CRDTs in Production

Jeremy Ong

Abstract

TODO

I. Introduction

There has been increasing interest in the usage of CRDTs in environments of relaxed temporal constraints. With the maturity of several distributed AP databases, developers now have a choice to emphasize availability in exchange for immediate consistency. These databases necessarily require distributed data to be subject to possible logical inconsistencies due to presence of multiple writers and readers in addition to replication latency. CRDTs, commutative or convergent replicated data types, offer a formal and structured approach to ensuring eventual consistency by enforcing the following conditions: termination, eventual effect, and convergence [3]. Many CRDTs have been identified of varying complexities and applicability (e.g. treedoc, registers, pn-counters) [2,3].

However, adoption of these CRDTs into production databases has been slow. In addition, it is not always practical to represent every application-sensitive database object as a CRDT. While it is likely that all objects can be represented as a set of disjoint CRDTs, this can exacerbate the logical divergence of inter-key relationships due to a lack of multi-key transactions.

A more pragmatic approach is to employ application level custom convergence code, modeled after CRDT concepts to manage conflict. Because of the generalized nature of this technique, it can be difficult to formalize. This write-up intends to discuss various common scenarios and show how by relaxing certain constraints, behavior very close to a CRDT can be observed. In situations where operations must be applied that cannot commute, mitigation strategies are introduced that reduce the operation in question to a delayed transaction.

II. Preliminaries

Let $x \in X$ denote an object in application space and $f \in F : X \to X$ an operation that acts on an element of X. Two versions of the same object are differentiated by subscript. The causal history of an object x, $C(x)$ is defined as a set of operations in terms of the following properties:

1. Initially, the causal history is an empty set: $C(x_0) = \emptyset$
2. A single atomic update augments the causal history: $C(x_{i+1}) = C(x_i) \cup \{f\}$
3. The causal history of a merged object is the union of causal history of the objects merged: $C(x_i \cup x_j) = C(x_i) \cup C(x_j)$

We can define a partial ordering on objects in X based on causal histories.

$$x_i \leq x_j \Leftrightarrow C\left(x_i\right) \subset C\left(x_j\right)$$

In words, this means that all operations that have occurred to x_i have also occurred to x_j.

CRDTs themselves are divided into two categories: CvRDTs (convergent replicated data types) are state based while CmRDTs (commutative replicated data types) are operation based. When clients request an operation to be applied, they must first retrieve a replica

on which base the request (termed the source replica). The operation is then applied either at the source (CvRDT) or downstream for all replicas of that object (CmRDT). The at-source and downstream phases are subject to preconditions. Both CvRDTs and CmRDTS must satisfy the following conditions in the traditional definition of a CRDT [3]:

1. Termination: The at-source and downstream phases terminate when preconditions are satisfied
2. Eventual Effect:

   $$\forall i \exists j \ni x_j \leq x_k \Leftrightarrow f \in C(x_i) \Rightarrow f \in C(x_k)$$

   Colloquially, if an operation is performed on an object, that operation will eventually exist in the causal history of all replicas.
3. Convergence:

   $$C(x_i) = C(x_j) \Rightarrow x_i \equiv x_j$$

   Replicas with identical histories contain the same state.

III. Managing Consistency in Practice

This paper is restricted to circumstances in which the developer has already made the tradeoff of leaning towards availability and relaxing temporal consistency in selecting a database. Otherwise, all operations would be wrapped in a transaction for which immediate consistency is guaranteed. Given this tradeoff, there are several alternatives that should also be considered before immediately implementing any merge strategy.

First, the developer has an option to serialize all writes to a particular object in a single queue. In practice, this will guarantee immediate consistency again but at the cost of availability gained at the database level. Nevertheless, this may be useful if immediate consistency is only needed for a small fraction of the keys for which availability is less important. If this strategy is to be used on the majority of keys, the developer is better off choosing a CP database. Second, the developer has the option to implement a locking system. This is similar to the first technique but allows the locking layer to be scaled independently of the writers which may be distributed across many machines.

In general, because there is no easy way to manage multi-key transactions in an eventually consistent setting, correlated data must be denormalized in a single key value pair. For example, if an operation to $x \in X$ is contingent on the result of an operation on $y \in Y$, the developer is better off denormalizing X and Y into a single object type $Z = X \times Y$ rather than attempting to coordinate the logical dependency.

A merge strategy can be useful if the data in question exhibits low merge intensity. Qualitatively, merge intensity corresponds to the fraction of writes that will resolve in the formation of replicas. Factors that correspond to low merge intensity include:

- Low write frequency to a single key for a given writer
- Few writers to a single key
- Short timespan between a key read and subsequent write

Conversely, factors that correspond to high merge intensity include:

- Low write frequency to a single key for a given writer
- Few writers to a single key
- Short timespan between a key read and subsequent write

By restricting the usage of a custom merge strategy to cases where merge intensity is low, the developer avoids scenarios where merges become inefficient or even impossible in certain scenarios.

Instead of a custom merge strategy of course, the developer may use one of the canon-

ical CRDTs. In practice, CRDTs are rarely implemented at the database level, although database companies are beginning to incorporate the simpler ones. At the time of this writing, Riak has implemented a PN-counter for example [1] with more data types on the way. Unfortunately, application data still exhibits more complexity than can be handled natively with most databases, so we are restricted with handling operations and merge logic on the application side. In addition, not all application operations commute in general, and yet these operations must still occur. Incorporating such operations in the data model will require the relaxation on some of the canonical CRDT constraints.

IV. Relaxing CRDT Constraints

Instead of the strict CRDT constraints imposed on the canonical CRDTs, I present here a set of relaxed constraints for a more general type of CRDT. First, adding to notation introduced previously, let $\overline{f}$ represent an operation attempt that may or may not complete successfully depending on some set of conditions specific to the operation f.

1. Termination: as before
2. Eventual Effect:

   $$\forall i \exists j \ni x_j \leq x_k \Leftrightarrow f \in C(x_i) \Rightarrow \overline{f} \in C(x_k)$$

   If an operation is performed on an object, the attempt to perform that operation will exist in the causal history of all replicas
3. All or nothing: An operation will either succeed with side effects, or fail with no side effects

V. Discussion

VI. Conclusion

References

[1] Basho. Counters in riak 1.4. `http://basho.com/counters-in-riak-1-4/`, 2013.

[2] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Crdts: Consistency without concurrency control. *arXiv preprint arXiv:0907.0929*, 2009.

[3] Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski, P Fatourou, et al. Convergent and commutative replicated data types. *Bulletin of the EATCS*, (104):67–88, 2011.