Programmable View Update Strategies in Relational Databases

by

TRAN Van Dang

Dissertation

submitted to the Department of Informatics

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

The Graduate University for Advanced Studies, SOKENDAI

September 2022

Acknowledgments

First of all, I would like to express my deep gratitude to my supervisor, Prof. Zhenjiang Hu, for his invaluable guidance, suggestions, advice, and constant support to carry out this thesis. Thank you for teaching me and always encouraging me to keep moving forward.

I would also like to deeply thank Prof. Hiroyuki Kato for taking time for weekly meetings and giving insightful comments on my research. I am very grateful to my advisors, Prof. Ichiro Hasuo and Prof. Nobukazu Yoshioka, for their support.

I would like to thank the committee members, Prof. Ichiro Hasuo, Prof. Kanae Tsushima, and Prof. Taro Sekiyama, for their insightful comments and suggestions to improve this thesis.

I would also like to thank all members of the BISCUITS project for giving me feedback that helped to improve this work. I am very grateful to Prof. Masato Takeichi, who gave me a lot of insightful comments to improve my research as well as my presentations. I would like to thank Prof. Soichiro Hidaka, I have learned a lot through discussions with him. Many thanks to Prof. Makoto Onizuka for his support during the short time I worked in his laboratory, and for many other helpful discussions on my framework.

I appreciate all members of the Programming Research laboratory, for their comments and discussions. I would like to thank Dr. Zirun Zhu, who helped me a lot when I joined the laboratory. Many thanks to Dr. Jumpei Tanaka, Xing Zhang, and Hoang-Long Huynh, who have worked with me and given me suggestions to improve my framework. I am also thankful to Dr. Yongzhe Zhang, Dr. Chunmiao Li, Bach-Trong Nguyen, and Zhixuan Yang. I really enjoyed the time with them in the last five years.

I would like to thank SOKENDAI and National Institute of Informatics for educational and financial support.

Finally, I am heartily thankful to my family for their support and encouragement at all times.

Abstract

View update is an important mechanism in relational databases. Since a view is defined by a query over the underlying database, updates on the view must be translated to the corresponding updates on the database. Existing literature has shown the ambiguity of this translation that there may be many strategies for updating the database in order to properly reflect view updates.

To address this ambiguity problem, we propose an effective language-based approach for making view update strategies programmable and validatable so that the corresponding view definition is automatically derived without ambiguity. Specifically, we design a fragment of the Datalog language for specifying update strategies and propose a validation algorithm for these strategies with an automatic derivation of the corresponding view definition. Moreover, we present a mechanism to incrementalize user-written update strategies in order to improve the performance of updatable views in realistic relational database management systems. We theoretically prove the soundness and completeness of our approach and practically validate the efficiency of the framework implementation by experiments on a benchmark collected from real-world applications. The experiments show that our validation algorithm is feasible for solving many view update strategies and our incrementalization can significantly reduce the running time of view updates in practical relational database management systems.

To enhance the ease of use of Datalog in programming view update strategies, especially for the case that the user-written programs are not valid, we propose an efficient approach to interactively debugging Datalog programs so that the user's burden is reduced. Specifically, we provide a syntax for users to specify properties of non-recursive Datalog programs. We present a counterexample generator that

verifies specified properties and generates counterexamples to show unexpected behaviors of user-written programs. We design a debugging engine combined with a dialog-based user interface to assist users in locating bugs in the programs with the generated counterexamples. We have implemented a prototype of our approach and demonstrated its feasibility and efficiency.

We further extend our approach to allow using Datalog for programming view update strategies on knowledge graphs in the RDF (Resource Description Framework) format. Recursion is the key that gives Datalog the capabilities to exploit the graph structure of RDF data. To free programmers from understanding and implementing recursive operators that are more complex than non-recursive parts, we propose an approach to formulate a Datalog-written view update strategy as the combination of two parts: a recursive part, which implements recursive patterns, and a non-recursive one, which is an inner update strategy. On the one hand, the recursive Datalog rules of the first part are pre-defined and pre-validated so that their well-behavedness is guaranteed. On the other hand, we allow programmers to manually write the inner update strategies in non-recursive Datalog, which are automatically validated. To guarantee the ease of use of pre-defined recursive programs in constructing a new one, we extend Datalog with a restricted form of higher-order predicate syntax. To improve the performance of the Datalog programs, we propose an algorithm to transform all higher-order predicates into equivalent first-order predicates that can be evaluated efficiently.

We have implemented a framework for our proposed methods. To integrate our framework with a relational database management system such as PostgreSQL, we design a compilation algorithm to transform Datalog-written view update strategies into procedural SQL code. The SQL program consists of all necessary statements for creating an updatable view in the database. The updatable view uses trigger mechanisms to automatically invoke the view update strategy in response to view updates in a single request or transaction. Therefore, programmers can run Datalog programs in a PostgreSQL database via our provided command-line tool or web-based user interface.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

1.1.1 Motivation

Nowadays, more and more software and information systems have been developed with billions of users and an increasingly huge amount of data. To manage such data, relational database management systems (RDBMS) such as Oracle Database, MySQL, PostgreSQL, and so forth are widely used as the backend systems. In relational databases, data are represented as collections of tables that allow applications to not only query but also update the source data.

Due to the separation of applications and backend databases, programmers routinely face situations in which data representation used by an application differs substantially from the data source tables. View is a well-known mechanism to fulfill this gap [1, 2]. A view is a logical table that is extracted from source tables by a defining query. By a view, the database exposes only a portion of source data

with a relevant and compatible representation to applications. Moreover, the views also contribute to hiding confidential data, reducing space consumption, speeding up query evaluation [1, 3], database refactoring [4], data integration [2, 5], interoperating data across multiple databases[6, 7, 8], and so forth. In practice, most commercial relational DBMSs provide SQL (Structured Query Language) syntax to define a view. As an example, considering a database of two source tables $r_1(A, B)$ and $r_2(A, B)$, the following SQL statement creates a view v as the union of the two tables:

```
CREATE VIEW v AS
    SELECT * FROM r1
    UNION
    SELECT * FROM r2;
```

A significant drawback of views is that a view provides read-only access to applications and other systems. Once a view is created, it can be queried exactly as source tables, and thus it is natural to update data on the view. The updatability of views plays an important role to enable read-write data flows in many essential applications of using views such as peer-to-peer data exchange, interoperating data across multiple databases, and so forth.

However, updating views is nontrivial in the sense that a desired update to the view must be translated into the corresponding update to the source tables so that the same view can be obtained by the query over the updated source tables [9, 10, 11, 12, 13].

The existing literature [10, 11] has shown the ambiguity of view updates. Because the query *get* is generally not injective, there are potentially many update translations on the source database that can be used to reflect a given view update. The following example illustrates the ambiguity issues of view updates.

Example 1.1.1 *Consider a database of two source relations $r_1(A, B)$ and $r_2(A, B)$, and a view relation v defined as the union of r_1 and r_2 as mentioned before. To illustrate the ambiguity of updates to v, consider an attempt to insert a tuple $\langle 3, 4\rangle$ into the view v. There are three simple ways to update the source database: (i) insert tuple $\langle 3, 4\rangle$ into r_1, (ii) insert tuple $\langle 3, 4\rangle$ into r_2, and (iii) insert tuple $\langle 3, 4\rangle$ into both r_1 and r_2.* □

The ambiguity issue makes view update an open challenging problem that that has attracted a lot of attention over the past three decades in database research [12, 10, 11, 9, 14, 13, 15, 16, 17, 18, 19]. Despite the long history in database literature, there is no standard solution yet to make relational views updatable. The existing approaches either impose too many syntactic restrictions on the view definition *get* that allow for limited unambiguous update propagation [11, 20, 9, 21, 22, 19, 23, 17, 24] or provide dialogue mechanisms for users to manually choose update translations with users' interaction [14, 18].

An alternative way to make a view updatable is to allow database administrators to decide and implement a strategy that specifies how view updates are propagated to the source. In RDBMSs, trigger is a well-known mechanism for developers to implement such a view update strategy in a trigger procedure. This procedure is automatically invoked in response to update requests on the view [25]. In this way, the trigger procedure can be implemented for calculating the corresponding updates on base tables for each modification on the view and then applying these updates to the base tables by INSERT, DELETE and UPDATE SQL statements.

In practice, commercial database systems such as PostgreSQL [26] provide very limited support to automatically make views updatable that even a simple union view cannot be updated. For the unsupported updatable views, the programmers commonly take the responsibility of implementing triggers and associated procedures to handle view updates. Most existing RDBMSs provide SQL procedural languages such as PL/pgSQL [27] (in PostgreSQL) for trigger implementation. Figure 1.1 shows an implementation for a trigger on the view v defined as the union of r_1 and r_2 as mentioned before. However, it is still difficult for programmers to define all the necessary triggers and associated actions for updatable views. Moreover, there is no support tool to verify the correctness of programmers' update strategies in the sense that the updatable view and its base tables are consistent for any view updates.

1.1.2 Research Objectives

To resolve the challenging ambiguity issue of view updates, our objective is to provide a formal language for programmers to explicitly specify their view update

```
1  CREATE OR REPLACE FUNCTION public.view_update() RETURNS TRIGGER
       LANGUAGE plpgsql
2  SECURITY DEFINER AS $$
3  DECLARE
4  ...
5  BEGIN
6  IF TG_OP = "INSERT" THEN
7      INSERT INTO r1 SELECT (NEW).*;
8  ELSIF TG_OP = "UPDATE" THEN
9      DELETE FROM r1 WHERE ROW(A, B) = OLD;
10     INSERT INTO r1 SELECT (OLD).*;
11     DELETE FROM r2 WHERE ROW(A, B) = OLD;
12 ELSIF TG_OP = "DELETE" THEN
13     DELETE FROM r1 WHERE ROW(A, B) = OLD;
14     DELETE FROM r2 WHERE ROW(A, B) = OLD;
15 END IF;
16 RETURN NULL;
17 EXCEPTION
18 ...
19 END;
20 $$;
21
22 CREATE TRIGGER view_update_trigger
23     INSTEAD OF INSERT OR UPDATE OR DELETE ON
24         v FOR EACH ROW EXECUTE PROCEDURE view_update();
```

Figure 1.1: A PL/pgSQL program of a trigger procedure.

strategies. This idea is inspired by the research on bidirectional programming [28, 29] in the programming language community, where update propagation from the view to the source is formulated as a so-called *putback transformation put*, which maps the updated view and the original source to an updated source. This *put* not only captures the view update strategy but also fully describes the view update behavior. First, it is clear that if we have such a putback transformation, view updates are reflected in the updated source. Second, and more interestingly, while there may be many putback transformations for a view definition *get*, there is at most one view definition for a putback transformation *put* for a well-behaved view update [30, 31, 32, 33, 34]. Thus, *get* can be deterministically derived from *put* in general. Although several languages have been proposed for writing *put* for

updatable views over tree-like data structures [35, 33, 34], whether we can design such a user-friendly and expressive language for solving the classical view update problem on relations remains unclear.

In contrast to the existing approaches [35, 33, 34] where new domain-specific languages (DSLs) are designed, our approach is to allow Datalog, a well-known query language, to be used as a formal language for describing view update strategies in relational databases.

It might be surprising that Datalog, which is a query language, can be also used for writing updates. Our main idea is to formulate a view update strategy as a query over the updated view and the original state of the source tables. Such a query results in source table updates, which can be represented in two atomic operations: insertions and deletions. The idea reveals that a view update strategy can be concisely written in a Datalog program. Specifically, for writing updates, we extend the Datalog syntax with two simple symbols "+" and "-" that indicate data insertions and deletions, respectively.

Example 1.1.2 *Consider the view in Example 3.2.1. A view update strategy can be specified in Datalog as follows:*

$$
\begin{aligned}
-r_1(X,Y) &:\!- \; r_1(X,Y), \neg v(X,Y).\\
-r_2(X,Y) &:\!- \; r_2(X,Y), \neg v(X,Y).\\
+r_1(X,Y) &:\!- \; v(X,Y), \neg r_1(X,Y), \neg r_2(X,Y).
\end{aligned}
$$

The first two rules state that if a tuple $\langle X, Y\rangle$ is in r_1 or r_2 but not in v, it will be deleted from r_1 or r_2, respectively. The last rule states that if a tuple $\langle X, Y\rangle$ is in v but in neither r_1 nor r_2, it will be inserted into r_1. □

1.1.3 Main Challenges

There are several challenges in designing a formal language for programming *put*, a view update strategy, in relational databases.

- The language is desired to be expressive in practice to cover users' update strategies for a variety of views.

- To make every view update consistent with the source database, an update strategy *put* must satisfy some certain properties, as formalized in previous work [28, 32, 31]. Therefore, there is a need for a validation algorithm to statically check the well-behavedness of user-written strategies and whether they respect the view definition if the view is defined beforehand.

- For the cases that the user-written view update strategies are not valid, it is essential to reduce the users' burden in detecting the unexpected behaviour of the program and locating the bugs.

- To be useful in practice rather than just a theoretical framework, the language must be efficiently optimized and implemented when running in relational database management systems (RDBMSs).

1.2 Contributions

1.2.1 View Update Strategies in Non-Recursive Datalog

We first consider relational views in practical SQL database management systems. In these databases, the views are commonly defined by SQL without recursion. To allow updates to these views, we propose a robust language-based approach for making view update strategies programmable and validatable. Specifically, we introduce a novel approach to use Datalog to describe these update strategies. We propose a validation algorithm to check the well-behavedness of the written Datalog programs. We present a fragment of the non-recursive Datalog language for which our validation is both sound and complete. This fragment not only has good properties in theory but is also useful for solving practical view updates. Furthermore, we develop an algorithm for optimizing user-written programs to efficiently implement updatable views in relational database management systems. We have implemented our proposed approach. The experimental results show that our framework is feasible and efficient in practice.

This is a joint work with Zhenjiang Hu and Hiroyuki Kato, and was published at the 46th International Conference on Very Large Data Bases (VLDB 2020) [36, 37]

1.2.2 Debugging Non-Recursive View Update Strategies

We consider the cases that the user-written view update strategies are not valid. Although Datalog is used in many potential applications including database queries, program analysis, and bidirectional transformations, very few approaches have been proposed for debugging Datalog programs. The existing approaches require much users' effort in finding out unintended behaviors or unexpected computations of the Datalog program that neither counterexamples nor bug explanations are provided. We propose an efficient approach to interactively debugging Datalog programs so that the user's burden is reduced. Specifically, we provide a syntax for users to specify properties of non-recursive Datalog programs. We present a counterexample generator that verifies specified properties and generates counterexamples to show unexpected behaviors of user-written programs. We design a debugging engine combined with a dialog-based user interface to assist users in locating bugs in the programs with the generated counterexamples. We have implemented a prototype of our approach and demonstrated its feasibility and efficiency.

This is a joint work with Zhenjiang Hu and Hiroyuki Kato, and was published at the 18th Asian Symposium on Programming Languages and Systems (APLAS 2020) [38]

1.2.3 Recursive View Update Strategies

We consider views in Resource Description Framework (RDF) where data is stored in special ternary relations of triples $(subject, predicate, object)$. In practice, these RDF views can be defined over relational databases (e.g., by using the W3C standard R2RML mapping language [39]) or knowledge graphs in RDF format (e.g., by using SPARQL). Due to the complex structure, manipulating graphs usually requires recursions. Recursion is the key that gives Datalog the capabilities to exploit the graph structure of RDF data. To free programmers from understanding and implementing recursive operators that are more complex than non-recursive parts, we propose an approach to formulate a Datalog-written view update strategy as the combination of two parts: a recursive part, which implements recursive patterns, and a non-recursive one, which is an inner update

strategy. On the one hand, the recursive Datalog rules of the first part are pre-defined and pre-validated so that their well-behavedness is guaranteed. On the other hand, we allow programmers to manually write the inner update strategies in non-recursive Datalog, which are automatically validated. To guarantee the ease of use of pre-defined recursive programs in constructing a new one, we extend Datalog with a restricted form of higher-order predicate syntax. To improve the performance of the Datalog programs, we propose an algorithm to transform all higher-order predicates into equivalent first-order predicates that can be evaluated efficiently. We show the expressiveness of our proposed approach by implementing several classes of view update strategies for some common recursive patterns of RDF graphs.

This is a joint work with Zhenjiang Hu and Hiroyuki Kato, and was partially published at the Ninth International Workshop on Bidirectional Transformations (Bx 2021) [40]

1.3 Organization of the thesis

This thesis is organized as follows. In Chapter 2, we introduce the basic notions and general related work of the thesis. Chapter 3 presents our approach of using non-recursive Datalog to program view update strategies. In Chapter 4, we present a novel method for debugging non-recursive Datalog of view update strategies. Chapter 5 presents our approach for view update strategies where recursions are allowed. Finally, we give a summary of the thesis and discuss future work in Chapter 6.

2 Preliminaries

In this chapter, we first briefly review the basic concepts and notations that will be used throughout this thesis. We then introduce the basics of view update. Finally, we present general related work as the rationale of our approach.

2.1 Relational Databases and Datalog

2.1.1 Relational databases

A database schema $\mathcal{D}$ is a finite sequence of relation names (or predicate symbols, or simply predicates) $\langle r_1, \ldots, r_n \rangle$. Each predicate r_i has an associated arity $n_i > 0$ or an associated sequence of attribute names $A_1, \ldots, A_{n_i}$. A database (instance) D of $\mathcal{D}$ assigns to each predicate r_i in $\mathcal{D}$ a finite n_i-ary relation R_i, $D(r_i) = R_i$.

An atom (or atomic formula) is of the form $r(t_1, \ldots, t_k)$ (or written as $r(\vec{t})$) such that r is a k-ary predicate and each t_i is a term, which is either a constant or

a variable. When $t_1, \ldots, t_k$ are all constants, $r(t_1, \ldots, t_k)$ is called a ground atom.

A database D can be represented as a set of ground atoms [41, 42], where each ground atom $r(t_1, \ldots, t_k)$ corresponds to the tuple $\langle t_1, \ldots, t_k \rangle$ of relation R in D. As an example of a relational database, consider a database D that consists of two relations with respective schemas $r_1(A, B)$ and $r_2(C)$. Let the actual instances of these two relations be $R_1 = \{\langle 1, 2\rangle, \langle 2, 3\rangle\}$ and $R_2 = \{\langle 3\rangle, \langle 4\rangle\}$, respectively. The set of ground atoms of the database is $D = \{r_1(1, 2), r_1(2, 3), r_2(3), r_2(4)\}$.

2.1.2 Datalog

A Datalog program P is a nonempty finite set of rules, and each rule is an expression of the form [41]:

$$H :- L_1, \ldots, L_n.$$

where $H, L_1, \ldots, L_n$ are atoms. H is called the rule head, and $L_1, \ldots, L_n$ is called the rule body. "$:-$" is a variant of the standard logical implication "$\leftarrow$" from the rule body in the right-hand side to the rule head in the left-hand side.

The input of P is a set of ground atoms, called the extensional database (EDB), physically stored in a relational database. The output of P is all ground atoms derived through the program P and the EDB, called the intensional database (IDB). Predicates in P are divided into two categories: the EDB predicates occurring in the extensional database, and the IDB predicates occurring in the intensional database. An EDB predicate can never be the head predicate of a rule. The head predicate of each rule is an IDB predicate. We assume that each EDB/IDB predicate r corresponds to exactly one EDB/IDB relation R. Following the convention used in [41], throughout this thesis, we use lowercase characters for predicate symbols and uppercase characters for variables in Datalog programs. In a Datalog rule, variables that occur exactly once can be replaced by an anonymous variable, denoted as "_".

We can extend the Datalog syntax with negation and built-in predicates, such as equality ($=$) and comparison ($<, >$), in Datalog rule bodies but in a safe way in which each variable occurring in the negated atoms or the built-in predicates must also occur in some positive atoms [41].

Let P be a Datalog program and D be the database of all the EDB and IDB

relations. A tuple $\vec{A}$ of r, or a fact $r(\vec{A})$, is immediately inferred from P and D if it satisfies one of the following conditions:

- $\vec{A} \in R$, where R is an EDB relation corresponding to predicate r.
- $r(\vec{A}) :- (\neg)r_1(\vec{A}_1), \ldots, (\neg)r_n(\vec{A}_n).$ is an instantiation of a rule in P, i.e., all variables in the rule are substituted with constants. Here, a negative fact $\neg r_i(\vec{A}_i)$ holds if the fact $r_i(\vec{A}_i)$ does not hold, i.e., $\vec{A}_i$ is not a tuple of r_i in D. This is based on the Closed World Assumption (CWA) [41].

Semantically, evaluating P is computing the minimum database D such that every tuple in D is immediately inferred from D and P. In other words, we compute the least fixpoint of the immediate inference operator. In the standard bottom-up evaluation strategy for Datalog, the least fixpoint is obtained from P and the input EDB database by deriving all IDB tuples with a finite number of immediate inferences. To deal with negations in the Datalog program, the Datalog program is stratified to ensure that all the tuples of an IDB relation are derived before using any negative facts of this IDB relation in other immediate inferences. This is because if an IDB relation is incomplete, it is not sufficient to judge a negative fact of the IDB relation. The sequence of immediate inferences used for deriving a fact is called a proof of the fact and can be represented in a proof tree with different levels of the applied rules and facts.

A Datalog program (with extensions) must satisfy the safety conditions (syntactic restriction) to guarantee that there are a finite number of facts that can be derived from the Datalog program:

1. Each variable that occurs in the head of a rule must also occur in the body of that rule [41]
2. Each variable occurring in a negative literal of a rule body also occurs in a positive literal of the same rule body [41]
3. For the equality of the form $x = y$ where x and y are variables, either x or y must occur in a positive literal of the same rule body
4. Each variable occurring in a negative literal of equality ($\neg t_1 = t_2$, where t_1 and t_2 are terms) must occur in a positive literal of the same rule body

A Datalog program takes as input a database of EDB relations to derive all IDB relations, corresponding to IDB predicates in P. A Datalog program P can have many IDB predicates. If restricting the output of P to an IDB relation R corresponding to IDB predicate r, we have a Datalog query, denoted as (P, R). We say that an IDB predicate r (or a query (P, R)) is satisfiable if there exists a database D such that the IDB relation R in the output of P over D is nonempty [43].

A Datalog rule is recursive if an IDB predicate appears in both the head and the body of the rule. A Datalog program is non-recursive (called NR-Datalog) if it has no recursive rule. Without recursion, Datalog is still an expressive query. It has been shown that non-recursive Datalog (NR-Datalog$^\neg$) query with a goal of a single IDB relation has the same expressiveness as several query languages for relational databases such as relational calculus query and relational algebra. Recall that a relational calculus query over a database is an expression of the form: $\{x_1, \ldots, x_m | \varphi\}$, where φ is a well-formed first-order formula over the database, $x_1, \ldots, x_m$ are free variables of φ. Relational algebra is an algebra with primitive operators including selection, projection, join, union, and set difference.

Theorem 2.1.1 (Equivalence theorem [43]) *The relational calculus queries, the relational algebras, and the family of non-recursive datalog with negation (NR-Datalog$^\neg$) programs that have single-relation output have equivalent expressive power.* □

It is known that query satisfiability of Datalog with negation is undecidable [43]. If we restrict all negations in Datalog rules to be guarded, the query satisfiability is decidable. We simplify the definition of guarded negation in Datalog presented in [44] for the case of non-recursive Datalog as the following:

Definition 2.1.2 (Guarded negation [44]) *A non-recursive guarded-negation Datalog (GN-Datalog) program is a non-recursive Datalog program with negation where each rule $L_0 :- L_1, \ldots, L_n$ is negation guarded that for each atom L_i that is either negative in the body or positive in the head, the body includes a positive atom L_j containing all variables occurring in L_i.* □

Theorem 2.1.3 (Decidability [44]) *Query satisfiability is 2ExpTime-complete for non-recursive guarded-negation Datalog queries.* □

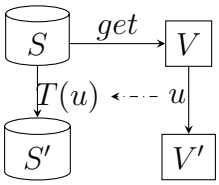


Figure 2.1: The view update problem.

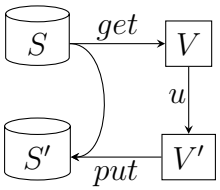


Figure 2.2: Bidirectional transformation.

2.2 View update

Consider a view V defined by a query *get* over the database S, as shown in Figure 2.1. An update translator T maps each update u on V to an update $T(u)$ on S such that it is *well-behaved* in the sense that after the view update is propagated to the source, we will obtain the same view from the updated source, i.e.,

$$u(V) = get(T(u)(S))$$

Given a view definition *get*, the known *view update problem* [11] is to derive such an update translator T.

2.3 Bidirectional Transformations

A bidirectional transformation (BX) [28] is a pair of a forward transformation *get* and a backward (putback) transformation *put*, as shown in Figure 2.2. The forward transformation *get* is a query over a source database S that results in a view relation V. The putback transformation *put* takes as input the original database S and an updated view V' to produce a new database S', and thus determines a view update strategy. Indeed, given a putback transformation *put*,

the view update translation T is obtained as the following:

$$T(u)(S) = put(S, u(get(S))).$$

To ensure consistency between the source database and the view, a BX must satisfy the following *round-tripping* properties, called GetPut and PutGet:

$$\forall\, S, \qquad put\,(S,\ get(S)) = S \tag{GetPut}$$

$$\forall\, S,\ V', \qquad get\,(put\,(S,\ V')) = V' \tag{PutGet}$$

The GetPut property ensures that unchanged views correspond to unchanged sources, while the PutGet property ensures that all view updates are completely reflected to the source such that the updated view can be computed again from the query *get* over the updated source.

Definition 2.3.1 (Validity of Update Strategy)
A view update strategy put is said to be valid if there exists a view definition get such that put and get satisfy both GetPut *and* PutGet. □

The important property that makes putback essential for BXs is that a valid view update strategy *put* uniquely determines the view definition *get*, which satisfies GetPut and PutGet with *put*. Therefore, although *put* is written in a unidirectional (backward) manner, if *put* is valid, it can capture both forward and backward directions. We state the uniqueness of the view definition *get* in the following theorem, and the proof can be found in [32].

Theorem 2.3.2 (Uniqueness of View Definition)
Given a view update strategy put, there is at most one view definition get that satisfies GetPut *and* PutGet *with put.* □

Proof. By contradiction. If there are two view definitions get_1 and get_2 that satisfy the condition, then by applying the GetPut and PutGet properties to the expression $E = get_1(put(S, get_2(S)))$, we have $E = get_1(S)$ and $E = get_2(S)$, respectively. This means that $get_1(S) = get_2(S)$ for any database S, i.e., get_1 and get_2 are equivalent. □

Some important characteristics of a valid update strategy *put* are given in the following lemma, the proof can be found in [32, 31]:

Lemma 2.3.3 *A view update strategy put is valid if, and only if, it satisfies all the following properties:*

1. *For all database S, the function f such that $f(V) = put(S, V)$ is injective (*PutInjectivity*).*

2. *put is surjective:*

$$\forall\ S'\ \exists\ S, V.\ put(S, V) = S' \qquad \text{(PutSurjectivity)}$$

3. *For all views V, the function g such that $g(S) = put(S, V)$ is idempotent:*

$$put(put(S, V), V) = put(S, V) \qquad \text{(PutTwice)}$$

□

2.4 Rationale of Our Approach

2.4.1 Classical Approaches

The view update problem is a classical problem that has a long history in database research [12, 10, 11, 9, 14, 45, 13, 15, 46, 47, 16, 23, 17, 24, 18, 19]. It was realized very early that a database update that reflects a view update may not always exist, and even if it does exist, it may not be unique [10, 11]. This makes view update become a very challenging problem.

To solve the ambiguity of translating view updates to updates on base relations, the concept of view complement is proposed to determine the unique update translation of a view [9, 21, 22, 19]. Keller [14] enumerates all view update translations and chooses the one through interaction with database administrators, thereby solving the ambiguity problem. Some other researchers allow users to choose the one through an interaction with the user at view definition time [14, 18]. Some other approaches restrict the syntax for defining views [11] that allow for

unambiguous update propagation. Recently, intention-based approaches have been proposed to find relevant update policies for several types of views [23, 17, 24]. However, these existing approaches restrict the syntax of view definition and provide a set of update policies for each kind of view, thus they do not give users full control over view update strategies.

In another aspect, because some updates on views are not translatable, some works permit side effects of the view update translator [45] or restrict the kind of updates that can be performed on a view [13]. Some other works use auxiliary tables to store the updates, which cannot be applied to the underlying database [15, 46]. The authors of [47, 16] studied approximation algorithms to minimize the side effects for propagating deletion from the view to the source database. However, these existing approaches can only solve a very restricted class of view updates.

2.4.2 Language-Based Approaches

In the programming language community, by generalizing view update as a synchronization problem between two data structures, considerable modern approaches has been devoted [29] to this problem not only in relational databases [20, 48] but also for other data types, such tree [28, 49], graph [50] or string data [51]. By employing the concept of bidirectional transformations, these approaches provide domain-specific languages (DSLs) for programmers to specify updatable views. They can be devided into two categories: get-based and put-based bidirectional languages.

In the get-based approaches, the language of the forward transformation is enriched such that it can capture some update intentions of the programmers. A typical work is [20] that employs bidirectional transformation for view update in relational databases. The authors propose a bidirectional language, called relational lenses, by enriching the SQL expression for defining views of projection, selection, and join. The language guarantees that every expression can be interpreted forwardly as a view definition and backwardly as an update strategy such that these backward and forward transformations are well-behaved. A recent work [48] has shown that incrementalization is necessary for relational lenses to make this language practical in RDBMSs. However, this language is less expressive

than general relational algebra; hence, not every updatable view can be written. Moreover, relational lenses still limit programmers from control over the update strategy.

In contrast to the get-based approaches, the put-based approahces design a new DSL for completely write a putback transformation. The key observation in this approach is that thanks to well-behavedness, putback transformation uniquely determines the forward one. The uniqueness of *get* enable us to desing languages that allow programmers to write their intended update strategies more freely and derive the *get* behavior from their putback program. A typical language of this putback-based approach is BiGUL [33, 34], which supports programming putback functions declaratively while automatically deriving the corresponding unique forward transformation. Based on BiGUL, Zan et al. [52] design a putback-based Haskell library for bidirectional transformations on relations. However, this language is designed for Haskell data structures; hence, it cannot run directly in database environments. The transformation from tables in relational databases to data structures in Haskell would reduce the performance of view updates.

Our work was greatly inspired by the putback-based approach in bidirectional programming [30, 53, 54, 31, 33, 34]. In contrast to other works, we propose adopting the Datalog language for implementing view update strategies at the logical level, which will be optimized and translated to SQL statements to run efficiently inside an SQL database system.

3 View Update Strategies in Non-Recursive Datalog

3.1 Introduction

As mentioned in Chapter 1, view update [9, 10, 11, 12, 13] is an important mechanism in relational databases. This mechanism allows updates on a view by translating them into the corresponding updates on the base relations [11]. The main challenge of view updates is the ambiguity issue. Because the query *get* is generally not injective, there may be many update translations on the source database that can be used to reflect view update [10, 11]. The view update problem have a long history in database research [12, 10, 11, 9, 14, 13, 15, 16, 17, 18, 19].

In this chapter, we propose a new approach for solving the view updating problem practically and correctly. The key idea is to provide a formal language for people to directly program their view update strategies. On the one hand, this language can be considered a formal treatment of Keller's dialogue [14], but on the

other hand, it is unique in that it can fully determine the behavior of bidirectional update propagation between the source and the view.

This idea is inspired by the research on bidirectional programming [28, 29] in the programming language community, where update propagation from the view to the source is formulated as a so-called *putback transformation put*, which maps the updated view and the original source to an updated source, as shown in Figure 2.2. This *put* not only captures the view update strategy but also fully describes the view update behavior. First, it is clear that if we have such a putback transformation, the translation T is obtained for free:

$$T(u)(S) = put(S, u(get(S))).$$

Second, and more interestingly, while there may be many putback transformations for a view definition *get*, there is at most one view definition for a putback transformation *put* for a well-behaved view update [30, 31, 32, 33, 34]. Thus, *get* can be deterministically derived from *put* in general. Although several languages have been proposed for writing *put* for updatable views over tree-like data structures [35, 33, 34], whether we can design such a language for solving the classical view update problem on relations remains unclear.

There are several challenges in designing a formal language for programming *put*, a view update strategy, on relations.

- The language is desired to be expressive in practice to cover users' update strategies.

- To make every view update consistent with the source database, an update strategy *put* must satisfy some certain properties, as formalized in previous work [28, 32, 31]. Therefore, there is a need for a validation algorithm to statically check the well-behavedness of user-written strategies and whether they respect the view definition if the view is defined beforehand.

- To be useful in practice rather than just a theoretical framework, the language must be efficiently implemented when running in relational database management systems (RDBMSs).

In contrast to the existing approaches [35, 33, 34] where new domain-specific languages (DSLs) are designed, we argue that Datalog, a well-known query language, can be used as a formal language for describing view update strategies in relational databases. Our contributions are summarized as follows.

- We introduce a novel way to use nonrecursive Datalog with negation and built-in predicates for describing view update strategies. We propose a validation algorithm for statically checking the well-behavedness of the described update strategies.

- We identify a fragment of Datalog, called linear-view guarded negation Datalog (LVGN-Datalog), in which our validation algorithm is both sound and complete. Furthermore, the algorithm can automatically derive from view update strategies the corresponding view definition to confirm the view expected beforehand.

- We develop an incrementalization algorithm to optimize view update strategy programs. This algorithm integrates the standard incrementalization method for Datalog with the well-behavedness in view update.

- We have implemented all the algorithms in our framework, called BIRDS[1]. The experiments on benchmarks collected in practice show that our framework is feasible for checking most of the view update strategies. Interestingly, LVGN-Datalog is expressive enough for solving many types of views and can be efficiently implemented by incrementalization in existing RDBMSs.

The remainder of this chapter is organized as follows. We present our proposed method for specifying view update strategies in Datalog in Section 3.2. The validation and incrementalization algorithms for these update strategies are described in Section 3.3 and Section 3.4, respectively. Section 3.5 shows the experimental results of our implementation. Section 3.6 summarizes related works. Section 3.7 concludes this chapter.

[1] A prototype implementation is available at https://dangtv.github.io/BIRDS/.

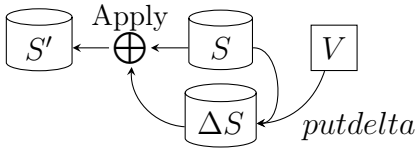


Figure 3.1: View update strategy *put*.

3.2 The Language for View Update Strategies

As mentioned in the introduction, it may be surprising that the base language that we are using for view update strategies is nonrecursive Datalog with negation and built-in predicates (e.g., $=$, $\neq$, $<$, $>$) [41]. One might wonder how the pure query language Datalog can be used to describe updates. In this section, we show that delta relations enable Datalog to describe view update strategies. We will define a fragment of Datalog, called LVGN-Datalog, which is not only powerful for describing various view update strategies but also important for our later validation.

3.2.1 Formulating Update Strategies as Queries Producing Delta Relations

Recall that a view update strategy is a putback transformation *put* that takes as input the original source database and an updated view to produce an updated source. Our idea of specifying the transformation *put* in Datalog is to write a Datalog query that takes as input the original source database and an updated view to yield updates on the source; thus, the new source can be obtained.

We use delta relations to represent updates to the source database. The concept of delta relations is not new and is used in the study on the incrementalization of Datalog programs [55]. Unlike the use of delta relations to describe incrementalization algorithms at the meta level, we let users consider both relations and their corresponding delta relations at the programming level.

Let R be a relation and r be the predicate corresponding to R. Following [56, 57, 58], we use two delta predicates $+r$ and $-r$ and write $+r(\vec{t})$ and $-r(\vec{t})$ to denote the insertion and deletion of the tuple $\vec{t}$ into/from relation R, respectively. An update that replaces tuple $\vec{t}$ with a new one $\vec{t'}$ is a combination of a deletion

$-r(\vec{t})$ and an insertion $+r(\vec{t})$. We use a delta relation, denoted as ΔR, to capture both these deletions and insertions. For example, consider a binary relation $R = \{\langle 1,2\rangle, \langle 1,3\rangle\}$; applying a delta relation $\Delta R = \{-r(1,2), +r(1,1)\}$ to R results in $R' = \{\langle 1,1\rangle, \langle 1,3\rangle\}$. Let Δ_R^+ be the set of insertions and Δ_R^- be the set of deletions in ΔR. Applying ΔR to the relation R is to delete tuples in Δ_R^- from R and insert tuples in Δ_R^+ into R. Considering set semantics, the delta application is the following:

$$R' = R \oplus \Delta R = (R \setminus \Delta_R^-) \cup \Delta_R^+$$

An update strategy for a view can now be specified by a set of Datalog rules that define delta relations of the source database from the updated view.

Example 3.2.1 *Consider a source database S, which consists of two base relations, R_1 and R_2, with respective schemas $r_1(A)$ and $r_2(A)$, and a view relation V defined by a union over R_1 and R_2: $V = get(S) = R_1 \cup R_2$. To illustrate the ambiguity of updates to V, consider an attempt to insert a tuple $\langle 3\rangle$ into the view V. There are three simple ways to update the source database: (i) insert tuple $\langle 3\rangle$ into R_1, (ii) insert tuple $\langle 3\rangle$ into R_2, and (iii) insert tuple $\langle 3\rangle$ into both R_1 and R_2. Therefore, the update strategy for the view needs to be explicitly specified to resolve the ambiguity of view updates. Given original source relations R_1 and R_2 and an updated view relation V, the following Datalog program is one strategy for propagating data in the updated view to the source:*

$$\begin{aligned} -r_1(X) &:\!- r_1(X), \neg v(X). \\ -r_2(X) &:\!- r_2(X), \neg v(X). \\ +r_1(X) &:\!- v(X), \neg r_1(X), \neg r_2(X). \end{aligned}$$

The first two rules state that if a tuple $\langle X\rangle$ is in R_1 or R_2 but not in V, it will be deleted from R_1 or R_2, respectively. The last rule states that if a tuple $\langle X\rangle$ is in V but in neither R_1 nor R_2, it will be inserted into R_1. Let the actual instances of the source and the updated view be $S = \{r_1(1), r_2(2), r_2(4)\}$ and $V = \{v(1), v(3), v(4)\}$, respectively. The input for the Datalog program is a database of both the source and the view $(S,V) = \{r_1(1), r_2(2), r_2(4), v(1), v(3), v(4)\}$. Thus, the result is delta relations $\Delta R_1 = \{+r_1(3)\}$ and $\Delta R_2 = \{-r_2(2)\}$. By applying these delta relations

to S, we obtain a new source database $S' = \{r_1(1), r_1(3), r_2(4)\}$. □

Formally, consider a database schema $\mathcal{S} = \langle r_1, \ldots, r_n \rangle$ and a single view v. Let S be a source database and V be an updated view relation. We use ΔS to denote all insertions and deletions of all relations in S. For example, the ΔS in Example 3.2.1 is $\Delta S = \{+r_1(3), -r_2(2)\}$. We say that ΔS is *non-contradictory* if it has no insertion/deletion of the same tuple into/from the same relation. Applying a non-contradictory ΔS to a database S, denoted as $S \oplus \Delta S$, is to apply each delta relation in ΔS to the corresponding relation in S. We use the pair (S, V) to denote the database instance I over the schema $\langle r_1, \ldots, r_n, v \rangle$ such that $I(r_i) = S(r_i)$ for each $i \in [1, n]$ and $I(v) = V$. A view update strategy *put* is formulated by a Datalog query *putdelta* over the database (S, V) that results in a ΔS (shown in Figure 3.1) as follows:

$$put(S, V) = S \oplus putdelta(S, V) \tag{3.1}$$

The Datalog program *putdelta* is called a *Datalog putback program* (or *putback program* for short). The result of *putdelta*, ΔS, should be non-contradictory to be applicable to the original source database S.

Definition 3.2.2 (Well-definedness) *A putback program is well defined if, for every source database S and view relation V, the program results in a non-contradictory ΔS.* □

3.2.2 LVGN-Datalog

We have seen that nonrecursive Datalog with extensions including negation and built-in predicates can be used for specifying view update strategies. We now focus on the extensions of Datalog in which the satisfiability of queries is decidable. This property plays an important role in guaranteeing that the validity of putback programs is decidable. Specifically, we define a fragment of Datalog, LVGN-Datalog, which is an extension of nonrecursive guarded negation Datalog (GN-Datalog [44]) with equalities, constants, comparisons [41] and linear view predicate. This Datalog fragment allows not only for writing many practical view update strategies but also for decidable checking of validity later.

Nonrecursive GN-Datalog with Equalities, Constants, and Comparisons

We consider a restricted form of negation in Datalog, called GN-Datalog [59, 44], in which we can decide the satisfiability of any queries. In this way, we define LVGN-Datalog as an extension of this GN-Datalog fragment without recursion as follows:

- Equality is of the form $t_1 = t_2$, where t_1/t_2 is either a variable or a constant.
- Comparison predicates $<$ ($>$) on totally ordered domains in the form of $X < c$ ($X > c$), where X is a variable and c is a constant.
- Constants may freely be used in Datalog rule bodies or rule heads without restriction.
- Every rule is negation guarded [44] such that for every atom L (or equality, or comparison) occurring either in the rule head or negated in the rule body, the body must have a positive atom or equality, called a *guard*, containing all variables occurring in L.

Example 3.2.3 *The following rule is negation guarded:*

$$h(X,Y,Z) :- \underbrace{r_1(X,Y,Z)}_{guard}, \neg \underbrace{Z = 1}_{equality}, \neg r_2(X,Y,Z).$$

because the negated atom $r_2(X,Y,Z)$, negated equality $\neg Z = 1$ and the head atom $h(X,Y,Z)$ are all guarded since all variables X, Y, and Z are in the positive atom $r_1(X,Y,Z)$. □

Linear View

As formally proven in [31], the putback transformation *put* must be lossless (i.e., injective) with respect to the view relation. This means that all information in the view must be embedded in the updated source. To enable tracking this behavior of putback programs in LVGN-Datalog, we introduce a restriction called *linear view*, which controls the usage of the view in the programs. By linear view, we mean that the view is linearly used such that there is no self-join and

projection on the view. Every program in LVGN-Datalog conforms to the linear view restriction defined as follows.

Definition 3.2.4 (Linear view) *A Datalog putback program conforms to the linear view restriction if the view occurs only in the rules defining delta relations, and in each of these delta rules, there is at most one view atom and no anonymous variable (_) occurs in the view atom.* □

Example 3.2.5 *Given a source relation R of arity 3 and a view relation V of arity 2, consider the following rules of the delta relation ΔR:*

$$-r(X,Y,Z) :- r(X,Y,Z), \neg \underbrace{v(X,Y)}_{\textit{linear view}}. \qquad (rule_1)$$

$$-r(X,Y,Z) :- r(X,Y,Z), \neg \underbrace{v(X,\_)}_{\textit{projection}}. \qquad (rule_2)$$

$$+r(X,Y,Z) :- \underbrace{v(X,Y), v(Y,Z)}_{\textit{self-join}}, \neg r(X,Y,Z). \qquad (rule_3)$$

$(rule_1)$ conforms to the linear view restriction because $v(X,Y)$ occurs once in the rule body, whereas $(rule_2)$ and $(rule_3)$ do not because there is an anonymous variable (_) in the atom of v in $(rule_2)$ and there is a self-join of v in $(rule_3)$. □

Integrity Constraints

Since an updatable view can be treated as a base table, it is natural to create constraints on the view. Similar to the idea of negative constraints introduced in [42], we extend the rules in LVGN-Datalog by allowing a truth constant *false* (denoted as $\bot$) in the rule head for expressing integrity constraints. The linear view restriction defined in Definition 3.2.4 is also extended that the view predicate can also occur in the rules having $\bot$ in the head. In this way, a constraint, called the *guarded negation constraint*, is of the form $\forall \vec{X}, \Phi(\vec{X}) \rightarrow \bot$, where $\Phi(\vec{X})$ is the conjunction of all atoms and negated atoms in the rule body and $\Phi(\vec{X})$ is a guarded negation formula. The universal quantifiers $\forall \vec{X}$ are omitted in Datalog rules.

Example 3.2.6 *Consider a view relation $v(X, Y, Z)$. To prevent any tuples having $Z > 2$ in the view v, we can use the following constraint: $\bot :- v(X, Y, Z), Z > 2$.* □

Properties

We say that a query Q is satisfiable if there is an input database D such that the result of Q over D is nonempty. The problem of determining whether a query in nonrecursive GN-Datalog is satisfiable is known to be decidable [44]. It is not surprising that allowing equalities, constants and comparisons in nonrecursive GN-Datalog does not make the satisfiability problem undecidable since the same already holds for guarded negation in SQL [44]. The idea is that we can transform such a GN-Datalog query into an equivalent guarded negation first-order (GNFO) formula whose satisfiability is decidable [59].

Lemma 3.2.7 *The query satisfiability problem is decidable for nonrecursive GN-Datalog with equalities, constants and comparisons.* □

Given a set of guarded negation constraints Σ and a query Q, we say that Q is satisfiable under Σ if there is an input database D satisfying all constraints in Σ such that the result of Q over D is nonempty.

Theorem 3.2.8 *The query satisfiability problem for nonrecursive GN-Datalog with equalities, constants and comparisons under a set of guarded negation constraints is decidable.* □

3.2.3 A Case Study

We consider a database of five base tables shown in Figure 3.2. The base tables `male`, `female` and `others` contain personal information. Table `ed` has all historical departments of each person, while `eed` contains only former departments of each person. We illustrate how to use LVGN-Datalog to describe update strategies for the views defined in Figure 3.2.

For the view `residents`, which contains all personal information, we use the attribute *gender* to choose relevant base tables for propagating updated tuples in

```
male(emp_name: string, birth_date: date).
female(emp_name: string, birth_date: date).
others(emp_name: string, birth_date: date,
       gender: string).
ed(emp_name: string, dept_name: string).
eed(emp_name: string, dept_name: string).
```

Base tables

```
ced(E, D)                :- ed(E, D), ¬ eed(E, D).
residents(E, B, G)       :- others(E, B, G).
residents(E, B, 'F')     :- female(E, B).
residents(E, B, 'M')     :- male(E, B).
residents1962(E, B, G) :- residents(E, B, G),
                          ¬B <'1962-01-01', ¬B >'1962-12-31'.
employees(E, B, G)       :- residents(E, B, G), ced(E, D).
retired(E)               :- residents(E, B, G), ¬ced(E, _).
```

Views

Figure 3.2: Database and view schema.

residents. More concretely, if there is a person in residents but not in any of the source tables male, female and other, we insert this person into the table corresponding to his/her *gender*. In contrast, we delete from the source tables the people who no longer appear in the view. The Datalog putback program for residents is the following:

```
+male(E, B)    :- residents(E, B, 'M'),
                  ¬ male(E, B), ¬ others(E, B, 'M').
-male(E, B)    :- male(E, B), ¬ residents(E, B, 'M').
+female(E, B) :- residents(E, B, G), G ='F',
                  ¬ female(E, B), ¬ others(E, B, G).
-female(E, B) :- female(E, B), ¬ residents(E, B, 'F').
+others(E, B, G) :- residents(E, B, G), ¬ G ='M',
                    ¬ G ='F', ¬ others(E, B, G).
-others(E, B, G) :- others(E, B, G),
                    ¬ residents(E, B, G).
```

The view ced contains information about the current departments of each employee. We express the following update strategy for propagating updated data in this view to the base tables ed and eed. If a person is in a department according to ed but he/she is currently no longer in this department according to ced, this department becomes his/her previous department and thus needs to be added to eed. If a person used to be in a department according to eed but he/she returned to this department according to ced, then this department of him/her needs to be removed from eed.

$$\texttt{ed}(E,D) \;\; :- \;\; \texttt{ced}(E,D),\; \neg\; \texttt{ed}(E,D).$$
$$\texttt{-eed}(E,D) \;:- \;\; \texttt{ced}(E,D),\; \texttt{eed}(E,D).$$
$$\texttt{+eed}(E,D) \;:- \;\; \texttt{ed}(E,D),\; \neg\; \texttt{ced}(E,D),\; \neg\; \texttt{eed}(E,D).$$

The view residents1962 is defined from the view residents such that residents1962 contains all residents that have a birth date in 1962. Interestingly, because the view residents is now updatable, residents can be considered as the source relation of residents1962. Therefore, we can write an update strategy on residents1962 for updating residents instead of updating the base tables male, female and others as follows:

```
% Constraints:
⊥ :- residents1962(E,B,G), B >'1962-12-31'.
⊥ :- residents1962(E,B,G), B <'1962-01-01'.
% Update rules:
+residents(E,B,G) :- residents1962(E,B,G),
                     ¬ residents(E,B,G).
-residents(E,B,G) :- residents(E,B,G),
                     ¬ B <'1962-01-01',
                     ¬ B >'1962-12-31',
                     ¬ residents1962(E,B,G).
```

We define the constraints to guarantee that in the updated view residents1962, there is no tuple having a value of the attribute *birth_date* not in 1962. Any view updates that violate these constraints are rejected. In this way, our update strategy is to insert into the source table residents any new tuples appearing in residents1962 but not yet in residents. On the other hand, we delete

only tuples in residents having *birth_date* in 1962 if they no longer appear in residents1962.

The view employees contains residents who are employed, whereas retired contains residents who retired. Since employees and retired are defined from two updatable views residents and ced, we can use residents and ced as the source relations to write an update strategy of employees:

```
% Constraints:
⊥ :− employees(E, B, G), ¬ ced(E, _).
% Update rules:
+residents(E, B, G) :− employees(E, B, G),
                       ¬ residents(E, B, G).
-residents(E, B, G) :− residents(E, B, G),
                       ced(E, _), ¬ employees(E, B, G).
```

Interestingly, in this strategy, we use a constraint to specify more complicated restrictions of updates on employees. The constraint implies that there must be no tuple $\langle E, B, G\rangle$ in the updated view employees having the value E of the attribute *emp_name*, which cannot be found in any tuples of ced. In other words, the constraint does not allow insertion into employees an actual new employee who is not mentioned in the source relation ced. The update strategy then reflects updates on the view employees to updates on the source residents.

For retired, we describe an update strategy to update the current employment status of residents as follows:

```
-ced(E, D)          :− ced(E, D), retired(E).
+ced(E, D)          :− residents(E, _, _), ¬ retired(E),
                       ¬ ced(E, _), D =‘unknown’.
+residents(E, B, G) :− retired(E), G =‘unknown’,
                       ¬ residents(E, _, _), B =‘00-00-00’.
```

We have presented the formal way to describe view update strategies using Datalog. In the next section, we will present our proposed validation algorithm for checking the validity of these update strategies. In fact, if an update strategy specified in LVGN-Datalog is valid, the corresponding view definition can be automatically derived and expressed in nonrecursive GN-Datalog with equalities,

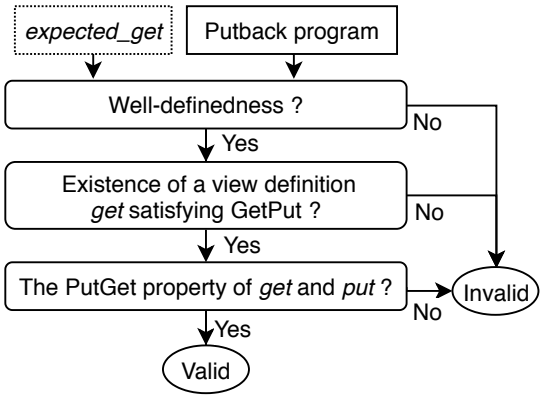


Figure 3.3: Validation algorithm.

constants and comparisons. For all the update strategies in our case study, the view definitions derived by our validation algorithm are the same as the expected ones in Figure 3.2.

3.3 Validation Algorithm

As mentioned in Chapter 2, a view update strategy must be valid (Definition 2.3.1) to guarantee that every view update is well-behaved. In this section, we present an algorithm for checking the validity of user-written view update strategies.

3.3.1 Overview

Checking the validity of a view update strategy based on Definition 2.3.1 is challenging since it requires constructing a view definition satisfying *both* the GetPut and PutGet properties. Instead, we shall propose another way for the validity check based on the following important fact.

Lemma 3.3.1 *Given a valid view update strategy put, if a view definition get satisfies* GetPut*, then get must also satisfy* PutGet *with put.* □

Lemma 3.3.1 implies that if *put* is valid, we can construct a view definition *get* that satisfies both GetPut and PutGet by choosing any *get* satisfying GetPut.

By Lemma 3.3.1, the idea of our validation algorithm is detecting contradictions for the assumption that the given view update strategy *put* is valid. Assuming that *put* is valid, we first check the existence of a view definition *get* satisfying GETPUT with *put*. We consider the expected view definition *expected_get* if available as a candidate for the *get* definition and construct the *get* definition if *expected_get* does not satisfy GETPUT. Clearly, if *get* does not exist, we can conclude that *put* is invalid. Otherwise, we continue to check whether *get* also satisfies PUTGET with *put* (Lemma 3.3.1). If this check passed, we actually complete the validation and it is sufficient to conclude that *put* is valid because the *get* found satisfies both GETPUT and PUTGET. Furthermore, the constructed *get* is useful to confirm the initially expected view definition especially when they are not the same. For the case in which the expected view definition is not explicitly specified, the view definition is automatically derived.

In particular, we are given a putback program *putdelta*, which is written in nonrecursive Datalog with negation and built-in predicates, and maybe an expected view definition (*expected_get*) if it is explicitly described. The validation algorithm consists of three passes (see Figure 3.3): (1) checking the well-definedness of the putback program, (2) checking the existence of a view definition *get* satisfying GETPUT with the view update strategy *put* specified by the putback program and deriving *get*, and (3) checking whether *get* and *put* satisfy PUTGET. If one of the passes fails, we can conclude that *put* is invalid. Otherwise, *put* is valid because the derived *get* satisfies GETPUT and PUTGET with *put*.

3.3.2 Well-definedness

Consider a database schema $\mathcal{S} = \langle r_1, \ldots, r_n \rangle$ and a view v. Given a putback program *putdelta*, the goal is to check whether the delta ΔS resulting from *putdelta* is non-contradictory for any source database S and any view relation V. In other words, we check whether in ΔS, there is no pair of insertion and deletion, $+r_i(\vec{t})$ and $-r_i(\vec{t})$, of the same tuple $\vec{t}$ on the same relation R_i. To check this property, we add the following new rules to *putdelta*:

$$d_i(\vec{X_i}) :- +r_i(\vec{X_i}), -r_i(\vec{X_i}). \quad (i \in [1, n]) \tag{3.2}$$

The problem of checking whether ΔS is non-contradictory is reduced to the problem of checking whether each IDB predicate d_i in the Datalog program is unsatisfiable. When *putdelta* is in LVGN-Datalog, because each rule (3.2) is trivially negation guarded, according to Theorem 3.2.8, the satisfiability of d_i is decidable.

3.3.3 Existence of A View Definition Satisfying GetPut

Consider a view update strategy *put* specified by a putback program *putdelta* and a set of constraints Σ. Assume that *put* is valid. If an expected view definition *expected_get* is explicitly written by users, we check whether *expected_get* satisfies GetPut with *put*. With the view defined by *expected_get*, the GetPut property means that *put* makes no change to the source. Therefore, checking the GetPut property is reduced to checking the unsatisfiability of each delta relation in the Datalog program *putdelta*. This check is decidable if *expected_get* and *putdelta* are in LVGN-Datalog due to Theorem 3.2.8.

If *expected_get* is not explicitly written or if it does not satisfy GetPut, we construct a view definition *get* satisfying GetPut as follows. For each source database S, we find a steady-state view V such that the putback transformation *put* makes no change to the source database S. In other words, V must satisfy the constraints in Σ and $put(S, V) = S$. We define *get* as the mapping that maps each S to the V. If there exists an S such that we cannot find any steady-state view, then there is no view definition satisfying GetPut, and we conclude that *put* is invalid. Otherwise, the constructed *get* satisfies GetPut with *put*. Moreover, the view relation V resulting from *get* over S always satisfies Σ.

Example 3.3.2 (Intuition) *Consider the update strategy put in Example 3.2.1. For an arbitrary source database instance S, the goal is to find a steady-state view V such that $put(S, V) = S$, i.e., both of the source relations R_1 and R_2 are unchanged. Recall that the putback transformation put is described by Datalog rules that compute delta relations of each source relation R_1 and R_2. For R_1, we compute $\Delta^+_{R_1}$ and $\Delta^-_{R_1}$, which are the set of insertions and the set of deletions on R_1, respectively. R_1 is unchanged if all inserted tuples are already in R_1 and all deleted tuples are actually not in R_1. Similarly, for R_2, all tuples in $\Delta^-_{R_2}$ must be*

not in R_2 (we do not have $\Delta^+_{R_2}$). This leads to the following:

$$\begin{aligned} &\Delta^-_{R_1} \cap R_1 = \emptyset \\ &\Delta^-_{R_2} \cap R_2 = \emptyset \\ &\Delta^+_{R_1} \setminus R_1 = \emptyset \end{aligned} \tag{3.3}$$

Let us transform each delta predicate $-r_1$, $-r_2$, and $+r_1$ in the Datalog program putdelta to the form of relational calculus query [43]: $\varphi_{-r_1} = r_1(X) \wedge \neg v(X)$, $\varphi_{-r_2} = r_2(X) \wedge \neg v(X)$, $\varphi_{+r_1} = v(X) \wedge \neg r_1(X) \wedge \neg r_2(X)$. The constraint (3.3) is equivalent to the constraint that all the relational calculus queries $\varphi_{-r_1}(X) \wedge r_1(X)$, $\varphi_{-r_2}(X) \wedge r_2(X)$ and $\varphi_{+r_1}(X) \wedge \neg r_1(X)$ result in an empty set over the database (S, V) of both the source and view relations. In other words, (S, V) does not satisfy the following first-order sentences:

$$\begin{cases} (S,V) \not\models \exists X, \varphi_{-r_1}(X) \wedge r_1(X) \\ (S,V) \not\models \exists X, \varphi_{-r_2}(X) \wedge r_2(X) \\ (S,V) \not\models \exists X, \varphi_{+r_1}(X) \wedge \neg r_1(X) \end{cases}$$

By applying $\neg\exists X, \xi(X) \equiv \forall X, \xi(X) \rightarrow \bot$, we have

$$\begin{aligned} &\begin{cases} (S,V) \models \forall X, \varphi_{-r_1}(X) \wedge r_1(X) \rightarrow \bot \\ (S,V) \models \forall X, \varphi_{-r_2}(X) \wedge r_2(X) \rightarrow \bot \\ (S,V) \models \forall X, \varphi_{+r_1}(X) \wedge \neg r_1(X) \rightarrow \bot \end{cases} \\ \Leftrightarrow &(S,V) \models \begin{cases} \forall X, r_1(X) \wedge \neg v(X) \wedge r_1(X) \rightarrow \bot \\ \forall X, r_2(X) \wedge \neg v(X) \wedge r_2(X) \rightarrow \bot \\ \forall X, v(X) \wedge \neg r_1(X) \wedge \neg r_2(X) \wedge \neg r_1(X) \rightarrow \bot \end{cases} \end{aligned}$$

The idea for checking whether a view relation V satisfying the above logical sentences exists is that we swap the atom $v(X)$ appearing in these sentences to either the right-hand side or the left-hand side of the implication formula. For this purpose, we apply $p \wedge \neg q \rightarrow \bot \equiv p \rightarrow q$ and obtain:

$$\Leftrightarrow (S,V) \models \begin{cases} \forall X, r_1(X) \rightarrow v(X) \\ \forall X, r_2(X) \rightarrow v(X) \\ \forall X, v(X) \rightarrow \neg(\neg r_1(X) \wedge \neg r_2(X)) \end{cases}$$

By combining all sentences that have $v(X)$ on the right-hand side and combining all sentences that have $v(X)$ on the left-hand side, we obtain:

$$(S, V) \models \begin{cases} \forall X, r_1(X) \vee r_2(X) \rightarrow v(X) \\ \forall X, v(X) \rightarrow \neg(\neg r_1(X) \wedge \neg r_2(X)) \end{cases} \tag{3.4}$$

Note that S is an instance over $\langle r_1, r_2 \rangle$ and V is the view relation corresponding to predicate v. The first sentence provides us the lower bound V_{min} of V, which is the result of a first-order (FO) query[2] $\psi_1 = r_1(X) \vee r_2(X)$ over S. The second sentence provides us the upper bound V_{max} of V, which is the result of the first-order query $\psi_2 = \neg(\neg r_1(X) \wedge \neg r_2(X))$ over S. In fact, for each S, all the V such that $V_{min} \subseteq V \subseteq V_{max}$ satisfy (3.4), i.e., are steady-state instances of the view. Thus, a steady-state instance V exists if $V_{min} \subseteq V_{max}$. Indeed, by applying equivalence $\neg(p \vee q) \equiv \neg p \wedge \neg q$ to ψ_2, we obtain the same formula as ψ_1; hence, $\forall X, \psi_1(X) \rightarrow \psi_2(X)$ holds, leading to that $V_{min} \subseteq V_{max}$ holds. Now by choosing V_{min} as a steady-state view instance, we can construct a get as the mapping that maps each S to V_{min}. In other words, get is a query equivalent to the FO query ψ_1 over the source S. Since ψ_1 is a safe-range formula[3], we transform ψ_1 to an equivalent Datalog query[4] as follows:

$$v(X) :- r_1(X). \tag{3.5}$$

$$v(X) :- r_2(X). \tag{3.6}$$

This is the view definition get that satisfies GetPut *with the given view update strategy put.* □

Checking the existence of a steady-state view

In general, similar to the idea shown in Example 3.3.2, for an arbitrary putback program *putdelta* and a set of constraints Σ in LVGN-Datalog, we can always construct a guarded negation first-order (GNFO) sentence to check whether a

[2] A FO query ψ over D results in all tuples $\vec{t}$ s.t. $D \models \psi(\vec{t})$.

[3] ψ is a safe-range FO formula if all the variables in ψ are range restricted [43].

[4] Due to the equivalence between nonrecursive Datalog queries and safe-range FO formulas [43].

steady-state view V satisfying Σ and $put(S, V) = S$ (i.e., $S \oplus putdelta(S, V) = S$) exists.

Lemma 3.3.3 *Given a LVGN-Datalog putback program putdelta and a set of guarded negation constraints Σ, there exist first-order formulas ϕ_1, ϕ_2, ϕ_3 such that for a given database instance S, a view relation V satisfies Σ and $S \oplus putdelta(S, V) = S$ iff*

$$\begin{cases} (S, V) \models \forall \vec{Y}, v(\vec{Y}) \wedge \phi_1(\vec{Y}) \rightarrow \bot \\ (S, V) \models \forall \vec{Y}, \neg v(\vec{Y}) \wedge \phi_2(\vec{Y}) \rightarrow \bot \\ (S, V) \not\models \phi_3 \end{cases} \tag{3.7}$$

where v is the predicate corresponding to the view relation V and ϕ_1, ϕ_2, ϕ_3 have no occurrence of the view predicate v. Both $\phi_2(\vec{Y})$ and ϕ_3 are safe-range GNFO formulas, and $v(\vec{Y}) \wedge \phi_1(\vec{Y})$ is equivalent to a GNFO formula. □

The third constraint $(S, V) \not\models \phi_3$ in (3.7) is simplified to $S \not\models \phi_3$ because the FO sentence ϕ_3 has no atom of v as a subformula. This means that ϕ_3 must be unsatisfiable over any database S. Since ϕ_3 is a GNFO sentence, we can check whether ϕ_3 is satisfiable. If it is satisfiable, we conclude that the view relation V does not exist; thus, *put* is invalid.

For the two other constraints in (3.7), by applying the logical equivalence $p \wedge \neg q \rightarrow \bot \equiv p \rightarrow q$, we have:

$$\begin{cases} (S, V) \models \forall \vec{Y}, v(\vec{Y}) \rightarrow \neg \phi_1(\vec{Y}) \\ (S, V) \models \forall \vec{Y}, \phi_2(\vec{Y}) \rightarrow v(\vec{Y}) \end{cases} \tag{3.8}$$

Because ϕ_1 and ϕ_2 do not contain an atom of v as a subformula, there exists an instance V if

$$\begin{aligned} & S \models \forall \vec{Y}, \phi_2(\vec{Y}) \rightarrow \neg \phi_1(\vec{Y}) \\ \Leftrightarrow & S \models \forall \vec{Y}, \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y}) \rightarrow \bot \end{aligned}$$

This means that the sentence $\exists \vec{Y}, \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y})$ is not satisfiable. In this way, checking the existence of a V is now reduced to checking the satisfiability of

$\exists\vec{Y}, \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y})$. The idea of checking the satisfiability of $\exists\vec{Y}, \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y})$ is to reduce this problem to that of a GNFO sentence. For this purpose, by introducing a fresh relation r of an appropriate arity, we have the fact that $\exists\vec{Y}, \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y})$ is satisfiable if and only if $\exists\vec{Y}, r(\vec{Y}) \wedge \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y})$ is satisfiable. Because $v(\vec{Y}) \wedge \phi_1(\vec{Y})$ is equivalent to a GNFO formula, $r(\vec{Y}) \wedge \phi_1(\vec{Y})$ is also equivalent to a GNFO formula. On the other hand, $\phi_2(\vec{Y})$ is equivalent to a GNFO formula; hence, we can transform $\exists\vec{Y}, r(\vec{Y}) \wedge \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y})$ into an equivalent GNFO sentence whose satisfiability is decidable [59].

Constructing a view definition

If both ϕ_3 and $\exists\vec{Y}, \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y})$ are unsatisfiable, there exists a steady-state view V satisfying Σ such that $S \oplus \mathit{putdelta}(S, V) = S$ for each database S. One steady-state view V is the one resulting from the FO formula ϕ_2 over S. Indeed, such a V satisfies (3.8); hence, it satisfies Σ and $S \oplus \mathit{putdelta}(S, V) = S$. By choosing this steady-state view, we can construct a view definition *get* as the Datalog query equivalent to ϕ_2 because ϕ_2 is a safe-range formula. The equivalence of safe-range first-order logic and Datalog was well studied in database theory [43, 44]. We present the detailed transformation from safe-range FO formula to Datalog query in Ex A.2. Due to Lemma 3.3.3, ϕ_2 is also negation guarded and hence, *get* is in nonrecursive GN-Datalog with equalities, constants and comparisons.

3.3.4 The PutGet Property

To check the PutGet property that $\mathit{get}(\mathit{put}(S, V)) = V$ for any S and V, we first construct a Datalog query over database (S, V) equivalent to the composition $\mathit{get}(\mathit{put}(S, V))$. Recall that $\mathit{put}(S, V) = S \oplus \mathit{putdelta}(S, V)$. The result of $\mathit{put}(S, V)$ is a new source S' obtained by applying ΔS computed from *putdelta* to the original source S. Let us use predicate r_i^{new} for the new relation of predicate r_i in S after the update. The result of applying a delta ΔS to the database S is equivalent to the result of the following Datalog rules ($i \in [1, n]$):

$r_i^{new}(\vec{X}_i)$:- $r_i(\vec{X}_i)$, ¬ -$r_i(\vec{X}_i)$.
$r_i^{new}(\vec{X}_i)$:- +$r_i(\vec{X}_i)$.

By adding these rules to the Datalog putback program *putdelta*, we derive a new Datalog program, denoted as *newsource*, that results in a new source database. The result of $get(put(S, V))$ is the same as the result of the Datalog query *get* over the new source database computed by the program *newsource*. Therefore, we can substitute each EDB predicate r_i in the program *get* with the new program r_i^{new} and then merge the obtained program with the program *newsource* to obtain a Datalog program, denoted as *putget*. The result of *putget* over (S, V) is exactly the same as the result of $get(put(S, V))$. For example, the Datalog program *putget* for the view update strategy in Example 3.3.2 is:

$$
\begin{aligned}
&\text{-}r_1(X) &&:- \ r_1(X), \ \neg \ v(X).\\
&\text{-}r_2(X) &&:- \ r_2(X), \ \neg \ v(X).\\
&\text{+}r_1(X) &&:- \ v(X), \ \neg \ r_1(X), \ \neg \ r_2(X).\\
&r_1^{new}(X) &&:- \ r_1(X), \ \neg \ \text{-}r_1(X).\\
&r_1^{new}(X) &&:- \ \text{+}r_1(X).\\
&r_2^{new}(X) &&:- \ r_2(X), \ \neg \ \text{-}r_2(X).\\
&v^{new}(X) &&:- \ r_1^{new}(X).\\
&v^{new}(X) &&:- \ r_2^{new}(X).
\end{aligned}
$$

Checking the PutGet property is now reduced to checking whether the result of Datalog query *putget* over database (S, V) is the same as the view relation V. By transforming *putget* to the FO formula $\phi_{putget}(\vec{Y})$, we reduce checking the PutGet property to checking the satisfiability of the two following sentences:

$$\Phi_1 = \exists \vec{Y}, \phi_{putget}(\vec{Y}) \wedge \neg v(\vec{Y}) \tag{3.9}$$

$$\Phi_2 = \exists \vec{Y}, v(\vec{Y}) \wedge \neg \phi_{putget}(\vec{Y}) \tag{3.10}$$

The PutGet property holds if and only if Φ_1 and Φ_2 are not satisfiable. Clearly, if *get* and *putdelta* are in LVGN-Datalog, *putget* is also in LVGN-Datalog, leading to that $\phi_{putget}(\vec{Y})$ is a GNFO formula. Therefore, Φ_2 is a GNFO sentence; hence, its satisfiability is decidable. Φ_1 is satisfiable if and only if $\Phi_1' = \exists \vec{Y}, \phi_{putget}(\vec{Y}) \wedge r(\vec{Y}) \wedge \neg v(\vec{Y})$ is satisfiable, where r is a fresh relation of an appropriate arity. Since Φ_1' is a guarded negation first-order sentence, its satisfiability is decidable by Theorem 3.2.8.

Algorithm 1: VALIDATE(*expected_get*, *putdelta*, Σ)

```
1  get ← null;
   // Checking the well-definedness of putdelta
2  check if all predicates d_i (i ∈ [1, n]) in (3.2) are unsatisfiable under Σ;
3  if expected_get is not null then
       // Checking if expected_get satisfies GETPUT
4      if all delta relations of putdelta are unsatisfiable under Σ with the view
         defined by expected_get then
5          get ← expected_get;
6  if (expected_get is null) or (get is null) then
       // Constructing a get satisfying GETPUT
7      check if φ_3 in (3.7) is unsatisfiable under Σ;
8      check if ∃Ȳ, φ_1(Ȳ) ∧ φ_2(Ȳ) (φ_1 and φ_2 in (3.8)) is unsatisfiable under Σ;
       // Constructing a get
9      get ← Translating FO formula φ_2 in (3.8) to an equivalent Datalog
         query;
   // Checking the PUTGET property
10 check if Φ_1 and Φ_2 in (3.9) and (3.10) are unsatisfiable under Σ;
11 return get;
```

3.3.5 Soundness and Completeness

Algorithm 1 summarizes the validation of Datalog putback programs *putdelta*. After all the checks have passed, the corresponding view definition is returned and *putdelta* is valid. For LVGN-Datalog in which the query satisfiability is decidable (Theorem 3.2.8), Algorithm 1 is sound and complete.

Theorem 3.3.4 (Soundness and Completeness)

- *If a LVGN-Datalog putback program putdelta passes all the checks in Algorithm 1, putdelta is valid.*
- *Every valid LVGN-Datalog putback program putdelta passes all the checks in Algorithm 1.*

□

It is remarkable that if *putdelta* is not in LVGN-Datalog, but in nonrecursive Datalog with unrestricted negation and built-in predicates, we can still perform

the checks in the validation algorithm by feeding them to an automated theorem prover. Though, Algorithm 1 may not terminate and not successfully construct the view definition *get* because of the undecidability problem [43, 60]. Therefore, Algorithm 1 is sound for validating the pair of *putdelta* and *expected_get* that once it terminates, we can conclude *putdelta* is valid.

3.4 Incrementalization

We have shown that an updatable view is defined by a valid *put*, which makes changes to the source to reflect view updates. However, when there is only a small update on the view, repeating the *put* computation is not efficient. In this section, we further optimize the computation of the putback program by exploiting its well-behavedness and integrating it with the standard incrementalization method for Datalog.

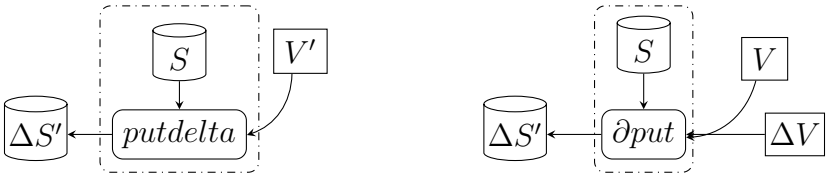
Consider the steady state before a view update in which both the source and the view are unchanged; due to the GetPut property, a valid *putdelta* results in a ΔS having no effect on the original source S: $S \oplus \Delta S = S$. This means that ΔS can be either an empty set or a nonempty set in which all deletions in ΔS are not yet in the original source S and all insertions in ΔS are already in S. If the view is updated by a delta ΔV, there will be some changes to ΔS, denoted as $\Delta^2 S$, that have effects on the original source S.

Example 3.4.1 *Consider the database in Example 3.2.1:* $S = \{r_1(1), r_2(2), r_2(4)\}$. *Let* $\Delta S = \{+r_1(1), +r_2(2), -r_2(3)\}$ *be a delta of* S. *Clearly,* $S \oplus \Delta S = S$. *Now, we change* ΔS *by a delta of* ΔS, *denoted as* $\Delta^2 S$, *which includes a set of deletions to* ΔS, $\Delta^{2-} S = \{+r_1(1), -r_2(3)\}$, *and a set of insertions to* ΔS, $\Delta^{2+} S = \{+r_1(3), -r_2(4)\}$. *We obtain a new delta of* S*:*

$$\Delta S' = (\Delta S \setminus \Delta^{2-} S) \cup \Delta^{2+} S = \{+r_1(3), +r_2(2), -r_2(4)\}$$

and the new database $S' = S \oplus \Delta S' = \{r_1(1), r_1(3), r_2(2)\}$. *In fact, we can also obtain the same* S' *by applying only* $\Delta^{2+} S$ *directly to* S*:* $S' = S \oplus \Delta^{2+} S$. □

Intuitively, for each base relation R_i in the source database S, we obtain the



(a) Original *putdelta* (b) Incremental ∂put

Figure 3.4: Incrementalization of *putdelta*.

new R_i' by applying to R_i the delta relations $\Delta^-_{R_i}$ and $\Delta^+_{R_i}$ from ΔS. Because all the tuples in $\Delta^-_{R_i}$ are not in R_i and all the tuples in $\Delta^+_{R_i}$ are in R_i, if we remove some tuples from $\Delta^-_{R_i}$ or $\Delta^+_{R_i}$, then the result R_i' has no change. Only the tuples inserted into $\Delta^-_{R_i}$ or $\Delta^+_{R_i}$ make some changes in R_i'. Therefore, S' can be obtained by applying to the original S the part $\Delta^{2+}S$ of $\Delta^2 S$, i.e., $\Delta S'$ and $\Delta^{2+}S$ are interchangeable.

Proposition 3.4.2 *Let S be a database and ΔS be a non-contradictory delta of the database S such that $S \oplus \Delta S = S$. Let $\Delta^2 S$ be a delta of ΔS, and the following equation holds:*

$$S' = S \oplus \Delta S' = S \oplus \Delta^{2+}S$$

where $\Delta S' = \Delta S \oplus \Delta^2 S$ and $\Delta^{2+}S$ is the set of new tuples inserted into ΔS by applying $\Delta^2 S$. □

Proposition 3.4.2 is the key observation for deriving from *putdelta* an incremental Datalog program ∂put that computes ΔS more efficiently (Figure 3.4). To derive ∂put, we first incrementalize the Datalog program *putdelta* to obtain Datalog rules that compute $\Delta^2 S$ from the change ΔV on the view V. This step can be performed using classical incrementalization methods for Datalog [55]. We then use $\Delta^{2+}S$ in $\Delta^2 S$ as an instance of ΔS for applying to the source S.

Example 3.4.3 (Intuition) *Given a source relation R of arity 2 and a view relation V defined by a selection on R: $v(X,Y) :- r(X,Y), Y > 2$. Consider the following update strategy with a constraint that updates on V must satisfy the*

selection condition $Y > 2$*:*

$$\begin{aligned} +r(X,Y) &:- v(X,Y), \neg r(X,Y). \\ m(X,Y) &:- r(X,Y), Y > 2. \\ -r(X,Y) &:- m(X,Y), \neg v(X,Y). \end{aligned}$$

Let Δ_V^+/Δ_V^- *be the set of insertions/deletions into/from the view* V*. We use two predicates* $+v$ *and* $-v$ *for* Δ_V^+ *and* Δ_V^-*, respectively. To generate delta rules for computing changes of* $\pm r$ *when the view is changed by* Δ_V^+ *and* Δ_V^-*, we adopt the incremental view maintenance techniques introduced in [55] but in a way that derives rules for computing the insertion set and deletion set for* $\pm r$ *separately. When* Δ_V^+ *and* Δ_V^- *are disjoint, by applying distribution laws for the first Datalog rule, we derive two rules that define the changes to* Δ_R^+*, a set of insertions* $\Delta^+(\Delta_R^+)$ *and a set of deletions* $\Delta^-(\Delta_R^+)$*, as follows:*

$$\begin{aligned} +(+r)(X,Y) &:- +v(X,Y), \neg r(X,Y). \\ -(+r)(X,Y) &:- -v(X,Y), \neg r(X,Y). \end{aligned}$$

where predicates $+(+r)$ *and* $-(+r)$ *correspond to* $\Delta^+(\Delta_R^+)$ *and* $\Delta^-(\Delta_R^+)$*, respectively. Similarly, we derive rules defining changes to* Δ_R^-*,* $\Delta^+(\Delta_R^-)$ *and* $\Delta^-(\Delta_R^-)$*, as follows:*

$$\begin{aligned} +(-r)(X,Y) &:- m(X,Y), -v(X,Y). \\ -(-r)(X,Y) &:- m(X,Y), +v(X,Y). \end{aligned}$$

Finally, as stated in Proposition 3.4.2, $\Delta^{2+}S$ *and* $\Delta S'$ *are interchangeable. Since* $\Delta^{2+}S$ *contains* $\Delta^+(\Delta_R^-)$ *and* $\Delta^+(\Delta_R^+)$*, we can substitute* $-r$ *and* $+r$ *for the predicates* $+(-r)$ *and* $+(+r)$*, respectively, to derive the program* ∂put *as follows:*

$$\begin{aligned} m(X,Y) &:- r(X,Y), Y > 2. \\ +r(X,Y) &:- +v(X,Y), \neg r(X,Y). \\ -r(X,Y) &:- m(X,Y), -v(X,Y). \end{aligned}$$

Because Δ_V^+ *and* Δ_V^- *are generally much smaller than the view* V*, the computation of* $\Delta^+(\Delta_R^\pm)$ *in the derived rules is more efficient than the computation of* $\Delta_R^\pm$ *in putdelta.* □

The incrementalization algorithm that transforms a putback program *putdelta* in nonrecursive Datalog with negation and built-in predicates into an equivalent program ∂put is as follows:

- *Step 1*: We first stratify the Datalog program *putdelta*. Let v, $l_1, \ldots, l_m$, $\pm r_1, \ldots \pm r_n$ be a stratification [41] of the Datalog program *putdelta*, which is an order for the evaluation of IDB relations of *putdelta*.

- *Step 2*: To derive rules for computing changes of each IDB relation $l_1, \ldots, l_m$ when the view v is changed, we adopt the incremental view maintenance techniques introduced in [55] but in a way that derives rules for computing each insertion set $(+l_i)$ and deletion set $(-l_i)$ on IDB relation l_i $(i \in [1, m])$ separately (see the details in Ex A.3).

- *Step 3*: Similar to *Step 2*, we continue to derive rules for computing changes of each IDB relation $\pm r_1, \ldots \pm r_n$ but only for insertions to these relations. The purpose is to generate rules for computing $\Delta^{2+}S$, i.e., computing the relations $+(\pm r_1), \ldots + (\pm r_n)$.

- *Step 4*: We finally substitute $\pm r_i$ for $+(\pm r_i)$ $(i \in [1, n])$ in the derived rules to obtain the incremental program ∂put. This is because $\Delta^{2+}S$ can be used as a instance of $\Delta S'$ to apply to the source database S (Proposition 3.4.2).

As shown in Example 3.4.3, for a LVGN-Datalog program in which the view predicate v occurs at most once in each delta rule, the transformation from a putback program *putdelta* to an incremental one ∂put is simplified to substituting $+v$ for positive predicate v and $-v$ for negative predicate $\neg v$.

Lemma 3.4.4 *Every valid LVGN-Datalog putback program putdelta for a view relation V is equivalent to an incremental program that is derived from putdelta by substituting delta predicates of the view, $+v$ and $-v$, for positive and negative predicates of the view, v and $\neg v$, respectively.* □

3.5 Implementation and Evaluation

3.5.1 Implementation

We have implemented a prototype for our proposed validation and incrementalization algorithms in Ocaml (The full source code is available at https://github.com/dangtv/BIRDS). For the case in which the view update strategy is not in LVGN-Datalog, our framework feeds each check in our validation algorithm to the Z3 automated theorem prover [61]. As mentioned in Subsection 3.3.5, the validation algorithm may not terminate, though it is sound for checking the pair of view definition and update strategy program. We have also integrated our framework with PostgreSQL [26], a commercial RDBMS, by translating both the view definition and update strategy in Datalog to equivalent SQL and trigger programs.

Our translation is conducted because nonrecursive Datalog queries can be expressed in SQL [43]. We use a similar approach to the translation from Datalog to SQL used in [46]. The SQL view definition is of the form `CREATE VIEW <view-name> AS <sql-defining-query>`. Meanwhile, the implementation for the update strategy is achieved by generating a SQL program that defines triggers [62] and associated trigger procedures on the view. These trigger procedures are automatically invoked in response to view update requests, which can be any SQL statements of `INSERT`/`DELETE`/`UPDATE`. Our framework also supports combining multiple SQL statements into one transaction to obtain a larger modification request on the view. When there are view update requests, the triggers on the view perform the following steps: (1) handling update requests to the view to derive deltas of the view (see Ex A.4), (2) checking the constraints if applying the deltas from step (1) to the view, and (3) computing each delta relation and applying them to the source. The main trigger is as follows:

```
CREATE TRIGGER <update-strategy>
INSTEAD OF INSERT OR UPDATE OR DELETE ON <view V>
BEGIN
  -- Deriving changes on the view
  Derive Δ_V^- and Δ_V^+ from view update requests
  -- Checking constraints
```

```
  FOR EACH <constraint ∀X⃗, Φ_i(X⃗) → ⊥> DO
    IF EXISTS (<SQL-query-of Φ_i(X⃗)>) THEN
      RAISE "Invalid view updates";
    END IF;
  END FOR;
  -- Calculating and applying delta relations
  FOR EACH <source relation R_i> DO
    CREATE TEMP TABLE Δ+_{R_i} AS <sql-query-of +r_i>;
    CREATE TEMP TABLE Δ-_{R_i} AS <sql-query-of −r_i>;
    DELETE FROM R_i WHERE ROW (R_i) IN Δ-_{R_i};
    INSERT INTO R_i SELECT * FROM Δ+_{R_i};
  END FOR;
END;
```

3.5.2 Evaluation

To evaluate our approach, we conduct two experiments. The goal of the first experiment is to investigate the practical relevance of our proposed method in describing view update strategies and to evaluate the performance of our framework in checking these described update strategies. In the second experiment, we study the efficiency of our incrementalization algorithm when implementing updatable views in a commercial RDBMS.

Benchmarks

To perform the evaluation, we collect benchmarks of views and update strategies from two different sources:

- View update examples and exercises collected from the literature: textbooks [62, 63], online tutorials [64, 65, 66, 67, 68] (triggers, sharded tables, and so forth), papers [20, 13] and our case study in Section 3.2.

- View update issues asked on online question & answer sites: Database Administrators Stack Exchange [69] and Stack Overflow Public Q&A [70].

Table 3.1: Validation results. S, P, SJ, IJ, LJ, RJ, FJ, U, D and A stand for selection, projection, semi join, inner join, left join, right join, full join, union, set difference and aggregation, respectively. PK, FK, ID, and C stand for primary key, foreign key, inclusion dependency, and domain constraint, respectively.

	ID	View	Operator in view definition	Program size (LOC)	Constraint	LVGN-Datalog	NR-Data-log$^{\neg,=,<}$	Validation Time (s)	SQL (Byte)
Literature	1	car_master	P	4		✓	✓	1.74	8447
	2	goodstudents	P,S	5	C	✓	✓	1.86	9182
	3	luxuryitems	S	5	C	✓	✓	1.77	8938
	4	usa_city	P,S	5	C	✓	✓	1.77	9059
	5	ced	D	6		✓	✓	1.72	8847
	6	residents1962	S	6	C	✓	✓	1.73	9699
	7	employees	SJ,P	6	ID	✓	✓	1.76	9358
	8	researchers	SJ,S,P	6		✓	✓	1.79	9058
	9	retired	SJ,P,D	6		✓	✓	1.76	9048
	10	paramountmovies	P,S	7		✓	✓	1.81	9721
	11	officeinfo	P	7		✓	✓	1.8	9963
	12	vw_brands	U,P	8	C	✓	✓	1.78	10932
	13	tracks2	P	8		✓	✓	1.81	9824
	14	residents	U	10		✓	✓	1.77	13504
	15	tracks3	S	11	C	✓	✓	1.88	14430
	16	tracks1	IJ	12	PK	✗	✓	1.92	95606
	17	bstudents	IJ,P,S	13	PK	✗	✓	2.13	22431
	18	all_cars	IJ	13	PK, FK	✗	✓	1.89	25013
	19	measurement	U	13	C, ID	✓	✓	1.78	12624
	20	newpc	IJ,P,S	15	JD	✗	✓	2.06	44665
	21	activestudents	IJ,P,S	19	PK, JD	✗	✓	2.19	31766
	22	vw_customers	IJ,P	19	PK,FK,JD	✗	✓	2.92	26286
	23	emp_view	IJ,P,A	-		✗	✗	-	-
Q&A sites	24	ukaz_lok	S	6	C	✓	✓	1.79	10104
	25	message	U	8	C	✓	✓	1.8	15770
	26	outstanding_task	P, SJ	10	ID, C	✓	✓	10.07	18253
	27	poi_view	P,IJ	12	PK	✗	✓	2.1	24741
	28	phonelist	U	14	C	✓	✓	1.94	16553
	29	products	LJ	16	PK,FK,C	✗	✓	3.6	58394
	30	koncerty	IJ	17	PK	✗	✓	1.93	29147
	31	purchaseview	P,IJ	19	PK,FK,JD	✗	✓	1.89	27262
	32	vehicle_view	P,IJ	20	PK,FK,JD	✗	✓	2.03	25226

All experiments on these benchmarks are run using Ubuntu server LTS 16.04 and PostgreSQL 9.6 on a computer with 2 CPUs and 4 GB RAM.

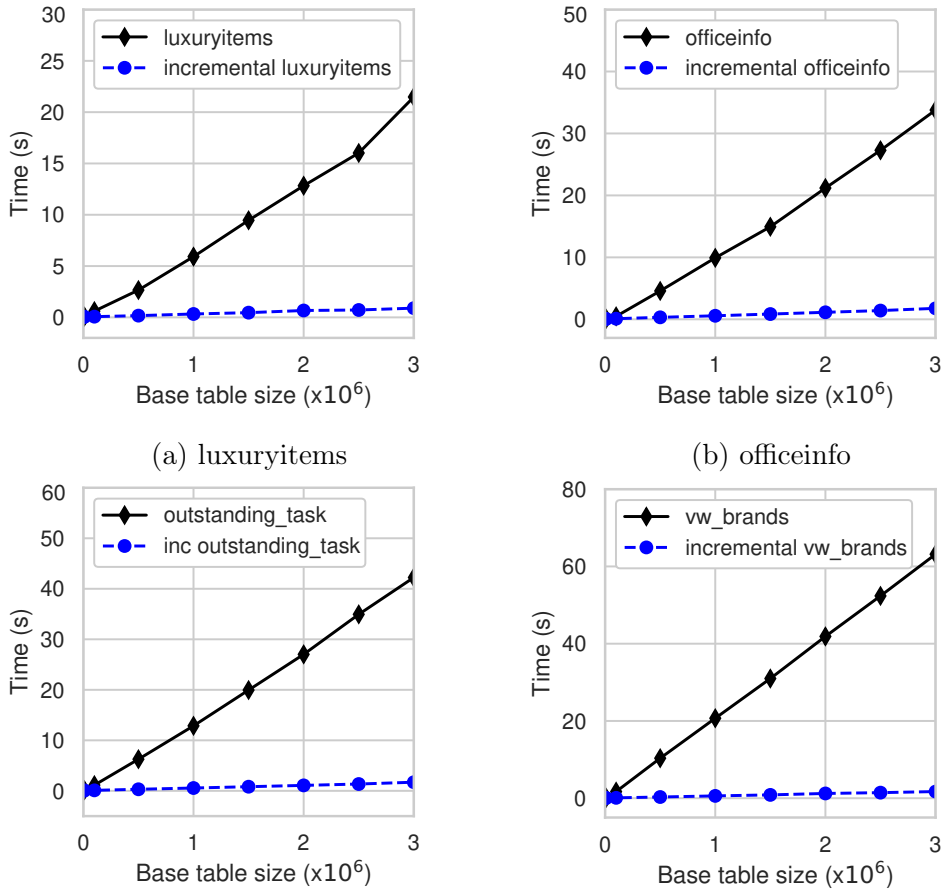
Results

As mentioned previously, we perform the first experiment to investigate which users' update strategies are expressible and validatable by our approach. In our benchmarks, the collected view update strategies are either implemented in SQL triggers or naturally described by users/systems. We manually use nonrecursive Datalog with negation and built-in predicates (NR-Datalog$^{\neg,=,<}$) to specify these update strategies as *putdelta* programs[5] and input them with the expected view definition to our framework. Table 3.1 shows the validation results. In terms of expressiveness, NR-Datalog$^{\neg,=,<}$ can be used to formalize most of the view update strategies with many common integrity constraints except one update strategy for the aggregation view `emp_view` (#23). This is because we have not considered aggregation in Datalog. Interestingly, LVGN-Datalog can also express many update strategies for many views defined by selection, projection, union, set difference and semi join. Inner join views such as `all_car` (#18) are not expressible in LVGN-Datalog because the definition of inner join is not in guarded negation Datalog[6]. LVGN-Datalog is also limited in expressing primary key (functional dependency) or join dependency because these dependencies are not negation guarded[7]. Even for the cases that LVGN-Datalog cannot express, thus far, all the well-behavedness checks in our experiment terminate after an acceptable time (approximately a few seconds). The validation time almost increases with the number of rules in the Datalog programs (program size), but this time also depends on the complexity of the source and view schema. For example, the update strategy of `message` (#25) has the longest validation time because this view and its source relations have many more attributes than other views. Similarly, the size of the generated SQL program is larger for the more complex Datalog update strategies.

[5]For the update strategies implemented in SQL triggers, rewriting them into *putdelta* programs can be automated.

[6]An example of inner join is $v(X, Y, Z) :- s_1(X, Y), s_2(Y, Z)$, which is not a guarded negation Datalog rule.

[7]Primary key A on relation $r(A, B)$ is expressed by the rule $\bot :- r(A, B_1), r(A, B_2), \neg B_1 = B_2$, where the equality $B_1 = B_2$ is not guarded.



(a) luxuryitems (b) officeinfo

(c) outstanding_task (d) vw_brands

Figure 3.5: View updating time.

We perform the second experiment to evaluate the efficiency of the incrementalization algorithm in optimizing view update strategies. Specifically, we compare the performance of the incrementalized update strategy with the original one when they are translated into SQL trigger programs and run in PostgreSQL database. For this experiment, we select some typical views in our benchmarks including: `luxuryitems` (Selection), `officeinfo` (Projection), `outstanding_task` (Join) and `vw_brands` (Union). For each view, we randomly generate data for the base tables and measure the running time of the view update strategy against the base table size (number of tuples) when there is an SQL statement that attempts to modify the view. Figure 3.5 shows the comparison between the original view

update strategies (black lines) and the incrementalized ones (blue lines). It is clear that as the size of the base tables increases, our incrementalization significantly reduces the running time to a constant value, thereby improving the performance of the view update strategies.

3.6 Related work

Melnik et al. [71] propose a novel declarative mapping language for specifying the relationship between application entity views and relational databases, which is compiled into bidirectional views for the view update translation. The user-specified mappings are validated to guarantee the generated bidirectional views to roundtrip. Furthermore, the authors introduce the concept of merge views that together with the bidirectional views contribute to determining complete update strategies, thereby solving the ambiguity of view updates. Though, merge views are exclusively used and validating the behavior of this operation with respect to the roundtripping criterion is not explicitly considered. In comparison to [71], where the proposed mapping language is restricted to selection-projection views (no joins), our approach focuses on a specification language, which is in lower level but more expressive that more view update strategies can be expressed. Moreover, the full behaviour of the specified view update strategies is validated by our approach.

3.7 Conclusion

In this chapter, we have introduced a novel approach for relational view update in which programmers are given full control over deciding and implementing their view update strategies. We have shown that a view update strategy can be concisely written in a Datalog program that computes source updates from the updated view and the original state of the source tables. In this chapter, we focus on non-recursive Datalog that is expressive enough for practical relational database management systems where a view and its update strategy are commonly defined in SQL without recursion.

Since writing view update strategies is error-prone, we propose a validation algorithm that statically checks the well-behavedness of user-written programs. We identify a fragment of Datalog, called linear-view guarded negation Datalog (LVGN-Datalog), in which our validation algorithm is both sound and complete. Furthermore, the algorithm can automatically derive from view update strategies the corresponding view definition to confirm the one expected beforehand. This fragment not only has good properties in theory but is also useful for solving practical view updates. Our experiments on benchmarks collected in practice show that our validation algorithm is feasible and LVGN-Datalog is expressive enough for solving many view update strategies.

To improve the performance of view update strategy programs, we introduce a new optimization method by incrementalizing the hand-written programs. The optimization algorithm integrates the standard incrementalization method for Datalog with the well-behavedness in view updates. The experiments show that our incrementalization can significantly reduce the running time of view updates in practical relational database management systems.

In this chapter, we consider a core fragment of Datalog for programming view update strategies. In our feature work, more extensions of the Datalog language will be considered with extended versions of both the validation and incrementalization algorithms.

The current implementation of our framework is integrated with PostgreSQL and supports only virtual views in this database management system. As presented in this chapter, our compiler translates Datalog programs into SQL code that consists of all necessary statements for creating views with associated triggers for view updates. By using the same trigger mechanisms for updatable views, it is straightforward to extend the framework to integrate with other database management systems such as MySQL, Oracle, and so forth. The designed triggers in our implementation can be also adapted to work with materialized views.

As mentioned in this chapter, our validation statically checks the well-behavedness of user-written programs so that programmers can early detect errors in their programs before executing in real databases. How programmers can find the details of bugs remains unclear. Therefore, it is important to further support programmers or the case that the programs are not valid. In the next chapter, we

shall present our approach to interactively debugging Datalog programs so that the user's burden is reduced.

4

A Debugger for Non-Recursive View Update Strategies

4.1 Introduction

Datalog, a declarative logic programming language, has many applications in a variety of domains such as deductive databases [58], data integration [72], program analysis [73, 74], bidirectional programming [36], and so forth. Verifying Datalog programs plays an essential role to guarantee the properties of these programs required by the applications. When a property is not satisfied, it is more important to reduce the user's burden in debugging the unexpected behavior of the program.

This kind of debugging problem, which arises when a property of a program is not satisfied, has not been well studied for Datalog. There are two challenges in practice. The first challenge is searching for a concrete input database, i.e., a counterexample that reveals the unexpected behavior of the program. The second challenge is locating the buggy Datalog rules that break the property. By

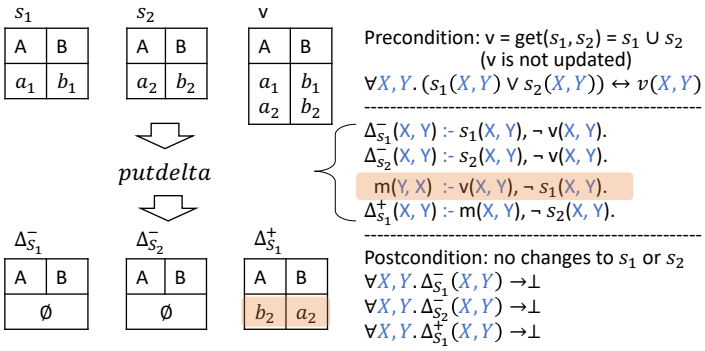


Figure 4.1: Motivating example. The unexpected tuple and the buggy rule are highlighted.

adopting the algorithmic debugging method [75], a few approaches were proposed for debugging Datalog programs [76, 77, 78]. However, the existing approaches neither provide users a way to specify the properties of Datalog programs nor generate counterexamples to show the incorrectness of the programs. To locate a bug, these approaches ask the users many questions about the computation correctness of the Datalog program. In other words, the users have to find out whether the Datalog program has unintended interpretations, e.g., the intention is not met by the program results. Identifying such unintended interpretations becomes costly when the input database of the program is not small.

An ideal approach to debugging would allow the user to specify the program's properties and automatically run all the checks. The properties of a program are commonly specified by a set of assertions such as equalities, domain constraints, containments, and so forth. For Datalog, which is a logic programming language in relational databases, it is intuitive for programmers to specify the assertions in the forms of relational predicates. For example, one may consider that some relations of the Datalog program must be equivalent or some relations must be empty, i.e., the corresponding predicates are always false.

We illustrate with the following example the property specifications and the debugging problem of Datalog programs.

Example 4.1.1 (Motivating Example: View Update Strategy) *In this example, we consider an application of Datalog in describing view update strategies. Suppose that we are given a database of two base relations* $s_1(A, B)$ *and* $s_2(A, B)$

(Figure 4.1) with a view $v(A, B)$ defined over these two relations by a union query: $v = get(s_1, s_2) = s_1 \cup s_2$. The following is a buggy Datalog program (denoted as putdelta) that describes a view update strategy, i.e., a description about how to update the base relations s_1 and s_2 through the view v.

$$\Delta^-_{s_1}(X, Y) :- s_1(X, Y), \neg v(X, Y). \quad \text{(r1)}$$

$$\Delta^-_{s_2}(X, Y) :- s_2(X, Y), \neg v(X, Y). \quad \text{(r2)}$$

$$m(Y, X) :- v(X, Y), \neg s_1(X, Y). \quad \text{(r3)}$$

$$\Delta^+_{s_1}(X, Y) :- m(X, Y), \neg s_2(X, Y). \quad \text{(r4)}$$

In putdelta, for a relation, Δ^+ and Δ^- denote the insertion and deletion sets on the relation, respectively. Rules (r1) and (r2) state that if a tuple $\langle X, Y\rangle$ is in s_1 or s_2 but not in v, it will be deleted from s_1 or s_2, respectively. Rule (r3) checks the tuples in v but not in s_1, and stores these tuples in a mediate relation m. The last rule states that if a tuple $\langle X, Y\rangle$ is in m but not in s_2, it will be inserted into s_1. putdelta takes as input the states of s_1, s_2, and v to produce the delta relations of s_1 and s_2.

Such a putback program putdelta is required to satisfy round-tripping properties to maintain the consistency of view updates, as formulated in the existing works [29, 36]. Here, we illustrate the problem with the property (called GetPut*) that in the input of putdelta, if the view is unchanged, i.e., $v = s_1 \cup s_2$, the output of putdelta must be empty. We use first-order logic sentences (Figure 4.1) to specify the constraints of the input (called precondition) and the constraints over the output (called postcondition).*

Figure 1 shows a counterexample of GetPut *that is a collection of tuples in the source tables and the view (s_1, s_2, v). Over this counterexample, the result of putdelta is $\Delta^-_{s_1} = \Delta^-_{s_2} = \emptyset$ and $\Delta^+_{s_1} = \{\langle b_2, a_2\rangle\}$. That means tuple $\langle b_2, a_2\rangle$ is inserted into s_1. This insertion is not expected by the postcondition. Since the input of putdelta satisfies the precondition but the output does not satisfy the postcondition, the* GetPut *property of putdelta is violated.*

The user may wonder why tuple $\langle b_2, a_2\rangle$ of $\Delta^+_{s_1}$ occurs unexpectedly in the output of putdelta. From this unexpected tuple, the problem now is to detect which rules in the original Datalog program are the causes. Here, in the head of rule (r3),

the variables X and Y are placed in the wrong positions, and thereby some wrong tuples are derived. This bug must be fixed to make putdelta satisfy the GetPut *property.* □

We believe that for a required property of a Datalog program, the user may not only have unexpected mistakes such as typos but also have wrong intentions that do not conform to the property. Providing suggestions on how to correct the program is very useful to users but is a challenging issue. In addition, debugging is an ambiguous process that there are many possible causes for a bug. Therefore, it is essential to design an interface that lets users interact with the underlying debugging engine. For example, the user can mark suspicious rules to inspect or decide how to proceed for the bug ambiguity.

The key insight of this chapter is that counterexamples play a central role in debugging Datalog programs. First, a program is buggy if and only if a counterexample exists. Second, to be useful for debugging the Datalog program, a counterexample is expected to be a realistic and simple database.

Our approach is statically generating such a counterexample rather than dynamically testing the program with randomly generated test cases as in other works such as [79]. Over the generated counterexample, bugs can be observed in the execution results of the Datalog program. Although data provenance techniques from the database literature [80] can provide useful support to explain how and why the unexpected results are derived, whether we can use this provenance information to efficiently track down the detailed source of bugs remains unclear. In this chapter, we fulfill this gap by a novel method that combines the provenance information with the user interaction for resolving the ambiguity in debugging. In summary, this chapter has the following contributions:

- We present a new way to use a syntactic extension of non-recursive Datalog for specifying the properties of a Datalog program.
- To explain to the user the behavior of the written Datalog program, we develop a counterexample generator that statically checks specified properties of non-recursive Datalog programs and generates counterexamples for showing why the properties are not satisfied.

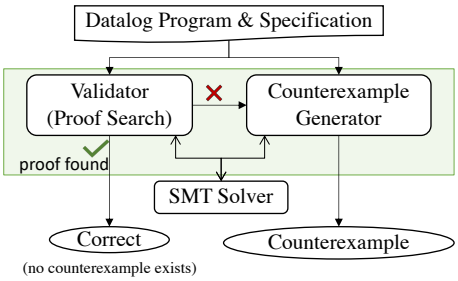


Figure 4.2: Counterexample generation architecture.

- To reduce the user's effort of correcting buggy Datalog programs, we design a user interface and a provenance-based debugging engine to assist the user in locating the bugs with the counterexamples. The debugging engine provides correction hints to the user when the bugs are found.

- To demonstrate the efficiency and the usability of the proposed approach, we have implemented a prototype of the approach and evaluated it with Datalog programs in practice. The source code is available upon request.

This chapter is organized as follows. In Section 4.2, we explain the design of our proposed counterexample generation method. We describe the counterexample-guided debugging approach in Section 4.3 and the experiment in Section 4.4. Section 4.5 presents related works. Section 4.6 wraps up this chapter.

4.2 Counterexample Generation

In this section, we present our approach to statically validating and generating counterexamples for a specified property of a non-recursive Datalog program.

Figure 4.2 shows our counterexample generation architecture. It consists of two main parts: a validator for statically checking the specified property and a counterexample generator for finding a counterexample for the property. The Datalog program with its property specification is first passed to the validator. If the validator successfully proves that the program satisfies the property, we conclude there is no counterexample. If the validator fails, the Datalog program is passed to the counterexample generator. Since many static checks such as

equivalence for Datalog programs are undecidable [60], in both the validator and generator, we transform the property of the Datalog program into logical constraints that can be solved by an SMT solver, even though the termination is not guaranteed.

4.2.1 Specifying Program Properties

As mentioned previously, rather than introducing a new language, our approach is to use the same language to specify properties of a non-recursive Datalog program using preconditions and postconditions. By following the syntax introduced in [36, 81], we allow Datalog rules to have truth constant false (denoted as $\bot$) in the head. In this way, a precondition, as well as a postcondition, is a set of Datalog rules that have the following form:

$$\bot :- r_1(\vec{X}_1), \ldots r_n(\vec{X}_n). \qquad (*)$$

That means $\forall \vec{X}, (r_1(\vec{X}_1) \wedge \ldots \wedge r_n(\vec{X}_n)) \rightarrow \bot$, where $\vec{X}$ are all the free variables.

Example 4.2.1 *Consider the* GetPut *property in Example 4.1.1, which says that if there is no change to the view v, there is no change to the base tables s_1 and s_2. We use non-recursive Datalog to specify the precondition as follows:*

$$\begin{aligned} v^{old}(X,Y) &:- s_1(X,Y). \\ v^{old}(X,Y) &:- s_2(X,Y). \\ \bot &:- v(X,Y), \neg v^{old}(X,Y). \\ \bot &:- v^{old}(X,Y), \neg v(X,Y). \end{aligned}$$

The first two rules store the union of s_1 and s_2 in a mediate relation v^{old}, and the last two rules indicate that v is the same as v^{old}, i.e., the view does not change. And we can specify the postcondition that there is no change to the base tables as follows.

$$\begin{aligned} \bot &:- \Delta^-_{s_1}(X,Y). \\ \bot &:- \Delta^-_{s_2}(X,Y). \\ \bot &:- \Delta^+_{s_1}(X,Y). \end{aligned}$$

□

4.2.2 Validation

We use an SMT solver to prove the specified property of the Datalog program by translating the property into a first-order logic (FO) sentence. If there is a proof such that the FO sentence is valid, the property is satisfied.

Our transformation from non-recursive Datalog to first-order logic is based on the standard transformation [41, 43]. Let P be a non-recursive Datalog program, we inductively transform each relation r in P and the rules of the precondition and the postcondition into an equivalent FO formula φ_r as follows:

If r is an EDB relation, $\varphi_r = r(\vec{X_r}) = r(X_1, \ldots, X_{arity(r)})$.

If r is an IDB relation, i.e., r occurs in the head of m rules:

$$
\begin{aligned}
& r(\vec{X_r}) :- \alpha_{1,1}, \ldots, \alpha_{1,n_1}. \\
& \ldots \\
& r(\vec{X_r}) :- \alpha_{m,1}, \ldots, \alpha_{m,n_m}.
\end{aligned}
$$

The FO formula of r, if considering only the i-th rule, is $\varphi_{r,i}(\vec{X_r}) = \exists \vec{E_i}, \bigwedge_{j=1}^{n_i} \beta_{i,j}$, where $\vec{E_i}$ contains the bound variables of the i-th rule, i.e., the variables not in the rule head, and

$$
\beta_{i,j} = \begin{cases} \varphi_w(\vec{Z}), \text{ if } \alpha_{i,j} \text{ is an atom } w(\vec{Z}) \\ \neg\varphi_w(\vec{Z}), \text{ if } \alpha_{i,j} \text{ is a negated atom } \neg w(\vec{Z}) \\ \alpha_{i,j}, \text{ if } \alpha_{i,j} \text{ is an equality or a built-in predicate, e.g., } x < y \end{cases}
$$

By combining all the rules of r, we have:

$$
\varphi_r(\vec{X_r}) = \bigvee_{i=1}^{m} \varphi_{r,i}(\vec{X_r}) = \bigvee_{i=1}^{m} \left(\exists \vec{E_i}, \bigwedge_{j=1}^{n_i} \beta_{i,j} \right)
$$

By having the first-order formulas of all the IDB relations, each special Datalog rule of (*), which has $\perp$ in the head in the precondition and postcondition, is transformed into a first-order sentence: $\forall \vec{X}, (\varphi_{r_1}(\vec{X_1}) \wedge \ldots \wedge \varphi_{r_n}(\vec{X_n})) \rightarrow \perp$. The precondition, as well as the postcondition, is a conjunction of all its FO sentences transformed from the special Datalog rules.

Let φ_{pre} and φ_{post} be the first-order sentences of the precondition and the postcondition, respectively. We employ an automated theorem prover to prove whether φ_{post} holds if φ_{pre} holds. In other words, we check whether the following first-order sentence is valid: $\varphi_{pre} \rightarrow \varphi_{post}$.

4.2.3 Generating Counterexamples

As mentioned previously, to assist the user in debugging a specified property, we shall generate counterexamples, which are used to guide the user to the location of bugs. The simpler the counterexamples are, the easier the user can succeed in debugging the program.

To generate a counterexample, our idea is to create a symbolic database and transform the evaluation of the Datalog program over the symbolic database with the specified property into a constraint program in Rosette [82]. The Rosette symbolic execution runtime translates the program into logical constraints that are performed by an underlying SMT solver such as Z3 [61]. The result obtained by the Rosette framework is an interpretation of the symbolic input over which the specified property of the Datalog program is violated.

To put it more concretely, we construct a symbolic input of the source and view tables by representing each table as a list of tuples, each tuple is a list, where each element is a symbolic value. The order and the duplicates of tuples are ignored because a relation is a set of tuples rather than a list. Considering Example 4.1.1, assuming that the types of attributes `A` and `B` are integer and real, respectively, we define a symbolic table v as follows (similarly for s_1 and s_2).

```
(define-symbolic a1 integer?)   (define-symbolic a2 integer?)
(define-symbolic b1 real?)      (define-symbolic b2 real?)
(define t1 (list a1 b1))        (define t2 (list a2 b2))
(define v (list t1 t2))
```

Since string values are not supported in the underlying SMT solvers, in our transformation, we use an integer symbol for a string attribute. A value for this integer symbol will be mapped to a string value by using a predefined dictionary, where the integer value is used as an index to determine the corresponding string value. In other words, we build up a partial bijective function that maps an integer

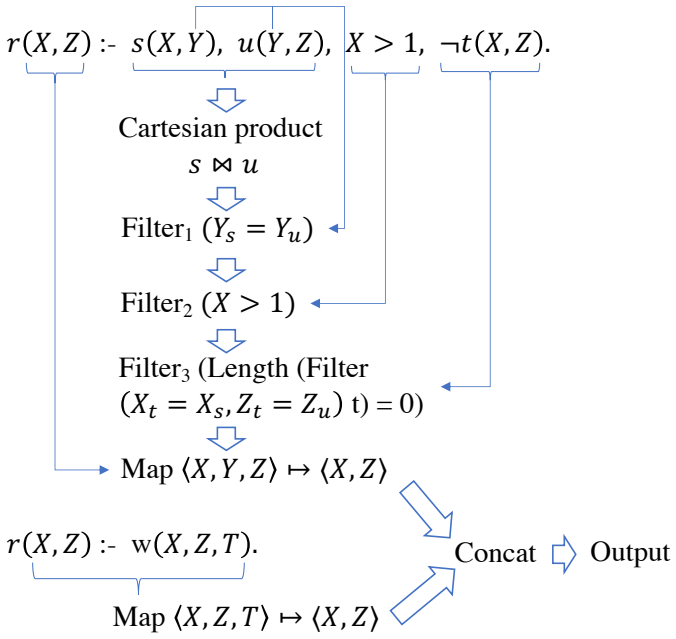


Figure 4.3: Transformation from Datalog to functions.

value to a string in the dictionary. Since the dictionary has finite words, we limit the values of a string attribute to be in the predefined dictionary. For example, for a relation $r(S : string)$, we define a symbolic tuple as the following:

```
(define-symbolic s1 integer?)
(assert (and (< -1  s1) (< s1 dictionary_size)))
(define t1 (list s1))
```

The assertion in the second line ensures that the value of s_1 is in the index range of the dictionary.

We evaluate a non-recursive Datalog program over a symbolic input by using four functions: Cartesian product, Filter, Map, and Concat. Figure 4.3 illustrates the steps for evaluating a relation r. For each rule of r, we first take a cartesian product over all positive relations in the rule body and then apply a filter (Filter_1) for the join attributes, a filter (Filter_2) for all built-in predicates, and another filter (Filter_3) for the negative relations. Over the tuples resulted from these tree filters, we use a mapping function to select the attributes appearing in the rule head[1].

[1]It is not necessary to filter duplicates here. The duplicates will be eliminated in all the other

If r is defined by multiple rules, we evaluate r in each rule and concatenate all the resulted tuples. For a non-recursive Datalog program, which has many IDB relations, we can inductively evaluate all the IDB relations in the program.

Example 4.2.2 *For the first rule in Figure 4.3, we take a cartesian product of the two positive relations s and u. The result is first filtered by $Filter_2$ to select only tuples, where the second attribute of s agrees with the first attribute of u, i.e., $Y_s = Y_u$. $Filter_2$ is applied to select the tuples satisfying $X > 1$. $Filter_3$ checks whether there exists a tuple $\langle X_t, Z_t \rangle$ in t that agrees with the attributes X_s and Z_u in the tuples resulted from $Filter_2$. The mapping function takes a projection over the three-dimension tuples and results in two-dimension tuples. Function Concat gets all the tuples computed by the two rules.* □

We now turn to encode the property that is specified by the precondition and the postcondition. Recall that the precondition, as well as the postcondition, is a set of Datalog rules having constant $\bot$ in the head. To encode these Datalog rules into Rosette constraints, we first replace $\bot$ with a normal predicate, named $\emptyset_{pre}$ for the precondition and $\emptyset_{post}$ for the postcondition, and then encode the evaluation of the obtained Datalog rules into functions as presented previously. These two relations, $\emptyset_{pre}$ and $\emptyset_{post}$, are both expected to be empty. With the evaluation of $\emptyset_{pre}$ and $\emptyset_{post}$ over the symbolic input presented previously, we first encode the precondition into an assertion that the length of table $\emptyset_{pre}$ is equal to 0 as the following:

```
(assert (= 0 (length ∅pre)))
```

We then add another assertion that the length of table $\emptyset_{post}$ is greater than 0 to solve the constraint on the symbolic input that the precondition is satisfied but the postcondition is violated:

```
(solve (assert (< 0 (length ∅post))))
```

Algorithm 2 summarizes the main steps in our proposed counterexample generation. Starting from 0, we increase the maximum size, denoted as n, of each input EDB table. With a value of n, we construct n symbolic tuples for each EDB

checks and algorithms.

Algorithm 2: Counterexample generation

```
1  n ← 0 // The maximum size of input tables
2  Success ← False
3  while not Success do
4  |  n ← n + 1
5  |  foreach EDB relation r_i do // Construct a symbolic input
6  |  |  Define r_i as a list of n symbolic tuples.
   |  // Encoding the property
7  |  Replace ⊥ in the precondition/postcondition with ∅_pre/∅_post.
8  |  Construct the evaluation of ∅_pre and ∅_post over the symbolic EDB
   |   relations.
9  |  Assert the constraints for ∅_pre and ∅_post:
10 |  |  (assert (= 0 (length ∅_pre)))
11 |  |  (solve (assert (< 0 (length ∅_post))))
12 |  (A list of symbol-value pairs, Success) ← Call the Rosette framework
   |   to resolve the constraints
13 |  if Success then
14 |  |  foreach r_i do // Instantiate all the EDB tables
15 |  |  |  Replace each symbol with the corresponding value.
16 |  |  |  Remove duplicates in r_i.
17 |  |  return the instance of all the EDB tables.
```

table. We encode the specified property by constructing assertions corresponding to the precondition and the postcondition. We input these assertions to the Rosette framework [82] to find a value for each symbol in the input that the precondition is satisfied but the postcondition is not. If it succeeds, we stop the while loop, instantiate all the EDB symbolic tables, and eliminate duplicates. Otherwise, we continue the loop with an increased value of n.

4.3 Interactively Locating Bugs with Counterexamples

In this section, we present our method for interactively debugging a non-recursive Datalog program with counterexamples. Our approach consists of a user interface and an underlying debugging engine that assists the user in determining

the location of bugs that cause the unexpected behavior of the program.

4.3.1 Checking Counterexamples

As presented in the previous section, a counterexample is an instance of the input database of the Datalog program such that the property, which is specified by the precondition and the postcondition, is not satisfied. Given an instance of the input database, to check whether the property is violated, we evaluate the output and check whether the input satisfies the precondition and the output does not satisfy the postcondition. Recall that both the precondition and the postcondition are written in Datalog rules with a constant $\bot$ in the head. We check these conditions by replacing $\bot$ with $\emptyset_{pre}(\vec{X})/\emptyset_{post}(\vec{X})$ for the precondition/postcondition, where $\vec{X}$ are variables in the rule body, and evaluating the obtained Datalog rules. The specified property is violated if $\emptyset_{pre}$ is empty but $\emptyset_{post}$ is not empty. Any tuple appearing in $\emptyset_{post}$ is the symptom of the unexpected behavior of the Datalog program with respect to the specified property.

Example 4.3.1 *Consider the putdelta program with an input database in Example 4.1.1 and its* GetPut *property specified in Example 4.2.1. To check* GetPut*, we check the emptiness of $\emptyset_{pre}$ and $\emptyset_{post}$ in the following rules:*

$$
\begin{aligned}
v^{old}(X,Y) &:- s_1(X,Y).\\
v^{old}(X,Y) &:- s_2(X,Y).\\
\emptyset_{pre}(X,Y) &:- v(X,Y), \neg v^{old}(X,Y).\\
\emptyset_{pre}(X,Y) &:- v^{old}(X,Y), \neg v(X,Y).\\
\emptyset_{post}(X,Y) &:- \Delta^-_{s_1}(X,Y).\\
\emptyset_{post}(X,Y) &:- \Delta^-_{s_2}(X,Y).\\
\emptyset_{post}(X,Y) &:- \Delta^+_{s_1}(X,Y).
\end{aligned}
$$

Clearly, in the result, there is no tuple in $\emptyset_{pre}$ but there is a tuple $\langle b_2, a_2\rangle$ in $\emptyset_{post}$. Therefore, GetPut *is violated.* □

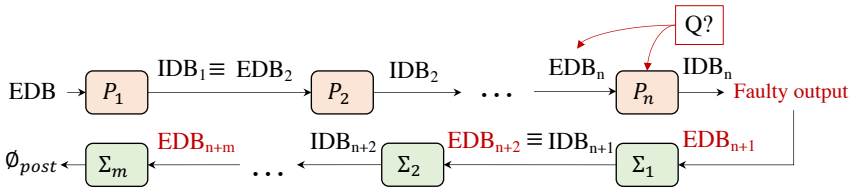


Figure 4.4: Strata-based sequentialization.

4.3.2 Dialog-based User Debugging Interface

Given a counterexample, the debugging problem is to locate the buggy Datalog rules that cause the symptom that the output is faulty. It is extremely ambiguous to determine the locations of bugs since there may be many possible reasons for a fault in the output. Therefore, we allow the user to be involved in the debugging process by designing a dialog-based interface that asks the user to confirm and choose relevant options to handle the ambiguity occurring in the debugging process.

Since Datalog is a declarative programming language, the computation is not explicitly described in the Datalog program. Rather than constructing the computation tree or graph from the Datalog program as in other existing works [76, 77, 78], we shall sequentialize the Datalog program to construct an order of the rules for the evaluation. In other words, we partition the original Datalog program into a sequence of smaller parts, where the final output of the program is obtained by evaluating these parts one by one in the order defined by the sequence. Similarly, we also sequentialize Datalog rules of the postcondition, where the head $\perp$ is replaced by $\emptyset_{post}$.

To construct a partition $\{P_1, P_2, \ldots, P_n\}$ of a Datalog program P, we use the well-known stratification method for Datalog [41] simplified for the case that there is no recursion in the Datalog program. Specifically, we use the precedence graph defined as the following.

Definition 4.3.2 *The precedence graph G_P of a Datalog program P is a directed graph, where nodes are the IDB relations of P and edges are relation dependencies: if $r(\vec{X}) :- \ldots r'(\vec{Y}) \ldots$ or $r(\vec{X}) :- \ldots \neg\, r'(\vec{Y}) \ldots$ is a rule in P, then $\langle r', r \rangle$, which represents that r' precedes r, is an edge in G_P.* □

For a precedence graph, we assign to each node, which is a relation, all the rules of the relation. The rules in each node in the precedence graph form a stratum. We assign to each stratum a unique position such that if stratum P_i precedes stratum P_j in the precedence graph, then $i < j$. Clearly, each stratum in the graph can be evaluated only after all its preceding stratums are evaluated.

Figure 4.4 shows a program P, which is partitioned into n parts $P_1, P_2, \ldots, P_n$, and postcondition rules, which are partitioned into m parts $\Sigma_1, \ldots, \Sigma_m$. The input of P, which consists of EDB relations, is the input for the first part P_1. We evaluate the output of P by evaluating each part individually that the output of P_{i-1} (IDB_{i-1}) becomes the input of P_i (EDB_i) for every part P_i. Similarly, the output of P is the input of the postcondition rules. By evaluating $\Sigma_1, \ldots, \Sigma_m$ in this order, we obtain $\emptyset_{post}$.

Any tuple unexpectedly appearing in $\emptyset_{post}$ indicates that the specified property is violated. From this fault symptom, the debugging process is to analyze how the data is changed after each stratum to detect which stratum contains the bugs. In the input/output of a stratum, there are two types of faulty tuples: *wrong tuples*, which unexpectedly appear, and *missing tuples*, which cannot be computed as expected. For example, all the tuples in $\emptyset_{post}$ are wrong. This is caused by wrong or missing tuples in the input of Σ_m, i.e., the output of Σ_{m-1}.

For each stratum P_i, if there is a wrong/missing tuple in the output of P_i (IDB_i), we have two possible reasons: P_i contains the buggy rules; or the input of P_i, which is the output of P_{i-1}, contains wrong/missing tuples.

Since the root cause of the property violation is in the original Datalog program P, only $P_1, P_2, \ldots, P_n$ need to be inspected. Meanwhile, the stratums of the postcondition rules, $\Sigma_1, \ldots, \Sigma_m$, do not need to be inspected. They are used to detect faulty tuples in the output of P. Our underlying debugging engine automatically predicts the possible faults in the input of each stratum Σ_i. In this way, the possible faults in the output of P are detected without user interaction.

The user interaction is allowed when the underlying debugging engine inspects the stratums from P_n to P_1. At each stratum P_i, when having a faulty tuple in the output of P_i, we let the user confirm and choose one of the two reasons for diagnosing the bugs by questioning the user about the validity of IDB_{i-1}, i.e., the input of P_i. Specifically, we evaluate all the stratums preceding P_i to obtain

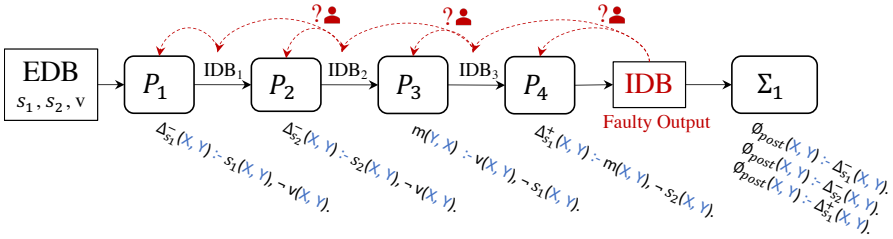


Figure 4.5: Debugging interaction example.

IDB$_{i-1}$ and use the faulty output of P_i (IDB$_i$) to predict faulty tuples in IDB$_{i-1}$. On one hand, if the user confirms that IDB$_{i-1}$ is valid, the underlying engine will suspect P_i to infer possible buggy rules. On the other hand, if the user finds suspiciousness in IDB$_{i-1}$, the underlying engine will infer possible wrong/missing tuples in IDB$_{i-1}$ assuming P_i is correct, and then question the user to confirm the relevant faulty tuples.

Example 4.3.3 *Figure 4.5 illustrates a debugging session for the putdelta program and its* GetPut *property shown in Example 4.1. Here, putdelta is stratified into four parts, P_1, P_2, P_3, P_4, corresponding to the four rules defining the four IDB relations in the program. There is only one stratum Σ_1 for the postcondition rules.* □

4.3.3 Debugging Engine

We now present our underlying debugging engine that generates debugging details for the dialog-based user interaction and performs the debugging process based on the user's choices. Specifically, the debugging engine traverses all the stratums from the last one to the first one. At each stratum P_i, the debugging engine predicts possible faults in the input of the stratum that cause the faults observed in the output of the stratum and lets the user confirm and choose one fault. If the user confirms the input of P_i is correct, the engine suspects P_i. In contrast, if the user chooses one fault, the engine goes to the preceding stratum P_{i-1} for inspecting.

Assuming that the rules in the stratum are correct, and there is a faulty (wrong or missing) tuple in the output of the stratum, we predict faulty tuples in the

input of the stratum based on the provenance information of the faulty tuple in the output that is how it is derived or how it is not derived.

For a wrong tuple in the output of the stratum, its provenance can be explained by constructing all the proof trees that are used by the stratum to derive the tuple. In our stratification strategy, each stratum contains only rules of an IDB relation. Therefore, the maximum height of the proof trees of wrong output tuples is 1. If a wrong tuple does not belong to the IDB relation, it is derived directly from the same wrong tuple in the input of the stratum. In contrast, if a wrong tuple belongs to the IDB relation, it is derived by an immediate inference with rules in the stratum, thus its proof trees have height 1. The proof trees can be extracted from the standard bottom-up evaluation strategy [41] of Datalog by assembling all the immediate inferences.

Example 4.3.4 *Considering the putdelta program in Example 4.3.1 and its stratification in Figure 4.5, the provenance of tuple* $\langle b_2, a_2\rangle$ *of* $\emptyset_{post}$ *in the output of the last stratum is explained by the following proof tree:*

$$\frac{\Delta^+_{s_1}(b_2, a_2)}{\emptyset_{post}(b_2, a_2)}\ [\emptyset_{post}(X, Y) :- \Delta^+_{s_1}(X, Y).]$$

where $\Delta^+_{s_1}(b_2, a_2)$ *is explained by the previous stratum as the following:*

$$\frac{m(b_2, a_2) \quad \neg s_2(b_2, a_2)}{\Delta^+_{s_1}(b_2, a_2)}\ [\Delta^+_{s_1}(X, Y) :- m(X, Y), \neg s_2(X, Y).]$$

□

From the constructed proof trees, we detect all the faulty tuples in the input that must be changed to make the wrong tuples in the output disappear. For a wrong tuple, which is derived directly from the same tuple in the input of the stratum, we conclude this tuple in the input of the stratum is wrong. For a wrong tuple derived by the rules of the stratum, all the proof trees of this tuple must be deconstructed by changing the facts used in these proof trees.

Let w be the IDB relation defined in a stratum P_i, and $w(\vec{A}_0)$ be a wrong tuple in the output of P_i. A proof tree of $w(\vec{A}_0)$ has the following form:

$$\frac{(\neg)r_1(\vec{A}_1) \quad \ldots \quad (\neg)r_n(\vec{A}_n)}{w(\vec{A}_0)}\ [w(\vec{X}_0) :- (\neg)r_1(\vec{X}_1), \ldots, (\neg)r_n(\vec{X}_n).]$$

Here, we apply the rule $w(\vec{X}_0) :- (\neg)r_1(\vec{X}_1), \ldots, (\neg)r_n(\vec{X}_n)$ with the facts

$(\neg)r_1(\vec{A}_1), \ldots, (\neg)r_n(\vec{A}_n)$ to infer $w(\vec{A}_0)$. Since $w(\vec{A}_0)$ is derived if all the facts $(\neg)r_1(\vec{A}_1)$, ..., and $(\neg)r_n(\vec{A}_n)$ hold, changing one of $(\neg)r_1(\vec{A}_1), \ldots, (\neg)r_n(\vec{A}_n)$ is sufficient to make $w(\vec{A}_0)$ not derived, and thus correct $w(\vec{A}_0)$. In other words, $w(\vec{A}_0)$ is wrong because one of the facts $(\neg)r_1(\vec{A}_1), \ldots, (\neg)r_n(\vec{A}_n)$ is wrong. We exclude facts that are from EDB relations because the EDB database is not computed by the Datalog program. We raise a question to the user interface to let the user confirm and choose one wrong tuple. This is repeatedly performed for each proof tree of each wrong tuple in the output of P_i.

Remark 4.3.5 *A fact $\neg r(\vec{A})$ is wrong iff $r(\vec{A})$ is missing. This follows from the closed world assumption (CWA).*

A missing tuple, which is not derived in the output of a stratum, is explained by any proof tree that fails to be constructed. The failed proof tree cannot be completed because of some facts that are required but do not hold. As presented previously, in our stratification strategy, each stratum contains only rules of an IDB relation that the proof trees of a tuple have maximum height 1. A proof tree, which has height 1, is constructed by instantiating a rule in the stratum. To avoid constructing an infinite number of proof trees that are not related to the context of the Datalog program, as other approaches [80], we restrict the Datalog program to its active domain, which is the set of all constants appearing in the EDB relations and the program. Specifically, only values in the active domain are used to instantiate a rule. In this way, we obtain a finite number of proof trees for a tuple in the output.

We detect the faulty tuples in the input that cause a missing tuple in the output as follows. If the missing tuple does not belong to the IDB relation defined by the rules in the stratum, we conclude it is missing in the input of the stratum. In contrast, we construct a proof tree of the missing tuple by instantiating a rule in the stratum and then find all the facts not holding in the rule body. Clearly, these faulty facts explain the missing tuple in the output of the stratum. In this way, by constructing all the proof trees, we enumerate all possible faults in the input and raise a question to the user for choosing the most suitable fault. To reduce the number of possible faults, we also prefer the smaller faults to the bigger

ones. A fault is smaller if the number of faulty facts in the fault is smaller. The smaller a fault is, the more easily it can be fixed.

We have predicted all the faults (wrong and missing tuples) in the input of a stratum based on the assumption that the rules in the stratum are correct. At the user interface level, we have raised questions to the user to confirm the faults in the input that cause the faulty tuples in the output. Since a stratum contains only rules of an IDB relation, named r_i, changing the rules in the stratum can only correct the faulty tuples of r_i in the output. Therefore, for the faulty tuples of r_i, if in the input, there is no possible fault or the user confirms no predicted fault is suitable, we can conclude that the rules in the stratum contain the bugs and start inspecting the stratum's rules.

Given a faulty tuple in the output of a stratum and assuming that all the tuples in the input are correct, the problem is to determine which rules of r_i are wrong or whether a rule is missing. For a wrong tuple in the output, to locate the corresponding buggy rules, we use the wrong tuple's proof trees constructed before. Specifically, all the rules applied in these proof trees are wrong since they must be changed to make the wrong tuple disappear in the output. For a missing tuple in the output, the user has two ways to fix the rules for producing the missing tuple. The first option is changing one of the rules in the stratum so that it can produce the missing tuple. The second option is adding to the stratum a new rule that can be applied to derive the missing tuple.

To assist the user in correcting the buggy rules in the stratum, we give the user correction hints by showing the proof trees of the faulty tuples and showing the input and the output expected for adding/changing the rules. To be efficient, at each stratum, we show all these observations to the users for finding the cheapest way to correct all the bugs found.

Example 4.3.6 *We illustrate our debugging approach by considering the putdelta program in Example 4.1.1 with the* PutGet *property (see Chapter 2), specified as follows. There is no rule for the precondition, and the postcondition is:*

$$s_1^{new}(X,Y) :- s_1(X,Y), \neg\Delta_{s_1}^-(X,Y) \quad \text{(r5)}$$

$$s_1^{new}(X,Y) :- \Delta_{s_1}^+(X,Y). \quad \text{(r6)}$$

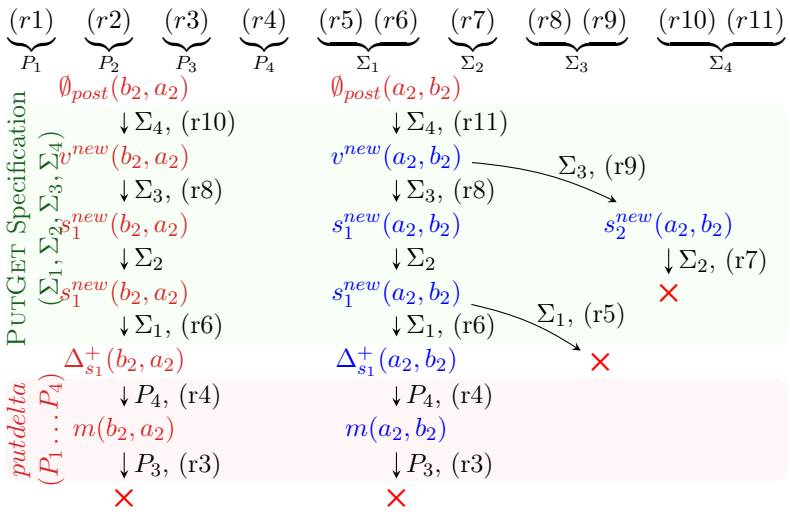


Figure 4.6: Debugging demonstration.

$$s_2^{new}(X,Y) :- s_2(X,Y), \neg\Delta_{s_2}^-(X,Y). \tag{r7}$$

$$v^{new}(X,Y) :- s_1^{new}(X,Y). \tag{r8}$$

$$v^{new}(X,Y) :- s_2^{new}(X,Y). \tag{r9}$$

$$\bot :- v^{new}(X,Y), \neg v(X,Y). \tag{r10}$$

$$\bot :- v(X,Y), \neg v^{new}(X,Y). \tag{r11}$$

That means if we apply delta relations, $\Delta_{s_1/s_2}^{\pm}$ obtained from the putdelta program, to the source relations, s_1 and s_2, and calculate the view v^{new} again, we expect v^{new} to be the same as the initial view v. Let us consider a counterexample of PutGet *as the following: $s_1 = \{\langle a_1, b_1\rangle\}$, $s_2 = \emptyset$, $v = \{\langle a_1, b_1\rangle, \langle a_2, b_2\rangle\}$. Over this counterexample, the result of putdelta is: $\Delta_{s_1}^- = \Delta_{s_1}^- = \emptyset$, $\Delta_{s_1}^+ = \{\langle b_2, a_2\rangle\}$. Thus, $v^{new} = \{\langle a_1, b_1\rangle, \langle b_2, a_2\rangle\}$, leading to that $\emptyset_{post} = \{\langle a_2, b_2\rangle, \langle b_2, a_2\rangle\}$ in the rules (r10) and (r11). Therefore, the* PutGet *property is violated.*

Figure 4.6 illustrates how the causes of the wrong tuples $\emptyset_{post}(a_2, b_2)$ and $\emptyset_{post}(b_2, a_2)$ are predicted. Here, the putdelta program is stratified into P_1, P_2, P_3, P_4 and the PutGet *precondition is stratified into Σ_1, Σ_2, Σ_3, Σ_4.*

For the wrong tuple $\emptyset_{post}(b_2, a_2)$, by using its proof trees at each stratum of $\Sigma_1, \Sigma_2, \Sigma_3$ and Σ_4, we have wrong tuples $v^{new}(b_2, a_2)$, $s_1^{new}(b_2, a_2)$, $s_1^{new}(b_2, a_2)$, and $\Delta_{s_1}^+(b_2, a_2)$, respectively. Since stratum Σ_2 does not contain any rules defining s_1^{new},

the wrong tuple $s_1^{new}(b_2, a_2)$ in the output of Σ_2 is simply derived from this wrong tuple $s_1^{new}(b_2, a_2)$ in the input of Σ_2.

For the wrong tuple $\emptyset_{post}(a_2, b_2)$, at stratum Σ_4, we predict a wrong fact $\neg v^{new}(a_2, b_2)$ in the input of Σ_4. That means $v^{new}(a_2, b_2)$ is missing. At stratum Σ_3, there are two possible proof trees corresponding to rules (r8) and (r9), respectively. Therefore, there are two possible causes of $v^{new}(a_2, b_2)$: $s_1^{new}(a_2, b_2)$ is missing or $s_2^{new}(a_2, b_2)$ is missing. We continue to predict the causes of each of these tuples $s_1^{new}(a_2, b_2)$ and $s_2^{new}(a_2, b_2)$. Eventually, some predicted causes are invalid. For example, at Σ_2, the cause of the missing tuple $s_2^{new}(a_2, b_2)$ is a missing tuple $s_2(a_2, b_2)$ which cannot be fixed because s_2 is an EDB relation. There is only one valid cause: $\Delta_{s_1}^+(a_2, b_2)$ is missing.

After predicting the faults in the output of P_4, i.e., the output of the putdelta program, the user interaction is triggered. At stratum P_4, assuming P_4 is correct, the cause of the wrong tuple $\Delta_{s_1}^+(b_2, a_2)$ is a wrong tuple $m(b_2, a_2)$ and the cause of the missing tuple $\Delta_{s_1}^+(a_2, b_2)$ is a missing tuple $m(a_2, b_2)$. Here, a question of confirming whether $m(b_2, a_2)$ is wrong and whether $m(a_2, b_2)$ is missing is raised to the user interface. If the user confirms there is no faulty tuple, the debugging engine will inspect P_4; in contrast, it goes to stratum P_3. For inspecting P_4, since there is only one rule (r4) that is used in the proof tree of $\Delta_{s_1}^+(b_2, a_2)$ and $\Delta_{s_1}^+(a_2, b_2)$, (r4) is a buggy rule. For P_3, because no fault in the input of P_3 is predicted, the engine inspects P_3 without user interaction. Interestingly, both the choices of inspecting P_4 or going to P_3 can detect the bug that can be solved. Specifically, changing $m(X, Y)$ in (r4) to $m(Y, X)$ can make $\Delta_{s_1}^+(b_2, a_2)$ disappear and make $\Delta_{s_1}^+(a_2, b_2)$ appear in the output, and thus PutGet *satisfied. Similarly, changing $m(Y, X)$ in (r3) to $m(X, Y)$ can also correct the program.* □

4.4 Implementation and Experiment

We have implemented a prototype for our debugging approach in Ocaml and integrated it with Rosette [82] and Z3 [61] as the SMT solvers for our counterexample generation. The user can interact with our system via a command-line tool. By the tool, the user can start a debugging session with a counterexample which is automatically generated by the tool or given by the user.

Table 4.1: Debugging results. ✓indicates that the property is satisfied.

ID	Program	Rules (Program & Properties)	Counterexample Generation Time (s)	Counterexample Size (tuples)			Number of Questions
				DeltaDis	GetPut	PutGet	
1	luxuryitems	12	8.721	✓	✓	2	0
2	ukaz_lok	13	7.162	✓	✓	2	0
3	message	21	10.652	3	2	3	1
4	poi_view	23	10.08	✓	2	3	1
5	all_cars	24	11.116	3	2	3	2
6	newpc	26	10.294	✓	✓	3	1
7	products	28	13.614	✓	✓	4	1
8	purchaseview	29	9.153	✓	5	✓	0
9	vehicle_view	30	timeout	-	-	-	-
10	koncerty	32	47.951	✓	✓	5	2
11	phonelist	33	11.035	4	3	4	1

To evaluate our approach, we use non-recursive Datalog programs collected in Chapter 3. These programs are written for implementing practical view update strategies that are required to be well-defined (called the DeltaDis property) and satisfy the round-tripping properties, i.e., GetPut and PutGet, with the corresponding view definitions to guarantee the consistency between the views and the source tables. We randomly add bugs to these programs and run an experiment to evaluate the performance of our approach in debugging these programs. Specifically, we measure the time for generating counterexamples, the size of the generated counterexamples, and the number of questions used to ask the user for locating the bugs. The experiment is performed on a computer of 2 CPUs and 4 GB RAM running Ubuntu Server LTS 16.04. We set up a timeout of 1 minute for generating counterexamples.

Table 4.1 summarizes the results of our experiment. The time for generating counterexamples and the size of counterexamples almost increase against the number of rules in the program and the specified properties. The generating time also depends on the difficulty of the bugs and the complexity of Datalog rules. For example, `phonelist` has a smaller generating time than `koncerty` because the rules of `phonelist` are more straightforward. `products` has a bigger generating time than `purchaseview` because PutGet is usually more complex than GetPut. For `vehicle_view`, the counterexample generator does not terminate after the maximum allowed running time. The results show that the number of questions

used in locating bugs is usually small. This number depends on the complexity of the program and the difficulty of the bugs. Some simple programs such as `luxuryitems` have no question, meanwhile, some bigger programs such as `all_cars` and `koncerty`, which contain more bugs or more user-written rules, need more questions with the user interaction to find the buggy rules.

4.5 Related Work

Algorithmic debugging [83], also known as declarative debugging, is a semi-automatic debugging technique that is based on the answers of the programmer to a series of questions generated automatically by the algorithmic debugger. Due to its abstraction level, this technique is relevant to declarative programming languages such as Datalog. Some approaches [76, 77, 78] have been proposed to apply algorithmic debugging to Datalog. These existing approaches can assist the user after a fault (i.e., a counterexample) is detected but suffer from the well-known scalability problems of algorithmic debugging [75] that more user interaction is required in the debugging process. In our approach, we strengthen the algorithmic debugging technique applied to non-recursive Datalog by statically generating minimum-size counterexamples for the debugging process. We exploit provenance techniques [84, 85, 80] to automatically predict the root causes of the observed faults of the Datalog programs for reducing the human effort of answering the questions raised by the algorithmic debugger.

4.6 Conclusion

In this chapter, we have presented a novel debugging approach to non-recursive Datalog programs. Specifically, we provide a syntax for users to specify properties of non-recursive Datalog programs. We present a counterexample generator that verifies specified properties and generates counterexamples to show unexpected behaviors of user-written programs. Our counterexample generator is combined with our validation algorithm presented in Chapter 3 to ensure that counterexamples exist. We use symbolic execution of the Datalog program and employ existing

constraint solvers to generate simple and useful input data that show the properties of the program are violated. We design a debugging engine combined with a dialog-based user interface to assist users in locating bugs in the programs with the generated counterexamples. The debugging engine exploits why and why-note data provenance techniques to detect and explain the cause of bugs. We have implemented a prototype for our approach and demonstrated its feasibility and efficiency.

Our future work includes improving the performance of the counterexample generators. As presented in this chapter, our current approach is statically generating counterexamples that the termination is not guaranteed. As shown in our experiments, the static generation works well on smaller Datalog programs but requires more time for bigger ones. In practice, programmers usually write Datalog programs for realistic database tables and views, and thus counterexamples can be detected via testing. Therefore, in the case that our counterexample generators do not terminate, the property specifications can be relaxed to generate a set of counterexample candidates, i.e., a set of test cases, for the original specifications. The suitable counterexamples are finally detected by testing over the test cases that whether the properties are violated.

5 Recursive View Update Strategies

5.1 Introduction

Resource Description Framework (RDF) has become a popular web standard where knowledge graphs are stored in the form of triples $(subject, predicate, object)$. The concept of views in RDF has been studied as a similar concept of views in relational databases. An RDF view specified by a query or mapping exposes a portion of source data that is relevant to the application or other system. RDF views can be used for controlling database access [86] or be materialized and incrementally maintained [87] for rapidly answering queries. In the setting of Ontology-Based Data Access (OBDA) [88], the W3C standard R2RML [39] has been proposed as a language for specifying mappings that expose relational data as a virtual RDF graph which plays as a view for answering RDF queries, e.g., SPARQL queries, over the underlying relational database.

However, most of the mapping languages and RDF query languages for specifying RDF views are unidirectional in the sense that views provide read-only

data access. As the classical view update problem, translating updates on an RDF view to updates on the underlying database is extremely ambiguous that update strategies need to be specified by the database administrators. Although some initial efforts [89, 90, 91] have been devoted to translating SPARQL updates on virtual RDF graphs to SQL update statements on the underlying relational database, the ambiguity issue of the translation has not been completely resolved.

In this chapter, we aim to solve the view update problem of RDF databases by a language-based approach as in other existing works [92, 20]. Instead of automatically translating RDF view updates to source updates, we allow database administrators to program view update strategies (backward programs) for the update translation and thus avoid the ambiguity issues. The defining query (forward program) of the view with the backward program, so-called bidirectional transformations, provides read-write access to the view.

Although an RDF graph can be stored as a ternary relation (triples), the existing approach to the relational view update problem is not expressive enough to deal with ternary relations of RDF graphs. Due to the complex structure, manipulating the graph requires recursive computation on the ternary relation such as reachability, closure, and so forth.

Many works in the programming language community have proposed a promising approach in which a new domain-specific language is designed for constructing a class of bidirectional transformations over a specific data type such as relational data, trees, graphs, and so forth. Although these approaches have a great advantage of guaranteeing the program's well-behavedness by construction, they have inherent drawbacks. Firstly, it is difficult to fully understand and control the behavior of the pre-designed constructors, especially for the backward direction. For instance, in some languages [20, 28, 49, 50], the backward transformations are automatically derived from the forward transformations with auxiliary parameters and thus are difficult to predict. Secondly, although some bidirectional languages [33] are designed for backward transformations and the forward transformation is uniquely derived, they require more effort and expertise for the programmers, who are unfamiliar with the new languages because the backward transformations are more complicated in comparison with the forward one. Thirdly, the DSLs are for a specific data type, the extensibility and the reusability of the language are

limited when dealing with other data types or other classes of update strategies. Extensibility not only to other data types but also to other classes of update strategies.

An alternative to designing a new domain-specific language is programming bidirectional transformations using an existing (unidirectional) language that frees the programmer from learning a new language and being forced to use new constructors. The main challenge of this approach is providing a mechanism to guarantee the well-behavedness of the user-written transformations such as testing [93] or static check (Chapter 3). In Chapter 3, we have proposed to use a class of non-recursive Datalog [43] to program view update strategies, i.e., backward transformations, in relational databases where program properties are statically validated and the forward transformation is uniquely derived.

Although in Chapter 3, we show that Datalog can be used for programming view update strategies on relations, the expressive power of recursions in Datalog, has never been used. Many works [94, 95, 96, 97, 98] have employed the recursion mechanisms of Datalog to express more recursive queries over RDF (Resource Description Framework) knowledge graphs that cannot be expressed in SPARQL [99], a well-known RDF query language. SPARQL has limited navigational capabilities to exploit the graph structure of RDF data. The language cannot capture some very natural and useful navigation patterns [100]. Meanwhile, Datalog is expressive enough for representing every SPARQL query [101, 102, 103, 104, 105, 98] and provides more general forms of recursions for more complex patterns and recursive queries. This makes Datalog a relevant language to exploit the structure of RDF graphs and propagate updates. Nevertheless, how Datalog can be also used in programming bidirectional transformations on knowledge graphs remains unclear.

Recursion is the key but also the challenge to using Datalog for writing bidirectional transformations. Firstly, understanding recursive computations and specifying a view update strategy over a graph or even a tree are expensive tasks and require more expertise from programmers. Secondly, it is even more challenging to automatically validate recursive transformation programs. The fixed-point semantics of Datalog makes the validation problem more complicated and not expressible in first-order logic.

Our key idea to solve the aforementioned challenges is to formulate a Datalog-written view update strategy as the combination of two parts: a recursive part, which implements recursive patterns, and a non-recursive one, which is an inner update strategy. On the one hand, the recursive Datalog rules of the first part are predefined and pre-validated so that their well-behavedness is guaranteed. On the other hand, we allow programmers to manually write the inner update strategies in non-recursive Datalog, which are automatically validated.

To guarantee the ease of use of pre-defined recursive programs in constructing a new one, we extend Datalog with a restricted form of higher-order predicate syntax proposed by previous works [106]. Specifically, the results of pre-defined recursive Datalog rules are parameterized to be a higher-order predicate and to be recalled in user-written programs.

Importantly, we syntactically extend Datalog with higher-order predicates but maintain the first-order semantics of the Datalog program by a translation algorithm. The algorithm encodes all higher-order predicates into first-order predicates defined by Datalog rules without additional syntax such as functional symbols[1]. Our higher-order syntax restriction not only ensures the translation algorithm is complete but also guarantees the expressiveness for writing view update strategies.

This chapter has the following contributions:

- We present a method to use Datalog for specifying view update strategies among relations and RDF graphs.
- We provide an extension of Datalog with a restricted form of higher-order syntax for predefining view update strategies that can be reused to define new ones. This allows view update strategies to be predefined and pre-validated, and enhances the ease of use for non-expert programmers.
- We design an algorithm to transform the higher-order predicates in user-written programs into equivalent Datalog rules without higher-order predicates. In this way, the user-written view update strategies can be efficiently evaluated by existing Datalog engines.

[1]This is different from HiLog

Resident

ID	Name	Gender	BornIn	Child
Person1	Bob	Male	1890	Person2
Person2	John	Male	1815	
Person3	Mary	Female	1900	
Person4	Alice	Female	1880	

Company1

ID	Position
Person1	Employee
Person3	Employee
Person4	Founder

Figure 5.1: The Relational Database.

- We show the expressiveness of our proposed approach by implementing several classes of view update strategies for some common recursive patterns of RDF graphs.
- We have implemented our approach and conducted experiments to evaluate the feasibility of our framework.

The remainder of this chapter is organized as follows. In Section 5.2, we illustrate using Datalog to write view update strategies on RDF graphs. In Section 5.3, we first present the higher-order syntax extension for Datalog. We then introduce the translation algorithm that transforms higher-order predicates into normal Datalog rules, and finally present the validation algorithm for the well-behavedness of user-written programs. In Section 5.4, we present the applications of our proposed approach in implementing a variety of recursive view update strategies. Section 5.5 describes an implementation of our approach and experimental results. Section 5.6. Section 5.7 wraps up this chapter.

5.2 Examples

In this section, we use several examples to illustrate how Datalog can be used for defining updatable RDF views over relations or RDF graphs. In our approach, these views are made updatable by explicitly specifying an update strategy. We

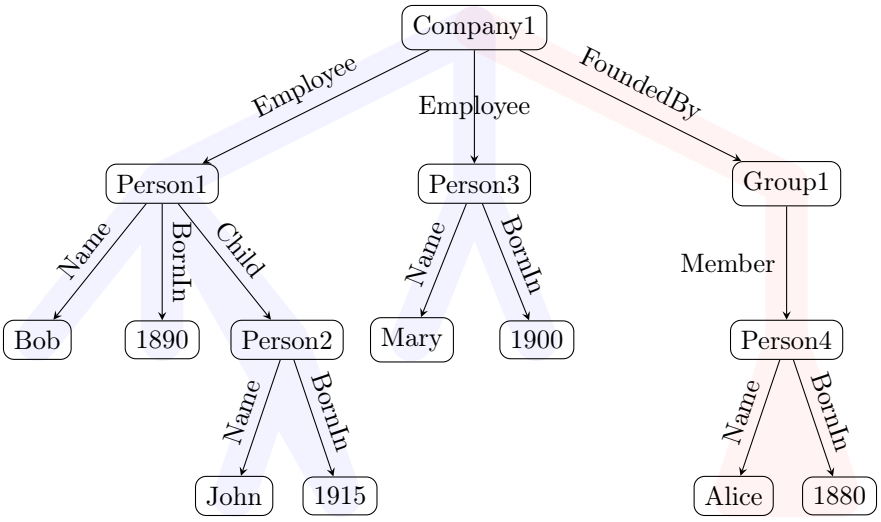


Figure 5.2: An RDF graph.

consider a tree-like RDF graph `src`[2] as shown in Figure 5.2. `src` is represented by the following triples:

```
(Company1, Employee, Person1)
(Person1, Name, "Bob")
(Person1, BornIn, 1890),
(Person1, Child, Person2)
(Person2, Name, "John")
(Person2, BornIn, 1915)
(Company1, Employee, Person3)
(Person3, Name, "Mary")
(Person3, BornIn, 1900)
(Company1, FoundedBy, Group1)
(Group1, Member, Person4)
(Person4, Name, "Alice")
(Person4, BornIn, 1880)
```

Where each triple of the form (Subject, Predicate, Object) represents an edge from a subject node to an object node. For example, (Company1, Employee, Person1)

[2]For simplicity, we use Person1, Company1, Employee, and so forth as the URI references of nodes and labels.

means that Company1 has Person1 as an employee.

5.2.1 RDF views over general relations

Since an RDF graph is represented by a set of triples, i.e., a ternary relation, an RDF view can be defined over other relations by using Datalog. Considering the database shown in Figure 5.1, we define an RDF graph rdf_view (as shown in Figure 5.2) over the two source relations as follows:

```
rdf_view(ID, "Name", N)     :- resident(ID, N, G, _, _).
rdf_view(ID, "BornIn", Y)   :- resident(ID, _, G, Y, _).
rdf_view(ID, "Child", C)    :- resident(ID, _, G, _, C).
rdf_view("Company1", P, ID):- company1(ID, P),
                               P = "Employee".
rdf_view("Group1", "Member", ID):- company1(ID, P),
                               P = "Founder".
rdf_view(C, "FoundedBy", ID):- C = "Company1",
                               ID = "Group1".
```

The RDF graph is constructed by taking the union of the results from projection operators over tables `resident` and `company1`. The Datalog rules of `rdf_view` are all non-recursive. Therefore, we follow our approach in Chapter 3 to use Datalog to specify an update strategy for `rdf_view` as the following:

```
+resident(ID, N, "null", Y, C) :- rdf_view(ID, "Name", N),
                                  rdf_view(ID, "BornIn", Y),
                                  rdf_view(ID, "Child", C),
                                  not resident(ID, N, Y, C).
-resident(ID, N, G, Y, C) :- resident(ID, N, G, Y, C),
                                  not rdf_view(ID, "Name", N).
-resident(ID, N, G, Y, C) :- resident(ID, N, Y, C),
                                  not rdf_view(ID, "BornIn", Y).
-resident(ID, N, G, Y, C) :- resident(ID, N, G, Y, C),
                                  not rdf_view(ID, "Child", C).
+company1(ID, P)           :- rdf_view("Company1", P, ID),
```

```
                          not company1(ID, P), P = "Employee".
founder(ID, "Founder") :- rdf_view("Company1","FoundedBy",GID),
                          rdf_view(GID, "Member", ID).
+company1(ID, P) :- founder(ID, P),
                    not company1(ID, P), P = "Founder".
-company1(ID, P) :- company1(ID, P),
                    not rdf_view("Company1", P, ID),
                    not founder(ID, P).
```

5.2.2 Views over RDF Graphs

Consider the source RDF graph `src` in Figure 5.2, we define a view relation `view` by retrieving all the founders'name of Company1 in `src` as described in the following Datalog rule:

```
view(X) :- src("Company1", "Foundedby", G),
           src(G, "Member", Z),
           src(Z, "Name", X).
```

The first triple `src`("`Company1`", "`Foundedby`", G) is used to retrieve elements g of `src` that have founded Company1, which are stored in variables G. Triple `src`(G, "`Member`", Z) gets all the pairs (g, z), stored in G and Z that z is a member of g. Similarly, from the third triple `src`(Z, "`Name`", X) we retrieve the pairs (z, x) that x is the name of z. The conjunction of these three triples, i.e., the rule body, joins the resulted elements g, pairs (g, z), and pairs (z, x). The rule head `view`(X) indicates that only elements x stored in X are returned as the final result: `view` = {("Alice")}.

The query is a forward transformation that transforms the source graph `src` into a view `view`. Assume that `view` is changed to `view` = {("Alice", "Peter")} that a new person is inserted, to maintain the consistency between the source and the view, we must specify a strategy to update the source graph. Interestingly, `view` is no longer a graph but is a unary relation. Therefore, we can use the approach in Chapter 3 to write a backward transformation that propagates the

updated view to the source as the following[3]:

```
newfounder(X, Y)    :- view(Y), not src(_, Name, Y),
                    :- URI_Gen^src(Y, X).
+src(X, Name, Y)    :- newfounder(X, Y).
+src(G, Member, X) :- src("Company1", "Foundedby", G),
                      newfounder(X, Y).
-src(G, "Member", X) :- src("Company1", "Foundedby", G),
                       src(G, "Member", X),
                       src(X, "Name", N), not view(N).
```

In the first rule, predicate `view`(Y) is used to retrieve elements y of `view`. Meanwhile, triple `src`$(\_, Name, Y)$, where anonymous variable "_" indicates that its value is not the interest, is used to retrieve elements y that are names of some subjects in graph `src`. The negation of `src`$(\_, Name, Y)$ indicates that only elements y of `view` that are not in `src` are selected to join with the pairs (y, x) obtained from predicate $URI\_Gen^{\texttt{src}}(Y, X)$. $URI\_Gen^{\texttt{src}}$ maps a name y to a fresh URI reference x of a new subject. All the pairs (x, y) are then stored in the head relation `newfounder`, and used in the second rule to retrieve triple +`src`$(x, Name, y)$. Notation +`src` indicates that the triple will be inserted to `src`. Similarly, in the third rule, for each pair (x, y) of `newfounder`, we also insert a triple $(g, Member, x)$ to `src` to ensure that x is a member of the founder group g. The last rule deletes the membership between g and x if g is the founder group and the name of x is not in the view.

5.2.3 RDF views defined with recursions

In the previous two examples, the definitions of view are non-recursive. In this example, we shall illustrate a simple example of a recursively defined view over the source RDF graph `src` in Figure 5.2. We define the view by extracting from `src` a subtree (the right part in Figure 5.2) that is connected to the root `Company1` by a label `FoundedBy`. To implement this forward transformation, a recursive computation is required to traverse through the graph structure as the following:

[3]For simplicity, we show only a part of the complete Datalog program.

```
view(X, L, Y) :- src(X, L0, Y), L0 = "FoundedBy".
view(Y, L, Z) :- view(X, L0, Y), src(Y, L, Z).
```

The first rule retrieves triple (X, L_0, Y) that is the edge with label "FoundedBy" from the root node to node `Group1`. The second rule recursively retrieves all the nodes with edges that can be reached from `Group1`.

In the view update strategy, given an arbitrary pair of a source `src` and an updated view `new_view`, our update strategy is removing the old view in the source and then adding to the source the updated view. By following our approach in Chapter 3, for the source `src`, we use notation $-$ and $+$ to indicate the deletion and insertion operations, respectively. The view update strategy is specified in Datalog as the following:

```
old_view(X, L, Y) :- src(X, L0, Y), L0 = "FoundedBy".
old_view(Y, L, Z) :- old_view(X, L0, Y), src(Y, L, Z).
-src(X, L, Y) :- old_view(X, L, Y).
+src(X, L, Y) :- new_view(X, L, Y).
```

In the first two rules, the old view `old_view` is computed over the source `src` by the same Datalog rules of the forward transformation. The last two rules obtain a new source from the old one by deleting all triple X, Y, Z in the old view `old_view` and inserting all triples in the updated view `new_view`.

5.2.4 Specifying views using higher-order predicates

The view definition in Subsection 5.2.3 extracts a subtree from the source by using a label "FoundedBy". The view can be formulated as the result of a forward function on the source: $\texttt{subtree}_{get}^{FoundedBy}(\texttt{src})$. Both the view and the source are ternary relations that correspond to predicates $view(X, L, Y)$ and $src(X, L, Y)$, respectively. Therefore, function $\texttt{subtree}_{get}^{FoundedBy}$ can be described as a higher-order predicate `subtree("get")("FoundedBy")(src)`(X, L, Y) that holds for all triple (X, L, Y) in the result of $\texttt{subtree}_{get}^{FoundedBy}(\texttt{src})$. The definition of higher-order predicate `subtree` is derived from the view definition as the following:

```
subtree(Label)("get")(input_src)(X, L, Y) :-
```

```
                input_src(X, L0, Y), L0 = Label.
subtree(Label)("get")(input_src)(Y, L, Z) :-
                subtree(Label)("get")(input_src)(X, L0, Y),
                input_src(Y, L, Z).
```

Where `subtree` is a second-order predicate that is parameterized with an input source `input_src` and a label `Label`

Similarly, the view update strategy can be described as a higher-order predicate `subtree(Label)("put")(src, new_view)`(X, L, Y) as the following:

```
subtree(Label)("put")(input_src, input_view)(X, L, Y) :-
                input_src(X, L, Y),
                not subtree(Label)("get")(input_src)(X, L, Y).
subtree(Label)("put")(input_src, input_view)(X, L, Y) :-
                input_view(X, L, Y).
```

These two rules obtain a new source from the old one (`input_src`) by deleting all triple X, Y, Z in the old view and inserting all triples in the updated view `input_view`. In the first rule, the old view is computed by the higher-order predicate `subtree("get")` on the source `input_src`. By using a negation, the first rule retrieves only triples (X, L, Y) in the source but not in the view. The second rule simply copies all triples (X, L, Y) in the updated view `input_view`.

By having higher-order predicates *subtree* for both the view definition and update strategy, we can instantiate one view definition:

```
view(X, Y, Z) :- subtree("FoundedBy"("get"))(src)(X, Y, Z).
```

and one view update strategy:

```
new_src(X, Y, Z) :- subtree("FoundedBy")("put")(src, new_view)(
   X, Y, Z).
```

The higher-order predicates have several advantages in programming in comparison with the Datalog language. Firstly, a higher-order predicate, such as `subtree` in our example, can be considered as a macro. Once the predicate is defined, it can be called later in other Datalog programs without rewriting its defining rules for different input sources and views. Secondly, by using higher-order predicates, the Datalog program is well-structured so that each part of the program

can be written independently by different users. More importantly, once the Datalog program is well-structured with higher-order predicates, it is possible to analyze and validate the program.

As an example, we shall further parameterize predicate `subtree` so that it takes another higher-order predicate of an inner view definition and an inner view update strategy, i.e., bidirectional transformation (`inner_bx`) and returns the composition of the two bidirectional transformations.

```
subtree'(inner_bx)(Label)("get")(input_src)(X, L, Y) :-
    inner_bx("get")(subtree(Label)("get")(input_src))(X, L, Y).

subtree'(inner_bx)(Label)("put")(input_src, input_view)(X,L,Y)
    :- subtree(Label)("put")(
        input_src,
        inner_bx("put")(
          subtree(Label)("get")(input_src),
          input_view)
    )(X, L, Y).
```

5.3 An Extension of Datalog with Higher-Order Predicates

As mentioned in Section 5.1 and Section 5.2, manually writing recursive Datalog rules for view update strategies is an expensive task for non-expert programmers. In this section, we formally present the core language used in our approach. As illustrated in Subsection 5.2.3, in a Datalog program of bidirectional transformations, we allow higher-order predicates that bring in more flexible features and advances for defining the transformations. Our core language can be considered as an extension of Datalog with higher-order predicates in a safe restricted form or a subclass of HiLog [106] where functional symbols are not allowed.

5.3.1 Datalog with Higher-order Predicates

Let $\mathscr{V}$ be a countably infinite set of variables, $\mathscr{C}$ be a countably infinite set of constants, and $\mathscr{P}$ be a countably infinite set of predicate symbols. We follow the syntax of higher-order predicates introduced by HiLog[106] that terms are inductively defined as follows:

- Each x in $\mathscr{V} \cup \mathscr{C} \cup \mathscr{P}$ is a term.
- If x_1, x_2 are variables or constants ($x_1, x_2 \in \mathscr{V} \cup \mathscr{C}$), then $x_1 = x_2$, $x_1 < x_2$, $x_1 > x_2$ are terms where $=, <, >$ are built-in predicates.
- If $t, t_1, \ldots, t_k$ are terms, then $t(t_1, \ldots, t_k)$ is a term. For the case that t is a predicate symbol p and $t_1, \ldots, t_k$ are all first-order variables, $p(t_1, \ldots, t_k)$ is a first-order predicate. Otherwise, $t(t_1, \ldots, t_k)$ is a higher-order predicate.

A Datalog program is a nonempty finite set of rules, each rule is an expression of the form:

$$L_0 :- L_1, \ldots, L_n,$$

and each literal L_i is of the form

$$(\neg)P(t_{11}, t_{12}, \ldots)(t_{21}, t_{22}, \ldots)\ldots(t_{n1}, t_{n2}, \ldots)$$

where P is a predicate symbol and the head literal L_0 must be positive. Our syntax is different from HiLog in the sense that we do not allow functional symbols in Datalog programs. In a rule, a variable x is called a higher-order variable if it plays as a predicate, i.e., a term $x(t)$ appears in the rule. The other variables are first-order variables. As a convention, throughout this chapter, we use uppercase letters to write first-order variables and use lowercase letters for higher-order variables.

5.3.2 Syntax Restriction

We extend the safety condition from Datalog to Datalog with higher-order syntax as follows:

- All first-order variables in the rule head or in a negative predicate must be safe, i.e., must appear in a positive predicate in the rule body. This follows the safety condition of Datalog [41].
- Any higher-order variable or predicate symbol t ($t \in \mathscr{V} \cup \mathscr{P}$) appearing in the rule body must satisfy one of the following conditions:
  - t is a higher-order variable appearing in the rule head.
  - t is a predicate symbol that is already defined by some rules or is an EDB relation symbol.

We accept only the recursion for one rule that a predicate in the rule body also appears in the rule head. Self-recursively-called higher-order predicates such as $R(p(R))$ are not allowed.

Example 5.3.1 *The following does not conform to the restriction for higher-order predicates:*

```
rdf_view(X, L, Y) :- subtree("FoundedBy")("get")(rel)(X, L, Y),
                     rel(Z).
```

Here a variable `rel` *appears in the rule body but not in the rule head.* □

5.3.3 Translating Higher-Order Predicate into First-Order Predicate

In our approach, we extend Datalog with a restricted form of higher-order predicates. It is remarkable that the higher-order predicates do not make Datalog more expressive. The higher-order predicates are convenient for reusing predefined Datalog rules in constructing a new program. Specifically, a higher-order predicate can be used to generalize the transformations defined in a Datalog program. We turn a Datalog-written transformation into a more general form defined by higher-order predicates, which can be reused to construct other Datalog programs.

In this section, we present our translation algorithm that transforms all higher-order predicates in a program into equivalent first-order predicates with additional defining Datalog rules. Specifically, the input of our translation algorithm is a

Datalog program (with higher-order predicates) that defines a first-order predicate, i.e., a normal relation. The algorithm returns an equivalent Datalog program in which only first-order predicates are used. The translated Datalog program is used for realistic execution.

Consider a Datalog program P (with higher-order predicate) and a target predicate t, the function $translate(P, t)$ returns a translated Datalog program P' and a translated predicate t' of t. First, for each rule in P we eliminate any constant c in the rule head by introducing an equality $X = c$ with a fresh variable X in the rule body and substitute X for c in the rule head. Then *translate* is defined for each case of t as follows.

Case 1: t is a variable or a constant or a predicate symbol ($t \in \mathscr{V} \cup \mathscr{C} \cup \mathscr{P}$). We keep t unchanged and the translated Datalog program has no rule ($\emptyset$):

$$translate(P, t) = (\emptyset, t)$$

Case 2: t is one of $x_1 = x_2$, $x_1 < x_2$, and $x_1 > x_2$. The result is the same as Case 1:

$$translate(P, t) = (\emptyset, t)$$

Case 3: t is $p(X_1, X_2, \ldots, X_n)$ where p is an EDB relational symbol, $X_1, X_2, \ldots$, X_n are first-order variables or constant:

$$translate(P, p(X_1, X_2, \ldots, X_n)) = (\emptyset, p(X_1, X_2, \ldots, X_n))$$

Case 4: t is $p(t_{11}, \ldots, t_{1m_1}) \ldots (t_{n1}, \ldots, t_{nm_n})$. If p is not defined in P, e.g., p is a higher-order variable, $translate(P, t)$ is undefined. Otherwise, let $(r_1), (r_2), \ldots (r_m)$ be all the defining rules of p. Each rule (r_i) is of the following form:

$$p(x_{11}, \ldots, x_{1m_1}) \ldots (x_{n1}, \ldots, x_{nm_n}) :- (\neg)t_1, (\neg)t_2, \ldots, (\neg)t_k.$$

We obtain a new rule (r_i') by substituting each t_{jk} for x_{jk} in rule (r_i):

$$p(t_{11}, \ldots, t_{1m_1}) \ldots (t_{n1}, \ldots, t_{nm_n}) :- (\neg)t_1', (\neg)t_2', \ldots, (\neg)t_k'.$$

Let $\vec{X_t}$ be all the first-order variables of t, we introduce a fresh predicate name [4] $translated_t$ for t. We translate t into a predicate $translated_t(\vec{X_t})$ and translate rule (r'_i) into a new rule (r''_i) by translating each term t'_j in the body of rule (r'_i). $translate(P, t)$ is undefined if $translate$ is undefined for any term t'_j. Otherwise, let $translate(P, t'_j) = (P^i_j, translated_{t'_j}(\vec{X_{t'_j}}))$. The translated rule (r''_i) is as the following:

$$translated_t(\vec{X_t}) :- (\neg)translated_{t'_1}(\vec{X_{t'_1}}), (\neg)translated_{t'_2}(\vec{X_{t'_2}}), \ldots, (\neg)translated_{t'_k}(\vec{X_{t'_k}}).$$

Where there is no higher-order predicate. The translated Datalog program of t is the one that consists of all the translated rules (r''_i) and all the translated Datalog programs P^i_j of each term t'_j in (r'_i).

$$translate(P, t) = \left(\{(r''_1), (r''_2), \ldots (r''_m)\} \cup \bigcup_{i=1}^{m} \bigcup_{j} P^i_j, translated_t(\vec{X_t}) \right)$$

It is remarkable that if $translate(P, t)$ is defined, the translated program of t is a Datalog program that has no higher-order predicate. This can be easily proven by induction.

Example 5.3.2 *Consider our example in Subsection 5.2.4. Let P be the Datalog program of the view definition, the target predicate is* `view`*. We shall find a translated Datalog program of* `view` *that has no higher-order predicate.* `view` *is defined by:*

```
view(X, Y, Z) :- subtree("FoundedBy")("get")(src)(X, Y, Z).
```

To translate `subtree("FoundedBy")("get")(src)`(X, Y, Z) *we shall unfold the defining rules of* `subtree` *by replacing* `input_src` *with* `src`*.*

```
subtree("FoundedBy")("get")(src)(X, L, Y) :-
                    src(X, L0, Y), L0 = "FoundedBy".
```

[4] We introduce the same fresh predicate name for t and t' if t' can be obtained by substituting the first-order variables of t' for the first-order variables in t.

```
subtree("FoundedBy")("get")(src)(Y, L, Z) :-
                    subtree("FoundedBy")("get")(src)(X, L0, Y),
                    src(Y, L, Z).
```

Where the two rules of `subtree("put")` *are trivially removed because label* `"put"` *does not match label* `"get"`*, and thus there is no derivable tuple from these rules. Predicate* `subtree("FoundedBy")("get")(src)` *is replaced with a dummy one* $translated_{\texttt{subtree("FoundedBy")("get")(src)}}$*. We obtain a new Datalog program:*

```
view(X, Y, Z) :- translated_subtree("FoundedBy")("get")(src)(X, Y, Z).
translated_subtree("FoundedBy")("get")(src)(X, L, Y) :-
    src(X, L0, Y), L0 = "FoundedBy".
translated_subtree("FoundedBy")("get")(src)(Y, L, Z) :-
    translated_subtree("FoundedBy")("get")(src)(X, L0, Y), src(Y, L, Z).
```

where no higher-order predicate appears in the program. □

Lemma 5.3.3 *For a first-order predicate* $\texttt{target}(\vec{X})$ *defined by a Datalog program (with higher-order predicates)* P*,* $translate(P, \texttt{target}(\vec{X}))$ *is defined.* □

Proof. (Sketch) As in the definition of *translate*, $translate(P, t)$ may be undefined only in case 4 either when p has no defining rules or when *translate* is undefined for a term appearing in the defining rules of p. These cases never occur in the performance of $translate(P, \texttt{target}(\vec{X}))$ due to the restriction of higher-order predicates in a Datalog program and the fact that $\texttt{target}(\vec{X})$ is a first-order predicate. Indeed, a defining rule of $\texttt{target}(\vec{X})$ is of the following form:

$$\texttt{target}(\vec{X}) :- (\neg)t_1, (\neg)t_2, \ldots, (\neg)t_n$$

In this Datalog rule, there is no higher-order variable in the head. Therefore, in each term t_i of the rule body, there is no higher-order variable and all higher-order predicate symbols must be defined by some rules in P. In other words, if t_i belongs to case 4 of *translate*, t_i has defining rules. If we substitute t_i for the head of a defining rule (r_i) of t_i, all the higher-order variables appearing in both the head and the body of (r_i) become predicate symbols defined by some other rules. Therefore, in each term in the body of rule (r_i), there is no higher-order variable

and all higher-order predicate symbols must be defined. We complete the proof by induction. □

Theorem 5.3.4 *Let t be a term and P be a Datalog program with higher-order predicates. If $translate(P, t)$ is defined that $translate(P, t) = (P', t')$, let D be an EDB database of P and v is a variable assignment for free first-order variables in t:*

$$P(D) \models_v t \Leftrightarrow P'(D) \models_v t'$$

Where $P(D)$ and $P'(D)$ are the results of P and P' over D, respectively. □

Proof. (Sketch) By induction. Case 4 of *translate* is the nontrivial case. In case 4, the mapping from t to $translated_t$ and the substitution of t_{jk} for x_{jk} in rule (r_i) preserve all free first-order variables. The translated rule (r_i'') and rule r_i' have the same free first-order variables. Therefore, for a variable assignment v, (r_i) is satisfied if and only if (r_i'') is satisfied. Moreover, for a translated rule (r_i'') of a translated predicate t', the original rule (r_i) of t exists. Recall that $P(D) \models_v t$ indicates that for a variable assignment v, t is derived by a rule (r_i), i.e., (r_i) is satisfied. We conclude $P(D) \models_v t$ if and only if $P'(D) \models_v t'$,

□

5.3.4 Validation

As mentioned before, our validation is to guarantee the well-behavedness of user-written view update strategies in Datalog with higher-order predicates. In other words, the validation statically checks whether the user-written program satisfies round-tripping properties.

Although all higher-order predicates can be transformed into normal Datalog rules by our translation algorithm presented in the previous Section, the recursion is not eliminated so automatically validating the program is challenging. Fortunately, there are some certain common recursive patterns for each type of RDF graph. For example, when the RDF graph has a tree structure, one common pattern is splitting the tree into smaller subtrees for applying simple operators. Therefore, all the pre-defined programs for these common recursive patterns can be manually proved.

In our approach, the pre-defined recursive patterns can be generalized and encapsulated in higher-order predicates such that the predicates take as input other higher-order predicates of some simple inner view update strategies (without recursive computation). The manual proof of a pre-defined recursive pattern is provided with the assumption that all inner view update strategies are well-behaved. In this way, validating a view update strategy, which is specified by using pre-defined recursive patterns with inner view update strategies, is reduced to validating the inner view update strategies. Since the inner view update strategies are non-recursive, we employ the validation algorithm (Algorithm 1) to automatically check the validity.

5.4 Predefining View update Strategies

As mentioned in Section 5.3, a Datalog program, in which both the input and output are relations, can be parameterized as a single higher-order predicate. In this section, we show the expressiveness of our approach by implementing update strategies (backward transformation) for a variety of views defined by common structural patterns. We first introduce some common computation patterns and operations that are used for view update strategies. We then show that these patterns can be implemented as higher-order predicates predefined by Datalog rules. We consider three types of RDF views and for each type, we introduce several basic common patterns and operators (e.g., map, split, condition, ...) as higher-order predicates:

- The RDF view is defined over general relations.
- The RDF view is defined over a tree-like RDF graph that has no cycle.
- The source of the RDF view is a general graph structure.

For the first types of RDF views, the Datalog program is a simple and non-recursive view update strategy. The purpose of representing such Datalog programs by higher-order predicates is to use them as parameters of other higher-order predicates so that we can achieve the composition of multiple Datalog programs

or iterate the execution of a simple Datalog program over the whole recursive structure by calling the predicates of recursive computation patterns. For the second two types of RDF views (over tree-like RDF graphs or general graph structures), we consider common and typical Datalog programs that implement several (recursive) computation patterns over the graph or tree structures. By encapsulating the program into a single higher-order predicate, the program can be reused in writing other programs.

5.4.1 General relations

We consider views defined over general relations by common operators in relational algebra such as projection, selection, join, and so forth. As presented in Chapter 3, we have shown that view update strategies for these views can be written in non-recursive Datalog. This is because the relations do not have recursive structures, both the programs of view definitions (forward direction) and update strategies (backward direction) are usually non-recursive. In this section, we show that these view update strategies for each common operator can be also pre-defined in Datalog rules with higher-order predicates. By using these pre-defined predicates, view update strategies for views defined with multiple operators can be clearly constructed in a compositional way rather than writing the whole new Datalog program.

As an example, consider the RDF view defined over relations in Subsection 5.2.1. In the backward direction, source updates are computed from the RDF view by complex operators such as self-join. It is clearer to construct the program as a composition of two subprograms. In the RDF view, column `Gender` is not exposed, thus we can create an intermediate view as a projection of the source relation `resident` for defining the final RDF view. In the first step, the intermediate view is defined as follows:

```
exposed_resident(ID, N, Y, C) :-
            projection("get")(resident)(ID, N, Y, C).
```

Where `projection("get")` is a higher-order predicate that describes the forward direction of the projection as follows:

```
projection("get")(input_relation)(ID, N, Y, C) :-
            input_relation(ID, N, G, Y, C).
```

In the second step, we define the RDF view over `exposed_resident` as follows:

```
rdf_view(ID, N, Y, C) :-
            r2r("get")(exposed_resident)(ID, N, Y, C).
```

The definition of `r2r("get")` is as the following:

```
r2r("get")(input_relation)(ID, "Name", N)    :-
                            input_relation(ID, N, _, _).
r2r("get")(input_relation)(ID, "BornIn", Y)  :-
                            input_relation(ID, _, Y, _).
r2r("get")(input_relation)(ID, "Child", C)   :-
                            input_relation(ID, _, _, C).
```

It is remarkable that `r2r("get")` exposes all the information in `rdf_view`, and thus enjoys the injectivity property. The backward direction `r2r("put")` is defined by outer self-join over a modified view `rdf_view'` on column `ID`.

```
r2r("put")(rdf_view')(ID, N, Y, C) :-
                        rdf_view'(ID, "Name", N),
                        rdf_view'(ID, "BornIn", Y),
                        rdf_view'(ID, "Child", C).
r2r("put")(rdf_view')(ID, N, Y, "null") :-
                        rdf_view'(ID, "Name", N),
                        rdf_view'(ID, "BornIn", Y),
                        ¬ rdf_view(ID, "Child", C).
r2r("put")(rdf_view')(ID, N, "null", C) :-
                        rdf_view'(ID, "Name", N),
                        rdf_view'(ID, "Child", C),
                        ¬ rdf_view(ID, "BornIn", Y).
r2r("put")(rdf_view')(ID, "null", Y, C) :-
                        rdf_view'(ID, "Name", N),
                        rdf_view(ID, "BornIn", Y),
                        ¬ rdf_view'(ID, "Child", C).
```

```
r2r("put")(rdf_view')(ID, N, "null", "null") :-
                        rdf_view'(ID, "Name", N),
                        ¬ rdf_view'(ID, "BornIn", Y),
                        ¬ rdf_view(ID, "Child", C).
r2r("put")(rdf_view')(ID, "null", "null", C) :-
                        rdf_view'(ID, "Child", C),
                        ¬ rdf_view'(ID, "Name", N),
                        ¬ rdf_view(ID, "BornIn", Y).
r2r("put")(rdf_view')(ID, "null", Y, "null") :-
                        rdf_view(ID, "BornIn", Y),
                        ¬rdf_view'(ID, "Name", N),
                        ¬ rdf_view'(ID, "Child", C).
```

The backward direction of `projection` (`projection("put")`) is ambiguous since the column `gender` is projected away that `projection("get")` is not injective. There are multiple ways to fill in the missing column `gender`. We choose one in the following definition:

```
projection("put")(source_resident, exposed_resident')(ID, N, G,
    Y, C) :-
                    source_resident(ID, N, G, Y, C),
                    exposed_resident'(ID, N, Y, C).
projection("put")(exposed_resident')(ID, N, G, Y, C) :-
                    exposed_resident'(ID, N, Y, C),
                    ¬ source_resident(ID, N, _, Y, C),
                    resident(ID, _, G, _, _),.
projection("put")(exposed_resident')(ID, N, "unknown", Y, C) :-
                    exposed_resident'(ID, N, Y, C),
                    ¬ source_resident(ID, N, _, Y, C),
                    ¬ source_resident(ID, _, _, _, _).
```

Where the first rule keeps all the unchanged tuples in `source_resident`, the second rule computes updated tuples and keeps the column `gender` unchanged. The third rule computes the inserted tuple with a value `unknown` for column `gender`. The deleted tuples are discarded.

The backward transformation from `rdf_view` to the source relation `resident` is a composition:

```
new_resident(ID, N, G, Y, C) :- projection("put")(
   source_resident, r2r("put")(rdf_view')) (ID, N, G, Y, C).
```

5.4.2 Tree-like RDF graphs

We now present some useful computation patterns that are implemented in Datalog and encapsulated in higher-order predicates. For the case that the RDF graph has a tree structure, we adapt several patterns described in existing works [92]. For such an inductive tree structure, the computation patterns are in the form of a three-step operation: split-process-merge. The tree is first decomposed into subtrees, each subtree is processed by an inner operation and the results are merged into a single tree. In the rest of this subsection, we shall present some typical patterns including `fork`, `conditional`, and `map` as presented in [92].

Fork

We shall implement the operation of the `xfork` lens presented in [92] using Datalog with higher-order predicates. As in [92] `xfork`, we first split the original tree in Figure 5.2 into two subtrees (a left subtree and a right subtree) according to the labels of the outgoing edges from the root. For each subtree, we apply an inner bidirectional transformation. Finally, the two tree views obtained from the two inner bidirectional transformations are merged into a single tree. The definition of this pattern, called `fork_by_label`, is written by using higher-order predicates as follows:

```
1 hsplit_by_label(rdf_input, Label)("r")(X, L, Y) :-
2                 subtree(Label)("get")(rdf_input)(X, L, Y).
3 hsplit_by_label(rdf_input, Label)("l")(X, L, Y) :-
4                 rdf_input(X, L, Y),
5                 ¬ subtree(Label)("get")(rdf_input)(X, L, Y)..
6
7 % forward interpretation
```

```
 8 fork_by_label(bx1, bx2, Label)("get")(rdf_src)(X, L, Y) :-
       hmerge(
 9      bx1("get")(hsplit_by_label(rdf_src, Label)("l")),
10      bx2("get")(hsplit_by_label(rdf_src, Label)("r")))(X, L, Y).
11
12 % backward transformation
13 fork_by_label(bx1, bx2, Label)("put")(rdf_src_view)(X, L, Y) :-
        hmerge(
14      bx1("put")(fork_hsplit_by_label(rdf_src_view, Label)("l")),
15      bx2("put")(fork_hsplit_by_label(rdf_src_view, Label)("r"))
16      )(X, L, Y).
```

`fork_by_label` is parameterized with a label and two inner higher-order predicates `bx1` and `bx2` corresponding to two inner bidirectional transformations.

The forward direction works as follows. In the definition of `hsplit_by_label`, to split the original tree, we use the definition of `subtree` as described in Subsection 5.2.4 to extract from the tree in Figure 5.2 the subtree (the right subtree) where the outgoing edges from the root are labeled by Label. Then the left subtree, which is the remaining part, is the one that has all the edges in `rdf_src` but not in the right subtree and is computed by a set difference operator. `hsplit_by_label` is applied to the source RDF graph to split the source into left and right parts. We use labels `"l"` and `"r"` for the left and right subtrees, respectively. The forward directions of the two inner higher-order predicates `bx1("get")` and `bx2("get")` are then applied to each part before calling higher-order predicate `hmerge` to merge the results.

In the backward direction, we have similar operations. `fork_hsplit_by_label` is a special version of `hsplit_by_label` that takes a pair (represented by `rdf_src_view`) of the source and the view RDF graphs, splits both the source and the view, and pairs the left part of the source with the left part of the view, similarly for the right parts. The backward directions of the two inner higher-order predicates `bx1("put")` and `bx2("put")` are then applied two each pair before calling higher-order predicate `hmerge` to merge the results.

Our observation is that an inner bidirectional transformation works on a

component that is structurally smaller than the original tree, and thus is easier to manually implement. Meanwhile, writing decomposing operations, e.g., splitting, requires more knowledge about recursion. Therefore, we predefine a variety of program templates that employ the recursive computation power of Datalog to work on tree data structures. Programmers can reuse these templates with their own arbitrary implementation of inner bidirectional transformations without recursion.

Condition

We consider the `ccond` lens presented in [92]. In the definition of `ccond`, given a source and a view, we apply one of two inner different bidirectional transformations (BXs) for the source and the view depending on a condition over the source. Specifically, if the source satisfies a condition, the forward transformation and the backward transformation of the first BX are selected as the forward transformation on the source, and the backward transformation on the source and the updated view. In contrast, the forward and backward transformations of the second BX are selected. The `ccond` lens can be implemented in Datalog with higher-order predicate as the following:

```
1  % Backward transformation
2  ccond(bxs)("put")(rdf_src_view)(X, L, Y) :-
3              bxs("condition")(rdf_src_view("source"))(),
4              bxs("first")("put")(rdf_src_view)(X, L, Y).
5  ccond(bxs)("put")(rdf_src_view)(X, L, Y) :-
6              not bxs("condition")(rdf_src_view("source"))(),
7              bxs("second")("put")(rdf_src_view)(X, L, Y).
8
9  % Forward transformation
10 ccond(bxs)("get")(src)(X, L, Y) :-
11             bxs("condition")(src)(),
12             bxs("first")("get")(src)(X, L, Y).
13 ccond(bxs)("get")(src)(X, L, Y) :-
14             not bxs("condition")(src)(),
```

```
15              bxs("second")("get")(src)(X, L, Y).
```

We use a single higher-order predicate `bxs` to encapsulate both the inner bidirectional transformations and the condition such that `bxs("condition")` corresponds to the higher-order predicate of the condition on the source, `bxs("first")` and `bxs("second")` are the first and the second bidirectional transformations, respectively. `rdf_src_view` is a higher-order predicate representing the pair of a source and a view such that `rdf_src_view("source")` corresponds to the source and `rdf_src_view("view")` corresponds to the view.

Map

We define the `map` bidirectional transformations in a similar way as the `map` lens presented in [92]. Given a source tree stored in a relation `rdf_src` with a root R, in the forward transformation, `map` extracts all the subtrees having a root as a child of R, applies an inner forward transformation `inner_bx("get")` to each subtree, and then merge all the resulted subtrees into a single tree, which is the view. In the backward transformation, `map` extracts all the subtrees of the source as well as the view in a similar way, pairs each view subtree with a source subtree, applies the inner putback transformation `inner_bx("put")` to each pair, and finally merge all the resulted subtrees as a new source. The Datalog program defining the higher-order predicate `map` is as the following[5]:

```
1  % forward interpretation
2  map_subview(inner_bx)(rdf_src)(R2)(R, L, R2) :-
3          root_of_tree(rdf_src)(R), rdf_src(R, L, R2).
4  map_subview(inner_bx)(rdf_src)(R2)(X2, L, Y2) :-
5          root_of_tree(rdf_src)(R), rdf_src(R, _, R2),
6          inner_bx("get")(rooted_tree(rdf_src, R2))(X2, L, Y2).
7
8  map(inner_bx)("get")(rdf_src)(X, L, Y) :-
9          map_hmerge(map_subview(inner_bx)(rdf_src))(X, L, Y).
10
11 % putback transformation
```

[5]For simplicity, we show only the main Datalog rules

```
12 map_subsource(inner_bx)(rdf_src_view)(R2)(X, L, Y) :-
13         inner_bx("put")(map_subpair(rdf_src_view)(R2))(X, L, Y).

14
15 map_subsource(inner_bx)(rdf_src_view)(R2)(R, L, R2) :-
16         root_of_tree(rdf_src_view("source"))(R),
17         rdf_src_view("source")(R, L, R2).
18
19 map(inner_bx)(rdf_src_view)(X, L, Y) :-
20     map_hmerge(map_subsource(inner_bx)(rdf_src_view))(X, L, Y).
```

Here, predicate `root_of_tree(rdf_src)(R)` holds only if R is the root of the tree `rdf_src`. `rooted_tree(rdf_src, R2)` corresponds to a subtree having a root $R2$. `map_subview(inner_bx)(rdf_src)(R2)` corresponds to a subtree obtained by applying the inner forward transformation (`inner_bx("get")`) to the source subtree rooted by $R2$. In the first rule, `map_subview` extract and keep all the outgoing edges $(R, L, R2)$ of the root R. In the second rule, we apply `inner_bx("get")` to each subtree `rooted_tree(rdf_src, R2)`. The third rules merge the subtrees resulted from `map_subview` by a predicate `map_hmerge`.

In the putback transformation of `map`, we have an additional predicate `map_subpair(rdf_src_view)(R2)` corresponding to each pair of a view subtree and a source subtree rooted by $R2$. By applying `inner_bx("put")` to each pair, in the fourth rule, we obtain `map_subsource(inner_bx)(rdf_src_view)(R2)` corresponding to the new source subtree rooted by $R2$. The fifth rule, `map_subsource` keeps all the outgoing edges from the root of the source subtree (`rdf_src_view ("source")`). Finally, we merge all the source subtrees resulted from `map_subsource` by using the predicate `map_hmerge`.

Discussion

The difference between our work and previous combinator lenses is that we do not limit the programmers to a set of predefined bidirectional transformations, i.e., combinator lenses. We allow programmers to write arbitrary implementations of inner bidirectional transformations. We focus on the RDF format that allows

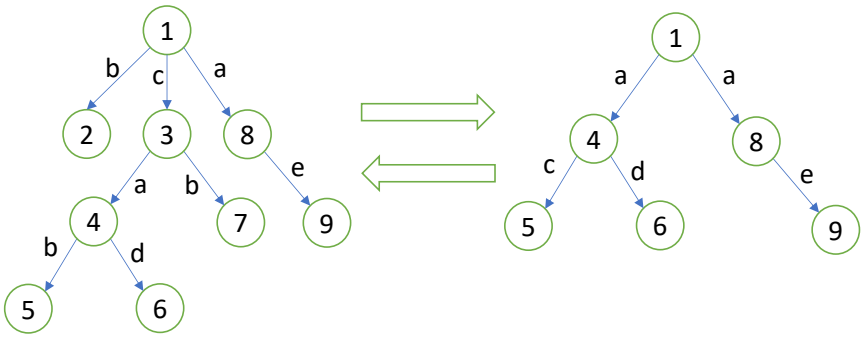


Figure 5.3: A source graph (on the left) and a view graph (on the right).

outgoing edges of a vertex to have the same label. In contrast, combinator lenses represent trees in the forms of mapping (from child labels to subtrees) that do not allow nodes to have the same labels.

5.4.3 General RDF Graphs

In this section, we present a recursive pattern on general RDF graphs, called matching strategy, that is useful to program update strategies for an RDF view defined on an RDF graph. We consider an example presented in [107] where a graph is transformed into a new graph by a query language (UnCAL) for graphs. Consider a rooted graph shown in Figure 5.3. The view graph is defined by a function f of two steps. Starting from the root of the graph, f first erases all edges until it reaches an a, and then copies the tree, but replaces every b with a c. Obviously, implementing such a forward transformation function f requires recursively traversing over all the edges of the graph. To implement f and a putback transformation (view update strategy) of the view graph using Datalog we first implement the forward transformation f by introducing an operator of eliminating ϵ edges, which are special edges of the graph. We then introduce a matching strategy that will be used to implement view update strategies for the graphs.

Forward transformation

The forward transformation consists of three steps:

1. From the root, erasing all the edges until reaching an edge `a`
2. Replacing each edge `b` with edge `c`
3. Eliminating ϵ edges

The first step can be simply implemented by finding all the edges, which are connected to the root by a path containing an edge `a`, and then replacing all the other edges with a special edge ϵ as the following:

```
1  a_descendant(rdf_src)(PARENT_ID) :-  rdf_src(_, "a",PARENT_ID).
2  a_descendant(rdf_src)(CHILD_ID) :-
3                          a_descendant(rdf_src)(PARENT_ID),
4                          rdf_src(PARENT_ID, _, CHILD_ID).
5
6  erase_until_a(rdf_src)(X, L, Y) :- rdf_src(X, L, Y), L="a".
7  erase_until_a(rdf_src)(X, L, Y) :- rdf_src(X, L, Y),
8                                     a_descendant(rdf_src)(Y).
9  erase_until_a(rdf_src)(X, ϵ, Y) :- rdf_src(X, L, Y),
10                                    not L = "a",
11                                    not a_descendant(rdf_src)(Y).
```

The first rule finds all the nodes that have incoming edge `a`. The second rule recursively collects all the descendants of the nodes found in the first rule. In this way, the first two rules define `a_descendant(rdf_src)` containing all the nodes that are connected to the root through edge `a` (and other edges). The third rule keeps all the edges `a` in the original graph `rdf_src`. The fourth rule keeps the incoming edges of all the nodes resulted by `a_descendant(rdf_src)`. The last rule replaces all the other edges with ϵ edges.

The second step simply maps each edge `b` to an edge `c`:

```
1  b_mapsto_c(rdf_src)(X, "c", Y) :- rdf_src(X, L, Y), L="b".
2  b_mapsto_c(rdf_src)(X, L, Y) :- rdf_src(X, L, Y), not L="b".
```

To eliminate all ϵ edges, for each node X that has an incoming edge, which is not ϵ, we find all the nodes Y that can be reached by X through only ϵ edges. We then copy all the outgoing edges of Y as outgoing edges of X as in the following definition of `extend_with_epsilon`:

```
1 extend_with_epsilon(rdf_src)(X, L, Y) :- rdf_src(X, L, Y),
2                                          not L = ϵ.
3 extend_with_epsilon(rdf_src)(X_SUBS, L, Y) :- rdf_src(X, L, Y),
      not L = ϵ, closure(epsilon_binary(rdf_src))(X_SUBS, X).
4
5 eliminate_epsilon(rdf_src)(X, L, Y) :-
6     rooted_graph(extend_with_epsilon(rdf_src), R)(X, L, Y),
7     root_of(rdf_src)(R).
```

Where the final rule extracts the graph resulted by `extend_with_epsilon` that have the same root as the original graph `rdf_src`

The view graph is defined by a composition of three steps as the following:

```
rdf_view(X, L, Y) :- eliminate_epsilon(b_mapsto_c(erase_until_a
   (rdf_src)))(X, L, Y).
```

Matching strategy

To implement view update strategies for the view graph, we now introduce a matching strategy as follows. When there are updates on the view such as edges that are inserted or deleted or modified, the essential issue is finding the corresponding edges in the source graph to insert or delete or modify. Our matching strategy finds all the correspondence between edges of the source and edges of the view as the following:

```
1 pair(matching)(X, L, Y) :- normal_pair(matching)(X, L, Y).
2 pair(matching)(X, L, Y) :- epsilon_pair((matching)X, L, Y).
```

`pair` takes as input a predicate `matching`, where `matching(X, Y)` indicates that node X of the source graph corresponds to node Y of the view graph.

Each triple X, L, Y in `pair` represents a pair of node X in the source graph and node Y in the view graph, label L indicates the types of pair that is either

"vertical" or "horizontal". There are two cases of computing the pairs:

- Normal pair: X is an existing node in the source graph and the view graph, respectively.
- ϵ pair: either X or Y does not exist and thus is inserted into the source or view graph with an ϵ edge.

The following Datalog program shows the main implementation of `normal_pair`. The full implementation of `pair` can be found in our code repository[6]:

```
1  normal_pair(src_root, "horizontal", view_root) :-
2      root_of_tree(edge_src)(src_root),
3      root_of_tree(edge_view)(view_root).
4  normal_pair(src_id, "vertical", view_id) :-
5      normal_pair(src_id, "horizontal", view_id),
6      vertical_match(src_id, view_id).
7  normal_pair(src_id, "vertical", other_view_id) :-
8      normal_pair(src_id, "horizontal", view_id),
9      src_will_match_view_closure(src_id, view_id,
       reachable_view_node),
10     edge_view(reachable_view_node, _, other_view_id),
11     vertical_match(src_id, other_view_id).
12 normal_pair(other_src_id, "vertical", view_id) :-
13     normal_pair(src_id, "horizontal", view_id),
14     not src_will_match_view_closure(src_id, view_id, view_id),
15     view_will_match_src_closure(view_id, src_id, src_id),
16     view_will_match_src_closure(view_id, src_id,
       reachable_src_node),
17     edge_src(reachable_src_node, _, other_src_id),
18     vertical_match(other_src_id, view_id).
19 normal_pair(src_child_node, "horizontal", view_child_node) :-
20     normal_pair(src_id, "horizontal", view_id),
21     not src_will_match_view_closure(src_id, view_id, view_id),
```

[6] https://github.com/dangtv/BIRDS

```
22      not view_will_match_src_closure(view_id, src_id, src_id),
23      edge_src(src_id, _, src_child_node),
24      minr(horizontal_match_view_childs(src_child_node,view_id))(
        view_child_node).
```

By having all the pairs, i.e. all the correspondence between nodes in the source and the view, we then reflect nodes and edges in the view to the corresponding nodes and edges in the source by a simple inner putback transformation. For example, we can use the putback transformation of `acond` [92], which changes the source edge to `b` if the view edge is `c`, and applies `identity` for other cases.

In our matching strategy, the input predicate `matching` and the inner bidirectional transformation `inner_bx` are iterated over all the nodes and edges of the graph. They are simple to write but decide the behaviour of the complete view update strategy.

5.5 Experiments

We have implemented a prototype for our proposed approach using Ocaml based on the framework presented in Chapter 3. Our framework allows programmers to define higher-order predicates as well as reuse pre-defined higher-order predicates in constructing new programs of view update strategies. The framework translates the user-written programs into Datalog programs without higher-order predicates that can be evaluated by Datalog engines.

To evaluate our approach in terms of expressiveness, we collect views from a variety of sources including: the literature ([92, 107, 108]), knowledge graph databases (D2RQ [109, 90], LUBM [110] and DBPedia[111]), examples of the W3C standard R2RML [39]. We have implemented a library that defines higher-order predicates of common recursive patterns and used these pre-defined predicates to write update strategies for the collected views. The results are summarized in the Table 5.1. The number of translated Datalog rules increases against the number of Datalog rules with higher-order predicates.

Table 5.1: View update strategies written in Datalog with higher-order predicates. ✓indicates that the property is validated.

ID	Dataset	Program	Rules	Translated rules	Validation (GetPut & PutGet)
1	R2RML	employee	9	10	✓
2	D2RQ	employee	9	10	✓
3	LUBM	professor	3	6	✓
4	DBPedia	have_child	21	21	✓
5	[92]	hoist	2	6	✓
6	[92]	plunge	2	6	✓
7	[92]	ccond	2	8	✓
8	[92]	acond	4	10	✓
9	[92]	fork	10	25	✓
10	[92]	map	22	120	✓
11	[107]	mutual_recursion	19	160	✓

5.6 Related Work

Using relational databases to store complex data structures such as trees and graphs have been attracted a lot of researchers. Edge shredding is a classical approach [112, 113] to storing tree-structured data such as XML or JSON documents as trinary relations and thus can be queried by the Datalog language. The Resource Description Framework (RDF) provides a flexible method for representing knowledge graphs as triples of the form (subject, predicate, object) that is compatible with Datalog [102].

Many works and practical systems [114, 109, 115] have been proposed to define virtual RDF graphs over relational databases. However, most of the mapping languages and RDF query languages for specifying RDF views are unidirectional in the sense that views provide read-only data access. Some initial effort has been devoted to solving the problem of updating such RDF views by translating SPARQL updates on the view to SQL update statements on the source database [89, 116, 117, 90, 118]. These existing works provide solutions for updating some simple types of RDF views and updates.

Datalog with stratified negation [43] has been shown to be expressive enough to represent every SPARQL query [105, 103, 104, 101, 102]. Therefore, Datalog has been used as a natural platform for the extensions of SPARQL with richer navigation and recursive capabilities [94, 119].

5.7 Conclusion

In this chapter, we have presented our approach of using Datalog (with recursion) for specifying view update strategies among relations and RDF graphs. Our key idea to overcome the challenges of both writing and validating recursive view update strategies is to formulate the program into two parts: a recursive part, which implements recursive patterns, and a non-recursive one, which is an inner update strategy. On the one hand, the recursive Datalog rules of the first part are predefined and pre-validated so that their well-behavedness is guaranteed. On the other hand, we allow non-expert users to manually write the inner update strategies in non-recursive Datalog, which are automatically validated.

By extending Datalog with a restricted form of higher-order syntax, we provide a convenient way to construct view update strategies based on the existing ones. Specifically, the results of pre-defined recursive Datalog rules are parameterized to be a higher-order predicate and to be recalled in user-written programs.

Importantly, we syntactically extend Datalog with higher-order predicates but maintain the first-order semantics of the Datalog program by a translation algorithm. The algorithm transforms all higher-order predicates into first-order predicates defined by Datalog rules without additional syntax such as functional symbols. Our higher-order syntax restriction not only ensures the translation algorithm is complete but also guarantees the expressiveness for writing view update strategies.

We have implemented our approach. We show the feasibility of our proposed approach by implementing a library of common recursive patterns and view update strategies for RDF views.

Our current implementation is based on a built-in Datalog engine. The framework can be further integrated with practical RDF database management systems and scalable Datalog engines to work with real-world big RDF graphs and views.

In our future work, we consider improving the systematic method for pre-defining and pre-validating common recursive patterns and view update strategies. There are theoretical results that the satisfiability of a fragment of Datalog with recursions and guarded negations is decidable. We believe the verification of

pre-defined recursive Datalog programs can be partially automated, and thus a more expressive and useful library of pre-defined programs can be implemented without much users' burden.

6 Conclusion

6.1 Summary

Views are very commonly used in database management systems and view update is an important mechanism that has many applications in practice. In general, the view definition is not injective because it is allowed to lose information in the view. Therefore, an update on the view can be propagated to the source in more than one way. Automatically choosing one strategy to propagate view updates to the source faces a lot of challenges because the chosen update strategy may not satisfy user intentions. The putback-based approach opens a new direction to resolve this ambiguity issue that rather than defining a view definition and automatically derive an update strategy for the view, we can explicitly specify an update strategy from the start and derive the corresponding view definition without ambiguity.

In this thesis, we have proposed a robust language-based approach to allow using Datalog, an expressive and user-friendly language, to completely program

view update strategies in database management systems. This is in sharp contrast to previous approaches in which the view defining queries are enriched to capture some certain update intentions. By allowing programmers to explicitly specify update strategies, our approach provides full programmability and flexibility to the programmers with the support for program validation, optimization, debugging, and compilation.

Since most views in relational database management systems are defined by SQL without recursion, in Chapter 3, we consider a fragment of the non-recursive Datalog language as a formal language for view update strategies and propose a validation algorithm to check the well-behavedness of user-written Datalog programs. We design an incrementalization method to optimize the Datalog programs before compiling them into SQL code that can run in a practical database.

Next, by considering the case that the user-written programs are invalid, we present in Chapter 4 an approach to generating counterexamples that explain to the users why the programs are not correct. We then design a debugging engine that assists the users in locating bugs in the programs that are useful to reduce the burden of correcting programs.

It has been shown in the Resource Description Framework (RDF) that complex data structures such as trees and graphs can be stored in relational databases as a special ternary relation, i.e., a set of triples. In Chapter 5, we extend our approach to use the Datalog language for specifying view update strategies for RDF databases. We use a restricted form of higher-order syntax of Datalog to provide a convenient way to encapsulate common recursive computation patterns that are useful for specific data structures. In this way, a set of common recursive patterns that are expressive enough to construct view update strategies is pre-defined and pre-validated.

We have implemented our framework as open-source software. The experimental results show the performance of our implementation in terms of both compilation time and running time.

6.2 Future Work

In our current approach, we consider fragments of Datalog and extensions as the language for specifying view update strategies. For Datalog, we consider only set semantics that a relation is a set of distinguished tuples. In most practical relational database management systems, a table can have duplicated tuples, thus it is considered as a bag of tuples. Therefore, one important future work of this thesis is extending the Datalog language with bag semantics so that duplicates in both the view and source database are allowed while the consistency between them is maintained. Although bag semantics for Datalog has been attracting researchers in recent years, how the view update strategies can be validated and optimized under bag semantics remains unclear.

We have considered several extensions of Datalog such as negations and comparison operators. Our approach can be further improved by considering more extensions including arithmetic/boolean expressions and simple functions in Datalog rules. These extensions allow programmers to conveniently describe more view update strategies but need to be carefully designed to guarantee the safety of Datalog programs so that they can be evaluated efficiently.

In another aspect, because an updatable view can be treated as a normal base table, there are security issues that should be considered in the future. The first issue arises when the view update strategies are defined by programmers, who do not have full permission to some underlying tables in the databases. We need a systematic way for verifying whether these view update strategies conform to the allowed permission. The second problem is managing data access when views are shared between different systems and updates to the shared views are made by untrusted users or untrusted sources. We need a mechanism for controlling access to the view as well as tracking the data provenance in view update strategies.

Bibliography

[1] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, Dec 2001.

[2] Behzad Golshan, Alon Halevy, George Mihaila, and Wang-Chiew Tan. Data integration: After the teenage years. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, pages 101–106, New York, NY, USA, 2017. ACM.

[3] Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao. Utilizing ids to accelerate incremental view maintenance. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1985–2000, New York, NY, USA, 2015. ACM.

[4] Scott W. Ambler and Pramodkumar J. Sadalage. *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley Professional, 2006.

[5] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[6] Y. Asano, D. Herr, Y. Ishihara, H. Kato, K. Nakano, M. Onizuka, and Y. Sasaki. Flexible framework for data integration and update propagation: System aspect. In *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 1–5, Feb 2019.

[7] Yasuhito Asano, Zhenjiang Hu, Yasunori Ishihara, Makoto Onizuka, Masato Takeichi, and Masatoshi Yoshikawa. Data integration models and architectures

for service alliances. In *4th International Workshop on Software Foundations for Data Interoperability, SFDI 2020, Tokyo, Japan, September 4, 2020, Proceedings*, volume 1281, pages 152–164. Springer, 2020.

[8] Kota Miyake, Yusuke Wakuta, Yuya Sasaki, and Makoto Onizuka. Towards guaranteeing global consistency for peer-based data integration architecture. In *4th International Workshop on Software Foundations for Data Interoperability, SFDI 2020, Tokyo, Japan, September 4, 2020, Proceedings*, volume 1281, pages 187–193. Springer, 2020.

[9] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, December 1981.

[10] Umeshwar Dayal and Philip A. Bernstein. On the updatability of relational views. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4*, VLDB '78, pages 368–377, 1978.

[11] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, September 1982.

[12] Ronald Fagin, Jeffrey D. Ullman, and Moshe Y. Vardi. On the semantics of updates in databases. In *Proceedings of the 2Nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '83, pages 352–365, New York, NY, USA, 1983. ACM.

[13] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 154–163, 1985.

[14] A. Keller. Choosing a view update translator by dialog at view definition time. In *VLDB*, pages 467–474, 1986.

[15] Y. Kotidis, D. Srivastava, and Y. Velegrakis. Updates through views: A new hope. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 2–2, April 2006.

[16] Benny Kimelfeld, Jan Vondrák, and Ryan Williams. Maximizing conjunctive views in deletion propagation. *ACM Trans. Database Syst.*, 37(4):24:1–24:37, December 2012.

[17] Yoshifumi Masunaga. An intention-based approach to the updatability of views in relational databases. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, IMCOM '17, pages 13:1–13:8, New York, NY, USA, 2017. ACM.

[18] James A Larson and Amit P Sheth. Updating relational views using knowledge at view definition and view update time. *Information Systems*, 16(2):145 – 168, 1991.

[19] Rom Langerak. View updates in relational databases with an independent scheme. *ACM Trans. Database Syst.*, 15(1):40–66, March 1990.

[20] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: A language for updatable views. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 338–347, 2006.

[21] Arthur M Keller and Jeffrey D Ullman. On complementary and independent mappings on databases. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 143–148, New York, NY, USA, 1984. ACM.

[22] Jens Lechtenbörger and Gottfried Vossen. On the computation of relational view complements. *ACM Trans. Database Syst.*, 28(2):175–208, June 2003.

[23] Yoshifumi Masunaga. A relational database view update translation mechanism. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 309–320. Morgan Kaufmann Publishers Inc., 1984.

[24] Yoshifumi Masunaga, Yugo Nagata, and Tatsuo Ishii. Extending the view updatability of relational databases from set semantics to bag semantics and its implementation on postgresql. In *Proceedings of the 12th International*

Conference on Ubiquitous Information Management and Communication, IMCOM '18, pages 19:1–19:8, New York, NY, USA, 2018. ACM.

[25] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[26] PostgreSQL. https://www.postgresql.org.

[27] PL/pgSQL. Sql procedural language. https://www.postgresql.org/docs/9.6/plpgsql.html.

[28] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[29] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, pages 260–283, 2009.

[30] Zhenjiang Hu, Hugo Pacheco, and Sebastian Fischer. Validity checking of putback transformations in bidirectional programming. In *FM 2014: Formal Methods*, pages 1–15, Cham, 2014. Springer International Publishing.

[31] Sebastian Fischer, ZhenJiang Hu, and Hugo Pacheco. The essence of bidirectional programming. *Science China Information Sciences*, 58(5):1–21, May 2015.

[32] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. A clear picture of lens laws. In *International Conference on Mathematics of Program Construction*, pages 215–223, 2015.

[33] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. Bigul: A formally verified core language for putback-based bidirectional programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 61–72, New York, NY, USA, 2016. ACM.

[34] Hsiang-Shang Ko and Zhenjiang Hu. An axiomatic basis for bidirectional programming. *Proc. ACM Program. Lang.*, 2(POPL):41:1–41:29, December 2017.

[35] Zirun Zhu, Yongzhe Zhang, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. Parsing and reflective printing, bidirectionally. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 2–14, New York, NY, USA, 2016. ACM.

[36] Van-Dang Tran, Hiroyuki Kato, and Zhenjiang Hu. Programmable view update strategies on relations. *PVLDB*, 13(5):726–739, 2020.

[37] Van-Dang Tran, Hiroyuki Kato, and Zhenjiang Hu. Birds: Programming view update strategies in datalog. *PVLDB*, 13(12):2897–2900, 2020.

[38] Van-Dang Tran, Hiroyuki Kato, and Zhenjiang Hu. A counterexample-guided debugger for non-recursive datalog. In *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020*, volume 12470, pages 323–342, 2020.

[39] Souripriya Das, Seema Sundara, and Richard Cyganiak. R2RML: RDB to RDF mapping language, W3C recommendation 27 september 2012, 2012. accessed 2022-02-09.

[40] Van-Dang Tran, Hiroyuki Kato, and Zhenjiang Hu. Toward recursive view update strategies on relations. In *Ninth International Workshop on Bidirectional Transformations (Bx 2021), June, 2021, Bergen, Norway*, 2021.

[41] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *TKDE*, 1(1):146–166, 1989.

[42] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *Journal of Web Semantics*, 14:57 – 83, 2012. Special Issue on Dealing with the Messiness of the Web of Data.

[43] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[44] Vince Bárány, Balder ten Cate, and Martin Otto. Queries with guarded negation. *Proc. VLDB Endow.*, 5(11):1328–1339, July 2012.

[45] Claudia Bauzer Medeiros and Frank Wm. Tompa. Understanding the implications of view update policies. *Algorithmica*, 1(1):337–360, Nov 1986.

[46] Kai Herrmann, Hannes Voigt, Andreas Behrend, Jonas Rausch, and Wolfgang Lehner. Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1101–1116, New York, NY, USA, 2017. ACM.

[47] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. On propagation of deletions and annotations through views. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 150–158, 2002.

[48] Rudi Horn, Roly Perera, and James Cheney. Incremental relational lenses. *Proc. ACM Program. Lang.*, 2(ICFP):74:1–74:30, July 2018.

[49] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 47–58, 2007.

[50] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. *SIGPLAN Not.*, 45(9):205–216, September 2010.

[51] Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on*

Functional Programming, ICFP '10, pages 193–204, New York, NY, USA, 2010. ACM.

[52] Tao Zan, Li Liu, Hsiang-Shang Ko, and Zhenjiang Hu. Brul: A putback-based bidirectional transformation library for updatable views. In *ETAPS*, pages 77–89, 2016.

[53] Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. Monadic combinators for "putback" style bidirectional programming. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 39–50, New York, NY, USA, 2014. ACM.

[54] Hugo Pacheco, Tao Zan, and Zhenjiang Hu. Biflux: A bidirectional functional update language for xml. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, PPDP '14, pages 147–158, New York, NY, USA, 2014. ACM.

[55] 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*, pages 157–166, 1993.

[56] Todd J. Green, Dan Olteanu, and Geoffrey Washburn. Live programming in the logicblox system: A metalogiql approach. *Proc. VLDB Endow.*, 8(12):1782–1791, August 2015.

[57] Christoph Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 87–98, 2010.

[58] Fernando Sáenz-Pérez, Rafael Caballero, and Yolanda García-Ruiz. A deductive database with datalog and sql query languages. In *Programming Languages and Systems*, pages 66–73, 2011.

[59] Vince Bárány, Balder Ten Cate, and Luc Segoufin. Guarded negation. *J. ACM*, 62(3):22:1–22:26, June 2015.

[60] Oded Shmueli. Equivalence of datalog queries is undecidable. *The Journal of Logic Programming*, 15(3):231 – 241, 1993.

[61] Z3: Theorem prover. https://z3prover.github.io.

[62] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 1999.

[63] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, 2 edition, 2008.

[64] MySQL Tutorial. http://www.mysqltutorial.org.

[65] Oracle Tutorial. https://www.oracletutorial.com.

[66] PostgreSQL Tutorial. http://www.postgresqltutorial.com.

[67] PostgreSQL 9.6.15 Documentation. https://www.postgresql.org/docs/9.6/.

[68] SQL Server Tutorial. http://www.sqlservertutorial.net.

[69] Database Administrators Stack Exchange. https://dba.stackexchange.com.

[70] Questions - Stack Overflow. https://stackoverflow.com/questions.

[71] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.*, 33(4):22:1–22:50, December 2008.

[72] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB*, pages 675–686, 2007.

[73] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.

[74] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

[75] Rafael Caballero, Adrián Riesco, and Josep Silva. A survey of algorithmic debugging. *ACM Comput. Surv.*, 50(4):60:1–60:35, August 2017.

[76] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. A theoretical framework for the declarative debugging of datalog programs. In *Semantics in Data and Knowledge Bases*, pages 143–159, 2008.

[77] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. A new proposal for debugging datalog programs. In *WFLP'07*, 2007.

[78] Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. Declarative datalog debugging for mere mortals. In *Datalog in Academia and Industry*, pages 111–122, 2012.

[79] Cláudio Amaral, Mário Florido, and Vítor Santos Costa. Prologcheck – property-based testing in prolog. In *Functional and Logic Programming*, pages 1–17, 2014.

[80] Seokki Lee, Sven Köhler, Bertram Ludäscher, and Boris Glavic. A sql-middleware unifying why and why-not provenance for first-order queries. In *ICDE*, pages 485–496, 2017.

[81] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog±: A unified approach to ontologies and integrity constraints. In *ICDT*, pages 14–30, 2009.

[82] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, pages 530–541, 2014.

[83] Ehud Y. Shapiro. Algorithmic program diagnosis. In *POPL*, pages 299–308, 1982.

[84] Melanie Herschel and Mauricio A. Hernández. Explaining missing answers to SPJUA queries. *PVLDB*, 3(1):185–196, 2010.

[85] Sven Köhler, Bertram Ludäscher, and Daniel Zinn. First-order provenance games. In *In Search of Elegance in the Theory and Practice of Computation*, pages 382–399, 2013.

[86] Alban Gabillon and Léo Letouzey. A view based access control model for SPARQL. In *Fourth International Conference on Network and System*

Security, NSS 2010, Melbourne, Victoria, Australia, September 1-3, 2010, pages 105–112. IEEE Computer Society, 2010.

[87] Edward Hung, Yu Deng, and V. S. Subrahmanian. RDF aggregate queries and views. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 717–728. IEEE Computer Society, 2005.

[88] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *J. Data Semant.*, 10:133–173, 2008.

[89] Matthias Hert, Gerald Reif, and Harald C. Gall. Updating relational data via sparql/update. In *Proceedings of the 2010 EDBT/ICDT Workshops, Lausanne, Switzerland, March 22-26, 2010*, ACM International Conference Proceeding Series, 2010.

[90] Vadim Eisenberg and Yaron Kanza. D2rq/update: updating relational data via virtual RDF. In *Proceedings of the 21st World Wide Web Conference, WWW 2012, Lyon, France, April 16-20, 2012 (Companion Volume)*, pages 497–498. ACM, 2012.

[91] Sunitha Ramanujam, Vaibhav Khadilkar, Latifur Khan, Murat Kantarcioglu, Bhavani M. Thuraisingham, and Steven Seida. Update-enabled triplification of relational data into virtual RDF stores. *Int. J. Semantic Comput.*, 4(4):423–451, 2010.

[92] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246, 2005.

[93] Xiao He, Xing Chen, Sibo Cai, Ying Zhang, and Gang Huang. Testing bidirectional model transformation using metamorphic testing. *Inf. Softw. Technol.*, 104:109–129, 2018.

[94] Leonid Libkin, Juan L. Reutter, and Domagoj Vrgoc. Trial for RDF: adapting graph query languages for RDF data. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 201–212. ACM, 2013.

[95] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. Expressive languages for querying the semantic web. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 14–26. ACM, 2014.

[96] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. Rdfox: A highly-scalable RDF store. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, volume 9367 of *Lecture Notes in Computer Science*, pages 3–20, 2015.

[97] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. The vadalog system: Datalog-based reasoning for knowledge graphs. *VLDB*, 11(9):975–987, 2018.

[98] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoc. Recursion in SPARQL. *Semantic Web*, 12(5):711–740, 2021.

[99] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. Sparql 1.1 query language. *W3C recommendation*, 21(10):778, 2013.

[100] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. nsparql: A navigational language for RDF. *J. Web Semant.*, 8(4):255–270, 2010.

[101] Axel Polleres. From SPARQL to rules (and back). In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 787–796. ACM, 2007.

[102] Simon Schenk. A SPARQL semantics based on datalog. In *KI 2007: Advances in Artificial Intelligence, 30th Annual German Conference on AI, KI 2007,*

Osnabrück, Germany, September 10-13, 2007, Proceedings, volume 4667 of *Lecture Notes in Computer Science*, pages 160–174, 2007.

[103] Renzo Angles and Claudio Gutiérrez. The expressive power of SPARQL. In *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*, volume 5318 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2008.

[104] Marcelo Arenas, Claudio Gutiérrez, and Jorge Pérez. Foundations of RDF databases. In *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, volume 5689 of *Lecture Notes in Computer Science*, pages 158–204, 2009.

[105] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Semant.*, 7(2):57–73, 2009.

[106] Weidong Chen, Michael Kifer, and David Scott Warren. HILOG: A foundation for higher-order logic programming. *J. Log. Program.*, 15(3):187–230, 1993.

[107] Peter Buneman, Mary F. Fernandez, and Dan Suciu. Unql: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.

[108] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.

[109] Christian Bizer and Andy Seaborne. D2rq-treating non-rdf databases as virtual rdf graphs. In *Proceedings of the 3rd international semantic web conference (ISWC2004)*, volume 2004. Springer, 2004.

[110] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.*, 3(2-3):158–182, 2005.

[111] Wikidata Query Service. https://query.wikidata.org.

[112] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[113] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 204–215, 2002.

[114] Orri Erling and Ivan Mikhailov. RDF support in the virtuoso DBMS. In *Networked Knowledge - Networked Media - Integrating Knowledge Management, New Media Technologies and Semantic Systems*, volume 221 of *Studies in Computational Intelligence*, pages 7–24. 2009.

[115] Sören Auer, Sebastian Dietzold, Jens Lehmann, Sebastian Hellmann, and David Aumueller. Triplify: light-weight linked data publication from relational databases. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 621–630. ACM, 2009.

[116] Raphael Volz, Daniel Oberle, and Rudi Studer. Towards views in the semantic web. In *2nd Int'l Workshop on Databases, Documents and Information Fusion (DBFUSION02)*, 2002.

[117] Nikos Bikakis, Chrisa Tsinaraki, Ioannis Stavrakantonakis, and Stavros Christodoulakis. Supporting SPARQL update queries in RDF-XML integration. In *Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014*, volume 1272 of *CEUR Workshop Proceedings*, pages 245–248. CEUR-WS.org, 2014.

[118] Sunitha Ramanujam, Vaibhav Khadilkar, Latifur Khan, Steven Seida, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Bi-directional translation of relational data into virtual RDF stores. In *Proceedings of the 4th IEEE*

International Conference on Semantic Computing (ICSC 2010), September 22-24, 2010, Carnegie Mellon University, Pittsburgh, PA, USA, pages 268–276. IEEE Computer Society, 2010.

[119] Sebastian Rudolph and Markus Krötzsch. Flag & check: data access with monadically defined queries. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 151–162. ACM, 2013.

A

More details of Chapter 3

A.1 Proofs

A.1.1 Proof of Lemma 3.2.7

Proof. Let P be a Datalog program in nonrecursive GN-Datalog with equalities, constants, and comparisons. We shall transform a query (P, R), where R is an IDB relation corresponding to IDB predicate r in P, into an equivalent guarded negation first-order (GNFO) formula [59]. Without loss of generality, we assume that in P, for every pair of head atoms $h_1(\vec{X}_1)$, $h_2(\vec{X}_2)$ in P, $h_1 = h_2$ implies $\vec{X}_1 = \vec{X}_2$ (this can be achieved by variable renaming).

Since there are constants that can occur in both atoms and equalities. We first remove all constants appearing in atoms by transforming them into constants appearing in equalities. This can be done by introducing a fresh variable X for each constant c in the atoms of the Datalog rule (head or body), then adding an equality $X = c$ to the rule body and substituting X for the constant c. By this

transformation, we consider equalities of the form $X = c$ and a positive atom as a guard for negative predicates or head atom of Datalog rules. In other words, for each head atom or negative predicate β, there is a positive atom $p(\vec{Y})$ such that all the free variables in β must appear in $p(\vec{Y})$ or in an equality of the form $X = c$. For example, the following rule

$$h(Z, 1) :- p(Z, W, 3), \neg r(W, 4).$$

is transformed into

$$\begin{aligned} h(Z, X_1) :- & p(Z, W, X_2), \neg r(W, X_3), X_1 = 1, X_2 = 3, \\ & X_3 = 4. \end{aligned}$$

in which the negated atom $r(W, X_3)$ is guarded by the positive atom $p(Z, W, X_2)$ and the equality $X_3 = 4$. The head atom $h(Z, X_1)$ is guarded by $p(Z, W, X_2)$ and $X_1 = 1$.

We shall define a FO formula φ_r equivalent to the Datalog query (P, R), i.e, for every database D, the IDB relation R (corresponding to IDB predicate r in P) in the output of P over D (denoted as $P(D)|_R$) is the same as the set of tuple $\vec{t}$ satisfying φ_r ($\{\vec{t} \mid \varphi(\vec{t})\}$). The construction of φ_r is inductively defined as the following:

- (Base case) r is an EDB predicate, i.e., $r \in \mathcal{S} \cup \{v\}$: $\varphi_r = r(\vec{X}_r)$, where $\vec{X}_r$ denotes $(X_1, \ldots, X_{arity(r)})$.

- (Inductive case) r is an IDB predicate, i.e., r occurs in the head of some rules. Suppose that there are m rules:

$$\begin{aligned} & r(\vec{X}_r) :- \alpha_{1,1}, \ldots, \alpha_{1,n_1}. \\ & \ldots \\ & r(\vec{X}_r) :- \alpha_{m,1}, \ldots, \alpha_{m,n_m}. \end{aligned}$$

Let $\varphi_{r,i}(\vec{X}_r)$ be the FO formula for r when considering only the i-th rule:

$$\varphi_{r,i}(\vec{X}_r) = \exists \vec{E}_i, \bigwedge_{j=1}^{n_i} \beta_{i,j}$$

where $\vec{E}_i$ contains the bound variables of the i-th rule (variables not in the rule head),

$$\beta_{i,j} = \begin{cases} \varphi_w(\vec{Z}), \text{ if } \alpha_{i,j} \text{ is an atom } w(\vec{Z}) \\ \neg\varphi_w(\vec{Z}), \text{ if } \alpha_{i,j} \text{ is a negated atom } \neg w(\vec{Z}) \\ \alpha_{i,j} \text{ if } \alpha_{i,j} \text{ is an equality or a negated equality} \\ C_{<c}(X) \text{ if } \alpha_{i,j} \text{ is a comparison predicate } X < c \\ C_{>c}(X) \text{ if } \alpha_{i,j} \text{ is a comparison predicate } X > c \end{cases}$$

Here we introduce fresh predicates $C_{<c}(X)$ and $C_{>c}(X)$ for the comparisons. We have:

$$\varphi_r(\vec{X}_r) = \bigvee_{i=1}^{m} \varphi_{r,i}(\vec{X}_r) = \bigvee_{i=1}^{m} \left(\exists \vec{E}_i, \bigwedge_{j=1}^{n_i} \beta_{i,j} \right)$$

It is not difficult to show that φ_r is equivalent to the Datalog query (P, R). Indeed, for any database instance D, by induction, we can show that for each IDB predicate r and each tuple $\vec{t}$,

$$r(\vec{t}) \in P(D) \Leftrightarrow D \models \varphi_r(\vec{t})$$

In each conjunction $\varphi_{r,i}(\vec{X}_r) = \exists \vec{E}_i, \bigwedge_{j=1}^{n_i} \beta_{i,j}$, each negative predicate $\beta_{i,j}$ is guarded by a positive atom $w_{i,j}(\vec{Y})$ and many equalities. Moreover, there exists a positive atom $w_i(\vec{Y})$ containing all the free variables of $\vec{X}_r$.

Let us briefly recall the syntax of GNFO formulas with constants proposed by Bárány et al. [59]. GNFO formulas with constants are generated by the following

definition:

$$\varphi ::= r(t_1, \ldots, t_n) \mid t_1 = t_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \exists x\varphi \mid \alpha \wedge \neg\varphi$$

where each t_i is either a variable or a constant symbol, and, in $\alpha \wedge \neg\varphi$, α is an atomic formula of EDB predicate containing all free variables of φ.

We now transform $\varphi_r(\vec{X_r})$ into a GNFO formula by structural induction on $\varphi_r(\vec{X_r})$. Since GNFO is close under disjunction ($\varphi_1 \vee \varphi_2$), we transform each conjunction $\varphi_{r,i}(\vec{X_r})$ in the formula $\varphi_r(\vec{X_r})$ into a GNFO formula. We first group each negative predicate $\beta_{i,j}$ with its guard atom $w_{i,j}(\vec{Y})$. If a free variable X appears in $\beta_{i,j}$ but not in $w_{i,j}(\vec{Y})$, X must appear in an equality $X = c$, we then substitute c for X in $\beta_{i,j}$ and obtain $\varphi_{w_{i,j}}(\vec{Y}) \wedge \beta_{i,j}$, where $\vec{Y}$ contains all the free variable of $\beta_{i,j}$. If two negative predicates share the same guard atom then the guard atom can be used twice.

$$\varphi_{r,i}(\vec{X_r}) = \exists \vec{E_i}, \left(\bigwedge_k \beta_{i,k} \right) \wedge \left(\bigwedge_j (\varphi_{w_{i,j}}(\vec{Y}) \wedge \beta_{i,j}) \right)$$

Because each $\beta_{i,k}$ in $(\bigwedge_k \beta_{i,k})$ is a positive predicate, we inductively transform each $\beta_{i,k}$ into a GNFO formula. Now consider each formula $\psi = (\varphi_{w_{i,j}}(\vec{Y}) \wedge \beta_{i,j})$.

- If $w_{i,j}$ is an EDB predicate, $\varphi_{w_{i,j}}(\vec{Y}) = w_{i,j}(\vec{Y})$, thus ψ is a GNFO formula.
- If $w_{i,j}$ is an IDB predicate, by the construction of $\varphi_{w_{i,j}}(\vec{Y})$, we have $\varphi_{w_{i,j}}(\vec{Y}) = \bigvee_l \varphi_{w^l_{i,j}}(\vec{Y})$. As mentioned before in each $\varphi_{w^l_{i,j}}(\vec{Y})$ there is an IDB atom $u_l(\vec{Z})$ containing all variables of $\vec{Y}$. Therefore,

$$\begin{aligned} \psi &= \left(\bigvee_l \varphi_{w^l_{i,j}}(\vec{Y}) \right) \wedge \beta_{i,j} \\ &\equiv \bigvee_l \varphi_{w^l_{i,j}}(\vec{Y}) \wedge \beta_{i,j} \\ &\equiv \bigvee_l \varphi_{w^l_{i,j}}(\vec{Y}) \wedge (\varphi_{u_l}(\vec{Z}) \wedge \beta_{i,j}) \end{aligned}$$

We continue to inductively transform each $\varphi_{w^l_{i,j}}(\vec{Y})$ and $\varphi_{u_l}(\vec{Z}) \wedge \beta_{i,j}$ into a

GNFO formula.

In this way, each formula $\varphi_{w_{i,j}}(\vec{Y}) \wedge \beta_{i,j}$ is transformed into a GNFO formula. Since GNFO is close under conjunction and existential quantifier, $\varphi_r(\vec{X_r})$ is transformed into a GNFO formula.

We have constructed an equivalent GNFO formula $\varphi_r(\vec{X_r})$ for the Datalog query (P, R). It is remarkable that in this transformation, we have introduced many predicate symbols $C_{<c}(X)$ and $C_{>c}(X)$ for built-in predicates $<$ and $>$ in P. The introduction of new predicates $C_{<c}(X)$ and $C_{>c}(X)$ does not preserve the meaning of comparison symbols $<$ and $>$. Therefore, to reduce the satisfiability of Datalog query (P, R) to the satisfiability of $\varphi_r(\vec{X_r})$, we need an axiomatization for the built-in comparison predicates. We construct a GNFO sentence for this axiomatization by using the similar technique for GN-SQL(LIN) by Bárány et al. [44]. Let the set of constant symbols in P be $\{c_1, \ldots, c_n\}$, which is a finite subset of a totally ordered domain dom, with $c_1 < c_2 < \ldots < c_n$. The GNFO sentence that axiomatizes built-in comparison predicates is as follows:

$$\Phi = \forall X, \varphi_{X<c_1} \vee \varphi_{X=c_1} \vee \varphi_{c_1<X<c_2} \vee \varphi_{X=c_2} \vee \ldots \vee \varphi_{X>c_n}$$

where

$$\varphi_{X<c_1} = \begin{cases} \bigwedge\limits_{i \leq n} (C_{<c_i}(X) \wedge \neg(X = c_i) \wedge \neg C_{>c_i}(X)) \\ \qquad \text{if } \exists c \in dom, c < c_1 \\ \bot \qquad \text{otherwise} \end{cases}$$

$$\begin{aligned} \varphi_{X=c_i} = \; & (X = c_i) \wedge \neg C_{<c_i}(X) \wedge \neg C_{>c_i}(X) \wedge \\ & \left(\textstyle\bigwedge_{j<i}(C_{>c_j}(X) \wedge \neg(X = c_j) \wedge \neg C_{<c_j}(X))\right) \wedge \\ & \left(\textstyle\bigwedge_{j>i}(\neg C_{>c_j}(X) \wedge \neg(X = c_j) \wedge C_{<c_j}(X))\right) \end{aligned}$$

$$\varphi_{c_i<X<c_{i+1}} = \begin{cases} \left(\bigwedge_{j \leq i}(C_{>c_j}(X) \wedge \neg(X = c_j) \wedge \neg C_{<c_j}(X))\right) \wedge \\ \left(\bigwedge_{j>i}(\neg C_{>c_i}(X) \wedge \neg(X = c_i) \wedge C_{<c_i}(X))\right) \\ \qquad \text{if } \exists c \in dom, c_i < c < c_{i+1} \\ \bot \qquad \text{otherwise} \end{cases}$$

$$\varphi_{X>c_1} = \begin{cases} \bigwedge\limits_{i\leq n}(\neg C_{<c_i}(X) \wedge \neg(X = c_i) \wedge C_{>c_i}(X)) \\ \qquad\qquad \text{if } \exists c \in dom, c > c_n \\ \bot \qquad \text{otherwise} \end{cases}$$

By this way, the Datalog query (P, R) is satisfiable if and only if the GNFO sentence $\Phi \wedge \varphi_r(\vec{X_r})$ is satisfiable. Indeed, if there is a database D over that the query (P, R) is not empty, we can construct a signature D' by copying all relations from D and use all (finite) the suitable values of the active domain of D to construct a relation corresponding to each predicate $C_{<c_i}(X)/C_{>c_i}(X)$. Clearly, D' satisfies Φ and $\varphi_r(\vec{X_r})$. Conversely, if there is a signature D' that satisfies Φ and $\varphi_r(\vec{X_r})$ we can construct a database D by an isomorphic copy of all relations from D' except the relations corresponding to predicates $C_{<c_i}(X)$ and $C_{>c_i}(X)$. It is known that for GNFO formulas, satisfiability over finite structure coincides with satisfiability over unrestricted structures. In other words, any structures satisfying the GNFO formula are finite. Therefore D' is a finite structure, i.e. a database. Since the satisfiability of a GNFO sentence is decidable, the satisfiability of the Datalog query (P, R) is also decidable. □

A.1.2 Proof of Theorem 3.2.8

Proof. As in Lemma 3.2.7, we first transform a query Q in nonrecursive GN-Datalog with equalities, constants, and comparisons into an equivalent guarded negation first-order formula $\varphi_r(\vec{Y})$. The result of Q over a database D is not empty iff D satisfies the sentence $\exists\vec{Y}, \varphi_r(\vec{Y})$. Let Σ be a set of guarded negation constraints and $\sigma_i = \forall\vec{X_i}, \Phi_i(\vec{X_i}) \rightarrow \bot$ $(i \in [1, m])$ be a constraint in Σ, where $\Phi_i(\vec{X_i})$ is a conjunction of (negative) atoms. Clearly, each $\Phi_i(\vec{X_i})$ is a guarded negation formula since there is a *guard* atom in the rule body $\Phi_i(\vec{X_i})$. We rewrite σ_i as an equivalent sentence $\sigma_i \equiv \neg\exists\vec{X_i}, \Phi_i(\vec{X_i})$. Now, the query Q is satisfiable under Σ iff there exists a database D satisfying all σ_i such that D satisfies $\exists\vec{Y}, \varphi_r(\vec{Y})$. This means that we need to check whether there exists a database D such that D satisfies all σ_i and $\exists\vec{Y}, \varphi_r(\vec{Y})$: $D \models (\bigwedge_{i=1}^{m} \neg\exists\vec{X_i}, \Phi_1(\vec{X_i})) \wedge (\exists\vec{Y}, \varphi_r(\vec{Y}))$. Note that there is no free variable in $\exists\vec{X_i}, \Phi_i(\vec{X_i})$ $(i \in [1, m])$ and all $\Phi_1, \ldots, \Phi_m$ and $\varphi_r(\vec{Y})$ are GNFO formulas, the conjunction $(\bigwedge_{i=1}^{m} \neg\exists\vec{X_i}, \Phi_1(\vec{X_I})) \wedge (\exists\vec{Y}, \varphi_r(\vec{Y}))$ is

a GNFO formula. Thus, the problem now is reduced to the satisfiability of a GNFO formula, which is decidable. □

A.1.3 Proof of Lemma 3.3.1

Proof. From Definition 2.3.1, we know that there exists a view definition get^d that satisfies both GetPut and PutGet with the given valid *put*. Let *get* be an arbitrary view definition satisfying GetPut with *put*, i.e., $put(S, get(S)) = S$ for any S. By applying the query get^d to both the right-hand side and left-hand side of this equation, and using the PutGet property of get^d and *put*, we obtain:

$$get^d(put(S, get(S))) = get^d(S) \Leftrightarrow get(S) = get^d(S)$$

This means that $get(S) = get^d(S)$ for any S, i.e., *get* and get^d are the same. Thus, *get* satisfies PutGet with *put*.

□

A.1.4 Proof of Lemma 3.3.3

Let $\langle r_1, \ldots, r_n \rangle$ be a source database schema and S be a database instance of this schema, i.e., S contains all relations $R_1, \ldots, R_n$ corresponding to the schema $r_1, \ldots, r_n$. Let v be a view over the source database. Let Σ be a set of m guarded negation constraints over the view and the source database; each constraint is of the form $\sigma_i = \forall \vec{X_i}, \Phi_{\sigma_i}(\vec{X_i}) \rightarrow \bot$.

Let us consider an LVGN-Datalog putback program *putdelta* for the view v. *putdelta* takes a (updated) view instance V and the original source database S to result in a delta ΔS of the source. V is a steady state of the view if ΔS has no effect on the original S, i.e., $S \oplus \Delta S = S$. Recall that ΔS contains all the tuples that need to be inserted/deleted into/from each source relation R_i ($i \in [1, n]$), represented by two sets $\Delta^+_{R_i}$ and $\Delta^-_{R_i}$ for these insertions and deletions, respectively. $S \oplus \Delta S = S$ iff

$$\Delta^-_{R_i} \cap R_i = \Delta^+_{R_i} \setminus R_i = \emptyset, \forall i \in [1, n] \tag{A.1}$$

Note that each $\Delta^+_{R_i}/\Delta^-_{R_i}$ is the IDB relation corresponding to delta predicate $+r_i/-r_i$ in the result of the Datalog program *putdelta* over the view and source database (S, V). Since *putdelta* is nonrecursive, we have an equivalent relational calculus query $\varphi_{-r_i}(\vec{X_i})/\varphi_{+r_i}(\vec{X_i})$ for each $\Delta^+_{R_i}/\Delta^-_{R_i}$. Equation (A.1) is equivalent to the condition that two relational calculus queries $\varphi_{-r_i}(\vec{X_i}) \wedge r_i(\vec{X_i})$ and $\varphi_{+r_i}(\vec{X_i}) \wedge \neg r_i(\vec{X_i})$ must be empty over the view and source database (S, V). In other words, the first-order sentences $\exists \vec{X_i}, \varphi_{-r_i}(\vec{X_i}) \wedge r_i(\vec{X_i})$ and $\exists \vec{X_i}, \varphi_{+r_i}(\vec{X_i}) \wedge \neg r_i(\vec{X_i})$ are not satisfiable over the view and source database (S, V). Combined with the constraint set Σ, a steady-state view V satisfies Σ and $S \oplus putdelta(S, V) = S$ iff:

$$\begin{cases} (S,V) \not\models \exists \vec{X_i}, \varphi_{-r_i}(\vec{X_i}) \wedge r_i(\vec{X_i}), i \in [1,n] \\ (S,V) \not\models \exists \vec{X_i}, \varphi_{+r_i}(\vec{X_i}) \wedge \neg r_i(\vec{X_i}), i \in [1,n] \\ (S,V) \not\models \exists \vec{X_i}, \Phi_{\sigma_i}(\vec{X_i}), i \in [n+1, n+m] \end{cases} \tag{A.2}$$

where $\vec{X_i}$ denotes a tuple of variables. Note that $(S, V) \not\models \xi_1$ and $(S, V) \not\models \xi_2$ are equivalent to $(S, V) \not\models \xi_1 \vee \xi_2$. Thus, we have:

$$\begin{cases} (S,V) \not\models \exists \vec{X_i}, (\varphi_{-r_i}(\vec{X_i}) \wedge r_i(\vec{X_i})) \vee \\ \qquad\qquad (\varphi_{+r_i}(\vec{X_i}) \wedge \neg r_i(\vec{X_i})), i \in [1,n] \\ (S,V) \not\models \exists \vec{X_i}, \Phi_{\sigma_i}(\vec{X_i}), i \in [n+1, n+m] \end{cases} \tag{A.3}$$

We now find such a V satisfying $(A.3)$.

Claim A.1.1 *Given a putback program putdelta written in LVGN-Datalog for a view v and a source schema $\langle r_1, \ldots, r_n \rangle$, each relational calculus formula $\varphi_r(\vec{X_r})$ of the query $(putdelta, R)$, where R is an IDB relation corresponding to IDB predicate r in P, can be rewritten in the following linear-view form:*

$$\left(\bigvee_{k=1}^{p} \exists \vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi_{1k} \right) \vee \left(\bigvee_{k=1}^{q} \exists \vec{E}_{2k}, \neg v(\vec{Y}_{2k}) \wedge \psi_{2k} \right) \vee \psi_3$$

where view atom v does not appear in ψ_{1k}, ψ_{2k} or ψ_3. Each of the formulas $\exists \vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi_{1k}$, $\exists \vec{E}_{2k}, \neg v(\vec{Y}_{2k}) \wedge \psi_{1k}$ and ψ_3 is a safe-range GNFO formula and has the same free variables $\vec{X_r}$.

Proof. The proof is conducted inductively on the transformation (presented in Sub-

section A.1.1 - the proof of Lemma 3.2.7) between the Datalog query $(putdelta, R)$ and an equivalent GNFO formula $\varphi_r(\vec{X}_r)$. Note that in this transformation, each $\varphi_{r,i}$ is a safe-range[1] formula, i.e. is a relational calculus [43].

We inductively prove that every φ_r can be transformed into the linear-view form. The base case is trivial. For the inductive case, due to the linear-view restriction, if r is a normal predicate (not a delta predicate), then there is no view atom v in all the rules defining r; thus, $\varphi_r = \bigvee_{i=0}^{m} \varphi_{r,i}$ is in the linear-view form, where $\psi_3 = \bigvee_{i=0}^{m} \varphi_{r,i}$ and $p = q = 0$. On the other hand, if r is a delta predicate, in each i-th rule $r(\vec{X}_r) :- \alpha_{i,1}, \ldots, \alpha_{i,n_1}$, there are two cases. The first case is that there is no α_{i,j_0} of a view atom v, $\varphi_{r,i} = \exists \vec{E}_i, \bigwedge_{j=0}^{n_i} \beta_{i,j}$ is in the linear-view form, where $\psi_3 = \varphi_{r,i}$ and $p = q = 0$. In the second case, there is only one α_{i,j_0}, which is an atom $v(\vec{Y}_i)$ or a negated atom $\neg v(\vec{Y}_i)$. Thus, $\varphi_{r,i} = \exists \vec{E}_i, v(\vec{Y}_i) \wedge \bigwedge_{j=0, j\neq j_0}^{n_i} \beta_{i,j}$ or $\varphi_{r,i} = \exists \vec{E}_i, \neg v(\vec{Y}_i) \wedge \bigwedge_{j=0, j\neq j_0}^{n_i} \beta_{i,j}$. Therefore, $\varphi_{r,i}$ is rewritten in the linear-view form. Note that if two formulas are in the linear-view form, then the disjunction of them can be transformed into the linear-view form. Indeed,

$$
\begin{aligned}
&\left(\bigvee_{k=1}^{p_1} \exists \vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi_{1k}\right) \vee \left(\bigvee_{k=1}^{q_1} \exists \vec{E}_{2k}, \neg v(\vec{Y}_{2k}) \wedge \psi_{2k}\right) \\
&\vee \psi_3 \vee \\
&\left(\bigvee_{k=p_1+1}^{p_2} \exists \vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi_{1k}\right) \vee \\
&\qquad\qquad \left(\bigvee_{k=q_1+1}^{q_2} \exists \vec{E}_{2k}, \neg v(\vec{Y}_{2k}) \wedge \psi_{2k}\right) \vee \psi'_3 \\
&\equiv \left(\bigvee_{k=1}^{p_2} \exists \vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi_{1k}\right) \vee \\
&\qquad\qquad \left(\bigvee_{k=1}^{q_2} \exists \vec{E}_{2k}, \neg v(\vec{Y}_{2k}) \wedge \psi_{2k}\right) \vee (\psi_3 \vee \psi'_3)
\end{aligned}
$$

In this way, $\varphi_r = \bigvee_{i=1}^{m} \varphi_{r,i}$ is rewritten in the linear-view form.

[1] A first-order formula ψ is a safe-range formula if all variables in ψ are range restricted [43]. In fact, for each nonrecursive Datalog query with negation, there is an equivalent safe-range first-order formula, and vice versa [43].

As proven in [44], we can continue to transform each safe-range formula $\varphi_{r,i}$ into a GNFO formula. In other words, in the linear-view form of φ_r, each $\exists\vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi_{1k}$ and $\exists\vec{E}_{2k}, \neg v(\vec{Y}_{2k}) \wedge \psi_{2k}$, and ψ_3 can be transformed into a safe-rage GNFO formula. In this transformation, if $\exists\vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi_{1k}$ is transformed into $\exists\vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi'_{1k} \vee v(\vec{Y}_{1k}) \wedge \psi''_{1k}$, we will transform it into $(\exists\vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi'_{1k}) \vee (\vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi''_{1k})$. In this way, we finally obtain a safe-range GNFO formula of φ_r, which is also in the linear-view form. □

We now know that the relational calculus formula $\varphi_{\pm r}$ of each delta predicate $\pm r$ is rewritten in the linear-view form. For each constraint σ_i of the form $\forall\vec{X}_i, \Phi_{\sigma_i}(\vec{X}_i) \rightarrow \bot$, we can also transform the conjunction $\Phi_{\sigma_i}(\vec{X}_i)$ into the linear-view form. Indeed, let us consider a new Datalog rule in *putdelta* as the following:

$$b_i(\vec{X}_i) :- \Phi_{\sigma_i}(\vec{X}_i).$$

in which the view is linearly used. The conjunction $\Phi_{\sigma_i}(\vec{X}_i)$ is equivalent to the relational calculus query $\varphi_{b_i}(\vec{X}_i)$ of relation b_i, which can be transformed into the linear-view form.

Since $\varphi_{\pm r}(\vec{X}_r)$ can be rewritten in the linear-view form, the conjunction $\varphi_{\pm r}(\vec{X}_r) \wedge r(\vec{X}_r)$ can be rewritten in the linear-view form by applying the distribution of existential quantifier over disjunction:

$$\varphi_{\pm r}(\vec{X}_r) \wedge r(\vec{X}_r) \equiv \left(\bigvee_{k=1}^{p} r(\vec{X}_r) \wedge \exists\vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge \psi_{1k}\right) \vee \left(\bigvee_{k=1}^{q} r(\vec{X}_r) \wedge \exists\vec{E}_{2k}, \neg v(\vec{Y}_{2k}) \wedge \psi_{2k}\right) \vee (r(\vec{X}_r) \wedge \psi_3)$$

$\vec{X}_r$ is the free variable of $\varphi_r(\vec{X}_r)$; hence, no existential variable in $\vec{E}_{1k}$ or $\vec{E}_{2k}$ is in $\vec{X}_r$. We can push $r(\vec{X}_r)$ into the existential quantifier $\exists\vec{E}_{1k}/\exists\vec{E}_{2k}$ and obtain:

$$\left(\bigvee_{k=1}^{p} \exists\vec{E}_{1k}, v(\vec{Y}_{1k}) \wedge r(\vec{X}_r) \wedge \psi_{1k}\right) \vee \left(\bigvee_{k=1}^{q} \exists\vec{E}_{2k}, \neg v(\vec{Y}_{2k}) \wedge r(\vec{X}_r) \wedge \psi_{2k}\right) \vee (r(\vec{X}_r) \wedge \psi_3)$$

This is in the linear-view form. Therefore, the disjunction $(\varphi_{+r}(\vec{X_r}) \wedge r(\vec{X_r})) \vee \varphi_{-r}(\vec{X_r}) \wedge r(\vec{X_r})$ can be rewritten in the linear-view form. The constraint (A.3) is now rewritten as:

$$\begin{cases} (S,V) \not\models & \exists \vec{X_i}, \left(\bigvee_{k=1}^{p_i} \exists \vec{E}^i_{1k}, v(\vec{Y}^i_{1k}) \wedge \psi^i_{1k} \right) \vee \\ & \left(\bigvee_{k=1}^{q_i} \exists \vec{E}^i_{2k}, \neg v(\vec{Y}^i_{2k}) \wedge \psi^i_{2k} \right) \vee \psi^i_3, i \in [1,n] \\ (S,V) \not\models & \exists \vec{X_i}, \left(\bigvee_{k=1}^{p_i} \exists \vec{E}^i_{1k}, v(\vec{Y}^i_{1k}) \wedge \psi^i_{1k} \right) \vee \\ & \left(\bigvee_{k=1}^{q_i} \exists \vec{E}^i_{2k}, \neg v(\vec{Y}^i_{2k}) \wedge \psi^i_{2k} \right) \vee \psi^i_3, \\ & \qquad\qquad\qquad\qquad i \in [n+1, n+m] \end{cases}$$

By applying the distribution of existential quantifier over disjunction

$$\exists \vec{X_i}, \xi_1(\vec{X_i}) \vee \xi_2(\vec{X_i}) \equiv (\exists \vec{X_i}, \xi_1(\vec{X_i})) \vee (\exists \vec{X_i}, \xi_2(\vec{X_i}))$$

we have:

$$\begin{cases} (S,V) \not\models & \left(\bigvee_{k=1}^{p_i} \exists \vec{X_i}, \exists \vec{E}^i_{1k}, v(\vec{Y}^i_{1k}) \wedge \psi^i_{1k} \right) \vee \\ & \left(\bigvee_{k=1}^{q_i} \exists \vec{X_i}, \exists \vec{E}^i_{2k}, \neg v(\vec{Y}^i_{2k}) \wedge \psi^i_{2k} \right) \vee \\ & \exists \vec{X_i}, \psi^i_3, i \in [1,n] \\ (S,V) \not\models & \left(\bigvee_{k=1}^{p_i} \exists \vec{X_i}, \exists \vec{E}^i_{1k}, v(\vec{Y}^i_{1k}) \wedge \psi^i_{1k} \right) \vee \\ & \left(\bigvee_{k=1}^{q_i} \exists \vec{X_i}, \exists \vec{E}^i_{2k}, \neg v(\vec{Y}^i_{2k}) \wedge \psi^i_{2k} \right) \vee \\ & \exists \vec{X_i}, \psi^i_3, i \in [n+1, n+m] \end{cases}$$

Here, we have a disjunction of many formulas on the right-hand side, and we can apply the equivalence between $(S,V) \not\models \xi_1 \vee \xi_2$ and $((S,V) \not\models \xi_1) \wedge ((S,V) \not\models \xi_2)$ to separate the disjunction on the right-hand side and obtain n_3 sentences as

follows:

$$\begin{cases} (S,V) \not\models & \exists \vec{E_k}, v(\vec{Y_k}) \wedge \psi_k, k \in [1, n_1] \\ (S,V) \not\models & \exists \vec{E_k}, \neg v(\vec{Y_k}) \wedge \psi_k, k \in [n_1+1, n_2] \\ (S,V) \not\models & \exists \vec{E_k}, \psi_k, k \in [n_2+1, n_3] \end{cases} \tag{A.4}$$

where $n_1 = \sum_{i=1}^{n+m} p_i$, $n_2 = n_1 + \sum_{i=1}^{n+m} q_i$ and $n_3 = n_2 + n + m$. All variables in $\vec{Y_k}$ are in $\vec{E_k}$ for any k.

Note that $\exists W, v(\vec{Y_k}) \wedge \psi_k \equiv v(\vec{Y_k}) \wedge \exists W, \psi_k$ if W is not a free variable in $v(\vec{Y_k})$. In this way, we push existential variables in $\vec{E_k}$ but not in $\vec{Y_k}$, denoted by $\vec{Z_k}$, into the subformula ψ_k. In the case that there is a variable X appearing more than once in $\vec{Y_k}$, we can introduce a new fresh variable X' and add the equality $X = X'$ to the formulas after the quantifier $\exists \vec{Y_k}$. For example,

$$\exists Y_1 Y_1 Y_2, v(Y_1, Y_1, Y_2) \equiv \exists Y_1 Y_1' Y_2, v(Y_1, Y_1', Y_2) \wedge Y_1 = Y_1'$$

We then substitute the variables in each $\vec{Y_k}$ to obtain the same $\vec{Y} = Y_1, \ldots, Y_{arity(v)}$ for each $\vec{Y_k}$. Then, we have n_3 FO sentences that (S,V) must not satisfy:

$$\begin{cases} (S,V) \not\models \exists \vec{Y}, v(\vec{Y}) \wedge \exists \vec{Z_k}, \psi_k(\vec{E_k}), k \in [1, n_1] \\ (S,V) \not\models \exists \vec{Y}, \neg v(\vec{Y}) \wedge \exists \vec{Z_k}, \psi_k(\vec{E_k}), k \in [n_1+1, n_2] \\ (S,V) \not\models \exists \vec{E_k}, \psi_k(\vec{E_k}), k \in [n_2+1, n_3] \end{cases}$$

Because $((S,V) \not\models \xi_1) \wedge ((S,V) \not\models \xi_2)$ is equivalent to $(S,V) \not\models \xi_1 \vee \xi_2$, we have:

$$\begin{cases} (S,V) \not\models \bigvee_{k=1}^{n_1} (\exists \vec{Y}, v(\vec{Y}) \wedge \exists \vec{Z_k}, \psi_k(\vec{E_k})) \\ (S,V) \not\models \bigvee_{k=n_1+1}^{n_2} (\exists \vec{Y}, \neg v(\vec{Y}) \wedge \exists \vec{Z_k}, \psi_k(\vec{E_k})) \\ (S,V) \not\models \bigvee_{k=n_2+1}^{n_3} (\exists \vec{E_k}, \psi_k(\vec{E_k})) \end{cases}$$

By applying the distribution of existential quantifier over disjunction $(\exists \vec{Y}, \xi_1(\vec{Y})) \vee (\exists \vec{Y}, \xi_2(\vec{Y})) \equiv \exists \vec{Y}, \xi_1(\vec{Y}) \vee \xi_2(\vec{Y})$, we have:

$$\begin{cases} (S,V) \not\models \exists \vec{Y}, \bigvee\limits_{k=1}^{n_1} (v(\vec{Y}) \wedge \exists \vec{Z_k}, \psi_k(\vec{E_k})) \\ (S,V) \not\models \exists \vec{Y}, \bigvee\limits_{k=n_1+1}^{n_2} (\neg v(\vec{Y}) \wedge \exists \vec{Z_k}, \psi_k(\vec{E_k})) \\ (S,V) \not\models \bigvee\limits_{k=n_2+1}^{n_3} (\exists \vec{E_k}, \psi_k(\vec{E_k})) \end{cases}$$

By applying the distribution of conjunction over disjunction $(p \wedge q) \vee (p \wedge r) \equiv p \wedge (q \vee r)$, we have:

$$\begin{cases} (S,V) \not\models \exists \vec{Y}, v(\vec{Y}) \wedge \phi_1(\vec{Y}) \\ (S,V) \not\models \exists \vec{Y}, \neg v(\vec{Y}) \wedge \phi_2(\vec{Y}) \\ (S,V) \not\models \phi_3 \end{cases} \tag{A.5}$$

$$\Leftrightarrow \begin{cases} (S,V) \models \forall \vec{Y}, v(\vec{Y}) \wedge \phi_1(\vec{Y}) \rightarrow \bot \\ (S,V) \models \forall \vec{Y}, \neg v(\vec{Y}) \wedge \phi_2(\vec{Y}) \rightarrow \bot \\ (S,V) \not\models \phi_3 \end{cases} \tag{A.6}$$

where $\phi_1 = \bigvee_{k=1}^{n_1} (\vec{Z_k}, \psi_k(\vec{E_k}))$, $\phi_2 = \bigvee\limits_{k=n_1+1}^{n_2} (\exists \vec{Z_k}, \psi_k(\vec{E_k}))$ and $\phi_3 = \bigvee\limits_{k=n_2+1}^{n_3} (\exists \vec{E_k}, \psi_k(\vec{E_k}))$.

Note that in (A.4), each $\exists \vec{E_k}, \psi_k$ $(k \in [n_2+1, n_3])$ is a safe-range GNFO formula; hence, ϕ_3 is a GNFO sentence. Each $\exists \vec{E_k}, \neg v(\vec{Y_k}) \wedge \psi_k$ $(k \in [n_1+1, n_2])$ is a safe-range GNFO formula, which means that each ψ_k $(k \in [n_1+1, n_2])$ is a safe-range GNFO formula; hence, ϕ_2 is a safe-range GNFO formula. Each $\exists \vec{E_k}, v(\vec{Y_k}) \wedge \psi_k, k \in [1, n_1]$ is a safe-range GNFO formula; hence, $v(\vec{Y}) \wedge \phi_1(\vec{Y}) \equiv \bigvee_{k=1}^{n_1} (\exists \vec{Y}, v(\vec{Y}) \wedge \exists \vec{Z_k}, \psi_k(\vec{E_k})) \equiv \bigvee_{k=1}^{n_1} (\exists \vec{E_k}, v(\vec{Y_k}) \wedge \psi_k)$, which is a safe-range GNFO formula.

A.1.5 Proof of Proposition 3.4.2

Proof. Consider a database S over schema $\langle r_1, \ldots, r_n \rangle$. $S \oplus \Delta S = S$ means that $\Delta^-_{R_i} \cap R_i = \emptyset$ and $\Delta^+_{R_i} \setminus R_i = \emptyset$ $(i \in [1, n])$. Let $\Delta^2 S$ be the change on ΔS, i.e., $\Delta^2 S$ contains insertions and deletions into/from each $\Delta^+_{R_i}$ and $\Delta^-_{R_i}$. We use $\Delta^\pm_{R_i}$ as an abbreviation for $\Delta^+_{R_i}$ and $\Delta^-_{R_i}$. Let $\Delta^+(\Delta^\pm_{R_i})$ and $\Delta^-(\Delta^\pm_{R_i})$ be the set of insertions and the set of deletions for $\Delta^\pm_{R_i}$, respectively. The new instance $\Delta'^\pm_{R_i}$ of

each $\Delta^{\pm}_{R_i}$ in ΔS is obtained by:

$$\Delta'^{\pm}_{R_i} = (\Delta^{\pm}_{R_i} \setminus \Delta^{-}(\Delta^{\pm}_{R_i})) \cup \Delta^{+}(\Delta^{\pm}_{R_i})$$

We finally obtain a new source database S' by applying each $\Delta'^{\pm}_{R_i}$ in $\Delta S'$ to the corresponding relation R_i in database S:

$$\begin{aligned} R'_i = \;& (R_i \setminus \Delta'^{-}_{R_i}) \cup \Delta'^{+}_{R_i} \\ = \;& (R_i \setminus ((\Delta^{-}_{R_i} \setminus \Delta^{-}(\Delta^{-}_{R_i})) \cup \Delta^{+}(\Delta^{-}_{R_i}))) \\ & \cup ((\Delta^{+}_{R_i} \setminus \Delta^{-}(\Delta^{+}_{R_i})) \cup \Delta^{+}(\Delta^{+}_{R_i})) \end{aligned}$$

Because $\Delta^{-}_{R_i}$ and $\Delta^{+}_{R_i}$ are disjoint, and because $\Delta^{-}_{R_i} \cap R_i = \emptyset$ and $\Delta^{+}_{R_i} \setminus R_i = \emptyset$, we can simplify the above equation to:

$$R'_i = R_i \setminus \Delta^{+}(\Delta^{-}_{R_i}) \cup \Delta^{+}(\Delta^{+}_{R_i}) \tag{A.7}$$

Note that $\Delta^{+}(\Delta^{-}_{R_i})$ and $\Delta^{+}(\Delta^{+}_{R_i})$ contain all the tuples inserted into $\Delta^{-}_{R_i}$ and $\Delta^{+}_{R_i}$, respectively. In other words, $\Delta^{+}(\Delta^{-}_{R_i})$ and $\Delta^{+}(\Delta^{+}_{R_i})$ are delta relations in $\Delta^{2+}S$. This means that the source database S' is obtained by applying $\Delta^{2+}S$ to S: $S' = S \oplus \Delta^{2+}S$. □

A.1.6 Proof of Lemma 3.4.4

Proof. Consider a valid LVGN-Datalog putback program *putdelta* for a view v and source database schema $\langle r_1, \ldots, r_n \rangle$. Since *putdelta* is in LVGN-Datalog, the view predicate occurs only in the rules defining delta relations of the source ($\pm r_1$, $\ldots$, $\pm r_n$), and at most once in each rule. When the view relation is changed, only delta relations, $\pm r_1, \ldots, \pm r_n$, are changed, all other relations (intermediate relations) in *putdelta* are unchanged. Therefore, to incrementalize *putdelta*, we use only rules defining delta relations (having a predicate $\pm r_i$ as the head) to derive the rules computing changes to the delta relations.

A Datalog rule having a delta predicate $\pm r_i$ in the head and a view predicate v in the body is in one of the following forms:

$$\pm r_i(\vec{X}) :- v(\vec{Y}), Q(\vec{Z}). \qquad \text{(positive view)}$$

$$\pm r_i(\vec{X}) :- \neg\ v(\vec{Y}), Q(\vec{Z}). \qquad \text{(negative view)}$$

where $Q(\vec{Z})$ is the conjunction of the rest of the rule body. $Q(\vec{Z})$ is unchanged, whereas the view relation v is changed to $v' = (v \setminus -v \cup +v)$, where $+v$ and $-v$ corresponds to the insertions set of deletions set, respectively. Similar to the incrementalization technique in [55], by distributing joins over set minus and union we obtain

$$+(\pm r_i)(\vec{X}) :- +v(\vec{Y}), Q(\vec{Z}).$$
$$-(\pm r_i)(\vec{X}) :- -v(\vec{Y}), Q(\vec{Z}).$$

for the case of positive view and

$$+(\pm r_i)(\vec{X}) :- -v(\vec{Y}), Q(\vec{Z}).$$
$$-(\pm r_i)(\vec{X}) :- +v(\vec{Y}), Q(\vec{Z}).$$

for the case of negative view, where new delta relations are obtained by $\pm r_i' = (\pm r_i \setminus -(\pm r_i)) \cup +(\pm r_i)$.

Proposition 3.4.2 implies that the set of insertions to the delta relation, $+(\pm r_i)$, can be used as $\pm r_i'$ to apply to the source relation r_i to obtain the same new source. Therefore, the rule computing $-(\pm r_i)$ is redundant, $\pm r_i'$ can be computed by the rules of $+(\pm r_i)$:

$$\pm r_i'(\vec{X}) :- +v(\vec{Y}), Q(\vec{Z}).$$

for the case of positive view and

$$\pm r_i'(\vec{X}) :- -v(\vec{Y}), Q(\vec{Z}).$$

for the case of negative view. This shows that the transformation from origin *putdelta* to an incremental one is substituting delta predicates of the view, $+v$ and $-v$, for positive and negative predicates of the view, v and $\neg v$, respectively. □

A.2 Transformation from safe-range FO formula to Datalog

In this section, we present the transformation from a safe-range FO formula φ to an equivalent Datalog query.

We first extend the algorithm that transforms a safe-range FO formula φ into an equivalent formula in relational algebra normal form (RANF) described in [43] to allow built-in predicates ($<$ and $>$) to occur in φ. Let us assume that φ is in safe-range normal form (SRNF) in which there are no universal quantifiers and no implications, and there is no conjunction or disjunction sign that occurs directly below a negation sign. Every FO formula can be transformed into an SRNF formula by inductively applying the following logical equivalences:

- $\forall \vec{x} \psi \equiv \neg \exists \vec{x} \neg \psi$
- $\varphi \rightarrow \psi \equiv \neg \varphi \vee \psi$
- $\neg \neg \psi \equiv \psi$
- $\neg(\psi_1 \vee \ldots \vee \psi_n) \equiv (\neg \psi_1 \wedge \ldots \wedge \neg \psi_n)$
- $\neg(\psi_1 \wedge \ldots \wedge \psi_n) \equiv (\neg \psi_1 \vee \ldots \vee \neg \psi_n)$

The set of range-restricted variables of the FO formula φ ($rr(\varphi)$) is inductively defined in the same way as [43]:

- if $\varphi = R(e_1, \ldots, e_n)$, $rr(\varphi) =$ the set of variables in $\{e_1, \ldots, e_n\}$
- if $\varphi = (x = a)$ or $\varphi = (a = x)$, $rr(\varphi) = x$
- if $\varphi = \varphi_1 \wedge \varphi_2$, $rr(\varphi) = rr(\varphi_1) \cup rr(\varphi_2)$
- if $\varphi = \varphi_1 \wedge x = y$, $rr(\varphi) = rr(\varphi_1)$ if $\{x, y\} \cap rr(\varphi_1) = \emptyset$ and $rr(\varphi) = rr(\varphi_1) \cup \{x, y\}$ otherwise
- if $\varphi = \varphi_1 \vee \varphi_2$, $rr(\phi) = rr(\phi_1) \cap rr(\phi_2)$
- if $\varphi = \neg \varphi_1$, $rr(\varphi) = \emptyset$

- if $\varphi \in \{(x > a), (x < a), (x > y), (x < y)\}$, $rr(\varphi) = \emptyset$
- if $\varphi = \exists \vec{x} \varphi_1$, $rr(\varphi) = rr(\varphi_1) - \vec{x}$ if $\vec{x} \subseteq rr(\varphi_1)$ and $rr(\varphi) = \perp$ otherwise

where for each Z, $\perp \cup Z = \perp \cap Z = \perp - Z = Z - \perp = \perp$, $\perp$ indicates that some quantified variables are not range restricted. Let $free(\varphi)$ be the set of free variables of φ. φ is a safe-range FO formula iff $rr(\varphi) = free(\varphi)$.

Definition A.2.1 ([43]) *An occurrence of a subformula ψ in φ is self-contained if its root is $\wedge$ or if*

- $\psi = \psi_1 \vee \ldots \vee \psi_n$ *and* $rr(\psi) = rr(\psi_1) = \ldots = rr(\psi_n) = free(\psi)$;
- $\psi = \exists \vec{x} \psi_1$ *and* $rr(\psi_1) = free(\psi_1)$;

A safe-range SRNF formula φ is in relational algebra normal form (RANF) if each subformula of φ is self-contained. □

The algorithm that transforms a safe-range SRNF formula φ into an equivalent RANF formula is based on the following rewrite rules for each subformula ψ in φ:

- Push-into-or: If $\psi = \psi_1 \wedge \ldots \wedge \psi_n \wedge \xi$, where $\xi = \xi_1 \vee \ldots \vee \xi_m$ and $rr(\psi) = free(\psi)$, but $rr(\xi) \neq free(\xi)$, we nondeterministically choose a subset $\{i_1, \ldots, i_k\}$ of $\{1, \ldots, n\}$ such that

$$\xi' = (\xi_1 \wedge \psi_{i_1} \wedge \ldots \wedge \psi_{i_k}) \vee \ldots \vee (\xi_m \wedge \psi_{i_1} \wedge \ldots \wedge \psi_{i_k})$$

  satisfies $rr(\xi') = free(\xi')$. Let $\{j_1, \ldots, j_l\} = \{1, \ldots, n\} \setminus \{i_1, \ldots, i_k\}$, we rewrite ψ into ψ':

$$\psi' = \psi_{j_1} \wedge \ldots \wedge \psi_{j_l} \wedge \xi'$$

- Push-into-quantifier: If $\psi = \psi_1 \wedge \ldots \wedge \psi_n \wedge \exists \vec{x} \xi$ and $rr(\psi) = free(\psi)$, but $rr(\xi) \neq free(\xi)$, assuming that no variable in $\vec{x}$ is a free in $free(\psi_1 \wedge \ldots \wedge \psi_n)$ (this can be achieved by variable renaming), we nondeterministically choose a subset $\{i_1, \ldots, i_k\}$ of $\{1, \ldots, n\}$ such that:

$$\xi' = \psi_{i_1} \wedge \ldots \wedge \psi_{i_k} \wedge \xi$$

satisfies $rr(\xi') = free(\xi')$. We replace ψ with

$$\psi' = \psi_{j_1} \wedge \ldots \wedge \psi_{j_l} \wedge \exists \vec{x} \xi'$$

where $\{j_1, \ldots, j_l\} = \{1, \ldots, n\} \setminus \{i_1, \ldots, i_k\}$.

- Push-into-negated-quantifier: If $\psi = \psi_1 \wedge \ldots \wedge \psi_n \wedge \neg \exists \vec{x} \xi$ and $rr(\psi) = free(\psi)$, but $rr(\xi) \neq free(\xi)$, assuming that no variable in $\vec{x}$ is a free in $free(\psi_1 \wedge \ldots \wedge \psi_n)$ (this can be achieved by variable renaming), we nondeterministically choose a subset $\{i_1, \ldots, i_k\}$ of $\{1, \ldots, n\}$ such that:

  $$\xi' = \psi_{i_1} \wedge \ldots \wedge \psi_{i_k} \wedge \xi$$

  satisfies $rr(\xi') = free(\xi')$. We replace ψ with

  $$\psi' = \psi_1 \wedge \ldots \wedge \psi_n \wedge \neg \exists \vec{x} \xi'$$

  ψ' is equivalent to ψ because the propositional formulas $p \wedge q \wedge \neg r$ and $p \wedge q \wedge \neg(p \wedge r)$ are equivalent. And we continue to apply the Push-into-quantifier procedure

Now we transform the RANF formula φ into an equivalent Datalog query (P_φ, G_φ). Suppose $\{x_1, \ldots, x_k\} = free(\varphi)$, (P_φ, G_φ) is inductively constructed as follows:

- If $\varphi = R(e_1, \ldots, e_n)$, where $\{x_1, \ldots, x_k\}$ is the set of free variables in $\{e_1, \ldots, e_n\}$:

  $$P_\varphi = \{G_\varphi(x_1, \ldots, x_k) :- R(e_1, \ldots, e_n).\}$$

  and the datalog query is (P_φ, G_φ).

- If φ is $x = a$ or $a = x$:

  $$P_\varphi = \{G_\varphi(x) :- x = a.\}$$

- If $\varphi = \psi_1 \wedge \ldots \wedge \psi_m$, we divide $\{\psi_1, \ldots, \psi_m\}$ into a set of positive subformulas $\{\psi_1, \ldots, \psi_{m_1}\}$ and a set of equalities/inequalities ($x = a$, $a = x$, $x = x$,

$x = y$, $x > a$, $x < a$, $x > y$, $x < y$) $\{\psi_{m_1+1}, \ldots, \psi_{m_2}\}$, and a set of negative subformulas $\{\neg\psi_{m_2+1}, \ldots, \neg\psi_m\}$. Let $\{x_{i1}, \ldots, x_{ik_i}\} = free(\psi_i)$, we inductively construct (P_{ψ_i}, G_{ψ_i}) for each ψ_i in $\{\psi_1, \ldots, \psi_{m_1}\}$, and for each ψ_i in $\{\psi_{m_2+1}, \ldots, \psi_m\}$. The Datalog query (P_φ, G_φ) is as follows:

$$P_\varphi = \left(\bigcup_{i=1}^{m_1} P_{\psi_i}\right) \cup \left(\bigcup_{i=m_2+1}^{m} P_{\psi_i}\right) \cup$$
$$\left\{\begin{array}{ll} G_\varphi(x_1, \ldots, x_k) :- & G_{\psi_1}(x_{11}, \ldots, x_{1k_1}), \ldots, \\ & G_{\psi_{m_1}}(x_{m_1 1}, \ldots, x_{m_1 k_{m_1}}), \\ & \psi_{m_1+1}, \ldots, \psi_{m_2}, \\ & \neg G_{\psi_{m_2+1}}(x_{(m_2+1)1}, \ldots, \\ & x_{(m_2+1)k_{(m_2+1)}}), \ldots, \\ & \neg G_{\psi_m}(x_{m1}, \ldots, x_{mk_m}). \end{array}\right\}$$

- If $\varphi = \psi_1 \vee \ldots \vee \psi_n$, where $free(\psi_1) = \ldots = free(\psi_n) = \{x_1, \ldots, x_k\}$. We construct (P_{ψ_i}, G) (with the same goal predicate G) for each ψ_i in $\{\psi_1, \ldots, \psi_n\}$ and obtain:
$$P_\varphi = \bigcup_{i=1}^{n} P_{\psi_i}$$
$$G_\varphi = G$$

- If $\varphi = \exists y_1, \ldots, y_m, \psi(z_1, \ldots, z_n)$, where $\{x_1, \ldots, x_k\} = \{z_1, \ldots, z_n\} \backslash \{y_1, \ldots, y_m\}$:
$$P_\varphi = P_\psi \cup \{G_\varphi(x_1, \ldots, x_k) :- G_\psi(z_1, \ldots, z_n).\}$$

To conclude that the transformation from safe-range FO formula to Datalog query is correct, i.e. φ and (P_φ, G_φ) are equivalent, we need to show that for any database instance D, $P_\varphi(D)|G_\varphi = \{\vec{t} \mid D \models \varphi(\vec{t})\}$, where $P_\varphi(D)|G_\varphi$ denotes the restriction of the output of P over D to the relation G_φ. Indeed, let D be fixed, by induction, we can show that for each subformula ψ of φ and each tuple $\vec{t}$,

$$D \models \psi(\vec{t}) \Leftrightarrow P_\psi(D) \ni G_\psi(\vec{t})$$

Join and Selection
$h(\vec{X}) :- r_1(\vec{Y}), r_2(\vec{Z}).$ $vars(\vec{X}) = vars(\vec{Y}) \cup vars(\vec{Z})$
$\Downarrow$
$-h(\vec{X}) :- -r_1(\vec{Y}), r_2(\vec{Z}).$ $-h(\vec{X}) :- r_1(\vec{Y}), -r_2(\vec{Z}).$ $+h(\vec{X}) :- +r_1(\vec{Y}), r_2^\nu(\vec{Z}).$ $+h(\vec{X}) :- r_1^\nu(\vec{Y}), +r_2(\vec{Z}).$ $h^\nu(\vec{X}) :- r_1^\nu(\vec{Y}), r_2^\nu(\vec{Z}).$

Negation
$h(\vec{X}) :- r_1(\vec{X}), \neg r_2(\vec{Y}).$ $vars(\vec{X}) \supseteq vars(\vec{Y})$
$\Downarrow$
$-h(\vec{X}) :- -r_1(\vec{X}), \neg r_2(\vec{Y}).$ $-h(\vec{X}) :- r_1(\vec{X}), +r_2(\vec{Y}).$ $+h(\vec{X}) :- +r_1(\vec{X}), \neg r_2^\nu(\vec{Y}).$ $+h(\vec{X}) :- r_1^\nu(\vec{X}), -r_2(\vec{Y}).$ $h^\nu(\vec{X}) :- r_1^\nu(\vec{X}), \neg r_2^\nu(\vec{Y}).$

Projection
$h(\vec{X}) :- r_1(\vec{X}, \vec{Y}).$
$\Downarrow$
$+h(\vec{X}) :- +r_1(\vec{X}, \vec{Y}), \neg h(\vec{X}).$ $-h(\vec{X}) :- -r_1(\vec{X}, \vec{Y}), \neg r_1^\nu(\vec{X}, \_).$ $h^\nu(\vec{X}) :- r_1^\nu(\vec{X}, \vec{Y}).$

Union
$h(\vec{X}) :- r_1(\vec{X}).$ $h(\vec{X}) :- r_2(\vec{X}).$
$\Downarrow$
$-h(\vec{X}) :- -r_1(\vec{X}), \neg r_2^\nu(\vec{X}).$ $-h(\vec{X}) :- -r_2(\vec{X}), \neg r_1^\nu(\vec{X}).$ $+h(\vec{X}) :- +r_1(\vec{X}).$ $+h(\vec{X}) :- +r_2(\vec{X}).$ $h^\nu(\vec{X}) :- r_1^\nu(\vec{X}).$ $h^\nu(\vec{X}) :- r_2^\nu(\vec{X}).$

Figure A.1: Rules for incrementalizing Datalog putback programs. $\vec{X}$ denotes a tuple of variables, $vars(\vec{X})$ denotes the set of all variables in $\vec{X}$.

A.3 Rules for incrementalizing putback programs

Given a putback program *putdelta* in nonrecursive Datalog with negation (NR-Datalog$^\neg$), we shall derive Datalog rules to compute changes to delta relations of the source database when the view relation is changed. The derived Datalog rules form an incrementalized program of *putdelta*.

Our idea is that we first transform *putdelta* into an equivalent Datalog program, in which every IDB relation is defined from at most 2 other relations. We then inductively apply the incrementalization rules in Figure A.1 to derive Datalog rules for computing changes to each IDB relation.

Lemma A.3.1 *For every NR-Datalog$^\neg$ program P with a goal IDB relation R, there is a NR-Datalog$^\neg$ program P' in which each IDB relation is defined from at most two other relations such that the queries (P, R) and (P', R) are equivalent.* □

Proof. (Sketch) There exists such a transformation between these two Datalog programs because a NR-Datalog$^{\neg}$ query (P, R) is equivalent to a relational algebra expression, in which each binary relational operator can be simulated by Datalog rules with two relations in the rule bodies. □

Considering the set semantics of the Datalog language, we propose rewrite rules (shown in Figure A.1) for calculating changes to a relation h which is defined from two relations r_1 and r_2. In each case of the definition of h, we derive Datalog rules that compute separately the set of insertions (Δ_h^+) and the set of deletions (Δ_h^-) to h when there are changes to relations r_1 and r_2. Note that in these derived Datalog rules, if $\Delta_{r_1}^+$ and $\Delta_{r_1}^-$ are disjoint, then the obtained Δ_h^+ and Δ_h^- are also disjoint. Therefore, we can inductively apply the four incrementalization rules when h is used to define other IDB relations. We have formally proven the correctness of these incrementalization rules by using an assistant theorem prover, stated as the following.

Lemma A.3.2 *For each case in Figure A.1, the new relation h^ν computed from its defining rules is the same as the result obtained by applying delta relations $+h$ and $-h$ computed by the derived Datalog rules to the original relation h.* □

Our incrementalization rules can be easily extended for built-in predicates (e.g., $=, <, >$) in the Datalog program by considering these predicates as unchanged relations in our incrementalization rules.

A.4 Deriving view deltas

Algorithm 3: VIEW-DELTA$(u_1, \ldots, u_n)$

```
1 Δ_V^+ ← ∅; Δ_V^- ← ∅;
2 for each DML statement u in u_1, ..., u_n do
3 |   Derive the set δ^+/δ^- of inserted/deleted tuples;
4 |   Δ_V^+ ← (Δ_V^+ \ δ^-) ∪ δ^+;
5 |   Δ_V^- ← (Δ_V^- \ δ^+) ∪ δ^-;
```

Our incrementalization on putback transformation requires deriving a delta relation ΔV of the view V in the form of insertions and deletions when there are

any view update requests. In RDBMSs, these update requests are declarative DML (data manipulation language) statements of the following forms [62]: `INSERT INTO` V `VALUES(...)`, `DELETE FROM V WHERE <condition>`, and `UPDATE V SET attr=expr, ... WHERE <condition>`. Fortunately, it is trivial to obtain from the `INSERT`/`DELETE` statement the tuples that need to be inserted or deleted. Meanwhile, an `UPDATE` statement on the view can be represented as deletions followed by insertions; hence, we can also derive the deleted/inserted tuples.

A view update request can be a sequence of DML statements rather than a single one. This sequence is combined into one transaction by using the SQL command `BEGIN` before the sequence and the command `END` after the sequence. To address this case, we propose a procedure for calculating Δ_V^+ and Δ_V^- of the whole view update transaction, as shown in Algorithm 3. Concretely, for each DML statement in the sequence, we derive the insertion set δ^+ and the deletion set δ^-, and we merge these changes to Δ_V^+ and Δ_V^-. In this way, later statements have stronger effects than earlier statements. For example, if the sequence is inserting a tuple $\vec{t}$ and then deleting this tuple, $\vec{t}$ is no longer inserted, i.e., we remove $\vec{t}$ from Δ_V^+.