Eventually-Consistent Replicated Relations and Updatable Views

Joachim Thomassen and Weihai Yu[0000−0002−8886−9047]

UIT - The Arctic University of Norway, Tromsø, Norway
weihai.yu@uit.no

Abstract. Distributed systems have to live with weak consistency, such as eventual consistency, if high availability is the primary goal and network partitioning is unexceptional. Local-first applications are examples of such systems. There is currently work on local-first databases where the data are asynchronously replicated on multiple devices and the replicas can be locally updated even when the devices are offline. Sometimes, a user may want to maintain locally a copy of a view instead of the entire database. For the view to be fully useful, the user should be able to both query and update the local copy of the view. We present an approach to maintaining updatable views where both the source database and the views are asynchronously replicated. The approach is based on CRDTs (Conflict-free Replicated Data Types) and guarantees eventual consistency.

Keywords: Data replication, eventual consistency, updatable views, lenses, CRDT

1 Introduction

Local-first software suggests a set of principles for software that enables both collaboration and ownership for users. Local-first ideals include the ability to both work offline and collaborate across multiple devices [11].

CRDT [13], or Conflict-free Replicated Data Type, has been a popular approach to constructing eventually-consistent local-first systems. With CRDT, a site updates its local replica without coordination with other sites. The states of replicas converge when they have applied the same set of updates. CRR [14], or Conflict-free Replicated Relation, is an application of CRDT to relational databases.

A user may want to maintain copies of views in her local devices, instead of the entire database. This may reduce both the amount of data stored on the device as well as the overhead of data communication and local data processing. For a view to be fully useful, the user should be able to both query and update the local copies of the views. The updates must then be translated and applied back to the original source database.

Supporting updatable views has been an active research topic for decades [2, 2–4, 8, 10]. A view V is a sequence of query operations $\mathscr{Q}_v = [q_1, q_2, \dots]$ over a source database s, i.e. $v = \mathscr{Q}_v(s)$. After an update u_v on view v, the new state of the view becomes $v' = u_v(v)$. A translation $T_\uparrow$ of u_v to source database s results in a sequence of updates in base relations $T_\uparrow(s, u_v) = [u_1, u_2, \dots]$. According to [4], $T_\uparrow(s, u_v)$ *exactly*

translates $u_v(v)$ iff $\mathscr{Q}_v(T_\uparrow(s,u_v)(s)) = u_v(\mathscr{Q}_v(s))$ and the integrity constraints defined in the source database are preserved.

There is at present no existing work on supporting updatable views in a distributed setting where both the source database and the views are asynchronously replicated and local data can be updated even when the replicas are offline. One challenge is that the translation $T_\uparrow(s,u_v)$ is dependent on the state s of the source database. The same view update u_v may have different translations at replicas in different concurrent states.

We present an approach based on delta-state CRDT [1, 5] (Section 2.1), applied to replication of relational databases [14] (Section 2.2). The states of the source database and the views form a join-semilattice [6, 13]. The updates in the database and the views are represented as join-irreducible states in the join-semilattice. A view update is translated into a set of join-irreducible states in the source database and the translation is independent of the current state of the database. When the database (and equally view) replicas have applied the same set of updates (i.e. join-irreducible states), their states converge.

The paper is organized as the following. In Section 2, we briefly review the necessary background of CRDT and CRR. In Section 3, we give a high-level overview of our approach. In Section 4, we use examples to describe how we translate view updates to the source database. In Section 5, we discuss related work. Finally, in Section 6, we conclude.

2 Technical background

In this section, we briefly review necessary technical background on CRDT and CRR.

2.1 CRDT

A CRDT [13] is a data abstraction specifically designed for data replicated at different sites. A site queries and updates its local replica without coordination with other sites. The data are always available for update, but the data states at different sites may diverge. From time to time, the sites send their updates asynchronously to other sites with an anti-entropy protocol. The sites also merge the received updates with their local data. A CRDT guarantees *strong eventual consistency* [13]: a site merges incoming remote updates without coordination with other sites; when all sites have applied the same set of updates, their states converge.

We adopt delta-state CRDTs [1, 5]. The possible states must form a join-semilattice [6], which implies convergence. Briefly, the states form a *join-semilattice* if they are partially ordered with $\sqsubseteq$ and a join $\sqcup$[1] of any two states (that gives the least upper bound of the two states) always exists. State updates must be inflationary. That is, the new state supersedes the old one in $\sqsubseteq$. The merge of two states s_1 and s_2 is the result of $s_1 \sqcup s_2$. With delta-state CRDTs, it is sufficient to only send and merge join-irreducible states. Basically, *join-irreducible* states are elementary states: every state in the join-semilattice can be represented as a join of some join-irreducible state(s).

[1] To avoid being confused with the join $\bowtie$ of relations, in the rest of the paper, we use the term *merge* for $\sqcup$.

$$\mathsf{CLSet}(E) \stackrel{\text{def}}{=} E \hookrightarrow \mathbb{N}$$

$$\mathsf{insert}^{\delta}(s,e) \stackrel{\text{def}}{=} \begin{cases} \{e \mapsto s(e)+1\} & \text{if } \neg\mathsf{in?}\big(s(e)\big) \\ \{\} & \text{otherwise} \end{cases}$$

$$\mathsf{delete}^{\delta}(s,e) \stackrel{\text{def}}{=} \begin{cases} \{e \mapsto s(e)+1\} & \text{if } \mathsf{in?}\big(s(e)\big) \\ \{\} & \text{otherwise} \end{cases}$$

$$(s \sqcup s')(e) \stackrel{\text{def}}{=} \mathsf{max}\big(s(e), s'(e)\big)$$

$$\mathsf{in?}(s,e) \stackrel{\text{def}}{=} \mathsf{odd?}\big(s(e)\big)$$

Fig. 1. CLSet CRDT [14]

Since a relation instance is a set of tuples, the basic building block of CRR is a general-purpose delta-state set CRDT ("general-purpose" in the sense that it allows both insertion and deletion of elements). We use CLSet (causal-length set, [14, 15]), a general-purpose set CRDT, where each element is associated with a *causal length*. Intuitively, insertion and deletion are inverse operations of one another. They always occur in turn. When an element is first inserted into a set, its causal length is 1. When the element is deleted, its causal length becomes 2. Thereby the causal length of an element increments on each update that reverses the effect of a previous one.

As shown in Fig. 1, the states of a CLSet are a partial function $s\colon E \hookrightarrow \mathbb{N}$, meaning that when e is not in the domain of s, $s(e) = 0$. Using partial function conveniently simplifies the specification of insert, $\sqcup$ and $\mathsf{in?}$. Without explicit initialization, the causal length of any unknown element is 0. insert^{δ} and delete^{δ} in Fig. 1 are delta-mutators that returns a join-irreducible state instead of the entire state.

An element e is regarded as being in the set when its causal length is an odd number. A local insertion has effect only when the element is not in the set. Similarly, a local deletion has effect only when the element is actually in the set. A local effective insertion or deletion simply increments the causal length of the element by one. For every element e in s and/or s', the new causal length of e, after merging s and s', is the maximum of the causal lengths of e in s and s'.

2.2 CRR

The relational database supporting CRR consists of two layers: an Application Relation (AR) layer and a Conflict-free Replicated Relation (CRR) layer (Fig. 2). The AR layer presents the same database schema and API as a conventional relational database. Application programs interact with the database at the AR layer. The CRR layer supports conflict-free replication of relations.

An AR-layer relation schema R has an augmented CRR-layer schema $\tilde{R}$. In Fig. 2, site A maintains both an instance r_A of R and an instance $\tilde{r}_A$ of $\tilde{R}$. A query q is performed on r_A without any involvement of $\tilde{r}_A$. An update operation u on r_A triggers an additional operation $\tilde{u}$ on $\tilde{r}_A$. The operation $\tilde{u}$ is later propagated to remote sites through an anti-entropy protocol. Merge with an incoming remote operation $\tilde{u}'(\tilde{r}_B)$ results in an operation $\tilde{u}'$ on $\tilde{r}_A$ as well as an operation u' on r_A.

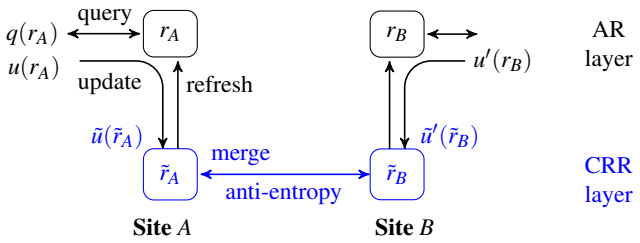


Fig. 2. A two-layer relational database system [14]

CRR has the property that when both sites A and B have applied the same set of operations, the relation instances at the two sites are equivalent, i.e. $r_A = r_B$ and $\tilde{r}_A = \tilde{r}_B$.

CRR adopts several CRDTs. Since a relation instance is a set of tuples, we use the CLSet CRDT (Fig. 1) for relation instances. We use the LWW (last-write wins) register CRDT [9, 12] for individual attributes in tuples.

The join-irreducible states in a CRR relation $\tilde{r}$ are simply the tuples as the result of the insertions, deletions and updates. In the rest of the paper, we use the term *delta* for the tuple as the join-irreducible state of an operation. As we apply delta-state CRDTs, the tuples of the latest changes are sent to remote sites in the anti-entropy protocol.

For an AR-layer relation $R(K, A_1, A_2, \dots)$, where K is the primary key, there is a CRR-layer relation $\tilde{R}(\tilde{K}, K, L, T_1, T_2, \dots, A_1, A_2, \dots)$. $\tilde{K}$ is the primary key of $\tilde{R}$ and its values are globally unique. L is the causal-lengths (Fig. 1) of the tuples in $\tilde{R}$. T_i is the timestamp of the last update on attribute A_i. In other words, the $(\tilde{K}, L)$ part represents the CLSet CRDT of tuples and the (A_i, T_i) parts represent the LWW register CRDT of the attributes.

When inserting a new tuple t into r, we insert a new tuple $\tilde{t}$ into $\tilde{r}$, with the initial $\tilde{t}(L) = 1$. When deleting t from r, we increment $\tilde{t}(L)$ with 1. Tuple t is in r, $t \in r$, if $\tilde{t}(L)$ is an odd number. That is,

$$\mathsf{in\_ar?}(\tilde{t}) \stackrel{\text{def}}{=} \mathsf{odd?}(\tilde{t}(L))$$

When updating $t(A_i)$ in r, we update $\tilde{t}(A_i)$ and $\tilde{t}(T_i)$ in $\tilde{r}$.

An update delta on an relation instance $\tilde{r}'$ at a remote site is actually a tuple $\tilde{t}'$. If a tuple $\tilde{t}$ in the local instance $\tilde{r}$ exists such that $\tilde{t}(\tilde{K}) = \tilde{t}'(\tilde{K})$, we update $\tilde{t}$ with $\tilde{t} \sqcup \tilde{t}'$ where the merge $\sqcup$ is the join operation of the join-semilattice (Section 2.1). Otherwise, we insert $\tilde{t}'$ into $\tilde{r}$. The merge $\tilde{t} \sqcup \tilde{t}'$ is defined as:

$$\tilde{t} \sqcup \tilde{t}' \stackrel{\text{def}}{=} \tilde{t}'', \text{ where } \tilde{t}''(L) = \max(\tilde{t}(L), \tilde{t}'(L)), \text{ and}$$
$$\tilde{t}''(A_i), \tilde{t}''(T_i) = \begin{cases} \tilde{t}'(A_i), \tilde{t}'(T_i) & \text{if } \tilde{t}'(T_i) > \tilde{t}(T_i) \\ \tilde{t}(A_i), \tilde{t}(T_i) & \text{otherwise} \end{cases}$$

After the update of $\tilde{r}$, we update r as the following. If $\mathsf{in\_ar?}(\tilde{t})$ evaluates to false, we delete t (where $t(K) = \tilde{t}(K)$) from r. Otherwise, we insert or update r with $\pi_{K, A_1, A_2, \dots}(\tilde{t})$.

3 Approach Overview

We consider distributed database systems where data are replicated at multiple sites. For the purpose of, say, high availability, the sites may update the data without coordination with other sites. The system is said to be *eventually consistent*, or convergent, if, when all sites have applied the same set of updates, the sites have the same state. The system is said to be *strongly eventually consistent* [13], if the sites unilaterally resolve any possible conflict, i.e., without coordination with other sites. We focus on strongly eventually-consistent relational database systems.

We restrict on which views can be updated, similar to [3, 8, 10]. More specifically, a view can only project away non-primary-key attributes that are given default values or can remain unspecified with NULL when inserted without given value. Moreover, when joining two relations, the join attribute(s) must contain one of the primary keys.

For a source database schema S, we define a view V with $V = \mathscr{Q}_v(S)$. Suppose when the database state is initially s_0, the view state is $v_0 = \mathscr{Q}_v(s_0)$ (Fig. 3). Concurrently, the view applies updates with delta state $\Delta v'$ and the source database applies updates with delta state $\Delta s'$. The new states in the view and the database become $v_1 = v_0 \sqcup \Delta v'$ and $s_1 = s_0 \sqcup \Delta s'$ respectively. When the database receives $\Delta v'$, it applies the translated delta $T_\uparrow(\Delta v')$ to s_1 and the new state becomes $s_2 = s_1 \sqcup T_\uparrow(\Delta v')$. Similarly, when the view receives $\Delta s'$, it applies the translated delta $T_\downarrow(\Delta s')$ to v_1 and the new state become $v_2 = v_1 \sqcup T_\downarrow(\Delta s')$. One important property of the translations $T_\downarrow$ and $T_\uparrow$ is that they are independent of the target state in which the translation results are going to be applied.

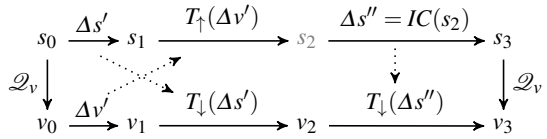


Fig. 3. Delta-states in source database and view

Unlike traditional work on updatable views, we do not restrict to side-effect-free view updates. However, we do respect integrity constraints, including the ones defined by application programs, for instance, functional dependencies enforced with triggers.

In Fig. 3, if the state s_2 violates an integrity constraint, s_2 is never visible to the application. Instead, the view immediately applies some additional delta (as side effect of $T_\uparrow(\Delta v')$), $\Delta s'' = IC(s_2)$ for integrity-constraint preservation, and the new state s_3 does not violate any integrity constraint. Finally, the view applies the translation of $\Delta s''$. Our approach guarantees that the updates in Fig. 3 commute. That is,

$$\mathscr{Q}_v(s_0 \sqcup \Delta s' \sqcup T_\uparrow(\Delta v') \sqcup \Delta s'') = \mathscr{Q}_v(s_0) \sqcup \Delta v' \sqcup T_\downarrow(\Delta s') \sqcup T_\downarrow(\Delta s'')$$

Since the merge operation $\sqcup$ is commutative, when the different replicas of the source database (or the view) have applied the same set of delta states, their final states converge.

We have implemented a prototype of CRR and updatable views with SQLite. We do not include the implementation and experiments in this paper due to space limit.

R_a

album	quantity
Disintegration	6
Show	3
Galore	1
Paris	4
Wish	5

R_t

track	year	instore
Lullaby	1989	TRUE
Lovesong	1989	TRUE
Trust	1992	FALSE

R_{ta}

track	**album**
Lullaby	Galore
Lullaby	Show
Lovesong	Galore
Lovesong	Paris
Trust	Wish

Fig. 4. Example database

4 Translation of view-update delta states

The translation from source database to views, $T_{\downarrow}$, is traditionally know as incremental maintenance of materialized views. In this section, we focus on $T_{\uparrow}$, the translation of view-update delta states to the source database. We describe the translation through examples. The example database (Fig. 4) is adapted from [3, 8].

We start with select and project views. In Fig. 5, the base relation R_t (top left) is first augmented to a CRR-layer relation $\tilde{R}_t$ (top right). $\tilde{R}_t$ has an attribute L for the causal lengths of the tuples. In addition, every non-primary-key attribute is associated with a timestamp attribute, indicating the last time at which the attribute value was set.

A project view has the same causal-length and timestamp attributes as the base relation, unless the attribute is projected away. A select view has two more attributes σ and T_σ that tell the last time the select predicate was evaluated. Initially, all σ values are TRUE and the timestamp value T_σ of a tuple is the maximum of the timestamp values of the attributes that occur in the select predicate. For tuples in CRR-layer $\tilde{v}_1$ in Fig. 5(a), the T_σ values are set to the T_y values of $\tilde{r}_t$. If later the year-attribute of a tuple is set to a value greater than or equal to 1990, the σ value becomes FALSE and the corresponding tuple disappears from the AR-layer view.

The delta state of an update is simply a tuple in a CRR-layer relation or view. For update $v_1\langle\text{Lullaby}, 1989 \nearrow 1988\rangle$ in Fig. 5(a), the delta state is $\tilde{v}'_1\langle\text{Lullaby}, 1988, 5.1, 1, \text{TRUE}, 5.1\rangle$. Here, $T_y = 5.1$ is the timestamp at which the new year-value is set. Since the year-attribute is used in the select predicate, T_σ is also set to 5.1.

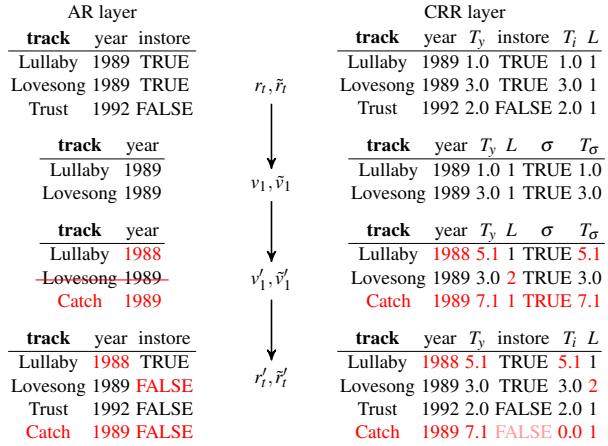
For deletion $-v_1\langle\text{Lovesong}, 1989\rangle$, the L attribute of the delta state is incremented with 1. As it is an even number, the tuple is regarded as being deleted in the AR layer.

For insertion $+v_1\langle\text{Catch}, 1989\rangle$, the initial L value is 1 and all timestamps are set according to the current time. For all insertions in select views, the T_σ value must be TRUE.

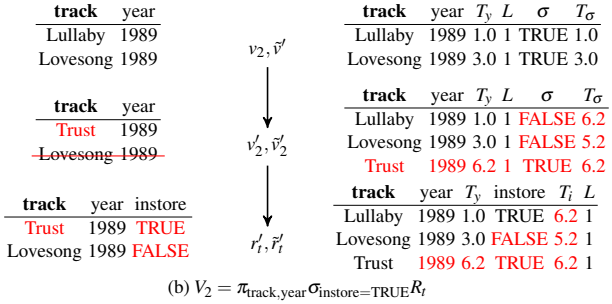
Recall that a project view is updatable only if it keeps the primary key of the base relation. Moreover, CRR-layer base and view relations keep all tuples regardless of whether they have been deleted or not. Therefore, for every tuple in a CRR-layer select-and-project view, there is exactly one tuple in the CRR-layer base relation.

Delta states of a view can be translated almost directly to the base relation. The only exception is for the attributes that are projected away. The instore-attribute of the Catch-tuple, which is missing in view V_1, is set to its default value (suppose it is FALSE). Its timestamp value T_i is set to 0.0, the smallest possible timestamp value. This means that a default value (or NULL) cannot override any value that is explicitly given.

Fig. 5(b) shows two additional cases. The first case shows that a deletion in some views can be handled differently. Here, we have an opportunity to achieve a least-effect translation of deletions in a view, when the select predicate includes a boolean attribute,



(a) $V_1 = \pi_{\text{track,year}} \sigma_{\text{year}<1990} R_t$



(b) $V_2 = \pi_{\text{track,year}} \sigma_{\text{instore=TRUE}} R_t$

Fig. 5. Updating select and project views

such as the instore-attribute in $\sigma_{\text{instore=TRUE}}$. Now, for the deletion $-v_2\langle\text{Lovesong}, 1989\rangle$, instead of deleting the Lovesong-tuple in the base relation (i.e. by incrementing the L value), we set the σ value to FALSE. When translating to the base relation, we set the boolean value of the attribute as the negation in the select predicate. That is, $T_\downarrow(\tilde{v}_2\langle\text{Lovesong}, 1989, 3.0, 1, \text{FALSE}, 5.2\rangle) = [\tilde{r}_t\langle\text{Lovesong}, 1989, 3.0, \text{FALSE}, 5.2\rangle]$.

In this particular example, setting the Lovesong-track to be not-in-store is less destructive than deleting the track. When a select predicate uses multiple boolean attributes, we choose to update the truth value of the leftmost one in the view definition.

The next case that Fig. 5(b) shows actually applies generally to updates in both view and base relations. An update of (part of) a primary-key value is regarded as a deletion and an insertion. In the figure, the update $v_2\langle\text{Lullaby} \nearrow \text{Trust}, 1989\rangle$ is interpreted as $[-v_2\langle\text{Lullaby}, 1989\rangle, +v_2\langle\text{Trust}, 1989\rangle]$.

For a view of two-way join $R_1 \bowtie R_2$ to be updatable, we require, as in [10], that the join attributes contain a primary key of R_1 or R_2. We can make a graph from a view of a multi-way join. The nodes are the base relations. If the join attributes of $R_i \bowtie R_j$ contains the primary key of R_j, there is a link from R_i to R_j in the graph. Currently, we require, also as [10], that the view graph is a tree. The primary key of the view is the primary key of the root relation of the tree. Since the primary keys of the base relations

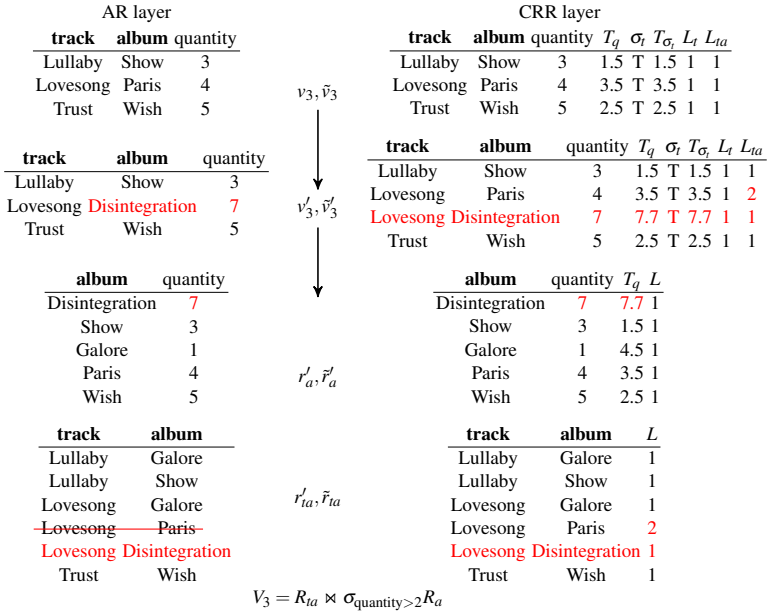


Fig. 6. Updating a join view

are not projected away, for a tuple t_v in the view, we can find the tuples in the base relations that contribute to t_v via their primary-key values.

For view $V = R_1 \ltimes R_2$, the set of attributes of the CRR layer $\tilde{V}$ is the union of the sets of attributes of the CRR-layer $\tilde{R}_1$ and $\tilde{R}_2$. For tuple $\tilde{t}_v$ in CRR-layer $\tilde{v}$, tuple t_v is in AR-layer v, if the L values of both $\tilde{r}_1$ and $\tilde{r}_2$ are odd and the σ values of both $\tilde{r}_1$ and $\tilde{r}_2$ are TRUE, i.e.,

$$\mathsf{in\_ar?}(\tilde{t}_v) \stackrel{\text{def}}{=} \mathsf{odd?}(\tilde{t}_v(L_{r_1})) \wedge \mathsf{odd?}(\tilde{t}_v(L_{r_2})) \wedge \tilde{t}_v(\sigma_{r_1}) \wedge \tilde{t}_v(\sigma_{r_2})$$

In Fig. 6, there is only one update in the view, $v_3\langle \text{Lovesong}, \text{Paris} \nearrow \text{Disintegration}, 4 \nearrow 7\rangle$. Since the album-attribute is part of the primary key of the view, the update is interpreted as a deletion $-v_3\langle \text{Lovesong}, \text{Paris}, 4\rangle$ and an insertion $+v_3\langle \text{Lovesong}, \text{Disintegration}, 7\rangle$.

For the deletion, we delete the corresponding tuple in the root base relation. Hence the tuple $\langle \text{Lovesong}, \text{Paris}\rangle$ is deleted from r_{ta}.

For the insertion, we first insert $\langle \text{Lovesong}, \text{Disintegration}\rangle$ into the root relation r_{ta}. Then, since there is already a Disintegration-tuple in r_a, we set the quantity-attribute to the new value 7.

5 Related work

[2] and [7] study the consistency of updatable views via mapping of states between source databases and views, where a source database is modeled as the product of the view and a complementary. When a chosen complementary is kept constant (side-effect

free) [2] or "shrinking" (under a partial order) [7], there is an unambiguous translation of a view update to the source database. [2] did not aim for computational algorithms that translate view updates to source databases.

To translate the updates from a view to a source database, [10] directly associates tuples and attributes in view relations with base relations in the source database. [4] makes the translation based on the tractability and functional dependency of attributes via view dependency graphs. [7] translates view programs (sequence of updates equipped with if-then-else statements) to base programs. [3] and [8] make bi-directional translation of every query operation (known as a lens) that defines the views. In most of the work on updatable views, translation of view updates is based on the attribute values. For example, since the view dependency graphs in [4] are defined on attributes, deletions are defined with predicates on attributes, for instance, "delete from V where $A = 7$". The source tuples can then be identified with queries on attributes with similar predicates. This may work well in a non-distributed system. In a distributed system where the source database and the view can be replicated, different replicas in different states may make different translations.

Our work is different from the previous work in that we use delta states (i.e. join-irreducible states in a join semilattice) to represent state updates. The translation is independent of the state to which the update is to be applied.

Regarding the restrictions on views that are updatable, [10] is the closest to our work, which are probably the most restrictive. There are at least two reasons for these restrictions. The first one is practical. Most related work assumes that all information about integrity constraints is available when a view is created, which is practically not true. In particular, the only functional dependencies that can be expressed in SQL is primary-key constraints. The second reason is that we are currently not able to express aggregate results (such as COUNT and MAX) as join-irreducible states.

In their seminal work [4], Dayal and Bernstein pointed out that a view update can be correctly (exactly) translated to the source relations if and only if there is a *clean source* of the update. It is possible to verify if a source is clean with the use of view dependency graphs. With the restrictions of the view that can be updated (Section 3), we guarantee that every update in a view has a clean source.

Unlike previous work, we allow translations of view updates to have side effects (Fig. 3). Avoiding side effect is probably more important in earlier work, which expects virtual (i.e. non-materialized) views. In fact, avoiding side effect is impossible without knowing all integrity constraints, such as the functional dependencies embedded in the view dependency graphs [4]. Notice that concurrent updates at different replicas may temporarily violate integrity constraints (like uniqueness and referential constraints) anyway [14]. We detect violations and repair constraints at the time of merge [14].

6 Conclusion

We presented an approach to asynchronously replicating both source databases and views. The local replicas of the database and the view can be updated even when they are offline. The approach guarantees eventual consistency. That is, the view updates are

correctly translated to the source database, and when the replicas have applied the same set of updates, their states converge.

References

1. ALMEIDA, P. S., SHOKER, A., AND BAQUERO, C. Delta state replicated data types. *J. Parallel Distrib. Comput. 111* (2018), 162–173.
2. BANCILHON, F., AND SPYRATOS, N. Update semantics of relational views. *ACM Trans. Database Syst. 6*, 4 (1981), 557–575.
3. BOHANNON, A., PIERCE, B. C., AND VAUGHAN, J. A. Relational lenses: a language for updatable views. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)* (2006), S. Vansummeren, Ed., ACM, pp. 338–347.
4. DAYAL, U., AND BERNSTEIN, P. A. On the correct translation of update operations on relational views. *ACM Trans. Database Syst. 7*, 3 (1982), 381–416.
5. ENES, V., ALMEIDA, P. S., BAQUERO, C., AND LEITÃO, J. Efficient Synchronization of State-Based CRDTs. In *IEEE 35th International Conference on Data Engineering (ICDE)* (April 2019).
6. GARG, V. K. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.
7. GOTTLOB, G., PAOLINI, P., AND ZICARI, R. V. Properties and update semantics of consistent views. *ACM Trans. Database Syst. 13*, 4 (1988), 486–524.
8. HORN, R., PERERA, R., AND CHENEY, J. Incremental relational lenses. *Proc. ACM Program. Lang. 2*, ICFP (2018), 74:1–74:30.
9. JOHNSON, P., AND THOMAS, R. The maintamance of duplicated databases. *Internet Request for Comments RFC 677* (January 1976).
10. KELLER, A. M. 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, March 25-27, 1985, Portland, Oregon, USA* (1985), ACM, pp. 154–163.
11. KLEPPMANN, M., WIGGINS, A., VAN HARDENBERG, P., AND MCGRANAGHAN, M. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, (Onward! 2019)* (2019), pp. 154–178.
12. SHAPIRO, M., PREGUIÇA, N. M., BAQUERO, C., AND ZAWIRSKI, M. A comprehensive study of convergent and commutative replicated data types. *Rapport de recherche 7506* (January 2011).
13. SHAPIRO, M., PREGUIÇA, N. M., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, (SSS 2011)* (2011), pp. 386–400.
14. YU, W., AND IGNAT, C.-L. Conflict-free replicated relations for multi-synchronous database management at edge. In *IEEE International Conference on Smart Data Services (SMDS)* (October 2020), pp. 113–121.
15. YU, W., AND ROSTAD, S. A low-cost set CRDT based on causal lengths. In *Proceedings of the 7th Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)* (2020), pp. 5:1–5:6.