Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge

Weihai Yu, Claudia-Lavinia Ignat

Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge

Weihai Yu
UIT - The Arctic University of Norway
N-9037 Tromsø, Norway
weihai.yu@uit.no

Claudia-Lavinia Ignat
Université de Lorraine, CNRS, Inria, LORIA
F-54000 Nancy, France
claudia.ignat@inria.fr

***Abstract*—In a cloud-edge environment, edge devices may not always be connected to the network. Still, applications may need to access the data on edge devices even when they are not connected. With support for multi-synchronous access, data on an edge device are kept synchronous with the data in the cloud as long as the device is online. When the device is off-line, the application can still access the data on the device, asynchronously with concurrent data updates either in the cloud or on other edge devices. Conflict-free Replicated Data Types (CRDTs) emerged as a technology for multi-synchronous data access. CRDTs guarantee that when all sites have applied the same set of updates, the replicated data converge. However, CRDTs have not been successfully applied to relational databases (RDBs) for multi-synchronous access. In this paper, we present Conflict-free Replicated Relations (CRRs) that apply CRDTs to RDBs for support of multi-synchronous data access. With CRR, existing RDB applications, with very little modification, can be enhanced with multi-synchronous access. We also present a prototype implementation of CRR with some preliminary performance results.**

***Index Terms*—CRDT; relational database; eventual consistency; integrity constraints**

I. Introduction

The cloud technology makes data globally shareable and accessible, as long as the end-user's device is connected to the network. When the user's device is not connected, the data become inaccessible.

There are scenarios where the users may want to access their data even when their devices are off-line. The researchers may want to access their research data when they are at field work without any network facility. The project managers and engineers may want to access project data when they are under extreme conditions such as inside a tunnel or in the ocean. People may want to use their personal applications during flight, in a foreign country or during mountain hiking.

In a cloud-edge environment, the cloud consists of servers with powerful computation and reliable data storage capacities. The edge consists of the devices outside the operation domain of the cloud. The above-mentioned scenarios suggest *multi-synchronous* access of data on edge devices. That is, there are two modes of accessing data on edge devices: *asynchronous mode*—the user can always access (read and write) the data on the device, even when the device is off-line, and *synchronous mode*—the data on the device is kept synchronous with the data stored in the cloud, as long as the device is online.

One of the main challenges of multi-synchronous data access is the limitation of a networked system stated in the CAP theorem [1,2]: it is impossible to simultaneously ensure all three desirable properties, namely (C) consistency equivalent to a single up-to-date copy of data, (A) availability of the data for update and (P) tolerance to network partition.

CRDTs, or Conflict-free Replicated Data Types, emerged to address the CAP challenges [3]. 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 (referred to as *strong eventual consistency* in [3]). CRDTs have been adopted in the construction of distributed key-value stores [4], collaborative editors [5]–[8] and local-first software [9,10]. All these bear the similar goal as multi-synchronous data access. There has also been active research on CRDT-based transaction processing [11] mainly to achieve low latency for geo-replicated data stores.

Despite the above success stories, CRDTs have not been applied to multi-synchronous access of data stored in relational databases (RDBs). To support multi-synchronous data access, the application typically has to be implemented using specific key-value stores built with CRDTs. These key-value stores do not support some important RDB features, including advanced queries and integrity constraints. Consequently, the large number of existing applications that use RDBs have to be re-implemented for multi-synchronous access.

In this paper, we propose Conflict-Free Replicated Relation (CRR) for multi-synchronous access to RDB data. Our contributions include:

- A set CRDT for RDBs. One of the hurdles for adopting CRDTs to RDBs is that there is no appropriate set CRDT for RDBs (Section III-A). We present CLSet CRDT (causal-length set) that is particularly suitable for RDBs (Section III-C).
- An augmentation of existing RDB schema with CRR in a two-layer system (Section IV). With CRR, we are able to enhance existing RDB applications with very little modification.
- Automatic handling of integrity violations at merge (Section V).
- A prototype implementation (Section VI). The implementation is independent of the underlying database management system (DBMS). This allows different devices of

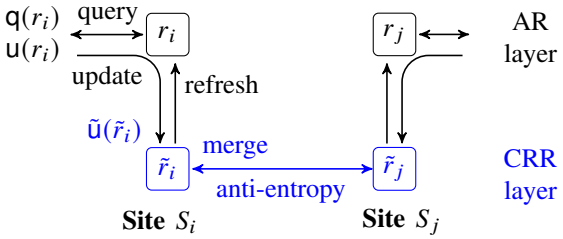

Fig. 1. A two-layer relational database system

the same application to deploy different DBMSs. This is particularly useful in a cloud-edge environment that is inherently heterogeneous. We also report some initial performance result of the prototype.

The paper is organized as follows. Section II gives an overview of CRR, together with the system model our work applies to. Section III reviews the background of CRDT and presents the CLSet CRDT that is an underlying CRDT of CRR. Section IV describes CRR in detail. Section VI presents an implementation of CRR and some preliminary performance results. Section VII discusses related work. Section VIII concludes.

II. Overview of CRR

The RDB supporting CRR consists of two layers: an Application Relation (AR) layer and a Conflict-free Replicated Relation (CRR) layer (see Figure 1). The AR layer presents the same RDB schema and API as a conventional RDB system. Application programs interact with the database at the AR layer. The CRR layer supports conflict-free replication of relations.

The AR-layer database schema R has an augmented CRR schema $\tilde{R}$. A site S_i maintains both an instance r_i of R and an instance $\tilde{r}_i$ of $\tilde{R}$. A query q on r_i is performed directly on r_i. A request for update u on r_i is translated into $\tilde{\mathsf{u}}$ and performed on the augmented relation instance $\tilde{r}_i$. The update is later propagated to remote sites through an anti-entropy protocol. Every update in $\tilde{r}_i$, either as the execution of a local update $\tilde{\mathsf{u}}(\tilde{r}_i)$ or as the merge with a remote update $\tilde{\mathsf{u}}(\tilde{r}_j)$, refreshes r_i.

CRR has the property that when both sites S_i and S_j have applied the same set of updates, the relation instances at the two sites are equivalent, i.e. $r_i = r_j$ and $\tilde{r}_i = \tilde{r}_j$.

The two-layered system also maintains the integrity constraints defined at the AR layer. Any violation of integrity constraint is caught at the AR layer. A local update of $\tilde{r}_i$ and refresh of r_i are wrapped in an atomic transaction: a violation would cause the rollback of the transaction. A merge at $\tilde{r}_i$ and a refresh at r_i is also wrapped in a transaction: a failed merge would cause some compensation updates.

An application developer does not need to care about how the CRR layer works. Little change is needed for an existing RDB application to function with CRR support.

We use different CRDTs for CRRs. Since a relation instance is a set of tuples or rows, we use a set CRDT for relation instances. We design a new set CRDT (Section III), as none of existing set CRDTs are suitable for CRRs (Section III-A). A row consists of a number of attributes. Each of the attributes can be individually updated. We use the LWW (last-write wins) register CRDT [12,13] for attributes in general cases. Since LWW registers may lose some concurrent updates, we use the counter CRDT [3] for numeric attributes with additive updates.

System Model

A distributed system consists of sites with globally unique identifiers. Sites do not share memory. They maintain durable states. Sites may crash, but will eventually recover to the durable state at the time of the last crash.

A site can send messages to any other site in the system through an asynchronous and unreliable network. There is no upper bound on message delay. The network may discard, reorder or duplicate messages, but it cannot corrupt messages. Through re-sending, messages will eventually be delivered. The implication is that there can be network partitions, but disconnected sites will eventually get connected.

III. A Set CRDT for CRRs

In this section, we first present a background of CRDTs. As a relation instance is a set of tuples, CRR needs an appropriate set CRDT as a building block. We present limitations of existing set CRDTs for RDBs. We then describe a new set CRDT that addresses the limitations.

A. CRDT Background

A CRDT 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 is 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. To apply the updates made at the other sites, a site merges the received updates with its local replica. A CRDT has the property that when all sites have applied the same set of updates, the replicas converge.

There are two families of CRDT approaches, namely operation-based and state-based [3]. Our work is based on state-based CRDTs, where a message for updates consists of the data state of a replica in its entirety. A site applies the updates by merging its local state with the state in the received message. The possible states of a state-based CRDT must form a join-semilattice [14], which implies convergence. Briefly, the states form a *join-semilattice* if they are partially ordered with $\sqsubseteq$ and a join $\sqcup$ 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 is the result of a join.

Figure 2 (left) shows GSet, a state-based CRDT for grow-only sets (or add-only set [3]), where E is a set of possible elements, $\sqsubseteq \stackrel{\text{def}}{=} \subseteq$, $\sqcup \stackrel{\text{def}}{=} \cup$, insert is a mutator (update operation) and in is a query. Obviously, an update through $\mathsf{insert}(s, e)$ is an inflation, because $s \subseteq \{e\} \cup s$. Figure 2 (right) shows

$$\mathsf{GSet}(E) \stackrel{\text{def}}{=} \mathcal{P}(E)$$
$$\mathsf{insert}(s, e) \stackrel{\text{def}}{=} \{e\} \cup s$$
$$\mathsf{insert}^{\delta}(s, e) \stackrel{\text{def}}{=} \begin{cases} \{e\} & \text{if } e \notin s \\ \{\} & \text{otherwise} \end{cases}$$
$$s \sqcup s' \stackrel{\text{def}}{=} s \cup s'$$
$$\mathsf{in}(s, e) \stackrel{\text{def}}{=} e \in s$$

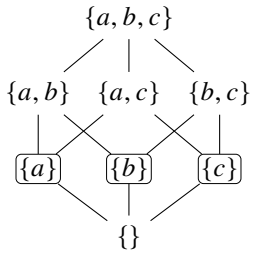

Fig. 2. GSet CRDT and Hasse diagram of states

the Hasse diagram of the states in a GSet. A Hasse diagram shows only the "direct links" between states.

GSet is an example of an *anonymous* CRDT, since its operations are not specific to the sites that perform the operations. Two concurrent executions of the same mutation, such as $\mathsf{insert}(\{\}, a)$, fulfill the same purpose.

Using state-based CRDTs, as originally presented [3], is costly in practice, because states in their entirety are sent as messages. Delta-state CRDTs address this issue by only sending join-irreducible states [15,16]. 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). In Figure 2, insert^{δ} is a delta-mutator that returns join-irreducible states which are singleton sets (boxed in the Hasse diagram).

Since a relation instance is a set of tuples, the basic building block of CRR is a set CRDT, or specifically, a delta-state set CRDT.

A CRDT for general-purpose sets with both insertion and deletion updates can be designed as a causal CRDT [15] such as ORSet (observed-remove set [12,17,18]). Basically, every element is associated with at least a causal context as meta data. A causal context is a set of event identifiers (typically a pair of a site identifier and a site-specific sequence number). An insertion or deletion is achieved with inflation of the associated causal context. Using causal contexts, we are able to tell explicitly which insertions of an element have been later deleted. However, maintaining causal contexts for every element can be costly, even though it is possible to compress causal contexts into vector states, for instance under causal consistency. Causal CRDTs are not suitable for RDBs, which generally do not allow multi-valued attributes such as causal contexts.

In the following, we present a new set CRDT, based on the abstraction of causal length. It is a specialization of our earlier work on generic undo support for state-based CRDTs [19].

B. Causal length

The key issue that a general-purpose set CRDT must address is how to identify the causality between the different insertion and deletion updates. We achieve this with the abstraction of causal length, which is based on two observations.

First, the insertions and deletions of a given element occur in turns, one causally dependent on the other. A deletion is an

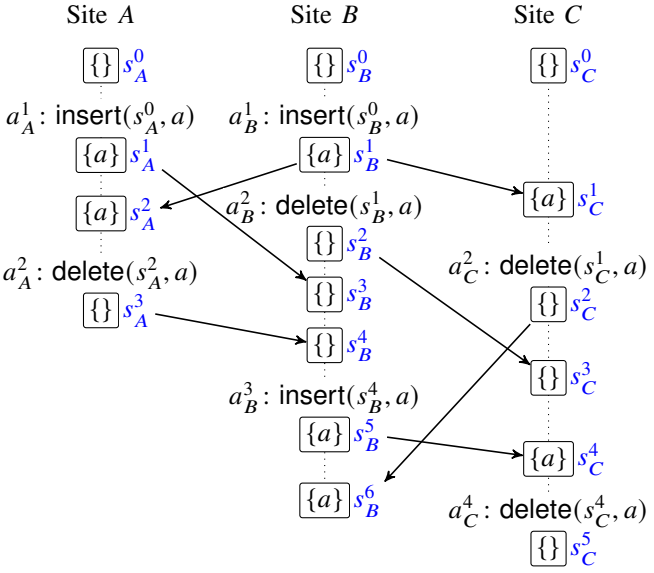

Fig. 3. A scenario of concurrent set updates

inverse of the last insertion it sees. Similarly, an insertion is an inversion of the last deletion it sees (or none, if the element has never been inserted).

Second, two concurrent executions of the same mutation of an anonymous CRDT (Section III-A) fulfill the same purpose and therefore are regarded as the same update. Seeing one means seeing both (such as the concurrent insertions of the same element in GSet). Two concurrent inverses of the same update are also regarded as the same one.

Figure 3 shows a scenario where three sites A, B and C concurrently insert and delete element a. When sites A and B concurrently insert a for the first time, with updates a_A^1 and a_B^1, they achieve the same effect. Seeing either one of the updates is the same as seeing both. Consequently, states s_A^1, s_A^2, s_B^1 and s_C^1 are equivalent as far as the insertion of a is concerned.

Following the same logic, the concurrent deletions on these equivalent states (with respect to the first insertion of a) are also regarded as achieving the same effect. Seeing one of them is the same as seeing all. Therefore, states s_A^3, s_B^2, s_B^3, s_B^4, s_C^2 and s_C^3 are equivalent with regard to the deletion of a.

Now we present the states of element a as the equivalence classes of the updates, as shown in the second column of Table I. The concurrent updates that see equivalent states and achieve the same effect are in the same equivalence classes. The columns r and $\tilde{r}$ in Table I correspond to the states of a tuple (as an element of a set) in relation instances in the AR and CRR layers (Section II).

When a site makes a new local update, it adds to its state a new equivalence class that contains only the new update. For example, when site B inserts element a at state s_B^0 with update a_B^1, it adds a new equivalence class $\{a_B^1\}$ and the new state s_B^1 becomes $\{\{a_B^1\}\}$. When site B then deletes the element with update a_B^2, it adds a new equivalence class $\{a_B^2\}$ and the new

TABLE I
STATES OF A SET ELEMENT

s	states in terms of equivalence groups	$\tilde{r}$	r
s_A^0	$\{\}$	$\{\}$	$\{\}$
s_A^1	$\{\{a_A^1\}\}$	$\{\langle a,1\rangle\}$	$\{a\}$
s_A^2	$\{\{a_A^1, a_B^1\}\}$	$\{\langle a,1\rangle\}$	$\{a\}$
s_A^3	$\{\{a_A^1, a_B^1\}, \{a_A^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_B^0	$\{\}$	$\{\}$	$\{\}$
s_B^1	$\{\{a_B^1\}\}$	$\{\langle a,1\rangle\}$	$\{a\}$
s_B^2	$\{\{a_B^1\}, \{a_B^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_B^3	$\{\{a_A^1, a_B^1\}, \{a_B^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_B^4	$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_B^5	$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2\}, \{a_B^3\}\}$	$\{\langle a,3\rangle\}$	$\{a\}$
s_B^6	$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2, a_C^2\}, \{a_B^3\}\}$	$\{\langle a,3\rangle\}$	$\{a\}$
s_C^0	$\{\}$	$\{\}$	$\{\}$
s_C^1	$\{\{a_B^1\}\}$	$\{\langle a,1\rangle\}$	$\{a\}$
s_C^2	$\{\{a_B^1\}, \{a_C^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_C^3	$\{\{a_B^1\}, \{a_B^2, a_C^2\}\}$	$\{\langle a,2\rangle\}$	$\{\}$
s_C^4	$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2, a_C^2\}, \{a_B^3\}\}$	$\{\langle a,3\rangle\}$	$\{a\}$
s_C^5	$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2, a_C^2\}, \{a_B^3\}, \{a_C^4\}\}$	$\{\langle a,4\rangle\}$	$\{\}$

state s_B^2 becomes $\{\{a_B^1\}, \{a_B^2\}\}$.

The merge of two states is handled as the union of the equivalence classes. For example, when states s_A^1 and s_B^2 merge, the new state s_B^3 becomes $\{\{a_A^1\} \cup \{a_B^1\}, \{a_B^2\}\} = \{\{a_A^1, a_B^1\}, \{a_B^2\}\}$.

Now, observe that whether element a is in the set is determined by the number of equivalence classes, rather than the specific updates contained in the equivalence classes. For example, as there is one equivalence class in state s_B^1, the element a is in the set. As there are two equivalence classes in states s_B^2, s_B^3 and s_B^4, the element a is not in the set.

Due to the last observation, we can represent the states of an element with a single number, the number of equivalence classes. We call that number the *causal length* of the element. An element is in the set when its causal length is an odd number. The element is not in the set when its causal length is an even number. As shown in Table I, a CRR-layer relation $\tilde{r}$ augments an AR-later relation r with causal lengths.

C. CLSet CRDT

Figure 4 shows the CLSet CRDT. The states are a partial function $s\colon E \hookrightarrow \mathbb{N}$, meaning that when e is not in the domain of s, $s(e) = 0$ (0 is the bottom element of $\mathbb{N}$, i.e. $\bot_{\mathbb{N}} = 0$). Using partial function conveniently simplifies the specification of insert, $\sqcup$ and in. Without explicit initialization, the causal length of any unknown element is 0. In the figure, insert^δ and delete^δ are the delta-counterparts of insert and delete respectively.

An element e is 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 insertion or deletion

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

$$\mathsf{insert}(s,e) \stackrel{\text{def}}{=} \begin{cases} s\{e \mapsto s(e)+1\} & \text{if } \mathsf{even}(s(e)) \\ s & \text{if } \mathsf{odd}(s(e)) \end{cases}$$

$$\mathsf{insert}^\delta(s,e) \stackrel{\text{def}}{=} \begin{cases} \{e \mapsto s(e)+1\} & \text{if } \mathsf{even}(s(e)) \\ \{\} & \text{if } \mathsf{odd}(s(e)) \end{cases}$$

$$\mathsf{delete}(s,e) \stackrel{\text{def}}{=} \begin{cases} s & \text{if } \mathsf{even}(s(e)) \\ s\{e \mapsto s(e)+1\} & \text{if } \mathsf{odd}(s(e)) \end{cases}$$

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

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

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

Fig. 4. CLSet CRDT

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'.

IV. Conflict-free Replicated Relations

In this section we describe the design of the CRR layer. We focus particularly on the handling of updates.

Without loss of generality, we represent the schema of an application-layer relation with $R(K, A)$, where R is the relation name, K is the primary key and A is a non-key attribute. For a relation instance r of $R(K, A)$, we use $A(r)$ for the projection $\pi_A r$. We also use $r(k)$ for the row identified with key k, and (k, a) for row $r(k)$ whose A attribute has value a.

A. Augmentation and caching

In a two-layer system as highlighted in Section II, we *augment* an AR-layer relation schema $R(K, A)$ to a CRR-layer schema $\tilde{R}(K, A, T_A, L)$ where T_A is the update timestamps of attribute A and L is the causal lengths of rows. Basically, $\tilde{R}(K, L)$ implements a CLSet CRDT (Section III-C) where the rows identified by keys are elements, and $\tilde{R}(K, A, T_A)$ implements the LWW-register CRDT [12,13] where an attribute of each row is a register.

We use hybrid logical clock [20] for timestamps, which is UTC time compatible and for two events e_1 and e_2 with clocks τ_1 and τ_2, $\tau_1 < \tau_2$ if e_1 happens before e_2.

For a relation instance $\tilde{r}$ of an augmented schema $\tilde{R}$, the relation instance r of the AR-layer schema R is a *cache* of $\tilde{r}$.

For a relation operation op on r, we use $\widetilde{\mathsf{op}}$ as the corresponding operation on $\tilde{r}$. For example, we use $\tilde{\in}$ on $\tilde{r}$ to detect whether a row exists in r. That is $\tilde{r}(k) \,\tilde{\in}\, \tilde{r} \Leftrightarrow r(k) \in r$.

According to CLSet (Figure 4), we define $\tilde{\in}$ and $\tilde{\notin}$ as

$$\tilde{r}(k) \,\tilde{\in}\, \tilde{r} \stackrel{\text{def}}{=} \tilde{r}(k) \in \tilde{r} \wedge \mathsf{odd}(L(\tilde{r}(k)))$$

$$\tilde{r}(k) \,\tilde{\notin}\, \tilde{r} \stackrel{\text{def}}{=} \tilde{r}(k) \notin \tilde{r} \vee \mathsf{even}(L(\tilde{r}(k)))$$

For an update operation $\mathsf{u}(r, k, a)$ on r, we first perform $\tilde{\mathsf{u}}(\tilde{r}, k, a, \tau_A, l)$ on $\tilde{r}$. This results in the new instance $\tilde{r}'$ and row $\tilde{r}'(k) = (k, a, \tau'_A, l')$. We then refresh the cache r as the following:

- Insert (k, a) into r if $\tilde{r}'(k) \tilde{\in} \tilde{r}'$ and $r(k) \notin r$.
- Delete $r(k)$ from r if $\tilde{r}'(k) \tilde{\notin} \tilde{r}'$ and $r(k) \in r$.
- Update $r(k)$ with (k, a) if $\tilde{r}'(k) \tilde{\in} \tilde{r}'$, $r(k) \in r$ and $A\big(r(k)\big) \neq a$.

All AR-layer queries are performed on the cached instance r without any involvement of the CRR layer.

The following subsections describe how the CRR layer handles the different update operations.

B. Update operations

The CRR layer handles a local row insertion as below:

$$\widetilde{\mathsf{insert}}(\tilde{r}, (k, a)) \stackrel{\text{def}}{=} \begin{cases} \mathsf{insert}\big(\tilde{r}, (k, a, \mathsf{now}, 1)\big) & \text{if } \tilde{r}(k) \notin \tilde{r} \\ \mathsf{update}\big(\tilde{r}, k, (a, \mathsf{now}, L(\tilde{r}(k)) + 1)\big) & \text{if } \tilde{r}(k) \in \tilde{r} \\ & \quad \wedge \tilde{r}(k) \tilde{\notin} \tilde{r} \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

A row insertion attempts to achieve two effects: to insert row $r(k)$ into r and to assign value a to attribute A of $r(a)$. If $r(k)$ has never been inserted, we simply insert $(k, a, \mathsf{now}, 1)$ into $\tilde{r}$. If $r(k)$ has been inserted but later deleted, we re-inserted it with the causal length incremented by 1. Otherwise, there is already a row $r(k)$ in r, thus we do nothing.

There could be an alternative handling in the last case. Instead of doing nothing, we could update the value of attribute A with a. We choose to do nothing, because this behavior is in line with the SQL convention.

The CRR layer handles a local row deletion as below:

$$\widetilde{\mathsf{delete}}(\tilde{r}, k) \stackrel{\text{def}}{=} \begin{cases} \mathsf{update}\big(\tilde{r}, k, (-, L(\tilde{r}(k)) + 1)\big) & \text{if } \tilde{r}(k) \tilde{\in} \tilde{r} \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

If there is a row $r(k)$ in r, we increment the causal length by 1. We do nothing otherwise. In the expression, we use the "–" sign for the irrelevant attributes.

The CRR layer handles a local attribute update as below:

$$\widetilde{\mathsf{update}}(\tilde{r}, k, a) \stackrel{\text{def}}{=} \begin{cases} \mathsf{update}\big(\tilde{r}, k, (a, \mathsf{now}, L(\tilde{r}(k)))\big) & \text{if } \tilde{r}(k) \tilde{\in} \tilde{r} \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

If there is a row $r(k)$ in r, we update the attribute and set a new timestamp. We do nothing otherwise.

When the CRR layer handles one of the above update operations and results in a new row $\tilde{r}(k)$ in $\tilde{r}$ (either inserted or updated), the relation instance with a single row $\{\tilde{r}(k)\}$ is a join-irreducible state in the possible instances of schema $\tilde{R}$. The CRR layer later sends $\{\tilde{r}(k)\}$ to the remote sites in the anti-entropy protocol.

A site merges a received join-irreducible state $\{(k, a, \tau, l)\}$ with its current state $\tilde{r}$ using a join:

$$\tilde{r} \sqcup \{(k, a, \tau, l)\} \stackrel{\text{def}}{=} \begin{cases} \mathsf{insert}\big(\tilde{r}, (k, a, \tau, l)\big) & \text{if } \tilde{r}(k) \notin \tilde{r} \\ \mathsf{update}\big(\tilde{r}, k, (a, \mathsf{max}(T(\tilde{r}), \tau), & \text{if } L(\tilde{r}) < l \\ \qquad \mathsf{max}(L(\tilde{r}), l))\big) & \quad \vee T(\tilde{r}) < \tau \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

If there is no $\tilde{r}(k)$ in $\tilde{r}$, we insert the received row into $\tilde{r}$. If the received row has either a longer causal length or a newer timestamp, we update $\tilde{r}$ with the received row. Otherwise, we keep $\tilde{r}$ unchanged. Notice that a newer update on a row that is concurrently deleted is still updated. The update is therefore not lost. The row will have the latest updated attribute value when the row deletion is later undone.

It is easy to verify that a new relation instance, resulted either from one of the local updates or from a merge with a received state, is always an inflation of the current relation instance regarding causal lengths and timestamps.

C. Counters

The general handling of attribute updates described earlier in Section IV-B is based on the LWW-register CRDT. This is not ideal, because the effect of some concurrent updates may get lost. For some numeric attributes, we can use the counter CRDT [3] to handle updates as increments and decrements.

We make a special (meta) relation *Ct* for counter CRDTs at the CRR layer: $Ct(Rel, Attr, Key, Sid, Icr, Dcr)$, where $(Rel, Attr, Key, Sid)$ is a candidate key of Ct.

The attribute *Rel* is for relation names, *Attr* for names of numeric attributes and *Key* for primary key values. For an AR-layer relation $R(K, A)$, and respectively the CRR-layer relation $\tilde{R}(K, A, T_A, L)$, where K is a primary key and A is a numeric attribute, (R, A, k) of Ct refers to the numeric value $A\big(r(k)\big)$ (and $A\big(\tilde{r}(k)\big)$).

The attribute *Sid* is for site identifiers. Our system model requires that sites are uniquely identified (Section II). The counter CRDT is *named* (as opposed to anonymous, described in Section III-A), meaning that a site can only update its specific part of the data structure.

The attributes *Icr* and *Dcr* are for the increment and decrement of numeric attributes from their initial values, set by given sites.

We set the initial value of a numeric attribute to: the default value, if the attribute is defined with a default value; the lower bound, if there is an integrity constraint that specifies a lower bound of the attribute; 0 otherwise.

If the initial value of the numeric attribute A of an AR-layer relation $R(K, A)$ is v_0, we can calculate the current value of $A\big(r(k)\big)$ as $v_0 + \mathsf{sum}(Icr) - \mathsf{sum}(Dcr)$, as an aggregation on the rows identified by (R, A, k) in the current instance of Ct.

We can translate a local update of a numeric attribute into an increment or decrement operation. If the current value of the attribute is v and the new updated value is v', the update is an increment of $v' - v$ if $v' > v$, or a decrement of $v - v'$ if $v' < v$.

Site s with the current Ct instance ct handles a local increment or decrement operation as below:

$$\widetilde{\mathsf{inc}}_s(\tilde{r}, A, k, v) \stackrel{\text{def}}{=}$$
$$\begin{cases} \mathsf{update}\big(ct, R, A, k, s, (i+v, d)\big) & \text{if } ct(R, A, k, s, i, d) \in ct \\ \mathsf{insert}\big(ct, (R, A, k, s, (v, 0))\big) & \text{otherwise} \end{cases}$$

$$\widetilde{\mathsf{dec}}_s(\tilde{r}, A, k, v) \stackrel{\text{def}}{=}$$
$$\begin{cases} \mathsf{update}\big(ct, R, A, k, s, (i, d+v)\big) & \text{if } ct(R, A, k, s, i, d) \in ct \\ \mathsf{insert}\big(ct, (R, A, k, s, (0, v))\big) & \text{otherwise} \end{cases}$$

If it is the first time the site updates the attribute, we insert a new row into Ct. Otherwise, we set the new increment or decrement value accordingly. Note that $v > 0$ and the updates in Ct are always inflationary. The relation consisting of a single row of the update is a join-irreducible state in the possible instance of Ct and is later sent to remote sites.

A site with the current Ct instance ct merges a received join-irreducible state with a join:

$$ct \sqcup \{(R, C, k, s, i, d)\}$$
$$\stackrel{\text{def}}{=} \begin{cases} \mathsf{insert}(ct, (R, C, k, s, i, d)) & \text{if } ct(R, C, k, s) \notin ct \\ \mathsf{update}\big(ct, R, C, k, s, & \text{if } (R, C, k, s, i', d') \in ct \\ \quad (\mathsf{max}(i, i'), \mathsf{max}(d, d'))\big) & \quad \wedge (i' < i \vee d' < d) \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

If the site has not seen any update from site s, we insert the received row into ct. If it has applied some update from s and the received update makes an inflation, we update the corresponding row for site s. Otherwise, it has already applied a later update from s and we keep ct unchanged.

Notice that turning an update into an increment or decrement might not always be an appropriate way to handle an update. For example, two sites may concurrently update the temperature measurement from $11°C$ to $15°C$. Increasing the value twice leads to a wrong value $19°C$. In such situations, it is more appropriate to handle updates as a LWW-register.

V. Integrity constraints

Applications define integrity constraints at AR layer. A DBMS detects violations of integrity constraints when we refresh the cached AR-layer relations.

We perform updates on a CRR-layer instance $\tilde{r}$ and the corresponding refresh of an AR-layer instance r in a single atomic transaction. A violation of any integrity constraint causes the entire transaction to be rolled back. Therefore a local update that violates any integrity constraint has no effect on either $\tilde{r}$ or r.

When a site detects a violation at merge, it may perform an undo on an offending update. We first describe how to perform an undo. Then, we present certain rules to decide which updates to undo. The purpose of making such decisions is to avoid undesirable effects. In more general cases not described below, we simply undo the incoming updates that cause constraint violations, although this may result in unnecessary undo of too many updates.

A. Undoing an update

For delta-state CRDTs, a site sends join-irreducible states as remote updates. In our case, a remote update is a single row (or more strictly, a relation instance containing the row).

To undo a row insertion or deletion, we simply increment the causal length by one.

To undo an attribute update augmented with the LWW-register CRDT, we set the attribute with the old value, using the current clock value as the timestamp. In order to be able to generate the undo update, the message of an attribute update also includes the old value of the attribute.

We handle undo of counter updates with more care, because counter updates are not idempotent and multiple sites may perform the same inverse operation concurrently. As a result, the same increment (or decrement) might be mistakenly reversed multiple times.

To address this problem, we create a new (meta) relation for counter undo: $CtU(Rel, Attr, Key, Sid, T)$.

Similar to relation Ct (Section IV-C), attributes Rel and $Attr$ are for the names of the relations and the counter attributes, and attributes Key and Sid are for key values and sites that originated the update. T is the timestamp of the original counter update. A row (R, A, k, s, τ) in CtU uniquely identifies a counter update.

A message for a counter update contains the timestamp and the delta of the update, in addition to the join-irreducible state of the update. The timestamp helps us uniquely identify the update. The delta helps us generate the inverse update.

Relation CtU works as a GSet (Figure 2). When a site undoes a counter update, it inserts a row identifying the update into CtU. When a site receives an undo message, it performs the undo only when the row of the update is not in CtU.

B. Uniqueness

An application may set a uniqueness constraint on an attribute (or a set of attributes). For instance, the email address of a registered user must be unique. Two concurrent updates may violate a uniqueness constraint, though none of them violates the constraint locally.

When we detect the violation of a uniqueness constraint at the time of merge, we decide a winning update and undo the losing one. Following the common sequential cases, an earlier update wins. That is, the update with a smaller timestamp wins.

As violations of uniqueness constraints occur only for row insertions and attribute updates (as LWW-registers), the only possible undo operations are row deletions and attribute updates (as LWW-registers).

C. Reference integrity

A reference-integrity constraint may be violated due to concurrent updates of a reference and the referenced row. Suppose relation $R^1(K^1, F)$ has a foreign key F referencing to $R^2(K^2, A)$. Assume two sites S_1 and S_2 have rows $r_1^2(k^2)$ and $r_2^2(k^2)$ respectively. Site S_1 inserts $r_1^1(k^1, k^2)$ and site S_2 concurrently deletes $r_2^2(k^2)$. The sites will fail to merge the concurrent remote updates, because they violate the reference-integrity constraint.

Obviously, both sites undoing the incoming remote update is undesirable. We choose to undo the updates that add references. The updates are likely row insertions. One reason of this choice is that handling uniqueness violations often leads to row deletions (Section V-B). If a row deletion as undo for a uniqueness violation conflicts with an update that adds a reference, undoing the row deletion will violate the uniqueness constraint again, resulting in an endless cycle of undo updates.

Notice that two seemingly concurrent updates may not be truly concurrent. One site might have already indirectly seen the effect of the remote update, reflected as a longer causal length. Two updates are truly concurrent only when the referenced rows at the two sites have the same causal length at the time of the concurrent updates. When the two updates are not truly concurrent, a merge of an incoming update at one of the sites would have no effect. In the above example, the merge of the deletion at site S_2 has no effect if $L(\tilde{r}_1^2(k^2)) > L(\tilde{r}_2^2(k^2))$.

D. Numeric constraints

An application may set numeric constraints, such as a lower bound of the balance of a bank account. A set of concurrent updates may together violate a numeric constraint, though the individual local updates do not.

When the merge of an incoming update fails due to a violation of a numeric constraint, we undo the latest update (or updates) of the numeric attribute. This, however, may undo too many updates than necessary. We have not yet investigated how to avoid undoing too many updates that violate a numeric constraint.

VI. Implementation

We have implemented a CRR prototype, Crecto, on top of Ecto[1], a data mapping library for Elixir[2].

For every application-defined database table, we generate a CRR-augmented table. We also generate two additional tables in Crecto, Ct for counters (Section IV-C) and CtU for counter undo (Section V-A).

For every table, Ecto automatically generates a primary-key attribute which defaults to an auto-increment integer. Instead, we enforce the primary keys to be UUIDs to avoid collision of key values generated at different sites.

In the CRR-augmented tables, every attribute has an associated timestamp for the last update. We implemented the hybrid logic clock [20] as timestamps.

In Ecto, applications interact with databases through a single Repo module. This allows us to localize our implementation efforts to a Crecto.Repo module on top of the Ecto.Repo module.

Since queries do not involve any CRR-layer relations, Crecto.Repo forwards all queries unchanged to Ecto.Repo.

A local update includes first update(s) of CRR-layer relation(s) and then a refresh of the cached relation at AR layer. All these are wrapped in a single atomic transaction. Any constraint violation is caught at cache refreshment, which causes the transaction to be rolled back.

A site keeps outgoing messages in a queue. When the site is online, it sends the messages in the queue to remote sites. The queue is stored in a file so that it survives system crashes.

A merge of an incoming update includes also first update(s) of CRR-layer relation(s) and then a refresh of the cached relation at AR layer. These are also wrapped in an atomic transaction. Again, any constraint violation is caught at cache refreshment, which causes the transaction to be rolled back. When this happens, we undo an offending update (Section V). If we undo the incoming update, we generate an undo update, insert an entry in CtU if the update is a counter increment or decrement, and send the generated undo update to remote sites. If we undo an update that has already been performed locally, we re-merge the incoming update after the undo of the already performed update.

For an incoming remote update, we can generate the undo update using the additional information attached in the message (Section V-A). For an already performed row insertion or deletion, we can generate the undo update with an incremental of the causal length. For an already performed attribute update, we can generate the undo update in one of two ways. We can use the messages stored in the queue. Or we can simply wait. An undo update will eventually arrive, because this update will violate the same constraint at a remote site.

Performance

To study the performance of CRR, we implemented an advertisement-counter application [18,21] in Crecto. Advertisements are displayed on edge devices (such as mobile phones) according to some vendor contract, even when the devices are not online. The local counters of displayed advertisements are merged upstream in the cloud when the devices are online. An advertisement is disabled when it has reached a certain level of impression (total displayed number).

We are mainly interested in the latency of database updates, which is primarily dependent on disk IOs. Table II shows the latency measured at a Lenovo ThinkPad T540p laptop, configured with an Intel quad-core i7-4710MQ CPU, 16 GiB Ram, 512 GiB SSD, running Linux 5.4. The Crecto prototype is implemented on Ecto 3.2 in Elixir 1.9.4 OTP 22, deployed with PostgreSQL[3] 12.1.

To understand the extra overhead of multi-synchronous support, we measured the latency of the updates of the application

[1] https://github/elixir-ecto/ecto

[2] https://elixir-lang.org

[3] https://www.postgresql.org

TABLE II
LATENCY OF DATABASE UPDATES (IN MS)

Ecto	insertion	deletion	update	
No-tx	0.9	0.9	0.9	
In-tx	1.5	1.5	1.5	
Crecto	insertion	deletion	lww	counter
Local	2.1	2.1	2.1	2.8
Merge	2.1	2.1	2.1	2.8

implemented in both Ecto and Crecto. Table II only gives an approximate indication, since the latency depends on the load of the system as well as the sizes of the tables. For example, the measured latency of counter updates actually varied from 1.1ms to 3.8ms to get an average of 2.8ms. Notice also that whether an update is wrapped in a transaction also makes a difference. We thus include the measured latency of Ecto updates that are either included (In-tx) or not included (No-tx) in transactions.

We expected that the latency of the updates in Crecto would be 2–3 times higher than that of the corresponding updates in Ecto, since handling an update request involves two or three updates in the two layers. The measured latency is less than that. One explanation is that several updates may share some common data-mapping overhead.

VII. RELATED WORK

CRDTs for multi-synchronous data access

One important feature of local-first software [10] is multi-synchronous data access. In particular, data should be first stored on user devices and be always immediately accessible. Current prototypes of local-first software reported in [10] are implemented using JSON CRDTs [9]. There is no support for RDB features such as SQL-like queries and integrity constraints.

The work presented in [22] aims at low latency of data access at edge. It uses hybrid logical clock [20] to detect concurrent operations and operational transformation [23] to resolve conflicts. In particular, it resolves conflicting attribute updates as LWW-registers. It requires a central server in the cloud to disseminate operations and the data model is restricted to the MangoDB[4] model.

Lasp [18,21] is a programming model for large-scale distributed programming. One of its key features is to allow local copies of replicated state to change during periods without network connectivity. It is based on the ORSet CRDT [12] and provides functional programming primitives such as map, filter, product and union.

CRDTs for geo-replicated data stores

Researchers have applied CRDTs to achieve low latency for data access in geo-replicated databases. Unlike multi-synchronous access, such systems require all or a quorum of the replicas to be available.

[4]https://www.mangodb.com

RedBlue consistency [11] allows eventual consistency for blue operations and guarantees strong consistency for red operations. In particular, it applies CRDTs for blue operations and globally serializes red operations.

[24] presents an approach to support global boundary constraints for counter CRDTs. For a bounded counter, the "rights" to increment and decrement the counter value are allocated to the replicas. A replica can only update the counter using its allocated right. It can "borrow" rights from other replicas when its allocated right is insufficient. This approach requires that replicas with sufficient rights be available in order to successfully perform an update.

Our work does not support global transactions [11] or global boundary constraints [24]. It is known [25] that it is impossible to support certain global constraints without synchronization and coordination among different sites. We repair asynchronously temporary violation of global constraints through undo. Support of undo for row insertion and deletion in CRR is straightforward, as CLSet (Section III-C) is a specialization of a generic undo support for CRDTs [19].

AntidoteSQL [26] provides a SQL interface to a geo-replicated key-value store that adopts CRDT approaches including those reported in [11,24]. An application can declare certain rules for resolving conflicting updates. Similar to our work, an application can choose between LWW-register and counter CRDTs for conflicting attribute updates. In AntidoteSQL, a deleted record is associated with a "deleted" flag, hence excluding the possibility for the record to be inserted back again. As AntidoteSQL is built on a key-value store, it does not benefit from the maturity of the RDB industry.

Set CRDTs

The CLSet CRDT (Section III) is based on our earlier work on undo support for state-based CRDTs [19]. Obviously, deletion (insertion) of an element can be regarded as an undo of a previous insertion (deletion) of the element.

As discussed in Section III-A, there exist CRDTs for general-purpose sets ([12,15,17,18]). The meta data of existing set CRDTs associated with each element are much larger than a single number (causal length) in our CLSet CRDT. In [27], we compare CLSet with existing set CRDTs in more detail.

VIII. CONCLUSION

We presented a two-layer system for multi-synchronous access to relational databases in a cloud-edge environment. The underlying CRR layer uses CRDTs to allow immediate data access at edge and to guarantee data convergence when the edge devices are online. It also resolves violations of integrity constraints at merge by undoing offending updates. A key enabling contribution is a new set CRDT based on causal lengths. Applications access databases through the AR layer with little or no concern of the underlying mechanism. We implemented a prototype on top of a data mapping library that is independent of any specific DBMS. Therefore servers in the cloud and edge devices can deploy different DBMSs.

IX. Acknowledgment

The authors would like to thank the members of the COAST team at Inria Nancy-Grand Est/Loria, in particular Victorien Elvinger, for inspiring discussions.

References

[1] A. Fox and E. A. Brewer, "Harvest, yield and scalable tolerant systems," in *The Seventh Workshop on Hot Topics in Operating Systems*, 1999, pp. 174–178.

[2] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[3] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, (SSS 2011)*, 2011, pp. 386–400.

[4] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott, "Riak DT map: a composable, convergent replicated dictionary," in *The First Workshop on the Principles and Practice of Eventual Consistency*, 2014, pp. 1–1.

[5] G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for P2P collaborative editing," in *CSCW*. ACM, 2006, pp. 259–268.

[6] N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *ICDCS*, 2009, pp. 395–403.

[7] W. Yu, L. André, and C.-L. Ignat, "A CRDT supporting selective undo for collaborative text editing," in *DAIS*, 2015, pp. 193–206.

[8] M. Nicolas, V. Elvinger, G. Oster, C.-L. Ignat, and F. Charoy, "MUTE: A peer-to-peer web-based real-time collaborative editor," in *ECSCW Panels, Demos and Posters*, 2017.

[9] M. Kleppmann and A. R. Beresford, "A conflict-free replicated JSON datatype," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2733–2746, 2017.

[10] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "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.

[11] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 265–278.

[12] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," *Rapport de recherche*, vol. 7506, January 2011.

[13] P. Johnson and R. Thomas, "The maintamance of duplicated databases," *Internet Request for Comments RFC 677*, January 1976.

[14] V. K. Garg, *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.

[15] P. S. Almeida, A. Shoker, and C. Baquero, "Delta state replicated data types," *J. Parallel Distrib. Comput.*, vol. 111, pp. 162–173, 2018.

[16] V. Enes, P. S. Almeida, C. Baquero, and J. Leitão, "Efficient Synchronization of State-based CRDTs," in *IEEE 35th International Conference on Data Engineering (ICDE)*, April 2019.

[17] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte, "A optimized conflict-free replicated set," *Rapport de recherche*, vol. 8083, October 2012.

[18] C. Meiklejohn and P. van Roy, "Lasp: a language for distributed, coordination-free programming," in *the 17th International Symposium on Principles and Practice of Declarative Programming*, 2015, pp. 184–195.

[19] W. Yu, V. Elvinger, and C.-L. Ignat, "A generic undo support for state-based CRDTs," in *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, ser. LIPIcs, vol. 153, 2020, pp. 14:1–14:17.

[20] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *Principles of Distributed Systems (OPODIS)*, ser. LNCS, vol. 8878. Springer, 2014, pp. 17–32.

[21] C. S. Meiklejohn, V. Enes, J. Yoo, C. Baquero, P. van Roy, and A. Bieniusa, "Practical evaluation of the lasp programming model at large scale: an experience report," in *the 19th International Symposium on Principles and Practice of Declarative Programming*, 2017, pp. 109–114.

[22] D. Mealha, N. Preguiça, M. C. Gomes, and J. Leitão, "Data replication on the cloud/edge," in *Proceedings of the 6th Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)*, 2019, pp. 7:1–7:7.

[23] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *SIGMOD*. ACM, 1989, pp. 399–407.

[24] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. M. Preguiça, "Extending eventually consistent cloud databases for enforcing numeric invariants," in *34th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2015, pp. 31–36.

[25] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 185–196, 2014.

[26] P. Lopes, J. Sousa, V. Balegas, C. Ferreira, S. Duarte, A. Bieniusa, R. Rodrigues, and N. M. Preguiça, "Antidote SQL: relaxed when possible, strict when necessary," *CoRR*, vol. abs/1902.03576, 2019.

[27] W. Yu and S. Rostad, "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.