View Updates seen through the Lens of Synchronization

Daniel Spoonhower

Carnegie Mellon University
Pittsburgh, Pennsylvania
spoons+@cs.cmu.edu

Abstract

We describe an implementation of mutable views in a relational database based on previous work in programming languages for describing data synchronization. This work allows us to uniformly handle updates to both concrete relations and to views. Through the introduction of a new language primitive, our implementation supports updates to a restricted class of relational views. We also discuss opportunities to support updates on broader classes of relational views.

1 Introduction

In a relational database, the structure of each relation is defined permanently at the moment of its creation. However, this structure may not be the one best suited to every user. *Views* provide a mechanism to define a *virtual* relation in terms of a query on the underlying relations of a database. Views may provide a means of security by restricting what data a given user can see. In addition, views allow large schema to be presented in more intuitive ways and effectively allow a database to be "customized" – each user or application can see a different set of attributes and records, all derived from the same underlying relations.

Although views have many benefits, there is one significant drawback of using views: given an arbitrary update to a view, there is no unambiguous way to update the underlying relation. Therefore, in most database implementations, views are read-only structures. The difficulty of propagating changes in the view down to the underlying relation is known as the *view update* problem.[1] Previous work [1, 2, 4] has explored where updates on views may be unambiguously translated to updates on the underlying relations.

Recently, languages describing the synchronization of data structures have been investigated by Greenwald *et al* [3]. These programming languages describe *bidirectional* transformations, both those from a concrete representation to an abstract view and back from an abstract view into a concrete form. The relationship between languages for synchronization and view updates is further discussed in Section 2.

These bidirectional transformations form an important basis for the present work. The primary contributions of this work are:

- an implementation of single-relation views supporting updates (following the work of Greenwald *et al*)
- the introduction of a new synchronization primitive used in the implementation of views defined by the join of two (or more) relations

Section 3 gives translations for typical SQL queries into the synchronization language and explains how lenses are used to implement database updates.

In working toward an implementation that supports updates on relational views, we considered many different extensions to the synchronization language; several candidate extensions are discussed in Section 4 along with a proof of some important properties of our chosen candidate. We also discuss several types of joins that are *not* mutable in our framework.

As we have concentrated on those aspects most closely related to the view update problem, our implementation is far from complete. Section 5 discusses some possible extensions of our implementation and relates some directions of future work. Section 6 summarizes the current work.

2 Views and Synchronization

Significant work on characterizing those types of views which may be meaningfully updated can be found within the database community. For example, Bancilhon and Spyratos [1] define the translation of view updates *under constant complement*. Assuming a view and its complement completely determine the underlying relation, a set of view updates has a translator if and only if that set of updates holds the complement constant. Gottlob *et al* [2] refine this work and

[1] A related problem, known as the *incremental views maintenance*, focuses on propagating changes in the underlying relation up to the view. While the problem has a similar name, it has little else in common with the problem considered here.

allow a larger set of *dynamic* views to be expressed. A dynamic view consists of a function (from concrete to abstract states) and an update translator defined in such a way that the update may be applied after the function, or translated and applied before the function.

In both cases, the authors have focused on translations of *update functions.* That is, for *any* fixed view, they have characterized which updates on that view can be unambiguously translated into updates on the underlying relation. In their work, Greenwald, Moore, Pierce and Schmitt [3] take a different approach. With respect to the current problem, they characterize those functions from relations to views for which there is a companion function from views back to relations. This gives rise to a set of views that may be unambiguously updated through arbitrary update functions. That is, they describe a translation of abstract to concrete *states* rather than a translation of update functions.

In the work of Greenwald *et al* [3], the Focal programming language allows updates to an abstract representation of some concrete structure to be reflected back in the concrete form. While the structures considered in this work are labeled trees, the ideas of this work can also be applied to views and relations.

2.1 Lenses

Expressions of the Focal programming language are known as *lenses.* Focal programs are build by combining primitive lenses with lens combinators. Each lens primitive (and therefore every Focal program) defines not one but *two* functions, known as *get* and *put.* The *get* function takes a concrete tree and produces an abstract one by "forgetting" parts of the structure. The *put* function, on the other hand, takes an abstract tree and a concrete tree. It produces a new concrete tree using values from the abstract tree whenever possible and values from the concrete tree to fill in the parts that were previously forgotten. The *put* function may be considered as a function from abstract states to updates on concrete trees and behaves independently of the original abstract tree.

Greenwald *et al* define three laws, called GetPut, PutGet, and PutPut, that constrain the lenses that we will admit to our programming language. Lenses that satisfy the first two are called *well-behaved*, and lenses that satisfy all three are called *very well-behaved.* All lenses of the Focal language must be at least well-behaved, so that, for example, every *put* function is idempotent (with respect to a given abstract tree).

Pierce and Schmitt [5] show a strong correspondence between the classes of update translators defined in previous work and lens satisfying the above laws. They show that the set of well-behaved lens (with respect to a set of concrete and abstract representations) is isomorphic to the set of *dynamic views* described by Gottlob *et al* and that the set of very well-behaved lenses is isomorphic to the set of view translators under constant complement (as defined by Bancilhon and Spyratos).

2.2 Trees

In order to apply this work to the view update problem, we restrict ourselves to relations in third normal form (each relation must have a key attribute). Relations may then be represented as labeled trees. We extend the proposal given by Greenwald *et al* and include multiple relations in a single tree in the obvious way. For example, consider the following two-relation database.

R :

K	A	B
k_1	a_1	b_1
k_2	a_2	b_2

S :

K	C	D
k_2	c_3	d_3
k_3	c_4	d_4

We then have the following tree, where each relation is a subtree labeled by its name and where each key labels a subtree containing labeled attribute values.

$$\left\{ \begin{array}{l} R \mapsto \left\{ \begin{array}{l} k_1 \mapsto \left\{ \begin{array}{l} A \mapsto a_1 \\ B \mapsto b_1 \end{array} \right. \\ k_2 \mapsto \left\{ \begin{array}{l} A \mapsto a_2 \\ B \mapsto b_2 \end{array} \right. \end{array} \right. \\ S \mapsto \left\{ \begin{array}{l} k_2 \mapsto \left\{ \begin{array}{l} C \mapsto c_3 \\ D \mapsto d_3 \end{array} \right. \\ k_3 \mapsto \left\{ \begin{array}{l} C \mapsto c_4 \\ D \mapsto d_4 \end{array} \right. \end{array} \right. \end{array} \right.$$

Below, we write $\mathsf{dom}(t)$ for the set of labels appearing at the top level of t (*i.e.* the labels of its immediate children) and $t(n)$ for the subtree of t associated with the label n. The *get* portion of a lens is denoted $\nearrow$, and the *put* portion as $\searrow$.

As a simple example, consider the definition of the `hoist` lens primitive [3].

$$\begin{array}{rcll} (\texttt{hoist}\ n)\nearrow(c) & = & t & \text{if } c = \{n \mapsto t \\ & & \text{undefined} & \text{otherwise} \\ (\texttt{hoist}\ n)\searrow(a,c) & = & \{n \mapsto a & \end{array}$$

Like all lenses, `hoist` defines a bidirectional transformation and, like many lenses, its name evokes its behavior in the *get* direction. In this case, `hoist` expects a tree with one child, labeled n. The resulting abstract tree is precisely this child. In the *put* direction, it yields a new tree with one child: the given abstract tree, labeled n.

3 Implementation

We have implemented a substantial portion of SQL using the Focal language. This has allowed us not only to implement view updates, but also simplify the implementation of traditional updates and deletions; the

$$T_{\text{query}}(\texttt{SELECT } \mathit{attrs} \texttt{ FROM } R \texttt{ WHERE } \mathit{pred}) \stackrel{\text{def}}{=}$$

$$\texttt{focus } R\texttt{; filter [}T_{\text{pred}}(\mathit{pred})\texttt{]} \ \{\}\texttt{;}$$
$$\texttt{map (filter [}\mathit{attrs}\texttt{]} \ \{\overline{\mathit{attrs}} \mapsto \texttt{NULL}\}\texttt{)}$$

Figure 1: Translation of single-relation SQL queries (The expression $\overline{attrs}$ stands for those attributes of R not found in the set *attrs*. The translation of query predicates is not shown.)

translations given below are used both for ordinary queries and the creation of new views. We give a description of the lenses used to implement SQL queries, including joins, followed by a discussion of our update mechanism.

3.1 Selection

A complete description of the translation of selection queries is beyond the scope of this document, in particular because of the many different predicates that may be found in `WHERE` clauses. Despite this, our implementation supports a wide range of arithmetic relations (*e.g.* $=, \leq$) and logical operators (*e.g.* $\wedge, \vee$) on attributes of a single relation. A description of the translation is given in Figure 1.

The following example shows that translation of a selection query.

$$\texttt{SELECT * FROM } R \texttt{ WHERE } A = a_1$$

is translated as

$$\texttt{focus } R\texttt{; filter [\_} \mapsto \{A \mapsto a_1\}\texttt{]} \ \{\}$$

A brief explanation of the above lens program follows. In the *get* direction, the `focus` lens forgets all subtrees that are *not* labeled R as well as the label R itself. The `filter` lens takes a lens predicate and a default tree (here the empty tree), and (in the *get* direction) forgets any subtree for which the predicate does not hold. The wild-card `_` as it appears in this predicate matches any label. In the *put* direction, `filter` uses the default tree where the corresponding part of the concrete tree is missing. Finally, `;` is the sequential composition of two lenses. For a complete definition and discussion of the lenses and lens combinators used in this paper, we refer the reader to the work of Greenwald *et al* [3].

It should be noted that we diverge from the notion of predicates given by Greenwald *et al.* In that work, predicates are restricted to sets of labels and are applied only to a tree's immediate children. Here, we allow predicates to apply to entire subtrees. While this is a significant change to the programming language, it is orthogonal to the definitions of lenses, and the presentation given by Greenwald *et al* requires only superficial changes.

3.2 Projection

In addition to selecting records from a relation, we may wish to isolate specific attributes. This example shows the translation of a simple projecting query.

$$\texttt{SELECT } A \texttt{ FROM } R$$

is translated as

$$\texttt{focus } R\texttt{; map (filter [}A\texttt{]} \ \{B \mapsto \texttt{NULL}\}\texttt{)}$$

(where R is a relation with two attributes, A and B) The `map` lens combinator takes a lens as a parameter and applies it to each child of the current tree. Here, we filter the attributes according to the specification given by the query. In addition, we must specify default values for each attribute that will *not* be displayed. If such a query is used to define a view and a new record is inserted into that view, then these default values are used for the hidden attributes.

Selection and projection queries may be combined simply by composing the respective `filter` and `map` lenses (as shown in Figure 1).

3.3 Join

The lens primitives given by Greenwald *et al* appear to be insufficient to allow joins of two or more relations: each such primitive treats every subtree independently. In order to perform a join, we must consider two subtrees simultaneously.

Figure 2 defines a new lens primitive implementing an outer join on primary keys.[2] Given a tree with two relations labeled R and S, the *get* direction considers each key found in either relation and yields a new subtree. Each such subtree is labeled with its key and has one child for each relation in which that key is found. In the *put* direction, each subtree is split into two parts and then combined with those parts of R and S that are not present in the join.

Consider the following example. Take a database containing two relations R and S. The action of the *get* direction of the `join` lens is shown below. (We elide the details of each relation, writing subtrees t_i in place of the labeled attribute sets.)

$$\left\{\begin{array}{l} R \mapsto \left\{\begin{array}{l} k_1 \mapsto t_1 \\ k_2 \mapsto t_2 \end{array}\right. \\ S \mapsto \left\{\begin{array}{l} k_2 \mapsto t_3 \\ k_3 \mapsto t_4 \end{array}\right. \end{array}\right. \quad \text{becomes} \quad \left\{\begin{array}{l} k_1 \mapsto \{\, R \mapsto t_1 \\ k_2 \mapsto \left\{\begin{array}{l} R \mapsto t_2 \\ S \mapsto t_3 \end{array}\right. \\ k_3 \mapsto \{\, S \mapsto t_4 \end{array}\right.$$

To perform an ordinary (inner) join, we must subsequently filter out any keys that have only one child (these correspond to keys that appear only in R or S but not both) and then remove references to R and S, renaming the attributes to prevent possible clashes between those in R and those in S.

[2] For a discussion of other types of joins, see section 4.2.

$$
\begin{array}{lcll}
(\texttt{join}\ R\ S)\nearrow(c) & = & \left\{n \mapsto \begin{cases} R \mapsto (c(R))(n) & \text{if } (c(R))(n) \text{ is defined} \\ S \mapsto (c(S))(n) & \text{if } (c(R))(n) \text{ is defined} \end{cases}\right. & \text{if } \mathsf{dom}(c) = \{R, S\} \\
 & & \text{undefined} & \text{otherwise} \\
(\texttt{join}\ R\ S)\searrow(a, c) & = & \begin{cases} R \mapsto \{n \mapsto (a(n))(R) & \text{if } (a(n))(R) \text{ is defined} \\ S \mapsto \{n \mapsto (a(n))(S) & \text{if } (a(n))(S) \text{ is defined} \end{cases} &
\end{array}
$$

Figure 2: The `join` lens primitive

$$\texttt{SELECT * FROM } R, S \texttt{ WHERE } R.K = S.K$$

is translated as

$$
\begin{array}{r}
\texttt{filter [}R \wedge S\texttt{] \{\};} \\
\texttt{join } R\ S\texttt{; filter [\_} \mapsto \texttt{\_} + \texttt{\_] \{\};} \\
\texttt{map (xfork [}R\texttt{] [}R.\texttt{\_]} \\
\texttt{(hoist } R\texttt{; rename [\_} = R.\texttt{\_])} \\
\texttt{(hoist } S\texttt{; rename [\_} = S.\texttt{\_]))}
\end{array}
$$

We use $+$ to denote at tree with two or more children. The `xfork` lens combinator splits a tree according to the first predicate and applies a lens to each half. (The second predicate is used in the *put* direction.) We use the `hoist` lens to (in the *get* direction) lift the child of a tree with a single label and `rename` to rename its attributes. Joins may be combined with other queries, again, by composing the respective lenses.

In addition to primary keys, the `join` lens may also be used to define joins on any uniquely valued attribute. It also can be extended to joins of three or more relations in straightforward manner.

3.4 Views

The above translation yields a lens for any restricted SQL query, thus a view, as represented in our implementation, is simply a cached copy of its defining lens. When a query is executed against a view, the lens for that query is simply composed with the lens for the view.

3.5 Updates

Recall that a lens defines two functions: a *get* function from concrete trees to abstract trees and a *put* function taking an abstract tree to an update function on concrete trees.

As an example, consider the following SQL command,

$$\texttt{DELETE FROM } R \texttt{ WHERE } A = a_1$$

The translation of an analogous selection query yields the following lens.

$$\texttt{focus } R\texttt{; filter [\_} \mapsto \{A \mapsto a_1\}\texttt{] \{\}}$$

When considering the above `DELETE` command, we note that the abstract tree resulting from this "update" is trivial: it is simply the empty tree. In other words, if we consider a view of records where $A = a_1$, the effect of this command is to remove all records from that view.

To execute this `DELETE` on a given concrete tree c, we apply the following *put* function,

$$(\texttt{focus } R\texttt{; filter [\_} \mapsto \{A \mapsto a_1\}\texttt{] \{\})}\searrow(\{\}, c)$$

The implementation of updates is similar: we define a lens to create an appropriate abstract tree, apply the changes to the *entire* abstract tree, and then use the lens to synchronize our changes with the underlying relation. It follows that we can apply updates using any query or to any view so long as such a query or view is defined by a lens. In addition, we can define new lenses and immediately apply our existing update mechanisms.

4 Discussion

Intuitively, the `join` lens primitive allows us to make unambiguous updates to a view because it preserves the structure of the underlying relation. In this section, we characterize this aspect of `join` more precisely and consider some other tree transformations which do not share these properties.

4.1 Properties of `join`

In order to be confident that programs containing the `join` lens are well-formed, we must show that our new lens primitive is well-behaved. Greenwald *et al* set out the characteristics defining very well-behaved lenses. In short, such a lens must obey three laws constraining the nature of *get* and *put*: where they are defined, the composition of *get* and *put* must be equal to the identity transformation (similarly for *put* and *get*), and two subsequent applications of *put* with trees a and a' must act just as a single application of *put* with the second tree a'.

Proposition 1. *`join` is a very well-behaved lens.*

Proof.

[PutGet] Assume the $(\texttt{join}\ R\ S)\nearrow(c)$ is defined, *i.e.* $\mathsf{dom}(c) = \{R, S\}$. Then,

$$(\texttt{join}\ R\ S)\searrow((\texttt{join}\ R\ S)\nearrow(c)\,,c)$$

$$= (\texttt{join}\ R\ S)\searrow\left(\left\{n\mapsto\begin{cases}R\mapsto(c(R))(n)\\S\mapsto(c(S))(n)\end{cases},c\right)\right.$$

where ever $(c(R))(n)$ and $(c(S))(n)$ are defined

$$= \begin{cases}R\mapsto\{n\mapsto(c(R))(n) & (c(R))(n)\text{ defined}\\S\mapsto\{n\mapsto(c(S))(n) & (c(S))(n)\text{ defined}\end{cases}$$

$$= c$$

[GetPut] Since $(\texttt{join}\ R\ S)\searrow(a,c)$ is always defined, we have,

$$(\texttt{join}\ R\ S)\nearrow(\texttt{join}\ R\ S)\searrow(a,c)$$

$$= (\texttt{join}\ R\ S)\nearrow\left(\begin{cases}R\mapsto\{n\mapsto(a(n))(R)\\S\mapsto\{n\mapsto(a(n))(S)\end{cases}\right)$$

where ever $(a(n))(R)$ and $(a(n))(S)$ are defined

$$= \left\{n\mapsto\begin{cases}R\mapsto(a(n))(R) & (a(n))(R)\text{ defined}\\S\mapsto(a(n))(S) & (a(n))(S)\text{ defined}\end{cases}\right.$$

$$= a$$

[PutPut]

$$(\texttt{join}\ R\ S)\searrow(a',(\texttt{join}\ R\ S)\searrow(a,c))$$

$$= (\texttt{join}\ R\ S)\searrow\left(a',\begin{cases}R\mapsto\{n\mapsto(a(n))(R)\\S\mapsto\{n\mapsto(a(n))(S)\end{cases}\right)$$

$$= \begin{cases}R\mapsto\{n\mapsto(a'(n))(R)\\S\mapsto\{n\mapsto(a'(n))(S)\end{cases}$$

$$= (\texttt{join}\ R\ S)\searrow(a',c)$$

□

4.2 Ambiguous Joins

In designing a lens primitive to support relational views, we considered a number of different tree transformations. In defining the translation of a join clause, we considered a transformation where the inner relation is replicated at each of leaf of the outer relation. Though such a "replicating" lens would already be suspect, one could imagine how subsequent filtering might produce the desired result.

Such a transformation might be thought of as an implementation of a nested-loops join: each record of the inner relation is considered once for each record of the outer relation. In contrast, the `join` lens may be considered as a form of restricted sort-merge join, where each record is considered exactly once during the merge. This restriction is precisely what makes the outer join of a uniquely valued attribute unambiguous with respect to updates: each record of the result corresponds to a unique record in each of the underlying relations.

If two records in the result were to correspond to a single record in one of the relations, and different changes were made to these two records, then ambiguity would arise in determining which change to apply to the relation.

If we consider an equijoin of an attribute with possibly duplicated values, a similar problem may arise. Finally, a join defined using an inequality relation (*e.g.* $\leq$, $>$) may also result in multiple records in the result corresponding to a single record of an underlying relation, so updates to such a join would also be ambiguous.

These ambiguities arise from the limitations of the SQL language (or perhaps from our attempts to apply it beyond it stated purpose as a general-purpose query language). Rather than define a translation from existing SQL syntax, one might consider extending the current syntax or building application-specific lenses. Doing so would allow more expressive joins by explicitly disambiguating the results of an update. Indeed, as noted above, the work of Pierce and Schmitt [5] states that if the set of updates is unambiguous, then there is a lens describing such a view. Exploring these possibilities is beyond the scope of the current work.

5 Extensions and Future Work

Our implementation has focused on the semantics of updates rather than query optimization or even persistence. These issues are largely orthogonal to the semantics of view updates. While we noted above that the `join` lens shares some resemblance with a restricted form of sort-merge join, a number of different techniques might be used to implement our tree representation. In fact, one might see labeled trees simply as an abstract data type with a variety of implementations providing sorting, indexing, caching, and persistence. Furthermore, different implementations may be applied at different levels in our database tree. Furthermore, just as SQL queries may be compiled into native machine code, lens programs may be compiled into (pairs of) native procedures.

Because our implementation represents relations as trees of labeled but *unordered* branches, records cannot be sorted as trees. However, the ordering of records is not normally considered a mutable characteristic of a relation, so we might implement sorting as part of the presentation of the result set, after any lenses have been applied.

Computed attributes, such as those determined by aggregate, arithmetic, and logical functions, are not supported in our implementation. These attributes are not well-suited to updates since they do not (in general) have well-defined inverses. For example, if the sum of two integers is updated, there are many possible values that two components might be given. If considered to be read-only, these computed attributes might easily be supported through a post-processing step, as noted for record sorting above.

As we have previously discussed, if we limit ourselves to views defined by SQL queries, only equijoins of uniquely valued attributes can be unambiguously updated. However, just as aggregate functions, *etc.* are supported in a read-only fashion, we might implement other kinds of joins as a post-process. Any attribute that is not defined by a lens operation would be read-only.

6 Conclusion

We have described an implementation of a relational database using trees as the primary data structure and bidirectional transformations as the primary operators. Our implementation supports a limited form of join based on a new, well-behaved transformation. Though we have argued that less restrictive classes of joins result in ambiguous update functions, our implementation might be extended to support them with read-only attributes. Alternatively, language extensions or application-specific lenses could allow unambiguous updates, unlike a general-purpose translation from SQL.

References

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

[2] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Trans. Database Syst.*, 13(4):486–524, 1988.

[3] M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. Technical Report MS-CIS-03-08, University of Pennsylvania, August 2003.

[4] 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. ACM Press, 1985.

[5] B. C. Pierce and A. Schmitt. Lenses and view update translation (draft). http://www.cis.upenn.edu/ bcpierce/papers/dblenses.pdf, 2003.