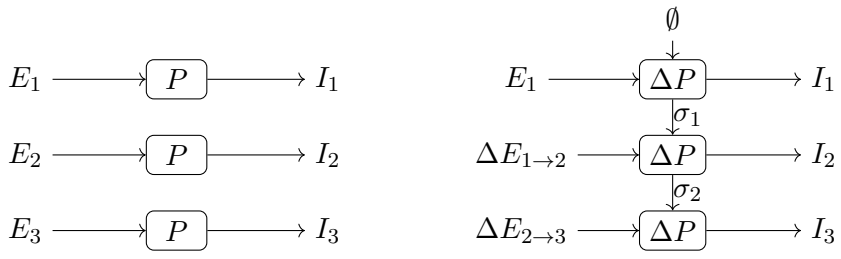
Algorithm 2 Semi-Naïve(P, E)

1: $\Delta_0 \leftarrow E$
2: **for all** $k \in \{1, 2, \ldots\}$ **do**
3: $\quad I_{k-1} \leftarrow \bigcup_{0 \le i < k} \Delta_i$
4: $\quad \Delta_k \leftarrow \{t \mid t :- t_1, \ldots, t_n$ is an instantiated rule with all $t_i \in I_{k-1}$ and some $t_i \in \Delta_{k-1}\} \setminus I_{k-1}$
5: $\quad$ **if** $\Delta_k = \emptyset$ **then**
6: $\quad\quad$ **return** I_{k-1}
7: $\quad$ **end if**
8: **end for**

Formally, semi-naïve evaluation is shown in Algorithm 2 for a single stratum. Compared to naïve evaluation, the main difference is the additional auxiliary sets Δ_i, which contain the newly generated tuples in iteration i. During the main rule evaluation step (line 4), the immediate consequence operator is extended so that at least one tuple in the body of the instantiated rule is contained in Δ_{k-1}. In other words, at least one body tuple should be newly computed in

the immediately previous iteration. As a result, previously generated tuples are not recomputed under semi-naïve evaluation.

Incremental Evaluation. A recent development in Datalog is in *incremental evaluation* [47, 48, 49, 50]. The main idea of incremental evaluation is to be able to update the result of a Datalog evaluation given some changes to the input (i.e., deleting or inserting some input tuples) without discarding and recomputing the result from scratch. Therefore, incremental evaluation is considerably more complex than other bottom-up evaluation strategies; however, it provides numerous benefits for large-scale Datalog programs where a large input set may change only slightly between runs of the Datalog program.



(a) Standard (b) Incremental

Figure 2.5: Standard vs. Incremental Evaluation

The general architecture of an incremental evaluation is presented in Figure 2.5, which shows how an evaluation proceeds for three *epochs* (an epoch is one round of evaluating a Datalog program given an input set). Figure 2.5a shows a standard bottom-up evaluation, where each of the three epochs is evaluated entirely independently of each other. In contrast, Figure 2.5b shows an incremental evaluation. This incremental evaluation carries a *computation state*, denoted σ, which allows the subsequent epoch to re-use computations from the previous epoch. For example, the second epoch uses the computation state from the first epoch and the input changes (the deletions and insertions, denoted $\Delta E_{1\to2}$) to produce the same resulting output I_2 more efficiently.

Semi-naïve evaluation can effectively handle input tuple insertions by treating new insertions as delta knowledge. However, deletions and negations are more challenging, and several different strategies have been proposed in the literature [47, 48, 49, 50]. We further discuss incremental evaluation in Section 3.2 and Chapter 5.

2.4.3 Top-Down Evaluation

While most of this thesis concerns extending bottom-up evaluation methods, Chapter 4 uses concepts of top-down evaluation and the related notion of *proof trees*. Therefore, we present a basic introduction to top-down evaluation in this section.

Top-down evaluation describes the opposite method of bottom-up evaluation, where top-down starts from a query and checks in each step whether there are rules or facts in the program that make the query satisfiable. The query takes the form of a *goal clause*, which is a sequence

of predicates. Generally, a query in top-down evaluation is given by the user, who wishes to check if a certain set of tuples is derivable by the program.

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

A top-down evaluation, such as *SLD resolution*, considers each predicate A_i in the goal clause and searches for some rule with A_i as the head. Constants may need to be substituted for variables, so they match (a process known as *unification*), then A_i in the goal clause is replaced with the body of that rule. This step is applied repeatedly until either we reach EDB facts, in which case the original goal clause holds true, or a valid instantiation cannot be found at some point, in which case the original goal clause does not hold.

An example of top-down evaluation is given in Figure 2.6

$$\begin{aligned}
&\leftarrow \texttt{vpt}(\texttt{userSession}, \texttt{L3})\\
&\leftarrow \texttt{assign}(\texttt{userSession}, \texttt{Var2}), \texttt{vpt}(\texttt{Var2}, \texttt{L3})\\
&\leftarrow \texttt{assign}(\texttt{userSession}, \texttt{ins}), \texttt{vpt}(\texttt{ins}, \texttt{L3})\\
&\leftarrow \texttt{assign}(\texttt{userSession}, \texttt{ins}), \texttt{new}(\texttt{ins}, \texttt{L3})\\
&\leftarrow \square
\end{aligned}$$

Figure 2.6: Example top-down evaluation for our running example (Figure 2.2c) showing $\texttt{vpt}(\texttt{userSession}, \texttt{L3})$ holds

However, basic top-down evaluation requires a notion of backtracking since it is unspecified which rule will be selected out of many possible rules matching a particular goal predicate. If it selects a rule that cannot generate an instantiation of A_i, backtracking may be needed 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 terminates.

The computation process of top-down evaluation can also be viewed as a tree. Formally, this is known as a *proof tree*, which shows the steps taken to prove that a tuple holds true. For the above example (Figure 2.6), a proof tree is as follows.

$$\dfrac{\texttt{assign}(\texttt{userSession}, \texttt{ins}) \qquad \dfrac{\texttt{new}(\texttt{ins}, \texttt{L3})}{\texttt{vpt}(\texttt{ins}, \texttt{L3})}}{\texttt{vpt}(\texttt{userSession}, \texttt{L3})}$$

Proof trees are also important as a form of *provenance*, discussed in Section 3.1 and Chapter 4. In particular, proof trees are especially relevant for debugging and are one of the advantages of top-down evaluation compared to bottom-up.

2.5 Datalog Engines

While Section 2.4 discusses bottom-up and top-down evaluation at an algorithmic level, real-world implementations of Datalog engines require sophistication in data structures, parallelization and other aspects, to achieve the high performance necessary for modern real-world workloads. Modern Datalog engines typically employ bottom-up evaluation (with some exceptions) due to the characteristics of modern applications that require higher efficiency for computing large amounts of output data.

We present a non-exhaustive list of modern Datalog engines:

- μZ [51] - μZ is a bottom-up solver for fixed points with constraints, based on the Z3 SMT solver. It includes several extensions over standard Datalog, with the ability to plug in alternative data structures, abstract relations, lattices, etc.
- LogicBlox [24] - LogicBlox is a commercial bottom-up Datalog engine designed to be accessible and easy to use but with enough performance to tackle large program analysis tasks.
- BDDBDDB [46] - BDDBDDB is a bottom-up Datalog engine using binary decision diagrams (BDDs) to represent relations. It was designed and specialized for program analysis use cases.
- Differential Datalog [52] - Differential Datalog is an incremental Datalog engine built on top of the Differential Dataflow framework [50]. The incremental evaluation allows to insert or delete input tuples to *update* the result of a computation efficiently.
- Socialite [37] - Socialite is another bottom-up Datalog engine designed for large-scale network analyses such as for social networks. It supports features such as recursively defined aggregates which provide concise semantics for these large-scale network applications.
- Coral [53] - Coral was an early Datalog engine, which was one of the earliest implementations of a bottom-up deductive database system.
- XSB [43] - XSB was developed at a similar time to Coral, but XSB instead uses a top-down approach. One of its main innovations is implementing *SLG resolution*, which provides extensions over SLD resolution such as tabling to mitigate the termination and performance issues of standard top-down.

However, in this thesis, our work focuses on extending *Soufflé*.

2.5.1 Soufflé

Soufflé [35] is a modern Datalog engine that uses a semi-naïve bottom-up evaluation strategy. It uses one of two main methods for executing Datalog programs: (1) using Datalog source code to synthesize high-performance parallel C++ code, or (2) interpreting a Datalog program directly. Figure 2.7 shows a flow chart of the execution of a Datalog program using Soufflé. In

the first stages, a Datalog source program is translated into an Abstract Syntax Tree (AST) and then into an intermediate representation called Relational Algebra Machine (RAM). Then, a RAM program is either synthesized into C++ code or interpreted directly. In either option, the execution takes a set of input facts (the EDB) and produces an output result (the IDB).

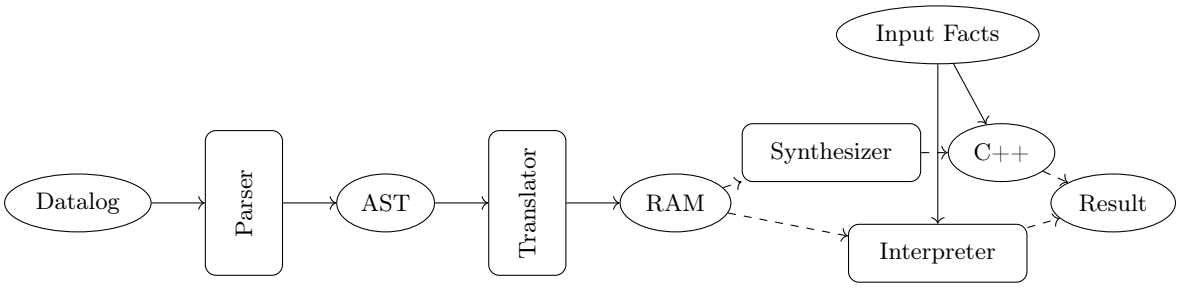


Figure 2.7: Flow chart of Soufflé

To illustrate the translation process, consider the rule r_2 in our running example:

$$\texttt{vpt}(\texttt{Var}, \texttt{Obj}) :- \texttt{assign}(\texttt{Var}, \texttt{Var2}), \texttt{vpt}(\texttt{Var2}, \texttt{Obj}).$$

During the execution of this rule, r_2 is translated into the RAM code shown in Figure 2.8. RAM is an intermediate representation that combines relational algebra operations and imperative constructs such as loops and conditions.

```
1  LOOP
2    IF ((NOT (assign = ∅)) AND (NOT (delta_vpt = ∅)))
3      FOR a IN assign
4        FOR b IN delta_vpt ON INDEX b.0 = a.1
5          IF (NOT (a.0, b.1) ∈ vpt)
6            INSERT (a.0, b.1) INTO new_vpt
7
8    ... // other vpt rules
9
10   BREAK IF (new_vpt = ∅)
11   MERGE new_vpt INTO vpt
12   SWAP (delta_vpt, new_vpt)
13   CLEAR new_vpt
14 END LOOP
```

Figure 2.8: RAM code for rule r_2

The above RAM code includes auxiliary relations for recursively defined relations, `new_vpt` and `delta_vpt` in this case, which are required to keep track of tuples that are new in the current iteration, and new in the previous iteration respectively. These auxiliary relations allow for semi-naïve evaluation of the recursive rule. The structure of the RAM code is a *loop nest*, which iterates through each relation in the body of the rule in sequence, ensuring at each stage that the variable bindings are compatible through the use of an index (line 4). Then, if the

newly generated tuple is actually new (i.e., not in `vpt`), it is inserted into `new_vpt` (line 6). The remaining lines perform book-keeping operations to ensure that `new_vpt` and `delta_vpt` are in the correct state for the next iteration.

In general, a recursive rule under semi-naïve evaluation is evaluated using a number of different *versions*, where a different literal in the body of the rule becomes delta in each version. For example, a rule

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

is translated to the following, where each literal in the body of the rule becomes delta in each version of the rule. In the following rules, R_k^i denotes the state of relation R_k in iteration i.

$$\begin{aligned}
\texttt{new}_{R_0}^{i+1}(X_0) &:- \texttt{delta}_{R_1}^{i}(X_1), R_2^i(X_2), \ldots, R_n^i(X_n) \\
&\ldots \\
\texttt{new}_{R_0}^{i+1}(X_0) &:- R_1^{i-1}(X_1), \ldots, \texttt{delta}_{R_k}^{i}(X_k), \ldots, R_n^i(X_n) \\
&\ldots \\
\texttt{new}_{R_0}^{i+1}(X_0) &:- R_1^{i-1}(X_1), \ldots, R_{n-1}^{i-1}(X_{n-1}), \texttt{delta}_{R_n}^{i}(X_n)
\end{aligned}$$

Each version of the rule in the Soufflé translation pipeline, becomes its own loop nest (corresponding to lines 3 to 6 in Figure 2.8). The relation book-keeping operations (corresponding to lines 11 to 13) are performed at the end of all loop nests for all relations in the stratum.

The translation from RAM to C++ code is rather natural given the procedural nature of RAM. The main challenge is to use high-performance data structures to store relations, with support for fast reads, insertions, and index operations.

Data Structures. The main requirements for a high-performance data structure for Datalog evaluation are detailed in [5]. This work introduces the concept of a *Datalog-Enabled Relational* (DER) data structure. The functional requirements of such a data structure are that it must support the following operations:

- `insert(Tuple t)` - inserting a fixed-size n-ary tuple, ignoring duplicates
- `begin()` and `end()` - provide iterators to traverse all tuples in the relation
- `lower_bound(Tuple t)` and `upper_bound(Tuple t)` - provide iterators to the lower bound or upper bound of a query tuple, according to some predefined order
- `find(Tuple t)` - provides an iterator to a query tuple if it is present
- `empty()` - whether the relation is empty

Furthermore, performance can also be increased by using *concurrent* data structures, which take advantage of modern CPUs having multiple cores. In a concurrent context, any DER data

structure must ensure that concurrent insertions are handled correctly and concurrent reads are correct (however, interleaved reads and insertions never happen due to the nature of loop nests for semi-naïve Datalog evaluation).

The paper [8] presents a specialized concurrent B-tree, explicitly designed for Datalog evaluation workloads. The B-tree data structure is a natural fit that fulfills all of the functional requirements for a DER data structure. Thus, the main design contribution in [8] is a locking mechanism to enable concurrent Datalog evaluation. The proposed solution is to use an *optimistic read-write lock* for each node. This locking scheme assumes that any thread accessing a node is reading the node (e.g., to find the correct position to perform an insertion), and if any modifications are to be made, then the lock is upgraded to allow writing.

Index Selection. While the data structures presented above are critical for high-performance parallel Datalog evaluation, they do not form the complete story. Another important aspect of Soufflé's performance is specializing the data structures to suit the operations performed in the RAM program. This process is called *automatic index selection* [54], and it adapts the data structures for each relation depending on the operations performed on that relation. For example, consider the RAM program in Figure 2.8. Line 4 performs an index operation on the `delta_vpt` relation, requiring that the data structure should be able to quickly find tuples where the first element (`b.0`) matches some given value. For this requirement, the data should be sorted based on the lexicographical order where the first element comes first (i.e., the sort order $a < b \iff a[0] < b[0]$ or $(a[0] == b[0]$ and $a[1] < b[1]))$ to form an index. Therefore, the data structure storing `delta_vpt` should be specialized so that it holds binary tuples, sorted with the above lexicographical order.

In general, a Soufflé RAM program may have multiple different operations on each relation, requiring multiple different index orders. Therefore, the *minimum index selection problem*, as defined in [54], aims to find the minimum number of indexes (i.e., lexicographical orders) required to accelerate all index operations on the relation.

Chapter 3

Related Work

This thesis covers several developments in Datalog evaluation strategies. While modern Datalog engines and language features enable effective programming and evaluation of large-scale Datalog programs, other features such as explainability, incrementalization, and debugging are still a challenge. Therefore, this thesis focuses on extending a traditional Datalog evaluation to support provenance, incremental evaluation, and debugging features. This chapter surveys the important works in these areas to put our contributions in context.

3.1 Provenance

Data provenance describes a general concept of tracing the path that data takes through a system. Provenance information allows us to trace the origins and history of data, for example, to track which inputs to a query are used to compute which outputs. Such information is vital for many reasons, including auditing, trustworthiness, debugging, and, more generally, understanding the origins of output data.

For instance, Cheney et al. [55] argue that the necessity of provenance is driven by the increasing ease of creating and storing data, and that 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. For this database domain, provenance is aptly defined in [56] as "an explanation of the ways [facts] are derived." This provenance information can also provide valuable information for debugging database queries, as described in [57]. In the Datalog world, which this thesis focuses on, provenance is closely related to *proof trees*, an approach used in practice by [58].

3.1.1 Classification of Provenance

Provenance can be classified in several different ways, according to the desired semantics. The survey [55] presents three of the possible approaches: *why*, *how*, and *where*-provenance.

***Why*-provenance.** *Why*-provenance, which is closely related to the notion of *lineage*, describes a relationship between the output tuples of a Datalog program with input tuples. In-

tuitively, the lineage of an output tuple is a subset of input data that 'contributed to' the production of that output tuple.

More precisely, [55] defines the why-provenance of an output tuple t as 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. Linking to proof trees, [59] shows that the set of tuples in the witness basis corresponds to the leaves of a proof tree for t.

The lineage framework is advantageous because it does not include the internal semantics of queries. Thus, lineage and why-provenance is a suitable framework for developing provenance mechanisms targeted to the end-users of a database system, who may not know the internal query operations. For instance, SubZero [60] and Taverna [61] are both scientific databases targeted towards bioinformatics and other life science users. These scientific databases also include a utility for lineage, which scientists can use to understand the sources of their data better.

***How*-provenance.** While why-provenance is a powerful framework for understanding the relationships between input and output data, it does not provide additional information about *how* the output tuple is derived. To this end, *how*-provenance is a generalization of why-provenance that describes how the operators in a query are used to derive the output. The paper [62] proposes a framework of using *semirings* [1] to describe provenance.

In [62], the authors propose to label each tuple by an element in the provenance semiring. This semiring element then describes the how-provenance of the corresponding tuple. Formally, this approach defines some semiring $(K, +, \cdot, 0, 1)$ (here, $+$, $\cdot$, 0, 1 are abstract objects, not related to the standard meaning of these symbols), and a function mapping from tuples to K:

$$f : \{t\} \to K$$

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

$$\sum_{\text{proof trees } \tau \text{ deriving } t} \left(\prod_{\text{tuples } t' \in \text{leaves of } \tau} f(t') \right)$$

Through this structure, the how-provenance of a tuple closely corresponds with its proof trees, where the resulting sum of products directly describes the leaves of the proof trees.

This general framework of using semirings to describe provenance has led to numerous concrete applications and systems. For example, probabilistic databases [63] and databases with uncertainty [64] use the set $[0, 1]$ as the basis of the semiring for tracking how probabilities propagate through a query. Then, the probability of an output tuple can be computed by propagating the probabilities of the inputs through the semiring operators. Another use of the semiring

[1] A semiring is an abstract 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.

framework is for trust and access control [65, 66], where trust or access policies are encoded as provenance information propagated to query outputs.

***Where*-provenance.** While why-provenance and how-provenance deal with the origins of *tuples* as they are computed through a query, where-provenance takes a different approach to describe the provenance of each element in a tuple. The paper [59] states that it is closely connected to why-provenance since the where-provenance of any value in an output tuple is a subset of the witnesses of that tuple. Formally, [59] provides a constructive definition of where-provenance, referred to as a *derivation basis*. A derivation basis traces the paths through a query, collecting all values along the way which derive the particular output elements. The survey [55] then 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 the ideal framework for applications where tracing the paths of data elements is important. For instance, annotation management systems [67, 68] use where-provenance to track and explain to users where specific output values are computed from. For applications such as debugging queries, where-provenance may also be used to discover parts of rules that lead to incorrect output values. However, this approach has not been explored in the literature as far as we are aware.

3.1.2 Provenance in Datalog

One of the main methods of encoding provenance information in a Datalog execution is rewriting the Datalog program to capture information such as rule firings or an execution graph. A number of previous approaches adopted this strategy [58, 57, 69, 56, 70, 71]. A common thread in each of these techniques is that they rewrite a Datalog program to store the full provenance information.

For example, [57] rewrites Datalog into Statelog [72] or more complex Datalog, with the resulting program capturing a provenance graph. One of the applications of this full provenance encoding is in profiling. For example, [57] demonstrate how the provenance system can be used to compare different re-writings of the transitive closure program in terms of the number of intermediate tuples, rule firings, and other statistics. This demonstrates that provenance can be used not just for debugging but also for profiling small Datalog instances.

However, the weakness of this approach is that it stores the full provenance information, which may be prohibitively large for real-world Datalog instances. This is reflected in the experiments of [57], in that the largest experiment presented was a transitive closure example with 1710 nodes and 3936 edges, containing 304,000 paths. This result suggests that the full provenance storage approach does not scale well to large databases containing millions of output tuples.

To avoid storing full provenance information, [56] introduces the notion of selective provenance. In this approach, the concept of a *derivation tree pattern* is proposed as a specification language to describe the desired provenance output. The derivation tree patterns are related to tree grammars, and a particular pattern provides a specification that all generated proof trees must match. Then, the Datalog program is instrumented based on the given derivation tree

pattern, such that the program computes the relevant proof trees.

A similar notion of *provenance questions* is introduced in [71], which allows a user to specify a pattern for the desired provenance output. Then, the Datalog program is instrumented based on the provenance question to produce explanations that match the question. One advantage of [71] is that it allows the explanation of *missing* answers. However, since it is largely impractical to explain why something is missing, the proposed solution requires the user to annotate the desired domain to perform a constrained search for all possible derivations of the missing tuple.

The main disadvantage of the strategies in [56, 71] is that the program needs to be rewritten and reevaluated for each new provenance query. Since Datalog evaluation can be fairly expensive, it may be prohibitive to perform this rewriting. However, the flexibility of derivation tree patterns means that the instrumented Datalog itself can be more efficient for certain small patterns. In experiments in [56], it is shown that selective provenance information for a small derivation tree pattern can be computed in a reasonable time for databases with 1.7M output tuples. However, computing full provenance information for this instance is still prohibitively costly, taking over 6.5 hours on a standard desktop computer.

Another important aspect of provenance in Datalog is the memory overhead. Since storing full provenance naïvely can be rather expensive, techniques such as [69] aim to improve on this. In [69], the authors propose using *Boolean circuits* to store provenance information. The main advantage of Boolean circuits is the ability to compress repetitive parts of the formula to reduce the memory overhead. This representation improves on previous approaches, which were shown to "incur a super-polynomial size blowup in data complexity [69]." Meanwhile, the circuit representation, leads to a representation polynomial in the size of the input database. Being closely related to Boolean formulas, these circuit representations can also be used to compute semiring annotations, as in how-provenance, using the semiring $(K, \vee, \wedge, false, true)$. Thus, the Boolean circuit representation has a close correspondence with semiring provenance.

Debugging and Provenance. Provenance is a widely used technique that facilitates debugging of logic programs. For example, [73] proposes an approach that uses the computation trees produced as a side effect of top-down evaluation for debugging wrong or missing answers. For general logic programming, *algorithmic debugging* [74, 32, 29] provides a framework that asks users questions based on a *debugging tree*. These questions may take the form of "is node X correct/incorrect?" and the user's answers guide the system to further explore the relevant portions of the debugging tree. In these approaches, the debugging tree is a form of provenance, which corresponds closely with semiring provenance, where the semiring structure is the set of debugging trees.

Outside of logic programming, approaches such as [75, 76], among others, take a similar approach of using fragments of a computation trace to answer debugging queries. For example, the debugger may ask whether variables have the correct value at some point in time and explore the trace further based on answers to these questions. However, for imperative or object-oriented languages, the notions of a trace through a computation are quite different from that in a logic language due to the differences in the computational model.

Other Applications for Datalog Provenance. Debugging Datalog specifications is not the only use case for provenance, with user-guided approaches [77, 78, 79, 80] for program analysis also relying on tracking the origins of data. In [79, 77, 78], a user may tag certain static analysis alarms to increase or decrease their importance in the next analysis cycle. In [80], the analysis system automatically generates an appropriate abstraction by iteratively trying and refining failing abstractions. All these approaches rely on an annotation framework for Datalog: the user-guided systems require the user to add an annotation representing the importance of an alarm, and the abstraction refinement system requires the system to tag failing analyses with annotations.

3.2 Incremental Evaluation

As discussed in Section 2.4.2, incremental evaluation is becoming an important problem for incrementally updating computations, where programs should efficiently process the insertion or deletion of input tuples.

3.2.1 Datalog

The main body of work concerning incremental computation strategies for Datalog focuses on rewriting techniques. For these strategies, it is noted that adding new tuples to a computation is a trivial problem since the new tuples can be inserted with semi-naïve evaluation, then the evaluation can be run until fixpoint [47]. Therefore, the approaches presented here focus on the *removal* of tuples.

The seminal paper on these rewriting strategies is [47], which presents an algorithm to update a materialized view for a database. The algorithm, *DRed*, works by deleting a superset of tuples that need to be deleted, then rederiving those that can still be derived from the remaining tuples.

Formally, the set-up is for standard Datalog. Given a Datalog program P and an input instance E, the result of evaluating P given E is $P(E)$. The problem of incremental computation can be expressed as follows. Given a set E^-, denoting a set of facts to be deleted from E and E^+, a set of facts to be added to E, how do we efficiently compute $P((E \setminus E^-) \cup E^+)$?

The strategy of [47] is to produce an auxiliary Datalog program which generates D, a superset of all the tuples deleted by the change to E. This auxiliary program is reminiscent of semi-naïve evaluation, and is produced as follows. Given a rule

$$R :- S_1, \ldots, S_n$$

the algorithm generates a set of rules

$$\begin{aligned}
\delta^- R &:- \delta^- S_1, \ldots, S_n \\
&\vdots \\
\delta^- R &:- S_1, \ldots, S_{i-1}, \delta^- S_i, S_{i+1}, \ldots, S_n \\
&\vdots \\
\delta^- R &:- S_1, \ldots, \delta^- S_n
\end{aligned}$$

Each $\delta^- R$ relation will contain a superset of tuples to be deleted from R. The union $D = \cup_{\text{relations } R \in P} \delta^- R$ denotes the full set of tuples that may be deleted. Since this is an over-approximation of the actual change, the tuples in D which are not actually deleted must then be rederived in a subsequent step. The rederivation is a simple application of semi-naïve evaluation.

Since [47], there have been a number of further optimizations to this approach. In particular, [81] introduces the backward/forward algorithm (B/F), which aims to reduce the over-approximation of the deletion set D by employing ideas from data provenance. For a tuple in D, if it can be proved from tuples in $E \setminus E^-$, then it should not get deleted. Therefore, the main contribution of this work is to employ backward and forward chaining to find proofs for tuples in $E \setminus E^-$. The approach is to evaluate a rule *backward*. Given a tuple t, and the rule producing t, we have an instantiation for the head of the rule. Then, the algorithm attempts to find matching tuples for the body of the rule in the remaining instance, and if these can be found then t is no longer considered for deletion.

The authors demonstrate that the B/F algorithm performs better than DRed in most cases in that it will consider less candidate tuples for deletion. However, it struggles when proofs for facts are difficult to produce, in which case DRed may perform better.

Further work on the Datalog rewriting strategies includes integrating an approach called *counting* [82], in combination with either DRed or B/F. The main idea is to add a counter to each tuple, which tracks the number of possible derivations for that tuple. By doing this, the check for provenance is reduced to a simple check of the counter - if the counter is 0 then the tuple can be deleted; otherwise, it cannot.

Difficulties arise, however, with recursive rules. In general, a tuple may have infinitely many derivations with a recursive program, so care must be taken to maintain correctness and termination. Attempts such as [82] use counting algorithms for non-recursive rules but fall back on DRed or B/F algorithms when counters cannot be accurate due to recursion.

Alternatively, the annotation could record the provenance in the form of a Boolean expression. This approach, presented in [18], annotates each tuple with a Boolean expression, which evaluates to *true* if the tuple should be kept or *false* if it should be deleted. This expression contains the expressions of the dependent tuples as sub-expressions, thus encoding some notion of provenance. However, for large data, the encoding of this provenance would become prohibitively big, as the provenance of every derived tuple must be retained.

3.2.2 Differential Dataflow

Dataflow programming is a paradigm where computation is modeled as a graph describing the path of data through the system. Data is represented as *collections*, which are similar to relations in a traditional database setting. Edges represent the flow of data, and nodes represent a single operation performed on some collections. These operations are specified in a manner similar to relational algebra, for example, joining two collections or filtering one collection. Through this computation model, dataflow programming has become popular for complex parallel computations involving large-scale data.

Incremental evaluation is also crucial in dataflow programming, where updates to input data

may occur during computation. An important work in this area, *Differential Dataflow*, was proposed by Frank McSherry et al. [50, 83, 84]. The approach here is to create differential semantics for each dataflow programming operator, allowing the operator to correctly update the output result given some changes to the input.

While similar approaches exist for databases, dataflow programming also supports recursion (or iteration), which requires a more sophisticated treatment of differential operator semantics. The solution proposed in [50] is that each collection is tagged with a version number, where the version numbers are taken from some partial order. For example, a collection A_{ij} could depend independently on indices i and j, where i may be the previous iteration and j may be the previous epoch (where an epoch is one cycle of adding updates and running the program). Then, differential versions of the operators process data by taking into account the version labels i and j to understand changes since $i-1$ or $j-1$ for example. These version tags are even generalizable, so versions may contain more dimensions than 2 or different semantics than iteration and epoch numbers which can be applied in different applications.

Using this framework, [50] develops differential semantics for common operations, including `select`, `join`, `concat`, etc. Furthermore, fixpoint operators, such as `extend`, `ingress`, `feedback`, and `egress`, also exhibit simple differential semantics. The formal mathematical background for these differential semantics is presented in [83], which provides a framework to derive differential semantics using abelian groups and Möbius transformations.

Furthermore, Datalog can be implemented on the Differential Dataflow framework. The work in [52] builds *Differential Datalog*, a Datalog engine that compiles a Datalog program into Differential Dataflow. The result is an engine that automatically incrementalizes a Datalog program, allowing for more efficient processing of insertions or deletions to the Datalog input, compared to naïvely re-executing the program with a new input.

Along with Differential Datalog, further work such as *Laddder* [85] have proposed extensions to the Differential Dataflow framework to enrich the Datalog semantics that can be expressed incrementally. In particular, Laddder proposes lattice aggregations which can be maintained incrementally. While extending the Differential Dataflow incremental semantics to support lattice aggregations, Laddder also imposes looser monotonicity rules on allowable programs, permitting programs that are *eventually* $\sqsubseteq$-monotonic as well as the standard $\sqsubseteq$-monotonic programs. These extensions enable incremental lattice-based data-flow analyses, which were previously challenging to implement without these semantics.

3.2.3 Databases

Similar problems of incremental view maintenance have been explored for databases in the 80s and 90s. This was important with the advent of materialized views, which are logical tables derived using queries from one or more base tables. Each time a base table was updated, the materialized view should be incrementally updated efficiently.

Thus, Blakeley et al. [86] explore techniques to facilitate these efficient incremental updates. Given a materialized view with its base tables and an update to the base tables, the aim is to compute the result of applying the update efficiently.

The first stage is to determine whether the update will affect the materialized view at all. The strategy is to construct a Boolean expression based on the updated data, and the satisfiability of this expression tells us whether the updates are relevant or not. For example, consider a view defined with a `select` operation over a Boolean expression C and the addition of a tuple t in an update. A unification procedure would replace matching variables in C with concrete values from t resulting in an expression C'. If C' can be satisfied, then the tuple t is relevant to the view. While in general, the satisfiability problem is NP-hard, the authors show that for the specific class of expressions produced, satisfiability can be determined efficiently.

The incremental update of the materialized view is performed through differential semantics for the underlying relational algebra statement. For example, a view might be defined by a join between two relations

$$v = R \bowtie S$$

Then, if R is updated by adding tuples from the set R^+, in other words $R' = R \cup R^+$, then the result of the update can be expressed as

$$\begin{aligned} v' &= R' \bowtie S \\ &= (R \cup R^+) \bowtie S \\ &= (R \bowtie S) \cup (R^+ \bowtie S) \end{aligned}$$

Therefore, to compute an updated view, we may only need to consider a subset of joins involving only the updated tuples. This technique can be easily generalized to multiple joins, and similar strategies are used to create differential semantics for other relational algebra operations.

Further optimizations to this technique include using common subexpressions to reduce repeated computations for updates, as presented in [87].

A formalization of the algebraic approach is presented in [88]. The concept of Δ relations is defined, where ΔR represents the changes in R due to an update. However, despite this new notation, the basic ideas for incremental analysis are similar to previous works, where incremental semantics are defined for each relational algebra operation.

An alternative viewpoint is to consider this problem from the language perspective. Rather than dealing with relational algebra, the authors of [31] build a language similar to SQL but focus on incremental view maintenance. Thus, the language supports specifying 'old' or 'new' versions of a relation, referring to the state before and after an update. Additionally, there is a method presented to automatically rewrite the program's rules into a differential form at compile time. The differential form allows efficient insertion and deletion of data.

These techniques, along with DRed and its related improvements, have been surveyed in [48]. The survey presents a classification taxonomy for the various strategies for incremental view maintenance.

3.3 Debugging and Repair

As highlighted in Section 3.1, provenance is an important component of debugging logic programs. In this sub-section, we discuss the wider picture of automated debugging techniques in

general, and repair techniques for logic programs.

3.3.1 Delta Debugging

Delta debugging [89, 90] is a general method for localizing faults introduced after changes are made to a system. The approach takes the set of changes and performs a divide-and-conquer algorithm to find a minimum subset of changes that reproduces the faults.

More formally, delta debugging follows the procedure shown in Algorithm 3. A failure-inducing string S (for example, a program input or a program itself) is divided into n partitions. Each partition and its complement are tested to check if the failure is reproduced. If it is, then the relevant partition or complement is used as the basis for the next iteration.

Algorithm 3 Delta-Debugging(S), reproduced from [91]

```
1: n ← 2
2: while true do
3:     Divide S into n partitions, Δ_1, ..., Δ_n
4:     Let ∇_1, ..., ∇_n be the complements of Δ_1, ..., Δ_n respectively
5:     Test each Δ_1, ..., Δ_n and ∇_1, ..., ∇_n
6:     if all pass then
7:         n ← 2n
8:         if n > |S| then
9:             return most recent failure-inducing substring
10:        end if
11:    else if Δ_i fails then
12:        n ← 2; S ← Δ_i
13:        if |S| = 1 then
14:            return S
15:        end if
16:    else if ∇_i fails then
17:        n ← n − 1; S ← ∇_i
18:    end if
19: end while
```

The simplicity of the delta debugging algorithm means that it applies to any language, as long as the errors are monotone (i.e., any superset of a failure-inducing input also induces a failure). Thus, delta debugging is a suitable approach for positive Datalog, which satisfies this monotonicity property.

Along with the basic delta debugging approach, several extensions have been proposed, such as Hierarchical Delta Debugging (HDD) [92] and Iterative Delta Debugging (IDD) [93]. HDD applies delta debugging by splitting only on boundaries of syntax tree elements, thus producing only syntactically valid input. While HDD is not practical for diagnosing syntax errors, it can be more efficient (needing an order of magnitude fewer test inputs according to [92]) for diagnosing runtime errors. HDD has also been an active area of research in recent years, with techniques

such as [94, 95, 96] proposing further extensions to improve the efficiency and expressive power of HDD. Meanwhile, IDD applies the delta debugging algorithm across multiple revisions instead of only a single change as the standard delta debugging algorithm does.

3.3.2 Logic Programming Repair

While delta debugging and its related approaches are designed to be generic and applicable to any language, the research area of repair in databases and logic programs is highly specialized to certain kinds of databases and queries. For example, the theoretical logic programming community has proposed *abductive reasoning* to interpret logic programs. The survey [97] provides an overview of this idea, showing how abduction can be used to interpret logic programs.

Abduction is a form of reasoning in contrast to deduction and induction. For example, consider the following example from [97]:

$$\texttt{grass-is-wet} \leftarrow \texttt{rained-last-night}$$
$$\texttt{grass-is-wet} \leftarrow \texttt{sprinkler-was-on}$$
$$\texttt{shoes-are-wet} \leftarrow \texttt{grass-is-wet}$$

Under abductive reasoning, if we know that `shoes-are-wet` is true, then `rained-last-night` is one possible explanation, and `sprinkler-was-on` is another possible explanation. One of the common characteristics of abductive reasoning is that multiple explanations might exist, and selecting the best of these is an important problem.

In logic programming, abduction is related to *provenance*, both being a way of reasoning backward through the logic rules [98]. Therefore, abduction can be used as a tool to facilitate program debugging and repair in the presence of violated integrity constraints [99, 100].

The database community also proposes several solutions for repairing database instances that violate some integrity constraints. The first comes from the community of *consistent query answering* [101, 102], which proposes to repair the output of a query rather than the database instance itself. In these works, the repaired query output is computed as an intersection across all repaired database instances, and thus, the result may change depending on the query. Another approach is to repair the database instance itself, using techniques such as abduction [103].

In summary, techniques such as provenance and abduction are important in the community of database repair. However, this field is largely studied at a theoretical level, with no practical implementations being widely adopted to the best of our knowledge.

3.3.3 Synthesis

In a similar vein to logic programming repair, which deals with repairing input databases, there is also a body of work in Datalog *synthesis*, which can be seen as an approach to repairing Datalog rules. The general framework for synthesis is where an input database and a desired output database are known, and the corresponding Datalog program should be synthesized through automated or semi-automated methods.

One major body of work has been in *syntax-guided synthesis* for Datalog programs [104, 105, 6]. In this framework, the user provides input-output examples, along with a syntactic component to guide the synthesis, usually in the form of a set of candidate rules or meta-rules from which to generate candidate rules. Then, the synthesis problem is to select a subset of candidate rules that correctly compute the given outputs from the corresponding inputs. The approach in [104] uses refinement and active learning techniques on Datalog syntax to traverse the space of possible programs. Meanwhile, [105] uses numerical methods, where each rule is associated with a weight between 0 and 1. Then, optimization methods such as Newton's root-finding algorithm are employed to find a subset of rules that minimizes the difference between output tuples and their expected outcomes. Finally, [6] uses a combination of provenance, delta debugging, and SAT solving techniques to perform a counter-example guided search to find a correctly synthesized program.

Another body of work has been in *inductive logic programming* (ILP) [106, 107, 108], which introduces inductive reasoning as a framework to describe the problem of inducing hypotheses given observations. Inductive logic programming algorithms usually consist of two parts: a hypothesis search and a hypothesis selection. Modern synthesis systems, such as [109] based on ILP have been introduced; however, their practicality is limited for large instances with recursive structure.

Chapter 4

Large-Scale Provenance in Datalog

This chapter outlines our approach for efficiently computing provenance information in Datalog and showcases how provenance enables easier debugging for Datalog programs. We use provenance in the form of *proof trees*, and we present a novel encoding that enables a bottom-up Datalog evaluation to produce proof trees with small runtime overheads.

This work was published in TOPLAS 2020, in the paper "Debugging Large-Scale Datalog: A Scalable Provenance Evaluation Strategy" [4].

This chapter is organized as follows: Section 4.2 motivates our provenance method and describes its use in a real-world program analysis use case. In Section 4.3, we detail the theoretical basis of our approach regarding the provenance evaluation strategy and the provenance queries which construct proof trees for tuples. We also demonstrate the minimality properties and present the practical solution that results from this theory. In Section 4.4, we detail the implementation of our system in Soufflé. Finally, in Section 4.5, we present experiments that show the feasibility of our provenance system, and we summarize in Section 4.6.

4.1 The Datalog Debugging Problem

As discussed in Chapters 2 and 3, Datalog has become a widely adopted logic programming language for various analysis tasks. These real-world applications of Datalog are often large, containing hundreds or thousands of potentially mutually recursive rules, with tens of millions of input and output tuples. Therefore, as with any complex piece of software, there is the potential of introducing bugs and faults into the Datalog program. In particular, a fault in a Datalog program typically manifests in one of two ways: (1) an unexpected output tuple appears, or (2) an expected output tuple does not appear. These bugs may appear due to a fault in the input data or a fault in the logic rules. Both scenarios call for mechanisms to explain how an output tuple is derived or why the tuple cannot be derived from the input tuples. One way to approach debugging for Datalog is through *proof trees*. In the case of explaining the existence of an unexpected tuple, a proof tree describes formally the input tuples and the sequence of rule applications involved in generating the tuple. On the other hand, a failed proof tree, where at least one part of the proof tree doesn't hold, may explain why an expected tuple cannot be derived in the logic program. These proof trees can be seen as a form of *provenance*, an

explanation vehicle for the origins of data [59, 55].

In the presence of complex Datalog programs and large datasets, Datalog debugging becomes an even bigger challenge due to the size and complexity of proof trees. Section 3.1 discusses several previous approaches to debugging Datalog programs, including provenance-based debugging, which stores a full trace of execution during the Datalog evaluation [71, 56, 57]. Using these techniques, Datalog users follow a *debugging cycle* which allows them to find anomalies in the input relations or the logic rules. In such setups, the typical debugging cycle comprises the phases of (1) defining an investigation query, (2) evaluating the logic program to produce provenance witness, (3) investigating the faults based on the provenance information, and (4) fixing the faults. For complex Datalog programs, the need for re-evaluation for each investigation is impractical. For example, DOOP [20] with a highly precise analysis setting may take multiple days to evaluate for medium to large-sized Java programs. Although these state-of-the-art approaches scale for smaller database querying use cases, such techniques are not practical for industrial-scale static analysis problems.

A further difficulty in developing debugging support for Datalog is providing understandable provenance witnesses. Use cases such as program analysis tend to produce proof trees of very large height. For example, investigations on medium-sized program analyses in DOOP have minimal height proof trees of over 200 nodes. Therefore, a careful balance must be struck between providing enough information and readability for the user. Our approach limits information overload and handles large proof trees by allowing users to explore relevant fragments of the proof trees interactively.

This chapter presents a novel debugging approach that targets Datalog programs with characteristics of those found in static program analysis. Our approach scales to large dataset and ruleset sizes and provides succinct and interactively navigable provenance information.

The first aspect of our technique is a novel provenance evaluation strategy that augments the intensional database (IDB) with *Proof Annotations* and hence allows fast proof tree exploration for *all* debugging queries *without the need for re-evaluation*. The exploration uses the proof annotations to construct proof trees for tuples lazily, i.e., a debugging query for a tuple produces the rule and the subproofs of the rule. The subproofs, when expanded in consecutive debugging queries, will produce a minimal height proof tree for the given tuple. Our system also supports non-existence explanations of a tuple. In this case, proof annotations are not helpful since they cannot describe non-existent tuples. Thus, we adopt an approach from [110] to provide a user-guided procedure for explaining the non-existence of tuples.

We implement the provenance evaluation strategy in the synthesis framework of Soufflé [35], using specialized data structures and an interactive debugging query system for each logic program. Our approach is tightly integrated into the Soufflé engine and achieves higher performance than existing provenance approaches when more than one provenance query is run. We demonstrate the feasibility of our technique through the complex Java points-to framework, DOOP, running the Java DaCapo benchmark suite, which produces tens of millions of output tuples. We demonstrate that the initial implementation of our novel provenance method incurs a runtime overhead of 1.31×, and memory consumption overhead of 1.76× on average.

Our contributions in this work are as follows:

- a provenance evaluation strategy for Datalog programs - defining a new evaluation domain based on a provenance lattice which extends the standard Datalog subset lattice with proof annotations, and leveraging parallel bottom-up evaluation to give minimal height proof trees,
- a provenance query system for constructing minimal height proof trees utilizing proof annotations, allowing effective bug investigation with a minimum number of user interactions,
- an efficient and scalable integration of the proof tree generator system into Soufflé, using specialized data structures for computing and storing proof annotations, and
- large-scale experiments using the Doop program analysis framework with DaCapo benchmarks with tens of millions of tuples, measuring on average 1.31× overheads for runtime and 1.76× overheads for memory.

4.2 Motivation and Problem Statement

Real-world Datalog programs for applications such as program analysis can often contain up to hundreds of mutually recursive rules. With such complex applications, bugs are a common occurrence during the development cycle of a Datalog program. Buggy Datalog code may manifest itself in two main ways: (1) it produces an unexpected output tuple, or (2) it fails to produce an expected output tuple.

A common approach to characterize the evaluation of a Datalog program is through *proof trees*. A proof tree for a tuple describes the derivation of that tuple from input tuples and rules. During the debugging cycle, the presence of any unexpected tuples can be explained by producing a valid proof tree, where all nodes of the proof tree hold. On the other hand, a failed proof tree, where at least one part of the proof tree fails to hold, provides valuable insight into why a tuple is not produced. Therefore, both valid and failed proof trees are critical for investigation into anomalies.

Note that there could potentially be an infinite number of valid proof trees to explain any given tuple. However, Datalog developers desire concise proof trees such that the faulty behavior of the logic program is revealed quickly. In this section, we describe how *minimal height* proof trees can be used to debug a Datalog program and an overview of our method for generating minimal height proof trees for output tuples.

4.2.1 Use Case: Program Analysis

Points-To Analysis

Recall, from Section 2.3, our running example of computing a pointer analysis in Datalog. In this example, a source program (Figure 2.2a) is encoded as a set of EDB input tuples (Figure 2.2b). Then, the Datalog program (Figure 2.2c) computes the output relations `vpt`, `alias`, and `safevar`.

Minimal Height Proof Trees. The example in Figure 2.2 computes the output relation `alias` that captures the alias relationship of any two variables. One such output tuple is `alias(userSession,ins)`, which a user may wish to investigate the existence of. In other words, the user may wish to show how the analysis derives `alias(userSession,ins)` from the input data via the rules. The proof tree in Figure 4.1 shows that `alias(userSession,ins)` is derived by rule r_4 using the facts `vpt(userSession,L3)` and `vpt(ins,L3)` where the constraints `userSession != ins` and `L3 != nullptr` are satisfied. This outcome is expected since it tells us that `userSession` and `ins` may point to the same object (`L3` in this case), and thus they may alias.

$$\dfrac{\dfrac{\texttt{assign(userSession,ins)} \quad \dfrac{\texttt{new(ins,L3)}}{\texttt{vpt(ins,L3)}}\, r_1}{\texttt{vpt(userSession,L3)}}\, r_2 \quad \dfrac{\texttt{new(ins,L3)}}{\texttt{vpt(ins,L3)}}\, r_1 \quad \begin{array}{l}\texttt{userSession} \neq \texttt{ins}\\ \texttt{L3} \neq \texttt{nullptr}\end{array}}{\texttt{alias(userSession,ins)}}\, r_4$$

Figure 4.1: Full proof tree for `alias(userSession,ins)`

While the above proof tree explains the existence of the query tuple, it is also important that computed proof trees are of minimal height. Figure 4.2 demonstrates what can happen if this minimality constraint is not enforced. Imagine that the line `ins = userSession;` was added to the source program, resulting in the EDB tuple `assign(ins,userSession)`. Then, this circular assignment means that the tuple `vpt(ins,L3)` can be derived in an arbitrary number of rule applications.

$$\dfrac{\texttt{assign(ins,userSession)} \quad \dfrac{\texttt{assign(userSession,ins)} \quad \dfrac{\cdots}{\texttt{vpt(ins,L3)}}\, r_2}{\texttt{vpt(userSession,L3)}}\, r_2}{\texttt{vpt(ins,L3)}}$$

Figure 4.2: Infinitely many derivations for `vpt(ins,L3)`, resulting from the circular assignment in line 11 and a newly inserted line `ins = userSession;` in the input program

Thus, even for this small example, there are infinitely many valid proof trees for the tuple `vpt(ins,L3)`. A provenance system ought to produce the most concise proof tree so that an end user can explore the tree to understand the derivation of a tuple in the least number of steps.

Proof Tree Fragments for Debugging. Suppose a Datalog user discovers an unexpected tuple in the output, which indicates that a fault exists somewhere in the logic program. The aim is to investigate the root cause of this fault. Since proof trees provide explanations for the existence of a tuple, the proof tree of an unexpected tuple will help identify the fault in the logic program.

An example fault could be if rule r_2 was altered as follows,

```
vpt(Var,Obj) :- assign(Var,Var2), vpt(Var3,Obj).
```

Note that the propagation of pointer information through an assignment no longer holds, and the `assign` and `vpt` in the body of the rule are no longer related to each other. Such a fault may have been introduced by a minor typo, and as a consequence, the analysis would produce the extra tuple `alias(userSession,admin)`. This additional tuple becomes a *symptom* of the fault, which can be diagnosed by using its proof tree to highlight the root cause of the fault.

However, in practice, a full proof tree may be too large to provide a meaningful explanation even if it is of minimal height, and as experiments in Section 4.5 show, proof trees for real-world program analyses (e.g., DOOP) can exceed heights of 200. Thus a Datalog user may want to explore only relevant *fragments* of it interactively. A fragment of a proof tree is a partial subtree, which consists of some number of levels. For instance, we may construct fragments comprising of 2 levels to explore only parts of the proof tree that are relevant.

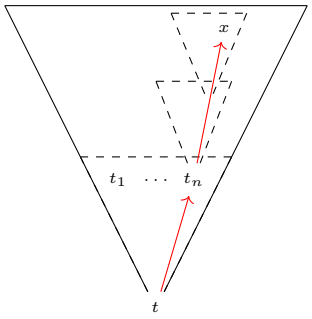


Figure 4.3: Interactive exploration of fragments of a proof tree for t

We illustrate the exploration of fragments of the proof tree in Figure 4.3. In the figure, tuple t denotes the symptom of the fault, i.e., t is an unexpected tuple in the output. The aim is to explore the proof tree for t to find the root cause for this fault. In our example, the user follows the scent of the fault by expanding proof tree fragments that show anomalies. This process produces a path of exploration in the proof tree. The path of exploration discovers the root cause of the fault efficiently, without constructing and displaying the full proof tree of an output tuple.

Concretely, we may wish to explain the tuple `alias(userSession,admin)` which is produced as a result of the fault. Figure 4.4 illustrates the exploration of an explanation for `alias(userSession,admin)` by generating proof tree fragments of 2 levels at a time. The user generates the first fragment, decides that `vpt(userSession,L1)` is the most relevant explanation for the fault, and continues down this path. As a result, the root cause (for example, the erroneous rule r_2) is discovered after two fragments. This interaction mechanism also justifies the choice to minimize the height of proof trees. By doing this, we minimize the number of user interactions (i.e., proof tree fragments) required to discover the root cause for an anomaly.

Debugging Non-Existent Tuples. On the other hand, the *non-existence* of a tuple may also indicate a fault in the Datalog program. Moreover, when the Datalog program includes negations, the non-existence of an expected tuple may lead to the existence of an unexpected tuple, and vice versa. In the running example, the input data may be missing the tuple

$$\dfrac{\texttt{assign(userSession,ins)} \qquad \texttt{vpt(admin,L1)}}{\texttt{vpt(userSession,L1)}}\, r_2$$

$$\dfrac{\texttt{vpt(userSession,L1)} \qquad \texttt{vpt(admin,L1)} \qquad \begin{array}{l}\texttt{userSession} \neq \texttt{admin} \\ \texttt{L1} \neq \texttt{nullptr}\end{array}}{\texttt{alias(userSession,admin)}}\, r_4$$

Figure 4.4: Exploring the proof of `alias(userSession,admin)` to find the erroneous rule r_2

`assign(userSession,ins)`. In this case, the tuple `vpt(userSession,L3)` would not be produced with the altered input data. However, a developer would expect that the assignment in line 11 would cause the tuple to exist, and thus may wish to examine the reason for the tuple's non-existence.

Logically, the non-existence of a tuple results from there being *no valid proof tree* for that tuple. Therefore, to 'explain' the non-existence of a tuple, we must show that every attempt to construct a proof tree eventually fails. However, automated techniques are not tractable since this is an infinite search space. Therefore, we adopt a semi-automated approach from [110], using algorithmic debugging ideas [29] to ask the user queries to aid the construction of a single failed proof tree. Such a failed proof tree may provide valuable insight into the non-existence of the tuple. Further details for debugging non-existent tuples are presented in Section 4.3.4.

4.2.2 Proof Trees and Problem Statement

Recall, from Section 2.2, the terminology and syntax of Datalog. We use this notation, with one additional notation to represent constraints and negations. With this additional notation, a Datalog rule is of the form $R(X) :- R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$. The term $\psi(X_1, \ldots, X_n)$ denotes a conjunction of constraints, including arithmetic constraints (such as less than), or negations.

Given a Datalog program P, an input instance EDB of P, and a tuple t produced by P, our debugging problem is to find a *proof tree* of minimal height for t. We define a proof tree as follows:

Definition 4.2.1 (Proof Tree)**.** *Let P be a Datalog program, and let EDB be an input instance. A* proof tree *τ_t for a tuple t computed by P is a labeled tree where (1) each vertex is labeled with a tuple, (2) each leaf is labeled with an input tuple in EDB, (3) the root is labeled with t, and (4) for a vertex labeled with t_0, there is a valid instantiation $t_0 :- t_1, \ldots, t_n$ of a rule r in P such that the direct children of t_0 are labeled with $t_1, \ldots, t_n$. Moreover, the vertex is associated with r.*

A proof tree for t can be viewed as an explanation for the existence of t, by showing how it is derived from other tuples using the rules in the Datalog program. To formalize the problem statement, we characterize proof trees of minimal height. Note that the set of proof trees for a Datalog program could be constructed inductively by the height of the trees. We denote τ_t to be a proof tree for tuple t and T^k to be the set of proof trees of height at most k. This construction

leads to a convenient description of what it means for a proof tree to be of minimal height.

Definition 4.2.2. *We define the set of all proof trees inductively. Let* $T^0 = \{\tau_t \mid t \in EDB\}$ *be the set of proof trees for tuples in the input instance. Then, define* T^k *in terms of* T^{k-1}*:* $T^k = \left\{\tau_t \mid t :- t_1, \ldots, t_n \text{ is a valid instantiation and } \forall t_i : \exists \tau_{t_i} \in T^{k-1}\right\}$ *Then,* $T = \bigcup_{i \geq 0} T^i$ *is the set of all proof trees produced by the program* P*.*

Note that each T^k consists of proof trees of height at most k since if $t :- t_1, \ldots, t_n$ is an instantiation of a rule, then the height of the proof tree for t is equal to the maximum height of the proof trees for $t_1, \ldots, t_n$ plus 1. By defining the set of proof trees inductively, a proof tree of minimal height for a given tuple t has height given by

$$\min \left\{ k \geq 0 \mid \exists \tau_t \in T^k \right\}.$$

Intuitively, this means that a proof tree for a tuple t is of minimal height if there does not exist another valid proof tree with a smaller height. We emphasize that a valid proof tree must exist since we have assumed that tuple is in the IDB of the Datalog program, and therefore its existence can be proved. Based on this inductive construction of proof trees, we reduce the problem of generating a fragment of a proof tree into the following incremental search problem.

Problem Statement. Let P be a Datalog program, and I be the instance computed by P. Given a tuple $t \in I$, find the tuples $t_1, \ldots, t_n$ such that $t :- t_1, \ldots, t_n$ is a valid instantiation of a rule in P leading to a minimal height proof tree.

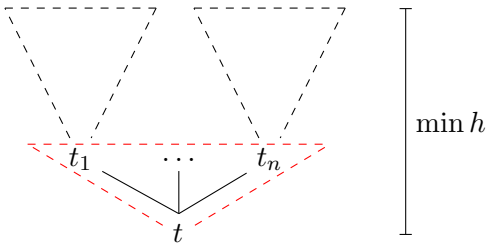


Figure 4.5: One level of a proof tree of minimal height for t

The problem statement is illustrated in Figure 4.5, where tuples $t_1, \ldots, t_n$ form the direct children of t in a minimal height proof tree. We can also denote $t_1, \ldots, t_n$ to be a *configuration* of the body of the corresponding rule. If this problem statement can be solved, such a solution can be applied recursively to construct the subtrees rooted at each t_i, which would then form valid proof trees for those tuples. Thus, this recursive construction solves the original problem of constructing a fragment of the proof tree of minimal height. Once a certain number of levels have been constructed, or if the only remaining leaves are in the EDB (characterized by having a proof tree consisting of only a single node), then the fragment is complete.

4.3 A New Provenance Method

A simple, partial solution to the problem might be to evaluate the Datalog program using a standard evaluation strategy, then generate a proof tree by brute-force searching for a matching

configuration for the body of a rule. However, this is an unfeasible approach for real-world problems where the resulting instance may contain millions of tuples, and there are also no guarantees that the proof trees produced in this manner are of minimal height. Alternatively, the Datalog evaluation strategy could be augmented to store minimal height proof trees as part of the computation; however, this is inefficient and would quickly run out of memory on even moderately sized instances.

Moreover, the two main evaluation strategies for Datalog, *bottom-up* and *top-down*, are unsuitable for solving this problem on their own. Bottom-up evaluation is an efficient method for generating tuples but does not store any information related to proof trees. On the other hand, top-down evaluation does compute proof trees as part of its execution, but there are no guarantees for minimality of height. Additionally, to prove the existence of a particular tuple requires proving the existence of every intermediate tuple up to the input tuples, and thus the problem of generating only fragments of proof trees cannot be solved by top-down. Therefore, we present a hybrid solution for generating proof trees, consisting of a provenance evaluation strategy based on bottom-up evaluation, plus a debugging query mechanism to construct proof trees.

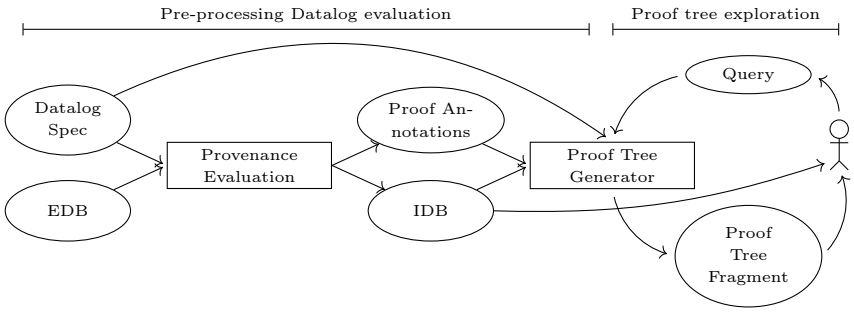


Figure 4.6: Synthesized Proof Tree Generator system

We summarize the system in Figure 4.6. The Datalog program and input tuples (EDB) are the input into the system. The provenance Datalog evaluation generates a set of tuples (IDB) alongside proof annotations for these tuples. For each tuple, the annotation stores two numbers: one referring to the rule generating that tuple and one referring to the height of a minimal height proof tree for that tuple. Using these annotated tuples as input, the interactive proof tree generator system allows the user to query for a proof tree fragment for any tuple in the IDB.

The proof tree generator is at the core of the interactive exploration of proofs for tuples. A user queries for a fragment of a proof tree, e.g., *two levels of a proof tree for* `vpt(ins,L3)`, and the system returns the corresponding result. This system can answer any number of queries, and the user can query for *any* fragment of the proof tree for *any* tuple. As previously mentioned, this allows the user to interactively explore the proof for a tuple and find a meaningful explanation for a tuple.

The provenance evaluation strategy resembles a pre-computation step for debugging. The evaluation is performed only *once*, but the IDB with proof annotations can subsequently answer *any* debugging query using the same IDB resulting from evaluation. The ability to answer

any debugging query without re-evaluation is an advantage over other selective provenance systems [56, 71], where the query is given prior to evaluation, which is then instrumented based on the query, and thus evaluation must be performed for each different query.

4.3.1 Standard Bottom-Up Evaluation

The basis of our approach is the standard bottom-up evaluation strategy for Datalog programs [12]. The computational domain of standard bottom-up evaluation is the subset lattice consisting of sets of tuples, denoted *instances* I. Recall, from Section 2.4.2, that the *naïve* algorithm for evaluation is based on the immediate consequence operator, Γ_P, which generates new tuples by applying rules in the Datalog program to tuples in the current instance.

$$\Gamma_P(I) = I \cup \{t \mid t :- t_1, \ldots, t_n \text{ is a valid instantiation of a rule in } P \text{ with each } t_i \in I\}$$

The result of Datalog evaluation is attained when Γ_P reaches a fixpoint, i.e., when $\Gamma_P(I) = I$. Note that this evaluation appears closely related to the inductive construction of proof trees, and indeed the set of tuples represented by T^i is equal to the set of tuples generated by the i-th application of Γ_P.

However, in practice, naïve evaluation is sub-optimal, and so the standard implementation is semi-naïve. With semi-naïve evaluation, a Datalog program is stratified, and each stratum is evaluated in order based on the dependency graph. Semi-naïve evaluation also uses a new knowledge optimization, which improves efficiency over naïve evaluation.

4.3.2 Provenance Evaluation Strategy

These standard bottom-up evaluation semantics are extended to compute a minimal height proof tree for each tuple – our extended semantics store *proof annotations* alongside the original tuples. In particular, for each tuple, the annotations are the *height* of the minimal height proof tree and a number denoting the *rule* which generated the tuple. By using this extra information, we can efficiently generate minimal height proof trees to answer provenance queries (see Section 4.3.3).

In the context of semi-naïve evaluation, particularly with stratified Datalog, we describe the provenance evaluation strategy here for a single stratum (i.e., a single fixpoint computation). The resulting correctness properties translate directly to the evaluation of the full Datalog program since correctness holds for every stratum in the evaluation.

The rule number annotation can be computed in a straightforward manner during bottom-up evaluation. With bottom-up evaluation, a new tuple t is computed if there is a rule r_k: $R(X) :- R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$ and a set of tuples $t_1, \ldots, t_n$ such that $t :- t_1, \ldots, t_n$ forms a valid instantiation of the above rule. If this is the case, then the rule firing of r_k generates t, and thus the identifier $r(t)$ takes on the value of k as the rule number annotation for t, thus tracking which rule is fired to generate that tuple.

However, the height annotations are more involved and relate closely to the semantics of bottom-up evaluation. Thus, we must develop a formalism for the height annotations to ensure that it correctly computes the height of the minimal height proof tree for each tuple. To formalize tuples with height annotations, we define a *provenance lattice* as our domain of computation,

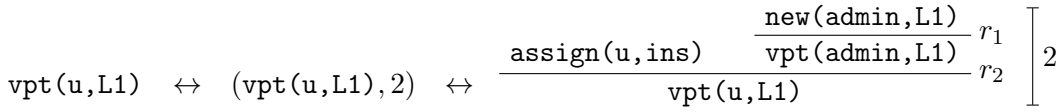


Figure 4.7: Connecting a tuple to a proof tree via a height annotation

which extends the standard subset lattice with proof annotations. An element of the provenance lattice is a provenance instance.

Definition 4.3.1 (Provenance Instance)**.** *A* provenance instance *is an instance of tuples I along with a function*

$$h : I \to \mathbb{N}$$

which provides a height annotation of each tuple in the instance. We denote a provenance instance to be the pair (I, h).

The aim of these height annotations is to connect a tuple to its proof tree, as depicted in Figure 4.7. The middle value is a tuple along with its height annotation, which is an example of an augmented tuple in a provenance instance. The corresponding proof tree on the right has height matching this annotation.

Similar to the subset lattice of standard bottom-up evaluation, the domain of provenance evaluation should also form a lattice, in our case, based on the subset lattice of standard bottom-up evaluation, but with elements being provenance instances rather than instances. We denote this to be the *provenance lattice* $\mathcal{L}$, where elements are provenance instances. The ordering $\sqsubseteq$ of elements in the lattice is defined by:

$$(I_1, h_1) \sqsubseteq (I_2, h_2) \iff I_1 \subseteq I_2 \text{ and } \forall t \in I_1 : h_1(t) \geq h_2(t)$$

Intuitively, this ordering specifies that an augmented instance (I_1, h_1) is 'less than' another augmented instance (I_2, h_2) if all tuples in I_1 also appear in I_2, with larger or equal height annotation. In $\mathcal{L}$, the bottom element is the empty instance, and a join between two instances (I_1, h_1) and (I_2, h_2) is the instance $(I_1 \cup I_2, h')$, where

$$h'(t) = \begin{cases} h_1(t) & \text{if } t \in I_1 \setminus I_2 \\ h_2(t) & \text{if } t \in I_2 \setminus I_1 \\ \min(h_1(t), h_2(t)) & \text{if } t \in I_1 \cap I_2 \end{cases}$$

Under this definition, moving 'up' the lattice towards the top element results in augmented instances with more tuples and smaller height annotations. This property guarantees the minimality of these height annotations since a bottom-up Datalog evaluation is equivalent to applying a monotone function to move 'up' a lattice.

The property that $\sqsubseteq$ is a valid partial order is essential to demonstrate that standard properties of Datalog evaluation hold.

Lemma 4.3.2. *The relation $\sqsubseteq$ is a partial order. Therefore, $\mathcal{L}$ is a lattice.*

In a similar fashion to the immediate consequence operator Γ_P operating on the subset lattice of Datalog instances, provenance evaluation is achieved with a consequence operator $\mathcal{T}_P$ operating on the provenance lattice. The result of evaluation is reached when $\mathcal{T}_P$ reaches a fixpoint, i.e., when $\mathcal{T}_P((I,h)) = (I,h)$. The main property $\mathcal{T}_P$ is that once a fixpoint has been reached, the proof tree height annotations are *minimal*, and they correspond to the heights of the smallest height proof trees.

The consequence operator $\mathcal{T}_P$ is defined in terms of the Γ_P operator:

Definition 4.3.3 (*Consequence* operator)**.** *The consequence operator, $\mathcal{T}_P$, generates a new provenance instance:*

$$\mathcal{T}_P((I,h)) = \left(\Gamma_P(I), h'\right)$$

where h' is defined as follows. For any tuple $t \in \Gamma_P(I)$, let

$$G_t = \{(t_1,\ldots,t_n) \mid t :\!-\, t_1,\ldots,t_n \text{ is a valid rule instantiation with each } t_i \in \Gamma_P(I)\}$$

be the set of all configurations of rule bodies generating t. Note this may be empty in the case of EDB tuples. Then,

$$h'(t) = \begin{cases} h(t) & \text{if } G_t = \emptyset \\ \min\left(h(t), \min_{g \in G_t}\left\{\max_{t_i \in g}\left\{h(t_i)\right\} + 1\right\}\right) & \text{otherwise} \end{cases}$$

The generation of new tuples behaves in the same way as Γ_P. To illustrate the height annotations, consider the rule instantiation $\texttt{vpt}(b,l_1) :\!-\, (\texttt{assign}(b,a),0),(\texttt{vpt}(a,l_1),1)$, with height annotations written alongside body tuples for convenience. From this rule instantiation, we would generate the tuple $\texttt{vpt}(b,l_1)$ with height annotation $\max(0,1)+1 = 2$. However, the instantiation $\texttt{vpt}(b,l_1) :\!-\, (\texttt{load}(b,c,f),0),(\texttt{store}(c,f,a),0),(\texttt{vpt}(a,l_1),1),(\texttt{alias}(c,c),2)$ would also generate $\texttt{vpt}(b,l_1)$, but with height annotation $\max(0,0,1,2)+1 = 3$. The resulting instance after applying $\mathcal{T}_P$ will contain only the smaller annotation, and thus the resulting provenance tuple is $(\texttt{vpt}(b,l_1),2)$.

Height Updates. Note that this semantics allow for the *update* of the height annotation for a tuple $t \in I$. If $\mathcal{T}_P(I,h) = (\Gamma_P(I), h')$ results in $h'(t) < h(t)$, then the height annotation of t is updated. An update may happen if $\mathcal{T}_P$ generates new tuples which form a valid configuration of a rule body generating t, with lower height annotations than a previous derivation.

We illustrate this definition of provenance evaluation strategy using the running example. As before, we denote (t,h) to be a tuple t with height annotation h. To highlight the importance of updating height annotations, we introduce a pre-processing step to generate the input instance for the points-to analysis. For example, a situation may arise in points-to analysis where a subclass constructer takes a superclass object as a parameter:

```
a2 = new O2(a);
a3 = new O3(a2);
```

In this situation, a store (e.g. `a.f = b;`) may also imply `a2.f = b;` and `a3.f = b;`. For the points-to analysis, a pre-processing step may be required to unroll the *store* and *assign* statements through the class hierarchy:

```
store(P,F,Q) :- instanceof(P,SuperP), store(SuperP,F,Q).
assign(Var1,Var2) :- instanceof(Var2,SuperVar2), assign(Var1,SuperVar2).
```

As a result of the recursive pre-processing step, the input instance to the points-to analysis fixpoint contains tuples with different height annotations. Figure 4.8 shows the derived `vpt` relation under the fixpoint computation with the provenance evaluation strategy. Importantly, note that the height annotation for `vpt(b,L1)` is updated in iteration 3. For the initial derivation in iteration 2, `vpt(b,L1)` is derived from `vpt(b,L1) :- assign(b,a), vpt(a,L1)`, giving a height annotation of 4. In iteration 3, however, `vpt(b,L1) :- assign(b,c), vpt(c,L1)` becomes a valid instantiation, therefore the height annotation of `vpt(b,L1)` is updated to be 3. Therefore, this example demonstrates that the height annotations may be updated after they are initially computed, which is essential to ensure minimality.

Input:

$$\{(\texttt{new(a,L1)},0),(\texttt{assign(b,a)},3),(\texttt{new(c,L3)},0),(\texttt{assign(b,c)},1),$$
$$(\texttt{assign(c,a)},1),(\texttt{load(b,c,f)},0),(\texttt{assign(a,b)},2)\}$$

Fixpoint iterations:

$i_0 : \emptyset$

$i_1 : \{(\texttt{vpt(a,L1)},1),(\texttt{vpt(c,L3)},1)\}$

$i_2 : \{(\texttt{vpt(a,L1)},1),(\texttt{vpt(c,L3)},1),(\texttt{vpt(b,L1)},4),(\texttt{vpt(b,L3)},2),(\texttt{vpt(c,L1)},2)\}$

$i_3 : \{(\texttt{vpt(a,L1)},1),(\texttt{vpt(c,L3)},1),(\texttt{vpt(b,L1)},3),(\texttt{vpt(b,L3)},2),(\texttt{vpt(c,L1)},2)\}$

$i_4 : \{(\texttt{vpt(a,L1)},1),(\texttt{vpt(c,L3)},1),(\texttt{vpt(b,L1)},3),(\texttt{vpt(b,L3)},2),(\texttt{vpt(c,L1)},2)\}$

Figure 4.8: IDB relation *vpt* in each iteration of the fixpoint computation for the example Datalog program

It remains to be shown that the provenance evaluation strategy is correct, i.e., that $\mathcal{T}_P$ terminates and results in the same set of tuples as Γ_P. Additionally, we must show that the height annotations resulting from the provenance evaluation strategy is minimal.

Lemma 4.3.4. *The tree consequence operator $\mathcal{T}_P$ computes the same tuples as Γ_P at fixpoint, i.e.*

1. *$\exists k$ s.t. $\mathcal{T}_P\left(\mathcal{T}_P^k\left((I,h)\right)\right) = \mathcal{T}_P^k\left(I,h\right)$, and*
2. *$\mathcal{T}_P^k\left(I,h\right) = \left(\Gamma_P^k\left(I\right),h^k\right)$ for some level annotation function h^k*

Proof. By definition, $\mathcal{T}_P$ generates tuples in the same fashion as Γ_P. Since Γ_P always reaches a fixpoint, say after l iterations, i.e., $\Gamma_P\left(\Gamma_P^l(I)\right) = \Gamma_P^l(I)$, we have

$$\mathcal{T}_P^l((I,h)) = \left(\Gamma_P^l(I), h^l\right)$$

Any further applications of $\mathcal{T}_P$ do not change the set of tuples since Γ_P has already reached a fixpoint. Thus, after l iterations, $\mathcal{T}_P$ computes the same tuples as Γ_P.

If there exists a $k \geq l$ such that $\mathcal{T}_P$ reaches fixpoint after k iterations, then the theorem is proved. Consider applying $\mathcal{T}_P$ to $\left(\Gamma_P^l(I), h^l\right)$. The set of tuples will not change. For any tuple $t \in \Gamma_P^l(I)$, the height annotation can only decrease as a result of applying $\mathcal{T}_P$ since $\mathcal{T}_P$ takes the minimum height over all rule configurations generating t and $h^l(t)$ also must result from such a configuration.

The height annotation is bounded from below by 0 since EDB tuples have non-negative annotations, and each subsequently generated tuple has increasing annotation. Therefore, applying $\mathcal{T}_P$ monotonically decreases the height annotation of t, which is bounded from below, so eventually, a fixpoint must be reached. Since this holds for all tuples in $\Gamma_P^l(I)$, $\mathcal{T}_P$ must reach a fixpoint after $k \geq l$ iterations. □

We have established that the provenance evaluation strategy terminates and computes the same set of tuples as standard bottom-up evaluation. It remains to be shown that the proof height annotations are minimal, i.e., that they reflect the real height of the minimal height proof tree for each tuple, and also that they correspond to real proof trees. The property of minimal height annotations is the major result of this section since it demonstrates that our method generates proof trees of minimal height.

Theorem 4.3.5. *Let $\mathcal{T}_P^k((I,h)) = \left(\Gamma_P^k(I), h^k\right)$ be the resulting instance at fixpoint of $\mathcal{T}_P$. Then, for any arbitrary tuple $t \in \Gamma_P^k(I)$,*

1. *there does not exist any sequence of tuples $t_1, \ldots, t_n$ such that $t :- t_1, \ldots, t_n$ is a valid instantiation of a rule in P with each $t_i \in \Gamma_P^k(I)$ and $h(t) > \max\{h(t_1), \ldots, h(t_n)\} + 1$, and*

2. *there is a valid proof tree for t with height $h^k(t)$*

Proof. The proof for part 1 is by contradiction. Assume such a sequence of tuples $t_1, \ldots, t_n$ exists. Consider applying $\mathcal{T}_P$ to the instance.

$$\mathcal{T}_P\left(\Gamma_P^k(I), h^k\right) = \left(\Gamma_P^k(I), h^{k+1}\right)$$

with $h^{k+1}(t) = \min_{g \in G_t}\left\{\max_{t_i \in g}\left\{h^k(t_i)\right\} + 1\right\}$ by definition of $\mathcal{T}_P$. The set of tuples does not change since we assume that a fixpoint of Γ_P has already been reached.

Since the sequence $t_1, \ldots, t_n$ is a valid rule body configuration generating t, it is an element of G_t, and therefore is considered when updating the height annotation of t. Since the height annotation resulting from this sequence is lower than $h^k(t)$, the update will happen, and thus a fixpoint has not yet been reached.

Thus, we have a contradiction, and so such a sequence producing a lower height annotation cannot exist.

The proof for part 2 is by induction on the height annotation of t. Let $h = h^k(t)$ for simplicity.

If $h = 0$, then t is in the EDB. In this case, the proof tree with a single node corresponding to t is a valid proof tree. Otherwise, for $h > 0$, assume the hypothesis is true for all tuples with height annotation less than h. By definition of $\mathcal{T}_P$, there exists a sequence $t :- t_1, \ldots, t_n$ such that

$$h = \max\left(h^k(t_1), \ldots, h^k(t_n)\right) + 1$$

By the assumption, there are valid proof trees for each t_i of height $h^k(t_i)$. We can generate a proof tree as follows:

$$\dfrac{\dfrac{\ldots}{t_1} \quad \ldots \quad \dfrac{\ldots}{t_n}}{t}$$

where each $\ldots$ represents the subtree forming a valid proof tree for each t_i. This resulting proof tree has height

$$\max\left(h^k(t_1), \ldots, h^k(t_n)\right) + 1$$

which equals h. This forms a valid proof tree for t of height $h^k(t)$. □

We have shown the correctness and minimal height annotations of the provenance evaluation strategy for a single fixpoint computation. To evaluate a stratified Datalog program in a semi-naïve fashion, each stratum is evaluated as a separate fixpoint using the provenance evaluation strategy. The correctness of the evaluation of a full Datalog program follows from the correctness of each fixpoint evaluation.

Constraints and Negation. While the above sections describe the provenance evaluation strategy for tuples, Datalog programs can also contain constraints and negations. However, constraints and negations need no special treatment and can be evaluated as they are in standard Datalog. The truth value of a constraint depends only on the values in the constraint and does not need to be explained further. For example, a constraint $1 \neq 2$ is always true and does not need to be explained in a proof tree. Similarly, under stratified Datalog, a negation can also be asserted to be true. While a negation might be explained by enumerating the relevant relation to assert that the negated tuple does not exist, this is impractical in practice. Therefore, a negation such as `!vpt(ins,L4)` is be asserted to be true with no further explanation in the proof tree.

Complexity of Provenance Evaluation Strategy. In this section, we discuss the complexity of the provenance evaluation strategy. We characterize this complexity by the number of rule firings during evaluation. With standard bottom-up evaluation, we say that a rule is fired if it generates a new tuple. Therefore, for each tuple generated by the Datalog program, there is exactly 1 rule firing. However, with the provenance evaluation strategy, a rule is also fired if it

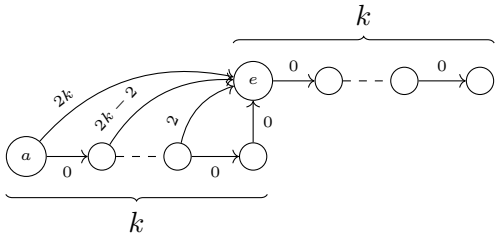
results in an update for the height annotation of a tuple. Therefore, we consider the number of updates performed during evaluation of the program as a characterization of the extra amount of work done by provenance evaluation compared to standard bottom-up evaluation.

Theorem 4.3.6. *An upper bound for the number of updates performed is* $\mathcal{O}(n \times \max h)$*, where* $\max h$ *denotes the maximum attained height annotation for any tuple during evaluation and* n *the number of tuples generated by the program.*

Proof. To prove this, we need to show two things: (1) that it is a true upper bound, and (2) that it is a tight bound.

To prove (1), consider a tuple t attaining a height annotation of $\max h$. Its annotation may only be updated if there is a valid derivation for t with a lower height. In the worst case, in each update, we reduce the annotation by 1, and thus we must perform $\max h$ updates to t. Considering all tuples produced by the program, we may update all tuples in this way in the worst case, and therefore, we have $\mathcal{O}(n \times \max h)$ updates.

To prove (2), we show an example attaining the upper bound, in Figure 4.9. In this example, the maximum height annotation is $2k$, and the tuple `reach`(a, e) will be updated k times as new derivations are computed using nodes in the bottom chain. Furthermore, each tuple `reach`(a, x) corresponding to nodes x in the 'leg' must be updated $\mathcal{O}(k)$ times as the tuple `reach`(a, e) is updated. Since there are k nodes in the leg, each of which is updated $\mathcal{O}(k)$ times, we have in total $\mathcal{O}(k^2)$ updates, which coincides with the upper bound. Therefore, this upper bound is tight. □




```
reach(a,a).
reach(a,N) :- reach (a,M), edge(M,N).
```

Figure 4.9: Example Datalog program demonstrating the upper bound is tight. The label on each edge (x, y) denotes the height annotation h (`edge(x,y)`). Although `edge` is an input relation for this stratum, the height annotations may be non-zero as a result of some pre-processing stage (see Figure 4.8 for an example of how this may occur).

We also note that $\max h$ cannot exceed n since in each iteration of $\mathcal{T}_P$ where a new tuple is generated, the height annotation of that tuple cannot exceed the maximum height annotation in the previous iteration, plus 1. If no more tuples are generated, then the height annotation for any tuple may not increase. Therefore, by generating a new tuple, we increase $\max h$ by at most 1, and therefore this value is at most the total number of tuples generated.

Therefore, in the worst case, the provenance evaluation strategy may have to do a quadratic amount of extra work compared to standard bottom-up evaluation. However, as real-world examples (see Section 4.5) show, such instances rarely occur, and scalability is maintained in most real-world applications.

4.3.3 Proof Tree Construction by Provenance Queries

Given a provenance instance (I, h) computed by the provenance evaluation strategy and a tuple $t \in I$, the aim is to construct one level of a minimal height proof tree for t. We utilize the height annotations h and rule number annotations that are stored alongside the instance during bottom-up evaluation. We use a top-down approach for proof tree construction, starting from a query tuple and recursively finding tuples that form a valid instantiation of a rule generating the query tuple. Denote $h(t)$ to be the height annotation and $r(t)$ to be the rule corresponding with the rule annotation for t.

The result of this search would be a sequence $t_1, \ldots, t_n$ such that $t :\!- t_1, \ldots, t_n$ is a valid instantiation of $r(t)$ leading to a minimal height proof tree. A pre-requisite is that the provenance instance (I, h) is the result of bottom-up evaluation, and since all possible tuples are computed during this evaluation, we know that each $t_1, \ldots, t_n$ exists in I. Thus, this problem would be solved by searching for tuples in the already computed instance I.

However, we must constrain this search such that the result is part of a proof tree of minimal height since there may be multiple valid configurations for the body of $r(t)$, and some configurations may not lead to minimal height proof trees. These constraints result from the annotations from bottom-up evaluation. From Theorem 4.3.5, there exists a configuration for the body that leads to a minimal height annotation for the head, and the height annotation for tuple t is generated as

$$h(t) = \max(h(t_1), \ldots, h(t_n)) + 1$$

by the consequence operator. Therefore, a configuration leading to the minimal height proof tree is $t_1, \ldots, t_n$ where $h(t_i) < h(t)$ for each t_i. Note that there may be multiple configurations leading to a proof tree of minimal height, and any of these configurations is a valid result for the problem.

The problem can be phrased as the following goal search. Given a tuple t and a rule $r(t) : R(X) :\!- R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$ generating t, we want to find tuples $t_1, \ldots, t_n \in I$ such that $t :\!- t_1, \ldots, t_n$ is a valid instantiation of $r(t)$, with proof annotations of each t_i satisfying the former constraints.

$$\begin{aligned} ?\!- \; & R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n), \text{matches}(t, X_1, \ldots, X_n), \\ & h(R_1(X_1)) < h(t), \ldots, h(R_n(X_n)) < h(t) \end{aligned}$$

The condition $\text{matches}(t, X_1, \ldots, X_n)$ denotes that for a result $t_1, \ldots, t_n$, the variable mapping from each X_i to t_i is consistent with the variable mapping from X to t. This is related to the unification problem in Prolog, and in our context is crucial to ensure that the resulting configuration forms a valid instantiation of r.

Example. We illustrate this construction using the running example. The query is for the tuple `alias(userSession,ins)`. From the initial bottom-up evaluation, the height annotation is $h(\texttt{alias(userSession,ins)}) = 3$, and the generating rule is r_4:

$$\texttt{alias(Var1,Var2)} :- \texttt{vpt(Var1,Obj)}, \texttt{vpt(Var2,Obj)}, \texttt{Var1} \neq \texttt{Var2}.$$

The proof tree construction searches for tuples forming a configuration for the body of r_4: `vpt(Var1,Obj), vpt(Var2,Obj)` satisfying the constraints:

$$\begin{aligned} \texttt{Var1} &= \texttt{userSession}, \\ \texttt{Var2} &= \texttt{ins}, \\ h(\texttt{vpt(Var1,Obj)}) &< 3, \\ h(\texttt{vpt(Var2,Obj)}) &< 3 \end{aligned}$$

In this example, the first two constraints correspond with matches$(t, X_1, \ldots, X_n)$, and the last two constraints enforce the conditions for proof height annotations. Therefore, the goal search is

$$\begin{aligned} ?- \ & \texttt{vpt(Var1, Obj)}, \texttt{vpt(Var2, Obj)}, \texttt{Var1} \neq \texttt{Var2}, \texttt{Obj} \neq \texttt{nullptr}, \\ & \texttt{Var1} = \texttt{a}, \texttt{Var2} = \texttt{b}, h(\texttt{vpt(Var1, Obj)}) < 3, h(\texttt{vpt(Var2, Obj)}) < 3 \end{aligned}$$

Solving this goal clause, we find the tuples `vpt(userSession,L3)` and `vpt(ins,L3)`, which form the next level of the proof tree:

$$\dfrac{\texttt{vpt(userSession,L3)} \quad \texttt{vpt(ins,L3)} \quad \begin{array}{l}\texttt{userSession} \neq \texttt{ins} \\ \texttt{L3} \neq \texttt{nullptr}\end{array}}{\texttt{alias(userSession,ins)}}\, r_4$$

The other constraints and negations in the rule, denoted $\psi(X_1, \ldots, X_n)$ in the goal search, are handled by finding the variable instantiation for $X_1, \ldots, X_n$ and displaying the instantiated constraint/negation as a node in the proof tree. No further proof search is required, as constraints with constants are trivially shown to be true, and negation is proved by asserting that the tuple does not appear in the IDB.

To illustrate how negations and constraints are handled, consider the recursive program in Figure 4.10a finding all pairs of nodes in a graph with distance at least 2:

```
path2(X,Z) :- edg(X,Y), edg(Y,Z), !edg(X,Z), X != Z.
path2(X,Z) :- edg(X,Y), path2(Y,Z), !edg(X,Z), X != Z.
```

(a) Example program with negation

```
edg(a,b).
edg(b,c).
edg(c,d).
```

(b) EDB Tuples

The output contains the tuple `path2`(a, d), and its proof tree would be:

$$\dfrac{\texttt{edg(a,b)} \quad \dfrac{\texttt{edg(b,c)} \quad \texttt{edg(c,d)} \quad \texttt{!edg(b,d)} \quad \texttt{b} \neq \texttt{d}}{\texttt{path2(b,d)}}\, r_1 \quad \texttt{!edg(a,d)} \quad \texttt{a} \neq \texttt{d}}{\texttt{path2(a,d)}}\, r_2$$

An important difference between our proof tree goal search and a standard conjunctive query evaluation is that our goal search terminates as soon as the first solution is found, which is sufficient for generating a minimal height proof tree. In contrast, a standard conjunctive query evaluation finds *all* possible configurations for the query.

The complexity of the goal search depends highly on the data structures used in the implementation. We assume fully (B-tree) indexed nested loop joins. Therefore searching for a tuple for a rule with an m nested join requires $\mathcal{O}(\log^m n)$ time. Given a proof tree height of k, we need $\mathcal{O}(k \log^m n) \equiv \mathcal{O}(\log^m n)$ to traverse a single branch.

4.3.4 Provenance for Non-Existence of Tuples via User Interaction

The provenance evaluation strategy of the previous section explains the existence of tuples in relations. However, the non-existence of tuples may also indicate faults in the input relations or in the rules.

Therefore, we extend our approach explaining why a tuple *cannot* be derived, i.e., if the user expects a tuple, but it does not appear in the IDB, the user may wish to investigate why the tuple is not produced. Alternatively, a user may want to understand why a negated body literal holds in a rule during the debugging process.

A non-existent tuple is characterized by *every* proof tree for the tuple failing to be constructed. The source of failure may be (1) tuples for the construction not being in the EDB/IDB, or (2) the constraints of rules not being satisfied. Given the potentially infinite number of failing proof trees, we avoid automatic procedures that represent a serious technical challenge and are not guaranteed to discover a failed proof tree containing the root cause of the fault. In practice, without a formal description of the root cause of the fault, the provenance system cannot decide which failed proof tree is most suitable[1].

Hence, in our system, we take a pragmatic, semi-automated approach that is inspired by existing work such as [71, 110]. Our system leverages user domain knowledge and allows user interactions to guide the construction of a single failing proof tree incrementally. Each user interaction produces a failing *subproof* or one level of the proof tree. This failing proof tree provides a succinct representation of valuable information for a Datalog user to discover why an expected tuple is not being produced by the program and does not burden the user with too much unnecessary information.

Formally, we define the problem as follows: given a provenance instance (I, h) computed by the provenance evaluation strategy, a tuple $t \notin I$ but expected to be in I, and a rule r: $R(X) :- R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$ with head relation matching t, we aim to find a configuration $t_1, \ldots, t_n$ for the body of r, such that either: (1) at least one $t_i \notin I$ or, (2) the constraints $\psi(X_1, \ldots, X_n)$ are not satisfied. Such a configuration forms a failing subproof, and recursively constructing subproofs results in a full failed proof tree. Note that it would be impossible to find a configuration where all tuples $t_i \in I$ and constraints $\psi(X_1, \ldots, X_n)$ hold since the prior assumption is that $t \notin I$. If such an instantiation cannot be found, then the tuple

[1] Proof annotations such as introduced in the previous section can only describe existent tuples in the IDB. It is impossible to consider such annotations for tuples that are *not* produced by the program.

t can be generated by the Datalog program, and thus $t \in I$.

For showing the non-existence of a tuple, the provenance system supports the Datalog user in constructing the failing proof tree in stages. The debugging query for non-existence has three user interaction steps that are repeated until the root cause of the fault is found. The user interaction steps are as follows:

1. the user defines a query for the non-existence of a tuple,
2. the user selects a candidate rule from which the tuple may have been derived,
3. the user selects candidate variable values of unbound variables in the rule.

The system displays the rule application in the failing proof tree indicating the portions of the rule that fail (i.e., at least one literal / constraint must fail) and the portions of the rule that hold.

The Datalog user can continue the query with the newly found failing literals guiding the system to find the root cause of the fault. This process is semi-automated since the nature of the fault is known by the Datalog user only.

Example: Consider the example from Figure 2.2 for which we want to query the non-existence of the tuple `vpt(sec,ins)`. In the first user interaction step, the Datalog user queries for an explanation for the non-existence of the tuple `vpt(sec,ins)`. Then, the Datalog user selects an appropriate rule such as rule r_2.

The system can then produce a partial instantiation for the body of the rule, where variables matching the head are replaced by concrete values from t such as,

$$\texttt{vpt(sec,ins)} \text{ :- } \texttt{assign(sec,ins)}, \texttt{vpt(ins,Obj)}$$

In the last user interaction step, the Datalog user selects instantiations for the remaining free variables in rule r_2. For example, the Datalog user may choose the value `L3` for the free variable `Obj`.

$$\texttt{vpt(sec,ins)} \text{ :- } \texttt{assign(sec,ins)}, \texttt{vpt(ins,L3)}$$

Given the instantiated rule, the provenance system will evaluate which portions of the sub-proof fail and which portions hold. With that information, the Datalog user can continue the exploration of the failing portions to find the root cause of the fault. A simple color labeling helps the user indicate which portions fail and hold, respectively.

$$\frac{\texttt{assign(sec,ins)}\ \mathsf{X} \qquad \texttt{vpt(ins,L3)}\ \checkmark}{\texttt{vpt(sec,ins)}}\ r_2$$

In the above example, the red color and X denotes the non-existence of the tuple `assign(sec,ins)` in the IDB, i.e., a failing portion of the proof tree. The blue color with ✓ indicates that `vpt(ins,L3)` holds.

In summary, our provenance system constructs a single failed subproof to explain the non-existence of a tuple. The construction of the failed subproof is guided by the Datalog user to ensure the answer contains a relevant explanation, given the infinitely many possible failed proof trees. The semi-automatic proof construction approach supports the Datalog user by highlighting which portions of the subproof hold and fail to guide the exploration.

4.3.5 Alternative Proof Tree Shapes

Our debugging strategy introduces an *interactive* system to explore fragments of proof trees to pinpoint faults in the Datalog program. Therefore, we wish to minimize the number of user interactions required to find the fault. For this aim, minimal height proof trees are critical for reducing the number of user interactions in the fault investigation phase. The utility of this approach is backed by several user experiences in industrial-scale applications (see cf. Section 7.1.2 [54]).

While generating proof trees of minimal height is useful for users, in principle, our framework is more general and can support a variety of metrics that may be beneficial in future applications. In this section, we outline general properties of proof tree metrics by having the following properties for function h:

1. The codomain of h must have a partial ordering $\sqsubseteq$ so that an update mechanism can be well defined. It is important that the annotation for a tuple can be updated if the same tuple is generated again with smaller (according to $\sqsubseteq$) annotation. This ensures that the resulting annotations are always minimal since tuples will continue being updated with smaller annotations until a fixpoint with annotations is reached.

2. The metric must be *compositional*, i.e., if t is generated by a rule instantiation $t :\!- t_1, \ldots, t_n$, then $h(t) = f(h(t_1), \ldots, (t_n))$. The importance of this property is two-fold. Firstly, it ensures that the values of the annotations can be computed during evaluation of the Datalog program by encoding f as a functor in the transformed Datalog program. For example, a rule may be transformed to be $R(X, f(h_1, \ldots, h_n)) :\!- R_1(X_1, h_1), \ldots, R_n(X_n, h_n)$. to compute the value of the annotation.

   Secondly, the compositional property is important for the reconstruction of the proof tree. In the backward search for a body configuration that may produce the head tuple, f may be encoded as a constraint. For example, a backward search may be

   $$\begin{aligned} ? :\!- & R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n), \text{matches}(t, X_1, \ldots, X_n), \\ & h(t) = f(h(R_1(X_1)), \ldots, h(R_n(X_n))) \end{aligned}$$

   where the last constraint ensures that the tuples found from the search correctly generate t with matching annotations.

3. The metric must be *monotone*, i.e., $h(t_i) \sqsubseteq h(t)$ for all $1 \leq i \leq n$, and *bounded*, i.e., there is a minimum value c such that $c \sqsubseteq h(t)$ for any tuple t. This property ensures that the provenance evaluation strategy terminates. Monotonicity ensures that with each rule

application, the annotation converges towards the minimum value c, and once it reaches c, then termination must occur.

If a given metric satisfies the above properties, then it can be used instead of proof tree height in our framework. Examples of such metrics could be the size of proof trees by the number of nodes or a sequence of k proof tree heights describing the smallest k proof trees for each tuple. One could also combine multiple metrics by a lexicographical ordering, for example producing proof trees of minimal height with a minimum number of nodes.

Given that our framework can be adapted to various proof tree shapes, the provenance system could be adapted for other applications that make use of other metrics. For example, given a program analysis written in Datalog, the origin of a bug alarm can be explained through its provenance. Such ideas, such as *thin slicing* [75], may also be able to use our provenance framework as a building block, and we leave this integration to future work.

4.4 Implementation in Soufflé

Recall, from Section 2.5.1, that the Soufflé Datalog engine is implemented as a synthesizer, which produces C++ code from a given Datalog program. During this process, an intermediate representation, RAM, represents the operational procedure of evaluating the Datalog program.

The main challenge of integrating the provenance evaluation strategy into Soufflé is to allow the synthesis to be aware of proof annotations. In particular, the semi-naïve evaluation machinery must be replaced by the provenance evaluation strategy as described in Section 4.3.2 to handle the proof annotations. Another critical part of this machinery is the synthesis of data structures [54, 8] for relations that are specialized for the operations in the program. The synthesized data structures have to be extended for proof annotations as well, enabling an update semantics in Datalog for the annotations.

For the provenance evaluation strategy, we need to amend relations with extra attributes to contain the proof annotations. We utilize the synthesis pipeline of Soufflé by introducing two provenance attributes for each relation. The first attribute represents the rule number of the rule which generated the tuple, and the second attribute represents the proof tree height. These two new attributes are introduced for each relation at the syntactic level in Soufflé. A predicate $R(X)$ is transformed into $R(X, @\text{rule}, @\text{height})$. For the sake of readability in this text, we distinguish between original and provenance tuples, where an original tuple is a provenance tuple without proof annotations. We rewrite all logic rules at the syntactic level to take account of the two provenance attributes constituting the proof annotation for our system and to compute the value of the annotations. The proof annotation instrumentation is performed as follows where a rule r_k:

$$R(X) \coloneq R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$$

is transformed into:

$$\begin{aligned} R(X, k, \max(@\text{height}_1, \ldots, @\text{height}_n) + 1) \coloneq \\ R_1(X_1, \_, @\text{height}_1), \ldots, R_n(X_n, \_, @\text{height}_n), \psi(X_1, \ldots, X_n) \end{aligned}$$

The transformed provenance rule computes level and height annotations for a new tuple, according to the semantics in Section 4.3.2. Since the rules are known statically, the rule number annotation k can be assigned a constant value for each rule in the transformed program. The rule numbers of the body predicates are ignored by using _ in each body predicate since they do not influence the head predicate.

The transformed provenance rule syntactically represents the computation of proof annotations during rule evaluation. However, the actual execution of provenance rules differs from a standard semi-naïve evaluation as presented in [12]. The reason is the update mechanism: a newly discovered provenance tuple may overwrite an existing provenance tuple if they are the same original tuple, but the new tuple has a smaller height annotation.

The provenance evaluation strategy extends the semi-naïve algorithm by updating the rule number and the height annotation of a tuple (as defined by $\mathcal{T}_P$) if the original tuple already exists and the newly generated tuple has a smaller height annotation. In other words, if a smaller proof tree could be found in a subsequent iteration for the same tuple, then an update occurs. Otherwise, if the original tuple does not already exist, the provenance tuple is inserted into the relation as is. Thus, the rule computing $\Delta_{R_0}^{i+1}$ in the semi-naïve evaluation is modified to accommodate the possibility of updates, i.e.,

$$\Delta_{R_0}^{i+1} = \left(new_{R_0}^{i+1} - R_0^i\right) \cup \left\{t \in R_0^i \mid h^i(t) > h^{i+1}(t)\right\}$$

where h^i denotes the height annotations in iteration i.

Therefore, with our provenance evaluation strategy, we integrate the possibility of an annotation update into the data structure. During an insertion operation, if the same original tuple is discovered, with a larger height annotation than the current tuple, then an update occurs. Therefore, we wish to call the insert operation if *either* the tuple does not already exist *or* the existing tuple has the larger proof annotation. A specialized existence check is implemented, implicitly implementing the semantics of the provenance Δ_R^{i+1} relation. Similar to the standard existence check, the special existence check is implemented as part of the data structure that has been specialized for each relation.

The result is the RAM snippet in Figure 4.11. The differences compared to the standard Datalog evaluation in Figure 2.8 are in lines 3-4, where the level annotation is computed within the loop nest as part of the insertion. Furthermore, the `PROV NOT IN` operation in line 3 denotes the special provenance existence check, which allows the `INSERT` to proceed if either the tuple does not already exist or exists with a larger proof height annotation. Thus, this implements the update semantics discussed above.

```
1  FOR a IN assign
2    FOR b IN delta_vpt ON INDEX b.0 = a.1
3      IF (a.0,b.1,_,(max(a.@height,b.@height)+number(1))) PROV NOT IN vpt
4        INSERT (a.0, b.1, number(2), (max(a.@height,b.@height)+number(1))) INTO new_vpt
```

Figure 4.11: Provenance version of RAM loop nest

However, the specializations in the data structures still remain to be discussed. Soufflé employs a highly specialized parallel B-tree data structure [8], with index orderings for the

attributes generated automatically via an optimization problem [54]. The B-tree employs a special optimistic read/write lock for each node, allowing high throughput for parallel insertion. During an `insert` operation, a thread may obtain a read lease for each node as it checks whether the tuple to be inserted already exists. If an insertion is required, it checks if the lease has changed and restarts the whole procedure if it has. Otherwise, it upgrades to a write lease and inserts the tuple into the correct position in the B-tree. The data structure also takes advantage of Soufflé's Datalog evaluation setting, where a single relation is either read from or written to, but never at the same time. Therefore, there are no interleaved reads and writes, so read operations are not synchronized.

With the proof annotations, we modify these specializations so that they can take into account the provenance semantics. The important step is the *update* semantics, and thus we integrate an update mechanism into the `insert` operation without requiring to delete and then re-insert. The provenance evaluation strategy requires two main modifications to our B-tree data structure. Firstly, the existence check for insertion should consider only the original tuple and ignore annotations. This ensures that Datalog set semantics are preserved and that no duplicate original tuples can exist. However, note that we still need to retrieve the full tuple, including its proof annotations. This is important for the proof tree construction, discussed in the next section. To address this concern, we use different lexicographical orderings of indices for the `insert` and `retrieve` operations. The `insert` index order does not include the attributes storing the proof annotations (so that annotations are not considered when checking existence), while the `retrieve` index order does. We also need to ensure that updating an annotation does not change the ordering of tuples according to the index; otherwise, subsequent index-supported searches will fail. Therefore, the `retrieve` index order requires the annotation attributes to be at the end, as this guarantees that an update to the annotations does not affect tuple ordering.

Secondly, we must have a mechanism to update existing tuples to implement the update semantics in Section 4.3.2. To achieve this, we modified the `insert` operation so it may also update any existing tuple with a smaller proof annotation. This insertion first requires to check if the original tuple exists in the B-tree. If it does, rather than aborting (as it would with standard Datalog evaluation), the insertion then checks the annotation. If the height annotation of the existing tuple is larger than the tuple to be inserted, then the annotations of the existing annotation are updated with new values. Note that during an update, a read lease also needs to be validated and upgraded to a write lease. The integration of the update into the `insert` operation avoids any need to delete and re-insert tuples, thus improving the efficiency of the provenance evaluation strategy. These modifications to the B-tree reflect the desired insertion semantics, and updates are handled directly in the `insert` operation. All other retrieval operations for the B-tree are not modified, and tuples can be retrieved as normal, including their proof annotations.

4.4.1 Implementing a Proof Tree Construction User Interface

After the provenance evaluation strategy is completed, the proof tree construction stage is driven by the user. It is critical that this process is also fully parallelized and highly performant.

The user interface is implemented as a command-line interface, where the user can enter queries to explain the existence and non-existence of a tuple. For example, the user can give the query `explain alias("userSession","ins")` which produces the proof tree in Figure 4.12. Explaining the non-existence of a tuple, i.e., the query `explainnegation vpt("userSession", "L4")`, results in the interaction in Figure 4.13. The user may also select the size of proof tree fragments to display, i.e., `setdepth 6` instructs the system to construct 6 levels of the proof tree in the next query. For each debugging query, the system invokes the relevant procedure to construct a proof tree fragment.

```
> explain alias("userSession", "ins")
                              new("ins", "L3")
                              -------------(R1)
assign("userSession", "ins") vpt("ins", "L3")   new("ins", "L3")
-------------------------------------------(R2) -------------(R1)
           vpt("userSession", "L3")             vpt("ins", "L3")  "userSession" != "ins"
                                                                  "L3" != "nullptr"
-------------------------------------------------------------------------------------(R1)
                             alias("userSession", "ins")
```

Figure 4.12: Explaining the tuple `alias(userSession,ins)`

```
> explainnegation vpt("userSession", "L4")
1: vpt(Var,Obj) :-
   new(Var,Obj).

2: vpt(Var,Obj) :-
   assign(Var,Var2),
   vpt(Var2,Obj).

3: vpt(Var,Obj) :-
   load(Var,Y,F),
   store(P,F,Q),
   vpt(Q,Obj),
   vpt(P,AliasObj),
   vpt(Y,AliasObj).

Pick a rule number: 2
Pick a value for Var2: "ins"

assign("userSession", "ins") ✓  vpt("ins", "L4") X
-------------------------------------------------(R2)
              vpt("userSession","L4")
```

Figure 4.13: Explaining the non-existence of the tuple `vpt(userSession,L4)`

The proof tree construction procedures must be highly performant since the constructed IDB may be very large, and we may need to search through many tuples to construct a proof tree fragment. Therefore, the proof tree construction procedures must be tightly integrated into the Soufflé system to enable a high-performance, parallel search. We integrate these procedures

into the Soufflé RAM, utilizing the existing translation from RAM to parallel C++. Moreover, since the provenance evaluation strategy uses specialized B-tree data structures, the proof tree construction phase can also utilize index-supported searches to find relevant tuples.

Recall that the proof tree construction is facilitated by searches for subproofs. Therefore, we require a specialized framework in the Soufflé RAM to implement a subproof search. We term this framework a *subroutine* framework. Each subproof search can be implemented as a subroutine, thus integrating with the Soufflé RAM.

To explain the existence of a tuple, a subproof search is required to search the body of a rule for matching body tuples, satisfying the constraint that the proof tree height is lower than the current tuple. This backward search for a single rule is implemented as a subroutine. For example, the rule r_2 : `vpt(Var, Obj)` :− `assign(Var, Var2), vpt(Var2, Obj)` is implemented as the subroutine in Figure 4.14.

```
1  SUBROUTINE vpt_2_subproof
2    FOR a IN assign WHERE a.0 = argument(0) AND a.@height < argument(2)
3      FOR b IN vpt ON INDEX b.0 = a.1 AND b.1=argument(1) WHERE b.@height < argument(2)
4        RETURN (a.0, a.1, a.@rule, a.@height, b.0, b.1, b.@rule, b.@height)
```

Figure 4.14: Subroutine for example program

Lines 2-5 represent a search through a database that is already constructed by the initial bottom-up evaluation, to find tuples which satisfy the constraints required for the construction of a proof tree fragment. The values of `argument(0)` and `argument(1)` are the values in the head tuple, and `argument(2)` is the height annotation of the head tuple. Therefore, this subproof search is parameterized by the head tuple. The relations of the body atoms `assign` and `vpt` are searched to find tuples `a` and `b` that match the rule's body. Importantly, the constraints for the level number are encoded in lines 2-3, ensuring that the resulting tuples have level number annotations lower than the query tuple. As shown in Section 4.3.3, applying this operation recursively allows us to generate the full proof tree.

Similarly, to generate a failed subproof to explain the non-existence of a tuple, the search for failing and holding parts of a subproof is implemented as a subroutine. Given an instantiated rule (which is produced via user interaction), a subroutine returns whether each body tuple is in the IDB and whether each constraint is satisfied.

4.5 Experiments

In this section, we conduct experiments with the provenance evaluation strategy and provenance queries implemented in Soufflé (see Section 4.4). The experiments are conducted for large-scale Datalog programs. We have the following experimental research claims:

> **Claim-I:** Our provenance evaluation strategy only has a minor impact on the runtime performance and remains scalable for realistic datasets and rulesets.
>
> **Claim-II:** The provenance queries for exploring proof tree fragments scale to large sizes, allowing efficient interactive exploration of proof trees.

Benchmark	context-insensitive		1-obj, 1-heap	
	# EDB	# IDB	# EDB	# IDB
antlr	8,319,095	21,832,232	8,319,095	24,145,648
bloat	4,468,277	13,104,020	4,468,277	15,417,516
chart	8,743,770	22,975,742	8,743,729	25,289,200
eclipse	4,389,770	13,076,265	4,389,799	15,389,708
fop	8,769,583	22,970,533	8,769,572	25,283,913
hsqldb	9,007,087	24,561,921	9,007,087	26,875,437
jython	5,203,400	17,158,375	5,203,400	19,471,797
luindex	4,396,394	13,415,336	4,396,394	15,728,788
lusearch	4,396,415	13,415,390	4,396,394	15,728,788
pmd	8,388,202	22,853,676	8,388,202	25,167,134
xalan	8,670,980	23,488,951	8,670,966	25,802,385

Table 4.1: Statistics for Doop benchmarks

Claim-III: Minimal height proof trees are very large for realistic benchmarks, substantiating the need for interactive proof tree exploration.

Our experiments were performed on a computer with an Intel Xeon Gold 6130 CPU and 192 GB of memory, running Fedora 27. Soufflé executables were generated using GCC 7.3.1.

4.5.1 Performance of the Provenance Evaluation Strategy

Doop. For the first set of experiments, we use the Doop [20] points-to analysis framework. We experiment with Doop's *context-insensitive* and *1-object-sensitive, 1-heap* (1-obj, 1-heap) analyses that exhibit different runtime complexities. As inputs for the points-to analyses, we compute the points-to sets for the DaCapo 2006 Java program benchmarks. Each analysis contains approx. 300 relations, 850 rules and produces up to approx. 26 million output tuples on the DaCapo benchmarks (See Table 4.1).

In Table 4.2, we present the runtime and memory consumption of Soufflé with 8 threads, comparing standard Soufflé with our provenance evaluation strategy with proof annotations. We use the DaCapo benchmarks with both the context-insensitive and 1-obj, 1-heap analysis. As expected, Soufflé with proof annotations incurs an overhead during evaluation. This overhead for the provenance evaluation strategy is typically within a factor of 1.3, which is a small overhead to pay for being able to generate minimal proof trees for all possible tuples in the IDB. Hence, we demonstrate the viability of the provenance evaluation strategy for large-scale Datalog programs, substantiating Claim I. We noticed that the runtime overhead for the context-insensitive analysis was smaller across all benchmarks than that of the 1-obj-1-heap analysis due to cache locality that was more prominent for smaller memory footprints. Note that the overhead for memory consumption is similar to performance overheads, at approximately 1.45×. This overhead results from the storage of extra proof annotations during evaluation.

In contrast, a naïve direct encoding approach (see Chapter 5, [111]), where each tuple is annotated with its full subproof (i.e., direct children in the proof tree), resulted in excessive memory usage (up to 100×) on a simple transitive closure experiment with 2000 tuples. Thus,

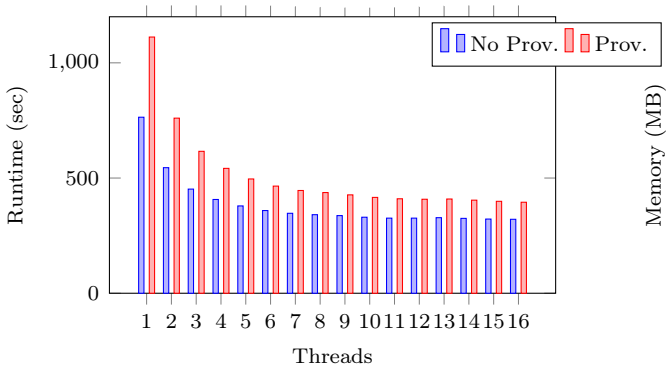
Benchmark	Runtime (sec)			Memory (MB)		
	No Prov.	Prov.	(×)	No Prov.	Prov.	(×)
context-insensitive						
antlr	9.73	12.29	1.26	595	900	1.51
bloat	9.54	12.25	1.28	596	900	1.51
chart	15.89	19.60	1.23	1,103	1,604	1.45
eclipse	9.64	11.76	1.22	593	898	1.51
fop	15.57	19.48	1.25	1,079	1,579	1.46
hsqldb	16.36	19.73	1.21	1,124	1,642	1.46
jython	11.00	13.62	1.24	731	1,090	1.49
luindex	9.62	12.00	1.25	594	905	1.52
lusearch	9.80	12.23	1.25	593	904	1.52
pmd	15.58	18.90	1.21	1,053	1,542	1.46
xalan	15.59	19.54	1.25	1,091	1,595	1.46
geo-mean			1.24			1.44
1-obj, 1-heap						
antlr	10.84	12.60	1.16	936	1,310	1.40
bloat	15.77	22.00	1.40	732	1,082	1.48
chart	21.84	28.13	1.29	1,242	1,788	1.44
eclipse	15.76	21.00	1.33	729	1,080	1.48
fop	22.21	29.63	1.33	1,216	1,756	1.44
hsqldb	23.01	29.43	1.28	1,256	1,823	1.45
jython	17.54	22.96	1.31	868	1,270	1.46
luindex	15.94	21.55	1.35	730	1,086	1.49
lusearch	15.95	21.25	1.33	731	1,087	1.49
pmd	21.58	28.21	1.31	1,190	1,725	1.45
xalan	22.09	28.68	1.30	1,224	1,773	1.45
geo-mean			1.31			1.46

Table 4.2: Runtime and memory usage overheads for Soufflé with and without proof annotations with 8 threads

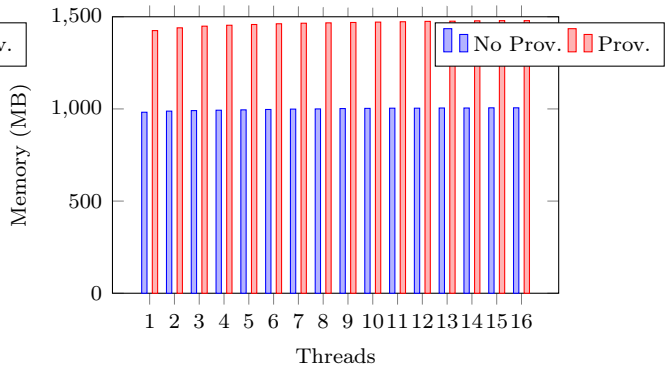
a direct encoding cannot be deployed for large-scale Datalog programs such as Doop.

Figures 4.15a and 4.15b show the total runtime and average memory usage for each of the Doop DaCapo benchmarks with both Doop (context-insensitive and 1-obj-1-heap) analyses, running with multiple threads. The figure demonstrates that the provenance evaluation strategy is scalable in that the overhead is sustainable with an increasing number of threads. We observe that the overall runtime decreases for provenance and without provenance until 5 threads and increases thereafter. This is caused by the synchronization of Soufflé's rule evaluation system and is not specific to provenance. Interestingly, the runtime overhead is larger with fewer threads, being 1.45× for 1 thread and 1.23× for 16 threads. Again, this is related to the underlying hardware architecture providing caches and memory lanes for each core. With more threads, the memory bandwidth to access the logical relations with proof annotations improves.

The memory usage of the provenance evaluation strategy has a consistent overhead of 1.45×, which aligns with our expectations that there would be a reasonable overhead associated with storing the provenance annotations per tuple. Note that this overhead is constant over any number of threads since the amount of extra information stored overall does not change with



(a) Total evaluation runtime of Soufflé on all DaCapo benchmarks with each Doop (context-insensitive and 1-obj, 1-heap) analysis with and without provenance



(b) Average evaluation memory usage of Soufflé on all DaCapo benchmarks with each Doop (context-insensitive and 1-obj, 1-heap) analysis with and without provenance

Benchmark	# EDB	# IDB
cactusADM	845,762	12,758,417
calculix	1,934,084	39,101,856
gamess	9,892,299	204,764,756
gcc	3,968,589	133,994,711
GemsFDTD	445,185	16,483,816
gobmk	4,211,765	34,935,275
gromacs	1,122,734	16,238,070
h264ref	638,837	12,991,980
omnetpp	720,581	18,261,432
perlbench	1,330,330	65,875,051
povray	1,160,773	55,857,237
tonto	4,767,218	285,090,829
wrf	4,690,140	97,986,011

Table 4.3: Statistics for Ddisasm benchmarks

the number of threads.

Ddisasm. For the second set of experiments, we use the Ddisasm [25] disassembler tool. The Ddisasm tool takes an executable binary as input and produces an assembly version of that binary as output. The main part of Ddisasm is a Soufflé program containing 535 relations and 1020 Datalog rules. Again, we run Ddisasm with and without provenance annotations, with 8 threads. As a benchmark suite, we use a subset of the SPEC CPU 2006 benchmarks, presenting only those with disassembly runtimes longer than 5 seconds. Each benchmark takes between 400 thousand and 9 million tuples as input, and produces between 12 and 285 million tuples as output (See Table 4.3).

The results in Table 4.4 demonstrate that the provenance evaluation strategy incurs a runtime and memory overhead. For Ddisasm, the runtime overhead is approximately 1.39× on average, which is an acceptable overhead for generating provenance annotations. However, the memory overhead for Ddisasm is 2.6×, which is considerably higher than for Doop. This is

	Runtime (sec)			Memory (MB)		
Benchmark	No Prov.	Prov.	(×)	No Prov.	Prov.	(×)
cactusADM	6.76	9.26	1.37	700	1,974	2.82
calculix	16.21	21.65	1.34	1,868	5,025	2.69
gamess	105.46	127.68	1.21	9,636	26,076	2.71
gcc	99.05	104.86	1.06	5,180	12,160	2.35
GemsFDTD	7.22	9.01	1.25	709	1,760	2.48
gobmk	19.39	35.56	1.83	1,450	3,453	2.38
gromacs	7.43	10.54	1.42	894	2,558	2.86
h264ref	5.54	7.85	1.42	644	1,679	2.61
omnetpp	10.4	13.17	1.27	763	1,952	2.56
perlbench	16.06	21.53	1.34	2,454	5,460	2.22
povray	11.57	15.71	1.36	2,071	4,532	2.19
tonto	340.49	749.63	2.20	24,395	82,152	3.37
wrf	58.28	74.2	1.27	4,739	13,085	2.76
geo-mean			1.39			2.60

Table 4.4: Runtime and memory usage overheads for DDISASM on SPEC benchmarks with and without provenance annotations with 8 threads

due to extra indices that were automatically generated to cover operations during the proof tree construction stage. In the worst case, a single relation, `instruction`, required 3 indices for standard Datalog evaluation, but 9 indices for the provenance evaluation strategy. This means that tuples in `instruction` are replicated 3× more with provenance, in addition to the overhead of the provenance annotations themselves. Future work optimizing the index generation algorithm in Soufflé will improve the memory overhead for situations where multiple indices are generated due to the provenance evaluation operations.

The main outlier in Table 4.4 is `tonto`, which exhibits a 2.2× runtime overhead and a 3.37× memory overhead. This particular benchmark generated 76% of its 285M tuples in the `string_part` relation, which had double the indices for the provenance evaluation strategy. As a result of this data replication, along with an increased cache miss rate (approximately double for the provenance evaluation strategy, resulting from worse cache coherence from storing provenance annotations), the runtime and memory overheads are larger than for other benchmarks.

While the runtime and memory overhead on DDISASM is higher than on DOOP, the result is still an acceptable price to pay to generate debugging annotations.

Comparison with Current Approaches. The current state of the art in tracking Datalog provenance is to instrument the program with a given provenance query. The instrumented Datalog program can then be evaluated using any Datalog engine. One example of this approach is the top-k approach [56, 112], where Datalog programs are instrumented based on a provenance query taking the form of a *derivation tree pattern*.

For our experiments, we implemented the instrumentation algorithm presented in [112], and evaluated the resulting Datalog using Soufflé, again using DOOP as the test Datalog program. Since the instrumentation requires a specific derivation tree pattern, we choose one that produces any proof tree for a single tuple from the `VarPointsTo` relation in DOOP. The tuple

Benchmark	Runtime (sec)			Memory (MB)		
	Top-k	Prov.	(×)	Top-k	Prov.	(×)
antlr	19.09	19.02	0.99	1,502	1,502	1.00
bloat	12.46	12.74	1.02	901	902	1.00
chart	19.69	20.12	1.02	1,606	1,605	1.00
eclipse	12.46	12.46	1.00	900	899	1.00
fop	19.73	19.87	1.01	1,579	1,582	1.00
hsqldb	20.47	20.44	1.00	1,647	1,646	1.00
jython	14.11	14.25	1.01	1,092	1,092	1.00
luindex	12.48	12.48	1.00	907	907	1.00
lusearch	12.53	12.47	1.00	905	905	1.00
pmd	19.40	19.42	1.00	1,542	1,543	1.00
xalan	19.63	19.73	1.00	1,596	1,595	1.00
geo-mean			1.01			1.00

Table 4.5: Runtime and memory usage overheads for our provenance approach compared to top-k [112], using Doop with the DaCapo benchmarks.

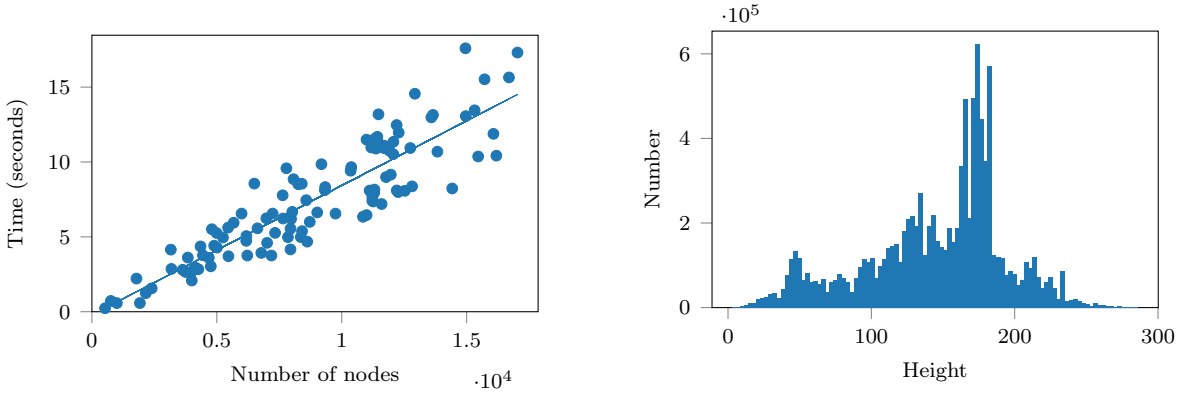
we choose describes a points-to relationship between two variables in `java.lang.Double` and `java.lang.Long`, which exists in the result for every DaCapo benchmark.

The results in Table 4.5 showed that during Datalog evaluation time, the difference in both runtime and memory usage is at most 2%. Note that the results differ from the previous section due to using an older version of Doop, which was better supported by our top-k implementation. Nevertheless, these results demonstrate that our provenance encoding scheme is at least as scalable as the state-of-the-art in terms of runtime performance. However, the main difference between the two approaches is that we can answer *any* provenance query during proof construction time, rather than only having the single proof tree matching the derivation tree pattern. Hence, our provenance evaluation strategy provides no runtime penalty for Datalog evaluation while having a considerable advantage when exploring the provenance.

4.5.2 Proof Tree Construction

For the construction of proof trees, the performance of the provenance queries is instrumental. A debugging query constitutes a backward search for a rule (i.e., reverting the computational direction of a rule). The construction of the proof tree is performed level by level. The expansion of a node in the proof tree represents a single debugging query.

In Figure 4.16a, we show the time taken to construct proof tree fragments with heights up to 20. We initiate the proof tree construction for randomly sampled output tuples in the Doop DaCapo benchmarks. In the figure, we plot the runtime against the number of nodes in the proof tree fragment. Even for 20 levels, these proof trees contain over 15,000 nodes. Considering that full proof trees may have heights over 200, the corresponding full proof trees would be intractable to compute and understand due to exponential growth. However, this experiment shows that the runtime for the construction of proof trees is approximately linear in the size of the tree. Therefore, provenance queries can be efficiently computed, and the method will scale well for interactive use. The interactive exploration of proof trees is scalable, with each



(a) Proof Tree Construction Time

(b) Proof Tree Heights

Figure 4.16: Proof Tree Construction and Statistics

debugging query on average taking less than 1 ms per node.

4.5.3 Characteristics of Proof Trees

In the following experiments, we demonstrate the difficulty of proof tree construction for Datalog programs at large scale. Figure 4.16b shows the distribution of heights of full proof trees for the DaCapo benchmarks. The proof tree heights can be more than 300. While this may not seem prohibitive, the expected number of nodes in the proof tree is exponential in height. A non-linear regression performed on the sizes of actual proof trees suggests that the branching factor of proof trees is approximately 1.466 for the DaCapo benchmarks. Therefore, since larger proof trees will have an exponential number of nodes, it is computationally intractable to construct full proof trees for these large programs. Besides, there is a usability challenge in generating meaningful explanations for the existence of a tuple, which is addressed by the interactive exploration of fragments of a proof tree, with the user exploring only relevant fragments. This is in contrast to a full proof tree, where a user may have to interpret millions of nodes to find an explanation.

4.6 Chapter Summary

This chapter presents a novel provenance encoding and evaluation strategy for Datalog, which allows a bottom-up Datalog computation to produce proof trees. The provenance encoding extends tuples in the IDB with proof annotations. Using these proof annotations, the proof construction phase can then incrementally construct minimal height proof trees based on user-defined provenance queries. Our approach has small overheads during the logic evaluation, exhibiting runtime overheads of $\sim$1.31$\times$ and memory overheads of $\sim$1.76$\times$.

Chapter 5

Elastic Incremental Evaluation for Datalog

Incremental evaluation is a technique that allows for updating the results of a Datalog program given some changes to its inputs. This chapter discusses the shortcomings of existing incremental evaluation strategies and details our novel approach, which we name *elastic* incremental evaluation. The main goal of elastic incremental evaluation is to provide reasonable performance with both small and large-sized incremental updates, which current state-of-the-art approaches can struggle with.

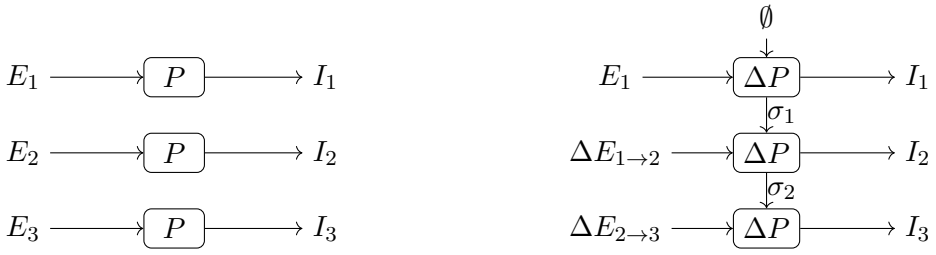
This work was published in PPDP 2021, in the paper "Towards Elastic Incrementalization for Datalog" [2].

This chapter is organized as follows. Section 5.1 introduces the incremental evaluation problem, and Section 5.2 gives background. Section 5.3 provides further background by detailing a *dense* encoding using our notation, which is algorithmically similar to existing incremental evaluation approaches. Then, Section 5.4 details our elastic incremental evaluation problem, showing the algorithms and their correctness. Finally, Section 5.5 discusses our implementation of incremental evaluation strategies in Soufflé, and Section 5.6 provides an experimental evaluation to validate our approach.

5.1 Incremental Evaluation

As described in Chapter 2, traditional bottom-up techniques in modern Datalog engines have become effective and performant for real-world applications, such as program analysis, networking, business applications, and others. This standard Datalog evaluation (which we refer to as *batch-mode* in this chapter, to distinguish it from *incremental*) computes the IDB (output tuples), given a set of rules and an EDB (input tuples. However, many real-world applications recompute most of their IDB with slight variations of the EDB [113, 50]. Hence, several state-of-the-art Datalog engines have proposed incremental evaluation techniques [114, 81, 50] to facilitate streaming, i.e., the evaluation reuses the IDB from the previous computation to compute the new IDB given some changes to the EDB.

State-of-the-art incremental evaluation approaches operate on several assumptions: (1) that the impact, i.e., number of overall tuple changes, is proportional to the update size, and (2) that the use cases exhibit a continuous stream of small impact updates. Indeed for several use cases [113], these assumptions tend to hold. However, for other notable use cases such as program analysis in a continuous integration/continuous delivery (CI/CD) setup [115, 116, 117], these assumptions do not hold. For example, static analyses written in Datalog can consist of hundreds or thousands of highly recursive rules and relations [20, 22]. Due to the complexity of the ruleset, one can no longer assume that update size is proportional to the impact size. Our experimental evaluation on the Doop program analysis framework found large variability in the *impact* of updates due to the connectivity of points-to analyses, where even small program changes may substantially change pointer sets of variables. Another concern is that static analyzers are deployed in CI/CD pipelines where state-of-the-art incremental evaluation gives no guarantee that updates will be structurally small. For instance, when the code base is updated, the initial change is often a refactor or new feature implementation. Such code changes typically result in large changes to the input of an analysis, deleting and inserting hundreds or thousands of input tuples. These changes may then be followed by smaller changes resulting from minor review suggestions, but as we show, even these smaller input changes do not necessarily result in small impacts. Thus, we argue the success of incremental evaluation techniques for such use cases requires minimizing the overhead of evaluating large impact updates.



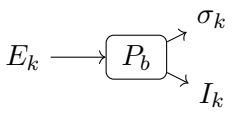
(a) Batch-mode (b) Incremental

Figure 5.1: Batch-mode vs. Incremental Evaluation

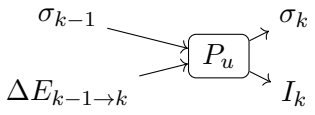
Consider Figure 5.1 that illustrates a generic incremental computation setup. Figure 5.1a shows a *batch-mode* evaluation for a Datalog program P and EDBs E_1, E_2, and E_3. The batch-mode evaluation runs the program P with each EDB separately to produce the IDBs I_1, I_2, and I_3. However, if only a small portion of the EDB and IDB changes between runs, many computations in the batch-mode evaluation may have been repeated. An incremental evaluation ΔP (illustrated in Figure 5.1b) can reuse computations from a previous run, called an *epoch*. A *Computational State* σ_1 encodes the previous computations for I_1 in a special format so that the next run can reuse the computations. With state σ_1 and the change in input $\Delta E_{1\to 2}$, the incremental evaluation produces the output I_2 and the new computational state σ_2. This process may be repeated, with a series of updates to the EDB being provided via $\Delta E_{i\to i+1}$. For the first epoch, we use the empty state as the computational state and E_1 as ΔE_1 to produce I_1 and the state σ_1. Any subsequent change $\Delta E_{i\to i+1}$ in the EDB is processed by using the previous computational state σ_i to generate I_{i+1}.

State-of-the-art incremental evaluation strategies [50, 49] represent their computational state exhaustively to perform small updates efficiently. Because of their exhaustive representation, however, the initiation of a stream and computing larger updates can both be prohibitively slow. For example, when a static program analysis seeks to reuse previous computations for a large code refactoring, significant portions of the control flow graph may have been replaced. In such a use case, an incremental evaluation will essentially perform two computations, one to delete the old control flow graph and one to compute the new control flow graph with additional overheads caused by incrementalization. Therefore, these heavyweight updates are better served by an evaluation strategy that is closer to standard batch-mode evaluation augmented with state for the future updates to be performed incrementally.

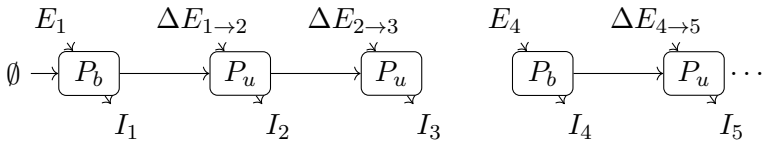
In this chapter, we demonstrate that both fully non-incremental and fully incremental strategies are not effective in some scenarios. Therefore, we propose an *elastic* incremental evaluation scheme called *Bootstrap-Update*, which is a hybrid approach. Figure 5.2 provides an overview. Our approach has two distinct strategies to evaluate an update: a specialized *Bootstrap* denoted as P_b (Figure 5.2a) and *Update* denoted as P_u (Figure 5.2b). The specialized Bootstrap resembles an augmented batch-mode evaluation that produces the computational state from scratch to allow subsequent updates, whereas Update is an incremental evaluation strategy. With the Bootstrap strategy, our approach can react to large incremental updates. For example, Figure 5.2c shows a scenario where the update from E_3 to E_4 is prohibitively large, and so the Bootstrap strategy is used to restart the incremental computation from scratch using E_4.



(a) Bootstrap Strategy



(b) Update Strategy



(c) Elastic Incremental, E_4 causes a restart in the stream using Bootstrap

Figure 5.2: Elastic Incremental Evaluation

Our approach proposes a novel *sparse* encoding that maintains a *lightweight* state σ. Our state exhibits a space complexity of $\mathcal{O}(|I|)$ (i.e. linear in the size of the output), whereas existing incremental encodings [50, 81] have a worst-case space complexity of $\mathcal{O}(m|I|)$ where m is the number of fixpoint iterations in the semi-naive evaluation algorithm [12]. Our lightweight state allows for an accelerated Bootstrap algorithm that can handle high-impact updates by efficiently recomputing the state from scratch, with the trade-off that the Update strategy may require more work for smaller updates. Furthermore, we provide a simple heuristic for choosing the appropriate strategy: we re-run the bootstrap when the incremental update takes more than a fraction (as a *switching parameter*) of the last bootstrap's runtime. This switching parameter

depends on the behavior of each application and the typical update characteristics for that application. Our solution operates under the insight that if we have comparable performance with batch-mode Datalog evaluation on large impact updates and a small slow down on low impact updates, we will have an overall net gain by selective application of incremental evaluation.

We have integrated our elastic Bootstrap-Update incremental evaluation in the open-source, high-performance Datalog engine Soufflé [23]. We have performed an extensive evaluation on a number of use cases that show our approach's utility compared to existing techniques on both large and small updates. We also provide a discussion of the practical considerations for building incremental evaluation in Soufflé that include relational data structures, parallelization, and scheduling strategies.

In summary, we make the following contributions in this chapter:

1. We present a new problem - that incremental evaluation should be *elastic*, i.e., it should be sensitive to the impact of an update.

2. We present a novel incremental evaluation using a sparse derivation counting encoding, exhibiting superior performance and lower memory overhead for elastic use cases.

3. We extend Soufflé, an open-source Datalog evaluation engine for elastic incremental evaluation, and propose several engine optimizations for superior performance.

4. We provide an extensive experimental evaluation validating the utility of our contribution.

5.2 Background

In this section, we use our running example to explain the background of standard and incremental Datalog evaluation. Recall, from Section 2.3, our running example, which encodes a pointer analysis in the form of Datalog rules. This is an ideal scenario where incremental evaluation may provide some benefit, where each change to the source program can be encoded as an incremental update to the Datalog input.

5.2.1 Semi-Naïve Evaluation

Section 2.4.2 describes the standard bottom-up semi-naïve evaluation algorithm. Here, we introduce an additional notation to describe the new knowledge optimization in a way consistent with the remainder of this chapter. We introduce an analog for the *immediate consequence operator*, which incorporates the new knowledge optimization of semi-naïve evaluation. Note that this notation is adapted from [49]. We call this new operator the *rule evaluation operator*, or Π:

$$\Pi_P[I \mid \Delta] = \left\{ t \;\middle|\; \begin{array}{l} t :- t_1, \ldots, t_n \text{ is an instantiated rule in } P \\ \text{where } \{t_1, \ldots, t_n\} \subseteq I \text{ and } \{t_1, \ldots, t_n\} \cap \Delta \neq \emptyset \end{array} \right\}$$

Under this definition, Π_P is analogous to the immediate consequence operator, where it computes head tuples resulting from instantiated rules where all body tuples are in I, but it only applies where at least one body tuple is also in Δ. For the rest of this chapter, the program

P is omitted from Π_P where it is clear. The dependence on Δ is the new knowledge optimization in semi-naïve evaluation.

Using the rule evaluation operator Π, the semi-naïve evaluation algorithm is presented in Algorithm 4 for a single stratum. The inputs for the algorithm are P, the set of Datalog rules forming the stratum, and E, the input set of tuples (since this is a single stratum, the input may be EDB tuples or tuples from earlier strata). This presentation of the algorithm differs from Algorithm 2 only in line 4, where we use the Π operator. By using the Π operator to require that at least one body tuple for each rule derivation is contained in Δ_{k-1}, the algorithm ensures that new tuples are only generated if at least one body tuple was new in the previous iteration and thus is functionally identical to Algorithm 2.

Algorithm 4 Semi-Naïve(P, E)

1: $\Delta_0 \leftarrow E$
2: **for all** $k \in \{1, 2, \ldots\}$ **do**
3: $\quad I_{k-1} \leftarrow \bigcup_{0 \leq i < k} \Delta_i$
4: $\quad \Delta_k \leftarrow \Pi_P[I_{k-1} \mid \Delta_{k-1}] \setminus I_{k-1}$
5: $\quad$ **if** $\Delta_k = \emptyset$ **then**
6: $\quad\quad$ **return** I_{k-1}
7: $\quad$ **end if**
8: **end for**

5.2.2 Incremental Datalog Evaluation

As discussed in Section 2.4.2, incremental evaluation is a procedure that *updates* the result of a Datalog computation, given some changes (inserted or deleted tuples) to the input, without performing a full recomputation. An incremental evaluation proceeds in *epochs*, where each epoch represents one round of inserting or deleting tuples from the input and computing the new result and incremental state. We refer to the insertions and deletions comprising the update as a *diff*. For the workflow in Figure 5.2c, each I_k represents the result of epoch k, and each $\Delta E_{k \to k+1}$ represents the diff consisting of insertions and deletions such that $\Delta E_{k \to k+1}$ applied to E_k results in E_{k+1}. The following definition formally defines the problem of incremental evaluation:

Definition 5.2.1 (Incremental Evaluation)**.** *Given a Datalog program P, an input data set E, the result $P(E)$, an insertion set E^+, and a deletion set E^-, compute the result $P((E \cup E^+) \setminus E^-)$.*

This chapter also discusses how the size of an incremental update impacts its performance. The following definition of *impact* forms a measure for the size of an incremental update, which coincides with a similar definition in [85]. Typically, higher impact changes result in greater computational overhead.

Definition 5.2.2 (Incremental Update Impact)**.** *The impact of an update is the number of IDB tuples changed as a consequence of the update, i.e., ΔI.*

Note that while the state-of-the-art incremental evaluation strategies, such as DRed [47], its related strategies [81, 82, 118], and counting-based algorithms [50, 49], have proven worthwhile for applications where each update has a small impact on the computed result, we have observed that this assumption does not hold in general for all incremental workloads.

For instance, consider our running example. We may remove line 11, `superuser = sec;` in Figure 2.2a, as part of an update to the software. This removed line would result in the EDB tuple `assign(superuser,sec)` being removed. However, the propagation of the pointer relationship from `sec` to `superuser` is already performed through the load/store pair `admin.session = sec;` and `superuser = admin.session;`. Therefore, no IDB tuples would be affected by the removal of line 11, and thus, this is a very low-impact update.

On the other hand, consider removing line 7, `admin.session = ins;`, which would cause the tuple `store(admin,session,ins)` to be removed. In this case, the pointer relationship of `ins` would no longer be propagated to `superuser`. As a result, `superuser` would no longer alias with `ins` and `userSession`. Therefore, multiple IDB tuples are affected by the removal of line 7, and thus, this is a higher impact update. This illustrates that even a single EDB tuple being removed as part of an incremental update results in different impacts. In these situations where both low- and high-impact updates may be present, current state-of-the-art incremental evaluation strategies may not be effective.

5.3 Current Incremental Evaluations

This section describes a current state-of-the-art incremental evaluation algorithm with some novel variations. The main idea of incremental evaluation algorithms, including this one, is to maintain an *incremental state* along with the normal set of output tuples. For the algorithm presented in this section, the incremental state is a pair of numbers per tuple, where the first number represents the fixpoint iteration in which the tuple is derived, and the second represents a count for the number of derivations for the tuple in that iteration. Importantly, the same tuple may be derived in multiple different iterations by different rule instantiations, and so this incremental state maintains all derivations in all iterations for each tuple up to fixpoint. We name this encoding the *dense* incremental evaluation due to this property of maintaining derivations in every iteration where a tuple is derived, in contrast to the elastic (or *sparse*) algorithm presented in Section 5.4.

Algorithmically, the encoding for dense incremental evaluation is very similar to current state-of-the-art incremental evaluation algorithms using a full counting approach [50, 49]. Therefore, the purpose of this section is to present the algorithm using our notation, along with some minor improvements and optimizations.

We introduce some notation for describing incremental evaluations. The main important feature of this notation is to describe a counting multiset of tuples per fixpoint iteration. To describe the notation formally, we first define a sequence of sets $\langle D_1, D_2, \ldots \rangle$ where set D_k denotes the set of *rule instantiations*. The set $D_k = \{(t :- t_1, \ldots, t_n)\}$ contains all the rule instantiations that are computed in iteration k. The derivation count of tuple t in iteration k is

the number of rule instantiations $(t :- t_1, \ldots, t_n)$ whose head is t.

While D_k is the explicit representation of rule evaluations that describes our incremental evaluation framework, it can be rather cumbersome and unnecessary for explaining the algorithms. Therefore, for the sake of simplicity, we define $\mathcal{N}^{\#}$ as a sequence of counting multisets for describing the derivation counts of tuples. We use the standard definition of multisets, where each $\mathcal{N}_k^{\#} = \{(t \mapsto c)\}$ denotes the number of rule instantiations $t :- t_1, \ldots, t_n$ for a tuple t in D_k. For notational convenience, we will express the elements with multiplicities $t \mapsto c$ as t^c, and we use N_k to denote the set projection of $\mathcal{N}_k^{\#}$.

5.3.1 Bootstrap Algorithm

Recall, from Figure 5.2, that our elastic incremental evaluation strategy contains two algorithms: a bootstrap and an update algorithm. Therefore, we also adapt this contribution to dense incremental by introducing a specialized *bootstrap* stage which accelerates the initiation of an incremental evaluation. The bootstrap algorithm is used when computing an epoch from scratch to produce a computational state for incremental evaluation, which can then be used in subsequent epochs by running the update algorithm (Section 5.3.2).

To illustrate the dense bootstrap algorithm, we use the running example. Here, iteration 0 contains the input tuples, the same as semi-naïve evaluation, with each tuple having a count of one.

$$\mathcal{N}_0^{\#} = \left\{ \begin{array}{l} \texttt{new(admin,L1)}^1,\ \texttt{new(sec,L2)}^1,\ \texttt{new(ins,L3)}^1,\ \texttt{new(userSession,nullptr)}^1, \\ \texttt{new(superuser,nullptr)}^1,\ \texttt{store(admin,session,ins)}^1, \\ \texttt{store(admin,session,sec)}^1,\ \texttt{load(superuser,admin,session)}^1, \\ \texttt{assign(userSession,ins)}^1,\ \texttt{assign(superuser,sec)}^1, \\ \texttt{assign(superuser,userSession)}^1 \end{array} \right\}$$

In iteration 1, we apply the non-recursive rule `r1`. In this case, all tuples have a count of 1, since this rule creates initial `vpt` relationships from the tuples in `new`:

$$\mathcal{N}_1^{\#} = \left\{ \begin{array}{l} \texttt{vpt(admin,L1)}^1,\ \texttt{vpt(sec,L2)}^1,\ \texttt{vpt(ins,L3)}^1, \\ \texttt{vpt(userSession,nullptr)}^1,\ \texttt{vpt(superuser,nullptr)}^1 \end{array} \right\}$$

In iteration 2, however, the counting semantics causes a divergence from the standard semi-naïve evaluation. Here, the tuple `vpt(superuser,L2)` is actually derivable in two different ways:

1. `vpt(superuser,L2) :- assign(superuser,sec), vpt(sec,L2).`, and
2. `vpt(superuser,L2) :- load(superuser,admin,session), store(admin,session,sec), vpt(sec,L2), vpt(admin,L1), vpt(admin,L1).`

Therefore, `vpt(superuser,L2)` has a count of 2 in iteration 2:

$$\mathcal{N}_2^{\#} = \left\{ \texttt{vpt(userSession,L3)}^1,\ \texttt{vpt(superuser,L2)}^2,\ \texttt{vpt(superuser,L3)}^1 \right\}$$

Now, in the third iteration, the tuple `vpt(superuser,L3)` can be derived alternatively as

```
vpt(superuser,L3) :- assign(superuser,userSession), vpt(userSession,L3).
```

In the elastic algorithm, this derivation would be excluded since the tuple already exists in $\mathcal{N}_2^\#$. However, in the dense algorithm, this new derivation is included, and thus

$$\mathcal{N}_3^\# = \{\texttt{vpt(superuser,L3)}^1\}$$

While iteration 3 includes the tuple `vpt(superuser,L3)`, this is not a *new* tuple since it already existed in iteration 2. Therefore, the set of tuples does not include any new tuples, and thus a fixpoint has been reached, and so the evaluation ends.

Algorithm 5 DenseBootstrap(P, E)

1: $\mathcal{N}_0^\# \leftarrow \{(t^1) \mid t \in E\}$
2: **for all** $k \in \{1, 2, \ldots\}$ **do**
3: $\quad N_{k-1} \leftarrow \text{Supp}(\mathcal{N}_{k-1}^\#) \setminus I_{k-2}$ ▷ When $k = 1$, $I_{-1} = \emptyset$
4: $\quad I_{k-1} \leftarrow \cup_{0 \le i \le k-1} N_i$
5: $\quad \mathcal{N}_k^\# \leftarrow \{(t^v) \in \Pi^\#[I_{k-1} \mid N_{k-1}]\}$
6: $\quad$ **if** $\text{Supp}(\mathcal{N}_k^\#) \subseteq I_{k-1}$ **then**
7: $\quad\quad$ **return** $(E, \mathcal{N}^\#)$
8: $\quad$ **end if**
9: **end for**

The dense bootstrap algorithm is presented in Algorithm 5. The overall structure is similar to semi-naïve evaluation but with the addition of operations to maintain the counting multisets of tuples. The main operator that we introduce in addition to semi-naïve evaluation is a counting version of the rule evaluation operator. To define this, we first introduce a version of the rule evaluation operator that computes sets of *rule instantiations* representing different derivations for a tuple:

$$\Pi_P^D[I \mid I_{\text{in}}] = \left\{ (t :- t_1, \ldots, t_n) \;\middle|\; \begin{array}{l} t :- t_1, \ldots, t_n \text{ in } P \text{ where } \{t_1, \ldots, t_n\} \subseteq I \\ \text{and } \{t_1, \ldots, t_n\} \cap I_{\text{in}} \neq \emptyset \end{array} \right\}$$

Using the above definition, we can define a counting version that computes the number of derivations for each tuple:

$$\Pi_P^\#[I \mid I_{\text{in}}] = \left\{ t^v \;\middle|\; v = \#\text{rule instantiations } (t :- t_1, \ldots, t_n) \in \Pi_P^D[I \mid I_{\text{in}}] \right\}$$

where v is the number of ways that the tuple t can be derived.

We also use a standard operator for the set projection of a multiset: $\text{Supp}(X) = \{t \mid (t^c) \in X \text{ and } c > 0\}$ collects all elements in a multiset with a count of at least 1.

The dense bootstrap algorithm begins by initializing iteration 0 to be equal to the input (line 1). Then, in the fixpoint loop, the algorithm first computes the equivalent of the delta in semi-naïve evaluation by taking the set projection of $\mathcal{N}_{k-1}^\#$ and excluding any tuples that were already previously computed in earlier iterations in I_{k-2} (line 3). The algorithm then computes

the current full set of tuples (line 4, in practice, the set I_k is maintained throughout, rather than computed in each iteration). The following step is the rule evaluation step, where the Datalog rules are evaluated using the counting rule evaluation operator with I_{k-1} and N_{k-1} (line 5). Note in this dense bootstrap algorithm, tuples computed in earlier iterations are *not* excluded, in contrast to the standard semi-naïve algorithm. This allows the dense algorithm to maintain the derivation counts for *all* iterations where the tuple is computed. The fixpoint break condition (line 6) holds if all tuples computed in the current iteration have already been computed before, and the algorithm returns the result once this fixpoint is reached. Note that the fixpoint is reached only if no *new* tuples are computed in the current iteration. Thus, despite the dense algorithm computing each tuple in potentially multiple iterations, the resulting fixpoint depends only on the set of tuples and is identical to the fixpoint result in the standard semi-naïve algorithm.

5.3.2 Incremental Update Algorithm

The dense update algorithm takes a computational state, computed either by the dense bootstrap algorithm or a previous dense update and applies a set of input changes. These input changes are represented as a set of input tuples, each marked as either an *insertion* or *deletion*. The algorithm returns a new computational state that reflects the result of applying the updates to the input set.

The dense update algorithm, presented in Algorithm 6, proceeds similarly to the full counting algorithm presented by Motik et al. [49]. To present the algorithm, we introduce a further extension to the rule evaluation operator, which computes tuples inserted or deleted as a result of an insertion or deletion in the body tuples of the rule instantiation. Recall, from Section 5.3.1 that $\Pi_P[I \mid I_{\text{in}}]$, denotes tuples computed by rules in P instantiated from I, with at least one body tuple also in I_{in}. This notation is extended with

$$\Pi^{\#}_P[I \mid I_1 \mid I_2] = \left\{ t^v \;\middle|\; \begin{array}{l} v = \text{number of rule instantiations } t \coloneq t_1, \ldots, t_n \\ \text{in } P \text{ where } \{t_1, \ldots, t_n\} \subseteq I \text{ and } \{t_1, \ldots, t_n\} \cap I_1 \neq \emptyset \\ \text{and } \{t_1, \ldots, t_n\} \cap I_2 \neq \emptyset \end{array} \right\}$$

This notation computes tuples from rule instantiations in I, where at least one body tuple is from I_1, and also at least one body tuple is from I_2. For the update algorithm, I_1 and I_2 would denote the deltas from semi-naïve evaluation and the diffs from the incremental update, respectively, allowing the rule evaluation to compute tuples that are newly changed in the current iteration of the current epoch. Additionally, the algorithm uses $\oplus$ and $\ominus$, the standard multiset addition and subtraction operators for operations involving multisets.

The update algorithm computes the updates to the sequence of multisets $\mathcal{N}^{\#}$, which result from applying the insertions and deletions to the input. The algorithm also makes use of a number of auxiliary sets: I^o_k and I_k maintain the full sets of tuples up to iteration k for the previous and current epoch, respectively, I^-_k and I^+_k maintain the tuples that are deleted and inserted respectively up to iteration k, and N^o_k and N_k are the set projections of $\mathcal{N}^{\#o}_k$ and $\mathcal{N}^{\#}_k$, storing the tuples that are new in iteration k in the previous and current epoch, respectively.

Algorithm 6 DenseIncrementalUpdate($P, (E, \mathcal{N}^{\#o}), (E^-, E^+)$)

Ensure: $E^- \subseteq E$, $E \cap E^+ = \emptyset$

1: $\mathcal{N}_0^{\#} \leftarrow E \setminus E^- \cup E^+$
2: $N_0 \leftarrow \text{Supp}(\mathcal{N}_0^{\#})$
3: $N_0^o \leftarrow \text{Supp}(\mathcal{N}_0^{\#o})$
4: $I_0^- \leftarrow E^-$
5: $I_0^+ \leftarrow E^+$
6: **for all** $k \in \{1, 2, \ldots\}$ **do**
7: $\quad I_{k-1} \leftarrow \cup_{0 \le i \le k-1} N_i$
8: $\quad I_{k-1}^o \leftarrow \cup_{0 \le 1 \le k-1} N_i^o$
9: $\quad \mathcal{N}_k^{\#} \leftarrow \mathcal{N}_k^{\#o} \ominus(\Pi^{\#}[I_{k-1}^o \mid N_{k-1}^o \mid I_{k-1}^-])$ ▷ Deletion term
$\ominus(\Pi^{\#}[(I_{k-1}^o \cap I_{k-1}) \setminus N_{k-1} \mid I_{k-2}^+ \cap N_{k-1}^o])$ ▷ Deletion update term
$\oplus(\Pi^{\#}[I_{k-1} \mid N_{k-1} \mid I_{k-1}^+])$ ▷ Insertion term
$\oplus(\Pi^{\#}[(I_{k-1} \cap I_{k-1}^o) \setminus N_{k-1}^o \mid I_{k-2}^- \cap N_{k-1}])$ ▷ Insertion update term
10: $\quad N_k \leftarrow \text{Supp}(\mathcal{N}_k^{\#}) \setminus I_{k-1}$
11: $\quad N_k^o \leftarrow \text{Supp}(\mathcal{N}_k^{\#o}) \setminus I_{k-1}^o$
12: $\quad I_k^- \leftarrow (I_{k-1}^- \setminus N_k) \cup (N_k^o \setminus I_k)$
13: $\quad I_k^+ \leftarrow (I_{k-1}^+ \setminus N_k^o) \cup (N_k \setminus I_k^o)$
14: $\quad$ **if** $N_k = \emptyset$ **then**
15: $\quad\quad$ **return** $(E \setminus E^- \cup E^+, \mathcal{N}^{\#})$
16: $\quad$ **end if**
17: **end for**

The algorithm is presented for a single stratum and takes the state of the previous epoch $(E, \mathcal{N}^{\#})$ and the incremental updates (E^-, E^+) to be applied to E. The initialization phase (lines 1 to 5) prepares the inputs by applying the diffs E^- and E^+, storing the result in the state for iteration 0, $\mathcal{N}_0^{\#}$.

In the fixpoint loop, the algorithm proceeds by computing the current full state of the database, both for the current epoch and the previous epoch (lines 7 and 8). As with the bootstrap algorithm, in practice, these sets are maintained throughout the running of the algorithm rather than computed in each iteration.

The rule evaluation in line 9 is split into four terms. The deletion term captures all tuples that are deleted in iteration k as a result of a body tuple in a rule derivation being deleted. Here, we consider only derivations where one tuple is in delta (N_{k-1}^o) and also one tuple is deleted (in I_{k-1}^-). The deletion update term captures the case where a derivation is updated so that it now occurs in an earlier iteration. For example, consider a derivation `t :- t1, t2`, where `t1` is derived in iteration 4 and `t2` is derived in iteration 2. In this case, `t` would be derived in iteration 5. However, in the current epoch, if an insertion leads to `t1` being derived in iteration 1, then `t` would now be derived in iteration 3. Hence, to ensure that only unique derivations are captured, the deletion update term would remove `t` from iteration 5. This deletion update term achieves this with $I_{k-2}^+ \cap N_{k-1}^o$, which states that a tuple in the delta in the previous epoch

(N^o_{k-1}), but inserted in an earlier iteration in the current epoch (I^+_{k-2}), is to be deleted since its iteration number is 'updated.' The insertion and insertion update terms act in the same way as the deletion and deletion update terms but for processing insertions.

After the rule evaluation is processed, lines 10 and 11 compute the deltas for the current and previous epoch, respectively, in preparation for the next iteration. Here, there is an explicit set minus to exclude tuples already existing before iteration k to ensure that the deltas contain only tuples that were new in the current iteration.

The algorithm continues by updating the I^-_k and I^+_k sets (lines 14 and 15). Computing I^-_k (line 14) takes the deletion set from the previous iteration I^-_{k-1} and excludes the tuples that are newly computed in the current iteration N_k, along with tuples that are deleted in the current iteration ($N^o_k \setminus I^n_k$). Similarly, computing I^+_k (line 15) takes the insertion set from the previous iteration and excludes tuples that already existed in the current iteration in the previous epoch (since these tuples already existed, so are not *newly* inserted in the current epoch), along with tuples that are inserted in the current iteration.

Since the dense incremental evaluation algorithm is adapted from existing algorithms, we refer to [49] for proofs of correctness.

5.4 Elastic Incremental Evaluation

This section describes our novel encoding and algorithms for elastic incremental evaluation. Recall, from Section 5.3, that the computational state in previous approaches [50, 49] involves a vector of numbers per tuple in the IDB. Each number in the vector represents a count in some fixpoint iteration. In the worst case, the length of the vector is determined by the number of iterations m in the fixpoint computation. Hence, the total state may exhibit a worst-case space complexity of $\mathcal{O}(m|I|)$ where $|I|$ is the size of the output.

In contrast, our elastic approach maintains a lightweight computational state consisting of two numbers per tuple. The first number is a *derivation count*, and the second number is the iteration in which the tuple is first derived. The derivation count represents the number of ways that the tuple can be derived in the iteration when it is *first* derived and allows the reuse of computation in the next epoch. Therefore, our encoding is a sparse version of the vector of numbers in previous approaches, keeping only the first iteration rather than the whole vector. As a result, the worst-case space complexity of this encoding is $\mathcal{O}(|I|)$.

With our lightweight computational state, we can switch between Bootstrap and Update to adapt to lightweight and heavyweight updates accordingly. When given an incremental update, we provide a heuristic for switching between both strategies. We first attempt the Update strategy. If it times out (the timeout is set to some fraction of the previous Bootstrap's runtime strategy, i.e., a *switching parameter*), we discard its partial state and produce the output and computational state from scratch using Bootstrap. The timeout is dependent on the application and needs to be fine-tuned appropriately. In contrast, previous approaches have a single strategy and cannot adapt to light and heavy updates.

In this section, we use the same notation introduced in Section 5.3, where $\mathcal{N}^\#_k$ is a multiset

containing tuples computed in iteration k, where the multiplicity of each tuple represents the number of its derivations in iteration k.

5.4.1 Bootstrap Algorithm

Our elastic incremental evaluation approach is structured similarly to the dense approach, with separate bootstrap and update algorithms. This section presents the Bootstrap algorithm, which is a specialized counting algorithm that efficiently computes the sequence of multisets from scratch, mimicking a semi-naive evaluation while also producing the incremental computation state.

For instance, consider our running example. In the initial phase, the input E becomes iteration 0, where the counting semantics mean that every tuple has a count of 1. Therefore,

$$\mathcal{N}_0^{\#} = \left\{ \begin{array}{l} \texttt{new(admin,L1)}^1,\ \texttt{new(sec,L2)}^1,\ \texttt{new(ins,L3)}^1,\ \texttt{new(userSession,nullptr)}^1, \\ \texttt{new(superuser,nullptr)}^1,\ \texttt{store(admin,session,ins)}^1, \\ \texttt{store(admin,session,sec)}^1,\ \texttt{load(superuser,admin,session)}^1, \\ \texttt{assign(userSession,ins)}^1,\ \texttt{assign(superuser,sec)}^1, \\ \texttt{assign(superuser,userSession)}^1 \end{array} \right\}$$

Iterations 1 and 2 are identical to the dense algorithm. Therefore,

$$\mathcal{N}_1^{\#} = \left\{ \begin{array}{l} \texttt{vpt(admin,L1)}^1,\ \texttt{vpt(sec,L2)}^1,\ \texttt{vpt(ins,L3)}^1, \\ \texttt{vpt(userSession,nullptr)}^1,\ \texttt{vpt(superuser,nullptr)}^1 \end{array} \right\}$$

$$\mathcal{N}_2^{\#} = \left\{\texttt{vpt(userSession,L3)}^1,\ \texttt{vpt(superuser,L2)}^2,\ \texttt{vpt(superuser,L3)}^1\right\}$$

In iteration 3, the elastic bootstrap diverges from the dense bootstrap algorithm. The tuple `vpt(superuser,L3)` would be derivable through the following rule instantiation:

```
vpt(superuser,L3) :- assign(superuser,userSession), vpt(userSession,L3).
```

However, this tuple already exists in $\mathcal{N}_2^{\#}$, and thus is not included in the elastic encoding. Therefore, $\mathcal{N}_3^{\#} = \mathcal{N}_2^{\#}$, and so a fixpoint has been reached and the Datalog evaluation ends.

Algorithm 7 presents the lightweight bootstrap algorithm for a single stratum. Its structure is almost identical to the dense bootstrap algorithm. The algorithm begins by initializing $\mathcal{N}_0^{\#}$ to be equal to E (line 1). In the fixpoint loop, the algorithm first creates a set projection of the current iteration's multiset (line 3). The algorithm also computes the full state of the relations up to iteration $k-1$ (line 4). These two auxiliary sets, N_{k-1} and I_{k-1}, are used in the rule evaluation on line 5. This rule evaluation computes all tuples that are new in the current iteration and excludes any tuples that were computed in earlier iterations. By excluding existing tuples, the algorithm maintains the sparsification property, exhibiting a space complexity of $\mathcal{O}(|I|)$. The algorithm exits and returns the evaluation state $(E, \mathcal{N}^{\#})$ (line 6) if no new tuples are generated in the current iteration, which is checked via the emptiness of the set projection of $\mathcal{N}_k^{\#}$.

In contrast to the dense bootstrap algorithm, the elastic algorithm excludes tuples in previous iterations during the rule evaluation step (line 5), which maintains the sparsification invariant. As a result, tuples in previous iterations no longer need to be excluded when computing the delta (line 3), as is done in the dense bootstrap algorithm.

Algorithm 7 Bootstrap(P, E)

1: $\mathcal{N}_0^{\#} \leftarrow \{(t^1) \mid t \in E\}$
2: **for all** $k \in \{1, 2, \ldots\}$ **do**
3: $\quad N_{k-1} \leftarrow \text{Supp}(\mathcal{N}_{k-1}^{\#})$
4: $\quad I_{k-1} \leftarrow \cup_{0 \leq i \leq k-1} N_i$
5: $\quad \mathcal{N}_k^{\#} \leftarrow \{(t^v) \in \Pi^{\#}[I_{k-1} \mid N_{k-1}] \mid t \notin I_{k-1}\}$
6: $\quad$ **if** $\text{Supp}(\mathcal{N}_k^{\#}) = \emptyset$ **then**
7: $\quad\quad$ **return** $(E, \mathcal{N}^{\#})$
8: $\quad$ **end if**
9: **end for**

Correctness. To demonstrate the correctness of Algorithm 7, we need to show that it computes the same resulting set of tuples as standard semi-naïve evaluation (Algorithm 4). To do this, we need to demonstrate two basic properties: *(a)* each N_k of Bootstrap is equal to Δ_k of semi-naïve, and *(b)* both Bootstrap and semi-naïve evaluation terminate after the same number of iterations.

To show this, we introduce the following lemma:

Lemma 5.4.1. *Given a Datalog program P, for all A, B such that $B \subseteq A$,* $\text{Supp}(\Pi_P^{\#}[A \mid B]) = \Pi_P[A \mid B]$.

This property can be shown since a tuple $t \in \Pi_P[A \mid B]$ if and only if there is a rule instantiation that computes it. If this is the case, then the same rule instantiation also fits $\Pi_P^{\#}[A \mid B]$ with a count of at least one. As a corollary, we can show that Bootstrap and semi-naïve both produce the same set of tuples in each iteration.

Lemma 5.4.2. *Given a Datalog program P and an input set E, each I_{k-1} of Bootstrap is equal to I_{k-1} of semi-naïve.*

The proof is by induction over k, since $\text{Supp}(\mathcal{N}_i^{\#}) = \Delta_i$ (from Lemma 5.4.1) for each iteration i, then each iteration's result must be identical to semi-naïve. Note that both Bootstrap and semi-naïve terminate after the same number of iterations, since $\text{Supp}(\mathcal{N}_i^{\#}) = \Delta_i$ for every iteration i, and therefore $\text{Supp}(\mathcal{N}_i^{\#}) = \emptyset$ if and only if $\Delta_i = \emptyset$. Therefore, both algorithms terminate after the same number of iterations and thus produce the same set of resulting tuples.

5.4.2 Incremental Update Algorithm

The Update algorithm is a procedure that takes a computational state, either computed by Bootstrap or by a previous Update, and a set of changes to the inputs. The algorithm returns the updated computational state after applying the input changes.

The Update algorithm produces a computational state $\mathcal{N}_k^{\#}$ from the computational state $\mathcal{N}_k^{o}$ of the previous epoch, following the iterations of the previous epoch's fixpoint. In each iteration, the algorithm applies the insertions and deletions resulting from the given changes to the input.

For instance, consider the running example, where we remove line 11, `superuser = sec;`, and insert a new line, `userSession = admin.session;`, as part of an update to the source code. As a result, the EDB tuple `assign(superuser,sec)` will be removed, and a new tuple `load(userSession,admin,session)` will be inserted as part of an incremental update. This incremental update can be expressed as the following diff:

$$\Delta E = \{\texttt{assign(superuser,sec)}^{-1},\ \texttt{load(userSession,admin,session)}^{+1}\}$$

After applying the above diff, iteration 0 is as follows, with the newly inserted tuple highlighted in blue and the deleted tuple highlighted in red:

$$\mathcal{N}_0^{\#} = \left\{ \begin{array}{l} \texttt{new(admin,L1)}^1,\ \texttt{new(sec,L2)}^1,\ \texttt{new(ins,L3)}^1,\ \texttt{new(userSession,nullptr)}^1, \\ \texttt{new(superuser,nullptr)}^1,\ \texttt{store(admin,session,ins)}^1, \\ \texttt{store(admin,session,sec)}^1,\ \texttt{load(superuser,admin,session)}^1, \\ \texttt{assign(userSession,ins)}^1,\ \texttt{load(userSession,admin,session)}^1, \\ \texttt{assign(superuser,userSession)}^1,\ \texttt{assign(superuser,sec)}^0 \end{array} \right\}$$

In iteration 1, the input for the non-recursive rule (i.e., the relation `new`) doesn't change, thus the result is identical to the bootstrap above:

$$\mathcal{N}_1^{\#} = \left\{ \begin{array}{l} \texttt{vpt(admin,L1)}^1,\ \texttt{vpt(sec,L2)}^1,\ \texttt{vpt(ins,L3)}^1, \\ \texttt{vpt(userSession,nullptr)}^1,\ \texttt{vpt(superuser,nullptr)}^1 \end{array} \right\}$$

In iteration 2, however, the deletion of `assign(superuser,sec)` means that the tuple `vpt(superuser,L2)` can no longer be derived from the following rule instantiation:

```
vpt(superuser,L2) :- assign(superuser,sec), vpt(sec,L2).
```

Meanwhile, the insertion of `load(userSession,admin,session)` means that a new tuple, `vpt(userSession,L2)`, can now be derived from

```
vpt(userSession,L2) :- load(userSession,admin,session), store(admin,session,sec),
                       vpt(sec,L2), vpt(admin,L1), vpt(admin,L1).
```

Therefore, the diff in iteration 2 can be expressed as

$$\{\texttt{vpt(superuser,L2)}^{-1}, \texttt{vpt(userSession,L2)}^{+1}\}$$

The result for iteration 2 after applying this diff is

$$\mathcal{N}_2^{\#} = \left\{ \begin{array}{l} \texttt{vpt(userSession,L2)}^1,\ \texttt{vpt(userSession,L3)}^1, \\ \texttt{vpt(superuser,L2)}^1,\ \texttt{vpt(superuser,L3)}^1 \end{array} \right\}$$

At this point, no new tuples can be further derived, either by tuples that already existed previously or by tuples that are newly inserted. Therefore, a fixpoint is reached, and the evaluation terminates.

Our novel sparse computational state requires some notion of *re-discovery* since a tuple is only kept track of in its first iteration. If a tuple is deleted in its first iteration, it may still be

derivable in a later iteration. In this case, the Update algorithm re-discovers whether the tuple is either derived in a later iteration or is truly deleted from the IDB. This re-discovery process is a notion of provenance [55, 4], where we find derivations for tuples that are deleted in earlier iterations.

The incremental update algorithm uses the same extended notation for rule evaluation as introduced in Section 5.3. Recall that the following operator derives tuples from rule instantiations where at least one body tuple is in I_1, and also at least one body tuple is in I_2. The intention is that I_1 represents the deltas between iterations and I_2 represents the diffs between epochs.

$$\Pi^{\#}_{P}[I \mid I_1 \mid I_2] = \left\{ t^v \;\middle|\; \begin{array}{l} v = \text{number of rule instantiations } t :- t_1, \ldots, t_n \\ \text{in } P \text{ where } \{t_1, \ldots, t_n\} \subseteq I \text{ and } \{t_1, \ldots, t_n\} \cap I_1 \neq \emptyset \\ \text{and } \{t_1, \ldots, t_n\} \cap I_2 \neq \emptyset \end{array} \right\}$$

Like in the dense update algorithm, this elastic update algorithm also makes use of a number of auxiliary sets: I^o_k and I_k maintain the full sets of tuples up to iteration k for the previous and current epoch, respectively, I^-_k and I^+_k maintain the tuples that are deleted and inserted, respectively up to iteration k, and N^o_k and N_k are the set projections of $\mathcal{N}^{\#o}_k$ and $\mathcal{N}^{\#}_k$, storing the tuples that are new in iteration k in the previous and current epoch, respectively.

Algorithm 8 is presented for a single stratum and takes the state of the previous epoch $(E, \mathcal{N}^{\#o})$ and the incremental update (E^-, E^+) consisting of a set of tuples to be deleted and a set of tuples to be inserted, respectively. Note that $\mathcal{N}^{\#}$ may be the IDB sequence from the bootstrap stage or the result of a previous incremental update. The algorithm begins by initializing the input state by applying E^- and E^+ and storing the result in $\mathcal{N}^{\#}_0$ (line 3). Then, the algorithm initializes the sets I^-_0 and I^+_0 to be the updates in iteration 0.

In the fixpoint loop, the rule evaluation on line 10 is the core part of this algorithm. This step starts with the multiset of tuples from the previous epoch and applies deletions and insertions resulting from applying Datalog rules with the insertions and deletions for the current epoch. This step is split into three terms: the deletion term, the insertion term, and the re-discovery term. The deletion term, $\Pi^{\#}[I^o_{k-1} \mid N^o_{k-1} \mid I^-_{k-1}] \setminus I^o_{k-1}$, computes tuples that are deleted in the current iteration as a result of a derivation where the body contains both a tuple in the delta (N^o_{k-1}) and a deleted tuple (I^-_{k-1}). The set minus notation excludes tuples that were in earlier iterations in the previous epoch, preventing over-deletion since the tuples would not be present in the current iteration due to sparsification. The insertion term, $\Pi^{\#}[I_{k-1} \mid N_{k-1} \mid I^+_{k-1}] \setminus I_{k-1}$, computes tuples that are inserted as a result of the body of a derivation containing an inserted tuple. Tuples that already exist in previous iterations (i.e., tuples that are contained in I_{k-1}) are excluded to maintain the sparsification invariant. The re-discovery term, $I^-_{k-1} \cap \Pi[I^o_{k-1} \cap I^n_{k-1} \mid N_{k-1}]$, computes tuples that are deleted in previous iterations I^-_{k-1}, but where an alternative derivation exists in the current iteration. In other words, the re-discovery term applies in the situation where a tuple is deleted from some iteration but can still be derived in a later iteration. In this case, the re-discovery term computes this later derivation.

The sparsification term (line 11) does not perform any rule evaluation but excludes tuples from iteration k that were inserted in an earlier iteration (as a result of a new derivation). These

Algorithm 8 Update(P, $(E, \mathcal{N}^{\#o}), (E^-, E^+)$)

Ensure: $E^- \subseteq E$, $E \cap E^+ = \emptyset$

1: $\mathcal{N}_0^{\#} \leftarrow E \setminus E^- \cup E^+$
2: $N_0 \leftarrow \text{Supp}(\mathcal{N}_0^{\#})$
3: $N_0^o \leftarrow \text{Supp}(\mathcal{N}_0^{\#o})$
4: $I_0^- \leftarrow E^-$
5: $I_0^+ \leftarrow E^+$
6: **for all** $k \in \{1, 2, \ldots\}$ **do**
7: $\quad I_{k-1} \leftarrow \cup_{0 \le i \le k-1} N_i$
8: $\quad I_{k-1}^o \leftarrow \cup_{0 \le 1 \le k-1} N_i^o$
9: $\quad$ *▷ Here, $\mathcal{A} \setminus B$ denotes $\{(t^v) \in \mathcal{A} \mid t \notin B\}$
10: $\quad \mathcal{N}_k^{\#} \leftarrow \mathcal{N}_k^{\#o} \ominus (\Pi^{\#}[I_{k-1}^o \mid N_{k-1}^o \mid I_{k-1}^-] \setminus I_{k-1}^o)$ ▷ Deletion term
$\oplus (\Pi^{\#}[I_{k-1} \mid N_{k-1} \mid I_{k-1}^+] \setminus I_{k-1})$ ▷ Insertion term
$\oplus (I_{k-1}^- \cap \Pi^{\#}[I_{k-1}^o \cap I_{k-1} \mid N_{k-1}] \setminus I_{k-1})$ ▷ Re-discovery term
11: $\quad \mathcal{N}_k^{\#} \leftarrow \{(t^v) \in \mathcal{N}_k^{\#} \mid t \notin I_{k-1}^+\}$
12: $\quad N_k \leftarrow \text{Supp}(\mathcal{N}_k^{\#})$
13: $\quad N_k^o \leftarrow \text{Supp}(\mathcal{N}_k^{\#o})$
14: $\quad I_k^- \leftarrow (I_{k-1}^- \setminus N_k) \cup (N_k^o \setminus I_k)$
15: $\quad I_k^+ \leftarrow (I_{k-1}^+ \setminus N_k^o) \cup (N_k \setminus I_k^o)$
16: $\quad$ **if** $N_k = \emptyset$ **then**
17: $\quad\quad$ **return** $(E \setminus E^- \cup E^+, \mathcal{N}^{\#})$
18: $\quad$ **end if**
19: **end for**

tuples should be deleted to maintain the sparsification invariant that a tuple is only present in a single iteration in any given epoch.

The algorithm continues by updating the I_k^- and I_k^+ sets (lines 14 and 15). Computing I_k^- (line 14) takes the deletion set from the previous iteration I_{k-1}^- and excludes the tuples that are newly computed in the current iteration N_k, along with tuples that are deleted in the current iteration $(N_k^o \setminus I_k^n)$. Similarly, computing I_k^+ (line 15) takes the insertion set from the previous iteration and excludes tuples that already existed in the current iteration in the previous epoch (since these tuples already existed, so are not *newly* inserted in the current epoch), along with tuples that are inserted in the current iteration.

The algorithm exits if we have reached a fixpoint and the current iteration is identical to the previous iteration, i.e., if $\mathcal{N}_k^{\#}$ is empty (checked via emptiness of the set projection, in line 16).

Correctness. To show the correctness of our incremental update algorithm, we must show that it computes the same sequence of multisets as if we had applied Bootstrap to the altered input. In other words, we need to show that given a Datalog program P, an input set E, a deletion set E^-, and an insertion set E^+, computing the result directly via Bootstrap$(E^b = E \setminus E^- \cup E^+)$ is equal to Update(Bootstrap$(E), (E^-, E^+)$). The central parts of the algorithm computing

these results are lines 10 and 11. Before the final correctness proof, we need some intermediate properties of the N_k sets and the I^- and I^+ sets. The following important properties are that the validity properties of the E sets (i.e., that $E^+ \cap E = \emptyset$ and $E^- \subseteq E$) also hold for the I^o, I^-, and I^+ sets during the incremental update algorithm. Similar properties relating I^- and I^+ sets to the current epoch's I sets are also required. The eventual goal is to show that $I_k = I^o_k \setminus I^-_k \cup I^+_k$ for each iteration k, which is an important result for showing the correctness of the rule evaluations.

Lemma 5.4.3. *For each iteration k, the I^-_k and I^+_k sets are correct in that (1) $I^-_k \subseteq I^o_k$ and $I^-_k \cap I_k = \emptyset$, and (2) $I^+_k \cap I^o_k = \emptyset$ and $I^+_k \subseteq I_k$.*

Proof. This proof is by induction over the iterations. For $k = 0$, $I^-_0 = E^-$ and $I^+_0 = E^+$ by definition, so properties (1) and (2) hold.

The induction hypothesis is that for iteration $k-1$, we have $I^-_{k-1} \subseteq I^o_{k-1}$, $I^-_{k-1} \cap I_{k-1} = \emptyset$, $I^+_{k-1} \cap I^o_{k-1} = \emptyset$, and $I^+_{k-1} \subseteq I_{k-1}$.

For property (1), we show $I^-_k \subseteq I^o_k$. Consider line 14 of Algorithm 8, where $I^-_k \leftarrow (I^-_{k-1} \setminus N_k) \cup (N^o_k \setminus I_k)$. In the first part of the union, $I^-_{k-1} \subseteq I^o_{k-1}$ by the induction hypothesis. Therefore, also $I^-_{k-1} \subseteq I^o_k$, since I^o_{k-1} grows monotonically. In the second part of the union, $N^o_k \subseteq I^o_k$ by definition of I^o_k. Therefore, $I^-_k \subseteq I^o_k$.

To show that $I^-_k \cap I_k = \emptyset$, consider the same line. In the first part of the union, $I^-_{k-1} \cap I_{k-1} = \emptyset$ by the induction hypothesis. We then exclude N_k, and since $I_k = I_{k-1} \cup N_k$ by definition, then $(I^-_{k-1} \setminus N_k) \cap I_k = \emptyset$. In the second part of the union, we exclude I_k. Therefore, $I^-_k \cap I_k = \emptyset$.

Property (2) holds by similar arguments on line 15. □

As a corollary, we can show that the I^- and I^+ sets are correct.

Corollary 5.4.4. *For each iteration k, we have $I_k = I^o_k \setminus I^-_k \cup I^+_k$.*

Proof. We first show that $I^o_k \setminus I_k = I^-_k$ by showing both directions of inclusion. The reverse direction, i.e., that $I^-_k \subseteq I^o_k \setminus I_k$ is a direct corollary of Lemma 5.4.3, that $I^-_k \subseteq I^o_k$ and $I^-_k \cap I_k = \emptyset$. For the forward direction, consider some tuple $t \in I^o_k \setminus I_k$. Then, t must be in some $N^o_i \setminus I_k$ for some $i \leq k$. Since $I_i \subseteq I_k$, t is also in $N^o_i \setminus I_i$. Therefore, $t \in I^-_i$. Also, t cannot be removed from I^- in a later iteration, since $t \notin I_k$, and therefore, $t \in I^-_k$.

We have shown both directions of inclusion, and therefore, $I^o_k \setminus I_k = I^-_k$. By a similar argument, $I_k \setminus I^o_k = I^+_k$. From these equalities:

$$\begin{aligned} I^o_k \setminus I^-_k \cup I^+_k &= I^o_k \setminus (I^o_k \setminus I_k) \cup (I_k \setminus I^o_k) \\ &= (I^o_k \cap I_k) \cup (I_k \setminus I^o_k) = I_k \end{aligned}$$

□

It remains to be shown that Update is correct. Our criteria for correctness is that it computes the same sequence of multisets as if we had applied the bootstrap algorithm to the updated input, i.e., that the multisets $\mathcal{N}^\#_i$ as computed by Update and Bootstrap are the same for each iteration i. The following is the central theorem for our correctness proof.

Theorem 5.4.5. *Given P, E, E^-, and E^+ as above, $\mathcal{N}_i^\#$ as computed by Update$(P, \text{Bootstrap}(P, E), (E^-, E^+))$ is equal to $\mathcal{N}_i^\#$ as computed by Bootstrap$(P, E \setminus E^- \cup E^+)$ for each iteration i.*

The proof of Theorem 5.4.5 is by induction over the iterations, and in each step, it considers all four parts of lines 10 and 11. By arguments over which sets each tuple is contained in, and careful consideration of the subset relationships between them, we can show that the counting multisets are the same as those produced by Bootstrap. The formal proof is as follows:

Proof. For this proof, we mainly consider the underlying sets of derivations rather than the counting multisets, since the counting multisets do not distinguish between different derivations. We introduce some new notations to represent sets of derivations: $\mathcal{N}_i^D$ and $\mathcal{N}_i^{Do}$ are the sets of derivations in iteration i of the current and previous epochs, respectively. Furthermore, we introduce the following notation to convert between derivations and tuples: $\phi((t :- t_1, \ldots, t_n)) := t$ takes the head tuple of a derivation. We also denote $\mathcal{N}_i^\#$ computed by Bootstrap to be $\mathcal{B}_i^\#$, to distinguish it from $\mathcal{N}_i^\#$ computed by Update.

The proof is by induction over the iterations. The initial step, where $k = 0$, is true since both $\mathcal{B}_0^\#$ and $\mathcal{N}_0^\#$ take on the value of $E \setminus E^- \cup E^+$, where every tuple has a count of 1.

The induction hypothesis is that for all $0 \leq i < k$, we have $\mathcal{B}_i^\# = \mathcal{N}_i^\#$. We consider each of the four terms in lines 10 and 11. We first need to show that the sets of derivations computed by these lines are disjoint, so that the algorithm does not double count.

- For the deletion term (we label it (1)), we have the derivations $\{d \in \Pi^D[I_{k-1}^o \mid N_{k-1}^o \mid I_{k-1}^-] \mid \phi(d) \notin I_{k-1}^o\}$.
- For the insertion term (labelled (2)), we have derivations $\{d \in \Pi^D[I_{k-1} \mid N_{k-1} \mid I_{k-1}^+] \mid \phi(d) \notin I_{k-1}\}$. Since $I_{k-1}^+ \cap I_{k-1}^o = \emptyset$ (from Corollary 5.4.4), then $(2) \cap (1) = \emptyset$, since (1) takes derivations only from I_{k-1}^o.
- For the re-discovery term (labelled (3)), we have derivations $\{d \in \Pi^D[I_{k-1}^o \cap I_{k-1} \mid N_{k-1}] \mid \phi(d) \in I_{k-1}^-$ and $\phi(d) \notin I_{k-1}\}$. Since this takes derivations from I_{k-1}^o, and $I_{k-1}^o \cap I_{k-1}^+ = \emptyset$, then $(3) \cap (2) = \emptyset$. Also, since $I_{k-1}^- \subseteq I_{k-1}^o$ (from Corollary 5.4.4), we have $(3) \cap (1) = \emptyset$, since (1) excludes tuples from I_{k-1}^o.
- For the sparsification term (labelled (4)), we have $\mathcal{N}_k^\# \ominus I_{k-1}^+$. However, note that this term is processed after the three other terms. Therefore, it naturally excludes (1), and so $(4) \cap (1) = \emptyset$. Moreover, we have $I_{k-1}^+ \subseteq I_{k-1}$ (from Corollary 5.4.4), and so $(4) \cap (3) = \emptyset$ and $(4) \cap (2) = \emptyset$, since both (3) and (2) exclude I_{k-1}.

Since all 4 terms produce disjoint derivations, the algorithm does not double count when adding or removing any derivations. Next, we need to prove that for any derivation in $\mathcal{N}_k^{Do}$ and not in $\mathcal{N}_k^D$, it is removed by one of the four terms and vice versa.

Consider a derivation $d \in \mathcal{N}_k^{Do} \setminus \mathcal{N}_k^D$. By definition, $d \in \{d \in \Pi^D[I_{k-1}^o \mid N_{k-1}^o] \mid \phi(d) \notin I_{k-1}^o\} \setminus \{d \in \Pi^D[I_{k-1} \mid N_{k-1}] \mid \phi(d) \notin I_{k-1}\}$. Then, there are two cases. The first case is that $\phi(d) \in I_{k-1}$. In this case, also $\phi(d) \notin I_{k-1}^o$, by our assumption, and so $\phi(d) \in I_{k-1}^+$ (by

Corollary 5.4.4). Therefore, d would be removed by the sparsification term which removes all tuples that are in I^+_{k-1}. The second case is that $d \notin \Pi^D[I_{k-1} \mid N_{k-1}]$. In this case, one of the body tuples of d is in $I^o_{k-1} \setminus I_{k-1}$ (or in $N^o_{k-1} \setminus N_{k-1}$, which implies also that it is in $I^o_{k-1} \setminus I_{k-1}$), which equals I^-_{k-1} (by Corollary 5.4.4). Therefore, $d \in \Pi^D[I^o_{k-1} \mid N^o_{k-1} \mid I^-_{k-1}]$, and since $\phi(d) \notin I^o_{k-1}$ by assumption, it would be removed by the deletion term.

Now, for the opposite case, consider a derivation $d \in \mathcal{N}^D_k \setminus \mathcal{N}^{Do}_k$. We want to show that this derivation is inserted by one of the four terms. By definition, $d \in \{d \in (\Pi^D[I_{k-1} \mid N_{k-1}] \mid \phi(d) \notin I_{k-1}\} \setminus \{d \in \Pi^D[I^o_{k-1} \mid N^o_{k-1}] \mid \phi(d) \notin I^o_{k-1}\}$. Like the deletion case, there are two cases. The first is that at least one of the body tuples of d are in $I_{k-1} \setminus I^o_{k-1}$. Then, this tuple is in I^+_{k-1}, and therefore, $d \in \Pi^D[I_{k-1} \mid N_{k-1} \mid I^+_{k-1}]$. Since $\phi(d) \notin I_{k-1}$ by assumption, then d will be inserted by the insertion term. The second case is if $\phi(d) \in I^o_{k-1}$. Then, since $\phi(d) \notin I_{k-1}$, $\phi(d) \in I^o_{k-1} \setminus I_{k-1} = I^-_{k-1}$. If the first case doesn't hold, we know that all of the body tuples are not in $I_{k-1} \setminus I^o_{k-1}$, and therefore, they must all be in I^o_{k-1}. Therefore, $d \in \Pi^D[I^o_{k-1} \cap I_{k-1} \mid N_{k-1}]$. Since $\phi(d) \notin I_{k-1}$ by assumption, d would be inserted by the re-discovery term. □

Another essential property of our elastic incremental evaluation strategy is the *sparsification invariant*, which describes the main difference between our elastic algorithm and the dense algorithm.

Lemma 5.4.6 (Sparsification Invariant)**.** *For each iteration k, the sets N_k are disjoint.*

This property ensures that every tuple is only computed in a single iteration, with this iteration being the earliest one in which it is computed.

Re-discovery Rules as a Notion of Provenance. The re-discovery term in the rule evaluation part of Algorithm 8 (the last term in line 10) is critical for maintaining the sparsification property of our algorithm. In particular, a tuple may be deleted in some iteration but still be derivable in a later iteration via a different rule or different body tuples. The re-discovery term allows the algorithm to recover these tuples in the later iteration.

The re-discovery is performed by the rule evaluation term $I^-_{k-1} \cap \Pi[I^o_{k-1} \cap I^n_{k-1} \mid N_{k-1}]$, which states that we compute tuples that were deleted in an earlier iteration (i.e., exist in I^-_{k-1}), but an alternative derivation exists in the current iteration ($\Pi[I^o_{k-1} \cap I^n_{k-1} \mid N_{k-1}]$).

Provenance, as described in Chapter 4, can be defined as "discovering the derivations for a tuple". In a similar vein, the re-discovery term discovers derivations in the current iteration for tuples that were deleted in earlier iterations. In particular, for each tuple deleted in an earlier iteration, the re-discovery term finds derivations from $I^o_{k-1} \cap I^n_{k-1}$. Since this process resembles provenance, the re-discovery rules in the Update algorithm are akin to the backward rule evaluation techniques in Chapter 4.

5.4.3 Stratified Negation and Constraints

Our algorithms thus far have omitted any notion of negation or constraints. However, both negation and constraints are powerful and common extensions of Datalog. Constraints are a simpler case than negation, and may take the form of arithmetic constraints such as `A < B`

or `A != B` where A and B are *grounded* variables (i.e., variables also occurring in a positive body predicate) or constants. In an instantiated rule, a constraint is satisfied if the instantiated arithmetic constraint is satisfied. For example,

```
alias(Var1,Var2) :- vpt(Var1,Obj), vpt(Var2,Obj), Var1 != Var2, Obj != nullptr.
```

is a rule with arithmetic constraints, and an instantiation of the rule only derives a tuple if the inequality constraint is satisfied by the values given to `Var1` and `Var2`. Since the truth value of a constraint never changes (e.g., $1 = 1$ is always true, and the semantics of $=$ never changes even under an incremental update), constraints need no special treatment. A tuple is only computed if all constraints in the rule hold true, and these constraint semantics do not change.

However, stratified negation is more complicated than simple arithmetic constraints. With incremental evaluation, the truth value of a negation may change due to tuples being inserted or deleted from the negated relation, unlike constraints which do not change truth value after an incremental update. To adapt our Datalog evaluation algorithms to support stratified negation and constraints, the rule evaluation is extended to support these features. The rule evaluation operator, $\Pi^{\#}$, is extended so that

$$\Pi^{\#}_{P}[I \mid I_{\text{in}}] = \left\{ t^{v} \;\middle|\; \begin{array}{l} v = \#\text{instantiations } t :\!- t_1, \ldots, t_n, !t_{n+1}, \ldots, !t_{n+m}, \psi \\ \text{in } P \text{ where } \{t_1, \ldots, t_n\} \subseteq I,\ \{t_1, \ldots, t_n\} \cap I_{\text{in}} \neq \emptyset, \\ \{t_{n+1} \ldots t_{n+m}\} \cap I = \emptyset, \text{ and } \psi \text{ is satisfied} \end{array} \right\}$$

where ψ denotes the instantiated arithmetic constraints occurring in the rule. Replacing the rule evaluation operator in Bootstrap (Algorithm 7) with this extended version allows the algorithm to support stratified negation and constraints. However, the extension is more involved for Update since introducing negation also introduces new cases for deleting/inserting tuples. For example, consider the rule

```
path(X,Z) :- edge(X,Y), path(Y,Z), !edge(X,Z).
```

If we have a rule instantiation

```
path(a,c) :- edge(a,b), path(b,c), !edge(a,c).
```

where `edge(a,c)` is inserted as a result of an incremental update, then the head tuple `path(a,c)` must be deleted since the negation is no longer satisfied. The opposite situation may arise where the deletion of a tuple may lead to the consequent insertion of a tuple. Therefore, we further extend the rule evaluation operator so that

$$\Pi^{\#}_{P}[I \mid I_1 \mid I_2, I_2'] = \left\{ t^{v} \;\middle|\; \begin{array}{l} v = \#\text{instantiations } t :\!- t_1, \ldots, t_n, !t_{n+1}, \ldots, !t_{n+m}, \psi \\ \text{in } P \text{ where } \{t_1, \ldots, t_n\} \subseteq I,\ \{t_1, \ldots, t_n\} \cap I_1 \neq \emptyset, \\ \{t_{n+1} \ldots t_{n+m}\} \cap I = \emptyset,\ \psi \text{ is satisfied, and} \\ (\{t_1, \ldots, t_n\} \cap I_2 \neq \emptyset \text{ or } \{t_{n+1} \ldots, t_{n+m}\} \cap I_2' \neq \emptyset) \end{array} \right\}$$

With this rule evaluation operator, a new tuple is derived if the rule instantiation contains body tuples from I, where at least one positive body tuple is also in I_1, and either there is a

positive body tuple in I_2 or a negative body tuple in I_2'. Using this notation, the rule evaluation step of Algorithm 8 (line 10) becomes

$$\begin{aligned}\mathcal{N}_k^{\#} \leftarrow \mathcal{N}_k^{\#} &\ominus (\Pi^{\#}[I_{k-1}^{o} \mid N_{k-1}^{o} \mid I_{k-1}^{-}, I_0^{+}] \setminus I_{k-1}^{o}) \\ &\oplus (\Pi^{\#}[I_{k-1} \mid N_{k-1} \mid I_{k-1}^{+}, I_0^{-}] \setminus I_{k-1}) \\ &\oplus (I_{k-1}^{-} \cap \Pi[I_{k-1}^{o} \cap I_{k-1} \mid N_{k-1}])\end{aligned}$$

where the first and second terms now handle stratified negation. The deletion term, $\Pi^{\#}[I_{k-1}^{o} \mid N_{k-1}^{o} \mid I_{k-1}^{-}, I_0^{+}] \setminus I_{k-1}^{o}$, now computes tuples that are deleted, either as a result of a deleted positive body tuple (I_{k-1}^{-}) or an inserted negated body tuple (I_0^{+}). We use iteration 0 for the negated tuples since stratified negation enforces that negations must be from the input of the current stratum (i.e., from previous strata). Similarly, the insertion term, $\Pi^{\#}[I_{k-1}^{n} \mid N_{k-1} \mid I_{k-1}^{+}, I_0^{-}] \setminus I_{k-1}^{n}$, computes tuples that are inserted either as a result of an inserted positive body tuple or a deleted negative body tuple. The other parts of the algorithms involve manipulating and merging relations and are independent of the Datalog rules. Therefore, no changes are needed to support negation or constraints. Hence, with the extensions to the rule evaluation presented above, our algorithms fully support Datalog with stratified negation and constraints.

5.5 Implementation in Soufflé

In this section we outline how our approach is integrated in the Soufflé Datalog engine, including several optimizations to support more efficient incremental evaluation.

5.5.1 Core Implementation

Specialized Data Structures. Soufflé internally uses highly specialized, parallel B-tree data structures to store relations. For incremental evaluation, we associate each tuple with an iteration number and a count in a similar fashion to the provenance annotations required in Chapter 4. Therefore, we similarly extend the internal data structures to allow for these auxiliary attributes. Importantly, these auxiliary attributes may be *updated*, e.g., if a new derivation is discovered, the count must be incremented. Thus, we implemented an update mechanism and adapted the existing optimistic locking mechanism to support parallel operation.

Auxiliary Relations. Auxiliary relations are necessary to represent the tuples that are inserted or deleted in Algorithm 8. These auxiliary relations are represented by separate instantiations of the original relations, with prefixes `diff_plus` and `diff_minus`, respectively. These `diff_plus` and `diff_minus` relations are not exact analogs of I^{+} and I^{-}, since `diff_plus` and `diff_minus` may contain tuples where the derivation count is incremented/decremented, rather than only tuples that are fully inserted/deleted.

Rule Evaluation. We extend the operations used in standard rule evaluation algorithms in Soufflé to support the extra operations required by the incremental evaluation algorithms. Soufflé uses nested loop joins for evaluating rules, which incorporate extra conditions and existence

checks to ensure correctness. For incremental evaluation, further specialized existence checks are required, e.g., a tuple in `diff_minus` may not actually be deleted, and only one of its derivations is deleted. Therefore, we need a specialized existence check that uses its count in the full relation to determine if the tuple is fully deleted or not. The Datalog rules are then instrumented for incremental evaluation using these extra operations and auxiliary relations. Moreover, separate versions of rule instrumentation are required for the Bootstrap and Update algorithms.

Other Operations. Other operations, such as merges between iterations and a cleanup operation between epochs, are also required, along with the rule evaluation extensions. In standard semi-naïve evaluation, at the end of each iteration, new tuples computed in that iteration are merged into the full relation, and this also becomes the delta for the following iteration. For incremental evaluation, further operations may take place, e.g., eager computation of the delta of the previous epoch and eager computation of `diff_plus` and `diff_minus`. In between epochs, the incremental evaluation algorithms also require a cleanup stage, where the `diff_plus` and `diff_minus` relations are merged into the full relations to update the state in preparation for the following epoch.

5.5.2 Optimizations

Eager vs. Lazy `diff_plus` and `diff_minus`. The `diff_plus` and `diff_minus` relations store tuples that are inserted and deleted in the current epoch, respectively. However, there is extra computation involved with the `diff_plus` and `diff_minus` relations in lines 14 and 15 of Algorithm 8. Here, a tuple in `diff_plus` may not actually be newly inserted - it may be a new derivation for a tuple that already existed. Similarly, a tuple in `diff_minus` may not actually be deleted - an alternative derivation may still hold. Thus, we need to check the full relation to determine if a tuple in `diff_plus` or `diff_minus` is actually inserted or deleted, respectively. This check may be performed eagerly during the merge step in each iteration, with results stored in separate relations `actual_diff_plus` and `actual_diff_minus`, or lazily inside the rule evaluation. For the sake of clarity, our algorithms are presented with eager diff computations, which can be seen in lines 14 and 15. A lazy diff version would incorporate this computation directly in the rule evaluation. This design decision is a trade-off: eagerly computing `diff_plus` and `diff_minus` may result in wasted computation for tuples that are not considered in any rules, while lazy computation may mean the same check of the full relation is performed multiple times for a single tuple, if it occurs in multiple rule derivations. However, our experiments indicate that this trade-off generally favors eager diffs, where it can amortize the checks for tuples that occur in multiple rule derivations. For our benchmarks, the difference is generally within 15% in favor of eager diffs, but it can provide up to 4× speed up in some situations where tuples are frequently repeated in multiple rule derivations.

Filtering for Re-discovery Rules. The elastic algorithm includes the notion of *re-discovery*, which is required due to its sparsification. In the re-discovery rules, the algorithm finds all tuples

which have been deleted in an earlier iteration but where an alternative derivation still exists for the current iteration. Naïvely, this could be done by instrumenting a rule to filter on `diff_minus`:

$$\texttt{R} :- \texttt{diff\_minus\_R}, \texttt{R}_1, \ldots, \texttt{R}_k.$$

However, in some cases this can cause a problematic join, if there are few variables in common between the `diff_minus_R` atom and the remaining atoms. For example,

$$\texttt{R}(x, y, z) :- \texttt{diff\_minus\_R}(x, y, z), \texttt{R}_1(x, a), \texttt{R}_2(y, a), \texttt{R}_3(z, a).$$

may cause duplication of work in $\texttt{R}_1(x, a)$ if there are many tuples in `diff_minus_R` with the same x value. Our solution is to divide the `diff_minus` relation so that it never causes extra work.

$$\begin{aligned}\texttt{R}(x, y, z) :- \; & \texttt{diff\_minus\_R}_x(x), \texttt{R}_1(x, a), \texttt{R}_2(y, a), \\ & \texttt{diff\_minus\_R}_y(y), \texttt{R}_3(z, a), \texttt{diff\_minus\_R}_z(z).\end{aligned}$$

Dividing the `diff_minus` relations ensures that each variable only acts as a filter and cannot multiply the work of the other atoms in the rule. Here, the x variable is scheduled first since we assume that $\texttt{diff\_minus\_R}_x(x)$ is smaller than $\texttt{R}_1(x, a)$. However, the other variables must be scheduled after their corresponding atom to prevent a cross-product with the previous atom.

This strategy of considering the variables in the filtering atom is inspired by worst-case optimal joins [119, 120]. These worst-case optimal join algorithms work by considering the *variables* in the atoms in some order, in contrast to traditional nested loop join algorithms that consider an atom order. For our re-discovery rules, this variable-based approach is used only for the filtering atom since the filtering atom is often performance-critical. Our benchmarks show that this technique is generally 2.5× faster than the naïve strategy, while in some situations, it can be up to 15× faster.

Scheduling. Scheduling for join orders plays a vital role in the performance of Datalog rules [121, 122, 37]. With incremental evaluation, the assumption that the diffs are smaller than the full relations allows for better heuristics for automatic scheduling. Using this assumption, scheduling `diff_plus` or `diff_minus` first in a rule evaluation generally improves performance by restricting the size of the search as early as possible. However, care must be taken to avoid cross-products. For example, consider the following rule:

$$\texttt{R}(a, d) :- \texttt{R}_1(a, b), \texttt{R}_2(b, c), \texttt{diff\_minus\_R}_3(c, d).$$

In this case, moving $\texttt{diff\_minus\_R}_3(c, d)$ to the front of the rule would create a cross-product with $\texttt{R}_1(a, b)$ and may lead to worse performance than the original schedule. Hence, using simple automatic scheduling techniques, such as maximizing the number of bound variables in each atom, is crucial to maintain the performance of incremental evaluation.

5.6 Experimental Evaluation

This experimental section aims to demonstrate the following claims:

- **Claim I:** Inviability of single incremental evaluation strategies on variable update use cases.
- **Claim II:** The elastic incremental evaluation with a simple switch heuristic performs better compared to existing single strategy incremental evaluations, both in terms of runtime and memory usage, over a series of varying sized incremental updates.

Experimental Setup. Our experiments are run on an AMD Threadripper 2990WX machine with 128 GB memory, running Ubuntu 20.10 with GCC 10.2 used to generate all Soufflé executables. All experiments are run with 8 threads, and all I/O time is excluded from measurements.

We evaluate three versions of Soufflé: (1) Soufflé: Non-Incremental Soufflé engine. (2) Soufflé-dense: The dense incremental evaluation algorithm, similar to DDLog, implemented and optimized for Soufflé. (3) Soufflé-elastic: The implementation of the elastic incremental evaluation strategy. When necessary, we differentiate between elastic-update and elastic-bootstrap algorithms.

We also compare our approach to an industrial-strength incremental Datalog engine, Differential Datalog (DDLog) [52], which uses Differential Dataflow [50] as a backend. DDLog with Differential Dataflow is a state-of-the-art incremental engine that uses a variant of the counting algorithm.

We perform our evaluations using a set of dynamic Datalog use cases adapted by Frank McSherry [1] for benchmarking incremental Datalog engines. The use cases are described below:

1. Doop [20]: a points-to program analysis framework for Java programs. This is a subset of the Doop program analysis library ported to DDLog. This use case contains a large number of rules and relations with complex recursion.
2. CRDT: an implementation of a conflict-free replicated data type in Datalog. This use case represents an in-between ruleset with a medium number of rules, relations of moderate complexity, and arithmetic constraints.
3. Galen [123]: a medical ontology inference task implemented in Datalog. This use case represents a typical ontological use case consisting of a small number of rules and relations with a simple recursive structure. However, the joins in Galen can be challenging.

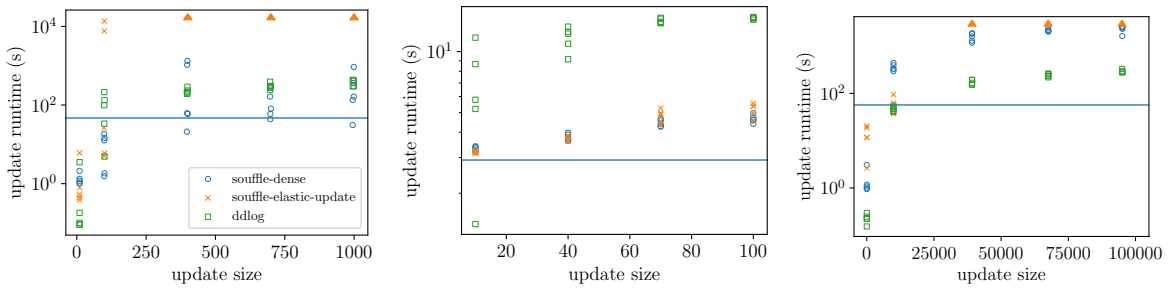
Some basic statistics for the benchmarks are included in Table 5.1. To evaluate the performance of incremental evaluation algorithms, we generated update sets of varying sizes for each benchmark by randomly choosing a subset of EDB tuples that are incrementally deleted and inserted. We acknowledge that these randomly generated update sets may not be representative of real-world workloads for these benchmarks, and thus they test only the incremental evaluation strategy in isolation.

[1] https://github.com/frankmcsherry/dynamic-datalog

Table 5.1: Benchmark Statistics

Benchmark	Number of rules	EDB size	IDB size
Doop	90	11,014,960	41,665,029
CRDT	31	259,778	2,668,247
Galen	6	976,552	24,483,561

5.6.1 Single Strategy Incremental Evaluation



(a) Doop (b) CRDT (c) Galen

Figure 5.3: Incremental update size vs. runtime. The horizontal line in each figure is the runtime of non-incremental Soufflé on the respective benchmark, and the upwards arrows indicate timeouts.

In this set of experiments, we only consider single strategy evaluations; that is, we only include our Update (elastic-update) evaluation and thus do not switch to Bootstrap. These experiments do not establish the supremacy of any one technique. Rather, we show that single strategies are not viable compared to non-incremental evaluation. The results for the runtime of incremental updates for each evaluation implementation are shown in Figure 5.3, while the impact sizes are in Table 5.2. These results are computed for one *cycle* of an update set. An update set is a randomly selected subset of EDB tuples. A cycle consists of one epoch where the update set is deleted, followed by one epoch where the update set is inserted. The horizontal line on each benchmark represents the runtime if non-incremental Soufflé performs the same task, i.e., running the whole benchmark twice from scratch. For each benchmark, there is a general trend that larger updates require more runtime. However, this performance is highly unpredictable, even if the size of the incremental update is constant.

Consider the performance of incremental evaluation for Doop in Figure 5.3a. Here, there are five separate small update sets, which are each generated by randomly choosing 10 EDB tuples and running one cycle. These small updates all finished within two seconds, which is vastly faster than non-incremental Soufflé. For these small incremental updates, all evaluations were very fast on average due to their very low impact, only affecting up to 25 of the IDB tuples. DDlog and elastic-update performed well and on par. Our general observation is that incremental evaluation is highly effective for these lightweight updates. For the 100 update size, the smallest impact was 88 IDB tuples, and the largest impact was 53,816 IDB tuples. As anticipated, this increased the variability of the results. Elastic-update exhibited large extremities, finishing within 5 seconds

Benchmark	Size of update				
	Impact of update				
Doop	10	100	400	700	1,000
	7, 19, 25	88, 22K, 53K	27K, 82K, 5.2M	49K, 356K, 3.6M	87K, 670K, 6.6M
CRDT	10	40	70	100	
	3.4K, 13K, 35K	41K, 50K, 61K	64K, 75K, 81K	86K, 89K, 91K	
Galen	10	10,000	40,000	70,000	100,000
	810, 2.0K, 3.7K	4.3M, 5.7M, 7.0M	21M, 27M, 32M	35M, 41M, 43M	43M, 46M, 54M

Table 5.2: The minimum, median, maximum impact for updates of each size; the impact is the overall number of IDB tuples inserted or deleted, K denotes thousands, M denotes millions

for the fastest, while more than 5,000 seconds for two update sets. DDLog also had high variance, with the fastest runtime being 5 seconds and the slowest being 213 seconds, well over the non-incremental engine time. Curiously, the fastest incremental update was also one of the higher impact ones, affecting 22,347 IDB tuples, while the slowest affected 140 IDB tuples, indicating that neither the size of the EDB updates nor the size of the impact is always helpful in predicting the runtime of the incremental update. While Soufflé-dense was generally faster than DDLog, it still exhibited a large variance, with runtimes ranging between 1.4 and 18 seconds. For the larger update sets, containing 400, 700, and 1000 tuples, respectively, all evaluation strategies failed to compete with non-incremental Soufflé. For example, elastic-update was unable to complete any of the update sets within the time limit. These timed-out updates contained tuples deep in a complex recursive structure, indicating that the elastic-update algorithm does not handle these large impact updates well. Likewise, the counting algorithms implemented in both DDLog and in Soufflé exhibited generally poor performance compared to non-incremental Soufflé. Furthermore, these larger updates exhibited even greater variability, particularly for Soufflé-dense.

The results for CRDT, in Figure 5.3b tell a similar story. Here, even small updates consisting of 10 EDB tuples exhibit unpredictable and poor performance. In comparison to Doop, the small updates for CRDT have a much larger impact, affecting between 3,444 and 35,130 IDB tuples. However, even this larger impact is around 1% of the IDB, and even with these overall small impacts, the runtime of incremental update is considerably slower than re-running the computation from scratch in Soufflé. Similar to Doop, the performance for larger updates only gets worse. For updates containing 40 EDB tuples, the runtimes varied between 9 and 13 seconds. While this variation is smaller than for Doop, the result still indicates that the performance of incremental evaluation is unpredictable. For larger updates containing 70 and 100 EDB tuples, DDLog was around 5× slower than non-incremental Soufflé, despite the update being only around 0.04% of the EDB and impacting only up to 3.4% of the IDB tuples. Update and Soufflé-dense were both more performant, but still slower than non-incremental Soufflé. The poor performance of incremental update algorithms may be due to the structure of the Datalog rules in CRDT. The rules contain several arithmetic inequality constraints, which cannot be indexed, and are checked after the corresponding value is known in the join. Therefore, incremental strategies that use indices to limit the computation to updated tuples are ineffective in the presence of performance-critical inequalities. It is also interesting to note that the impact on

the IDB tuples was much more consistent for CRDT when compared with Doop. For example, with updates containing 100 EDB tuples, the impact on IDB tuples ranged between 85,726 and 91,384 tuples. This may be due to the much simpler structure of the CRDT application, which contains a larger pre-processing stage followed by a very small recursive stratum.

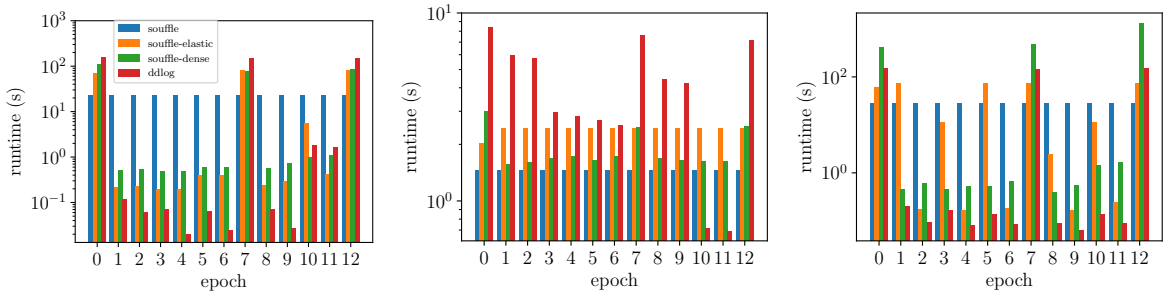
On the other hand, Galen performed far better with DDLog for incremental evaluation. One reason for this is that Galen has a simple ruleset consisting of only 6 Datalog rules but with challenging join characteristics. DDLog is better optimized for these joins, and can outperform Soufflé for these incremental workloads. For small updates consisting of 10 EDB tuples, an incremental update takes between 0.1 and 0.2 seconds, providing far superior performance compared to a non-incremental engine. Even for medium-sized updates consisting of 10,000 EDB tuples, DDLog's incremental update performance is generally faster than non-incremental Soufflé. Only when we consider larger updates of 40,000, 70,000, and 100,000 EDB tuples, or 4%, 7%, and 10% respectively, does the performance of incremental evaluation slow down considerably compared to non-incremental Soufflé. The impact of these larger updates on the IDB is up to 53M tuples, which is almost double the original IDB size. This impact indicates that not only are most of the IDB tuples affected, but they are even affected in multiple iterations. Given this large impact, it is no surprise that the runtime for such an incremental update is slower than simply recomputing the result from scratch. Overall, DDLog performs well on this benchmark compared to non-incremental Soufflé. For Galen, the Soufflé incremental strategies do not perform as well. Soufflé-dense is generally an order of magnitude slower for updates than DDLog, due to unfavorable join orderings. Elastic-update fares even worse, timing out for the larger updates above 10,000 EDB tuples. This is a result of these updates impacting tuples across multiple iterations, which the sparsification of the elastic strategy does not handle well.

These results indicate that state-of-the-art single strategy incremental evaluation algorithms perform well on small impact updates. However, they may be outperformed by a standard non-incremental Datalog engine for more complex applications or high-impact changes. Overall, we demonstrate Claim I by highlighting the unpredictability and tendency for degraded performance of single strategy evaluations on large impact updates compared to non-incremental Soufflé.

5.6.2 Elastic Incremental Evaluation

In this section, we evaluate the performance of our elastic incremental evaluation strategy. That is, we evaluate the combination of the Update algorithm with Bootstrap. We use an empirically determined switching parameter of 20% to determine when to use the Update and when to switch to Bootstrap. That is, if the update time is more than 20% of the previous bootstrap time, we restart using Bootstrap.

For this experiment, we use example workloads for incremental evaluation, which consists of 13 epochs. The first epoch is the initial evaluation, then the following 6 epochs are small updates (containing 10 tuples), with alternating deletion and insertions. These are followed by one large update (1,000 for Doop, 100 for CRDT, and 100,000 for Galen) in epoch 7, then followed by another 4 small updates, with a large update as the final epoch. We note that these patterns may appear in all three of these benchmarks. For Doop, there is a common



(a) Doop (b) CRDT (c) Galen

Figure 5.4: Runtimes for an elastic workload. For each benchmark, the first epoch is an initial evaluation, followed by 6 epochs of small updates, then one large update, then 4 epochs of small updates, then one large update.

pattern of software updates consisting of a large refactor, followed by several smaller commits addressing minor comments. For CRDT, an application commonly used for collaborative online text editing, a large update may result from a large portion of text being moved around, while a smaller update may result from smaller additions or deletions from the text. For Galen, a medical ontology application associated with patient diagnosis, a large update may result from a medical test result being updated, while a smaller update may result from a minor symptom change.

For Doop, in Figure 5.4a, all of the incremental evaluation strategies are able to effectively incrementalize for the small updates. However, the main differences across the full workload result from the bootstrap strategy, with both the initial evaluation and the large updates being faster or on-par with the state-of-the-art counting strategy. As a result, the elastic incremental strategy can complete this workload in 245 seconds, compared to 284 seconds for Soufflé-dense and 467 seconds for DDLog. In comparison, non-incremental Soufflé, which evaluates each epoch from scratch, achieves 304 seconds for this workload. Thus, this use case demonstrates that an elastic incremental evaluation is effective for the complex Doop benchmark. Overall, we demonstrate an amortized net gain compared to non-incremental Soufflé as well as single strategy evaluations.

For CRDT, in Figure 5.4b, none of the incremental evaluation strategies are effective for reasons illustrated in Section 5.6.1, even for the small updates. Here, the elastic strategy hits the 20% heuristic threshold for all updates, despite Update strategy actually being slightly faster than Bootstrap if it were allowed to run to completion. For this workload, non-incremental Soufflé completes all epochs in 19 seconds, followed by 25 seconds for the Soufflé-dense, 31 seconds for Soufflé-elastic, and 56 seconds for DDLog. For this particular application, we conclude that incremental evaluation is ineffective in general.

For Galen, in Figure 5.4c, the incremental evaluation strategies were able to perform reasonably well. For epochs 1 and 5, the elastic update strategy reached the 20% heuristic threshold, thus triggering a bootstrap. If this threshold were not in place, the elastic update would have been faster for these small updates. Despite this, Soufflé-elastic is still highly competitive compared to the other incremental evaluation strategies, being able to finish the workload in 384

seconds, compared to 445 seconds for DDLog. Soufflé-dense was ineffective for the large updates for Galen and times out overall. In comparison, non-incremental Soufflé required 370 seconds for this workload. The results demonstrate that our elastic evaluation is competitive for the Galen use case.

Overall, the experimental evaluation has validated Claim II by showing a performance improvement compared to single strategy incremental evaluation approaches. The limited overhead of our Bootstrap evaluation makes up for any cost induced by the Update evaluation. We believe with improved heuristics and tuning, this improvement can be further maximized. On the other hand, however, these results also demonstrate that incremental evaluations are still outperformed by the standard non-incremental Soufflé overall. Therefore, there are still vast opportunities for improvement in incremental evaluation strategies.

Table 5.3: Memory usage for each engine, showing the minimum, average, and maximum memory usage across all of the update sets

Bench.	Engine	Min (MB)	Avg (MB)	Max (MB)
Doop	Soufflé	1,759	1,762	1,764
	Soufflé-elastic	7,473	7,492	7,505
	Soufflé-dense	9,106	9,449	11,387
	DDLog	17,381	23,352	27,851
CRDT	Soufflé	42	42	42
	Soufflé-elastic	335	346	352
	Soufflé-dense	328	337	344
	DDLog	786	829	858
Galen	Soufflé	901	931	960
	Soufflé-elastic	5,641	5,672	5,698
	Soufflé-dense	14,588	17,974	21,034
	DDLog	15,333	20,862	26,461

Along with runtime, another aspect of performance is memory usage. For example, in large program analysis use cases, memory has been shown to be a limiting factor [23]. Table 5.3 shows the minimum, average, and maximum memory usage across all the update sets for each benchmark. These results show that non-incremental Soufflé uses the least memory by far since it does not need to keep the extra state that incremental evaluation requires. Among the incremental engines, Soufflé-elastic performs best since it only keeps the counts for one iteration for each tuple. On the other hand, the counting algorithm, both in Soufflé and in DDLog, requires keeping the count of each tuple for *every* iteration it is generated in, thus using extra memory to maintain this additional state.

5.7 Chapter Summary

This chapter has demonstrated the pitfalls of existing incremental evaluation algorithms for use cases with varying sizes of updates. We have proposed the use of an elastic approach for incremental evaluation. This elastic approach switches between a low overhead Bootstrap strategy that targets high impact updates and an Update strategy that targets low impact

updates. We propose a simple heuristic for switching between the two strategies. Using this setup, we have shown that the elastic approach is effective in use cases where single strategy incremental evaluation struggles to perform adequately compared to regular Datalog evaluation.

Chapter 6

Input Debugging with Incremental Provenance

This chapter discusses the problem of *incremental fault localization and debugging*. As presented in Chapter 5, incremental evaluation is becoming more and more suitable for Datalog users to express incrementally updating computations. However, incremental evaluation can also cause anomalies introduced between updates, which can be difficult to debug with traditional debugging approaches. Therefore, this chapter introduces an approach that discovers the connection between output and input tuples to debug these anomalies.

This chapter is organized as follows. Section 6.1 introduces the problem of incremental debugging in the context of real-world Datalog use cases, while Section 6.2 provides background. Section 6.3 details our novel approach to providing provenance information for incremental computation, as a basis for incremental debugging. Section 6.4 discusses our algorithms for computing a fault localization or input debug suggestion, using incremental provenance and other techniques. Finally, Section 6.5 provides an experimental evaluation of our technique.

6.1 Fault Localization and Input Debugging

One important area of use cases for Datalog is in bug-finding and analysis tools [20, 124, 21, 22, 16, 125, 19]. In an industrial setting, these tools are deployed in a continuous integration setup to perform checks and validations after changes are made to a system [115, 126]. Changes between two analysis runs (aka. epochs) are frequently small and can be effectively processed by incremental evaluation strategies for Datalog that reuse computations of the previous run, as described in Chapter 5. In this setup, faults, which can manifest as missing or unwanted output tuples at the Datalog level, can appear and disappear between these changes, which can be challenging to diagnose with the complexity of modern bug-finding tools.

For instance, consider the example in Figure 6.1. The diagram shows the use of incremental evaluation for program analysis use cases. On the left, the source program is updated, resulting in a change ΔE, which is input to the incremental program analysis ΔP. After computing the incremental update, some result tuples are unchanged, some are inserted, and some are deleted.

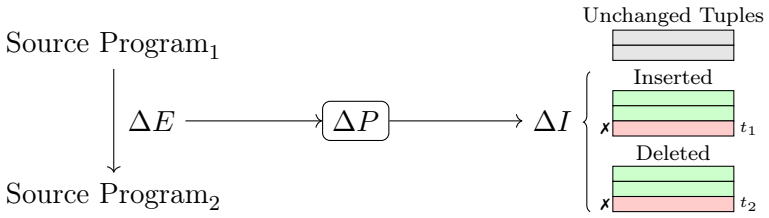


Figure 6.1: A scenario where an incremental update results in faults in the output

However, some of the changes (insertions or deletions) may be *unwanted* (i.e., the user does not agree with the change), and hence we can view these as *faults* that appeared as a result of the incremental update.

For the problem of incremental debugging, faults manifest in two possible scenarios: (1) unwanted tuples which appear, such as t_1 in the above diagram, or (2) desirable tuples which disappear, such as t_2. To investigate and debug these faults, a system should aid the user to discover the connection between the faulty output tuples and the input tuples changed as part of the incremental update. These concepts are an example of the framework of *why-provenance* (Section 3.1.1). Why-provenance formalizes the notion of *lineage*, which describes the relationship between input and output tuples.

Existing debugging approaches, such as graphs [127] and proof trees (Chapter 4), are designed and targeted towards the tool developer. Another example is *Algorithmic Debugging* [29], a general framework for debugging logic programs and diagnosing faults based on execution traces within the Datalog program. However, these explanations describe the execution of the Datalog rules and thus may be unintelligible for end-users who lack a deep understanding of a tool's implementation.

For an end-user of the tools, a debugging framework should explain the faults as localizations and debugging suggestions at the level of the input data rather than explaining failing rules in the Datalog program implementing the analysis. A natural candidate for such an approach is *delta debugging* [89, 90], a debugging framework for generalizing and simplifying a failing test case. It has recently been shown to scale well when integrated with state-of-the-art Datalog synthesizers [6] to obtain better synthesis constraints. Delta debugging uses a divide-and-conquer approach to localize the faults when changes are made to a program, thus providing a concise witness for the fault. However, the standard delta debugging approach is a brute-force search among the space of possible witnesses, while the well-defined mathematical semantics of Datalog enables much more efficient algorithms.

In this chapter, we introduce a debugging approach for incremental Datalog. We characterize this approach as *model-theoretic* debugging because, unlike previous approaches, our strategy does not explain why a tuple was or was not computed using proofs guided by rules. Instead, we use the connection between input and output tuples to propose a debugging suggestion to alter the incremental update such that the output satisfies some intended result. Our approach comprises a novel incremental provenance technique and two intertwined algorithms that diagnose and compute a debugging suggestion for a set of faults, i.e., missing and unwanted tuples.

The first algorithm performs *fault localization*, which diagnoses the cause of a set of faults. The second algorithm finds *debugging suggestions* which are subsets of the incremental change that can be removed to prevent the faults from appearing.

Our technique goes beyond delta debugging in that our approach takes advantage of our succinct incremental *provenance* information, which encodes a trace of the incremental Datalog execution. The incremental provenance information allows us to localize causes for any faulty tuples quickly. For debugging the faults, provenance allows our technique to narrow down the relevant parts of the execution trace quickly. We use an Integer Linear Program (ILP) [128] to encode the provenance information to compute the debugging suggestion. Therefore, instead of performing a global search in the space of the Datalog semantics, the incremental provenance information computed by the Datalog engine allows our technique to consider only the relevant portions of a particular execution. This results in significant performance improvements, allowing our approach to scale to real-world program analysis cases.

We have implemented our technique using an extended incremental version of the Soufflé [23, 2] Datalog engine and evaluated its effectiveness on DaCapo [129] Java program benchmarks analyzed by the Doop [20] static analysis library. Compared to delta debugging, we can localize and debug faults with a speedup of over 26.9× while providing smaller debugging suggestions in 27% of the benchmarks. To the best of our knowledge, we are the first to offer such a debugging feature in a Datalog engine, particularly for large workloads within a practical amount of time.

We summarize our contributions as follows:

- We propose a novel incremental provenance mechanism for Datalog engines. Our provenance technique allows for succinct proof trees to be constructed using incremental information.
- We propose a novel incremental localization and debugging technique for Datalog that scales to real-world program analyses. Our new debugging technique is a model-theoretic approach to explain input/output data faults rather than a proof-theoretic approach.
- We implement our technique in the state-of-the-art Datalog engine Soufflé, including extending incremental evaluation to also compute provenance.
- We evaluate our technique using the Doop static analysis, analyzing real-world Java programs.

6.2 Motivating Example

In this section, we use our running example to illustrate the problem of incremental input debugging. Recall, from Figure 2.2, our running example, which illustrates the use of Datalog for expressing a pointer analysis. In this scenario, incremental updates may be used to represent changes to the underlying source program. These incremental updates may cause the pointer analysis to compute unwanted alias relationships. Note that while the following example uses program analysis to showcase our incremental debugging approach, these techniques only target the Datalog portion of an overall program analysis framework. In general, program analysis

involves a series of abstractions, such that the input tuples for the Datalog program may not directly reflect the source program. Hence, while our techniques can debug faults appearing in the Datalog program, translating these results back to the source program is not straightforward. Of course, our technique can be applied to other Datalog applications, where such issues of abstraction may not be as prohibitive.

For instance, suppose that as part of an update to the input program in Figure 2.2a, we add a method to upgrade a user session to an admin session, containing the code

```
upgradedSession = userSession;
userSession = admin.session;
```

To update the result of the points-to analysis, we can perform an incremental update, where new tuples `assign(upgradedSession,userSession)` and `load(userSession,admin,session)` are inserted. After computing the incremental update, the new tuple `alias(userSession,sec)` is now contained in the output. However, we may wish to maintain an invariant that `userSession` *should not* be able to alias with the secure session `sec`. Therefore, the incremental update has introduced a *fault* into our program, and we wish to debug the appearance of the fault.

A fault localization is a subset of the incremental update such that the fault can be reproduced, and a debugging suggestion is a subset of the update such that the fault no longer appears when the debugging suggestion is excluded from the update. In this particular situation, the fault localization and debugging suggestion are identical, containing only the insertion of `load(userSession,admin,session)`. Notably, the other tuple in the update, the insertion of `assign(upgradedSession,userSession)`, is irrelevant for reproducing or fixing the fault and thus should not be included in an incremental debugging result. In general, however, an incremental update may contain hundreds or thousands of inserted and deleted tuples, and a set of faults may contain multiple tuples that are changed due to the incremental update. Moreover, the fault tuples may each have multiple alternative derivations, meaning that localization and debugging suggestions are not necessarily the same subset of the incremental change. In these situations, automatically localizing and debugging the faults to find a small relevant subset of the incremental update may be essential to provide a concise explanation of the faults to the user.

The scenario presented above is common during software development, where making changes to a program causes faults to appear. While our example concerns a points-to analysis computed for a source program, our incremental debugging techniques are in principle applicable for any Datalog program. Furthermore, they can even be applied to debug the Datalog program itself, where the update can be a set of Datalog *rules* that are inserted or deleted.

6.2.1 Delta Debugging

Delta debugging [89] is a general technique that finds a small subset out of a set of changes to any system, such that a bug can be reproduced in the small subset. It has been applied in many different areas and for numerous languages, such as diagnosing an HTML page causing a Mozilla browser to crash, investigating a GCC optimization causing a crash, and many other applications [90].

The basic delta debugging algorithm is presented in Section 3.3.1. While Algorithm 3 is presented in the context of strings (e.g., a program can be represented as a string), the basic concept can be easily adapted for an incremental Datalog update. For example, instead of considering a string of characters, the algorithm could process the set of insertions and deletions made during an incremental update.

In the context of an incremental Datalog evaluation, consider running the delta debugging algorithm where a fault tuple t appears after an incremental update ΔE. The result is a small subset $\delta E \subseteq \Delta E$ such that the fault tuple t is produced as a result of δE. This subset is *1-minimal*, i.e., if any one tuple was removed from δE, then the fault would no longer be reproduced. The main weakness of the delta debugging algorithm is that it views the Datalog engine as a black box. The algorithm specifies an EDB set in each iteration and evaluates the Datalog program to check if the fault tuple is contained in the resulting output. However, this process may take numerous iterations for large Datalog programs and updates, becoming prohibitive for debugging.

For instance, consider our running example. The full diff ΔE is comprised of two tuples `assign(upgradedSession,userSession)` and `load(userSession,admin,session)`. For the delta debugging algorithm, this diff is partitioned into two subsets, each containing one tuple. The algorithm then checks if each partition is buggy by evaluating the Datalog program with the added tuple to check if `alias(userSession,sec)` is produced. In this case, `load(userSession,admin,session)` indeed forms a buggy update, and so it is assigned to be ΔE. Since ΔE now contains only one tuple, the algorithm ends, and the resulting partition `load(userSession,admin,session)` is returned as the result. While this example is small and only requires one iteration, in general, delta debugging requires many iterations of executing the Datalog program, which can become very expensive.

6.3 Incremental Provenance

Recall, from Chapter 4, that provenance provides a mechanism to produce proof trees for tuples computed by the Datalog program. For example, the tuple `vpt(userSession,L3)` could be explained in our running example by the following proof tree:

$$\dfrac{\texttt{assign(userSession,ins)} \qquad \dfrac{\texttt{new(ins,L3)}}{\texttt{vpt(ins,L3)}}\, r_1}{\texttt{vpt(userSession,L3)}}\, r_2$$

Such a proof tree shows how the tuple is derived from the inputs using the rules in the program, and thus the programmer may explore the proof tree of an erroneous tuple to find the root cause of the fault. However, exploring the proof tree in this way requires deep knowledge of the Datalog program itself. Thus, provenance on its own is an excellent utility for the tool developer but is unsuitable for an end-user who is unfamiliar with the Datalog rules.

To allow for incremental input debugging of faults that appear after an incremental update, we extend provenance to explain the *appearance* of new tuples after the update. Thus, *incremental provenance* provides a mechanism to explain how a new tuple is computed from the deletions

and insertions in an incremental update. Incremental provenance provides these explanations by restricting the computed proof trees to only the portions affected by (i.e., containing tuples inserted or deleted as a result of) the incremental update. For example, Figure 6.2 shows an incremental proof tree for the inserted tuple `alias(userSession,sec)`. The tuples labeled with `(+)` indicate tuples that were inserted by an incremental update. Incremental provenance would only compute provenance information for these newly inserted tuples and would not explore the tuples in red color already established in a previous epoch.

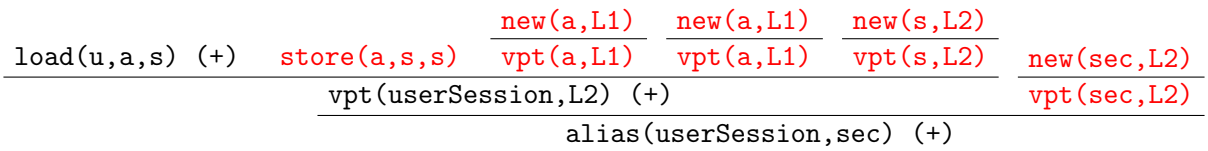


Figure 6.2: The proof tree for `alias(userSession,sec)`. The top two rows have shortened variable names. `(+)` denotes tuples that are inserted as a result of the incremental update, and red denotes tuples that were not affected by the incremental update.

While incremental provenance, as described above, computes subtrees of full proof trees, incremental debugging deals with incremental provenance as sets of tuples. Therefore, to formalize incremental provenance, we define inc-prov as follows. Given an incremental update ΔE, inc-prov$(t, \Delta E)$ should consist of tuples that were updated due to the incremental update.

Definition 6.3.1. *The set* inc-prov$(t, \Delta E)$ *is the set of tuples that appear in a proof tree for* t*, that are also inserted as a result of* ΔE*.*

Provenance with Incremental Annotations. Recall, from Chapter 4, that provenance information is computed using *provenance annotations*. The important annotation is that each tuple is associated with the height of its minimal height proof tree. Meanwhile, recall, from Chapter 5, that standard incremental evaluation strategies use *incremental annotations*; for example, each tuple is associated with an iteration number and a count for the number of derivations in that iteration.

To compute provenance information in an incremental evaluation setting, we observe a correspondence between the incremental annotations and provenance annotations. For the iteration number in incremental evaluation, a tuple is produced in some iteration i if at least one of the body tuples was produced in the previous iteration $i-1$. Therefore, the iteration number I for a tuple produced in a fixpoint is equivalent to

$$I(t) = \max\{I(t_1), \ldots, I(t_k)\} + 1 \text{ if } t :- t_1, \ldots, t_k$$

This definition of iteration number corresponds closely to the height annotation in provenance, with the only difference being that the iteration number is reset to zero in each stratum, while the height annotation is preserved across strata. Therefore, the iteration number annotation is suitable for constructing the same constraints that guide the proof tree construction. The rule

number annotation can be excluded if we extend the provenance query mechanism to search for *all* rules for a relation instead of the single rule given by the rule number.

For incremental debugging, it is important that the Datalog engine can produce provenance information *relevant* for faults that appear after an incremental update. Therefore, the provenance information produced by the Datalog engine should be restricted to tuples inserted or deleted by the incremental update. To achieve this, we adapt the user-driven proof tree exploration process presented in Chapter 4 to an automated procedure. This automated procedure only explores portions of the proof tree where the tuples are inserted or deleted due to the incremental update.

Recall from Chapter 4 that the proof tree construction is performed as a series of goal searches, which construct the proof tree level by level. For example, to discover the bottom level of the proof tree in Figure 6.2, we have the goal search following rule r_4:

$$?\text{:}-\texttt{vpt(Var1,Obj)}, \texttt{vpt(Var2,Obj)}, \texttt{Var1} \neq \texttt{Var2},$$
$$\texttt{Var1} = \texttt{userSession}, \texttt{Var2} = \texttt{sec}$$

The result of this goal search contains the tuples `vpt(userSession,L2)` and `vpt(sec,L2)`. For standard provenance, each of these tuples would create a corresponding goal search, thus constructing the next level of the proof tree. However, for incremental provenance, we restrict the next goal search to only compute provenance for tuples that are inserted by the incremental update. For this, we compute the intersection of the result with Δ IDB, the changes that resulted from the incremental update. Then, only tuples that exist in this intersection are considered in the next iteration of provenance construction.

As a result, our approach for incremental provenance produces proof trees containing only tuples that were inserted or deleted due to an update. For incremental debugging, this property is crucial for minimizing the search space. Note also that provenance also handles stratified negation by asserting that a negation holds true if the program does not compute the corresponding tuple. Therefore, an asserted negation can be treated as an input fact since negations are not further explained in the provenance framework.

6.4 Incremental Input Debugging

This section describes our approach and algorithms for the incremental fault localization and debugging problems. We begin by formalizing the problem and then presenting basic versions of both problems. Finally, we extend the algorithms to handle missing faults, negations, and user interaction.

To formalize incremental input debugging, we first define a fault. For a Datalog program, a fault may manifest in one of two situations: (1) an undesirable tuple that appears, or (2) a desirable tuple that disappears. In other words, a fault is a tuple that does not match the *intended semantics* of a program.

Definition 6.4.1 (Intended Semantics)**.** *The* intended semantics *of a Datalog program P is a pair of sets (I_+, I_-). We call I_+ a desirable tuple set and I_- an undesirable tuple set. An input*

set E is correct *w.r.t P and (I_+, I_-) if all desirable tuples are produced, i.e., $I_+ \subseteq P(E)$ and no undesirable tuples are produced, i.e., $I_- \cap P(E) = \emptyset$.*

Given an intended semantics for a program, a *fault* can be defined as follows:

Definition 6.4.2 (Fault)**.** *Let P be a Datalog program, with input set E and intended semantics (I_+, I_-). Assume that E is incorrect w.r.t P with (I_+, I_-). Then, a* fault *of E is a tuple t such that either t is desirable but missing, i.e., $t \in I_+ \setminus P(E)$ or t is undesirable but produced, i.e., $t \in P(E) \cap I_-$.*

Now, we can formalize the situation where an incremental update for a Datalog program introduces a fault. Let P be a Datalog program with intended semantics $I_\checkmark = (I_+, I_-)$ and let E_1 be an input EDB. Then, let $\Delta E_{1\to 2}$ be an incremental update, such that $E_1 \uplus \Delta E_{1\to 2}$ results in another input EDB, E_2. Then, assume that E_1 is correct w.r.t $I_\checkmark$, but E_2 is incorrect.

Fault Localization. The fault localization problem allows the user to pinpoint the sources of faults. This is achieved by providing a minimal subset of the incremental update that can still reproduce the fault. Formally,

Definition 6.4.3 (Fault Localization)**.** *A* fault localization *is a subset $\delta E \subseteq \Delta E_{1\to 2}$ such that $P(E_1 \uplus \delta E)$ exhibits* all *faults of E_2.*

Debugging Suggestion. A debugging suggestion provides a suggested 'fix' for a set of faults. The goal is to provide a *debugging suggestion*, which is a subset of the diff, where removing that subset would prevent all faults from appearing. Formally,

Definition 6.4.4 (Debugging Suggestion)**.** *A* debugging suggestion *is a minimal subset $\delta E_\times \subseteq \Delta E_{1\to 2}$ such that $P(E_1 \uplus (\Delta E_{1\to 2} \setminus \delta E_\times))$ does not produce any faults of E_2.*

Following the above definitions, the *fault localization problem* is to find a fault localization automatically, and similarly, the *incremental debugging problem* is to find a debugging suggestion automatically. Ideally, the fault localization or debugging suggestion that is found should be of minimal size to provide the most succinct explanation to the user. However, in practice, even non-optimal solutions can provide utility for the user, as shown in Section 6.5.

6.4.1 System Overview

The algorithms computing fault localizations and debugging suggestions depend on an *incremental evaluation* for Datalog, with an engine that produces *incremental provenance* information. The incremental debugging algorithms are computed after an incremental evaluation has already happened, and the algorithms use the diff information produced by the incremental evaluation along with the provenance.

Figure 6.3 shows a flow chart of how the system is used. The first portion of the system is the incremental Datalog evaluation. Here, the incremental evaluation takes an EDB and an incremental update containing tuples inserted or deleted from the EDB, denoted ΔEDB. The

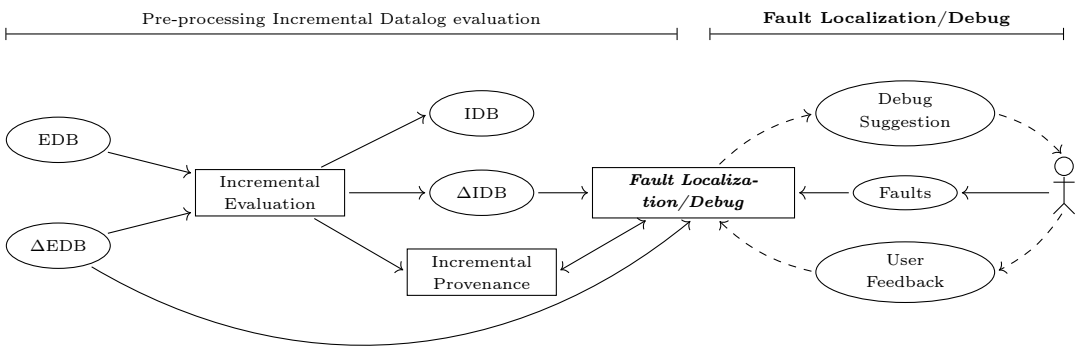


Figure 6.3: Incremental Debugging System

result of the incremental evaluation is the output IDB, along with the set of IDB tuples inserted or deleted as a result of the incremental update, denoted ΔIDB. The incremental evaluation also provides a provenance utility, which produces a proof tree for some given query tuple.

The second portion of the system is the fault localization/debugging. This process takes a set of faults provided by the user. This set of faults is a subset of ΔIDB, where each tuple is either unwanted and inserted in ΔIDB or is desirable but deleted in ΔIDB. Then, the fault localization/debugging algorithms use the full ΔIDB and ΔEDB and querying for provenance to produce a localization or debugging suggestion. The user can provide feedback in the form of rejecting a subset of the localization or debugging suggestion, in which case the system will incrementally reverse the rejected tuples to find a different result.

The main fault localization and debugging algorithms work together to provide localization or debugging suggestions to the user. The main idea of these algorithms is to compute proof trees for fault tuples, using the provenance utility provided by the incremental Datalog engine. These proof trees directly provide a localization for the faults. Then, to find debugging suggestions, the algorithms create an Integer Linear Programming (ILP) instance that encode the proof trees, with the goal of *disabling* all proof trees to prevent the fault tuples from appearing.

6.4.2 Fault Localization

In the context of incremental Datalog, the *fault localization problem* provides a small subset of the incremental changes that allow the fault to be reproduced. Consider the example in Figure 6.4. This diagram illustrates that a fault localization is a subset of the input changes $L \subseteq \Delta E$ such that when L is used as the input changes in the incremental evaluation, the resulting update still produces the faults.

We begin by first considering a basic version of the fault localization problem. In this basic version, we have a positive Datalog program (i.e., with no negation), and we localize a set of faults that are undesirable but appear (i.e., $P(E) \cap I_{-}$). The main idea of the fault localization algorithm is to compute a proof tree for each fault tuple. The tuples forming these proof trees are sufficient to localize the faults since the presence of these tuples allows the proof trees to be valid and thus the fault tuples to be reproduced.

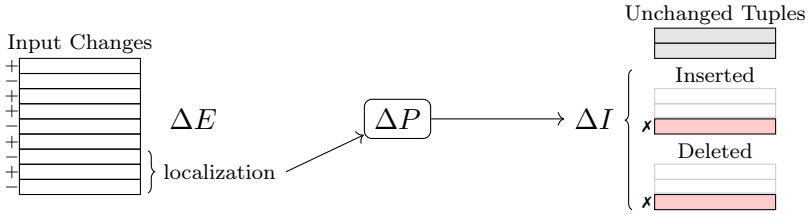


Figure 6.4: A fault localization is a subset of input changes such that the faults are still reproduced

Algorithm 9 Localize-Faults(P, E_2, $\Delta E_{1\to 2}$, F): Given a diff $\Delta E_{1\to 2}$ and a set of fault tuples F, returns a subset $\delta E \subseteq \Delta E_{1\to 2}$ such that $E_1 \uplus \delta E$ produces all $t \in F$

1: **for** tuple $t \in F$ **do**
2: Let inc-prov$(t, \Delta E_{1\to 2})$ be an incremental proof tree of t w.r.t P and E_2, containing tuples that were inserted due to $\Delta E_{1\to 2}$
3: **end for**
4: **return** $\cup_{t\in F}$(inc-prov$(t, \Delta E_{1\to 2}) \cap \Delta E_{1\to 2}$)

The basic fault localization is presented in Algorithm 9. For each fault tuple $t \in F$, the algorithm computes one incremental proof tree inc-prov$(t, \Delta E_{1\to 2})$. These proof trees contain the set of tuples that were inserted due to the incremental update $\Delta E_{1\to 2}$ that cause the existence of each fault tuple t. Therefore, by returning the union $\cup_{t\in F}$(inc-prov$(t, \Delta E_{1\to 2}) \cap \Delta E_{1\to 2}$), the algorithm produces a subset of $\Delta E_{1\to 2}$ that reproduces the faults.

6.4.3 Input Debugging Suggestion

An input debugging suggestion is a subset of the input changes such that the remaining changes 'fix' the faults. Consider Figure 6.5, which shows that a debug suggestion is a subset of the input changes, such that the remainder of the changes, when used as the incremental update, no longer produce the faults.

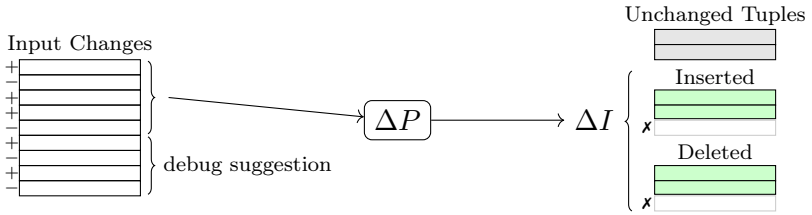


Figure 6.5: An input debugging suggestion is a subset of input changes such that the remainder of the input changes no longer produce the faults

As with fault localization, we begin with a basic version of the incremental debugging problem, where we have only a positive Datalog program and debug a set of unwanted fault tuples. The basic debugging suggestion algorithm involves computing *all* non-cyclic proof trees for each fault tuple and 'disabling' each of those proof trees as shown in Algorithm 10. If all proof trees

are invalid, then the fault tuple will no longer be computed by the resulting EDB.

Algorithm 10 Debug-Suggestion(P, E_2, $\Delta E_{1\to2}$, F): Given a diff $\Delta E_{1\to2}$ and a set of fault tuples F, return a subset $\delta E \subseteq \Delta E_{1\to2}$ such that $E_1 \uplus (\Delta E_{1\to2} \setminus \delta E)$ does not produce t_r

Let all-inc-prov$(t, \Delta E_{1\to2}) = \{T_1, \ldots, T_n\}$ be the total incremental provenance for a tuple t w.r.t P and E_2, where each T_i is a non-cyclic proof tree containing tuples inserted due to $\Delta E_{1\to2}$.
Construct an integer linear program instance as follows:
Create a 0/1 integer variable x_{t_k} for each tuple t_k that occurs in the proof trees in all-inc-prov$(t, \Delta E_{1\to2})$ for each fault tuple $t \in F$
for each tuple $t_f \in F$ **do**
    for each proof tree $T_i \in$ all-inc-prov$(t, \Delta E_{1\to2})$ **do**
        for each line $t_h \leftarrow t_1 \wedge \ldots \wedge t_k$ in T_i **do**
            Add a constraint $x_{t_1} + \ldots + x_{t_k} - x_{t_h} \leq k - 1$
        end for
    end for
    Add a constraint $x_{t_f} = 0$
end for
Add the objective function maximize $\sum_{t_e \in EDB} x_{t_e}$
Solve the ILP
Return $\{t \in \Delta E_{1\to2} \mid x_t = 0\}$

Algorithm 10 computes a minimum subset of the diff $\Delta E_{1\to2}$, which would prevent the production of each $t \in F$ when excluded from the diff. The key idea is to use integer linear programming (ILP) [128] as a vehicle to model the proof trees so that the solution of the ILP represents a debugging suggestion. We phrase the proof trees as a pseudo-Boolean formula [130] whose propositions represent tuples in the EDB and IDB. In the ILP, the faulty tuples are constrained to be false (i.e., have a value of zero), and the EDB tuples assuming true values are to be maximized, i.e., we wish to maximize the number of EDB tuples kept unchanged while producing a correct debugging suggestion.

The ILP created in Algorithm 10 introduces a variable for each tuple (either IDB or EDB) that appears in any of the incremental proof tree for the fault tuples. For the ILP model, we have three types of constraints: (1) to encode a single-step proof, (2) to enforce the false value for fault tuples, and (3) to ensure that variables are in the 0-1 domain.

The constraints for encoding proof trees are introduced for each one-step derivation:

$$\frac{t_1 \quad \ldots \quad t_k}{t_h}$$

that can be expressed as a Boolean constraint $t_1 \wedge \ldots \wedge t_k \implies t_h$ for the rule application where $t_1, \ldots, t_k$ and t_h are Boolean variables. Converting the Boolean constraints to disjunctive normal form, we obtain: $\overline{t_1} \vee \ldots \vee t_k \vee t_h$ by applying the definition of implication and De Morgan's laws. The implication is transformed to a pseudo (linear) Boolean formula:

$$\varphi\left(\overline{t_1} \vee \ldots \vee t_k \vee t_h\right) \equiv (1 - x_{t_1}) + \ldots + (1 - x_{t_k}) + t_h > 0$$

where φ maps a Boolean function to the $0-1$ domain and x_t corresponds to the 0-1 variable of proposition t in the ILP. The pseudo-Boolean constraint can be further simplified to $x_{t_1} + \ldots + x_{t_k} - x_{t_h} \leq k - 1$.

The constraints assuming false values for fault tuples $t_f \in F$ are simple equalities, i.e., $x_{t_f} = 0$. The objective function for the ILP is to maximize the number of inserted tuples that are kept, which is equivalent to minimizing the number of tuples in $\Delta E_{1\to 2}$ that are disabled by the debugging suggestion. In ILP form, this is expressed as maximizing $\sum_{t\in\Delta E_{1\to 2}} x_t$.

$$\begin{array}{lll} \text{max.} & \sum_{t\in\Delta E_{1\to 2}} x_t & \\ \text{s.t.} & x_{t_1} + \ldots x_{t_k} - x_{t_h} \leq k-1 & (\forall t_h \Leftarrow t_1 \wedge \ldots \wedge t_k \in T_i) \\ & x_{t_f} = 0 & (\forall t_f \in F) \\ & x_t \in \{0,1\} & (\forall \text{tuples } t) \end{array}$$

The solution of the ILP permits us to determine the EDB tuples for debugging:

$$\delta E = \{t \in \Delta E_{1\to 2} \mid x_t = 0\}$$

This is a minimal set of inserted tuples that must be removed from $\Delta E_{1\to 2}$ so that the fault tuples disappear.

This ILP formulation encodes the problem of disabling all proof trees for all fault tuples while maximizing the number of inserted tuples kept in the result. If there are multiple fault tuples, the algorithm simply computes proof trees for each fault tuple and combines all proof trees in the ILP encoding. The result of the algorithm is a set of tuples that is minimal but sufficient to disable the fault tuples from being produced.

6.4.4 Extensions

Missing Tuples. The basic versions of the fault localization and debugging problems only handle a tuple which is undesirable but appears. The opposite kind of fault, i.e., a tuple which is desirable but missing, can be debugged by considering a *dual* version of the problem.

For the dual version of the problem, we first observe that a tuple t that disappears after applying a diff $\Delta E_{1\to 2}$ would appear if we consider the update in the opposite direction, $\Delta E_{2\to 1}$. Then, the dual problem of localizing the disappearance of t is to find a *debugging suggestion* for the appearance of t after applying the opposite diff, $\Delta E_{2\to 1}$.

To localize a disappearing tuple t, we want to provide a small subset δE of $\Delta E_{1\to 2}$ such that t is still not computable after applying δE to E_1. To achieve this, *all* ways to derive t must be invalid after applying δE. Considering the dual problem, debugging the appearance of t in $\Delta E_{2\to 1}$ results in a subset δE such that $E_2 \uplus (\Delta E_{2\to 1} \setminus \delta E)$ does not produce t. Now, consider the reverse of δE (i.e., insertions become deletions and vice versa), which we denote δE^{-1}. If we apply δE^{-1} to E_1, then $E_1 \uplus \delta E^{-1} = E_2 \uplus (\Delta E_{2\to 1} \setminus \delta E)$, since $E_1 = E_2 \uplus \Delta E_{2\to 1}$. Therefore, δE^{-1} is the desired minimal subset that localizes the disappearance of t.

Similarly, to debug a disappearing tuple t, we apply the dual problem of localizing the appearance of t after applying the opposite diff $\Delta E_{1\to 2}$. Here, the debugging suggestion of a

disappearing tuple is to introduce *one* way of deriving t. Therefore, localizing the appearance of t in the opposite diff provides one derivation for t, and thus is the desired solution.

In summary, to localize or debug a tuple t that is missing after applying $\Delta E_{1\to 2}$, we compute a solution for the dual problem. The dual problem for localization is to debug the appearance of t after applying $\Delta E_{2\to 1}$, and the dual problem for debugging is to localize the appearance of t in $\Delta E_{2\to 1}$.

Stratified Negation. Consider the problem of localizing the appearance of an unwanted tuple t. If the Datalog program contains stratified negation, then the appearance of t can be caused by two possible situations. Either (1) there is a positive tuple in the proof tree of t that appears, or (2) there is a negated tuple in the proof tree of t that disappears. The first case is the standard case, but in the second case, if a negated tuple disappears, then its disappearance can be localized or debugged by computing the dual problem as in the missing tuple strategy presented above.

In executing the dual version of the problem, we may encounter further negated tuples. For example, consider the set of Datalog rules:

```
A(x) :- B(x), !C(x).
C(x) :- D(x), !E(x).
```

If we wish to localize an appearing but unwanted tuple `A(x)`, we may encounter a disappearing tuple `C(x)`. Then, executing the dual problem, we may encounter an appearing tuple `E(x)`.

In general, we can continue flipping between the dual problems to solve the localization or debugging problem. This process is guaranteed to terminate due to the stratification of negations. Each time the algorithm encounters a negated tuple, it must appear in an earlier stratum than the previous negation. Therefore, eventually, the negations reach the input EDB, and the process terminates.

User Interaction. An automatically produced fault localization or debugging suggestion may not be the user's intended result. For example, a localization may include an EDB tuple that the user decides is actually correct and should not be considered to be causing the fault. On the other hand, a debugging suggestion may include a tuple that the user does not wish to include. Therefore, our algorithms should allow the user to apply their domain knowledge during the debugging process.

The main idea for this user interaction hinges on the incremental evaluation in the Datalog engine. The user should be able to specify that some tuples in the localization or debugging suggestion should be excluded from the result. These specified tuples are incrementally *reversed* (a tuple that was originally inserted in the diff $\Delta E_{1\to 2}$ should be removed and vice versa), then the fault localization or debugging suggestion is computed again. The result will then not include the user-specified tuples.

Changes in Datalog Rules. The algorithms are presented above in the context of localizing or debugging a change to the input tuples. However, with a simple transformation, the same

algorithms can also be applied to changes in Datalog rules. For each Datalog rule, introduce a predicate `Rule(i)`, where `i` is a unique number per rule. Then, the unary relation `Rule` can be considered input, and thus the set of rules can be changed by providing a diff containing insertions or deletions into the `Rule` relation. For example, a transformed set of rules may be:

```
P(x, y) :- E(x, y), Rule(1).
P(x, z) :- E(x, y), P(y, z), Rule(2).
```

Then, by including or excluding 1 or 2 in the EDB relation `Rule`, the underlying Datalog rules can be 'switched on or off,' and a change to the Datalog program can be expressed as a diff in the `Rule` relation.

6.4.5 Full Algorithm

The full incremental debugging algorithm presented in Algorithm 11 incorporates the basic version of the problem, along with all of the extensions presented above. The algorithm begins by initializing the EDB after applying the diff (line 1) and separate sets of unwanted faults (lines 2) and missing faults (3). The set of candidate tuples forming the debugging suggestion is initialized to be empty (line 4).

The main part of the algorithm is a worklist loop (lines 5 to 15). In this loop, the algorithm first processes all unwanted but appearing faults (F^+, line 6), by computing a debugging suggestion for F^+. The result is a subset of tuples in the diff such that the faults F^+ no longer appear when the subset is excluded from the diff. In the provenance system, negations are treated as EDB tuples, and thus the resulting debugging suggestion may contain negated tuples. These negated tuples are added to F^- (line 7) since a tuple appearing in F^+ may be caused by a negated tuple disappearing. The algorithm then debugs the tuples in F^- by computing the dual problem, i.e., localizing F^- with respect to $\Delta E_{2\to1}$. Again, this process may result in negated tuples, which are added to F^+, and the loop begins again. This worklist loop must terminate, due to the semantics of stratified negation, as discussed above.

At the end of the worklist loop, L contains a candidate debugging suggestion. However, the user may choose to reject some of these tuples, and the algorithm should find a different debugging candidate. The user can provide a subset $U \subseteq L$ (line 18). Then, the algorithm is rerun, excluding U from the original diff.

While Algorithm 11 presents a full algorithm for computing a debugging suggestion, the fault localization problem can be solved in a similar way. Since debugging and fault localization are dual problems, the full fault localization algorithm simply switches Debug-Suggestion in line 6 and Localize-Faults in line 11.

Example. We demonstrate how these algorithms work through our running example. Recall the incremental update which inserts two tuples `assign(upgradedSession,userSession)` and `load(userSession,admin,session)`. As a result, the system computes the unwanted fault tuple `alias(userSession,sec)`. To localize or debug the appearance of the fault tuple, the algorithms start by computing its provenance, as shown in Figure 6.2. For localization, the algorithm

Algorithm 11 Incremental-Debugging(P, E_1, $\Delta E_{1\to 2}$, (I_+, I_-)): Given a diff $\Delta E_{1\to 2}$ and an intended semantics $(I_+, I_,)$, compute a subset $\delta E \subseteq \Delta E_{1\to 2}$ such that $\Delta E_{1\to 2} \setminus \delta E$ satisfies the intended semantics

1: Let E_2 be the EDB after applying the diff: $E_1 \uplus \Delta E_{1\to 2}$
2: Let F^+ be appearing unwanted faults: $\{I_- \cap P(E_2)\}$
3: Let F^- be missing desirable faults: $\{I_+ \setminus P(E_2)\}$
4: Let L be the set of tuples forming a debugging suggestion, initialized to $\emptyset$
5: **while** both F^+ and F^- are non-empty **do**
6: Add Debug-Suggestion(P, E_2, $\Delta E_{1\to 2}$, F^+) to L
7: **for** negated tuples $!t \in L$ **do**
8: Add t to F^-
9: **end for**
10: Clear F^+
11: Add Localize-Faults(P, E_1, $\Delta E_{2\to 1}$, F^-) to L
12: **for** negated tuples $!t \in L$ **do**
13: Add t to F^+
14: **end for**
15: Clear F^-
16: **end while**
17: **if** user rejects some tuples **then**
18: Let U be set of user-rejected tuples
19: **return** Full-Fault-Localization(P, E_1, $\Delta E_{1\to 2} \setminus U$, (I_+, I_-))
20: **end if**
21: **return** L

returns the updated EDB tuple in the proof tree, i.e., `load(userSession,admin,session)`. For debugging, the algorithm creates a set of ILP constraints, where each tuple (with shortened variables) represents an ILP variable:

$$\begin{aligned} \texttt{load}(\texttt{u},\texttt{a},\texttt{s}) - \texttt{vpt}(\texttt{u},\texttt{L2}) &\leq 0 \\ \texttt{vpt}(\texttt{u},\texttt{L2}) - \texttt{alias}(\texttt{u},\texttt{s}) &\leq 0 \\ \texttt{alias}(\texttt{u},\texttt{s}) &= 0 \\ \max \sum \texttt{load}(\texttt{u},\texttt{a},\texttt{s}) & \end{aligned}$$

For this simple ILP, the result is identical to the localization solution, which is that the tuple `load(userSession,admin,session)` should be excluded to fix the fault.

6.4.6 Correctness and Optimality

In this section, we discuss the correctness and optimality of our algorithms. Here, correctness means that the subset of the diff produced by localization correctly reproduces the faults, and a

debugging suggestion correctly prevents the faults from appearing. Optimality can be measured by the size of the subset given by the localization or debugging suggestion, where an optimal solution means that no smaller subset of the localization or debugging suggestion is a correct result.

Fault Localization. The correctness of fault localization (Algorithm 9) lies in the semantics of the proof trees. If every EDB tuple of the proof trees exist, then each proof tree would eventually be fully instantiated, and thus the fault tuples would be produced. The optimality of fault localization is dependent on the properties of the proof trees produced by the Datalog engine. If these proof trees are minimal in terms of the number of EDB tuples, then our fault localization algorithm would also be optimal.

Debugging Suggestion. The correctness and optimality of a debugging suggestion (Algorithm 10) depend directly on the properties of the ILP encoding. Since debugging must use *all* proof trees for fault tuples, the properties of each individual proof tree do not affect optimality. For debugging, the ILP constraints encode the implications in the proof trees and the property that we disable the fault tuples, and therefore the result is correct. The optimality of a debugging suggestion is a direct consequence of the optimization target in the ILP instance, and since the ILP minimizes the number of tuples included in the result, the solution is optimal.

Full Algorithm. Each component of the full algorithm is correct, as discussed above, and therefore it only remains to be shown that considering the dual problem for negations is correct. This correctness is discussed in Section 6.4.4, and thus the full algorithm identifies a correct debugging suggestion.

However, the full algorithm is not necessarily optimal in the presence of negation. For example, consider when an initial debugging suggestion includes a negated tuple, t. Then, the full algorithm computes the dual problem of localizing the appearance of t with the opposite diff $\Delta E_{2\to1}$. However, this opposite diff does not consider the initial debugging suggestion (only the negated tuple), and thus, the result may not be optimal. In practice, this sub-optimality rarely affects the solution, and the result is generally optimal or close to optimal.

6.5 Experiments

This section evaluates our technique on real-world benchmarks to determine its effectiveness and applicability. We aim to address the following three research questions:

- RQ1: Is the new technique faster than a delta-debugging strategy?
- RQ2: Does the new technique produce more precise localization/debugging candidates than delta debugging?
- RQ3: Does the new technique scale effectively to real-world use cases?

Table 6.1: Results for debugging size and runtime, our fault localization/debugging technique compared to delta debugging

		Debugging Suggestion Runtimes				Delta Debugging		
Benchmark	No.	Size	Overall (s)	Localize (s)	Debugging (s)	Size	Runtime (s)	Speedup (×)
antlr	1	2	73.6	0.51	73.1	3	3057.8	41.5
	2	1	79.4	0.00	79.4	1	596.5	7.5
	3	1	0.95	0.95	-	1	530.8	558.7
	4	2	77.8	1.89	75.9	3	3017.6	38.8
bloat	1	2	3309.5	0.02	3294.1	2	2858.6	0.9
	2	1	356.3	0.00	355.4	1	513.6	1.4
	3	1	0.33	0.33	-	1	557.7	1690.0
	4	3	3870.6	0.10	3854.7	2	2808.3	0.7
chart	1	1	192.6	0.00	192.6	1	685.0	3.6
	2	1	3.01	3.01	-	1	675.3	224.4
	3	1	78.8	0.00	78.8	1	667.6	8.5
	4	2	79.9	3.24	76.7	3	3001.1	37.6
eclipse	1	2	177.3	0.04	177.2	3	2591.2	14.6
	2	1	79.2	0.00	79.1	1	416.1	5.3
	3	1	0.12	0.12	-	1	506.3	4219.2
	4	3	91.9	0.09	91.8	3	2424.4	26.4
fop	1	2	83.8	0.05	83.8	2	3446.6	41.1
	2	1	76.9	0.00	76.9	1	670.7	8.7
	3	1	0.66	0.66	-	1	721.8	1093.6
	4	6	74.8	0.50	74.3	Timeout (7200)		96.3+
hsqldb	1	2	83.3	0.04	83.3	2	2979.2	35.8
	2	1	79.4	0.00	79.4	1	433.8	5.5
	3	1	74.0	0.00	74.0	1	663.1	9.0
	4	3	75.5	0.04	75.5	5	6134.8	81.3
jython	1	1	83.3	0.00	83.3	1	609.4	7.3
	2	1	78.2	0.00	78.2	1	590.4	7.5
	3	1	76.6	0.00	76.6	1	596.2	7.8
	4	1	75.8	0.00	75.8	1	587.6	7.8
luindex	1	2	81.3	0.07	81.2	3	2392.1	29.4
	2	1	79.8	0.00	79.8	1	511.0	6.4
	3	1	0.10	0.10	-	1	464.8	4648.0
	4	4	77.9	0.12	77.8	5	4570.4	58.7
lusearch	1	2	110.2	0.06	110.0	3	2558.8	23.2
	2	1	1062.1	0.00	1057.4	1	370.4	0.3
	3	1	0.12	0.12	-	1	369.6	3080.0
	4	2	294.2	0.06	293.2	3	2420.9	8.2
pmd	1	2	78.1	0.02	78.1	3	3069.8	39.3
	2	1	77.0	0.00	77.0	1	600.2	7.8
	3	1	0.08	0.08	-	1	717.8	8972.5
	4	3	74.3	0.08	74.2	3	2828.3	38.1
xalan	1	1	84.9	0.00	84.9	1	745.3	8.8
	2	1	82.2	0.00	82.2	1	728.9	8.9
	3	1	100.1	0.00	100.1	1	1243.7	12.4
	4	1	521.6	0.00	518.3	1	712.5	1.4

6.5.1 Experimental Setup

We implemented the incremental fault localization and debugging algorithms using Python. The Python code calls out to an incremental version of the Soufflé Datalog engine [2] extended with incremental provenance. Our implementation of incremental provenance uses the default metric of minimizing proof tree height, as it provides near-optimal solutions with slight runtime improvements. For solving integer linear programs, we use the GLPK library.

Our main point of comparison in our experimental evaluation is the delta debugging approach, such as that used in the ProSynth Datalog synthesis framework [6]. We adapted the implementation of delta debugging used in ProSynth to support input tuple updates. Like our fault debugging implementation, the delta debugging algorithm was implemented in Python; however, it calls out to the standard Soufflé engine since that provides a lower overhead than the incremental or provenance versions.

All of our experiments were run on a server with an Intel Xeon Gold 6130 CPU and 192 GB of memory. The Soufflé executables are compiled with GCC 10.3.1, and the fault localization/debugging and delta debugging implementations are executed with Python 3.8.10.

Use Case. For our benchmarks, we use the Doop program analysis framework [20] with the DaCapo set of Java benchmarks [131]. The analysis contains approx. 300 relations, 850 rules, and generates approx. 25 million tuples from an input size of 4-9 million tuples per DaCapo benchmark. For each of the DaCapo benchmarks, we selected an incremental update randomly containing 50 tuples to insert and 50 tuples to delete, which is representative of a typical commit. From the resulting IDB changes, we selected four different fault sets for each benchmark, each containing between 3 and 10 updated outputs, which represents an analysis error. Then, we debug each error with both our technique and the delta debugging technique. We consider two aspects of the debugging procedure for each benchmark: (1) the execution time required to compute a debugging suggestion, and (2) the precision of the debugging suggestion measured by the number of tuples. For delta debugging, we exclude the time for reading the result from Soufflé's output files.

6.5.2 Performance

The results of our experiments are shown in Table 6.1. Our incremental debugging technique exhibits much better performance overall compared to the delta debugging technique. We observe a geometric mean speedup of over 26.9× [1] compared to delta debugging. For delta debugging, the main cause of performance slowdown is that it is a black-box search technique, and it requires multiple iterations of invoking Soufflé (between 6 and 19 invocations for the presented benchmarks) to direct the search. Since each invocation of Soufflé takes between 30-50 seconds, depending on the benchmark and EDB, the overall runtime for delta debugging is in the hundreds of seconds at a minimum. Indeed, we observe that delta debugging takes between 370 and 6135 seconds on our benchmarks, with one instance timing out after two hours (7200 seconds).

[1] We say "over" because we bound timeouts to 7200 seconds.

On the other hand, our incremental debugging technique simply calls for provenance information from an already initialized instance of incremental Soufflé. For eight of the benchmarks, the faults only contained missing tuples. Therefore, only the Localize-Faults method was called, which only computes one proof tree for each fault tuple and does not require any ILP solving. The remainder of the benchmarks called the Debug-Suggestion method, and the main bottleneck there is to construct and solve the ILP instance. For three of the benchmarks, `bloat-1`, `bloat-4`, and `lusearch-2`, the runtime was slower than delta debugging. This result is due to the fault tuples in these benchmarks having many different proof trees, which took longer to compute and led to a larger ILP instance, which took longer to solve.

6.5.3 Quality

While the delta debugging technique produces 1-minimal results, we observe that despite no overall optimality guarantees, our approach could produce more minimal debugging suggestions in 27% of the benchmarks. Moreover, our technique produced a larger debugging suggestion in only one of the benchmarks. This difference in quality is due to the choices made during delta debugging. Since delta debugging has no view of the internals of Datalog execution, it can only partition the EDB tuples randomly. Then, the choices made by delta debugging may lead to a locally minimal result that is not globally optimal.

For our fault localization technique, most of the benchmarks computed one iteration of debugging and did not encounter any negations. Therefore, the results were optimal in these situations due to the ILP formulation. Despite our technique overall not necessarily being optimal, it still produces high-quality results in practice.

6.5.4 Overall Scalability

Our technique is able to find all debugging suggestions in 4.6 minutes on average. 86% of benchmarks can be debugged in under 3 minutes compared 0% using delta debugging. Moreover, 93% of the benchmarks can be debugged in under 10 minutes compared to 32% with delta debugging. We encountered only two outlier cases requiring approx. 1 hour, where delta debugging performs slightly better in those cases. Overall, our technique exhibits a significant improvement compared to delta debugging, allowing users to obtain debugging suggestions much more efficiently than with delta debugging. Moreover, our debugging suggestions are smaller than the results of delta debugging, and the user can provide feedback and compute a different result well within a reasonable timeframe.

6.6 Chapter Summary

This chapter presents a new debugging technique that localizes faults and provides debugging suggestions for incrementally updated Datalog program inputs. Unlike previous approaches, our technique does not entirely rely on a black-box solver to perform the debugging, instead utilizing incremental provenance information obtained from the Datalog computation. Our technique exhibits speedups of 26.9× compared to delta debugging and finds more minimal

debugging suggestions 27% of the time. We can debug 93% of the faults in our program analysis benchmarks in under 10 minutes, providing high-performance debugging for users.

Chapter 7

Conclusion

The Datalog programming language has seen a recent trend towards more widespread adoption, both in research and industry. This increased popularity has largely been driven by the advent of new applications, including graph analyses, networking, and program analysis. To support these new applications, modern engines, such as Soufflé, have allowed for high-performance evaluation of Datalog programs while maintaining the ease of use resulting from Datalog's clean declarative semantics. However, tooling and infrastructure support, which would be of huge benefit for these modern large-scale applications, is still in its infancy for Datalog and logic programming. This thesis partly fills the tooling gaps, aiming to improve the productivity of Datalog programmers. In particular, this thesis focuses on debugging and incremental evaluation.

As presented in Chapter 4, the provenance framework provides a critical tool for Datalog programmers to discover and debug faults that may occur in Datalog rules. The provenance framework allows a Datalog program to generate proof trees, which trace the execution from inputs, through the rules, to the outputs. A programmer can use this provenance utility to investigate faulty output tuples, to discover potentially faulty rules. While designing this provenance framework, we develop a novel encoding for provenance information in proof annotations. These proof annotations are used by a subsequent proof construction phase, which allows the user to query for proof trees for any computed tuple. The result was a practical framework, with overheads of only 1.31× compared to standard Datalog evaluation.

The second area which we tackle in this thesis is incremental evaluation. For this, we present our *elastic* incremental evaluation in Chapter 5. Our elastic incremental evaluation strategy can out-perform previous state-of-the-art incremental evaluation approaches on workloads that include both small and large-sized updates. However, we also discover some pitfalls in the performance of incremental evaluation algorithms in general, where a standard batch-mode evaluation can exhibit superior performance.

Lastly, we address the opportunity to provide automated debugging solutions in Datalog. With incremental evaluation algorithms, Datalog programs can exhibit faults that arise between incremental updates. Furthermore, proof trees as computed by provenance may not be understandable for users since they require a deep understanding of the Datalog rules. Thus, Chapter 6 presents our solution, which uses incremental evaluation, provenance, and integer linear programming to automatically localize faulty input tuples that cause faulty outputs. This

technique can even suggest debugging fixes for these input tuples, such that the faulty outputs no longer appear.

In combination, the above contributions showcase the vast opportunities in enhancing the usability of logic programming languages. This thesis tackles the important problems of debugging and incremental evaluation, with the aim of making our solutions practical and usable.

7.1 Future Work

This thesis presents several approaches to address tooling gaps that exist in Datalog. However, there are many promising avenues for future research in this direction. These future research topics may include expanding the utility and applicability of our techniques and improving their performance.

Firstly, there remain several possible improvements for the provenance system. One important aspect currently lacking in the provenance framework is to support advanced language features such as aggregation and functors. In principle, these features can be supported; however, the encoding and semantics of the provenance system would require novel extensions. Additionally, there is also a body of work implementing *subsumption* in Datalog [132], allowing general lattices to be expressed. Provenance can then be implemented under the subsumption framework, which would allow for more generic provenance annotations. To utilize this, we could develop a user-facing language in which proof tree queries and patterns can be expressed, aiding the programmer to find more relevant proof trees.

Secondly, the provenance framework can provide utility beyond debugging. One example is our use of provenance information to provide automated debugging suggestions, as presented in Chapter 6. However, even further applications are possible. For example, one important research area in the past few years has been *explainability*, particularly concerning AI systems [133, 134]. For the Datalog language, provenance provides an ideal candidate for building explanation systems, where the reasons for specific outputs are made more transparent by their proof trees. Another research area is in ranking database or computation outputs based on quality or relevance [79, 135]. Our provenance framework can also be adapted to provide better measurements for such metrics, for example, by keeping track of scores instead of proof tree heights.

Finally, the pursuit of faster Datalog evaluation strategies is an ongoing one. Techniques such as worst-case optimal joins (WCOJ) [119, 120] have been developed in the literature but have yet to find a foothold in practical systems. These techniques have the potential to asymptotically speed up Datalog computations with a large number of joins, which is especially critical for our incremental evaluation framework, as discussed in Section 5.5.2. Since incremental evaluation results in many different versions of each original Datalog rule, each with several joins, techniques such as WCOJ can provide huge potential speedups. Even in the absence of WCOJ, query optimization techniques that are well-established in the database community can also be of potential interest. Moreover, there may be further opportunities for optimizing incremental evaluation by using techniques such as user-defined functors.

Bibliography

[1] Xiaowen Hu et al. "The Choice Construct in the Soufflé Language". In: *Asian Symposium on Programming Languages and Systems*. Springer. 2021, pp. 163–181.

[2] David Zhao et al. "Towards Elastic Incrementalization for Datalog". In: *23rd International Symposium on Principles and Practice of Declarative Programming*. 2021, pp. 1–16.

[3] Xiaowen Hu et al. "An efficient interpreter for Datalog by de-specializing relations". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 681–695.

[4] David Zhao, Pavle Subotić, and Bernhard Scholz. "Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42.2 (2020), pp. 1–35.

[5] Herbert Jordan et al. "Specializing parallel data structures for Datalog". In: *Concurrency and Computation: Practice and Experience* (2020), e5643.

[6] Mukund Raghothaman et al. "Provenance-guided synthesis of Datalog programs". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–27.

[7] Patrick Nappa et al. "Fast parallel equivalence relations in a datalog compiler". In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2019, pp. 82–96.

[8] Herbert Jordan et al. "A Specialized B-tree for Concurrent Datalog Evaluation". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: ACM, 2019, pp. 327–339. ISBN: 978-1-4503-6225-2. DOI: 10.1145/3293883.3295719. URL: http://doi.acm.org/10.1145/3293883.3295719.

[9] Herbert Jordan et al. "Brie: A specialized trie for concurrent datalog". In: *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*. 2019, pp. 31–40.

[10] David Zhao et al. "Scalable Repair of Input Faults in Datalog Using Incrementalized Provenance". In: (Under Submission). 2022.

[11] David Maier et al. "Datalog: concepts, history, and outlook". In: *Declarative Logic Programming: Theory, Systems, and Applications*. 2018, pp. 3–100.

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

[13] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. Springer Science & Business Media, 2012.

[14] James R Groff, Paul N Weinberg, and Andrew J Oppel. *SQL: the complete reference*. Vol. 2. McGraw-Hill/Osborne, 2002.

[15] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. "A general datalog-based framework for tractable query answering over ontologies". In: *Journal of Web Semantics* 14 (2012), pp. 57–83.

[16] Wenchao Zhou et al. "Efficient Querying and Maintenance of Network Provenance at Internet-Scale". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), pp. 615–626.

[17] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. "MulVAL: A Logic-based Network Security Analyzer". In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM'05. Baltimore, MD: USENIX Association, 2005, pp. 8–8. URL: http://dl.acm.org/citation.cfm?id=1251398.1251406.

[18] M. Liu et al. "Recursive Computation of Regions and Connectivity in Networks". In: *2009 IEEE 25th International Conference on Data Engineering*. 2009, pp. 1108–1119. DOI: 10.1109/ICDE.2009.36.

[19] John Backes et al. "Reachability Analysis for AWS-Based Networks". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. 2019, pp. 231–241.

[20] Martin Bravenboer and Yannis Smaragdakis. "Strictly Declarative Specification of Sophisticated Points-to Analyses". In: *SIGPLAN Not.* 44.10 (2009), pp. 243–262. ISSN: 0362-1340. DOI: 10.1145/1639949.1640108. URL: http://doi.acm.org/10.1145/1639949.1640108.

[21] Neville Grech et al. "MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts". In: *SPLASH 2018 OOPSLA*. 2018.

[22] Neville Grech et al. "Gigahorse: Thorough, Declarative Decompilation of Smart Contracts". In: *Proceedings of the 41th International Conference on Software Engineering, ICSE 2019*. Ed. by Joanne M. Atlee, Tevfik Bultan, and Jon Whittle. Montreal, QC, Canada: ACM, 2019, (to appear). DOI: 10.1109/ICSE.2019.00120. URL: https://doi.org/10.1109/ICSE.2019.00120.

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

[24] Molham Aref et al. "Design and Implementation of the LogicBlox System". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1371–1382. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742796. URL: http://doi.acm.org/10.1145/2723372.2742796.

[25] Antonio Flores-Montoya and Eric Schulte. "Datalog Disassembly". In: *arXiv preprint arXiv:1906.03969* (2019).

[26] Andre Meyer et al. "Today was a good day: The daily life of software developers". In: *IEEE Transactions on Software Engineering* (2019).

[27] Richard Stallman, Roland Pesch, Stan Shebs, et al. "Debugging with GDB". In: *Free Software Foundation* 675 (1988).

[28] J Wiegand et al. "Eclipse: A platform for integrating development tools". In: *IBM Systems Journal* 43.2 (2004), pp. 371–383.

[29] Rafael Caballero, Adrián Riesco, and Josep Silva. "A survey of algorithmic debugging". In: *ACM Computing Surveys (CSUR)* 50.4 (2017), p. 60.

[30] Wentao Han et al. "Chronos: a graph engine for temporal graph analysis". In: *Proceedings of the Ninth European Conference on Computer Systems*. 2014, pp. 1–14.

[31] Stefano Ceri and Jennifer Widom. "Deriving Production Rules for Incremental View Maintenance". In: *Proceedings of the 17th International Conference on Very Large Data Bases*. VLDB '91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 577–589. ISBN: 1-55860-150-3. URL: http://dl.acm.org/citation.cfm?id=645917.672169.

[32] Josep Silva. "A Comparative Study of Algorithmic Debugging Strategies". In: *Logic-Based Program Synthesis and Transformation*. Ed. by Germán Puebla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 143–159. ISBN: 978-3-540-71410-1.

[33] Roger Hoover. "Alphonse: Incremental computation as a programming abstraction". In: *ACM SIGPLAN Notices* 27.7 (1992), pp. 261–272.

[34] Mordechai Ben-Ari. *Mathematical logic for computer science*. Springer Science & Business Media, 2012.

[35] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. "Soufflé: On Synthesis of Program Analyzers". In: *Proceedings of Computer Aided Verification* 28 (2016), pp. 422–430.

[36] Leon Sterling and Ehud Y Shapiro. *The art of Prolog: advanced programming techniques*. MIT press, 1994.

[37] Jiwon Seo, Stephen Guo, and Monica S Lam. "Socialite: Datalog extensions for efficient social network analysis". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 278–289.

[38] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[39] Maarten H Van Emden and Robert A Kowalski. "The semantics of predicate logic as a programming language". In: *Journal of the ACM (JACM)* 23.4 (1976), pp. 733–742.

[40] Anders Møller and Michael I Schwartzbach. "Static program analysis". In: *Notes. Feb* (2012).

[41] Alan Mathison Turing et al. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936), p. 5.

[42] Patrick Cousot. "Abstract interpretation". In: *ACM Computing Surveys (CSUR)* 28.2 (1996), pp. 324–328.

[43] Konstantinos Sagonas, Terrance Swift, and David S. Warren. "XSB: An Overview of its Use and Implementation". In: *SUNY Stony Brook* (1993), pp. 11794–4400.

[44] Sergio Greco and Cristian Molinaro. *Datalog and Logic Databases*. Morgan & Claypool Publishers, 2015.

[45] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285–309. URL: https://projecteuclid.org:443/euclid.pjm/1103044538.

[46] John Whaley et al. "Using Datalog with Binary Decision Diagrams for Program Analysis". In: *Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005. Proceedings*. Ed. by Kwangkeun Yi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 97–118. ISBN: 978-3-540-32247-4. DOI: 10.1007/11575467_8. URL: https://doi.org/10.1007/11575467_8.

[47] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. "Maintaining Views Incrementally". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. SIGMOD '93. Washington, D.C., USA: ACM, 1993, pp. 157–166. ISBN: 0-89791-592-5. DOI: 10.1145/170035.170066. URL: http://doi.acm.org/10.1145/170035.170066.

[48] Ashish Gupta and Inderpal Singh Mumick. *Maintenance of Materialized Views: Problems, Techniques, and Applications*. 1995.

[49] Boris Motik et al. "Maintenance of datalog materialisations revisited". In: *Artificial Intelligence* 269 (2019), pp. 76–136.

[50] Frank McSherry et al. "Differential Dataflow". In: *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. 2013. URL: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf.

[51] Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura. "μZ– An Efficient Engine for Fixed Points with Constraints". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 457–462. ISBN: 978-3-642-22110-1.

[52] Leonid Ryzhyk and Mihai Budiu. "Differential Datalog." In: *Datalog* 2 (2019), pp. 4–5.

[53] Raghu Ramakrishnan et al. “Implementation of the CORAL deductive database system”. In: *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 1993, pp. 167–176.

[54] Pavle Subotić et al. “Automatic Index Selection for Large-Scale Datalog Computation”. In: *PVLDB* 12.2 (2018), pp. 141–153.

[55] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. “Provenance in Databases: Why, How, and Where”. In: *Foundations and Trends in Databases* 1 (2009), pp. 379–474. DOI: 10.1561/1900000006.

[56] Daniel Deutch, Amir Gilad, and Yuval Moskovitch. “Selective Provenance for Datalog Programs Using Top-K Queries”. In: *Proceedings of the VLDB Endowment* 8 (2015), pp. 1394–1405.

[57] Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. “Declarative Datalog Debugging for Mere Mortals”. In: *Lecture Notes in Computer Science* 7494 (2012), pp. 111–122.

[58] Tarun Arora et al. “Explaining Program Execution in Deductive Systems”. In: *Proceedings of Deductive and Object-Oriented Databases: Third International Conference* (1993), pp. 101–119.

[59] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. “Why and Where: A Characterization of Data Provenance”. In: *Proceedings of the International Conference on Database Theory* 1973 (2001), pp. 316–330.

[60] Eugene Wu, Samuel Madden, and Michael Stonebraker. “Subzero: a fine-grained lineage system for scientific databases”. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 865–876.

[61] Tom Oinn et al. “Taverna: lessons in creating a workflow environment for the life sciences”. In: *Concurrency and computation: Practice and experience* 18.10 (2006), pp. 1067–1100.

[62] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. “Provenance Semirings”. In: *Proceedings of the ACM Symposium on Principles of Database Systems* (2007), pp. 675–686.

[63] Dan Suciu et al. “Probabilistic databases”. In: *Synthesis lectures on data management* 3.2 (2011), pp. 1–180.

[64] Omar Benjelloun et al. “An Introduction to ULDBs and the Trio System”. In: *IEEE Data Engineering Bulletin* (2006).

[65] Todd J. Green et al. “Update Exchange with Mappings and Provenance”. In: *In Very Large Data Bases (VLDB)*. 2007, pp. 675–686.

[66] Michael Curtiss et al. “Unicorn: A system for searching the social graph”. In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1150–1161.

[67] Deepavali Bhagwat et al. "An Annotation Management System for Relational Databases". In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 900–911. ISBN: 0-12-088469-0. URL: http://dl.acm.org/citation.cfm?id=1316689.1316767.

[68] Pierre Senellart et al. "Provsql: Provenance and probability management in postgresql". In: *Proceedings of the VLDB Endowment (PVLDB)* 11.12 (2018), pp. 2034–2037.

[69] Daniel Deutch et al. "Circuits for Datalog Provenance". In: *Conference on Database Theory* 17 (2014), pp. 201–212. DOI: 10.5441/002/icdt.2014.22.

[70] Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. "Debugging of wrong and missing answers for Datalog programs with constraint handling rules". In: July 2015. DOI: 10.1145/2790449.2790522.

[71] Seokki Lee, Bertram Ludäscher, and Boris Glavic. "PUG: a framework and practical implementation for why and why-not provenance". In: *The VLDB Journal* 28.1 (2019), pp. 47–71.

[72] Georg Lausen, Bertram Luascher, and Wolfgang May. "On Active Deductive Databases: The Statelog Approach". In: *Lecture Notes in Computer Science* 1472 (1998), pp. 69–106.

[73] Lee Naish. *A declarative debugging scheme*. Citeseer, 1995.

[74] Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. "A theoretical framework for the declarative debugging of datalog programs". In: *International Workshop on Semantics in Data and Knowledge Bases*. Springer. 2008, pp. 143–159.

[75] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. "Thin slicing". In: *ACM SIGPLAN Notices*. Vol. 42. 6. ACM. 2007, pp. 112–122.

[76] Andrew Ko and Brad Myers. "Debugging reinvented". In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE. 2008, pp. 301–310.

[77] Xin Zhang et al. "Effective Interactive Resolution of Static Analysis Alarms". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 57:1–57:30. ISSN: 2475-1421. DOI: 10.1145/3133881. URL: http://doi.acm.org/10.1145/3133881.

[78] Ravi Mangal et al. "A User-guided Approach to Program Analysis". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 462–473. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786851. URL: http://doi.acm.org/10.1145/2786805.2786851.

[79] Mukund Raghothaman et al. "User-guided program reasoning using Bayesian inference". In: June 2018, pp. 722–735. DOI: 10.1145/3192366.3192417.

[80] Xin Zhang et al. "On Abstraction Refinement for Program Analyses in Datalog". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 239–248. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594327. URL: http://doi.acm.org/10.1145/2594291.2594327.

[81] Boris Motik et al. “Incremental update of datalog materialisation: the backward/forward algorithm”. In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

[82] Pan Hu, Boris Motik, and Ian Horrocks. “Optimised maintenance of datalog materialisations”. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[83] Martín Abadi, Frank McSherry, and Gordon D. Plotkin. “Foundations of Differential Dataflow”. In: *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 71–83. DOI: 10.1007/978-3-662-46678-0_5. URL: https://doi.org/10.1007/978-3-662-46678-0_5.

[84] Derek Gordon Murray et al. “Naiad: a timely dataflow system”. In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP ’13, Farmington, PA, USA, November 3-6, 2013*. 2013, pp. 439–455. DOI: 10.1145/2517349.2522738. URL: http://doi.acm.org/10.1145/2517349.2522738.

[85] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. “Incremental whole-program analysis in Datalog with lattices”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 1–15.

[86] Jose A. Blakeley et al. “Efficiently Updating Materialized Views”. In: 1986, pp. 61–71.

[87] Eric Hanson. “Efficient Support for Rules and Derived Objects in Relational Database Systems”. PhD thesis. University of California, Berkeley, 1987.

[88] A. Rosenthal et al. “Situation Monitoring for Active Databases”. In: *Proceedings of the 15th International Conference on Very Large Data Bases*. VLDB ’89. Amsterdam, The Netherlands: Morgan Kaufmann Publishers Inc., 1989, pp. 455–464. ISBN: 1-55860-101-5. URL: http://dl.acm.org/citation.cfm?id=88830.88915.

[89] Andreas Zeller. “Yesterday, my program worked. Today, it does not. Why?” In: *ACM SIGSOFT Software engineering notes* 24.6 (1999), pp. 253–267.

[90] Andreas Zeller and Ralf Hildebrandt. “Simplifying and isolating failure-inducing input”. In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200.

[91] Purdue University Department of Computer Science. *Delta Debugging*. Accessed: 08-12-2021. 2016. URL: https://www.cs.purdue.edu/homes/suresh/408-Spring2017/Lecture-9.pdf.

[92] Ghassan Misherghi and Zhendong Su. “HDD: Hierarchical delta debugging”. In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 142–151.

[93] Cyrille Artho. “Iterative delta debugging”. In: *International Journal on Software Tools for Technology Transfer* 13.3 (2011), pp. 223–246.

[94] Renáta Hodován and Ákos Kiss. “Modernizing hierarchical delta debugging”. In: *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. 2016, pp. 31–37.

[95] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. "Coarse hierarchical delta debugging". In: *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2017, pp. 194–203.

[96] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. "HDDr: a recursive variant of the hierarchical delta debugging algorithm". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 2018, pp. 16–22.

[97] A. C. Kakas, R. A. Kowalski, and F. Toni. *Abductive Logic Programming*. 1993.

[98] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. "Justifications for logic programming". In: *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer. 2013, pp. 530–542.

[99] Xue Li, Alan Bundy, and Alan Smaill. "ABC Repair System for Datalog-like Theories." In: *KEOD*. 2018, pp. 333–340.

[100] J Balsa, V Dahl, and JG Pereira Lopes. "Datalog grammars for abductive syntactic error diagnosis and repair". In: *Proc. Natural Language Understanding and Logic Programming Workshop*. Citeseer. 1995.

[101] Loreto Bravo and Leopoldo E. Bertossi. "Consistent query answering under inclusion dependencies". In: *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, October 5-7, 2004, Markham, Ontario, Canada*. Ed. by Hanan Lutfiyya, Janice Singer, and Darlene A. Stewart. IBM, 2004, pp. 202–216.

[102] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. "Answer sets for consistent query answering in inconsistent databases". In: *Theory Pract. Log. Program.* 3.4-5 (2003), pp. 393–424.

[103] Ofer Arieli et al. "Database repair by signed formulae". In: *International Symposium on Foundations of Information and Knowledge Systems*. Springer. 2004, pp. 14–30.

[104] Xujie Si et al. "Syntax-guided synthesis of datalog programs". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 515–527.

[105] Xujie Si et al. "Synthesizing datalog programs using numerical relaxation". In: *arXiv preprint arXiv:1906.00163* (2019).

[106] Stephen Muggleton. "Inductive logic programming". In: *New generation computing* 8.4 (1991), pp. 295–318.

[107] Stephen Muggleton and Luc De Raedt. "Inductive logic programming: Theory and methods". In: *The Journal of Logic Programming* 19 (1994), pp. 629–679.

[108] Luc De Raedt. *Logical and relational learning*. Springer Science & Business Media, 2008.

[109] Stephen H Muggleton et al. "Meta-interpretive learning: application to grammatical inference". In: *Machine learning* 94.1 (2014), pp. 25–49.

[110] Seokki Lee, Bertram Ludäscher, and Boris Glavic. "Provenance Summaries for Answers and Non-answers". In: *Proc. VLDB Endow.* 11.12 (Aug. 2018), pp. 1954–1957. ISSN: 2150-8097. DOI: 10.14778/3229863.3236233. URL: https://doi.org/10.14778/3229863.3236233.

[111] David Zhao. "Honours Thesis: Large-Scale Provenance for Soufflé". In: (2017).

[112] Daniel Deutch, Amir Gilad, and Yuval Moskovitch. "Efficient provenance tracking for datalog using top-k queries". In: *The VLDB Journal* 27.2 (2018), pp. 245–269. ISSN: 0949-877X. DOI: 10.1007/s00778-018-0496-7. URL: https://doi.org/10.1007/s00778-018-0496-7.

[113] Derek Murray et al. "Naiad: A Timely Dataflow System". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2013.

[114] Grigoris Karvounarakis et al. "Collaborative Data Sharing via Update Exchange and Provenance". In: *ACM Trans. Database Syst.* 38.3 (Sept. 2013). ISSN: 0362-5915.

[115] Dino Distefano et al. "Scaling Static Analyses at Facebook". In: *Commun. ACM* 62.8 (July 2019), 62–70. ISSN: 0001-0782.

[116] Pavle Subotić. "Concise Explanations in Static Analysis Driven Code Reviews". Infer Practitioners 2020. 2020. URL: https://www.youtube.com/watch?v=FPCZ2TIxrpg&t=8888s.

[117] Nathan Chong et al. "Code-Level Model Checking in the Software Development Workflow". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '20. Seoul, South Korea: Association for Computing Machinery, 2020, 11–20. ISBN: 9781450371230. DOI: 10.1145/3377813.3381347. URL: https://doi.org/10.1145/3377813.3381347.

[118] Pan Hu, Boris Motik, and Ian Horrocks. "Modular Materialisation of Datalog Programs". In: (2019).

[119] Todd L Veldhuizen. "Leapfrog triejoin: A simple, worst-case optimal join algorithm". In: *arXiv preprint arXiv:1210.0481* (2012).

[120] Hung Q Ngo et al. "Worst-case optimal join algorithms". In: *Journal of the ACM (JACM)* 65.3 (2018), pp. 1–40.

[121] Alexander Shkapsky et al. "Big data analytics with datalog queries on spark". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 1135–1149.

[122] Muhammad Imran, Gábor E Gévay, and Volker Markl. "Distributed Graph Analytics with Datalog Queries in Flink". In: *Software Foundations for Data Interoperability and Large Scale Graph Data Analytics*. Springer, 2020, pp. 70–83.

[123] Alan L Rector, Jeremy E Rogers, and Pam Pole. "The GALEN high level ontology". In: *Medical Informatics Europe'96*. IOS Press, 1996, pp. 174–178.

[124] Nicholas Allen, Bernhard Scholz, and Padmanabhan Krishnan. "Staged Points-to Analysis for Large Code Bases". In: *Compiler Construction: 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. Springer Berlin Heidelberg, 2015, pp. 131–150. ISBN: 978-3-662-46663-6. DOI: 10.1007/978-3-662-46663-6_7.

[125] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. "Datalog and Emerging Applications: An Interactive Tutorial". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. Athens, Greece: ACM, 2011, pp. 1213–1216. ISBN: 978-1-4503-0661-4. DOI: 10.1145/1989323.1989456. URL: http://doi.acm.org/10.1145/1989323.1989456.

[126] *Github CodeQL*. Accessed: 19-10-2021. 2021. URL: https://codeql.github.com/.

[127] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. "Querying Data Provenance". In: SIGMOD '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, 951–962. ISBN: 9781450300322. DOI: 10.1145/1807167.1807269. URL: https://doi.org/10.1145/1807167.1807269.

[128] Alexander Schrijver. *Theory of Linear and Integer Programming*. USA: John Wiley & Sons, Inc., 1986. ISBN: 0471908541.

[129] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[130] John Hooker. "Generalized resolution for 0–1 linear inequalities". In: *Annals of Mathematics and Artificial Intelligence* 6 (Mar. 1992), pp. 271–286. DOI: 10.1007/BF01531033.

[131] Stephen M Blackburn et al. "The DaCapo benchmarks: Java benchmarking development and analysis". In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 2006, pp. 169–190.

[132] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. "From Datalog to Flix: A Declarative Language for Fixed Points on Lattices". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: ACM, 2016, pp. 194–208. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908096. URL: http://doi.acm.org/10.1145/2908080.2908096.

[133] Scott M Lundberg et al. "From local explanations to global understanding with explainable AI for trees". In: *Nature machine intelligence* 2.1 (2020), pp. 56–67.

[134] Wojciech Samek et al. *Explainable AI: interpreting, explaining and visualizing deep learning*. Vol. 11700. Springer Nature, 2019.

[135] Ke Yang and Julia Stoyanovich. "Measuring fairness in ranked outputs". In: *Proceedings of the 29th international conference on scientific and statistical database management*. 2017, pp. 1–6.