A Declarative Framework for Updatable Views in Relational Databases

Van-Dang Tran[2,1,a)] Hiroyuki Kato[1,b)] Zhenjiang Hu[3,1,c)]

Abstract: In this paper, we present the design and implementation of a framework for updatable views in relational databases. Our framework allows developers to use Datalog, a declarative language, for programming update strategies in order to make relational views updatable. We firstly implement an algorithm for automatically verifying the correctness of the user-written update strategy. Secondly, the verified strategy is translated into SQL trigger procedures that are automatically invoked in response to view update requests. We have successfully integrated our framework with PostgreSQL as the backend database system.

Keywords: View Update, Relational database, Datalog, Bidirectional Transformation

1. Introduction

The view concept plays an important role in relational databases. Since a view is defined by a query over source tables, it can be read as a normal table but cannot be updated. To be updatable, updated data in the view must be propagated to the source. However, in many cases, it is impossible to automatically propagate updates on the view to the source because there are multiple ways for each view update [1]. Although this view update problem has a long history in database literature [1], [2], [3], [4], [5], there is no standard solution yet. Commercial database systems such as PostgreSQL [6] provide very limited support for developers to create updatable views.

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. 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 [7]. By 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. Although existing RDBMSs provides SQL procedural languages such as PL/pgSQL [8] (in PostgreSQL) for implementing trigger procedures, 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] National Institute of Informatics, Chiyoda, Tokyo 101-8430, Japan
[2] The Graduate University for Advanced Studies (SOKENDAI), Hayama, Kanagawa 240-0193 Japan
[3] Peking University, Beijing, China
[a)] dangtv@nii.ac.jp
[b)] kato@nii.ac.jp
[c)] zhenjianghu@pku.edu.cn

nation	KEY	NAME	DESCRIPTION
	1	Japan	none
	2	China	none
	3	Vietnam	none

customer	KEY	NAME	ADDRESS	PHONE	NATIONKEY
	1	A	Tokyo	1234	1
	2	B	Hanoi	3241	3
	3	C	Beijing	5345	2

jcustomer	KEY	NAME	ADDRESS
	1	A	Tokyo

Fig. 1 Base tables (`nation` and `customer`) and view (`jcustomer`)

In this paper, we aim to design and implement a framework, which frees programmers from the burden of manually creating triggers and trigger procedures on updatable views. By this framework, programmers can declaratively specify their view update strategies in Datalog. The Datalog-based update strategies are then verified before compiled to SQL scripts for creating a real updatable view in relational database management systems (RDBMSs). We have integrated our framework with PostgreSQL as the backend database system. The prototype implementation of our framework is available at [9] and is used to implement the running example in this paper.

2. Running Example

As our running example, consider a database of two tables, `nation` and `customer`, and a view `jcustomer` as shown in Figure 1. The view `jcustomer`, which contains all customers having Japanese citizenship, is defined by a Datalog query [10] over the two base tables as the following:

```
jcustomer(K, N, A) :- customer(K, N, A, P, NK), nation(NK,
      NATION, D), NATION='Japan'.
```

That is a join over the tables `customer` and `nation` on the attribute `NATIONKEY` with a condition that the nationality is `Japan`. We keep only three attributes `KEY`, `NAME` and `ADDRESS` from the table `customer` in the view. This defining query of `jcustomer`

```
1  % Constraint on the source:
2  ⊥ :- not nation(_, 'Japan', _).
3
4  % Update strategy
5  -customer(K, N, A, P, NK) :- customer(K, N, A, P, NK),
       nation(NK, NATION, _), NATION ='Japan', not
       jcustomer(K, N, A).
6
7  tmp(K, N, A) :- customer(K, N, A, _, NK), nation(NK, '
       Japan', _).
8  +customer(K, N, A, P, NK) :- jcustomer(K, N, A), not
       tmp(K, N, A), nation(NK, 'Japan', _), customer(K,
       _, _, P, _).
9  +customer(K, N, A, P, NK) :- jcustomer(K, N, A), not
       tmp(K, N, A), nation(NK, 'Japan', _), not
       customer(K, _, _, _, _), P = 'unknown'.
```

Fig. 2 An update strategy for `jcustomer`

can be considered as a forward transformation *get* that takes as input the source database, which is the pair of tables `customer` and `nation`, to produce the view `jcustomer`:

$$\texttt{jcustomer} = \mathit{get}(\langle \texttt{nation}, \texttt{customer} \rangle)$$

To illustrate the ambiguity of propagating updates on the view `jcustomer`, let's consider a state of the base tables and the view as in Figure 1 and a simple request to delete tuple ⟨`1, A, Tokyo`⟩ from the view `jcustomer`. Obviously, there are three options for propagating this deletion to the source database. The first option is to delete from the table `customer` the tuple ⟨`1, A, Tokyo, 1234, 1`⟩. The second is to delete from the table `nation` the tuple ⟨`1, Japan, none`⟩. The third is to perform both deletions in the first and the second options.

Because of the ambiguity issue, to make `jcustomer` updatable, we need to explicitly specify a strategy for propagating all updated data on the view to the source tables. The update strategy can be formulated as a putback transformation *put* that maps the updated view `jcustomer'` back to an updated source database ⟨`nation'`, `customer'`⟩ as the following:

$$\langle \texttt{nation'}, \texttt{customer'} \rangle = \mathit{put}(\langle \texttt{nation}, \texttt{customer} \rangle, \texttt{jcustomer'})$$

In addition to the updated view, we also need the original source tables as the input for *put* in order to recover all information of the source tables, which is discarded in the view.

Interestingly, it is still possible to use Datalog to specify such a putback transformation *put*. The idea is to use delta predicates to describe how to update data in the source tables by using the original source tables and the updated view. Figure 2 show an update strategy for the view `jcustomer`, where the predicate `customer` preceded by a symbol +/- corresponds to the set of tuples being inserted into/deleted from the source table `customer`.

We briefly describe our update strategy in the Datalog program of Figure 2. We assume that in the source table `nation` there exists a tuple having the attribute `NATION` equal to `'Japan'`: $\exists x, y, \texttt{nation}(x, \text{`}Japan', y)$. We express this constraint by a special Datalog rule with a truth constant $\bot$ in the head as in line 2 of Figure 2. The meaning of this rule is a first-order logic sentence $(\neg\exists x, y, \texttt{nation}(x, \text{`}Japan', y)) \rightarrow \bot$, which is equivalent to $\exists x, y, \texttt{nation}(x, \text{`}Japan', y)$. Given an updated view `jcustomer`,

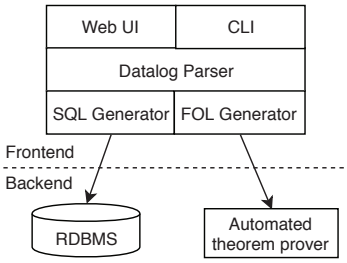

Fig. 3 System architecture

our update strategy is to keep the table `nation` unchanged, and update the table `customer` to reflect the view updates as follows. First, if there is a Japanese customer, who does not appear in the view, we choose the option of deleting this customer from the source table `customer` (line 5 in Figure 2) that is more reasonable than deleting the tuple $\langle 1, \text{`}Japan' \rangle$ from the table `nation`. Second, if there is a customer in the view `jcustomer` but there is no Japanese customer in the source tables having the same values for `KEY, NAME, ADDRESS`, then we insert a new customer to the table `customer` (lines 7, 8 and 9 in Figure 2). More concretely, to fill in the attribute `NATIONKEY`, we find a key from the table `nation` where the nationality is `'Japan'`. Due to the constraint on `nation` presented before, we can always find such a key. To fill in the attribute `PHONE`, we search for the existing one in the old table `customer`. If it is not found, we fill in the attribute `PHONE` with a default value `'unknown'`.

To guarantee the consistency between the view and the source tables, the pair of backward and putback transformations, so-called a bidirectional transformation [11], [12], must satisfy the following properties for any updated view V' and any source database S:

$$\mathit{put}(S,\ \mathit{get}(S)) = S \qquad \text{(GetPut)}$$

$$\mathit{get}(\mathit{put}(S,\ V')) = V' \qquad \text{(PutGet)}$$

The GetPut property ensures that if the view is unchanged then the base tables are unchanged, while the PutGet property ensures all view updates are completely propagated to the source so that the updated view can be computed again from the query *get* over the updated source.

3. Designing Architecture

The overall architecture of our proposed framework is shown in Figure 3. The framework consists of two main parts. The upper part is the frontend, which provides browser-based and command line interfaces for users to write Datalog programs. Our framework accepts the standard syntax of Datalog [10] with two symbols + and - for denoting delta predicates of insertions and deletions, respectively. We allow users to use Datalog to write view definitions, integrity constraints and view update strategies. Recursion in Datalog is currently not allowed and is our future work. From an input Datalog program, we first generate first-order logic (FOL) sentences corresponding to GetPut and PutGet properties of the view definition (forward transformation *get*) and the update strategy (putback transformation *put*). We then translate the Datalog program to a SQL script for creating the corresponding updatable view in RDBMSs.

The bottom part consists of two backend systems including an *automated theorem prover* for validating all the generated first-order sentences and an RDBMS for executing the generated SQL script. By integrating with existing tools for the automated theorem prover, our framework assists developers in verifying their update strategies. The corresponding updatable views with all the necessary triggers are also automatically created in RDBMS without more users' interactions.

4. Implementation

We have implemented our framework in Ocaml and integrate the framework with Lean theorem prover [13] and PostgreSQL database [6] as the backend systems. In this section, we present the two main aspects of the implementation of our framework: FOL generator for transforming GetPut and PutGet properties of the input Datalog program to first-order logic sentences; and SQL generator for generating SQL script that creates the specified updatable view in RDBMSs.

4.1 From Datalog to First-Order Logic

It is well known that the expressiveness of non-recursive Datalog is equivalent to relational calculus [14], which is a form of first-order (FO) formulas. Given a non-recursive Datalog program, there is a transformation from each output relation to an FO formula. We implement this transformation by a function, named `fol-of-query`. For example, the FO formula for the predicate `-customer` of the Datalog program *put* in our running example is the following:

```
fol-of-query (put, -customer) = ∃ NATION, customer(K,N,A,P,
    NK) ∧ (∃ D, nation(NK,NATION,D)) ∧ NATION='Japan' ∧
    ¬ jcustomer(K, N, A)
```

Data integrity constraints specified in the Datalog program can be also transformed to FO sentences by using the function `fol-of-query` if we consider $\bot$ as a special output relation that must be empty. In this way, the FO transformation for constraints is implemented by a function, named `fol-of-constraints`, as follows:

```
fol-of-constraints (put) = (fol-of-query (put, ⊥)) → ⊥
```

The GetPut property requires that if the view is unchanged, meaning that the view is computed by the query *get*, there is no update to the source. This means all delta relations resulted by the Datalog program *put*, where the view is computed by *get*, must be empty. For this composition of *put* after *get*, we can simply construct an equivalent Datalog program, denoted by *getput*, by merging all rules in *put* and *get*. For each delta relation, let φ_i be the corresponding FO formula, this delta relation is empty if there is no tuple $\vec{X_i}$ such that $\varphi_i(\vec{X_i})$ is true, i.e., the sentence $\exists \vec{X_i}, \varphi_i(\vec{X_i})$ does not hold. Therefore, we take a conjunction $\Phi = \bigwedge_i(\neg\exists \vec{X_i}, \varphi_i(\vec{X_i}))$ that does not hold if, and only if, the GetPut property holds. In this way checking the GetPut property is reduced to checking the validity of the FO sentence Φ. We implement a function `getput-sentence` for generating this FO sentence of the GetPut property. For example, this sentence for the Datalog program in our running example is the following:

```
getput-sentence (get, put) =
    (¬ ∃ K,N,A,P,NK, fol-of-query (getput, +customer)) ∧ (
     ¬ ∃ K,N,A,P,NK, fol-of-query (getput, -customer))
```

The PutGet property says that given an updated view V', after updating the source by the *put*, V' must be exactly the same as the result of the defining query *get* over the updated source: $V' = get(put(S, V'))$. In other words, every tuple $\vec{X}$ in V' must be in the result of $get(put(S, V'))$, and vice versa. Let *putget* be the query $get(put(S, V'))$ and φ_{putget} be the FO formula equivalent to *putget*. The PutGet property holds if, and only if, the sentence $\Psi = \forall \vec{X}, \varphi_{putget}(\vec{X}) \leftrightarrow V(\vec{X})$ holds. Therefore, checking the PutGet property is reduced to checking the validity of Ψ. We implement a function `putget-sentence` for generating this FO sentence. For example, this sentence for the Datalog program in our running example is the following:

```
putget-sentence (get, put) = ∀ K,N,A, fol-of-query (putget,
      jcustomer) ↔ jcustomer(K, N, A)
```

To check the validity of the FO sentences of GetPut and PutGet properties, we feed these sentences to Lean theorem prover. It is remarkable that the Datalog program for a view update strategy may contain integrity constraints. These constraints should be used as hypothesises for validating FO sentences of GetPut and PutGet properties. In this way, we finally generate a proof script in Lean theorem prover for checking the validity of FO sentences. The proof script performs several tactics for automatically searching a proof that proves the FO sentences are valid. For example, the proof script for the FO sentence of GetPut property in our running example is the following:

```
theorem getput {customer:int → string → string → string
      → int → Prop} {nation: int → string → string →
     Prop}: (fol-of-constraints (put)) → (getput-sentence
     (get, put)):=
    begin
        intro h,
        rw[imp_false] at *,
        simp at *,
        revert h,
        z3_smt,
    end
```

4.2 From Datalog to SQL

In fact, for every non-recursive Datalog query, there exists an equivalent SQL query [14]. We implement the transformation from Datalog queries to SQL queries by a function, named `sql-of-query`. By this transformation, the view definition in Datalog is translated into a SQL statement of creating the view. For example, the following is for creating the view `jcustomer`

```
CREATE VIEW jcustomer AS sql-of-query (get, jcustomer);
```

The view update strategy *put* is transformed into SQL statements to create triggers and associated procedures on the view. When there are view update requests, these trigger procedures are automatically invoked to perform the following steps:

Step 1: Based on each `INSERT`/`DELETE`/`UPDATE` statement in the view update requests, derive the set of tuples to be inserted/deleted to/from the view Δ_V^+ / Δ_V^-. This step is performed by the following trigger procedure, which is associated with an `INSTEAD OF` trigger on the view.

```
CREATE OR REPLACE FUNCTION public.v_update()
```

```
RETURNS TRIGGER
LANGUAGE plpgsql
BEGIN
    IF TG_OP = 'INSERT' THEN
        DELETE FROM Δ_V^- WHERE ROW(V) = NEW;
        INSERT INTO Δ_V^+ SELECT (NEW).*;
    ELSIF TG_OP = 'UPDATE' THEN
        DELETE FROM Δ_V^+ WHERE ROW(V) = OLD;
        INSERT INTO Δ_V^- SELECT (OLD).*;
        DELETE FROM Δ_V^- WHERE ROW(V) = NEW;
        INSERT INTO Δ_V^+ SELECT (NEW).*;
    ELSIF TG_OP = 'DELETE' THEN
        DELETE FROM Δ_V^+ WHERE ROW(V) = OLD;
        INSERT INTO Δ_V^- SELECT (OLD).*;
    END IF;
END;
```

The updated view is obtained by applying these changes, Δ_V^+ and Δ_V^-, on a materialization of the view.

Step 2: Checking integrity constraints: recall that a constraint has the form $\bot \leftarrow L_1, \ldots, L_n$. By considering $\bot$ as a special output relation, which must be empty in the Datalog program, we can also use the function `sql-of-query` to generate the corresponding SQL query of $\bot$ that must be empty. In this way, we create a boolean SQL expression of the form `EXISTS (sql-of-query (`*get*`, ⊥))`. If this expression returns a true value, update on the view is rejected:

```
IF EXISTS (sql-of-query (get, ⊥))
  THEN RAISE check_violation USING MESSAGE ='Invalid
     update on view';
END IF;
```

Step 3: Calculate all insertions and deletions on the source: in order to calculate all the insertions and deletions to the source, we translate each delta predicate $+R/-R$ in the Datalog program to an equivalent SQL query that results in a set of tuple need to be inserted/deleted into/from the corresponding table R.

```
CREATE TEMPORARY TABLE Δ_{R1}^+ WITH OIDS ON COMMIT DROP AS
     SELECT * FROM sql-of-query(put, +R1);
CREATE TEMPORARY TABLE Δ_{R1}^- WITH OIDS ON COMMIT DROP AS
     SELECT * FROM sql-of-query(put, -R1);
...
CREATE TEMPORARY TABLE Δ_{Rn}^+ WITH OIDS ON COMMIT DROP AS
     SELECT * FROM sql-of-query(put, +Rn);
CREATE TEMPORARY TABLE Δ_{Rn}^- WITH OIDS ON COMMIT DROP AS
     SELECT * FROM sql-of-query(put, -Rn);
```

Step 4: We apply each set of insertions (Δ_R^-) and deletions (Δ_R^+) calculated in Step 3 to the corresponding source table R. The application operation is to remove all the tuples in Δ_R^- from R and then add all tuples in Δ_R^+ to R. This is performed by `INSERT` and `DELETE` SQL statements as follows:

```
DELETE FROM R1 WHERE ROW (R1) IN (SELECT * FROM  Δ_{R1}^-);
INSERT INTO R1 SELECT * FROM Δ_{R1}^+;
...
DELETE FROM Rn WHERE ROW (Rn) IN (SELECT * FROM  Δ_{Rn}^-);
INSERT INTO Rn SELECT * FROM Δ_{Rn}^+;
```

5. Conclusion

In this paper, we have presented the design and implementation of our proposed framework for updatable views in relational databases. Our framework lets database administrators declaratively implement their update strategies for making views updatable. The framework brings advantage to users by automatically verifying user-written update strategies and generating all the SQL code to create updatable views in RDBMSs.

Acknowledgments This work is partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (S) No. 17H06099.

References

[1] Dayal, U. and Bernstein, P. A.: On the Correct Translation of Update Operations on Relational Views, *ACM Trans. Database Syst.*, Vol. 7, No. 3, pp. 381–416 (online), DOI: 10.1145/319732.319740 (1982).

[2] Keller, A. M.: Choosing a View Update Translator by Dialog at View Definition Time, *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., pp. 467–474 (online), available from ⟨http://dl.acm.org/citation.cfm?id=645913.671458⟩ (1986).

[3] Keller, A. M.: Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins, *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '85, New York, NY, USA, ACM, pp. 154–163 (online), DOI: 10.1145/325405.325423 (1985).

[4] Masunaga, Y.: An Intention-based Approach to the Updatability of Views in Relational Databases, *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, IMCOM '17, New York, NY, USA, ACM, pp. 13:1–13:8 (online), DOI: 10.1145/3022227.3022239 (2017).

[5] Bohannon, A., Pierce, B. C. and Vaughan, J. A.: Relational Lenses: A Language for Updatable Views, *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, New York, NY, USA, ACM, pp. 338–347 (online), DOI: 10.1145/1142351.1142399 (2006).

[6] PostgreSQL: The World's Most Advanced Open Source Relational Database. `https://www.postgresql.org`.

[7] Garcia-Molina, H.: *Database systems: the complete book*, Pearson Education India (2008).

[8] PL/pgSQL: SQL Procedural Language. `https://www.postgresql.org/docs/9.6/plpgsql.html`.

[9] BIRDS: Bidirectional Transformation for Relational View Update Datalog-based Strategy. `https://dangtv.github.io/BIRDS/`.

[10] Ceri, S., Gottlob, G. and Tanca, L.: What you always wanted to know about Datalog (and never dared to ask), *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 1, pp. 146–166 (online), DOI: 10.1109/69.43410 (1989).

[11] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C. and Schmitt, A.: Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem, *ACM Trans. Program. Lang. Syst.*, Vol. 29, No. 3 (online), DOI: 10.1145/1232420.1232424 (2007).

[12] Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A. and Terwilliger, J. F.: Bidirectional Transformations: A Cross-Discipline Perspective, *Theory and Practice of Model Transformations* (Paige, R. F., ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 260–283 (2009).

[13] Lean: A theorem prover. `https://leanprover.github.io`.

[14] Abiteboul, S., Hull, R. and Vianu, V.(eds.): *Foundations of Databases: The Logical Level*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition (1995).