CRDTs

Background

Many database systems today are moving more towards a multi-leader/leaderless replication system in order to increase write throughput, which inevitably leads to conflicts amongst the databases.

As such, we have seen the invention of things like Conflict Free Replicated Data Types (CRDT for short) pop up, which aim to implement certain data structures like counters, sets, maps, and lists over a multileader replication setup.

The goal of CRDTs is convergence: all replicas will eventually show the same value for the data structures without losing any of the updates.

CRDT use cases

- Collaborative editing
  - Can use a CRDT here or an algorithm called operational transform
- Online chat systems
  - To ensure ordering of the chats is eventually the same
- Any application that allows offline editing
- Distributed leaderless key value stores such as Riak and Redis
  - Hence the reason for this video

Types of CRDTs

- Operation Based CRDTs
  - Propagate state by only transmitting the local update operation to other nodes
  - Useful for when state is very large and expensive to transmit over the network and when there are relatively few operations compared to the size of state
  - Not idempotent, network needs to ensure they are delivered only once
- State Based CRDTs
  - Send the entire local CRDT state over the network to a remote node, remote node merges it with its own local CRDT version
  - The merge function must be commutative and idempotent
    - The order of merges doesn’t matter
    - Calling the merge function multiple times with the same inputs has no effect
  - Works very well with gossip protocols

Examples of CRDTs

- Grow Only Counter
- Incrementing and Decrementing Counter
- Sets
- Sequence

Grow Only Counter

[0, 0]

[0, 0]

1) Each node in the database starts off with an array of 0s (number of elements in array equal to number of nodes)

Grow Only Counter

[1, 0]

[0, 0]

1) Each node in the database starts off with an array of 0s (number of elements in array equal to number of nodes)
2) Jordan writes “increment”, request handled by replica 1, which increments its own counter in its list

Grow Only Counter

[1, 0]

[0, 5]

1) Each node in the database starts off with an array of 0s (number of elements in array equal to number of nodes)
2) Jordan writes “increment”, request handled by replica 1, which increments its own counter in its list
3) Client B writes “increment” five times, handled by replica 2, which increments its own counter in its list

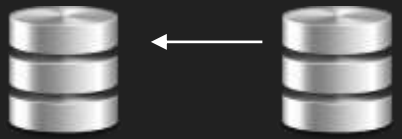
Grow Only Counter

[1, 0]

[0, 5]

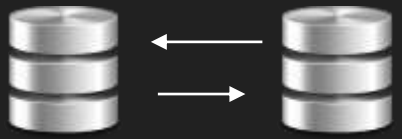
1) Each node in the database starts off with an array of 0s (number of elements in array equal to number of nodes)
2) Jordan writes “increment”, request handled by replica 1, which increments its own counter in its list
3) Client B writes “increment” five times, handled by replica 2, which increments its own counter in its list
4) Jordan queries the counter value, request handled by replica 1, returns 1 (sum of local array)

Grow Only Counter



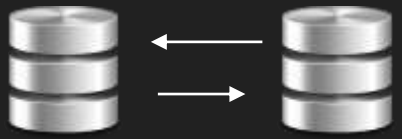
1) Each node in the database starts off with an array of 0s (number of elements in array equal to number of nodes)
2) Jordan writes “increment”, request handled by replica 1, which increments its own counter in its list
3) Client B writes “increment” five times, handled by replica 2, which increments its own counter in its list
4) Jordan queries the counter value, request handled by replica 1, returns 1 (sum of local array)
5) Changes from replica 2 merged into replica 1, merge function just chooses the maximum element of each index in the list

Grow Only Counter



1) Each node in the database starts off with an array of 0s (number of elements in array equal to number of nodes)
2) Jordan writes “increment”, request handled by replica 1, which increments its own counter in its list
3) Client B writes “increment” five times, handled by replica 2, which increments its own counter in its list
4) Jordan queries the counter value, request handled by replica 1, returns 1 (sum of local array)
5) Changes from replica 2 merged into replica 1, merge function just chooses the maximum element of each index in the list
6) Changes from replica 1 merged into replica 2, merge function just chooses the maximum element of each index in the list

Grow Only Counter



1) Each node in the database starts off with an array of 0s (number of elements in array equal to number of nodes)
2) Jordan writes “increment”, request handled by replica 1, which increments its own counter in its list
3) Client B writes “increment” five times, handled by replica 2, which increments its own counter in its list
4) Jordan queries the counter value, request handled by replica 1, returns 1 (sum of local array)
5) Changes from replica 2 merged into replica 1, merge function just chooses the maximum element of each index in the list
6) Changes from replica 1 merged into replica 2, merge function just chooses the maximum element of each index in the list
7) Querying the counter value from either node will now return the value 6 (sum of all elements in the array)

Incrementing and Decrementing Counter

Basically the same as the grow only counter, with the following adjustments:

- Each node keeps track of two arrays
  - One for increments that it has seen
  - One for decrements that it has seen (clients can now issue a “decrement operation”)
- To get the counter value
  - Sum up the increments array, and subtract the sum of the decrements array
- To merge the counter values
  - The merged increment array is the element-wise max of the two increment arrays
  - The merged decrement array is the element-wise max of the two decrement arrays

Sets

Basically the same as the grow only counter, with the following adjustments:

- Each node keeps track of two arrays
  - One for elements it has added
  - One for elements it has removed
- To get the set contents
  - Take the added set and remove all of the elements in the remove set
- To merge two sets
  - The merged added set is the union of the two added sets
  - The merged removed set is the union of the two removed sets

Sets Continued

You may notice that with sets, once an element is in the removed set, it can never be re-added!

To mitigate this, some variations have been created:

- Every element in the added and removed set has a timestamp
  - If element is in both add set and remove set, the one with the higher timestamp prevails
  - This obviously poses some issues due to unreliable clocks in distributed systems
- Attach a unique tag to every element in the add set (now can put the same element in the add set many times with different tags)
  - When removing that element put said tag in the remove set
  - An element is only present if it is still present in the add set less the remove set (do not count all element, tag tuples in the add set that are in the remove set)

Sequence CRDTs

Super useful for things like collaborative text editing where we need to determine an order of the elements in a document.

Won’t touch upon these at the moment, because text editing CRDTs and operational transform deserve their own video!

CRDTs Conclusion

Very useful data structures to ensure convergence/conflict resolution in database configurations with multiple leaders. Are often used in conjunction with (or are implemented with logic similar to) version vectors.

Since this is one of the main differentiating features of databases like Riak, it is important to understand how CRDTs before we talk about them!