Augmenting SQLite for Local-First Software

Iver Toft Tomter and Weihai Yu

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

Abstract. Local-first software aims at both the ability to work offline on local data and the ability to collaborate across multiple devices. CRDTs (conflict-free replicated data types) are abstractions for offline and collaborative work that guarantees strong eventual consistency. RDB (relational database) is a mature and successful computer industry for management of data, and SQLite is an ideal RDB candidate for offline work on locally stored data. CRRs (conflict-free replicated relations) apply CRDTs to RDB data. This paper presents our work in progress that augments SQLite databases with CRR for local-first software. No modification or extra software is needed for existing SQLite applications to continue working with the augmented databases.

Keywords: Data replication, eventual consistency, conflict-free replicated data types, conflict-free replicated relations, edge computing

1 Introduction

Local-first software suggests a set of principles for software that enables both collaboration and ownership for users. Local-first ideals include the ability to work offline and collaborate across multiple devices [7].

RDB (relational database) is a mature and successful computer industry for management of data, and SQLite is an open source RDB engine that is an ideal candidate for local-first software, because its operation does not rely on network connectivity. Citing its homepage[1]: "SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day."

One of the main challenges of supporting local-first software is the general limitation of a networked system, as stated in the CAP theorem [3, 5]: it is impossible to simultaneously ensure all three desirable properties, namely consistency equivalent to a single up-to-date copy of data, availability of the data for update and tolerance to network partition.

CRDTs (conflict-free replicated data types) [10] emerged to address the CAP challenges. 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 [10]).

[1] https://sqlite.org

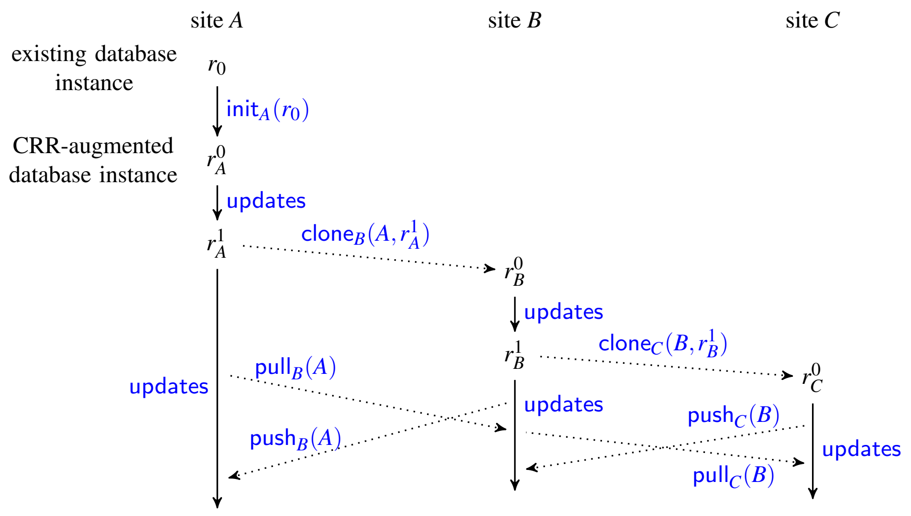

Fig. 1. A scenario of asynchronous database updates

CRRs (conflict-free replicated relations) apply CRDTs to RDBs [12]. In [12], we reported a CRR prototype that was built on top of an ORM (object-relation mapping) called Ecto[2]. Applications are therefore limited to those using the particular ORM. Unfortunately, Ecto does not support SQLite in the latest versions.

In this paper, we report our work-in-progress implementation that augments SQLite databases with CRR support. With a single command, we augment an existing SQLite database instance with CRR. All applications using the database, including the `sqlite3` shell[3], continue to work without any modification. Later, we can clone the augmented database to different sites. We can query and update the database instances at different sites independently. We can synchronize the instances when the sites are connected. We are not locked in, though. We can easily drop the CRR-augmentation on any of the database instances without losing any of the original database features.

Fig. 1 shows a scenario of using our software. Initially we have a SQLite database instance r_0 at site A. We run $\mathsf{init}(r_0)$ to augment r_0 to r_A^0 with CRR support. We then apply some updates that lead to instance r_A^1. Now at site B we run $\mathsf{clone}(r_A^1)$ to get a clone of the database instance. Independently, we make updates on the local instances at sites A and B. Later, we make yet another clone from site B to site C. From now on, we make local updates at all three sites and occasionally push our local updates to remote sites and pull remote updates to local instances.

2 Requirements

A primary requirement for local-first software is that a site should be able to independently perform queries and updates on the local database instances.

[2] `https://github.com/elixir-ecto/ecto`

[3] `https://sqlite.org/cli.html`

When two sites are connected, one site should be able to merge the updates performed at the other site without coordination. In particular, the site should be able to resolve conflicts without collecting votes from other sites.

The instances at different sites should be eventually consistent, or convergent [11]. That is, when they have applied the same set of updates, they should have the same state.

Database integrity constraints should be enforced. In particular, a merge of concurrent updates may cause the violation of an integrity constraint, though none of the updates violated any constraint locally at the sites. When this happens, one of the offending updates should be undone. It is important that the sites independently undo the same offending update.

Finally, existing applications should continue to work without any modification. In particular, performing queries and updates on local instances should not depend on the augmentation or any additional third-party software.

3 Technical background

In this section, we review the necessary background information about CRDT and CRR.

3.1 CRDT

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 [10]. 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 [4], 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.

Fig. 2 (left) shows GSet, a state-based CRDT for grow-only sets [10], where E is a set of possible elements, $\sqsubseteq \overset{\text{def}}{=} \subseteq$, $\sqcup \overset{\text{def}}{=} \cup$, insert is a mutator (update operation) and in is a query. Obviously, an update through $\mathsf{insert}(s,e)$ is an inflation, because $s \subseteq \{e\} \cup s$. Fig. 2 (right) shows the Hasse diagram of the states in a GSet. A Hasse diagram shows only the "direct links" between states.

Using state-based CRDTs, as originally presented [10], is costly in practice, because states in their entirety are sent as messages. Delta-state CRDTs address this issue

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

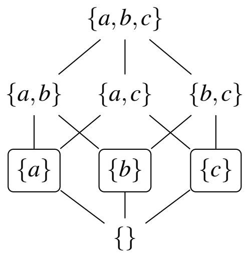

Fig. 2. GSet CRDT and Hasse diagram of states

by only sending join-irreducible states [1,2]. 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 Fig. 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 general-purpose set CRDT ("general-purpose" in the sense that it allows both insertion and deletion of elements), or more specifically, a delta-state set CRDT.

We use CLSet (causal-length set, [12, 13]), a general-purpose set CRDT, where each element is associated with a *causal length*. Intuitively, insertion and deletion are inverse operations of one another. They always occur in turn. When an element is first inserted into a set, its causal length is 1. When the element is deleted, its causal length becomes 2. Thereby the causal length of an element increments on each update that reverses the effect of a previous one.

$$\begin{aligned}
\mathsf{CLSet}(E) &\overset{\text{def}}{=} E \hookrightarrow \mathbb{N}\\
\mathsf{insert}(s,e) &\overset{\text{def}}{=} \begin{cases} s\{e \mapsto s(e)+1\} & \text{if } \mathsf{even}\big(s(e)\big)\\ s & \text{if } \mathsf{odd}\big(s(e)\big)\end{cases}\\
\mathsf{insert}^{\delta}(s,e) &\overset{\text{def}}{=} \begin{cases} \{e \mapsto s(e)+1\} & \text{if } \mathsf{even}\big(s(e)\big)\\ \{\} & \text{if } \mathsf{odd}\big(s(e)\big)\end{cases}\\
\mathsf{delete}(s,e) &\overset{\text{def}}{=} \begin{cases} s & \text{if } \mathsf{even}\big(s(e)\big)\\ s\{e \mapsto s(e)+1\} & \text{if } \mathsf{odd}\big(s(e)\big)\end{cases}\\
\mathsf{delete}^{\delta}(s,e) &\overset{\text{def}}{=} \begin{cases} \{\} & \text{if } \mathsf{even}\big(s(e)\big)\\ \{e \mapsto s(e)+1\} & \text{if } \mathsf{odd}\big(s(e)\big)\end{cases}\\
(s \sqcup s')(e) &\overset{\text{def}}{=} \mathsf{max}\big(s(e), s'(e)\big)\\
\mathsf{in}(s,e) &\overset{\text{def}}{=} \mathsf{odd}\big(s(e)\big)
\end{aligned}$$

Fig. 3. CLSet CRDT [12]

As shown in Fig. 3, the states of a CLSet are a partial function $s\colon E \hookrightarrow \mathbb{N}$, meaning that when e is not in the domain of s, $s(e) = 0$ (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 regarded to be in the set when its causal length is an odd number. A local insertion has effect only when the element is not in the set. Similarly, a local deletion has effect only when the element is actually in the set. A local effective insertion or deletion simply increments the causal length of the element by one. For every element e in s and/or s', the new causal length of e after merging s and s' is the maximum of the causal lengths of e in s and s'.

3.2 CRR

The RDB supporting CRR consists of two layers: an Application Relation (AR) layer and a Conflict-free Replicated Relation (CRR) layer (see Fig. 4). 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.

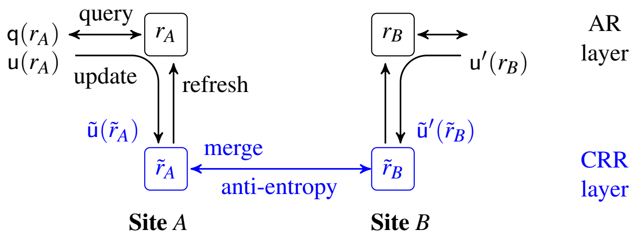

Fig. 4. A two-layer relational database system [12]

An AR-layer database schema R has an augmented CRR schema $\tilde{R}$. In Fig. 4, site A maintains both an instance r_A of R and an instance $\tilde{r}_A$ of $\tilde{R}$. A query q is performed on r_A without the involvement of $\tilde{r}_A$. An update u on r_A triggers an additional update $\tilde{\mathsf{u}}$ on $\tilde{r}_A$. The update $\tilde{\mathsf{u}}$ is later propagated to remote sites through an anti-entropy protocol. Merge with an incoming remote update $\tilde{\mathsf{u}}'(\tilde{r}_B)$ results in an update $\tilde{u}'$ on $\tilde{r}_A$ as well as an update u' on r_A.

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

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 failed merge would cause some compensation updates.

We adopt several CRDTs for CRRs. Since a relation instance is a set of tuples or rows, we use the CLSet CRDT (Fig. 3) for relation instances. We use the LWW (last-write wins) register CRDT [6, 9] for individual attributes in tuples.

4 Work-in-progress implementation

We implement all functionality required for local updates completely inside the SQLite database, so no modification to existing applications or extra software is required for the applications to be able to continue working with the database. We implement the command-line features in Python (that in turn calls SQL statements).

4.1 Command-line operations

For command-line operations, we adopt `git`[4] operation names.

– `init` augments an existing SQLite database instance with CRR support.
– `clone` copies a remote augmented database instance to a local location.
– `pull` merges remotely applied updates to the local instance.
– `push` merges locally applied updates to a remote instance.
– `remote` queries and configures the settings of remote instances.

4.2 CRR-augmented database

For an AR-layer relation schema $R(A_1, A_2, \dots)$, we generate a new CRR-layer schema $\tilde{R}(K, L, T_1, T_2, \dots, A_1, A_2, \dots)$, ignoring all integrity constraints in R. K is the primary key of $\tilde{R}$. K values are globally unique. L is the causal-lengths (Fig. 3) of the tuples in $\tilde{R}$. T_i is the timestamp of the last update on attribute A_i. In other words, the (K, L) part represents the CLSet CRDT of tuples and the (A_i, T_i) parts represent the LWW register CRDT of the attributes.

In what follows, we write $t(K)$, $t(A_i)$ etc. for the K and A_i values of tuple t.

We use `randomblob(32)` of SQLite to generate K values. The chance that two tuples in the same relation have the same K value is extremely small.

For the AR-layer relations, we also generate triggers. We describe the triggers later in Section 4.3.

In addition to the augmentation of the AR-layer relations, we generate three more relations. A *Clock* relation implements a hybrid logical-physical clock [8] at a (virtual) nanosecond scale (Section 4.6). A *Site* relation maintains the information about the sites known at this instance. The information includes the hosts and paths of the remote instances, the last time this instance applied a push and a pull to the sites, etc. A *History* relation maintains a history of all the updates that have been applied.

4.3 Local updates

The `init` operation generates triggers on relation R. Every insertion, deletion and update on an instance r of R triggers the corresponding update on the instance $\tilde{r}$ of $\tilde{R}$.

When inserting a new tuple t into r, we insert a new tuple $\tilde{t}$ into $\tilde{r}$, with the initial $\tilde{t}(L) = 1$. When deleting t from r, we increment $\tilde{t}(L)$ with 1, so that the new $\tilde{t}(L)$ becomes an even number. When inserting the deleted t back to r, we increment $\tilde{t}(L)$

[4] `https://git-scm.com`

with 1, so that the new $\tilde{t}(L)$ turns back to an odd number. When updating $t(A_i)$ in r, we update $\tilde{t}(A_i)$ and $\tilde{t}(T_i)$ in $\tilde{r}$.

Since no integrity constraint is defined in $\tilde{R}$, a successful local update on r will always lead to a successful update in $\tilde{r}$.

In addition to the triggers on R, the `init` operation also generates triggers on $\tilde{R}$. For every update on an instance $\tilde{r}$ of $\tilde{R}$, a trigger inserts a tuple in the *History* relation.

4.4 Merges

The pull of the concurrent updates from a remote site consists of the following steps: 1) generating the concurrent updates at the remote site; 2) transferring the generated updates to the local site (to be described in Section 4.5); 3) merging the received concurrent updates. A push is handled in a similar way.

As CRRs are based on delta-state CRDTs, the updates are join-irreducible states in a join-semilattice (Section 3.1). In our case, the updates are in fact the tuples in $\tilde{r}$ (Section 3.2). Using the *History* relation and the information of the last push and pull in relation *Site*, we can generate the updates since the last push or pull.

We generate the concurrent updates of the remote instance in a temporary database[5] and transfer it to the pulling site. This way, we avoid encoding individual tuples into an intermediate representation.

During the merge of received updates, we temporarily disable the generated triggers on AR-layer relation instances by setting a flag on the triggers.

An update on an relation instance $\tilde{r}'$ at a remote site is actually a tuple $\tilde{t}'$. If a tuple $\tilde{t}$ in the local instance $\tilde{r}$ exists such that $\tilde{t}(K) = \tilde{t}'(K)$, we update $\tilde{t}$ with $\tilde{t} \sqcup \tilde{t}'$ where the merge $\sqcup$ is the join operation of the join-semilattice (Section 3.1). Otherwise, we insert $\tilde{t}'$ into $\tilde{r}$. The merge $\tilde{t} \sqcup \tilde{t}'$ is defined as:

$$\tilde{t} \sqcup \tilde{t}' \overset{\text{def}}{=} \tilde{t}'', \text{ where } \tilde{t}''(L) = \mathsf{max}(\tilde{t}(L), \tilde{t}'(L)), \text{ and}$$

$$\tilde{t}''(A_i), \tilde{t}''(T_i) = \begin{cases} \tilde{t}'(A_i), \tilde{t}'(T_i) & \text{if } \tilde{t}'(T_i) > \tilde{t}(T_i) \\ \tilde{t}(A_i), \tilde{t}(T_i) & \text{otherwise} \end{cases}$$

After the update of $\tilde{r}$, we update r as the following. If $\tilde{t}(L)$ is an even number, we delete t (where $t(A_1) = \tilde{t}(A_1) \wedge t(A_2) = \tilde{t}(A_2) \wedge \ldots$) from r. Otherwise, we insert or update r with $\pi_{A_1, A_2, \ldots}(\tilde{t})$.

If the update on r violates an integrity constraint, we first roll back the updates on r and $\tilde{r}$ and then start an compensation update [12] (remaining to complete, see Section 4.7).

4.5 Network connections

At present, we support access to remote database instances in two possible cases: 1) the remote database instance is located on the same host as the local instance; or 2) the remote instance is located on a host where we have `ssh`[6] access.

[5] `https://sqlite.org/lang_attach.html`

[6] `https://www.ssh.com/`

When performing a clone, we specify a database instance stored on a remote host as `ssh://user@host#port:path/to/db`.

Since a SQLite database instance is stored as a file, we may (accidentally) copy or move the file to a different location. Every time we open an instance for push or pull, we verify the location information of the local instance stored in the *Site* relation and make modifications accordingly. We run the `remote` command-line operation to explicitly set or modify the location information of remote instances.

4.6 Timestamp values

The *Clock* relation, which the `init` operation creates, addresses two issues. The first issue is that the finest time resolution that SQLite provides is at a sub-millisecond level, so consecutive updates may have the same timestamp value. Notice that a third-party library with higher time resolution does not help, since our goal is to implement all features related with local updates completely inside the SQLite database instance. The second issue is that the physical clock (or “wall” clock) values are not sufficient to represent the happen-before relationship between updates, so a concurrent update may mistakenly win a competition when the physical clocks at different sites are skewed.

To address the first issue, we use the $Clock(Ms,Ns)$ relation, where Ms is the physical clock value in milliseconds and Ns is the offset within a millisecond at nanosecond scale. The *Clock* relation has only one tuple (τ_{ms}, τ_{ns}), which is the last timestamp value that has been generated or merged. The comparison of two timestamp values is defined as $(\tau'_{ms}, \tau'_{ns}) > (\tau_{ms}, \tau_{ns})$ iff $\tau'_{ms} > \tau_{ms}$ or $\tau'_{ms} = \tau_{ms} \wedge \tau'_{ns} > \tau_{ns}$.

To generate a new timestamp value, we first generate a new physical clock value τ'_{ms} (derived from the `julianday` function of SQLite) and a random number τ'_{ns} such that $0 \leq \tau'_{ns} < 10^6$. If the generated value (τ'_{ms}, τ'_{ns}) is greater than the old value (τ_{ms}, τ_{ns}), the new timestamp value is (τ'_{ms}, τ'_{ns}). Otherwise, we generate a new random number τ''_{ns} such that $\tau_{ns} < \tau''_{ns} < 10^6$ and the new timestamp value becomes (τ_{ms}, τ''_{ns}).

To address the second issue, we implement a hybrid logical-physical clock [8], which has the property that for two updates u_1 and u_2 with timestamp values τ_1 and τ_2, $\tau_1 < \tau_2$ iff u_1 happens before u_2 or u_1 and u_2 are concurrent. In other words, u_2 does not happen before u_1 when $\tau_1 < \tau_2$.

At a merge, if a timestamp value (τ'_{ms}, τ'_{ns}) of the incoming tuple is greater than the (τ_{ms}, τ_{ns}) tuple in the *Clock* instance, we update the instance with (τ'_{ms}, τ'_{ns}).

4.7 Current implementation status

At the time of this writing, we have not finished all the features described in [12]. The remaining features include: enforcement of integrity constraints that are violated at the time of merge, and using a counter CRDT for lossless resolution of concurrent attribute updates with additive update semantic. In addition to the merge in batch mode (push and pull), we are going to implement a continuous synchronization mode, like in [12], so that the sites can frequently exchange latest updates without explicit command-line push and pull. We can use our earlier implementation on top of an ORM [12] as a guideline to implement the features that we have not implemented so far, in particular the features that do not need to be implemented fully in SQL.

We currently focus on making a working prototype and have not put much effort on performance issues. We expect performance penalties for database updates, since an update at the AR layer now involves multiple updates. The performance of queries should not be affected, since they do not involve the CRR-augmented parts.

5 Related work

We limit the comparison to an alternative implementation reported in our earlier paper [12] and refer the interested reader to [12] for discussions on the other research work that are generally related to CRR.

Earlier, we implemented CRR on top of the Ecto ORM. One advantage of an implementation on top of an ORM is that it supports all RDBMSs (relational database management systems) that the ORM supports. It is even possible to synchronize between the instances running with different RDBMSs. There are some drawbacks, though. Only the applications using the ORM (and in the programming language of the ORM) can benefit from the CRR support. The supported RDBMSs are limited to those supported by the ORM. Unfortunately, the ORM of our choice, Ecto, does not support SQLite in the latest versions, and SQLite is an ideal candidate for local-first software (Section 1).

Implementing direct CRR support for SQLite addresses the above-mentioned drawbacks, at the cost of not benefiting from the advantages.

In [12], the implementation was mostly in the Elixir[7] programming language. Now, we try to keep the implementation as much in SQLite as possible. In particular, we aim at implementing all features related to local updates inside SQLite, so that existing applications continue to work without making any modification. We even restrict our implementation to the SQLite distribution that does not include any extension.

Since Elixir facilitates actor-based programs, data communication is built in. Little programming effort is needed for data communication. On the other hand, every database update is encoded and decoded between RDBMS and Elixir representations. This increases run-time overhead. Moreover, data security is not taken into account. Since we now transfer data through `ssh` connections, we do not have to worry about security and configuration issues.

There are some further differences in implementation details. In [12], a local update is first made in the CRR layer and then refreshed to the AR layer. Now the update first happens in the AR layer which then triggers updates in the CRR layer.

Currently, our implementation has not yet been as complete as the earlier implementation (Section 4.7).

6 Conclusion

We have presented a work-in-progress implementation of a software prototype that augments existing SQLite databases for local-first software. With a single command-line operation, we augment an existing database instance with CRR support. Existing applications using the existing database instance, without any modification, continue to

[7] `https://elixir-lang.org/`

work with the augmented instances. We can then maintain multiple instances of the same database at different devices and independently query and update the different instances. We can synchronize the updates at different instances when the devices are connected. The instances are eventually consistent. That is, they will have the same state when they have applied the same set of updates. The implementation is still in its early stage and more features remain to be completed.

References

1. ALMEIDA, P. S., SHOKER, A., AND BAQUERO, C. Delta state replicated data types. *J. Parallel Distrib. Comput. 111* (2018), 162–173.
2. ENES, V., ALMEIDA, P. S., BAQUERO, C., AND LEITÃO, J. Efficient Synchronization of State-based CRDTs. In *IEEE 35th International Conference on Data Engineering (ICDE)* (April 2019).
3. FOX, A., AND BREWER, E. A. Harvest, yield and scalable tolerant systems. In *The Seventh Workshop on Hot Topics in Operating Systems* (1999), pp. 174–178.
4. GARG, V. K. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.
5. GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (2002), 51–59.
6. JOHNSON, P., AND THOMAS, R. The maintamance of duplicated databases. *Internet Request for Comments RFC 677* (January 1976).
7. KLEPPMANN, M., WIGGINS, A., VAN HARDENBERG, P., AND MCGRANAGHAN, M. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, (Onward! 2019)* (2019), pp. 154–178.
8. KULKARNI, S. S., DEMIRBAS, M., MADAPPA, D., AVVA, B., AND LEONE, M. Logical physical clocks. In *Principles of Distributed Systems (OPODIS)* (2014), vol. 8878 of *LNCS*, Springer, pp. 17–32.
9. SHAPIRO, M., PREGUIÇA, N. M., BAQUERO, C., AND ZAWIRSKI, M. A comprehensive study of convergent and commutative replicated data types. *Rapport de recherche 7506* (January 2011).
10. SHAPIRO, M., PREGUIÇA, N. M., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, (SSS 2011)* (2011), pp. 386–400.
11. VOGELS, W. Eventually consistent. *Comminications of the ACM 52*, 1 (2009), 40–44.
12. YU, W., AND IGNAT, C.-L. Conflict-free replicated relations for multi-synchronous database management at edge. In *IEEE International Conference on Smart Data Services (SMDS)* (2020), pp. 113–121.
13. YU, W., AND ROSTAD, S. A low-cost set CRDT based on causal lengths. In *Proceedings of the 7th Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)* (2020), pp. 5:1–5:6.