Reversible Conflict-free Replicated Data Types

Yunhao Mao
yunhao.mao@mail.utoronto.ca
University of Toronto

Zongxin Liu
zongxin.liu@mail.utoronto.ca
University of Toronto

Hans-Arno Jacobsen
jacobsen@eecg.toronto.edu
University of Toronto

ABSTRACT

Conflict-free replicated data types (CRDTs) are popular for optimistic replication and ensuring strong eventual consistency (SEC) in distributed systems. However, *reversibility* is an underdeveloped functionality for CRDTs, despite its usefulness in system restoration from an erroneous state or undoing unwanted operations. In this paper, we define the concept and design of reversible CRDTs (rCRDTs). Reverse operations *compensate* for the *effect* of reversed updates, and they extend existing CRDT interfaces. Three abstractions for reversibility are proposed: reversing a single update, multiple causally related updates, and multiple logically related updates that capture the user intention behind the updates. Moreover, a replicated and distributed key-value store, rKVCRDT, is implemented as a proof of concept that integrates the support of reversible CRDTs. The rCRDTs' evaluation show that although adding reversibility affects the system's performance, the end result depends on multiple factors and varies based on the underlying CRDTs. System designers must consider the trade-off between the benefit of reversibility and the performance impact.

CCS CONCEPTS

• **Theory of computation → Distributed algorithms**.

KEYWORDS

Conflict-free Replicated Data Types (CRDTs), Eventual Consistency, Replication

ACM Reference Format:
Yunhao Mao, Zongxin Liu, and Hans-Arno Jacobsen. 2022. Reversible Conflict-free Replicated Data Types. In *23rd ACM/IFIP International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3528535.3565252

1 INTRODUCTION

With the increasing popularity of cloud computing, cloud service providers are more reliant on geo-replicated data to ensure service reliability and improve performance in large-scale distributed systems, such as massively scalable replicated data stores [19]. However, to remain strongly consistent, replicated data may suffer from low availability to account for network partition or high latency when conducting updates because of the CAP theorem [10]. As a result, weaker consistency models, such as eventual consistency, are used in replicated systems when high availability instead of strong data consistency is favored [3]. For example, Amazon's DynamoDB [15] employs eventual consistency to ensure data are "always writable". This means that users can expect to perform updates on any replica and that any writes are always preserved even when replicas cannot instantaneously communicate with each other and real-time coordination is impossible.

An important problem with such *optimistic replication* [30] methods is the reconciliation of conflicting concurrent updates. Solutions such as DynamoDB simply present all concurrent versions to the user to perform reconciliation. *Conflict-free replicated data types* (CRDTs) [32] adopt a convenient and practical method that uses predefined concurrency semantics in the form of abstract data types (ADTs) to achieve automatic reconciliation no matter the orders of the updates.

CRDT updates are immediately executed on a replica when they are received, and CRDTs guarantee *strong eventual consistency* (SEC) to ensure that all correct replicas that are eventually propagated with the same updates expose an *equivalent* state without any ordering or coordination, which results in "conflict-free" operations. CRDTs are seen in key-value databases [29] [28] [1] [34], collaborative editing tools [23] [22], application-specific data synchronization [17] [9] and even in blockchains [18] [25].

However, CRDTs usually lack *reversibility* or the ability to "undo". This is because, first, the "undo" is rarely a concern for regular ADTs, and there is no universal abstraction for reversing an ADT operation. As a result, developing a definition that applies to different data types and that makes universal sense is challenging. Second, the traditional linearized undo is not directly applicable to concurrent updates in CRDTs [36]. Last, many CRDTs require their internal data structures to be inflationary; that is, any new state must encompass earlier states, thereby restricting the ability to undo by removing existing updates [27].

Including reversibility in CRDTs can be beneficial. For example, it can be applied to maintain data invariants: consider a stock counter that cannot go below zero or an inventory set that has an upper limited of items stored. Maintaining such an invariant may be difficult in eventually consistent systems and particularly in CRDTs, because the boundary detection may accept the allowed precondition on one replica, but the precondition on another replica is rejected [5]. Using reversible CRDTs, such boundary violations can be simply undone when detected.

Reversibility can also be applied in conjunction with distributed transaction protocols: algorithms that attempt to coordinate updates on different nodes as a single atomic commit [26]. For example, the SAGA transaction [16] conducts a series of database operations and rolls back executed operations if one of the steps fails. If one or more steps employ CRDTs to store data, reversibility alleviates the need to explicitly define rollback operations or modify the distributed transaction framework in-use, by simply letting

Middleware '22, November 7–11, 2022, Quebec, QC, Canada

ACM ISBN 978-1-4503-9340-9/22/11…$15.00
https://doi.org/10.1145/3528535.3565252

the datatype itself be responsible for the rollback. Furthermore, CRDTs are usually served as bundled packages through libraries or distributed data stores. Even if adding reversibility is type-specific, doing so to commonly-used data types can benefit many users with minimal effort.

In this paper, we propose a novel solution to incorporate reversibility in CRDTs. Reversibility is defined as achieving the *counter-effect* of selected updates through the use of *compensation*, similar to the concept of compensating transactions found in databases transaction [10] [21]. For example, the reverse of an "add" operation can be the "subtract" operation for a counter CRDT. The *effect* of an update is defined as the change in the *value* before and after an update because only user-perceived values are meaningful. Since CRDT updates may be independent of each other, a past update can be reversed individually with unrelated updates unaffected, instead of conducting a linear undo of updates.

Finally, reversible CRDTs (rCRDT) are extensions to existing CRDTs with the additional reverse operation in their interfaces and the compensating operations must be designed on a type-by-type basis, as we have discussed. We also propose bulk reverse operations for reversing multiple causally, semantically, or logically related updates to capture the user's intention behind the updates; that is, updates that are dependent on their execution orders, interface definitions, or the context of the operation are reversed in bulk.

In this paper, we make the following contributions:

(1) We propose a universal definition of reversibility for CRDTs. The definition is generic and compatible with most CRDTs that include variants to reverse multiple updates through the causal and logical bulk reverse operations.
(2) We design the *operation-history log* that captures the causal relationships among updates and is used to identify the updates to be reversed. We also propose two methods to achieve reversibility: the lazy approach and the eager approach with different performance trade-offs under different workloads.
(3) We present a distributed and replicated key-value database, the rKVCRDT, that interfaces with CRDTs and implements our reversible algorithms as a proof of concept. We also perform comprehensive evaluations using rKVCRDT to determine the overhead of reversibility and performance characteristics of different data types.

This paper is organized as follows. Section 2 discusses the background of CRDTs. Section 3 introduces our definition of reversible CRDTs for reversing a single update and multiple updates. Section 4 describes algorithms that achieve reversibility. Section 5 presents the replicated key-value database (rKVCRDT) and the evaluations of rCRDTs. Section 6 discusses the related work.

2 BACKGROUND

2.1 System Model

We assume an asynchronous distributed system consisting of a set of nodes that exchange information via message passing. Node failures may only occur as *crash faults*, and a crashed node may either recover with its memory intact or may never recover. The network is reliable, with no duplicated, lost, or corrupted messages; it may partition but *eventually* recovers. A node that is not crashed is referred to as a correct node and there is at least one correct node in the system. If a correct node sends a message, the message will eventually be delivered to all other correct nodes.

2.2 Conflict-Free Replicated Data Types (CRDTs)

Conflict-free replicated data types are ADTs designed for optimistic replication [32] [30]. A CRDT replica always executes any received request immediately upon arrival and then asynchronously propagates the update to other replicas

A CRDT consists of three major components: the *state*, the *interface* and the *update propagation mechanism*. The state is the internal data structure that represents the data type and is sometimes referred to as the *payload*. The interface semantically defines the behaviour of the CRDT by providing a set of operations that can be interacted with. We refer to operations with *side effect*s (can cause a change in states) as *updates*. The update propagation mechanism, sometimes referred to as the *anti-entropy protocol*, defines how an update is propagated and applied to other replicas. CRDTs guarantee strong eventual consistency [32], this consistency model ensures that the same value can be retrieved through the CRDT's *query* operations from any correct replica that has received the same updates. Two replicas are considered *equivalent* if the query returns the same value. The two most common update propagation mechanisms are *state-based* and *operation-based* [32] [27].

State-based CRDTs (CvRDTs). In a CvRDT, the state of the source replica after update execution is propagated to other replicas for synchronization. The receiving replica *merges* the received state with its own state, yielding a converged state. The merge function must be commutative, associative and idempotent. In addition, states must monotonically increase after updates, which ensures the set of all possible states to construct a join-semilattice. This allows the states to eventually converge to the same value [32].

The *state-based causal history* defined by Shapiro et al. [32] is a set that contains all the existing states and updates to a CRDT replica, representing the progression of the states over the updates. The order of the updates is determined by the order in which they are inserted into the set [32]. An update is *executed* if it is successfully included as part of the causal history.

Definition 2.1. Causal Ordering

For any two updates u_i and u_j, u_i *happened-before* u_j if u_i is included in the causal history of the replica when u_j is received, denoted as $u_i \prec u_j$. Additionally, u_i *happens-after* u_j if the opposite is true, denoted as $u_j \prec u_i$. Finally, u_i and u_j are *concurrent* if neither has a *happened-before* relation with the other.

By Definition 2.1, if an update *happens-after* another update, the first update, at its execution, must know of the existence of the second update from the causal history.

Operation-based CRDTs (CmRDTs). CmRDTs rely on commutative update operations. When an update is executed on one replica through a process referred to as *prepare-update*, an encoded version of the update is also propagated and the receiving replica executes the same update through the *effect-update* process, resulting in an equivalent state. Commutativity ensures that the execution order does not affect the final state; however, a causally reliable delivery is required.

A state-based CRDT can emulate an operation-based CRDT by representing the state as a set of updates and by modifying the query interface to apply the saved updates on read. Similarly, a state-based CRDT can be emulated by an operation-based CRDT by making the *effect-update* just contains state. Therefore, they are equivalent [7] [32].

3 REVERSIBLE CRDT (RCRDT) DESIGN

In this section, we introduce rCRDTs as extensions to existing *underlying CRDTs* where the original behaviours are not changed. Although exact behaviours of reversibility depend on the underlying CRDT, we present three generic definitions that reverse operations can be expected to follow.

3.1 User Intention Prediction

One important characteristic of CRDTs is that the client no longer involves the traditional read-modify-write process while updating a value. Since CRDT updates are executed on the replicas and "reading the most recent writes" is not guaranteed for the clients, *blind updates* (inserting a new value regardless of the old value) [12] could happen, thus detaching the causality among reads and writes [2]. Therefore, not only are there concurrent updates that depend on the same value, but also updates that depend on no previous value or any arbitrary previous value. This means that causally ordered updates in CRDTs may not represent user-intended relations among them. For example, when a user clicks the "like" button on a social media post to increment the "like" counter, they may not check the "likes" number in advance.

Thus, one of the main aims of our reversible CRDT design is to provide reversibility abstractions that capture the "real" *user intention* behind a series of updates where any past update can be reversed: if an update is a standalone update, reversing it should not affect other updates, even if there are new updates that occur chronologically after; if later updates depend on this update in some way, they will be reversed in bulk.

3.2 Operation History

Definition 3.1 (Operation History). The *operation history* of a state-based CRDT instance is a tuple $H = (S,U,Q)$, where S is a set of all states of all replicas. U is a set of all updates with *side effect*s , including merges, in the form of a tuple: $(before\_state, after\_state)$ where $before\_state \in S$ and $after\_state \in S$ represent the state of a replica before and after an update. Q is a set of all values that are read through the CRDT's *query* or *read* interface operation corresponding to each state in S. There is a one-to-one mapping $Query : S \rightarrow Q$ for states set (S) to readable values set (Q).

We adopt the state-based causal history discussed in Section 2 to keep track of all updates. The operation history (op-history) captures the state transitions of a CRDT instance as it progresses through time and the causal execution order of the updates. It provides an abstraction for accessing causal history.

Theorem 3.2 (Causally Dependent Updates). *For any pair of updates u_i,$u_j \in U$, if there is a sequence of transitions $U' \in U, \forall u_i \in U'$, where $0 \le i < j \le |U'|$, such that u_i's $after\_state \in before\_state$ of u_{i+1}, then $u_i \prec u_j$. Similarly, if there is a sequence of transitions that brings u_j to u_i, then $u_j \prec u_i$. If there is no such sequence, then u_i and u_j are concurrent.*

Proof. Consider two updates u_i,u_j, where u_i's $after\_state$ is s_0 and u_j's $before\_state$ is also s_0. By the definition of *happens-after* in Definition 2.1, if s_0 is the state after the execution of u_i, then u_i is included in s_0's *causal history* because both $after\_state$ and $before\_state$ in S by Definition 3.1. Since u_j is executed with the knowledge of s_0, it must have known u_i as well; therefore, $u_i \prec u_j$.

If a pair of updates does not share a state u as the $after\_state$ and $before\_state$, respectively, we can traverse the pairs of intermediate updates between u_i and u_j that do recursively share the same $after\_state$ and $before\_state$ recursively until u_j is reached. Using the same reasoning, we can say $u_i \prec u_{i+1} \prec ... \prec u_{j-1} \prec u_j$ through intermediate updates. If a merge is included, it can be indicated by checking whether the list of $before_state$s contains the $after\_state$ of the previous update. □

Theorem 3.2 demonstrates how the operation history can be applied to determine the partial order among updates and how one state can lead to another through the causal ordering of updates. For example, consider a state-based CRDT where there are four states s_0,s_1,s_2,s_3 and three updates $u_0 = (s_0,s_1)$, $u_1 = (s_1,s_2)$, $u_2 = (s_2,s_3)$; then, there is a sequence u_0,u_1,u_2 that goes from state s_0 to s_3, and we conclude $u_0 \prec u_2$.

If there is a merge, we again assume that there are three updates but an additional state s_4 as follows: $u_0 = ([s_0,s_1],s_2)$, $u_1 = ([s_0,s_1],s_3)$, $u_2 = (s_3,s_4)$. Next, $u_0 \prec u_2$ because s_2 and s_3 are equivalent (they have the same value in Q).

If two updates are executed on the same $before\_state$, $u_0 = (s_0,s_1)$, $u_1 = (s_0,s_2)$, this means that they are *concurrent* with any subsequent updates until the replicas converge.

By connecting updates in U, the operation history can be plotted as a directed acyclic graph to represent how the states change, diverge and converge. We can traverse the graph and locate the updates to reverse.

Definition 3.3 (Op-History Diagram (HG)). *HG* is a DAG (directed acyclic graph) in which the vertices are ordered by the connecting edges. *HG* can be constructed as follows:

(1) We start with an initial update $u_0 = (\_, initial\_state)$ as the origin vertex.
(2) Add a directed edge to pairs of updates such that the head vertex's $before\_states$ contains the tail vertex's $after\_states$.
(3) Remove all vertices that are merge operations and connect the previous and subsequent vertices of merges if they are not connected.
(4) By Theorem 3.2, the updates are causally ordered. Since all states must have predecessors (except for the initial state), *HG* must be a DAG.

An example of a state-based nested array CRDT's *HG* after several updates is shown in Figure 1. Consider an array of characters, and there is an update operation $insert(c,i)$: inserting a character c or an empty array at a given index i. If a nested array resides at i, c is inserted to the inner array. The read operation returns the array, and the merge function concatenates the arrays and discards

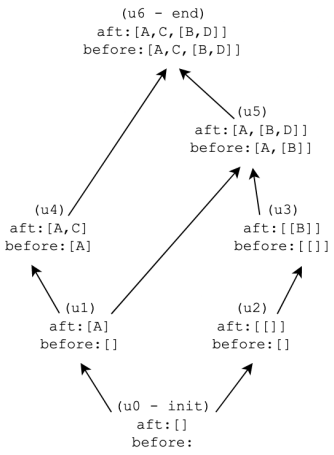


Figure 1: Example of the *HG* of the nested array CRDT

duplicates (in this and future illustrations, $before\_state$ shows the state after merging if a divergence exists).

The updates depicted in Figure 1 are carried out as follows:

(1) A CRDT instance with more then three replicas has an initial state of an empty array: [].
(2) Two concurrent updates are executed on different replicas. Update $u1$ inserts element A, and $u2$ inserts an empty inner array [], causing a divergence.
(3) Before the replicas can converge, $u3$ inserts B into the inner array. Next, some replicas converge, and their states reach $[A,[B]]$.
(4) C is inserted into a replica with converged state $[A]$ by $u4$, and D is concurrently inserted into $[A,[B]]$ on another replica by $u5$, causing another divergence.
(5) Finally, when all replicas have converged, the final state is $[A,C,[B,D]]$. Update $u6$ is trivial for displaying the final result.

3.3 Reversing a Single Update

The most basic reverse operation operates at the granularity of a *single* previous update.

Definition 3.4 (Value Difference t). For an update $u_k \in U$ with $before\_state = s_i \in S$ and $after\_state = s_j \in S$, the *value difference* t_k is the difference in the *queried* value $t_k = q_i - q_j$, where $q_i, q_j \in Q$ and $Query(s_i) = q_i$, $Query(s_j) = q_j$.

The definition of the subtraction operator and t depend on the underlying CRDT or is defined by the CRDT designer. For example, between two lists, $[a,b,c]$ and $[b,c,d]$, t can be $(insert(a), delete(d))$.

Definition 3.5 (Compensation Function). $C(u_k)$ is a user-defined function that takes an update as the only argument and then returns a *compatible* compensating operation f s.t. $C(u_k) = f(t_k)$, where f is based on update u_k's operation and the change in value t_k.

Definition 3.6 (Single Reverse). Executing $R\_Single(u)$ for update u will apply a compensating operation $C(u)$ to the current state, denoted as $R\_Single(u) = execute(C(u))$.

The compensation function that is described in Definitions 3.5 and 3.6 is a function that considers an update as input, reads the changes in values t imposed by the update and then generates a *compensating operation* based on the update content. It is type-depended, and it can be designed to have arbitrary behaviour. The reverse operation simply executes it as a new update.

The most intuitive design for a compensating operation is to obtain the *inverse* function of the original update. For example, u_4 in Figure 1 is an insertion operation that adds element C. Therefore, the compensating $C(u_4)$ should remove C from the set. After executing $R(u_4)$, the result would be $[A,[B,D]]$. In addition, the compensating operation must also be *compatible*, where it must fit existing CRDT operations. For example, a *grow-only counter* [31] is a counter where its only update operation is to add a non-negative integer. In this case, the compensating operation cannot be subtraction, the reverse is not simply the inverse of addition and an alternative must be defined.

We use compensating operations because it is just another update executed on a replica from the system's perspective, where complicated issues, such as execution and concurrency are handled automatically with the CRDT's replication mechanism. No extra failure handling is required either.

3.4 Reversing Causally Related Updates

As discussed in Section 3.1, we wish to account for related updates that depend on the reversed updates. The first method for predicting user intention among related updates is the traditional causal dependency (even if it may not be accurate as discussed in Section 3.1). This dependency is defined by the causal ordering in Definition 2.1 and captured by the op-history of Theorem 3.2.

Definition 3.7 (Causal Bulk Reverse). Given a *start* update u_s and an *end* update u_e. Let a set $C' = \{C(u_i) | \forall u_i \in U \text{ s.t. } u_s \prec u_i \wedge (u_i \prec u_e \vee u_i \text{ concurrent with } u_e)\}$ and $R\_Bulk\_Causal(u_s, u_e) = execute(\{\forall u \in C' \cup C(u_s) \cup C(u_e)\})$.

We use the $start$ and *end* updates as anchor points to indicate the range of causally related updates. An update that *happens-after* the $start$ must have knowledge of its existence, implying that they may be related. If an update *happens-after* the *end*, it is included because it knows the *end* exists and thus cannot be related.

Updates that are concurrent with *end* are reversed, but updates that are concurrent with $start$ are not reversed. This is because if an update is concurrent with *end*, the update cannot depend on *end*, but it may depend on $start$, so it may still be relevant. In contrast, being concurrent with $start$ signifies that this update cannot depend on $start$. Similarly, updates that are concurrent with both $start$ and *end* are not included either.

An example of a causal bulk reverse execution on a counter CRDT is shown in Figure 2: The CRDT has an initial value of 5. Update $u1$ is $start$ and update $u7$ is *end*. Assume that $start$ and *end* are special updates that do not change the value in this example. The updates with star labels are the updates to be reversed. Update $u2$ is concurrent with $u1$, so it is not included, even if its effect is known by other replicas between $u1$ and $u7$; update $u6$ is concurrent with $u7$; therefore, it is included even if its effect is known only after $u7$. Updates $u3$, $u4$, and $u5$ are obviously included. Finally, if $R\_Bulk\_Causal(u1, u7)$ is executed, the compensation will deduct $1+3+3+2=9$ from this counter, and the final result is $15-9=6$, coinciding with $u2$.

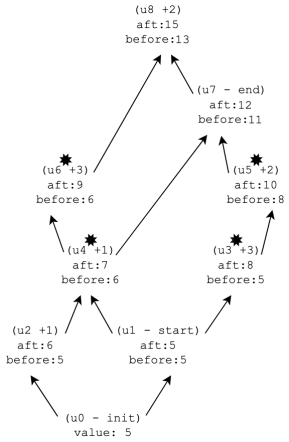


Figure 2: Causal bulk reverse

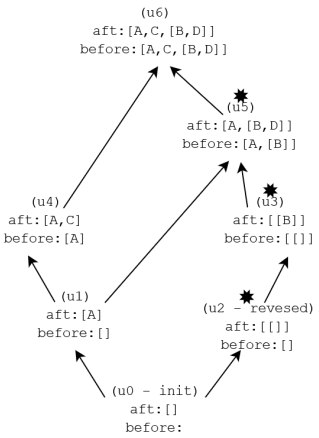


Figure 3: Logical bulk reverse

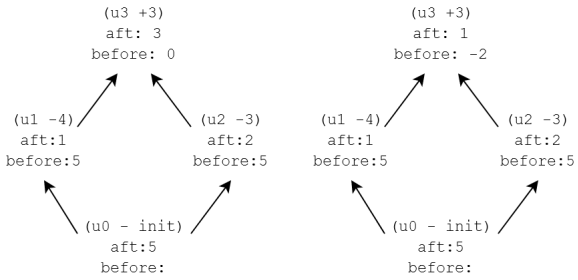


Figure 4: Non-serializable concurrent updates

An example use case would be a system where users are required to examine the current value before requesting an update, the causal bulk reverse can be used for undoing. When reversing an update u, it can be set to the *start*, and the most recent update observed on the operating replica is the *end* so all dependent updates to u are reversed.

3.5 Reversing Logically Related Updates

To capture the "real" logic behind a series of dependent updates, we can use the semantics of operations to determine their dependency. For example, the insertion into the inner array in Figure 1 requires the nested array to exist, because the index of the inner array is a parameter to insertion. Another alternative is simply requiring the user to input the dependency of related updates. These dependencies are also type-specific and need to be individually designed.

Definition 3.8 (Relative Set). Each $u_i \in U$ includes a set L_i that stores its *related* updates. While a new update u_j is added to U, $\forall u_i \in U$, u_i is checked with a user-defined CRDT-specific $Relate(u_i,u_j)$ function. If it returns true, u_i is added to L_j, and u_j is added to L_i.

Definition 3.9 (Logical Bulk Reverse). Let $l(u_i)$ be a function that returns the relative set L_i of u_i and recursively adds updates to L_i's relative sets: $l(u_i) = \{L_i \cup \{l(u_j)|\forall u_j \in L(i)\})\}$. Given an update u_i to be reversed, let set $C' = \{C(u_k)|\forall u_k \in l(u_i)\}$ and $R\_Bulk\_Logical(u_i) = execute(\{\forall u \in C' \cup C(u_i)\})$.

With the logical bulk reverse, the relative set is searched recursively during a reverse operation, and every related update is reversed as per Definition 3.9.

Figure 3 demonstrates logical bulk reverse with the nested array example, where there exist semantic dependencies among updates $u2$, $u3$ and $u5$. Update $u3$ and $u5$ are related to the creation of the array, which is $u2$. Thus, if $u2$ is reversed, $u3$ and $u5$ are also reversed.

One use case for the logical bulk reverse can be a collaborative graphical editing tool that uses a reversible CRDT-Graph [31] to store editable shapes. Adjacent shapes are linked to each other by graph edges, so they can be moved or jointly deleted. Logical bulk reverse can make sure that reversing the creation of one shape automatically removes all linked shapes without affecting unrelated shapes.

3.6 Concurrent and Non-serializable Updates

When attempting to linearly order CRDT updates, it is possible that the original outcome of certain concurrent updates cannot be reached or that there are multiple possible serial orders [7] [8]. For example, a multi-value register acts like a single-value last-write win register with any sequential order of inputs, but if there are concurrent writes, its value becomes a set [31].

A more detailed example is illustrated in Figure 4, where two replicated counters both have an invariant which their values must be greater than 0 with slightly different behaviours. On the left, the constraint resets the value to 0 if any update decreases it below 0, so the result is 0 after $u1$, $u2$ are merged. On the right, the constraint allows concurrent updates to produce results that are temporarily below 0 after merging, but the subsequent update *must* increase the value above 0.

On the left, both execution sequences of $u0 \rightarrow u1 \rightarrow u2 \rightarrow u3$ and $u0 \rightarrow u2 \rightarrow u1 \rightarrow u3$ work because the third update always brings the value to 0. If $u2$ is the second update, then $t_2 = 3$. If $u2$ is the third update, then $t_2 = 1$. This process causes the reversal of $u2$ to become nondeterministic depending on the order that a replica observes. On the right, it is not possible to create a total ordering because making either $u1$ or $u2$ as the third update is not permitted by the constraint, so the subsequent update cannot occur and $u3$ is irreversible.

Since this issue is caused by the behaviours of the underlying CRDTs, a generic solution that suits all scenarios is unattainable. We propose two compromises:

(1) Simply apply the compensating operations in the order in which they are observed by the replica and disregard the nondeterminism, provided that common CRDTs do not exhibit such problems.
(2) Modify $R\_Single(u)$ so that it inspects u's concurrent updates, then combine all concurrent updates into a single u' and reverse u'. Bulk reverse operations need to be modified to avoid duplicated reverses. However, if there is one update that is concurrent with all other updates, this method combines all updates into a single large update, preventing any of the updates from being reversed individually.

4 REVERSIBLE CRDT IMPLEMENTATION

In this section, we present algorithms that achieve the causal bulk reverse. Since logical bulk reverse can be achieved with a few modifications to the given algorithms, it is not separately shown due to space constraints. Concurrent updates are handled by the first approach discussed in Section 3.6 for simplicity.

4.1 Op-History Log

We adopt the op-history H from Definition 3.1 into a partially ordered log data structure called the *op-history log* (*HL*) to store past updates, and it is replicated to each CRDT replica. Each update is an entry in the log that contains the causal and logical relations to other updates, and the state change. The causal relation among updates are identified with logical timestamps (vector clocks) [14]. *HL* is implemented as an operation-based CRDT to ensure strong eventual consistency when used by reversible CRDTs. *HL* has the following attributes:

HL.log: a set that stores every update as a log entry.

HL.heads: a set of pointers to the latest updates (more than one pointer if there are concurrent latest updates).

HL.currtime: a vector clock of the latest update time.

Each update u is an entry object in *HL.log* with the following attributes:

$u.aft$: a set of pointers to updates that are executed immediately *after* u.

$u.prev$: a set of pointers to updates that are executed immediately *before* u.

$u.diff$: the difference t in values between $before\_state$ and $after\_state$, utilized by the compensating operation.

$u.timestamp$: a vector clock used to arbitrate the causal ordering among updates, is set to *HL.currtime* when u is initialized.

$u.related$: optional for logical bulk reverse as the L set.

In the remainder of this paper, we use the symbol u to refer to either an update, an entry in *HL.log* or the propagated update message after inserting an update for convenience.

4.2 Log Insertion

The log needs to record all updates on the CRDT with causal information and to synchronize it to all replicas as an operation-based CRDT. The $prepare()$ method (Algorithm 1) is called after any update is executed on a replica, acting as the *prepare-update* of an operation-based CRDT. The $effect()$ method (Algorithm 2) is called on the replicas after receiving the propagated message, acting as the *effect-update*.

Algorithm 1 Prepare Update

```
1: procedure PREPARE(diff)                ▷ on the replica executing a new update
2:     u.init()
3:     u.diff=diff
4:     u.prev.insert(heads)
5:     u.timestamp=HL.currtime
6:     effect(u)
7:     propagate(u)
8:     HL.currtime.increment              ▷ advance the vector clock
```

When a new update u is executed on a replica, the $prepare()$ method initializes a new entry, records the changes in the value t and sets other metadata; then, u takes *HL.heads* as its *prev* pointers.

Algorithm 2 Effect Update

```
1: procedure EFFECT(u)        ▷ on all replicas when receiving the propagated update
2:     if u in HL.log then                                          ▷ see Line 12
3:         update other attributes of u
4:     else
5:         HL.log.insert(u)
6:     if u.timestamp > HL.heads[0].timestamp then
7:         HL.heads ← [u]     ▷ if u is newer than the latest updates on this replica
8:     else if u.timestamp == HL.heads[0].timestamp then
9:         HL.heads.insert(u) ▷ if u is concurrent with the latest updates on this
           replica
10:    HL.currtime.update(u.timestamp)                  ▷ advance the logical clock
11:    for all u' in u.prev do
12:        if u' not in HL.log then          ▷ in case u is propagated before u'
13:            create an empty placeholder u'
14:        u'.aft.insert(u)
```

HL.heads contains the latest known updates on this replica, and u must immediately succeed *HL.heads* in terms of causal ordering. It is then stored in the log.

When a replica receives a propagated u, first, the $effect()$ method updates *HL.log*, then the timestamp is used to determine the causal order between the new update and previous updates. The pointers of the relevant updates are also updated accordingly. For logical reverse, the $u.related$ set is also updated by checking the *Relate* function of Definition 3.8 among new and existing updates.

4.3 Bulk Reverse Approaches

In this section, we discuss how the compensations are applied to the current replica and how this information is propagated.

Lazy compensation. The reverse operation stores only the *start* and *end* timestamps of the causal bulk reverse or the *start* updates of the logical bulk reverse in a $reversed\_list$. At each query operation, a search procedure traverses the *HL.log* to determine the updates that need to be reversed and then applies the compensation to the queried value before returning it to the user. The $reversed\_list$ should be implemented with a unique set CRDT, such as a grow-only set [31], to provide the necessary consistency guarantees.

Eager compensation. Each reverse operation immediately applies the compensating operations to the current state according to the updates that are already recorded on the executing replica. A $reversed\_list$ is still required in this approach. When new updates are propagated from other replicas, they are checked against the $reversed\_list$ to determine whether they are a part of previous bulk reverses. Reading requires no extra procedures in this approach.

Algorithm 3 Lazy Compensation Reverse

```
1: procedure BULKREVERSE(u_s,u_e) ▷ u_s is the start and u_e is the end; u_e is only
   used in the causal bulk reverse
2:     reversed_list.insert((u_s,u_e))
3:     reversed_list.propagate((u_s,u_e)) ▷ synchronize via a grow-only set
       CRDT, details omitted
```

The processes of the reverse operation and query operation for the lazy approach to the causal bulk reverse are specified in Algorithms 3 and 4. The logical bulk reverse can be implemented by modifying $reversed\_list$ to store only the *start* updates and by changing Line 5 in $query()$ to conduct a recursive search with $l(u)$

Algorithm 4 Lazy Compensation Query

```
1: procedure QUERY
2:     temp_state ← state                    ▷ does not modify the current state
3:     to_reverse ← [ ]
4:     for all (u_s,u_e) in reversed_list do
5:         loop traverse from u_s to u_e with breadth-first traversal
6:             for the discovered u, to_reverse.insert(u) ▷ search for the u that
               satisfies Definition 3.7
7:     for all u' in to_reverse do
8:         temp_state += C(u')
9:     return value based on temp_state
```

Algorithm 5 Eager compensation reverse

```
1: procedure BULKREVERSE(u_s,u_e)
2:     reversed_list.insert((u_s,u_e))
3:     reversed_list.propagate((u_s,u_e))
4:     loop traverse from u_s to u_e with breadth-first search
5:         execute C(u)
```

Algorithm 6 Eager compensation on received effect-update

```
1: procedure EFFECT(u)
   ...                                       ▷ see Algorithm 2
2:     for all (u_s,u_e) in reversed_list do
3:         if u_s ≺ u and (u ≺ u_e or u concurrent with u_e) then
4:             execute C(u)
5:             done
```

in Definition 3.9 on the relative set L of the reversed update. The overhead in this approach is derived from the read operations.

The eager approach is shown in Algorithm 5, where compensations are applied during the reverse operation. Algorithm 6 is modified from Algorithm 2 so that whenever a propagated u is received on a replica, the *effect()* method checks whether this update is a part of a previous bulk reverse that is not accounted for. For the logical bulk reverse, a recursive search with $l(u)$ of Definition 3.9 replaces Line 3 of Algorithm 6. The overhead in this approach is mainly derived from the log synchronization instead of the read operations.

4.4 Algorithm Correctness

Here, we show that the algorithms are correct by showing that both approaches adhere to strong eventual consistency. Specifically, when the same updates and op-history log HL entries are received by correct replicas, they will have equivalent states after separately running reverse operations on each replica. This can be proven by showing the following:

(1) HL is an operation-based CRDT (updating HL commutes).
(2) The algorithms of the op-history capture the relations of the updates in Definition 3.3.
(3) Reversing algorithms are deterministic given the same recorded updates, and the compensation either executes exactly once or has no side effect.

Lemma 4.1. *Given two updates, u_j and u_k, $effect(u_j)$ and $effect(u_k)$ of Algorithm 2 commute.*

Proof. We show that with two consecutive $effect(u_j)$ and $effect(u_k)$ executed on a single replica, either order of execution will produce the same state in $HL.log$, $HL.heads$ and each u_j in $HL.log$.

HL.log: Line 3 and Line 5 are where $HL.log$ is updated; both $u_j, u_k \in HL.log$ regardless of the order of execution, because it is incrementally appending to the set.

HL.heads: $HL.heads$ is only updated on Line 7 to Line 11, and there are four cases:

(1) When both $u_j.timestamp$ and $u_k.timestamp < HL.heads[0]$, $HL.heads$ is not updated.
(2) When one of $u_j.timestamp$ or $u_k.timestamp \geq HL.heads[0]$, only one is inserted or set to head regardless of the order.
(3) When $u_j.timestamp == u_k.timestamp \geq HL.heads[0]$, the first arrival is either inserted or set to head, and the second arrival has to be inserted in Line 9.
(4) When $u_j.timestamp > u_k.timestamp \geq HL.heads[0]$, if u_j arrives first, $HL.heads \leftarrow [u_j]$ in line 8, and u_k is disregarded; if u_k arrives first, u_k is inserted or set to $HL.heads[0]$, but as u_j arrives, it overwrites u_k and is set to head.

HL.currtime is only advanced whenever a new update is received, and a vector clock must commute by definition.

For each update *u*: Attributes $u.diff$, $u.prev$, $u.timestamp$ and $u.related$ are never changed after initialization. Attribute $u.aft$ is updated in Line 17, and since $u.prev$ does not change, all of $u' \in u_i.prev$ and $u' \in u_j.prev$ are visited regardless of the order. □

Theorem 4.2. *The op-history log HL is an operation-based CRDT.*

Proof. Define state $S = (HL.log, HL.heads)$ and let each new log entry insertion consist of $prepare()$ and $effect()$ operations. Then, by Lemma 4.1, it is sufficient to conclude that HL is an operation-based CRDT according to the definition since $effect()$ is commutative [7] [32]. □

Lemma 4.3. *For two updates u_i and u_j, if $u_i \prec u_j$, then a path can be created from u_i to u_j by traversing with $u.aft$ pointers.*

Proof. Assume by contradiction that there is no path between u_i and u_j. Without loss of generality, let $u_{i'}, u_{j'}$ be two updates such that both $u_i \prec u_{i+1} ... u_{i'-1} \prec u_{i'}$ and $u_{j'} \prec u_{j'+1} ... u_{j-1} \prec u_j$ have paths between them. The paths can have any length > 0. Additionally, $u_{i'} \prec u_{j'}$ with no update between them.

Since there are paths for $(u_i, u_{i'})$ and $(u_{j'}, u_j)$, no path can exist between $u_{i'} \prec u_{j'}$ because of the assumption. There are two possible cases:

(1) Assume that $u_{i'}$ and $u_{j'}$ are executed on the same replica. When $u_{i'}$ is added, either it is propagated from other replicas or locally executed; Line 8 or Line 10 of $effect()$ is executed because $u_{i'}$ always has a newer timestamp than the previous heads u'_{i-1} by our assumption. Therefore, $u_{i'}$ must be the head of Line 8. Then, by Line 4 of $prepare()$, $u_{i'}$ is inserted into $u_{j'}.prev$, and after Line 17 of $effect()$, u_j is inserted to $u_{i'}.aft$. This process contradicts the assumption that $u_{i'}$ and $u_{j'}$ cannot be executed on the same replica.
(2) Assume that $u_{i'}$ and $u_{j'}$ are executed on different replicas. If $u_{i'}.aft$ does not contain $u_{j'}$, this means that $u_{i'}$ is not in $HL.heads$ of the replica with $u_{j'}$ when $u_{j'}$ is executed, so Line 8 or Line 9 of $effect(u)$ is not executed. This outcome can mean either of two things: $u_{i'}$ is executed before some other update in $HL.heads$, which is impossible because $u_{i'} \prec u_{j'}$ directly; or $effect(u_{i'})$ is not executed, which is also impossible because this means that $u_{i'}$ and $u_{j'}$ are concurrent and contradicts the assumption. Therefore, $u_{i'}$ and $u_{j'}$ cannot be executed on different replicas.

By contradiction, there must be a path connecting $u_{i'}$ and $u_{j'}$ and a path connecting u_i and u_j.

□

Theorem 4.4. *HL.log represents HG in Definition 3.3.*

Proof. Let u in *HL.log* be vertices and the $u.aft$ pointers be edges. It is obvious that *HL* and *HG* have the same properties as shown in Definition 3.1 by Lemma 4.3 and Theorem 3.2. □

We show that *HL* is an operation-based CRDT by Theorem 4.2, and it captures the relations among updates by Theorem 4.4. Since the op-history log represents *HG*, related updates can be located by traversing the graph, according to the causal or logical requirements specified in Definitions 3.7 and 3.9. The *reversed_list* is also a CRDT so it guarantees strong eventual consistency.

For both the lazy and eager approaches, traversal is deterministic since there is no modification to *HL.log*. Applying the compensation is also failure-prune because: for the lazy approach, compensation is applied only on the read value without side effects; for the eager approach, the *reversed_list* (as an append-only unique set CRDT) ensures the only once execution of compensating operation at local replicas.

Therefore, we can conclude that as long as *HL* converges, the reversing algorithms will produce a correct and strongly eventually consistent result on all correct replicas and tolerate any number of failures by the properties of CRDTs under our network assumption.

4.5 Complexity Analysis

Algorithms	Op-History Log Insert	Reverse	Read
Lazy	$O(1)$	$O(1)$	$O(Cr)$
Eager	$O(r)$	$O(C)$	$O(1)$

Table 1: Time complexity of the eager and lazy algorithms

In the lazy approach, Algorithms 1 and 2 for updating *HL* both have $O(1)$ time complexity given that only a constant number of updates are concurrent. Algorithm 3 has $O(1)$ time complexity. Algorithm 4 for read operations has a time complexity of $O(Cr) = O(r)$, where r is derived from reading the entire *reversed_list* and C is the constant representing the number of reversed updates per bulk reverse operation. If *HL* is implemented as a hash table, locating u_s and u_e in *HL.log* is $O(1)$. Traversal only takes the constant time of C, since the updates in each bulk reverse are consecutive and linked by $u.aft$. Both the worst case and the best case are identical because every bulk reverse in the *reversed_list* is always checked. This finding is true for both causal and logical reverses.

In the eager approach, Algorithm 5 has $O(C) = O(1)$ time complexity. Algorithm 6, which modifies Algorithm 2, has a time complexity of $O(Cr) = O(r)$ for traversal similar to the lazy approach.

The space complexity of *HL.log* is $O(n)$, where n is the total number of updates because every update is stored in *HL*. The *reversed_list* has a space complexity of $O(r)$. Memory usage could be a concern in a high-throughput environment due to the high memory footprint of the metadata storage.

4.6 Examples of Reversible CRDTs

Reversible Counter (rCounter). The rCounter is an integer counter CRDT using a PN-Counter [31] as the underlying CRDT. The rCounter has two operations, *increment* and *decrement*, which adds or subtracts an integer from its current value. The compensation for an *increment* operation is a *decrement* of the same value and vice versa. Each update request returns a *uid* (update id) to the client. The *uid*s are arguments of the reverse operations to determine the updates to reverse. The rCounter is designed to support the causal bulk reverse by providing *uid*s of the *start* and the *end* updates.

Reversible Graph (rGraph). The rGraph is built upon the operation-based directed graph CRDT (CRDT-Graph) [32] as the underlying CRDT with the support of the logical bulk reverse. The CRDT-Graph has *addVertex*, *removeVertex*, *addEdge* and *removeEdge* operations.

Compensating operations are the inverse of the corresponding operations (adding edges/vertices vs. removing edges/vertices). Additionally, *addEdge* operations are logically related to *addVertex* operations of the edges' head vertex, which means that if the creation of a head vertex is reversed, all associated edges are removed as part of the logical bulk reverse. Because the creation of an edge semantically relates to the vertices that it is connecting, as adding a new edge requires specifying these vertices. Reversing an *addVertex* operation is different from the *removeVertex* operation: if one attempts to *removeVertex(v)* where v is the head of an edge, it returns a "cannot remove vertex" error to the user. A *uid* is also returned for each update.

4.7 "False-Positive" Updates

The lazy approach may produce "false-positive" updates, that is, with certain CRDTs, some updates seem to be successfully executed even though they should fail because their dependent updates have already been reversed. However, the dependent update remains visible to the system until the lazy read removes it.

For example, if a vertex in the rGraph is reversed, an edge containing this vertex can still be created with the lazy approach. However, with the eager approach, a "vertex not found" error will be returned. This does not violate our guarantees, since the values are still consistent when they are eventually read by the clients, but it may cause issues in certain scenarios. The situation can be mitigated by switching to the eager approach or adding a special check on the validity of the updates.

5 EXPERIMENTAL ANALYSIS

In this section, we present the experimental analysis. The objective of our evaluations is to answer the following questions: How much overhead is introduced by enhancing CRDTs with our reversible algorithms? What are the performance characteristics of a reversible CRDT?

We implemented a replicated key-value (KV) database (rKVCRDT) for evaluations [24]. Multiple rKVCRDT server instances can form a cluster, and each KV pair is replicated among the cluster as CRDTs. Clients send requests that are appropriate to KV pairs' CRDT interface, where values can be read and updated at any server in the cluster. Four data types, the rCounter, PN-Counter, rGraph, and CRDT-Graph, are included.

5.1 Experimental Setup

We hosted an rKVCRDT cluster on five Ubuntu 20.04 cloud VMs provided by the Compute Canada Cloud. Each VM had two Intel Xeon vCPUs clocked at 2.4 GHz, 15 GB of RAM and hosted one server instance. Clients were run on a separate VM within the same cloud location. The VMs were interconnected by high-speed LAN with less than 1 ms latency.

For workload generation, we did not use existing database benchmark tools because they do not consider reversibility in their workloads. In addition, there are no universally applicable benchmarks for CRDTs that can be easily modified to conduct such experiments. Thus, synthetic workloads were used by specifying a read/write (update) ratio and then randomly generating a sequence of requests that were equally assigned to all clients.

5.2 Results

The rCounter and the rGraph were used to demonstrate the performance of reversible CRDTs and their underlying CRDTs, PN-Counter and CRDT-Graph, were employed as baselines.

For each experiment, we first initialized 200 KV pairs, and performed 1000 writes per KV pair with a variable amount of interleaving reverse operations. We measured the performance metrics of an additional 1000 operations (both reads and writes) *after* those reversed had been completed, for a total of 200000 operations. This was because the performance impact of reversed updates persisted after they took effect, as discussed in the last section. Performance was measured with write-heavy workloads that consisted of 30% of reads and 70% of writes, balanced workloads with 50% of reads and 50% of writes, and read-heavy workloads with 70% of reads and 30% of writes.

Throughput vs. Latency and Peak Throughput. Figure 5 depicts the throughput vs. latency for both counters and graphs using both reversing approaches; Figure 6 depicts the peak throughput varied by the number of reverse operations (indicated by r). In Figure 5, we gradually increased the system load in terms of input throughput by changing the number of clients and the rate of requests until they saturated the system (peak throughput), then we measured the *end-of-end* latency.

We noticed that all measurements followed a similar trend where the latency increased slightly while the load increased. Just adding the history log introduced a significant overhead in both latency and peak throughput. For the lazy counter, we observed that when there were few reverses, the write-heavy workload performed better, but when there were more reverses, the read-heavy workload performed better, as shown in Figure 5a and Figure 6a. This was because the lazy approach conducted all compensations during read as discussed in Section 4.5. The read-heavy workload in Figure 5c always performed better in the eager approach, until the gap became narrower with a higher number of reverses.

For graphs, we obtained intersecting results. The lazy graph in Figure 5c and Figure 6a followed a trend similar to that of the lazy counter, but the impact on performance by the number of reverse operations was less pronounced. In the eager graph (Figure 5d), we even observed a performance increase when r increased from 50 to 100. This increase was caused by the expensive implementation of CRDT-Graph and rGraph (baseline peak throughput of PN-Counter was approximately 43,000 ops/s but CRDT-Graph was in the range of approximately 16,000 to 21,000 ops/s) in maintaining a large state: two sets for vertices and edges, and the read operations iteratively checking all elements. Thus, the overhead of reverses was overshadowed when the graph expanded. For the eager graph, the eager compensation effectively *removed* elements from the sets, rendering the checks less expensive.

Throughput Over Executed Operations. In this evaluation, we inserted the reverse operations into experiments that measured throughput to visualize how reverses impacted the performance in real-time. Each KV pair was applied with a 2,000-operation balanced workload in order to maintain a consistent number of updates vs. number of reversed ratio for a total of 400,000 operations.

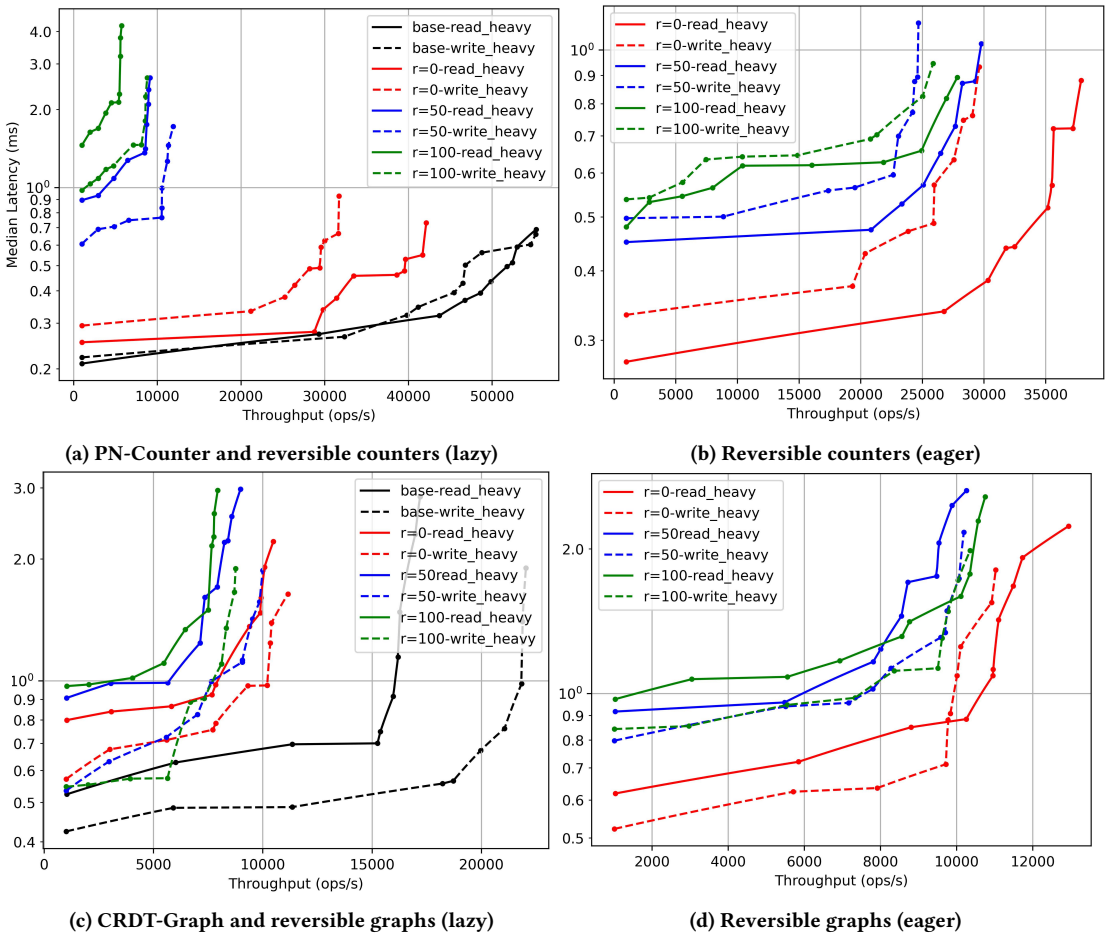
Figure 7a depicts the throughput vs. percentage of executed operations for counters. We noted that lazy counters gradually decreased as the experiments progressed, but eager counters did not. The lazy approach needed to apply increasing compensation as the number of reverses increased, but the eager approach amortized the cost during every update. For graphs, all experiment subjects gradually decreased in performance even with the baseline. This outcome was again caused by the implementation of the CRDT-Graph, whose size constantly expanded.

Latency. To evaluate the system responsiveness and the efficiency of the replication algorithms, we reduced the input throughput to 1,000 ops/s so that the system was under a light load and then measured the latency distribution.

For both counters shown in Figure 9a and Figure 9b, when the number of reverses was low, latency for all operations was comparable to the base PN-Counter where 99% of requests were below 1 ms. When the number of reverses increased, the lazy approach saw a separation in read and write (update) latency. While the write latency remained mostly low with approximately 90% being below 1 ms for $r = 50$ and $r = 100$, the read latency increased to 3 ms and 4 ms, respectively, for the $90th$ percentile. However, neither the read latency nor the write latency changed much with the eager approach.

Figure 9c and Figure 9d depict the latency distribution for graphs. For the lazy approach, the relative increase in the average read latency were less pronounced than those for the rCounter, and we saw that writes were much faster than reads (same for the baseline CRDT-Graph), and adding reverses pronounced the difference. For the lazy approach, we saw that the write latency slightly increased, and the read latency did not change much.

Memory Usage. Figure 8 depicts the final memory usage after the experiments that measured peak throughput by averaging the memory usage at each server. For the rCounter, logging updates in the op-history log consumed 6x to 7x the memory compared to the PN-Counter. The memory usage linearly increased with the number of reverses due to the increase in $reversed\_list$. We also observed that with a read-heavy workload, the lazy approach consumed more memory; with a write-heavy workload, the eager approach consumed more memory. The rGraph consumed approximately 4x to 5x more memory compared to the baseline. However, the memory usage did not show an increase comparable with the number of reverses as the graph decreased in size.



(a) PN-Counter and reversible counters (lazy)

(b) Reversible counters (eager)

(c) CRDT-Graph and reversible graphs (lazy)

(d) Reversible graphs (eager)

Figure 5: Throughput vs. latency of counters and graphs for different number of reverse operations

5.3 Summary and Potential Optimizations

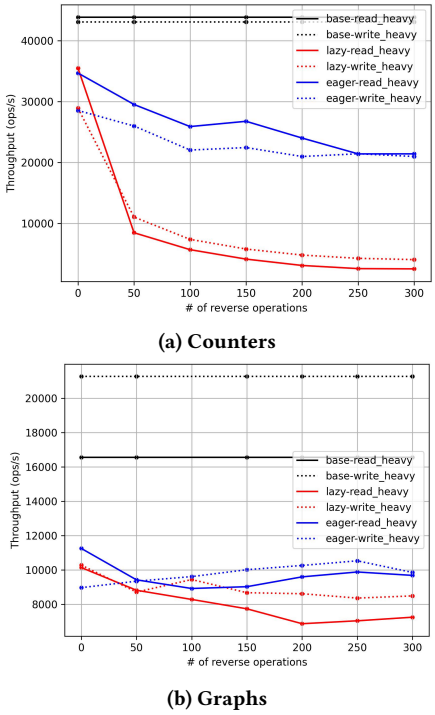
We observed that the performance of reversible CRDTs was heavily influenced by the implementation of the underlying CRDTs. There was no universal conclusion on the performance of all reversible CRDTs but only certain patterns. For lightweight CRDTs such as the PN-Counter, the impact of adding reversibility was significantly larger and more aligned with expectations. However, for CRDTs such as the CRDT-Graph, the overhead may be overshadowed by other factors. This finding indicates that, against intuition, adding reversibility to more sophisticated and "heavier" CRDTs requires fewer trade-offs than simpler CRDTs.

Based on the evaluation results, we also propose a few potential optimization techniques. First is the "hybrid approach": we can apply compensating operations when reverse operations are received as performed in the eager approach, but only check for newly propagated updates on read like the lazy approach. This avoids the repeating application of the same compensations on every read, and also reduces the synchronization messages. Second, we can support check-pointing to perform compaction and garbage collection when all replicas are *stable* (reach the same state) [31], which can greatly reduce the metadata size and log search time. However, there is the trade-off of the longevity of possible reversible updates, as garbage-collected updates can no longer be reversed.

6 RELATED WORK

Although reversibility is useful in many situations, the concept of reversible CRDTs was largely unexplored in previous work when this concept was surveyed by Preguica et al. (2018) [27]. In the survey, the authors mentioned two potential *reversible computation* use cases in CRDTs. The first case involves adding an undo function in CRDTs that are used by collaborative editing tools. This case was already investigated as a nontrivial problem prior to the use of CRDTs in collaborative editing tools [13] [35] [33]. The second case involves a system composed of multiple subsystems, some of which are interdependent, for example, the previously mentioned SAGA transactions [16].

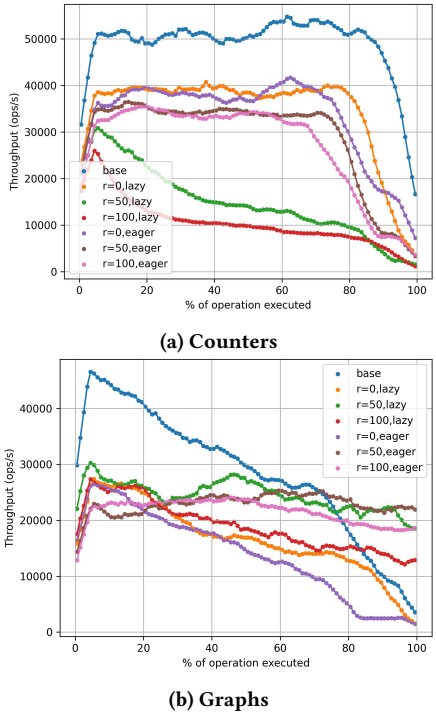
Generic Undo Support for State-based CRDTs. Yu et al. (2020) developed a generic undo support for state-based CRDTs that expanded previous research on *undo* in collaborative editing to any state-based CRDTs [36]. The *state-delta* values are used to represent the changes in states, which are also stored in a log-like structure similar to that of our approach. However, this definition of undoing



(a) Counters

(b) Graphs

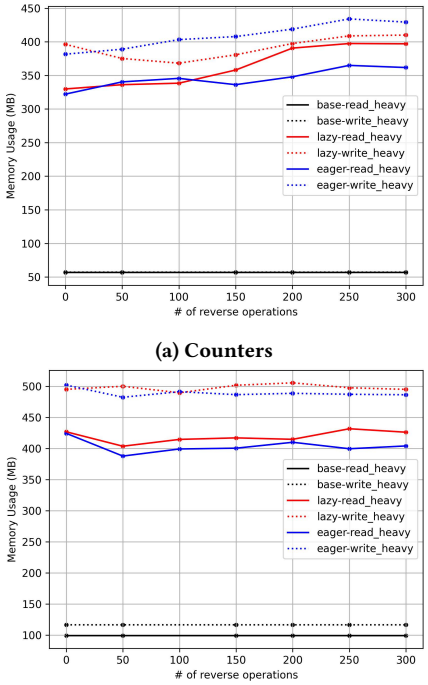
Figure 6: Throughput vs number of reverse operations



(a) Counters

(b) Graphs

Figure 7: Throughput overtime (balanced workload)



(a) Counters

(b) Graphs

Figure 8: Memory usage vs. number of reverse operations

again follows the traditional definition employed in text editors, i.e., reversing the changes made by the last few updates in a last-in, first-out fashion, which does not allow the undoing of arbitrary updates. Therefore, compensating operations do not need to be explicitly defined. Their research focuses on defining the concurrency behaviour for concurrent undo and redo operations instead of defining the removing effect of any given update, such as in our approach.

Pure Operation-Based Replicated Data Types. Baquero, et al. (2014, 2017) [6] [7] proposed a partially ordered log called PO-Log. It is used as a causally reliable broadcast middleware to implement pure-operational based replicated data types. This allows CmRDTs to disseminate *effect-update* messages that neither rely on any current state information nor causal ordering for correct convergence.

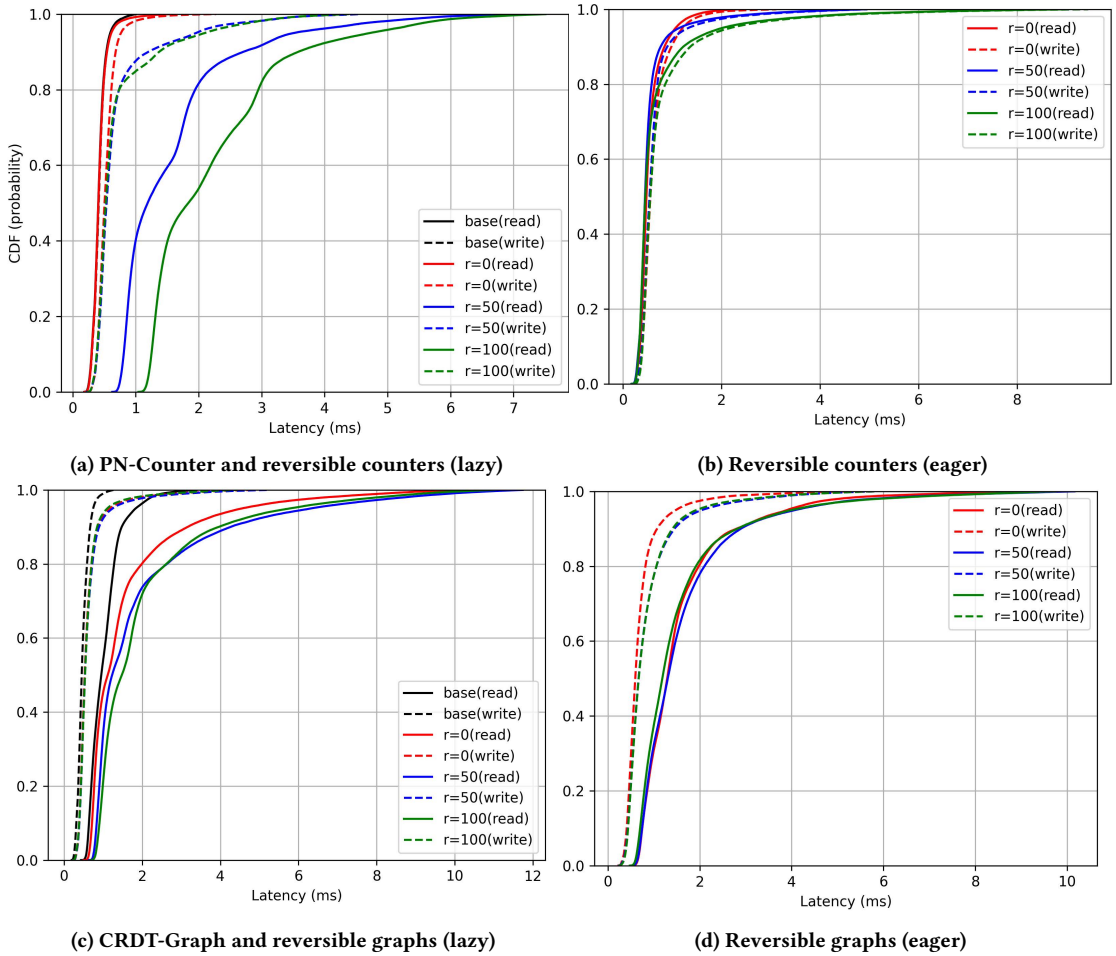
PO-Log is similar to the OP-History log with respect to using vector clocks to determine the causal ordering of updates. However, the OP-history log is also responsible for recording state changes and providing means for searching through the log, whereas PO-Log serves athe different purpose of providing causal information during the *effect-update* process of pure op-based CRDTs. Still, some of their optimization techniques are potentially useful with our OP-history log, such as discarding stable operations as garbage collection.

Hash Graph of Updates. Recent work in CRDTs may allow for more efficient replication of the operation history log. In Kleppmann (2022)'s work [20], the author proposed a method to construct updates' causal dependency graph linked by hash values, similar to certain blockchains [4]. In the hash graph, an update's hash is calculated based on its predecessors' hash. This approach not only ensures the causal dependency but also provides the ability of tamper-resistance. Replication of the hash graph can be accelerated by using a Merkle tree to compare the differences efficiently.

Other Work. In all the previous attempts at making CRDTs reversible, we noticed that the solutions were either limited to specifically designed CRDTs or did not provide the flexibility that we hoped to obtain. We also sought concepts in other areas of distributed systems for intuitions. For example, the *eventually consistent transaction* [11] provides a formalization referred to as the *revision diagram* to depict the history of a series of eventually consistent operations that are similar to our op-history log and the causal bulk reverse definitions. However, in this approach, transactions can never fail, and it is impossible to undo a past operation.

7 CONCLUSIONS

In this paper, we introduced and defined the concept of *reversible CRDTs* based on the idea of compensation. We presented designs and algorithms for extending CRDT systems that allow users to reverse historical updates. As a proof of concept and an evaluation tool, we presented rKVCRDT, a replicated KV store with the support of reversible CRDTs. The evaluations showed that extending CRDTs with reversibility incurs a performance and memory consumption penalty. However, the overall effect is influenced by many other factors, especially the implementation of the underlying CRDTs.



(a) PN-Counter and reversible counters (lazy)

(b) Reversible counters (eager)

(c) CRDT-Graph and reversible graphs (lazy)

(d) Reversible graphs (eager)

Figure 9: Latency CDF for counters and graphs (balanced workload)

In future work, we hope to establish a generic method that offers more predictable behaviour for reversing concurrent updates, as discussed in Section 3.6. This approach requires some type of coordination or stronger consistency requirements on replicas. For example, we can run the coordination algorithm only when the CRDTs' replication mechanisms cannot resolve the concurrency. In addition, we intend to further optimize reversible CRDTs as discussed in Section 5.3.

ACKNOWLEDGMENTS

This work was in part supported by NSERC.

REFERENCES

[1] AntidoteDB. 2021. *Antidote: A planet scale, highly available, transactional database built on CRDT technology.* Retrieved July, 2021 from https://github.com/AntidoteDB/antidote

[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[3] Peter Bailis and Ali Ghodsi. 2013. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Commun. ACM* 56, 5 (May 2013), 55–63. https://doi.org/10.1145/2447976.2447992

[4] Leemon Baird. 2016. *The Swirlds Hashgraph Consensus Algorithm: Fair, fast, Byzantine Fault Tolerance.* Retrieved May 2022 from https://eclass.upatras.gr/modules/document/file.php/CEID1175/Pool-of-Research-Papers%5B0%5D/31.HASH-GRAPH.pdf

[5] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno M. Preguiça. 2015. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *34th IEEE Symposium on Reliable Distributed Systems, SRDS 2015* (Montreal, QC, Canada). IEEE Computer Society, 31–36. https://doi.org/10.1109/SRDS.2015.32

[6] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making Operation-Based CRDTs Operation-Based. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency* (Amsterdam, The Netherlands) *(PaPEC '14)*. Association for Computing Machinery, New York, NY, USA, Article 7, 2 pages. https://doi.org/10.1145/2596631.2596632

[7] Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. 2017. Pure Operation-Based Replicated Data Types. (2017). https://doi.org/10.48550/ARXIV.1710.04469 arXiv:1710.04469

[8] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. An optimized conflict-free replicated set. (2012). https://doi.org/10.48550/ARXIV.1210.3368 arXiv:1210.3368

[9] Peter Bourgon. 2014. *Roshi: a CRDT system for timestamped events.* Retrieved July 2021 from https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events

[10] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (Jan. 2012), 23–29. https://doi.org/10.1109/MC.2012.37

[11] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. 2012. Eventually Consistent Transactions. In *Programming Languages and Systems - 21st European Symposium on Programming* (Tallinn, Estonia) *(ESOP 2012,*

Vol. 7211). Springer, 67–86. https://doi.org/10.1007/978-3-642-28869-2_4

[12] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://doi.org/10.1145/3183713.3196898

[13] David Chen and Chengzheng Sun. 2001. Undoing Any Operation in Collaborative Graphics Editing Systems. In *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work* (Boulder, Colorado, USA) *(GROUP '01)*. Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/500286.500316

[14] George Coulouris, Jean Dollimore, and Tim Kindberg. 2002. *Distributed Systems - Concepts and Designs (3. ed.)*. Addison-Wesley-Longman.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[16] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '87)*. Association for Computing Machinery, New York, NY, USA, 249–259. https://doi.org/10.1145/38713.38742

[17] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. 2021. OWebSync: Seamless Synchronization of Distributed Web Clients. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (March 2021), 2338–2351. https://doi.org/10.1109/TPDS.2021.3066276

[18] Kolbeinn Karlsson, Weitao Jiang, Stephen Wicker, Danny Adams, Edwin Ma, Robbert van Renesse, and Hakim Weatherspoon. 2018. Vegvisir: A Partition-Tolerant Blockchain for the Internet-of-Things. In *38th International Conference on Distributed Computing Systems* (Vienna, Austria) *(ICDCS 2018)*. IEEE Computer Society, 1150–1158. https://doi.org/10.1109/ICDCS.2018.00114

[19] Martin Kleppmann. 2016. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly.

[20] Martin Kleppmann. 2022. Making CRDTs Byzantine Fault Tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data* (Rennes, France) *(PaPoC '22)*. Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3517209.3524042

[21] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. 1990. A Formal Approach to Recovery by Compensating Transactions. In *16th International Conference on Very Large Data Bases* (Brisbane, Queensland, Australia) *(VLDB '90)*. Morgan Kaufmann, 95–106. http://www.vldb.org/conf/1990/P095.PDF

[22] Xiao Lv, Fazhi He, Weiwei Cai, and Yuan Cheng. 2018. Supporting selective undo of string-wise operations for collaborative editing systems. *Future Generation Computer Systems* 82 (May 2018), 41–62. https://doi.org/10.1016/j.future.2017.11.046

[23] Xiao Lv, Fazhi He, Yuan Cheng, and Yiqi Wu. 2018. A novel CRDT-based synchronization method for real-time collaborative CAD systems. *Advanced Engineering Informatics* 38 (Oct. 2018), 381–391. https://doi.org/10.1016/j.aei.2018.08.008

[24] Yunhao Mao. 2022. *rKVCRDT*. Retrieved June 2022 from https://github.com/MSRG/rKVCRDT

[25] Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. 2019. FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 110–122. https://doi.org/10.1145/3361525.3361540

[26] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of Distributed Database Systems, 4th Edition*. Springer. https://doi.org/10.1007/978-3-030-26253-2

[27] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. 2018. Conflict-free Replicated Data Types (CRDTs). (2018). arXiv:1805.06358 http://arxiv.org/abs/1805.06358

[28] Redis. 2021. *Redis: Developing applications with Active-Active databases*. Retrieved July, 2021 from https://docs.redis.com/latest/rs/databases/active-active/develop/data-types/

[29] Riak. 2021. *NoSQL Key Value Database | Riak KV*. Retrieved July, 2021 from https://riak.com/products/riak-kv/

[30] Yasushi Saito and Marc Shapiro. 2005. Optimistic Replication. *ACM Comput. Surv.* 37, 1 (March 2005), 42–81. https://doi.org/10.1145/1057977.1057980

[31] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Technical Report. Inria Centre Paris-Rocquencourt.

[32] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium* (Grenoble, France) *(SSS 2011, Vol. 6976)*. Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[33] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems* (Montreal, QC, Canada) *(ICDCS 2009)*. IEEE Computer Society, 404–412. https://doi.org/10.1109/ICDCS.2009.75

[34] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2021. Anna: A KVS for Any Scale. *IEEE Transactions on Knowledge and Data Engineering* 33, 2 (Feb. 2021), 344–358. https://doi.org/10.1109/TKDE.2019.2898401

[35] Weihai Yu. 2014. Supporting String-Wise Operations and Selective Undo for Peer-to-Peer Group Editing. In *Proceedings of the 18th International Conference on Supporting Group Work* (Sanibel Island, Florida, USA) *(GROUP '14)*. Association for Computing Machinery, New York, NY, USA, 226–237. https://doi.org/10.1145/2660398.2660401

[36] Weihai Yu, Victorien Elvinger, and Claudia-Lavinia Ignat. 2019. A Generic Undo Support for State-Based CRDTs. In *23rd International Conference on Principles of Distributed Systems* (Neuchâtel, Switzerland) *(OPODIS 2019, Vol. 153)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:17. https://doi.org/10.4230/LIPIcs.OPODIS.2019.14