Synql: Replicated Relations and Integrity Maintenance

Victorien Elvinger, Claudia-Lavinia Ignat

Synql: Replicated Relations and Integrity Maintenance

Victorien Elvinger
victorien.elvinger@inria.fr
Inria, Université de Lorraine, CNRS, LORIA, F-54500
Nancy, France

Claudia-Lavinia Ignat
claudia.ignat@inria.fr
Inria, Université de Lorraine, CNRS, LORIA, F-54500
Nancy, France

Abstract

Many offline-first applications use an embedded relational database, such as SQLite, to manage their data. The replication of the database eases the addition of collaborative features to its applications. Most of the approaches for replicating a relational database require coordination at some extent. A few approaches propose a coordination-less replication to allow offline work. These approaches are limited in two ways: (i) They don't respect *Strong Eventual Consistency* that states that two replicas converge as soon as they integrate the same set of modifications; (ii) They fail to preserve user intent in complex scenarios. We propose a new *CRDT* that addresses these two limitations. Our replicated state is defined by the composition of *CRDT* primitives. The state of the database is computed over the replicated state. The user modifications are compensated so that the computed state corresponds to what the users saw and changed.

1 Introduction

The Covid-19 pandemic has led to a massive adoption of applications that dematerialize workspaces. These applications rely heavily on collaborative features. Adding collaborative features to existing applications is hard. An approach consists in replicating the application data without knowing the application internals. This approach has shown its efficiency and simplicity for applications that rely on JSON data [6].

Many applications use embedded relational databases, such as *SQLite*, to manage their data. Distributed relational databases have relied heavily on coordination to maintain data integrity. Several works showed that coordination is costly and can often be avoided [2, 8]. However, they still use coordination to enforce some integrity constraints. This makes these approaches impractical for applications that support offline work.

[11] adapts [6] for replicating relational databases. This allows the addition of collaborative features to offline-first applications that embed *SQLite* for handling their data. This approach enables concurrent insertions, updates, and deletions without coordination. [11] presents also a strategy for maintaining the most commonly used integrity constraints: uniqueness integrity and referential integrity. However, it fails to preserve user intent in several scenarios. Moreover, it doesn't respect *Strong Eventual Consistency* [10] – a property that ensures convergence as soon as every replica has integrated the same modifications. We propose new mechanisms and semantics that address these shortcomings.

2 How to maintain integrity constraints in face of concurrency?

2.1 Uniqueness integrity

In a relational database, a unique key is a set of attributes of a relation that uniquely identifies every tuple of the relation. The primary key of a relation is one of its unique keys.

In a replicated context, tuples can be concurrently inserted and updated. This can lead to uniqueness violation during synchronization. In the Figure 1, Alice (A) and Bea (B) concurrently insert a tuple in the relation *player*. They pick the same primary key. The synchronization results in a uniqueness violation.

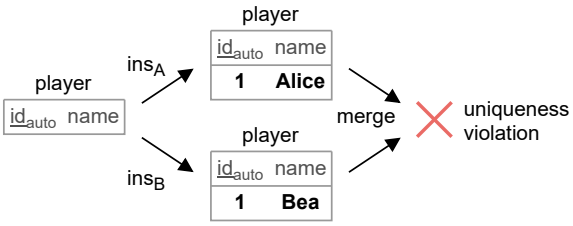

Figure 1. Uniqueness violation

Some replicated databases [9] fall back on coordination to ensure uniqueness integrity. Others [3] only support globally unique identifiers. During synchronization, [11] applies every modification individually in order to catch integrity violations. Upon a uniqueness violation, the conflict resolver undoes the newest modification that caused the violation. In the Figure 2, we assume that the timestamp of the insertion of Alice is lower than the timestamp of the insertion of Bea. The synchronization preserves the insertion of Alice and undoes the insertion of Bea.

This strategy makes the result of a merge sensitive to the order in which the operations are integrated. Two replicas that receive and integrate operations in distinct order can end with different states. They need extra rounds of synchronization to converge to an identical state. Thus, [11] does not respect *Strong Eventual Consistency* [10].

While [11] maintains uniqueness integrity, it fails to preserve the intent of Bea in the Figure 2. The relation *player* uses an automatically incremented primary key. Existing databases widely use automatically incremented primary keys as a generic way to identify a tuple. The specific value of these primary keys is generally not relevant. Based on this observation, we handle automatically incremented primary

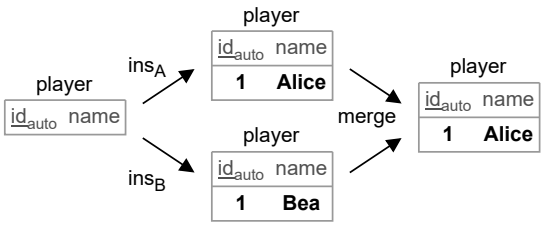

Figure 2. Maintenance of uniqueness integrity in [11].

keys differently from other unique keys. We treat them as local identifiers that are not replicated and for which the convergence doesn't need to be assured. Distinct replicas can assign distinct keys to a same tuple. In the Figure 3, the synchronization preserves the two insertions. The tuple inserted by Alice is identified by 1 on the replica A, and it is identified by 2 on the replica B. Conversely, the tuple inserted by Bea is identified by 1 on the replica B, and it is identified by 2 on the replica A.

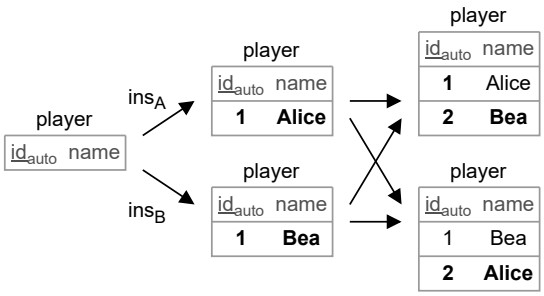

Figure 3. Proposal for the maintenance of uniqueness integrity of automatically incremented primary keys.

2.2 Referential integrity

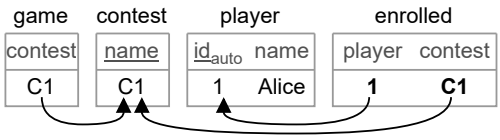

Figure 4. Example of foreign keys.

In a relational database, a foreign key is a set of attributes in a relation that references the primary key of another relation. In the Figure 4, the foreign key of the relation *game* consists of the attribute *contest* and references the primary key of the relation *contest*. The relation *enrolled* has two foreign keys: one that references the primary key of the relation *player* and another one that references the primary key of the relation *contest*. Referential integrity ensures the existence of the tuples referenced by any tuple. Relational databases allow to customize the behavior of the deletion of a tuple when another tuple references it. The deletion can be aborted or propagated to the referencing tuple.

In a replicated context, the deletion of a tuple and its referencing can happen in concurrence. In Figure 5, Alice enrolls herself in the contest $C1$, while Bea concurrently deletes $C1$. The synchronization breaks referential integrity.

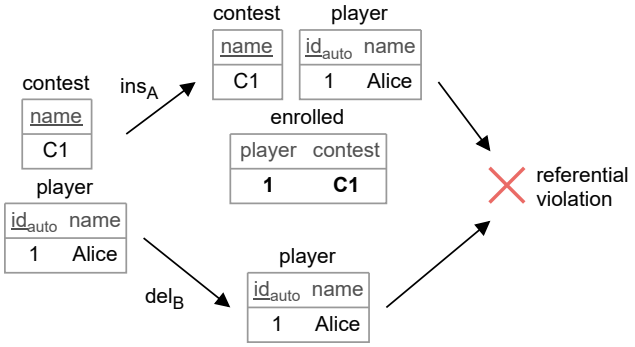

Figure 5. The concurrent deletion of a contest and insertion of an enrollment that references the contest lead to a violation of referential integrity.

[11] catches the violations of referential integrity. When a violation occurs, the conflict resolver undoes the insertion of the tuple that references the deleted tuple. In the Figure 6, the synchronization maintains referential integrity by undoing the enrollment of Alice.

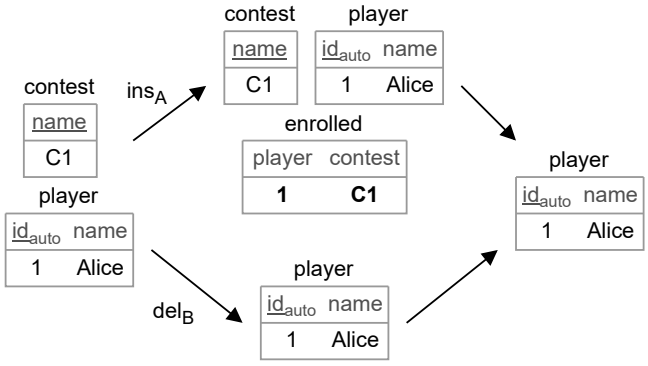

Figure 6. Maintenance of referential integrity in [11].

This strategy doesn't support alternative merge semantic in which the insertion of an enrollment wins over the deletion of the referenced contest. It requires that each operation be applied individually to catch any integrity violation. This limits the implementation space. Moreover, some databases, such as SQLite, don't verify referential integrity in their default configuration. In this case, the synchronization results in a dangling reference as illustrated in Figure 7.

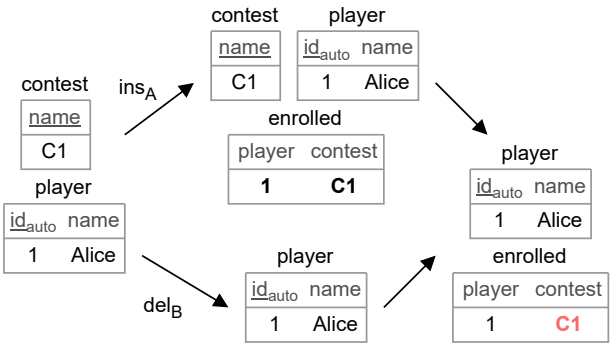

Figure 7. Dangling reference in SQLite.

[3] proposes another approach to maintain referential integrity. The enrollment of Bea embeds a *compensation* that ensures the existence of the contest. The synchronization restores the contest as illustrated in Figure 8.

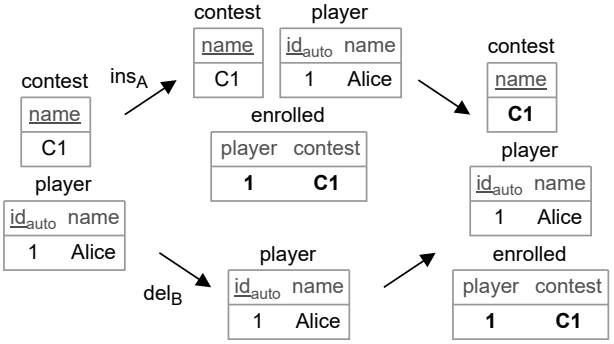

Figure 8. Maintenance of referential integrity in [3].

This approach preserves referential integrity, but doesn't preserve user intent in other scenarios. In the Figure 9, a *game* references the contest $C1$. The deletion of $C1$ leads to the deletion of the game (we assume propagated deletions). The synchronization restores the contest $C1$, but doesn't restore the *game*. We could extend this approach in order to embed the insertion of the game in the enrollment action. However, if a game is concurrently inserted, then we cannot embed its insertion in the enrollment. Their proposal also allows to choose the alternative merge semantic where the concurrent deletions of a contest and its references lead to the deletion of all referencing enrollments. To do this, the compensation deletes all referencing tuples.

Our proposal takes a different path. Instead of just restoring the contest, it also restores the game that references the contest as illustrated in Figure 10. We also leverage existing annotations of the database schema to determine which merge semantic to adopt. If the deletion of a tuple is propagated to its referencing tuple, then we adopt a *remove-win* semantic. Upon the deletion of a tuple, all referencing tuples,

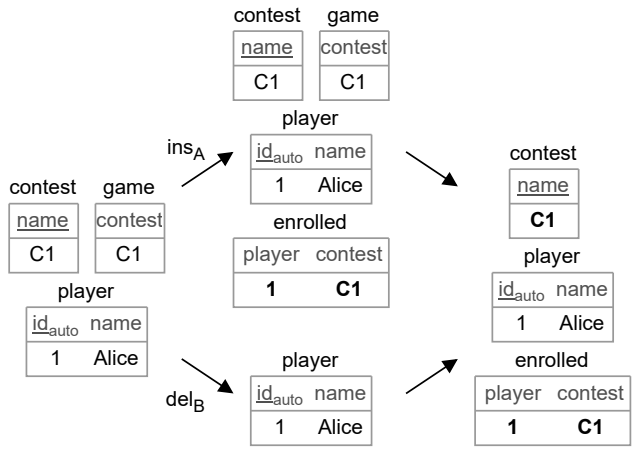

Figure 9. Limits of compensated operations in [3].

including concurrently inserted tuples, are then deleted. If the deletion is aborted, then we adopt an *add-win* semantic. The deleted tuple is restored if a referencing tuple was concurrently inserted.

[3] tries to preserve the invariants of the application. This is at a high level. In our approach we preserve low-level invariants (referential integrity and uniqueness integrity). This makes our approach more general. Indeed, their approach requires to perform a static analysis for every application to be replicated. It requires to model the application and may require user input during the static analysis of the model.

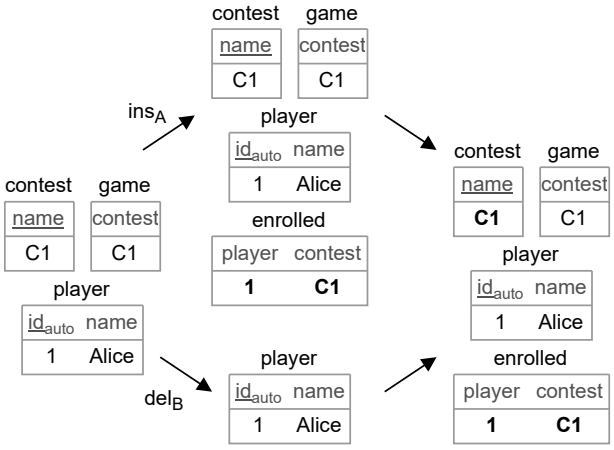

Figure 10. Proposal for the maintenance of referential integrity in the case of an abort semantic.

3 Synql: A CRDT for replicated relations and integrity maintenance

Synql allows to replicate an existing relational database without modifying the database engine or the application. To do

this, *Synql* relies on a *Git*-like model as illustrated in Figure 11. First the administrator has to initialize an existing database in order to obtain a replicated database (1.). The initialization creates new relations and new triggers that store and maintain replicated metadata. An administrator can add replicas by cloning an existing replica (2.). The replicas can be concurrently updated without any coordination (3.). The application reads and updates its database in the usual way by submitting SQL requests. The database triggers automatically update the replicated metadata. The replicas are synchronized in background (4.).

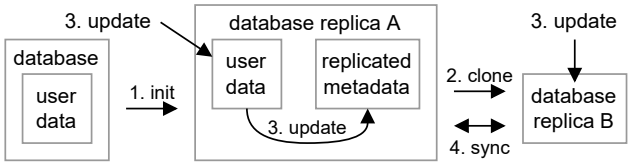

Figure 11. Architecture of *Synql*.

The basis of the replication system relies on the identification of every inserted tuple with a globally unique identifier consisting of a monotically increasing timestamp [7] and a unique replica identifier. This is closely related to the concept of *dot* [1] with an important difference: the main component of the identifier is the timestamp instead of the replica identifier. This allows to use its identifiers as timestamps. Thus, we name them *labeled timestamps*. Labeled timestamps are ordered lexicographically. This induces a total order between them. For example, $1_A < 1_B < 2_A < \dots$ where A, B are replica identifiers and $1, 2$ are timestamps. $\mathbb{N}_{\mathbb{I}}$ denotes the set of labeled timestamps with $\mathbb{I}$ the set of replica identifiers.

Similar to [11], we use Last-Writer-Win (LWW) Registers [5] for replicating tuple attributes. A LWW-Register allows concurrent reads and writes without coordination. It is part of the Conflict-free Replicated Data Types (*CRDTs*) that ensures Strong Eventual Consistency [10] – a property that ensures convergence as soon as every replica has integrated the same modifications. As illustrated in Figure 12, it associates a timestamp to a value (Equation 1) and keeps the value with the most recent timestamp upon merging (Equation 4). A read returns the value without its timestamp (Equation 2). A write associates to a value a new timestamp (Equation 3).

$$\mathsf{LWWReg} \stackrel{\text{def}}{=} \mathsf{Value} \times \mathbb{N}_{\mathbb{I}} \tag{1}$$

$$\mathsf{rd}(\langle v, t\rangle) \stackrel{\text{def}}{=} v \tag{2}$$

$$\mathsf{wr}_t(v) \stackrel{\text{def}}{=} \langle v, t\rangle \tag{3}$$

$$\langle v, t\rangle \sqcup \langle v', t'\rangle \stackrel{\text{def}}{=} \langle v, t\rangle \textbf{ if } t > t' \textbf{ else } \langle v', t'\rangle \tag{4}$$

Figure 12. Last-Writer-Win Register [5]

In contrast to [11], *Synql* replicates a foreign key as a single attribute that stores the identifier of the referenced tuple. Also, *Synql* doesn't replicate auto-incremented attributes. It uses a local mapping to find the local value of an auto-incremented attribute of a tuple from the identifier of the tuple. To avoid any ambiguity, we use the term *field* to denote an attribute of a replicated tuple. For a relation $r \in \mathsf{Rel}$, we denote by $\mathsf{Fields}(r)$ the set of fields of r.

We use the concept of *causal length* [12] to support undoing and redoing insertions. We represent a deletion as an undone insertion. We formalize the concept of *causal length* through a new *CRDT*: the *Causal-Length Flag* (*CLFlag*). The Figure 13 presents its implementation. The flag consists of a natural number (Equation 5). If the number is odd, then the flag is enabled (Equation 6). The state of the flag is toggled by incrementing by 1 its state (Equation 7 and Equation 8). Upon merging, the maximum number wins (Equation 9).

$$\mathsf{CLFlag} \stackrel{\text{def}}{=} \mathbb{N}_0 \tag{5}$$

$$\mathsf{enabled}(n) \stackrel{\text{def}}{=} \mathsf{odd}(n) \tag{6}$$

$$\mathsf{enable}(n) \stackrel{\text{def}}{=} n \textbf{ if } \mathsf{enabled}(n) \textbf{ else } n+1 \tag{7}$$

$$\mathsf{disable}(n) \stackrel{\text{def}}{=} n+1 \textbf{ if } \mathsf{enabled}(n) \textbf{ else } n \tag{8}$$

$$n \sqcup n' \stackrel{\text{def}}{=} max(n, n') \tag{9}$$

Figure 13. Causal-Length Flag CRDT

Our replication model relies on the composition of *CRDT* primitives [4]. In the Figure 14, we summarize the merge semantic of the primitives we are interested in. The merge of a pair is the point-wise merge of its components (Equation 10). The merge of two maps (partial functions) is the point-wise merge of the values that share the same key (Equation 11). Note that in the merge, each map is extended to a total function that returns the bottom element $\bot$ when the key is not part of the domain of the map, i.e. $m_\bot = m \cup \{k \mapsto \bot \mid k \notin \mathsf{dom}(m)\}$.

$$\langle a, b\rangle \sqcup \langle a', b'\rangle \stackrel{\text{def}}{=} \langle a \sqcup a', b \sqcup b'\rangle \quad \textbf{where } \langle a, b\rangle, \langle a', b'\rangle \in A \times B \tag{10}$$

$$m \sqcup m' \stackrel{\text{def}}{=} \left\{k \mapsto m_\bot(k) \sqcup m'_\bot(k) \;\middle|\; k \in \mathsf{dom}(m) \cup \mathsf{dom}(m')\right\} \quad \textbf{where } m, m' \in K \hookrightarrow V \tag{11}$$

Figure 14. CRDT primitives and their merge semantic

The Figure 15 summarizes our replicated state (Equation 12) and associated δ-mutators. Every replicated tuple is a set of

Last-Writer-Win Registers [5] indexed by the fields of a given relation r. Every replicated relation consists of a map that associates to a replicated tuple its identifier, i.e. creation labeled timestamp, and a Causal-Length Flag. When the flag is set, the tuple is marked as deleted. Finally, a replicated database is a set of replicated relations indexed by the relations.

$$\mathsf{RDb} \stackrel{\text{def}}{=} \{r \in \mathit{Rel}\} \hookrightarrow \mathbb{N}_{\mathbb{I}} \hookrightarrow \big(\mathsf{Fields}(r) \hookrightarrow \mathsf{LWWReg}\big) \times \mathsf{CLFlag} \tag{12}$$

$$\mathsf{read}(\{r \mapsto t \mapsto \langle\{f \mapsto \mathsf{reg}\}, \mathsf{delFlag}\rangle\}) \stackrel{\text{def}}{=} \{r \mapsto t \mapsto \langle\{f \mapsto \mathsf{rd}(\mathsf{reg})\}, \mathsf{enabled}(\mathsf{delFlag})\rangle\} \tag{13}$$

$$\mathsf{ins}^{\delta}_{t}(\mathsf{db}, r, \{f \mapsto v\}) \stackrel{\text{def}}{=} r \mapsto t \mapsto \langle\{f \mapsto \mathsf{wr}_t(v)\}, \bot\rangle \tag{14}$$

$$\mathsf{del}^{\delta}_{t}(\mathsf{db}, r, t') \stackrel{\text{def}}{=} r \mapsto t' \mapsto \langle\bot, \mathsf{enable}(\mathsf{delFlag})\rangle \quad \textbf{where } \langle\_, \mathsf{delFlag}\rangle = \mathsf{db}(r)(t') \tag{15}$$

$$\mathsf{update}^{\delta}_{t}(\mathsf{db}, r, t', f, v) \stackrel{\text{def}}{=} r \mapsto t' \mapsto \langle f \mapsto \mathsf{wr}_t(v), \bot\rangle \tag{16}$$

Figure 15. Replicated State

δ-mutators [1] return the minimal state that encodes the change to propagate and merge. The current labelled timestamp denoted by t is an implicit parameter of the presented mutators. The ins δ-mutator (Equation 14) returns a state that includes a newly inserted tuple of a relation r identified by its creation timestamp t [5]. The del δ-mutator (Equation 15) returns a state that turns on the associated flag (delFlag) of the tuple identified by t' in the relation r. The update δ-mutator (Equation 16) returns a state that updates to v the field f of the tuple identified by t' in the relation r. The read (Equation 13) evaluates the registers and the flags.

Thanks to the composition of *CRDT* primitives, we obtain a new *CRDT*. However, this *CRDT* doesn't guarantee integrity constraints. Indeed, several tuples may share the same unique key and a tuple can reference a deleted tuple. Several approaches, such as [11], change the merge operation in order to ensure integrity constraints. The merge depends on the current state of the replica. This makes their approach Eventual Consistent, but not Strong Eventual Consistent.

Instead of modifying the merge operation, we propose to (deterministically) compute a state without integrity violations from the replicated state. This has the advantage to ensure both integrity constraint and Strong Eventual Consistency. To obtain the computed state, we clone the replicated state and apply successive removals and additions:

1. Remove all replicated tuples marked as deleted.
2. Add replicated tuples transitively referenced by a tuple that is not marked as deleted and has an abort semantic upon the deletion of the referenced tuple.
3. For all set of replicated tuples that have at least one conflicting unique key, keep the oldest (according to their identifiers) one and remove others.
4. Remove all replicated tuples that transitively reference at least one replicated tuple not present in the computed state.

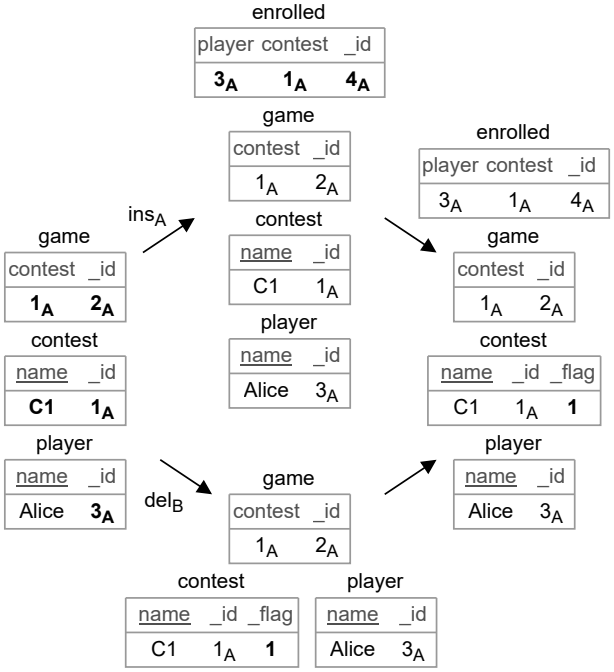

Figure 16. Example of replicated state merging.

To illustrate the merge of two replicated databases, we revisit the example of the Figure 10 which represents the user data. The Figure 16 represents a simplified view of the replicated state. It doesn't include the timestamps of the registers and depicts a causal length flag only when it is not equal to its initial state 0. Note that every tuple is associated to its identifier _id, e.g. the contest $C1$ is identified by 1_A. Moreover, references use tuple identifiers. For instance, the game identified by 2_A references the contest identified by 1_A. Upon the deletion of the contest 1_A, the replica B marks the contest as deleted by enabling its causal-length flag. The state of the flag is thus 1. Although the deletion of the contest is propagated to the game that references it, *Synql* doesn't mark the game as deleted. Its deletion is effective in the computed state (step 4). We talk more about this choice in the following paragraphs. Replica A concurrently enrolls Alice in the contest. Upon synchronization, the new state of user data is computed from the replicated state. The computed state removes replicated tuples marked as deleted (step 1). Here, only the contest is marked as deleted. Then, we add again the contest because it is transitively referenced by an enrollment (step 2). We assume that the enrollment has an

abort semantic. The (step 3) and (step 4) don't change the computed state. We end with a computed state in which the deletion of the contest is undone.

The computation of the database state from the replicated state may lead to surprising effects when local modifications are performed. In the Figure 16, if a replica deletes the enrollment after the merging, then the contest and the game are also deleted in the computed state. This is due to the fact that the contest is still marked as deleted. This violates user intent. To address this issue, local modifications must be compensated. In the previous example, the deletion of the enrollment has to redo the insertion of the contest as illustrated in Figure 17.

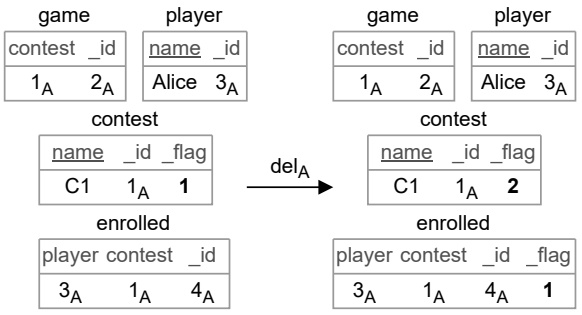

Figure 17. Example of compensated deletion.

A compensation is applied in two cases: (i) the deletion or the update of a foreign key that uses an abort semantic, and (ii) the deletion or the update of a unique key. In the first case, the compensation marks all tuples that are transitively referenced by the reference as non-deleted. In the second case, the compensation marks all tuples that share a conflicting unique key as deleted. The deletion of a unique key or foreign key happen when its tuple is deleted.

4 Conclusions and ongoing works

We proposed a new *CRDT* for replicating relations and maintaining integrity constraints in face of concurrent modifications. In contrast to previous approaches, our proposal enforces Strong Eventual Consistency and respects user intent in complex scenarios. Its replicated state consists of the composition of *CRDT* primitives. The state of the database is computed from the replicated state by deterministically resolving all integrity violations. Local modifications are compensated in a way that ensures user intent. An implementation of a proof of concept is underway and available at https://github.com/coast-team/synql.

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distributed Comput.*, 111:162–173, 2018. doi: 10.1016/j.jpdc.2017.08.003. URL https://doi.org/10.1016/j.jpdc.2017.08.003.

[2] Peter Bailis, Alan D. Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, 2014. doi: 10.14778/2735508.2735509. URL http://www.vldb.org/pvldb/vol8/p185-bailis.pdf.

[3] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno M. Preguiça. IPA: invariant-preserving applications for weakly consistent replicated databases. *Proc. VLDB Endow.*, 12(4):404–418, 2018. doi: 10.14778/3297753.3297760. URL http://www.vldb.org/pvldb/vol12/p404-balegas.pdf.

[4] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. Composition in state-based replicated data types. *Bull. EATCS*, 123, 2017. URL http://eatcs.org/beatcs/index.php/beatcs/article/view/507.

[5] Paul R. Johnson and Robert Thomas. Maintenance of duplicate databases. *RFC*, 677:1–10, 1975. doi: 10.17487/RFC0677. URL https://doi.org/10.17487/RFC0677.

[6] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In Hidehiko Masuhara and Tomas Petricek, editors, *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*, pages 154–178. ACM, 2019. doi: 10.1145/3359591.3359737. URL https://doi.org/10.1145/3359591.3359737.

[7] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, volume 8878 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2014. doi: 10.1007/978-3-319-14472-6_2. URL https://doi.org/10.1007/978-3-319-14472-6_2.

[8] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 265–278. USENIX Association, 2012. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

[9] Pedro Lopes, João Sousa, Valter Balegas, Carla Ferreira, Sérgio Duarte, Annette Bieniusa, Rodrigo Rodrigues, and Nuno M. Preguiça. Antidote SQL: relaxed when possible, strict when necessary. *CoRR*, abs/1902.03576, 2019. URL http://arxiv.org/abs/1902.03576.

[10] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011. doi: 10.1007/978-3-642-24550-3_29. URL https://doi.org/10.1007/978-3-642-24550-3_29.

[11] Weihai Yu and Claudia-Lavinia Ignat. Conflict-free replicated relations for multi-synchronous database management at edge. In *IEEE International Conference on Smart Data Services, SMDS 2020, Beijing, China, October 19-23, 2020*, pages 113–121. IEEE, 2020. doi: 10.1109/SMDS49396.2020.00021. URL https://doi.org/10.1109/SMDS49396.2020.00021.

[12] Weihai Yu, Victorien Elvinger, and Claudia-Lavinia Ignat. A generic undo support for state-based crdts. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi: 10.4230/LIPIcs.OPODIS.2019.14. URL https://doi.org/10.4230/LIPIcs.OPODIS.2019.14.