Synql: A CRDT-based Approach for Replicated Relational Databases with Integrity Constraints

Claudia-Lavinia Ignat[1], Victorien Elvinger[1], and Habibatou Ba[1]

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

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 do not 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 the combined effect of operations' intent in complex scenarios. We propose Synql, an approach based on Conflict-free Replicated Data Types (CRDTs) that addresses these two limitations. Synql relies on a replicated state 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 pandemic and the generalisation of remote working 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. Local-first software approach proved its simplicity and efficiency in adding collaboration to existing applications by replicating the application data without knowing the application internals [13].

Many applications use embedded relational databases, such as SQLite, to manage their data. In order to ensure a high availability and low latency, databases are commonly replicated [9,8]. As stated by the CAP theorem [10,7,11] high availability and data consistency is difficult to achieve in distributed systems in the presence of network partitions. Most database systems such as SQL, NewSQL and some NoSQL (e.g. graph databases such as Neo4j) aim to achieve the ACID properties (Atomicity, Consistency, Isolation, Durability) and to maintain a strong consistency by means of serialisability and its many implementations including lock-based and pre-scheduling mechanisms [19,21,22]. These mechanisms for ensuring strong consistency require coordination which limits data availability and system scalability and increases latency.

Those NoSQL database systems that want to ensure a high data availability relax consistency and they conform to BASE (Basically Available, Soft State,

Eventually Consistent) rather than ACID transaction model. They usually implement lazy replication which may result in a situation where reads on replicas might be inconsistent for a short period of time. Voldemort [20] and Cassandra [15] use quorums for achieving consistency which require coordination.

Several applications built upon databases used a weak consistency in order to avoid a costly coordination [3,16]. However, these application invariants might get violated due to concurrent executions of the operations [2]. These applications still use coordination to enforce some integrity constraints. They are therefore impractical for applications that support offline work.

In [24], authors propose Conflict-free Replicated Relations (CRR) where they adapt the Local-First Software [13] for replicating relational databases. This allows the addition of collaborative features to offline-first applications that embed SQLite for handling their data. CRR approach enables concurrent insertions, updates, and deletions without coordination. It presents also a strategy for maintaining the most commonly used integrity constraints: uniqueness integrity and referential integrity. However, it fails to preserve the combined effect of operations' intent in several scenarios. In a collaborative environment, the combined effect of operations' intent represents the ideal merger of individual operations performed on the different replicas. It combines the highest possible number of operations with respect to their original impact on their specific replica state. Moreover, it does not respect *Strong Eventual Consistency (SEC)* [18] – a property that ensures convergence as soon as every replica has integrated the same modifications without further coordination.

We propose a novel solution which combines CRDT primitives with specific merging semantics and compensation mechanisms to ensure *SEC* and preserve the combined effect of operations' intent.

This paper presents the following contributions:

- A scenario-based study of the state of the art which highlights its limitations.
- A definition of the essential requirements necessary for a solution to address these limitations.
- A novel Synql approach based on CRDTs to ensure *SEC* and preserve the combined effect of operations' intent for integrity maintenance in replicated databases.
- A compensation technique to restore deleted tuples when required.

This paper is structured as follows. Firstly, we explore the preservation of integrity constraints in face of concurrency. Next, we introduce Synql, our CRDT-based approach for replication and integrity maintenance. Then, we study and evaluate the related work before concluding the paper in the following section.

2 Integrity constraints maintenance

Integrity constraints in a database are rules used to ensure the accuracy and consistency of data in a relational database. These constraints enforce certain conditions that the data in the database must comply with. While there are

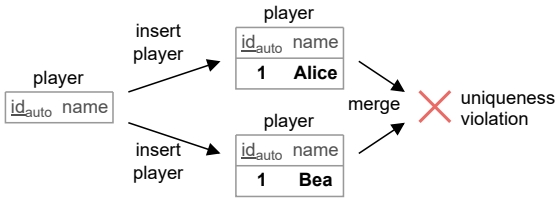

Fig. 1. Uniqueness violation

several types of integrity constraints, in this paper we address uniqueness and referential integrities.

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. Uniqueness integrity ensures that no two tuples in the relation have the same unique key value.

In a replicated context, tuples can be concurrently inserted and updated. This can lead to uniqueness violation during synchronization. In 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.

Some replicated databases such as Invariant Preserving Applications (IPA) [5] only support globally unique identifiers. Others such as AntidoteSQL (AQL) [17] occasionally fall back on coordination when the user wants to achieve strict consistency. AQL relies on the use of specific concurrency semantics to ensure uniqueness integrity.

During synchronization, CRR [24] 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 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. While CRR maintains uniqueness integrity, it fails to preserve the combined effect of operations' intent (the insertion of Bea is not included in the final state).

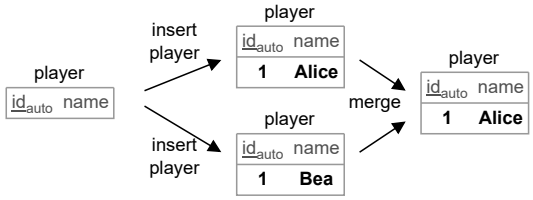

Fig. 2. Maintenance of uniqueness integrity in CRR [24].

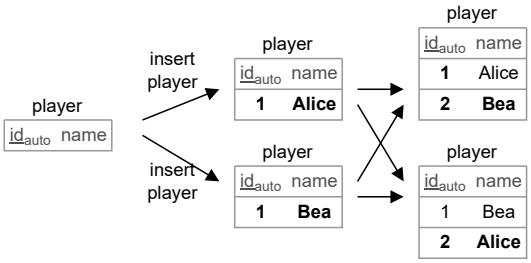

Fig. 3. Proposal for uniqueness integrity of automatically incremented primary keys.

Fig. 4. Example of foreign keys.

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 keys differently from other unique keys. We treat them as local identifiers that are not replicated and for which the convergence does not need to be ensured. 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.

2.2 Referential integrity

In a relational database, a foreign key is a set of attributes in a relation that references the primary key of another relation. In 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.

CRR [24] 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 Figure 6, the synchronization maintains referential integrity by undoing the enrolment of Alice.

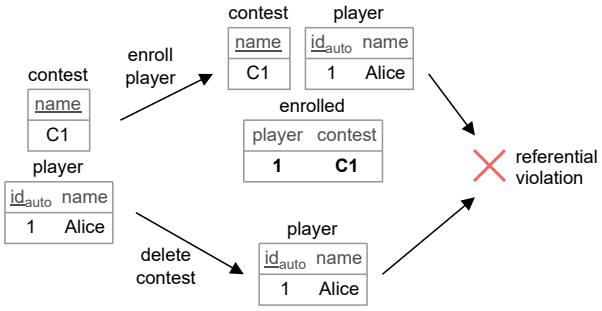

Fig. 5. The concurrent deletion of a contest and insertion of an enrolment that references the contest lead to a violation of referential integrity.

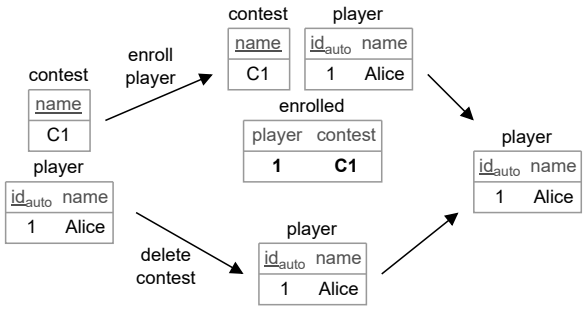

Fig. 6. Maintenance of referential integrity in CRR [24].

This CRR strategy does not support alternative merge semantic in which the insertion of an enrolment wins over the deletion of the referenced contest. It requires that each operation be applied individually to catch any integrity violation. This limits the possibilities for implementation of state-based [18] or delta-based CRDTs [1]. Moreover, some databases, such as SQLite, do not verify referential integrity in their default configuration. In this case, the synchronization results in a dangling reference as illustrated in Figure 7.

Moreover, in several scenarios where the same set of operations is executed in different order by the different users, CRR might end up with divergent states. In the scenario in Figure 8, Alice and Bea start working from the same state of the database consisting of the relation *player* containing *P1* and *P2* and of the relation *contest* containing *C1*. Alice creates an enrolment *E1(K1,P1,C1)*, where the primary key of the *enrolled* relation is an automatically incremented identifier, while Bea removes *P1*. Afterwards, Alice sends her changes to Bea and Bea sends her changes to Alice. When Bea receives the enrolment of player *P1*, the enrolment is deleted as there is an integrity constraint violation as *P1* was deleted. Afterwards, Alice updates the enrolment *E1* to *E1(K1,P2,C1)* and sends her changes to Bea. When Bea receives the update enrolment of Alice, the operation is not executed as *E1* does not exist on the Bea's site. Finally,

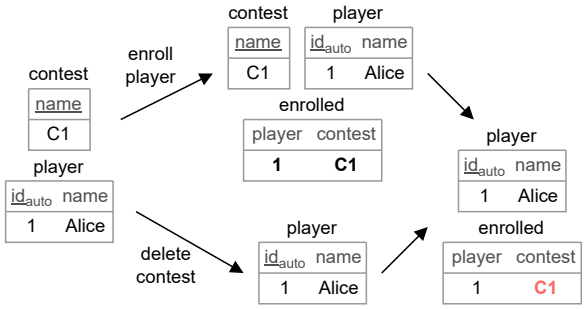

Fig. 7. Dangling reference in SQLite.

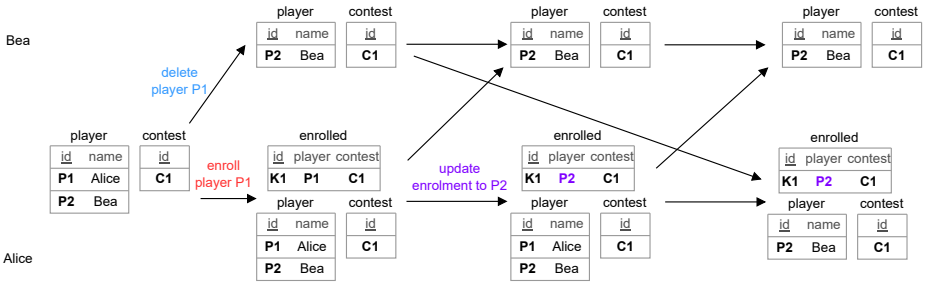

Fig. 8. Divergence of final states in CRR [24].

when Alice receives the deletion of Bea, she executes it. We can notice that, even though Alice and Bea received the same operations, their states diverge. Therefore, CRR does not satisfy SEC.

IPA [5] and AQL [17] propose another approach to maintain referential integrity. The enrolment of Alice embeds a *compensation* that ensures the existence of the contest. The synchronization restores the contest as illustrated in Figure 9.

This approach of IPA and AQL preserves referential integrity, but does not preserve the combined effect of operations' intent in other scenarios. In IPA and AQL, respectively Figure 10 and Figure 11, 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 does not restore the *game*. This breaks operation intent as the deletion of the contest resulted in the deletion of the game and not the contest. In IPA (Figure 10), the deleted tuples are directly removed from the computed state, while in AQL (Figure 11), the deletion is based on visibility flags (I for inserted or updated tuples, T for referenced tuples and D for deleted tuples added to the visibility column "$\#st$").

To preserve the combined effect of operations' intent, we could extend this approach in order to embed the insertion of the game in the enrolment action. However, if a game is concurrently inserted, then we cannot embed its insertion in

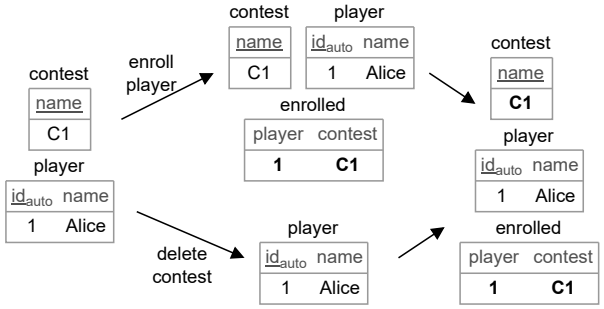

Fig. 9. Maintenance of referential integrity in IPA [5] and AQL [17].

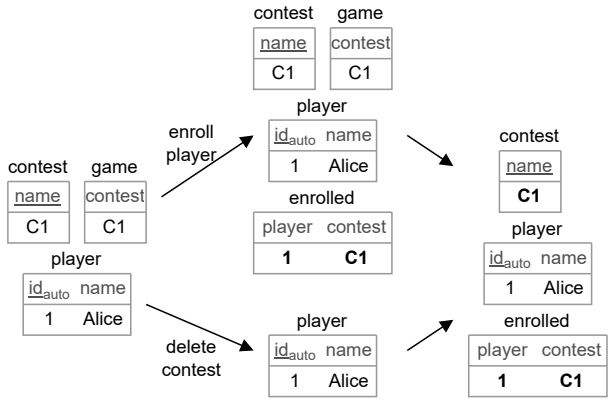

Fig. 10. Limits of compensated operations in IPA [5].

the enrolment. The proposal of IPA and AQL 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 enrolments. To do this, the compensation deletes all referencing tuples.

Our proposal takes a different path. Instead of just restoring the contest, it also preserves the game that references the contest as illustrated in Figure 12. 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 (e.g. DELETE CASCADE in SQLite), then we adopt a *remove-win* semantic. Upon the deletion of a tuple, all referencing tuples, including concurrently inserted tuples, are then deleted. If the deletion is aborted (e.g. DELETE RESTRICT in SQLite), then we adopt an *add-win* semantic. The deleted tuple is restored if a referencing tuple was concurrently inserted.

IPA [5] tries to preserve the application invariants at a high level. Synql preserves low-level invariants (referential integrity and uniqueness integrity). This makes Synql more general. Indeed, IPA requires to perform a static analysis for

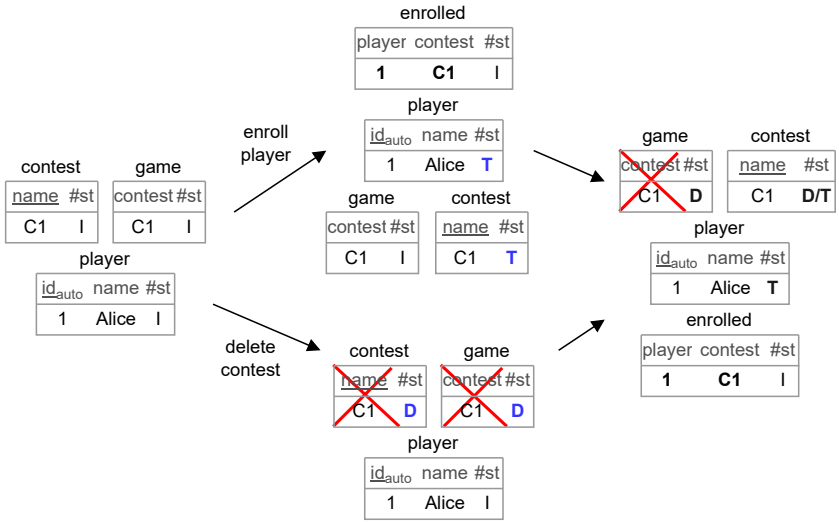

Fig. 11. Limits of compensated operations in AQL [17].

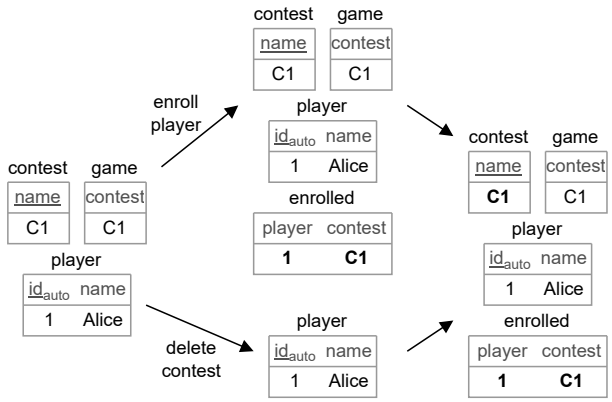

Fig. 12. Proposal for the referential integrity in the case of an abort semantic.

every application to be replicated. It requires to model the application and may require user input during the static analysis of the model.

3 Synql

Synql approach 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 13. First the administrator has to initialize an existing database in order to obtain a replicated database (1.). The initialization

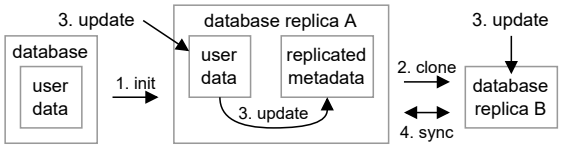

Fig. 13. Architecture of *Synql*.

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

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

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

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

Fig. 14. Last-Writer-Win Register [12]

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.).

The basis of the replication system relies on the identification of every inserted tuple with a globally unique identifier consisting of a monotonically increasing timestamp [14] 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, $1_A < 1_B < 2_A < \ldots$ 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 CRR [24], we use Last-Writer-Win (LWW) Registers [12] for replicating tuple attributes. LWW Registers can be created, updated and deleted using the LWW rule to arbitrate between concurrent changes without coordination. LWW ensures a total order of operations, at the cost of losing concurrent updates. An LWW Register is a Conflict-free Replicated Data Type (*CRDT*) that ensures Strong Eventual Consistency [18] – a property that ensures convergence as soon as every replica has integrated the same modifications. As illustrated in Figure 14, it associates a timestamp to a value (Equation 1). A read returns the value without its timestamp (Equation 2). A write associates to a value a new timestamp (Equation 3). Upon merging, it keeps the value with the most recent timestamp (Equation 4).

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

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

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

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

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

Fig. 15. Causal-Length Flag CRDT

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

$$m \sqcup m' \stackrel{\text{def}}{=} \{k \mapsto m_\perp(k) \sqcup m'_\perp(k) \mid k \in \mathsf{dom}(m) \cup \mathsf{dom}(m')\} \textbf{ where } m, m' \in K \hookrightarrow V \tag{11}$$

Fig. 16. CRDT primitives and their merge semantic

In contrast to CRR [24], *Synql* replicates a foreign key as a single attribute that stores the identifier of the referenced tuple. Also, *Synql* does not 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* [23] 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 15 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).

Our replication model relies on the composition of *delta-based CRDT* primitives [6]. A delta-based CRDT is a variant of CRDTs that optimizes data synchronization by transferring only the recent changes, or *deltas*, rather than the entire state or all updates since the last synchronization. In the Figure 16, 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 $\perp$ when the key is not part of the domain of the map, i.e. $m_\perp = m \cup \{k \mapsto \perp \mid k \notin \mathsf{dom}(m)\}$.

Figure 17 summarizes our replicated state (Equation 12) and associated δ-mutators. Every replicated tuple is a set of Last-Writer-Win Registers [12] 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

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

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

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

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

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

Fig. 17. Replicated State

as deleted. Finally, a replicated database is a set of replicated relations indexed by the relations.

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 read (Equation 13) evaluates the registers and the flags. 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 [12]. 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.

Thanks to the composition of *CRDT* primitives, we obtain a new *CRDT*. However, this *CRDT* does not guarantee integrity constraints. Indeed, several tuples may share the same unique key and a tuple can reference a deleted tuple. Several approaches, such as CRR [24], 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 constraints 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.

To illustrate the merge of two replicated databases, we revisit the example of the Figure 12 which represents the user data where Alice enrolled a player

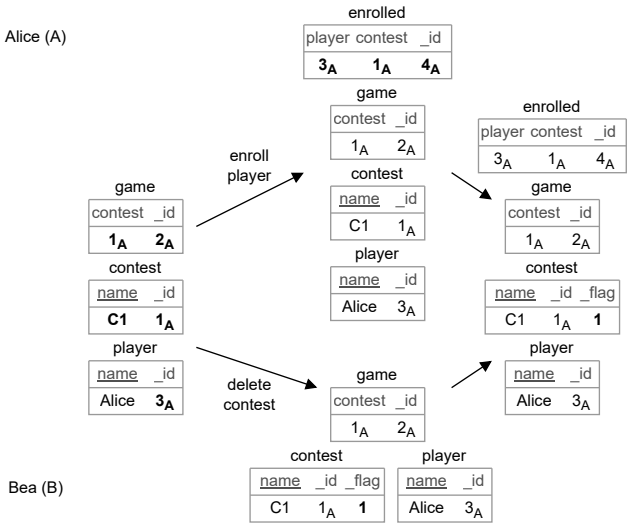

Fig. 18. Example of replicated state merging.

and Bea deleted a contest. We identify the replica of Alice as A and the one of Bea as B. The Figure 18 represents a simplified view of the replicated state. It does not 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 contest's deletion is propagated to the game that references it, *Synql* does not mark the game as deleted yet. 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 while replica B deletes 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. We assume that the enrolment has an abort semantic. At step 2, as contest is referencing a non-deleted tuple which has an abort semantic, it is added back in the merged state. The steps 3 and 4 do not change the computed state. In fact, for step 4, as contest is restored at step 2, its deletion is finally not propagated to the game. 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 18, if a replica deletes the enrolment 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

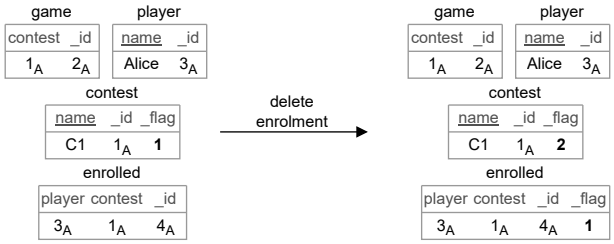

Fig. 19. Example of compensated deletion

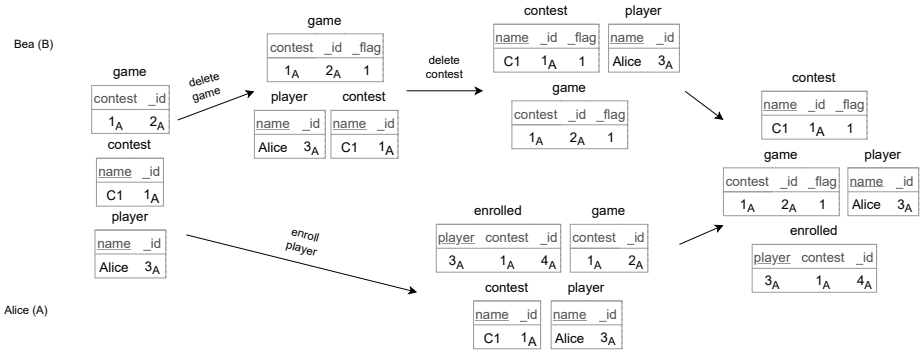

Fig. 20. Synql merging when game and contest are deleted at replica B.

is still marked as deleted despite its restoration in the previous computed state. This violates operation intent. To address this issue, local modifications must be compensated. In the previous example, as the deletion of the enrolment will impact the referenced contest, the insertion of the contest has to be redone when performing the deletion. This is achieved by incrementing the delete flag of the referenced tuple by one as illustrated in Figure 19.

A compensation is applied in two cases: (i) the deletion or the update of a foreign key that uses an abort semantic, (ii) the insertion of a tuple which references another tuple marked as deleted but present in the database state. In the first case, the compensation marks all tuples that are referenced by the reference as non-deleted, if those tuples are also referenced by a tuple not marked as deleted. For example, if the game is previously deleted at one replica (Figure 20), then the compensation would behave as illustrated in Figure 21. As the game is already marked as deleted, the deletion of the enrolment would not lead to a constraint violation (since both contest and game are marked as deleted). Moreover, if the deletion of the enrolment led to the compensation of both the contest and the game, then that would break the initial operation intent and counteract the effect of the game deletion at replica A.

In the second case, the compensation marks all tuples referenced by the new insertion as non-deleted, if those tuples are still present in the database

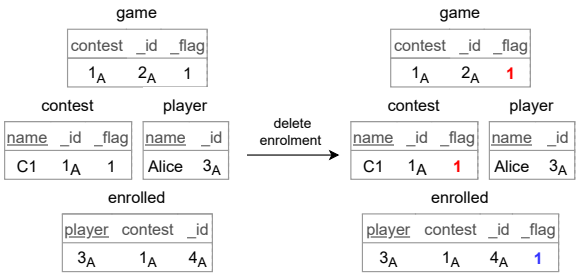

Fig. 21. Compensation for the scenario in Figure 20

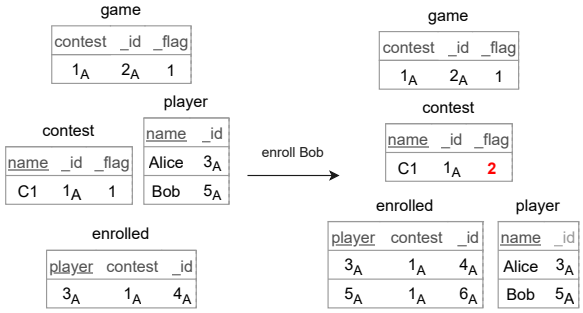

Fig. 22. Compensation for the insertion of a tuple referencing another tuple marked as delete

state, meaning that they have been restored by a previous merging process. For example, if we perform an insertion where we enroll Bob to the same contest as Alice, we will end up in a state where the contest is compensated (delete flag incremented by one) as it is referenced by the new enrolled tuple. This situation is illustrated in Figure 22. Note that, if more than one of the previously mentioned compensation cases happen consecutively at the same replica, compensation is only applied once.

4 Related Work

Several coordination-less approaches, such as AQL [17], CRR [24] and IPA [5], have been designed to allow asynchronous work on databases. However, AQL and CRR fail to provide Strong Eventual Consistency and all of them fail to preserve the combined effect of operations' intent in complex scenarios.

AQL [17] is a database system that enables programmers to mitigate SQL consistency when possible. It is a preventive technique which allows application developers to choose between various semantics (no concurrency, update-wins,

delete-wins) in order to avoid constraint violation in an update-delete scenario. For the no concurrency semantic, AQL uses multi-level locks in shared or exclusive mode to regulate the concurrent execution of actions in its databases [4,17]. Therefore, this method employs coordination in environments where strict consistency is required [17]. Moreover, the update-wins and delete-wins semantics frequently result in scenarios where the original operation intent is compromised (Figure 11).

CRR [24] is a database management system which resolves constraint violation, without coordination, by combining the concept of causal length, an abstraction used to identify the causality between various updates, with a timestamp for each row in the database. In face of a conflict, the application undoes the newest operation causing the violation by incrementing its causal length by one and rolls back to the previous state. If the causal length is odd, then the action has been performed and included in the final state. If the causal length is even and different than 0, the action has been undone and is not taken into account in the final state [24].

IPA [5] is a database management system method which aims to preserve applications' invariants under weak consistency without impacting their availability and their latency. Rather than relying on coordination to prevent concurrency, this approach enables concurrent execution of operations and utilizes conflict resolution policies to guarantee a deterministic outcome (given the concurrently executing operations) while preserving invariants. However, these resolution and compensation techniques often lead to situation where the initial operation intent is broken (Figure 10).

Synql achieves coordination-less replication while maintaining Strong Eventual Consistency. It provides support and compensation in face of constraint violations while preserving the combined effect of operations' intent.

5 Conclusions

We proposed a new approach called Synql 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 combined effect of operations' 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 the preservation of combined effect of operations' intent. The implementation is available at `https://github.com/coast-team/synql`.

6 Acknowledgements

This project was supported by the following public fundings: a French government grant managed by the Agence Nationale de la Recherche as part of the

France 2030 program, reference ANR-22-EXEN-0003 (PEPR eNSEMBLE / PILOT) and the France 2030 program, supporting the DXP project as part of the IPCEI-CIS European initiative.

References

1. Almeida, P.S., Shoker, A., Baquero, C.: Delta state replicated data types. Journal of Parallel Distributed Computing **111**, 162–173 (2018). https://doi.org/10.1016/j.jpdc.2017.08.003
2. Bailis, P., Fekete, A., Franklin, M.J., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Feral concurrency control: An empirical investigation of modern application integrity. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. SIGMOD'15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2723372.2737784
3. Bailis, P., Fekete, A.D., Franklin, M.J., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Coordination avoidance in database systems. Proceedings of the VLDB Endowment **8**(3), 185–196 (2014). https://doi.org/10.14778/2735508.2735509
4. Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N., Najafzadeh, M., Shapiro, M.: Putting consistency back into eventual consistency. In: Proceedings of the Tenth European Conference on Computer Systems. pp. 1–16. EuroSys'15 (2015). https://doi.org/10.1145/2741948.2741972
5. Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N.M.: IPA: invariant-preserving applications for weakly consistent replicated databases. Proceedings of the VLDB Endowment **12**(4), 404–418 (2018). https://doi.org/10.14778/3297753.3297760
6. Baquero, C., Almeida, P.S., Cunha, A., Ferreira, C.: Composition in state-based replicated data types. Bulletin of European Association for Theoretical Computer Science **123** (2017), http://eatcs.org/beatcs/index.php/beatcs/article/view/507
7. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing. PODC'00 (2000). https://doi.org/10.1145/343477.343502
8. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally distributed database. ACM Transactions on Computer Systems **31**(3) (2013). https://doi.org/10.1145/2491245
9. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles. SOSP'07 (2007). https://doi.org/10.1145/1294261.1294281
10. Fox, A., Brewer, E.: Harvest, yield, and scalable tolerant systems. In: Proceedings of the Seventh Workshop on Hot Topics in Operating Systems. pp. 174–178 (Mar 1999). https://doi.org/10.1109/HOTOS.1999.798396
11. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News **33**(2), 51–59 (Jun 2002). https://doi.org/10.1145/564585.564601

12. Johnson, P.R., Thomas, R.: Maintenance of duplicate databases (Jan 1975). https://doi.org/10.17487/RFC0677
13. Kleppmann, M., Wiggins, A., van Hardenberg, P., McGranaghan, M.: Local-first software: You own your data, in spite of the cloud. In: Masuhara, H., Petricek, T. (eds.) Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. Onward! 2019 (2019). https://doi.org/10.1145/3359591.3359737
14. Kulkarni, S.S., Demirbas, M., Madappa, D., Avva, B., Leone, M.: Logical physical clocks. In: Proceedings of the 18th International Conference on Principles of Distributed Systems. OPODIS 2014 (2014). https://doi.org/10.1007/978-3-319-14472-6_2
15. Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (apr 2010). https://doi.org/10.1145/1773912.1773922
16. Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N.M., Rodrigues, R.: Making geo-replicated systems fast as possible, consistent when necessary. In: Thekkath, C., Vahdat, A. (eds.) Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation. OSDI 2012 (2012). https://doi.org/10.5555/2387880.2387906
17. Lopes, P., Sousa, J., Balegas, V., Ferreira, C., Duarte, S., Bieniusa, A., Rodrigues, R., Preguiça, N.: Antidote SQL: relaxed when possible, strict when necessary (2019), http://arxiv.org/abs/1902.03576
18. Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems. SSS 2011 (2011). https://doi.org/10.1007/978-3-642-24550-3_29
19. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (it's time for a complete rewrite). In: Proceedings of the 33rd International Conference on Very Large Data Bases. VLDB'07 (2007). https://doi.org/10.5555/1325851.1325981
20. Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., Shah, S.: Serving large-scale batch computed data with project voldemort. In: Proceedings of the 10th USENIX Conference on File and Storage Technologies. FAST'12, USENIX Association (2012). https://doi.org/10.5555/2208461.2208479
21. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: Fast distributed transactions for partitioned database systems. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. SIGMOD'12 (2012). https://doi.org/10.1145/2213836.2213838
22. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP'13 (2013). https://doi.org/10.1145/2517349.2522713
23. Yu, W., Elvinger, V., Ignat, C.L.: A generic undo support for state-based crdts. In: Felber, P., Friedman, R., Gilbert, S., Miller, A. (eds.) Proceedings of the 23rd International Conference on Principles of Distributed Systems. OPODIS 2019 (2019). https://doi.org/10.4230/LIPIcs.OPODIS.2019.14
24. Yu, W., Ignat, C.L.: Conflict-free replicated relations for multi-synchronous database management at edge. In: Proceedings of the IEEE International Conference on Smart Data Services. SMDS 2020 (2020). https://doi.org/10.1109/SMDS49396.2020.00021