A Study of CRDTs that do Computations

David Navalho  Sérgio Duarte  Nuno Preguiça
NOVA LINCS, FCT, Universidade NOVA de Lisboa

Abstract
A CRDT is a data type specially designed to allow instances to be replicated and modified without coordination, while providing an automatic mechanism to merge concurrent updates that guarantees eventual consistency. In this paper we present a brief study of computational CRDTs, a class of CRDTs whose state is the result of a computation over the executed updates. We propose three generic designs that reduce the amount of information that each replica maintains and propagates for synchronizations. For each of the designs, we discuss the properties that the function being computed needs to satisfy.

1. Introduction
Cloud infrastructures, composed of multiple data centers spread across the globe, have become central for the deployment of novel Internet services, from social networks to business applications. A large number of cloud databases have been developed in recent years, providing different levels of consistency, from strong [5] to eventual consistency [2,6,7].

In this paper we focus on cloud databases that provide eventual consistency only. When using an eventually consistent database, applications can be made highly available by replicating the application code and data in multiple data centers and allowing a user to access any of these data centers. Low latency is achieved by routing the client requests to the closest data center and executing the request in the data center without coordinating with other data centers.

In such settings, concurrent updates may be executed in different replicas. Systems must provide a mechanism to handle concurrent updates and enforce eventual convergence of all replicas. CRDTs [10] have been proposed as a technique for helping application programmers to deal with concurrent updates. They provide eventual consistency with well defined semantics and thus make these systems more amenable to programmers. CRDTs have been adopted as a key feature in a leading cloud database, Riak, and are used in multiple large-scale systems, such as SoundCloud and Twitter’s Summingbird[3].

Most CRDTs proposed in literature are replicated forms of collections. In such data types, a replica needs to maintain all data elements in all replicas. Thus, a model where every data replica maintains the same state and where all updates are propagated to all replicas is natural.

In some cases, applications are not interested in actual elements or updates, but instead on the result of a computation over them. We call computational CRDTs to the class of CRDTs whose state is the result of a computation over the executed updates. For example, a counter CRDT [10] counts the number of times an increment operation has been executed. In such cases, each replica does not need to maintain every individual update, but can instead maintain for each replica an integer that counts the number of increments executed at that replica. For synchronizing replicas, it also suffices to propagate an integer that summarizes a set of updates.

In the remaining of this paper we present a brief study of the properties of computational CRDTs. In particular, we propose three generic designs that minimize the data that needs to be maintained in each replica and that needs to be propagated for synchronizing replicas. We study the properties of functions suitable to each of the designs. Notably, our last design departs from the strict model of state-based CRDTs by the fact that the state of each replica does not need to converge, although the result of all queries executed in every replica is the same.

1.1 Related work
Aggregation techniques have been studied extensively in different settings, such as as sensor networks [11]. Our work can build on the proposed algorithms for creating replicated data types that perform computations in a cloud database.

The techniques used to model CRDTs have been used to express a distributed deterministic dataflow model for concurrent communication between processes [8]. They have also been used to provide algebraic structures for integration between batch and stream processing of aggregations [3] and to support incremental computations [9]. Unlike these works, this paper studies CRDTs that can be integrated in a cloud database as an elementary abstraction to perform computations without requiring additional support from the system.
The problem of optimizing information propagated for synchronizing replicas has been studied by Almeida et al. [1], who have proposed a principled approach to merge the changes produced by multiple operations and use this information to update a remote replica. In our work, the information propagated to synchronize replicas also summarizes multiple updates. However, all information is handled in the context of the CRDT. Additionally, our last design departs from the strict state-based CRDT model by allowing replicas to maintain different states.

2. System model

We adopt the CRDT state-based model [10], where replicas synchronize in a peer-to-peer way, by sending their state to other replicas, where the received state is merged with the current state. A CRDT has an interface that includes update operations that modify the state of the object. In our presentation, we define an event as an invocation of an update operation. For simplicity, we consider a single read-only operation that returns the state of the object. A CRDT includes an additional operation, merge, to merge a copy of a remote replica with the current replica state. In one design, we extend this model to allow a replica to send only a subset of its state to other replicas.

For fault-tolerance, we assume a crash-recovery model, where a replica that fails recovers with its state intact. In a remote replica with the current replica state. In one design, merging two replicas, for the partial result of each replica, the most recently computed result must be kept, which is returned by \( \text{fun}^{\text{max}} \). If the values are monotonic, it is immediate to know what is the most recent version. Otherwise, it might be necessary to maintain this information explicitly. The value of a replica is computed by applying the function \( \text{fun} \) to the contributions of all replicas.

As an example, a positive-negative counter, with an increment and a decrement operations can be defined by making:

\[
\begin{align*}
V_0 &= (0, 0) \\
\text{fun}(\text{inc}) &= (1, 0) \\
\text{fun}(\text{dec}) &= (0, 1) \\
\text{fun}(p, m, (p', m')) &= (p + p', m + m'') \\
\text{fun}^{\text{max}}(p, m, (p', m')) &= (\max(p, p'), \max(m, m'))
\end{align*}
\]

A CRDT that computes the average of values added to an object, which could be used for example to present the average rating in a web application, can be defined by making:

\[
\begin{align*}
V_0 &= (0, 0) \\
\text{fun}(\text{add}(x)) &= (x, 1) \\
\text{fun}(s, c, (s', c')) &= (s + s', c + c') \\
\text{fun}^{\text{max}}(s, c, (s', c')) &= (s, c), \text{iff } c > c' \\
&= (s', c'), \text{iff } c \leq c'
\end{align*}
\]

The average is computed as \( s/c \), with \((s, c)\) the result of the read defined in the generic CRDT design.

Other CRDTs can be defined using a similar approach, including a CRDT that computes a histogram.

3. Design 1: Incremental Computations

Our first design considers computations that can be done incrementally. In this case, computing the function over two disjoint sets of events and combining the results is equal to computing the function over the union of the two sets. Formally, a computation is incremental if there is a function \( \text{fun} \), such that:

\[
\mathcal{F}_{\text{inc}}(E_1 \cup E_2, hb_{E_1 \cup E_2}) = \text{fun}(\mathcal{F}_{\text{inc}}(E_1, hb_{E_1}), \mathcal{F}_{\text{inc}}(E_2, hb_{E_2}))
\]

where \( E_1 \) and \( E_2 \) are disjoint sets of events (operation invocations), \( hb_E \) is a partial causality order on \( E \), and \( \mathcal{F}_{\text{inc}} \) is the function that defines the state of a CRDT that computes \( \text{fun} \) over the observed events (following loosely the formalization proposed by Burckhardt et. al. [4]).

For example, a counter with a single update operation for increment, \( \text{inc} \), can be defined as follows:

\[
\begin{align*}
\mathcal{F}_{\text{ctr}}^-(E, hb) &= |\{ e \in E : e = \text{inc}\}| \\
\mathcal{F}_{\text{ctr}}^+(E_1 \cup E_2, hb) &= \mathcal{F}_{\text{ctr}}^+(E_1, hb) + \mathcal{F}_{\text{ctr}}^+(E_2, hb)
\end{align*}
\]

For these computations, Figure 1 presents a generic CRDT design that is parameterized by the following elements: (i) \( V_0 \), the initial state associated with a replica; (ii) \( \text{fun}(o) \), the value of the computation for a single operation \( o \); (iii) \( \text{fun}(s_1, s_2) \), the function to compose two partial results; and (iv) \( \text{fun}^{\text{max}}(v_1, v_2) \), that returns the latest of two values.

In this design, each replica computes its contribution to the final value of the CRDT independently. Each replica maintains a map for the contributions of each replica. When executing an update operation, a replica updates its contribution by using function \( \text{fun} \) to combine the previous computed contribution and the contribution of the new operation (with \( s[i] \mapsto \text{fun}(s[i], \text{fun}(\text{op})) \)) representing the replacement in \( s \) of the value of entry \( i \) by the new computed value). When merging two replicas, for the partial result of each replica, the most recently computed result must be kept, which is returned by \( \text{fun}^{\text{max}} \). If the values are monotonic, it is immediate to know what is the most recent version. Otherwise, it might be necessary to maintain this information explicitly. The value of a replica is computed by applying the function \( \text{fun} \) to the contributions of all replicas.

As an example, a positive-negative counter, with an increment and a decrement operations can be defined by making:

\[
\begin{align*}
V_0 &= (0, 0) \\
\text{fun}(\text{inc}) &= (1, 0) \\
\text{fun}(\text{dec}) &= (0, 1) \\
\text{fun}(p, m, (p', m')) &= (p + p', m + m'') \\
\text{fun}^{\text{max}}(p, m, (p', m')) &= (\max(p, p'), \max(m, m'))
\end{align*}
\]

A CRDT that computes the average of values added to an object, which could be used for example to present the average rating in a web application, can be defined by making:

\[
\begin{align*}
V_0 &= (0, 0) \\
\text{fun}(\text{add}(x)) &= (x, 1) \\
\text{fun}(s, c, (s', c')) &= (s + s', c + c') \\
\text{fun}^{\text{max}}(s, c, (s', c')) &= (s, c), \text{iff } c > c' \\
&= (s', c'), \text{iff } c \leq c'
\end{align*}
\]

The average is computed as \( s/c \), with \((s, c)\) the result of the read defined in the generic CRDT design.

Other CRDTs can be defined using a similar approach, including a CRDT that computes a histogram.

4. Design 2: Incremental Idempotent Computations

In some cases, the computation to be performed besides being incremental is also idempotent. In this case, computing the function over two (potentially overlapping) sets of events and combining the results is equal to computing the function over the union of the two sets. Formally, a computation is incremental and idempotent if there is a function \( \text{fun} \), such that for any sets of events \( E_1 \) and \( E_2 \) we have:

\[
\mathcal{F}_{\text{inc}}(E_1 \cup E_2, hb) = \text{fun}(\mathcal{F}_{\text{inc}}(E_1, hb), \mathcal{F}_{\text{inc}}(E_2, hb))
\]
We now consider computations that are only partially incre-
mental property previously defined, while others do not. An
example of such an object is a top-K object where an element
can be deleted. In such cases, a value that does not belong to
the top-K elements may later become part of the top, after a
top element is deleted.

To address this case, a possible approach is to use a Set
CRDT to maintain the set of elements that have not been
deleted. In this case, all replicas maintain the complete set,
and all updates need to be propagated to all replicas. The
top-K can be computed locally on the value of each replica.

In Figure 3 we present an alternative approach, in which
each replica maintains all operations locally executed, and
each replica only propagates to other replicas the operations
that might affect the computed result. Each replica maintains
a set of operations and the results of the computation per-
fomed at other sites — for simplicity of notation, we assume
that the result of the computation is a subset of operations.
An update operation updates the local set of operations. A
read operation makes the computation considering the local
operations and the results of the computation at the other
replicas. For synchronizing replicas, a replica sends the re-
sults of the computations to all replicas and the subset of
operations and the results of the computation at the other
replicas. When receiving the state from a remote replica, the local replica is updated by merging
the local set of operations with the remote operations that
may affect the result of the computation, and by registering
the most recent version of the computation for each site.

A top-K replicated data type that supports an add \((n, v)\)
and del \((n)\) operations can be defined as follows:

\[
\begin{align*}
V_0 &= \emptyset \\
\text{fun}(n, v) &= \{n, v\} \\
\text{fun}(s, s') &= \max_k\left\{\{n, v\} : \exists\ (n, v_1) \in (s \cup s') : v_1 > v\right\}
\end{align*}
\]

with \(\max_k(s)\) a function that returns the \(k\) largest elements
\((n, v) \in s\), with the elements ordered using the total order
defined previously.

In general, this approach can be used to create CRDTs that
compute a filter over the values added to the object, for which
an element that does not match the filter at some moment will
not match the filter at a later moment.

5. Design 3: Partially Incremental Computations

We now consider computations that are only partially incre-
mental, in the sense that some updates observe the incre-
mentally previously defined, while others do not. An
example of such an object is a top-K object where an element
can be deleted. In such cases, a value that does not belong to
Replica state $\Sigma = (P(op), I \rightarrow V)$
Initial state $\sigma^0 = (\{1\}, i \rightarrow V_0)$
Update op at replica $i$ $op_i((s, m)) = (s \cup \{op\}, m)$
Read at replica $i$ $op_i((s, m)) = \text{fun} (s \bigcup m[j])$
State to send from replica $i$ $\text{diff}(s, m) = (a \in s : \text{fun}(a \bigcup m[j]) \not= \text{fun}(\bigcup m[j]), m[i \rightarrow \text{fun}(s \bigcup m[j])])$
Merge replica states $\text{deliver}((s, m), (s', m')) = (s \cup s', m[j] \rightarrow \text{fun}^{ast}(m[j], m'[j]) \forall j)$

Figure 3: Generic replicated data type for partially incremental computation.

6. Final remarks

In this paper we have proposed three generic designs for replicated data types that perform a computation on the operations executed by users. These designs can be used in a system that maintains CRDT replicas at multiple sites and synchronizes them using a state-based model. We present the properties that computations must obey in order to use each of the designs. These designs try to minimize the information that each replica has to maintain and propagate to other replicas for synchronization.

The last proposed design departs from the strict CRDT state-based model, while still enforcing eventual consistency. We are currently formalizing the new model and studying the relations between replicated data types implemented using this design and state-based CRDTs that implement the same functionality. In the future, we intend to study how to integrate these designs in an eventually consistent cloud database, such as Riak.

Acknowledgments

This research is supported in part by FCT scholarship SFRH/BD/65070/2009, FCT projects PTDC/EEL-SCR/1837/2012 and PEst-OE/EEL/UI0527/2014 and EU FP7 SyncFree project (609551).

References


