BIRDS: Programming view update strategies in Datalog

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

In relational database management systems, views are rarely automatically updatable because of the inherent ambiguity of view updates. To allow view updates, database administrators have to decide and implement an update strategy that must be well-behaved with the view definition to guarantee consistency between the view and the underlying database. In this demonstration, we explore the development process of such view update strategies with the assistance of our framework, called BIRDS. BIRDS enables users to specify view update strategies declaratively using Datalog. BIRDS validates the well-behavedness of user-written update strategies, then optimizes and compiles them into SQL code run in PostgreSQL databases. BIRDS further explains the unexpected behavior of user-written Datalog programs using generated counterexamples, thereby assists users in correcting their programs. We demonstrate all the steps in developing view update strategies via an easy to use interface provided by our system.

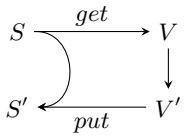
PVLDB Reference Format:
Van-Dang Tran, Hiroyuki Kato, Zhenjiang Hu. BIRDS: Programming view update strategies in Datalog. *PVLDB*, 13(12): 2897-2900, 2020.
DOI: https://doi.org/10.14778/3415478.3415503

1. INTRODUCTION

View is an important concept in relational databases. A view can be updated by translating the updates into the corresponding updates on the base tables [7, 8]. However, in many cases, it is impossible to automatically propagate view updates to the source because there are multiple ways for each view update.

Because of the ambiguity of view updates, in commercial database systems such as PostgreSQL [1], very restricted types of views are automatically updatable [2]. To make an arbitrary view updatable, database administrators have to explicitly decide and implement a strategy for propagating view updates to the source tables. This propagation can be implemented in a trigger procedure associated with a trigger on the view. The trigger procedure is automatically invoked in response to update requests on the view. Although there exist procedural languages such as PL/pgSQL in PostgreSQL for trigger procedures, developing a view update strategy is nontrivial and error-prone.

In general, given an updated view, to propagate it to the source, we must also use the original state of the source. An update strategy can be formulated as a so-called *putback transformation put* [10], which takes as input the original source database S and an updated view V' to produce a new source database S'. *put* and the view definition, denoted as *get*, form a bidirectional transformation [6] as the following.



To guarantee consistency between the source and the view, *put* must be well-behaved with *get* in the sense that they satisfy the following *round-tripping* properties:

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

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

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.

In our demonstration, we explore the development process of view update strategies with novel techniques for guaranteeing the well-behavedness of view updates. This is based on our prototype, called BIRDS [12] that enables users to use Datalog to specify view update strategies. BIRDS validates the well-behavedness of user-written Datalog programs, optimizes and compiles them into SQL code run in a PostgreSQL RDBMS. Furthermore, in this demonstration, we present a novel technique in BIRDS that generates counterexamples and explains the unexpected behaviours of view update strategies, thereby assists users in correcting their programs. The source code of our prototype and demonstrated examples is available at `https://dangtv.github.io/BIRDS/`.

2. USER INTERFACE

The users interact with the BIRDS system via a web-based graphical interface (WebUI) shown in Figure 1, including the following features:

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415503

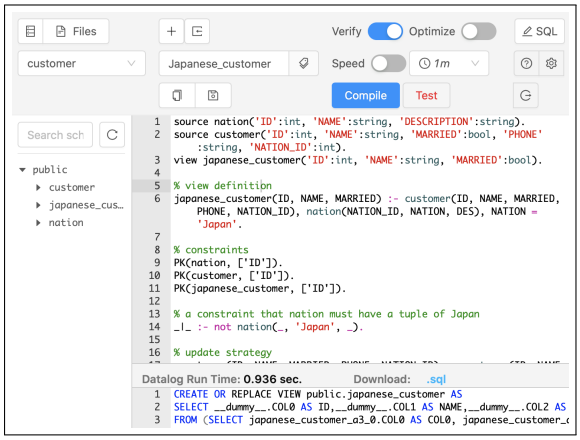


Figure 1: Web-based user interface (WebUI).

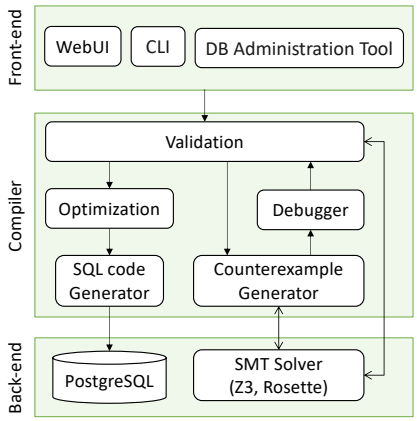


Figure 2: System architecture.

(1) Database tool: Connecting to PostgreSQL to interact with the database such as importing data schema or running compiled SQL code to create updatable views in PostgreSQL.

(2) Online editor: The user writes a view update strategy in Datalog by an online editor. By connecting to PostgreSQL, the editor can also import the data schema from the database.

(3) Compilation: The user submits the hand-written Datalog program to compile into SQL code and run the SQL code in PostgreSQL.

(4) Verification and optimization: The user can enable validating and optimizing before compiling the hand-written Datalog program.

(5) Counterexample generation: Generating a counterexample that is an instance of the view and source tables to show that the program breaks the round-tripping properties.

(6) Debugging: Given instances of the view and the source tables, inspecting the Datalog program to understand the unexpected behaviour of the program.

3. SYSTEM ARCHITECTURE

The overall architecture of our system is shown in Figure 2. BIRDS consists of two main parts, front-end and compiler, and is integrated with back-end systems including a PostgreSQL RDBMS and SMT solvers. The front-end provides a web-based interface (WebUI) and a command line tool (CLI) for users to write and submit Datalog programs to the system. The compiler takes the user-written Datalog programs, validates and compiles them into SQL code run in the PostgreSQL database. The compiler also allows users to debug the unexpected behaviour of their Datalog programs.

3.1 The Datalog Syntax

Figure 3 shows an example of a view update strategy written in Datalog, where the view `japanese_customer` is defined over two source tables, `customer` and `nation`. The Datalog program (partially shown in Figure 3) contains four parts: schema declaration, rules for the definition of `japanese_customer` (*get*), integrity constraints, and rules for the view update strategy (*put*). Specifically, our framework accepts the standard syntax of Datalog [5] without recursion for programming view update strategies as follows.

Schema declaration. We use two keywords `source` and `view` to distinguish the view and source tables. Each column of each relation is assigned one of the following data types: boolean, integer, real, and string. We accept multiple source tables but only one view for each program.

View definition (*get*). The view can be defined over the source tables by normal Datalog rules. BIRDS also allows Datalog extensions including negation and built-in predicates (e.g., $=$, $<$, and so forth).

Delta predicates. A delta predicate is a normal predicate following a symbol $+$ or $-$. Given a table R, $+R$ corresponds to the set of tuples inserted into R and $-R$ corresponds to the set of tuples deleted from R.

Update strategy (*put*). By using the delta predicates, we can use normal Datalog rules to specify which tuples are inserted into or deleted from the source tables for a given updated view. In these update rules, the rule head is a delta predicate and the rule body is a conjunction of the view, the source tables, other intermediate relations introduced in the program, and built-in predicates.

Integrity constraints. We allow using Datalog rules with a special symbol $\bot$ in the head to specify data integrity constraints. Such a Datalog rule means that the rule body does not hold for any tuples. For example, the following rule means no tuple of r has a value of X less than 2:

$$\bot :- r(X, Y), X < 2.$$

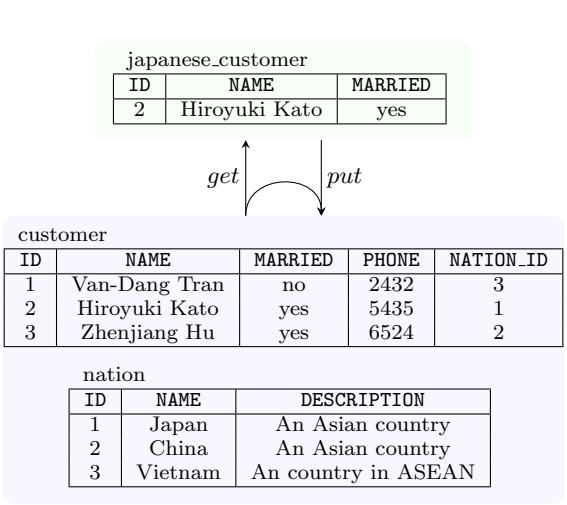
BIRDS also provides a shorthand syntax to conveniently declare primary keys on relations (see Line 7 in the program of Figure 3). Given a relation $r(A, B)$, we declare a primary key on A by $PK(r, [A])$ that is an abbreviation for the following:

$$\bot :- r(X, Y_1), r(X, Y_2), \neg Y_1 = Y_2.$$

3.2 Workflow

A typical workflow for developing a view update strategy includes the following steps. First, the user submits a Datalog program to the validation to verify its well-behavedness. The system verifies and returns a counterexample to the user if the user's update strategy is not correct. The debugger tests the round-tripping properties over the counterexample and shows an explanation of the unexpected behaviour. The user fixes the program and runs the validation again.

Validation. Checking whether the view update strategy *put* is well-behaved, i.e., satisfies the GetPut and PutGet




```
1  source nation('ID':int, 'NAME':string, 'DESCRIPTION':string).
2  source customer('ID':int, 'NAME':string, 'MARRIED':bool,
       'PHONE':string, 'NATION_ID':int).
3  view japanese_customer('ID':int, 'NAME':string,  'MARRIED':
       bool).
4  % View definition (get):
5  japanese_customer(ID, N, M) :- customer(ID, N, M, P, NID),
       nation(NID, NATION, D), NATION = 'Japan'.
6  % Integrity constraints:
7  PK(nation, ['ID']). % a primary key on column 'ID'
8  ...
9  % Update strategy (put):
10 -customer(ID, N, M, P, NID) :- customer(ID, N, M, P, NID),
       nation(NID, NATION, _), NATION = 'Japan', not
       japanese_customer(ID, N, M).
11 existing_japanese_customer(ID, N, M, P, NID) :- customer(ID,
       N, M, P, NID), nation(NID, 'Japan', _).
12 +customer(ID, N, M, P, NID) :- japanese_customer(ID, N, M),
       not existing_japanese_customer(ID, N, M, _, _), nation(
       NID, 'Japan', _), customer(ID, _, _, P, _).
13 ...
```

Figure 3: An example of view update strategy for a view `japanese_customer` and two source tables `nation` and `customer`.

properties, with *get* plays an essential role in our framework. Clearly, to be well-behaved, there should be no instance of the source and view for which the Datalog program of *put* triggers simultaneously an insertion and deletion of the same tuple. BIRDS also checks this disjointness of insertions and deletions. Furthermore, in the case that *put* is well-behaved with another view definition, BIRDS can also discover such a view due to its uniqueness as proven in previous work [9]. BIRDS guarantees soundness and completeness of the validation if all Datalog rules are negation guarded [12]. The validation is sound when the guarded negation restriction is not satisfied. BIRDS uses Z3 SMT solver [4] as the backend system in checking the well-behavedness of *put*.

Counterexample generation. If the view update strategy does not pass the validation, BIRDS generates a counterexample for the satisfaction of the round-tripping properties. A counterexample is an instance of the view and source tables. The counterexample should be as simple as possible, i.e. the sizes of the view and source tables are minimum, for easy debugging. Recall that for the GetPut property, if we run *get* and then run *put*, the source is unchanged. For the PutGet property, if we run *put* to update the source and then run *get* we will get the same view as the initial one. We encode the evaluation of both *get* and *put* into equivalent Racket functions over symbolic view and source tables, and thereby encode GetPut and PutGet into constraints in Rosette [11]. Rosette is a solver-aided extension to Racket [3] that compiles Racket programs into logic constraints that are checked by satisfiability solvers such as Z3 [4]. We iteratively increase the size of the symbolic view and source tables and call the Rosette symbolic execution runtime to check the constraints. From the result returned Rosette which is an interpretation of all the symbolic values, we instantiate the view and the source tables as a counterexample that the round-tripping properties are not satisfied.

Debugger. By evaluating *get* and *put* over the generated counterexample, BIRDS automatically detects and explains the unexpected behaviour of the user-written program.

Optimization. BIRDS further optimizes *put* by exploiting its well-behavedness and integrating it with the standard incrementalization methods for Datalog. Intuitively, as mentioned before, if the view is unchanged, i.e., the view is the result of *get*, *put* makes no update on the source. An update on the view leads to some changes in the output of *put* and therefore makes the source updated. This is the key observation to optimize *put* by transforming it into an incremental program that computes source updates from view updates more efficiently.

SQL code generation. BIRDS finally generates SQL code for implementing the view update strategy specified in the Datalog program. The SQL code is run in a back-end PostgreSQL database system to create the corresponding updatable view. Specifically, the SQL code contains procedures associated with triggers on the view to compute all the delta predicates in the Datalog program before using them to update the source tables. Since there is no recursion in the Datalog program, the view update strategy can be run efficiently by the equivalent SQL code.

4. DEMONSTRATION SCENARIOS

Our demonstration consists of several sample databases with examples for users to understand the system. We start the demonstration by a brief introduction to the system architecture. We then show sample databases with views that can not be updated in PostgreSQL and how to use BIRDS to program update strategies that make these views updatable. The user can either write SQL statements or use a database administration tool to update the created views and see changes reflected in the database.

Taking a sample database provided in our demonstration such as the one shown in Figure 3, the user can also write their own update strategies for a view. With the assistance of our system, the user will experience the following steps:

(1) The user can consider a view over some other source tables and have an update strategy in mind for the view. The user writes Datalog rules to describe the view update strategy via the BIRDS online editor.

(2) The user submits the written Datalog program to BIRDS to compile into SQL code and run in PostgreSQL to create the corresponding updatable view.

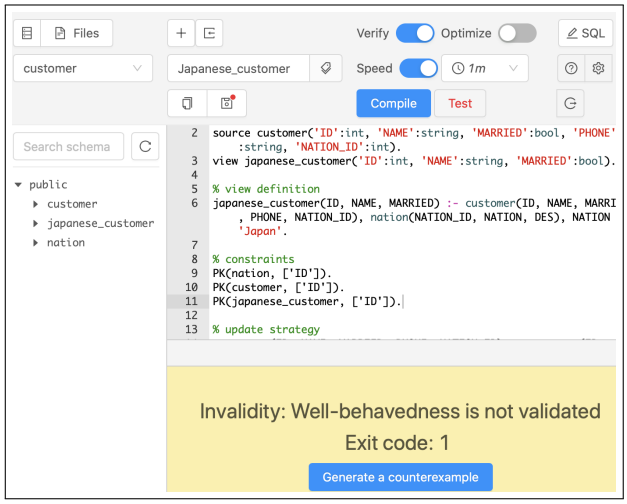


Figure 4: A validation result.

```
>>> The putget property is not satisfied:
+-------------- Updated View -----------------+
| japanese_customer(1, '..aa..', true)
+-----------------------------------------------+
     ||
     || put to the source
     \/
+-------------- Deltas ------------------------+
| empty
+-----------------------------------------------+
     ||
     || apply to the source
     \/
+---------- New source ------------------------+
| customer(0, '..aa..', true, '..aa..', 1)
| nation(0, 'Japan', '..aa..')
+-----------------------------------------------+
     ||
     || get the view again
     \/
+---------- Recomputed View ------------------+
| empty
|      >>> Unexpected result:
|      The following tuples are missing:
|         japanese_customer(1, '..aa..', true)
+-----------------------------------------------+
```

Figure 5: An explanation for the PutGet violation w.r.t a generated counterexample.

(3) The user connects to the PostgreSQL database to try some updates on the view and check changes applied to the source tables as specified in the Datalog program.

(4) The user enables BIRDS to validate whether the update strategy is well-behaved with the view definition.

(5) For the cases where the user-written Datalog program does not pass the validation (e.g., the validation result shown in Figure 4), the user uses BIRDS to generate a counterexample that shows a round-tripping property (GetPut or PutGet) is not satisfied.

(6) The user applies the generated counterexample to the Datalog program and debugs the unexpected behaviour of the program with the assistance of BIRDS. BIRDS evaluates the Datalog program over the counterexample and generates an explanation that illustrates the violation of a round-tripping property. Figure 5 shows an example of the explanation for the PutGet violation of the view update strategy in Figure 3. Specifically, a counterexample with values that match the schema declaration is generated. For example, a string `'..aa..'` is generated for the attribute `PHONE`, which has type `string` in the schema declaration. Over the updated view and the source given in the counterexample, we run the *put* Datalog rules to calculate delta relations, i.e., updates to the source, then apply them and recompute the view by the view definition *get*. The recomputed view is compared with the initial one to check whether they are equivalent as specified in PutGet and show the wrong or missing tuples in the recomputed view to the user.

(7) The user corrects the Datalog program to fix the unexpected behaviour. The user runs the validation again by using BIRDS to confirm the correctness.

5. CONCLUSIONS

In this proposal, we have presented a demonstration of our BIRDS system in solving the view update problem in practice. We hope to convey to the visitors how to write a view update strategy in Datalog and the important properties of the hand-written program required for consistent view updates.

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.

6. REFERENCES

[1] PostgreSQL. `https://www.postgresql.org`.

[2] PostgreSQL 9.6.17 Documentation: CREATE VIEW. `https://www.postgresql.org/docs/9.6/sql-createview.html`.

[3] The Racket programming language. `racket-lang.org`.

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

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

[6] 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.

[7] U. Dayal and P. A. Bernstein. On the updatability of relational views. In *VLDB*, pages 368–377, 1978.

[8] 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.

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

[10] 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.

[11] E. Torlak and R. Bodík. Growing solver-aided languages with rosette. In *Onward!*, 2013.

[12] V.-D. Tran, H. Kato, and Z. Hu. Programmable view update strategies on relations. *PVLDB*, 13(5):726–739, 2020.