# Programmable View Update Strategies on Relations

Van-Dang Tran[3,1], Hiroyuki Kato[1,3], Zhenjiang Hu[2,1]

[1]National Institute of Informatics, Japan
[2]Peking University, China
[3]The Graduate University for Advanced Studies, SOKENDAI, Japan

{dangtv, kato}@nii.ac.jp, huzj@pku.edu.cn

## ABSTRACT

View update is an important mechanism that allows updates on a view by translating them into the corresponding updates on the base relations. The existing literature has shown the ambiguity of translating view updates. To address this ambiguity, 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 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.

## 1. INTRODUCTION

View update [11, 20, 21, 22, 33] 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 [21]. Consider a view $V$ defined by a query $get$ over the database $S$, as shown in Figure 1a. 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* [21] is to derive such an update translator $T$.
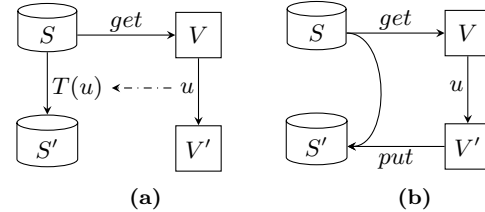
**Figure 1: The view update problem (a) and bidirectional transformation (b).**

However, there is an ambiguity issue here. 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 [20, 21]. This ambiguity makes view update an open challenging problem that has a long history in database research [22, 20, 21, 11, 34, 33, 40, 36, 45, 42, 41]. The existing approaches either impose too many syntactic restrictions on the view definition *get* that allow for limited unambiguous update propagation [21, 15, 11, 35, 43, 41, 44, 45, 46] or provide dialogue mechanisms for users to manually choose update translations with users' interaction [34, 42]. In practice, commercial database systems such as PostgreSQL [4] provide very limited support for updatable views such that even a simple union view cannot be updated.

In this paper, 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 [34], 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 [25, 19] 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 1b. 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 [32, 24, 23, 38, 37]. 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 [57, 38, 37], 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 [25, 23, 24]. 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 [57, 38, 37] 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 program. 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 paper is organized as follows. After presenting some basic notions in Section 2, we present our proposed method for specifying view update strategies in Datalog in Section 3. The validation and incrementalization algorithms for these update strategies are described in Section 4 and Section 5, respectively. Section 6 shows the experimental results of our implementation. Section 7 summarizes related works. Section 8 concludes this paper.

---

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

## 2. PRELIMINARIES

In this section, we briefly review the basic concepts and notations that will be used throughout this paper.

### 2.1 Datalog and Relational Databases

**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 [18, 17], 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 $D = \{r_1(1, 2), r_1(2, 3), r_2(3), r_2(4)\}$.

**Datalog.** A Datalog program $P$ is a nonempty finite set of rules, and each rule is an expression of the form [18]:

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

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. 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 [18], throughout this paper, 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 "_".

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 [10].

We can extend Datalog by allowing negation and built-in predicates, such as equality ($=$) or 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 [18].

### 2.2 Bidirectional Transformations

A bidirectional transformation (BX) [25] is a pair of a forward transformation *get* and a backward (putback) transformation *put*, as shown in Figure 1b. The forward transformation *get* is a query over a source database $S$ that results
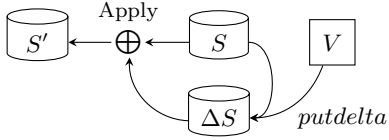
**Figure 2: View update strategy _put_.**

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 $S'$. To ensure consistency between the source database and the view, a BX must satisfy the following _round-tripping_ properties, called GETPUT and PUTGET:

$$\forall S, \qquad put\,(S,\ get(S)) = S \qquad \text{(GETPUT)}$$

$$\forall S,\ V', \qquad get\,\big(put\,(S,\ V')\big) = V' \qquad \text{(PUTGET)}$$

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 2.1    (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 [23].

THEOREM 2.1    (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._

## 3. 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., $=, \neq, <, >$) [18]. 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.

### 3.1 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 [28]. 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 [27, 39, 53], 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 $\Delta R$, 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 $\Delta_R^+$ be the set of insertions and $\Delta_R^-$ be the set of deletions in $\Delta R$. Applying $\Delta R$ to the relation $R$ is to delete tuples in $\Delta_R^-$ from $R$ and insert tuples in $\Delta_R^+$ 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 3.1. _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:_

$$-r_1(X) :\!- r_1(X), \neg v(X).$$
$$-r_2(X) :\!- r_2(X), \neg v(X).$$
$$+r_1(X) :\!- v(X), \neg r_1(X), \neg r_2(X).$$

_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 $S' = \{r_1(1), r_1(3), r_2(4)\}$._ $\qquad\square$

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 $\Delta S$ to denote all insertions and deletions of all relations in $S$. For example, the $\Delta S$ in

Example 3.1 is $\Delta S = \{+r_1(3), -r_2(2)\}$. We say that $\Delta S$ is *non-contradictory* if it has no insertion/deletion of the same tuple into/from the same relation. Applying a non-contradictory $\Delta S$ to a database $S$, denoted as $S \oplus \Delta S$, is to apply each delta relation in $\Delta S$ 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 $\Delta S$ (shown in Figure 2) as follows:

$$put(S, V) = S \oplus putdelta(S, V) \qquad (1)$$

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

**DEFINITION 3.1** (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 $\Delta S$.*

## 3.2 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 [13]) with equalities, constants, comparisons [18] 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.

### 3.2.1 Nonrecursive GN-Datalog with Equalities, Constants, and Comparisons

We consider a restricted form of negation in Datalog, called GN-Datalog [12, 13], 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 [13] 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.** *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 $r_1(X, Y, Z)$.* □

### 3.2.2 Linear View

As formally proven in [24], 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** (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 3.3.** *Given a source relation $R$ of arity 3 and a view relation $V$ of arity 2, consider the following rules of the delta relation $\Delta R$:*

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

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

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

$(rule_1)$ *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 $(rule_3)$.* □

### 3.2.3 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 [17], 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 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 3.4.** *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: $\bot :- r(X, Y, Z), Z > 2$.* □

### 3.2.4 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 [13]. 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 [13]. 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 [12].

```
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**

$$
\begin{aligned}
&\texttt{ced}(E,D) & &:\!- \ \texttt{ed}(E,D), \neg\ \texttt{eed}(E,D). \\
&\texttt{residents}(E,B,G) & &:\!- \ \texttt{others}(E,B,G). \\
&\texttt{residents}(E,B,\text{`F'}) & &:\!- \ \texttt{female}(E,B). \\
&\texttt{residents}(E,B,\text{`M'}) & &:\!- \ \texttt{male}(E,B). \\
&\texttt{residents1962}(E,B,G) & &:\!- \ \texttt{residents}(E,B,G), \\
& & & \quad \neg B <\text{`1962-01-01'}, \neg B >\text{`1962-12-31'}. \\
&\texttt{employees}(E,B,G) & &:\!- \ \texttt{residents}(E,B,G),\ \texttt{ced}(E,D). \\
&\texttt{retired}(E) & &:\!- \ \texttt{residents}(E,B,G), \neg\texttt{ced}(E,\_).
\end{aligned}
$$

**Views**

**Figure 3: Database and view schema.**

LEMMA 3.1. *The query satisfiability problem is decidable for nonrecursive GN-Datalog with equalities, constants and comparisons.*

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

THEOREM 3.2. *The query satisfiability problem for nonrecursive GN-Datalog with equalities, constants and comparisons under a set of guarded negation constraints is decidable.*

## 3.3 A Case Study

We consider a database of five base tables shown in Figure 3. 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.

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

$$
\begin{aligned}
&\texttt{+male}(E,B) & &:\!- \ \texttt{residents}(E,B,\text{`M'}), \\
& & & \quad \neg\ \texttt{male}(E,B), \neg\ \texttt{others}(E,B,\text{`M'}). \\
&\texttt{-male}(E,B) & &:\!- \ \texttt{male}(E,B), \neg\ \texttt{residents}(E,B,\text{`M'}). \\
&\texttt{+female}(E,B) & &:\!- \ \texttt{residents}(E,B,G),\ G =\text{`F'}, \\
& & & \quad \neg\ \texttt{female}(E,B), \neg\ \texttt{others}(E,B,G). \\
&\texttt{-female}(E,B) & &:\!- \ \texttt{female}(E,B), \neg\ \texttt{residents}(E,B,\text{`F'}). \\
&\texttt{+others}(E,B,G) & &:\!- \ \texttt{residents}(E,B,G), \neg\ G =\text{`M'}, \\
& & & \quad \neg\ G =\text{`F'}, \neg\ \texttt{others}(E,B,G). \\
&\texttt{-others}(E,B,G) & &:\!- \ \texttt{others}(E,B,G), \\
& & & \quad \neg\ \texttt{residents}(E,B,G).
\end{aligned}
$$

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`.

$$
\begin{aligned}
&\texttt{+ed}(E,D) & &:\!- \ \texttt{ced}(E,D), \neg\ \texttt{ed}(E,D). \\
&\texttt{-eed}(E,D) & &:\!- \ \texttt{ced}(E,D),\ \texttt{eed}(E,D). \\
&\texttt{+eed}(E,D) & &:\!- \ \texttt{ed}(E,D), \neg\ \texttt{ced}(E,D), \neg\ \texttt{eed}(E,D).
\end{aligned}
$$

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:
```
$$
\begin{aligned}
&\bot :\!- \ \texttt{residents1962}(E,B,G), B >\text{`1962-12-31'}. \\
&\bot :\!- \ \texttt{residents1962}(E,B,G), B <\text{`1962-01-01'}.
\end{aligned}
$$
```
% Update rules:
```
$$
\begin{aligned}
&\texttt{+residents}(E,B,G) :\!- \ \texttt{residents1962}(E,B,G), \\
& \qquad\qquad\qquad\qquad \neg\ \texttt{residents}(E,B,G). \\
&\texttt{-residents}(E,B,G) :\!- \ \texttt{residents}(E,B,G), \\
& \qquad\qquad\qquad \neg\ B <\text{`1962-01-01'}, \\
& \qquad\qquad\qquad \neg\ B >\text{`1962-12-31'}, \\
& \qquad\qquad\qquad \neg\ \texttt{residents1962}(E,B,G).
\end{aligned}
$$

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:
```
$$
\bot :\!- \ \texttt{employees}(E,B,G), \neg\ \texttt{ced}(E,\_).
$$
```
% Update rules:
```
$$
\begin{aligned}
&\texttt{+residents}(E,B,G) :\!- \ \texttt{employees}(E,B,G), \\
& \qquad\qquad\qquad\qquad \neg\ \texttt{residents}(E,B,G). \\
&\texttt{-residents}(E,B,G) :\!- \ \texttt{residents}(E,B,G), \\
& \qquad\qquad\qquad \texttt{ced}(E,\_), \neg\ \texttt{employees}(E,B,G).
\end{aligned}
$$

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
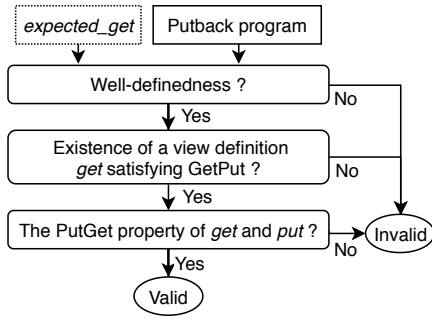
**Figure 4: Validation algorithm.**

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, 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.

# 4. VALIDATION ALGORITHM

As mentioned in Section 2, a view update strategy must be valid (Definition 2.1) 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.

## 4.1 Overview

Checking the validity of a view update strategy based on Definition 2.1 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 4.1. *Given a valid view update strategy put, if a view definition get satisfies* GETPUT*, then get must also satisfy* PUTGET *with put.*

Lemma 4.1 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 4.1, 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 4.1). 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 4): (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*.

## 4.2 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 $\Delta S$ resulting from *putdelta* is non-contradictory for any source database $S$ and any view relation $V$. In other words, we check whether in $\Delta S$, 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*:

$$d_i(\vec{X}_i) :- +r_i(\vec{X}_i), -r_i(\vec{X}_i). \quad (i \in [1, n]) \quad (2)$$

The problem of checking whether $\Delta S$ 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 (2) is trivially negation guarded, according to Theorem 3.2, the satisfiability of $d_i$ is decidable.

## 4.3 Existence of A View Definition Satisfying GetPut

Consider a view update strategy *put* specified by a putback program *putdelta* and a set of constraints $\Sigma$. 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.

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 $\Sigma$ 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 $\Sigma$.

EXAMPLE 4.1 (INTUITION). *Consider the update strategy put in Example 3.1. 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:*

$$\begin{array}{l} \Delta_{R_1}^- \cap R_1 = \emptyset \\ \Delta_{R_2}^- \cap R_2 = \emptyset \\ \Delta_{R_1}^+ \setminus R_1 = \emptyset \end{array} \quad (3)$$

*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 [10]: $\varphi_{-r_1} = r_1(X) \wedge \neg v(X)$, $\varphi_{-r_2} = r_2(X) \wedge \neg v(X)$, $\varphi_{+r_1} = v(X) \wedge \neg r_1(X) \wedge \neg r_2(X)$. The constraint (3) 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:*

$$\left\{ \begin{array}{l} (S, V) \not\models \exists X, \varphi_{-r_1}(X) \wedge r_1(X) \\ (S, V) \not\models \exists X, \varphi_{-r_2}(X) \wedge r_2(X) \\ (S, V) \not\models \exists X, \varphi_{+r_1}(X) \wedge \neg r_1(X) \end{array} \right.$$

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

$$\left\{ \begin{array}{l} (S, V) \models \forall X, \varphi_{-r_1}(X) \wedge r_1(X) \to \bot \\ (S, V) \models \forall X, \varphi_{-r_2}(X) \wedge r_2(X) \to \bot \\ (S, V) \models \forall X, \varphi_{+r_1}(X) \wedge \neg r_1(X) \to \bot \end{array} \right.$$

$$\Leftrightarrow (S, V) \models \left\{ \begin{array}{l} \forall X, r_1(X) \wedge \neg v(X) \wedge r_1(X) \to \bot \\ \forall X, r_2(X) \wedge \neg v(X) \wedge r_2(X) \to \bot \\ \forall X, v(X) \wedge \neg r_1(X) \wedge \neg r_2(X) \wedge \neg r_1(X) \to \bot \end{array} \right.$$

*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 $p \wedge \neg q \to \bot \equiv p \to q$ and obtain:*

$$\Leftrightarrow (S, V) \models \left\{ \begin{array}{l} \forall X, r_1(X) \to v(X) \\ \forall X, r_2(X) \to v(X) \\ \forall X, v(X) \to \neg(\neg r_1(X) \wedge \neg r_2(X)) \end{array} \right.$$

*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:*

$$(S, V) \models \left\{ \begin{array}{l} \forall X, r_1(X) \vee r_2(X) \to v(X) \\ \forall X, v(X) \to \neg(\neg r_1(X) \wedge \neg r_2(X)) \end{array} \right. \quad (4)$$

*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*

---

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

*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 (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 $\psi_2$, we obtain the same formula as $\psi_1$; hence, $\forall X, \psi_1(X) \to \psi_2(X)$ 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 $\psi_1$ over the source $S$. Since $\psi_1$ is a safe-range formula[3], we transform $\psi_1$ to an equivalent Datalog query[4] as follows:*

$$v(X) :- r_1(X). \quad (5)$$
$$v(X) :- r_2(X). \quad (6)$$

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

### 4.3.1 Checking the existence of a steady-state view

In general, similar to the idea shown in Example 4.1, for an arbitrary putback program *putdelta* and a set of constraints $\Sigma$ in LVGN-Datalog, we can always construct a guarded negation first-order (GNFO) sentence to check whether a steady-state view $V$ satisfying $\Sigma$ and $put(S, V) = S$ (i.e., $S \oplus putdelta(S, V) = S$) exists.

LEMMA 4.2. *Given a LVGN-Datalog putback program putdelta and a set of guarded negation constraints $\Sigma$, there exist first-order formulas $\phi_1, \phi_2, \phi_3$ such that for a given database instance $S$, a view relation $V$ satisfies $\Sigma$ and $S \oplus putdelta(S, V) = S$ iff*

$$\left\{ \begin{array}{l} (S, V) \models \forall \vec{Y}, v(\vec{Y}) \wedge \phi_1(\vec{Y}) \to \bot \\ (S, V) \models \forall \vec{Y}, \neg v(\vec{Y}) \wedge \phi_2(\vec{Y}) \to \bot \\ (S, V) \not\models \phi_3 \end{array} \right. \quad (7)$$

*where $v$ is the predicate corresponding to the view relation $V$ and $\phi_1, \phi_2, \phi_3$ have no occurrence of the view predicate $v$. Both $\phi_2(\vec{Y})$ and $\phi_3$ 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 (7) is simplified to $S \not\models \phi_3$ because the FO sentence $\phi_3$ has no atom of $v$ as a subformula. This means that $\phi_3$ must be unsatisfiable over any database $S$. Since $\phi_3$ is a GNFO sentence, we can check whether $\phi_3$ 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 (7), by applying the logical equivalence $p \wedge \neg q \to \bot \equiv p \to q$, we have:

$$\left\{ \begin{array}{l} (S, V) \models \forall \vec{Y}, v(\vec{Y}) \to \neg \phi_1(\vec{Y}) \\ (S, V) \models \forall \vec{Y}, \phi_2(\vec{Y}) \to v(\vec{Y}) \end{array} \right. \quad (8)$$

Because $\phi_1$ and $\phi_2$ do not contain an atom of $v$ as a subformula, there exists an instance $V$ if

$$S \models \forall \vec{Y}, \phi_2(\vec{Y}) \to \neg \phi_1(\vec{Y})$$
$$\Leftrightarrow S \models \forall \vec{Y}, \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y}) \to \bot$$

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

---

[3]$\psi$ is a safe-range FO formula if all the variables in $\psi$ are range restricted [10].
[4]Due to the equivalence between nonrecursive Datalog queries and safe-range FO formulas [10].

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 [12].

### 4.3.2  Constructing a view definition

If both $\phi_3$ and $\exists \vec{Y}, \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y})$ are unsatisfiable, there exists a steady-state view $V$ satisfying $\Sigma$ such that $S \oplus putdelta(S, V) = S$ for each database $S$. One steady-state view $V$ is the one resulting from the FO formula $\phi_2$ over $S$. Indeed, such a $V$ satisfies (8); hence, it satisfies $\Sigma$ and $S \oplus putdelta(S, V) = S$. By choosing this steady-state view, we can construct a view definition $get$ as the Datalog query equivalent to $\phi_2$ because $\phi_2$ is a safe-range formula. The equivalence of safe-range first-order logic and Datalog was well studied in database theory [10, 13]. We present the detailed transformation from safe-range FO formula to Datalog query in the full paper [55]. Due to Lemma 4.2, $\phi_2$ is also negation guarded and hence, $get$ is in nonrecursive GN-Datalog with equalities, constants and comparisons.

## 4.4  The PutGet Property

To check the PutGet property that $get(put(S, V)) = V$ for any $S$ and $V$, we first construct a Datalog query over database $(S, V)$ equivalent to the composition $get(put(S, V))$. Recall that $put(S, V) = S \oplus putdelta(S, V)$. The result of $put(S, V)$ is a new source $S'$ obtained by applying $\Delta S$ 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 $\Delta S$ to the database $S$ is equivalent to the result of the following Datalog rules ($i \in [1, n]$):

```
r_i^new(X_i)  :-  r_i(X_i),  ¬ -r_i(X_i).
r_i^new(X_i)  :-  +r_i(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 4.1 is:

```
-r_1(X)   :-  r_1(X),  ¬ v(X).
-r_2(X)   :-  r_2(X),  ¬ v(X).
+r_1(X)   :-  v(X),  ¬ r_1(X),  ¬ r_2(X).
r_1^new(X)  :-  r_1(X),  ¬ -r_1(X).
r_1^new(X)  :-  +r_1(X).
r_2^new(X)  :-  r_2(X),  ¬ -r_2(X).
v^new(X)   :-  r_1^new(X).
v^new(X)   :-  r_2^new(X).
```

---

**Algorithm 1:** VALIDATE($expected\_get$, $putdelta$, $\Sigma$)

> $get \leftarrow$ null;
> // Checking the well-definedness of $putdelta$
> **check** if all predicates $d_i$ ($i \in [1, n]$) in (2) are unsatisfiable under $\Sigma$;
> **if** $expected\_get$ **is not** null **then**
> > // Checking if $expected\_get$ satisfies GETPUT
> > **if** all delta relations of $putdelta$ are unsatisfiable under $\Sigma$ with the view defined by $expected\_get$ **then**
> > > $get \leftarrow expected\_get$;
>
> **if** ($expected\_get$ **is** null) **or** ($get$ **is** null) **then**
> > // Constructing a $get$ satisfying GETPUT
> > **check** if $\phi_3$ in (7) is unsatisfiable under $\Sigma$;
> > **check** if $\exists \vec{Y}, \phi_1(\vec{Y}) \wedge \phi_2(\vec{Y})$ ($\phi_1$ and $\phi_2$ in (8)) is unsatisfiable under $\Sigma$;
> > // Constructing a $get$
> > $get \leftarrow$ Translating FO formula $\phi_2$ in (8) to an equivalent Datalog query;
>
> // Checking the PUTGET property
> **check** if $\Phi_1$ and $\Phi_2$ in (9) and (10) are unsatisfiable under $\Sigma$;
> **return** $get$;

---

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:

$$\Phi_1 = \exists \vec{Y}, \phi_{putget}(\vec{Y}) \wedge \neg v(\vec{Y}) \tag{9}$$

$$\Phi_2 = \exists \vec{Y}, v(\vec{Y}) \wedge \neg \phi_{putget}(\vec{Y}) \tag{10}$$

The PUTGET property holds if and only if $\Phi_1$ and $\Phi_2$ 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, $\Phi_2$ is a GNFO sentence; hence, its satisfiability is decidable. $\Phi_1$ 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 $\Phi_1'$ is a guarded negation first-order sentence, its satisfiability is decidable by Theorem 3.2.

## 4.5  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), Algorithm 1 is sound and complete.

THEOREM 4.3    (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 [10, 54]. Therefore, Algorithm 1 is sound for validating the pair of *putdelta* and *expected_get* that once it terminates, we can conclude *putdelta* is valid.

# 5. 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 $\Delta S$ having no effect on the original source $S$: $S \oplus \Delta S = S$. This means that $\Delta S$ can be either an empty set or a nonempty set in which all deletions in $\Delta S$ are not yet in the original source $S$ and all insertions in $\Delta S$ are already in $S$. If the view is updated by a delta $\Delta V$, there will be some changes to $\Delta S$, denoted as $\Delta^2 S$, that have effects on the original source $S$.

EXAMPLE 5.1. *Consider the database in Example 3.1:* $S = \{r_1(1), r_2(2), r_2(4)\}$. *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* $\Delta S$ *by a delta of* $\Delta S$, *denoted as* $\Delta^2 S$, *which includes a set of deletions to* $\Delta S$, $\Delta^{2-}S = \{+r_1(1), -r_2(3)\}$, *and a set of insertions to* $\Delta S$, $\Delta^{2+}S = \{+r_1(3), -r_2(4)\}$. *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* $S$: $S' = S \oplus \Delta^{2+}S$. $\square$

Intuitively, for each base relation $R_i$ in the source database $S$, we obtain the new $R_i'$ by applying to $R_i$ the delta relations $\Delta_{R_i}^-$ and $\Delta_{R_i}^+$ from $\Delta S$. 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 5.1. *Let* $S$ *be a database and* $\Delta S$ *be a non-contradictory delta of the database* $S$ *such that* $S \oplus \Delta S = S$. *Let* $\Delta^2 S$ *be a delta of* $\Delta S$, *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* $\Delta S$ *by applying* $\Delta^2 S$.

Proposition 5.1 is the key observation for deriving from *putdelta* an incremental Datalog program $\partial put$ that computes $\Delta S$ more efficiently (Figure 5). To derive $\partial put$, we first incrementalize the Datalog program *putdelta* to obtain Datalog rules that compute $\Delta^2 S$ from the change $\Delta V$ on the view $V$. This step can be performed using classical incrementalization methods for Datalog [28]. We then use $\Delta^{2+}S$ in $\Delta^2 S$ as an instance of $\Delta S$ for applying to the source $S$.



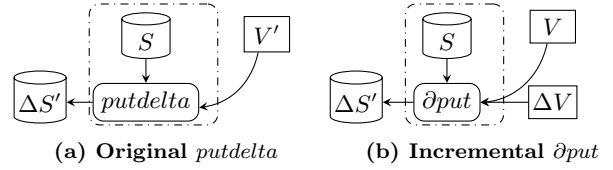**(a) Original** *putdelta*      **(b) Incremental** $\partial put$

**Figure 5: Incrementalization of** *putdelta.*

EXAMPLE 5.2 (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 $Y > 2$:*

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

*Let* $\Delta_V^+ / \Delta_V^-$ *be the set of insertions/deletions into/from the view $V$. We use two predicates $+v$ and $-v$ for $\Delta_V^+$ and $\Delta_V^-$, respectively. To generate delta rules for computing changes of $\pm r$ when the view is changed by $\Delta_V^+$ and $\Delta_V^-$, we adopt the incremental view maintenance techniques introduced in [28] but in a way that derives rules for computing the insertion set and deletion set for $\pm r$ separately. When $\Delta_V^+$ and $\Delta_V^-$ are disjoint, by applying distribution laws for the first Datalog rule, we derive two rules that define the changes to $\Delta_R^+$, a set of insertions $\Delta^+(\Delta_R^+)$ and a set of deletions $\Delta^-(\Delta_R^+)$, as follows:*

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

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

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

*Finally, as stated in Proposition 5.1, $\Delta^{2+}S$ 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 $\partial put$ as follows:*

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

*Because $\Delta_V^+$ and $\Delta_V^-$ 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.* $\square$

The incrementalization algorithm that transforms a putback program *putdelta* in nonrecursive Datalog with negation and built-in predicates into an equivalent program $\partial put$ 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 [18] 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

734

introduced in [28] 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 full paper for details [55]).

- *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 $\partial put$. This is because $\Delta^{2+}S$ can be used as an instance of $\Delta S'$ to apply to the source database $S$ (Proposition 5.1).

As shown in Example 5.2, 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 $\partial put$ is simplified to substituting $+v$ for positive predicate $v$ and $-v$ for negative predicate $\neg v$.

LEMMA 5.2. *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.*

# 6. IMPLEMENTATION AND EVALUATION

## 6.1 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 [9]. As mentioned in Subsection 4.5, 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 [4], 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 [10]. We use a similar approach to the translation from Datalog to SQL used in [29]. 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 [52] 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 [55] for details), (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, Φᵢ(X⃗) → ⊥> DO
    IF EXISTS (<SQL-query-of Φᵢ(X⃗)>) THEN
      RAISE "Invalid view updates";
    END IF;
  END FOR;
  -- Calculating and applying delta relations
  FOR EACH <source relation Rᵢ> DO
    CREATE TEMP TABLE Δ⁺_Rᵢ AS <sql-query-of +rᵢ>;
    CREATE TEMP TABLE Δ⁻_Rᵢ AS <sql-query-of −rᵢ>;
    DELETE FROM Rᵢ WHERE ROW (Rᵢ) IN Δ⁻_Rᵢ;
    INSERT INTO Rᵢ SELECT * FROM Δ⁺_Rᵢ;
  END FOR;
END;
```

## 6.2 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.

### 6.2.1 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 [52, 26], online tutorials [2, 3, 6, 5, 8] (triggers, sharded tables, and so forth), papers [15, 33] and our case study in Section 3.

- View update issues asked on online question & answer sites: Database Administrators Stack Exchange [1] and Stack Overflow Public Q&A [7].

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.

### 6.2.2 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 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

---

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

**Table 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-Datalog$^{\neg,=,<}$ | Validation Time (s) | Compiled 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 |

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.

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 6 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.

## 7. RELATED WORK

The view update problem is a classical problem that has a long history in database research [22, 20, 21, 11, 34, 48, 33, 40, 29, 16, 36, 44, 45, 46, 42, 41]. 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 [20, 21]. 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 [11, 35, 43, 41]. Keller [34] enumerates all view
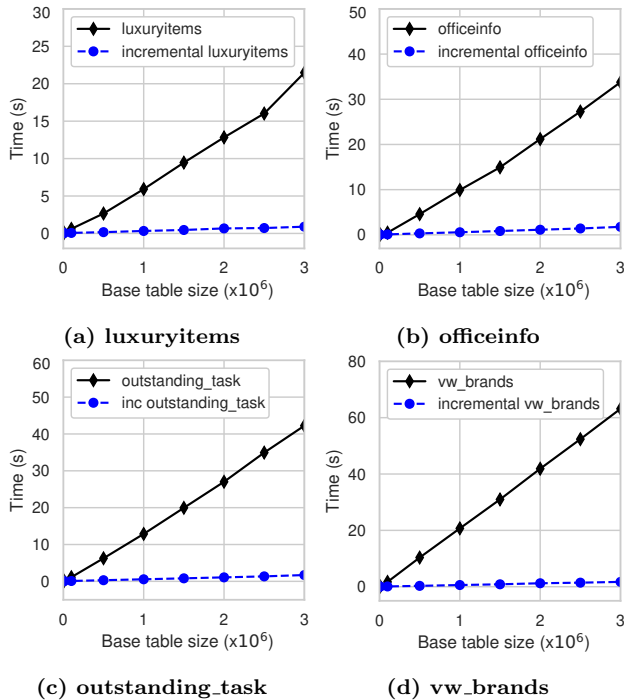
---

[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.

**Figure 6: View updating time.**

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 [34, 42]. Some other approaches restrict the syntax for defining views [21] that allow for unambiguous update propagation. Recently, intention-based approaches have been proposed to find relevant update policies for several types of views [44, 45, 46]. In another aspect, because some updates on views are not translatable, some works permit side effects of the view update translator [48] or restrict the kind of updates that can be performed on a view [33]. Some other works use auxiliary tables to store the updates, which cannot be applied to the underlying database [40, 29]. The authors of [16, 36] 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.

By generalizing view update as a synchronization problem between two data structures, considerable research effort has been devoted to bidirectional programming [19] for this problem not only in relational databases [15, 31] but also for other data types, such tree [25, 47], graph [30] or string data [14]. The prior work by Bohannon et al. [15] 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 [31] 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.

Melnik et al. [49] 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 [49], 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.

Our work was greatly inspired by the putback-based approach in bidirectional programming [32, 50, 51, 24, 38, 37]. The key observation in this approach is that thanks to well-behavedness, putback transformation uniquely determines the get one. In contrast to the other approaches, the putback-based approach provides 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 [38, 37], which supports programming putback functions declaratively while automatically deriving the corresponding unique forward transformation. Based on BiGUL, Zan et al. [56] 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. In contrast, 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.

## 8. CONCLUSIONS

In this paper, 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. By using nonrecursive Datalog with extensions as the language for describing view update strategies, we propose algorithms for validating user-written update strategies and optimizing update strategies before compiling them into SQL scripts to run effectively in RDBMSs. The experimental results show the performance of our framework in terms of both validation time and running time.

# 9. REFERENCES

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

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

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

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

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

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

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

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

[9] Z3: Theorem Prover. https://z3prover.github.io.

[10] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[12] V. Bárány, B. T. Cate, and L. Segoufin. Guarded negation. *J. ACM*, 62(3):22:1–22:26, June 2015.

[13] V. Bárány, B. ten Cate, and M. Otto. Queries with guarded negation. *PVLDB*, 5(11):1328–1339, 2012.

[14] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: Alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 193–204, 2010.

[15] A. Bohannon, B. C. Pierce, and J. 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.

[16] P. Buneman, S. Khanna, and W.-C. 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.

[17] A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *Journal of Web Semantics*, 14:57 – 83, 2012.

[18] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[19] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, pages 260–283. Springer Berlin Heidelberg, 2009.

[20] U. Dayal and P. A. Bernstein. On the updatability of relational views. In *Proceedings of the Fourth International Conference on Very Large Data Bases*, pages 368–377, 1978.

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

[22] R. Fagin, J. D. Ullman, and M. Y. Vardi. On the semantics of updates in databases. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 352–365, 1983.

[23] S. Fischer, Z. Hu, and H. Pacheco. A clear picture of lens laws. In *International Conference on Mathematics of Program Construction*, pages 215–223, 2015.

[24] S. Fischer, Z. Hu, and H. Pacheco. The essence of bidirectional programming. *Science China Information Sciences*, 58(5):1–21, May 2015.

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

[26] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, 2 edition, 2008.

[27] T. J. Green, D. Olteanu, and G. Washburn. Live programming in the logicblox system: A metalogiql approach. *PVLDB*, 8(12):1782–1791, 2015.

[28] A. Gupta, I. S. 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.

[29] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, and W. 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*, pages 1101–1116, 2017.

[30] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 205–216, 2010.

[31] R. Horn, R. Perera, and J. Cheney. Incremental relational lenses. *Proc. ACM Program. Lang.*, 2(ICFP):74:1–74:30, July 2018.

[32] Z. Hu, H. Pacheco, and S. Fischer. Validity checking of putback transformations in bidirectional programming. In *FM 2014: Formal Methods*, pages 1–15. Springer International Publishing, 2014.

[33] A. 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.

[34] A. M. Keller. Choosing a view update translator by dialog at view definition time. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 467–474, 1986.

[35] A. M. Keller and J. D. Ullman. On complementary and independent mappings on databases. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 143–148, 1984.

[36] B. Kimelfeld, J. Vondrák, and R. Williams. Maximizing conjunctive views in deletion propagation. *ACM Trans. Database Syst.*, 37(4):24:1–24:37, Dec. 2012.

[37] H.-S. Ko and Z. Hu. An axiomatic basis for bidirectional programming. *Proc. ACM Program. Lang.*, 2(POPL):41:1–41:29, Dec. 2017.

[38] H.-S. Ko, T. Zan, and Z. 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*, pages 61–72, 2016.

[39] C. 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.

[40] 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.

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

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

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

[44] Y. Masunaga. A relational database view update translation mechanism. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 309–320, 1984.

[45] Y. 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*, pages 13:1–13:8, 2017.

[46] Y. Masunaga, Y. Nagata, and T. 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*, pages 19:1–19:8, 2018.

[47] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. 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.

[48] C. B. Medeiros and F. W. Tompa. Understanding the implications of view update policies. *Algorithmica*, 1(1):337–360, Nov 1986.

[49] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.*, 33(4):22:1–22:50, Dec. 2008.

[50] H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for putback style bidirectional programming. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, pages 39–50, 2014.

[51] H. Pacheco, T. Zan, and Z. Hu. Biflux: A bidirectional functional update language for xml. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, pages 147–158, 2014.

[52] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., 2nd edition, 1999.

[53] F. Sáenz-Pérez, R. Caballero, and Y. García-Ruiz. A deductive database with datalog and sql query languages. In *Asian Symposium on Programming Languages and Systems*, pages 66–73, 2011.

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

[55] V.-D. Tran, H. Kato, and Z. Hu. Programmable view update strategies on relations. *CoRR*, abs/1911.05921, 2020.

[56] T. Zan, L. Liu, H. Ko, and Z. Hu. Brul: A putback-based bidirectional transformation library for updatable views. In *ETAPS*, pages 77–89, 2016.

[57] Z. Zhu, Y. Zhang, H.-S. Ko, P. Martins, J. a. Saraiva, and Z. Hu. Parsing and reflective printing, bidirectionally. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 2–14, 2016.