CRDTs in Production

Jeremy Ong

October 27, 2013

Objective

- Understand tradeoffs developers must consider when interfacing with an AP database (Riak specifically this talk)

Objective

- Understand tradeoffs developers must consider when interfacing with an AP database (Riak specifically this talk)
- Learn the knobs and levers at our disposal to manage consistency

Before you reject ACID...

...understand the withdrawal symptoms

Before you reject ACID...

...understand the withdrawal symptoms

- relationships between keys loosened

Before you reject ACID...

...understand the withdrawal symptoms

- relationships between keys loosened
- loss of immediate consistency

Before you reject ACID...

...understand the withdrawal symptoms

- relationships between keys loosened
- loss of immediate consistency

Why leave in the first place?

A not C please

Sometimes, availability is more valuable than *immediate* consistency.

A not C please

Sometimes, availability is more valuable than *immediate* consistency.

- Data sets are independent (user profile, shopping cart)

A not C please

Sometimes, availability is more valuable than *immediate* consistency.

- Data sets are independent (user profile, shopping cart)
- Writes or reads should still be allowable in the presence of network partition of hardware failure

A not C please

Sometimes, availability is more valuable than *immediate* consistency.

- Data sets are independent (user profile, shopping cart)
- Writes or reads should still be allowable in the presence of network partition of hardware failure
- Scalability is a real concern

A not C please

Sometimes, availability is more valuable than *immediate* consistency.

- Data sets are independent (user profile, shopping cart)
- Writes or reads should still be allowable in the presence of network partition of hardware failure
- Scalability is a real concern

Desire: Cure the ACID hangover

Low conflict intensity

Excellent candidates for a CRDT approach exhibit low conflict intensity:

Low conflict intensity

Excellent candidates for a CRDT approach exhibit low conflict intensity:

- One writer to a given key

Low conflict intensity

Excellent candidates for a CRDT approach exhibit low conflict intensity:

- One writer to a given key
- Majority of writes come from a single writer

Low conflict intensity

Excellent candidates for a CRDT approach exhibit low conflict intensity:

- One writer to a given key
- Majority of writes come from a single writer
- Writes occur sparingly

CRDT Preliminaries

Consensus: RDTs are replicated data types.

CRDT Preliminaries

Consensus: RDTs are replicated data types.

Does the C in CRDT mean:

- Convergent? or...
- Commutative?

CRDT Preliminaries

Consensus: RDTs are replicated data types.

Does the C in CRDT mean:

- Convergent? or...
- Commutative?

It depends!

- Convergent - state based RDTs
- Commutative - op based RDTs

CvRDTs

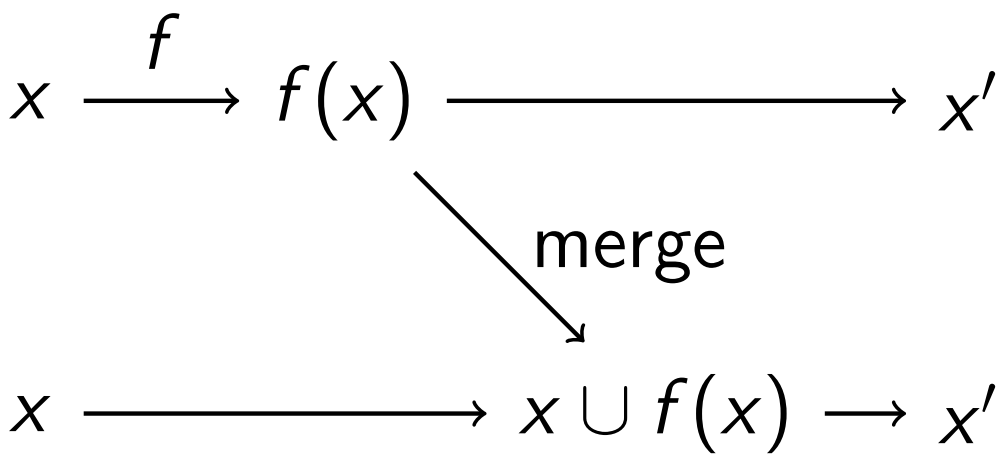
Definition

Convergent RDT (CvRDT) - State based datatype that relies on merges to synchronize divergent replica copies (e.g. git)

CvRDTs

Definition

Convergent RDT (CvRDT) - State based datatype that relies on merges to synchronize divergent replica copies (e.g. git)



CmRDTs

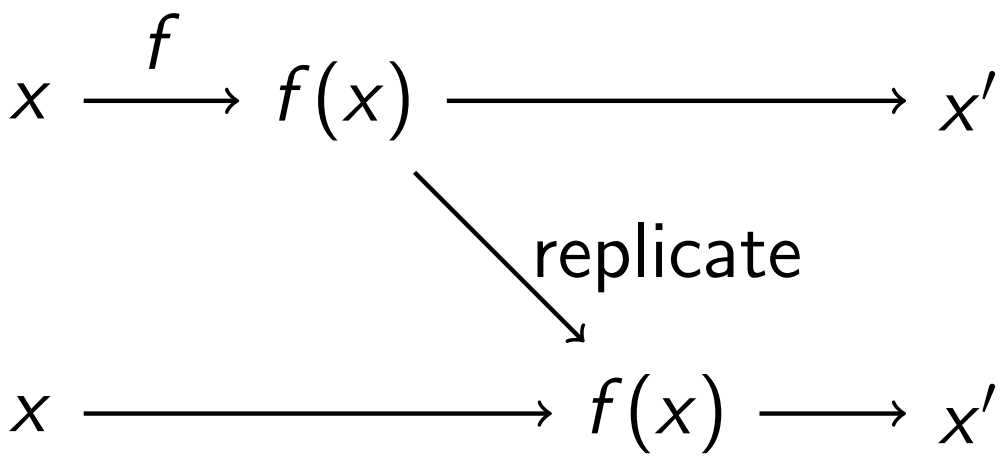
Definition

Commutative RDT (CmRDT) - Operation based datatype that replicates commutative operations across divergent replicas (e.g. Simon Says)

CmRDTs

Definition

Commutative RDT (CmRDT) - Operation based datatype that replicates commutative operations across divergent replicas (e.g. Simon Says)



CRDT Constraints

1. Termination: At-source and/or downstream phases terminate when preconditions are satisfied.

CRDT Constraints

1. Termination: At-source and/or downstream phases terminate when preconditions are satisfied.
2. Eventual Effect (violated by LWW):

$$\forall x_i \exists x_j \geq x_i \ni x_j \leq x_k \Leftrightarrow C(x_i) \subseteq C(x_k)$$

CRDT Constraints

1. Termination: At-source and/or downstream phases terminate when preconditions are satisfied.
2. Eventual Effect (violated by LWW):

$$\forall x_i \exists x_j \geq x_i \ni x_j \leq x_k \Leftrightarrow C(x_i) \subseteq C(x_k)$$

3. Convergence:

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

Application side CvRDT case study

CRDTs are an app developers responsibility (not just a database implementer's responsibility). Why?

Application side CvRDT case study

CRDTs are an app developers responsibility (not just a database implementer's responsibility).
Why?
Data denormalization!

Application side CvRDT case study

CRDTs are an app developers responsibility (not just a database implementer's responsibility).
Why?
Data denormalization! Your database can't be expected to understand your schema.

Application side CvRDT case study

CRDTs are an app developers responsibility (not just a database implementer's responsibility).
Why?
Data denormalization! Your database can't be expected to understand your schema.
Replica copies don't just exist in the database (think caching for example).

Application side CvRDT case study

Merges vs Commutative ops (why not a CmRDT)

Application side CvRDT case study

Merges vs Commutative ops (why not a CmRDT)

- Maintainability - Avoid the merge function of doom (unless your data is canonical and merge-oriented)

Application side CvRDT case study

Merges vs Commutative ops (why not a CmRDT)

- Maintainability - Avoid the merge function of doom (unless your data is canonical and merge-oriented)
- Tight coupling of operation with resolution behavior

Application side CvRDT case study

Merges vs Commutative ops (why not a CmRDT)

- Maintainability - Avoid the merge function of doom (unless your data is canonical and merge-oriented)
- Tight coupling of operation with resolution behavior
- Resilience against schema changes

Shopping Cart Implementation

The schema is of the form (d, f), a tuple combining shopping cart data (d) and the optional last operation used (initially $\varnothing$).

Shopping Cart Implementation

The schema is of the form (d, f), a tuple combining shopping cart data (d) and the optional last operation used (initially $\varnothing$).

Definition

`apply(d, f)` - Given data and an operation, returns the tuple $(f(d), f)$. Then write to the database.

Shopping Cart Impl. 2

Definition

`resolve`$(\{s_1, s_2, \ldots, s_n\})$ - Given a list of siblings, returns the tuple $(f_2 \circ f_3 \circ \cdots \circ f_n(d_1), \varnothing)$. Then issues a write to the database.

Example: AddToCart

```
void AddToCart(Cart &cart,
               const Item &item,
               const unsigned int quantity) {
    if (cart.HasItem(item)) {
        cart.Get(item) += quantity;
    } else {
        cart.AddItem(item, quantity);
    }
}
```

When things don't commute

The function `AddToCart` behaves nicely because it commutes. What about something like `RemoveFromCart`?

When things don't commute

The function `AddToCart` behaves nicely because it commutes. What about something like `RemoveFromCart`?

```
void RemoveFromCart(Cart &cart,
                    const Item &item) {
    if (cart.HasItem(item)) {
        ... item removal code
    } else {
        // What should I do here?
```

When things don't commute

The function `AddToCart` behaves nicely because it commutes. What about something like `RemoveFromCart`?

```
void RemoveFromCart(Cart &cart,
                    const Item &item) {
    if (cart.HasItem(item)) {
        ... item removal code
    } else {
        // What should I do here?

        // nothing
    }
}
```

Isn't that the same as LWW?

Isn't that the same as LWW?

No.

Isn't that the same as LWW?

No.
LWW would handle the case of duplicate removals of the same object, but doesn't handle conflicts of heterogeneous operations.

Shopping Cart Summary

The shopping cart is easy because

- The cart behaves like a set of PN-counters

Shopping Cart Summary

The shopping cart is easy because

- The cart behaves like a set of PN-counters
- Operations that don't commute on a cart can simply be nullified if a condition is void

Shopping Cart Summary

The shopping cart is easy because

- The cart behaves like a set of PN-counters
- Operations that don't commute on a cart can simply be nullified if a condition is void

We are essentially using *deferred transactions*.

Shopping Cart Summary

The shopping cart is easy because

- The cart behaves like a set of PN-counters
- Operations that don't commute on a cart can simply be nullified if a condition is void

We are essentially using *deferred transactions*. Consider the behavior of the shopping cart if we did not use this interface.

Relaxed CRDT constraints

1. Termination: unchanged

Relaxed CRDT constraints

1. Termination: unchanged
2. Eventual Effect (relaxed): For every operation on some x_i, an attempt to perform that operation is eventually performed on all later replica copies.

Relaxed CRDT constraints

1. Termination: unchanged
2. Eventual Effect (relaxed): For every operation on some x_i, an attempt to perform that operation is eventually performed on all later replica copies.
3. Convergence (relaxed): Replicas converge in the absence of write conflict but the worst case effects of a conflict are loss of operation.

Differences from a standard transaction

- Op cancellation is code dependent

Differences from a standard transaction

- Op cancellation is code dependent
- Client is not immediately aware of issue (it's eventual)

Differences from a standard transaction

- Op cancellation is code dependent
- Client is not immediately aware of issue (it's eventual)
- Local to a single object.

Differences from a standard transaction

- Op cancellation is code dependent
- Client is not immediately aware of issue (it's eventual)
- Local to a single object.
- Passive vs active merge resolution

Improving our database interface

- Spawn events when an op gets canceled so client can be notified

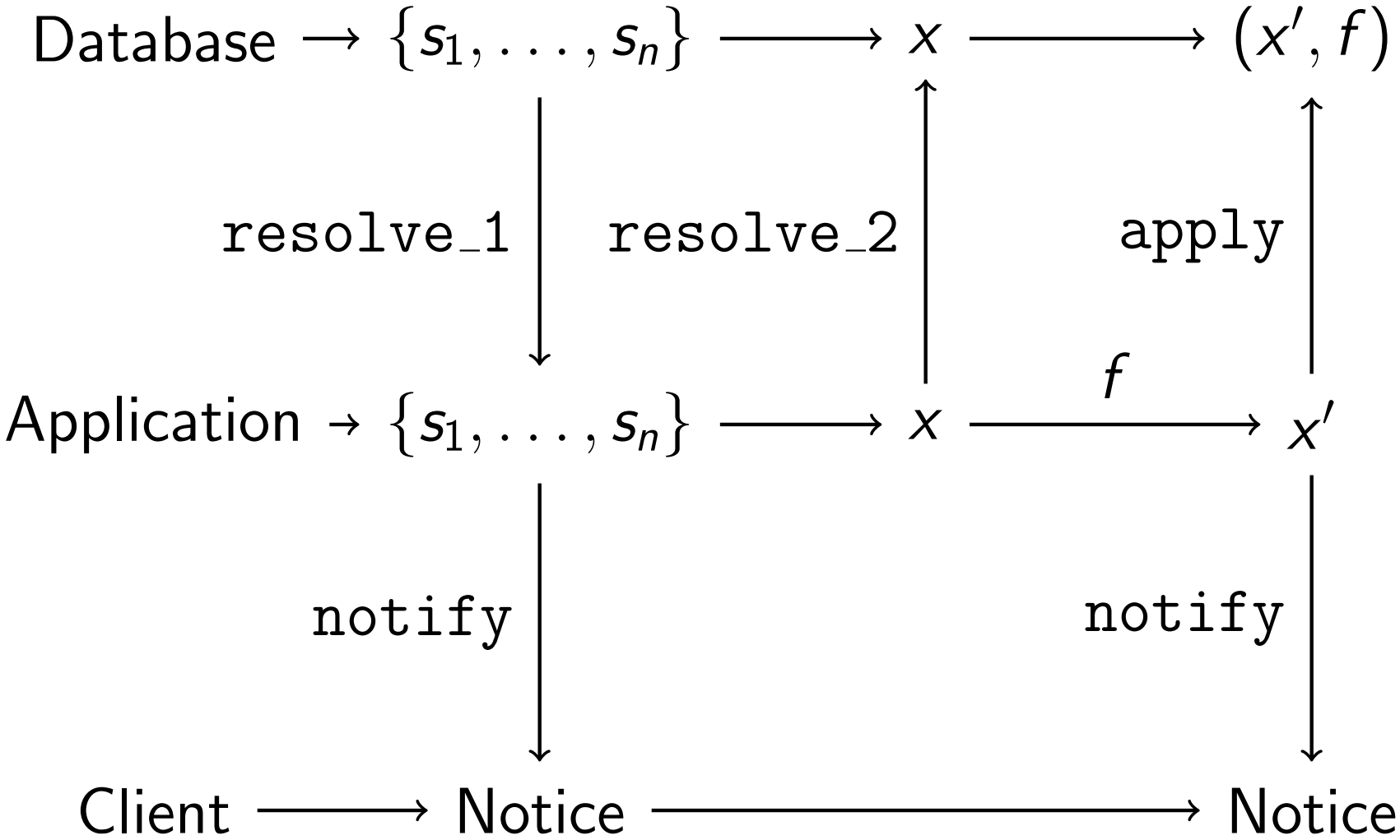
Improving our database interface

- Spawn events when an op gets canceled so client can be notified
- Have our update function take a precondition function as an additional argument instead of embedding it in the code

Improving our database interface

- Spawn events when an op gets canceled so client can be notified
- Have our update function take a precondition function as an additional argument instead of embedding it in the code
- Handling multiple key updates... ignore for now

Interface Flow



Read/Write Quorums

How do we choose what R or W should be?

Read/Write Quorums

How do we choose what R or W should be? There are three different database interactions to consider:

1. `resolve_1`: initial read which may return multiple siblings

Read/Write Quorums

How do we choose what R or W should be? There are three different database interactions to consider:

1. `resolve_1`: initial read which may return multiple siblings
2. `resolve_2`: write for resolving conflicts

Read/Write Quorums

How do we choose what R or W should be? There are three different database interactions to consider:

1. `resolve_1`: initial read which may return multiple siblings
2. `resolve_2`: write for resolving conflicts
3. `apply`: write to record a new operation

`resolve_1` and `apply`

`resolve_1` is application dependent

resolve_1 and apply

resolve_1 is application dependent

Use of a high R value will:

- (-) reduce availability

`resolve_1` and `apply`

`resolve_1` is application dependent

Use of a high R value will:

- (-) reduce availability
- (+) keep sibling count low

`resolve_1` and `apply`

`resolve_1` is application dependent

Use of a high R value will:

- (-) reduce availability
- (+) keep sibling count low
- (+) improve consistency

`resolve_1` and `apply`

`resolve_1` is application dependent

Use of a high R value will:

- (-) reduce availability
- (+) keep sibling count low
- (+) improve consistency

`apply` is the opposite of `resolve_1` and has similar characteristics

`resolve_2`

When attempting to flatten siblings, additional care must be taken.

resolve_2

When attempting to flatten siblings, additional care must be taken.

- What if multiple writers attempt to flatten a list of siblings at the same time?

resolve_2

When attempting to flatten siblings, additional care must be taken.

- What if multiple writers attempt to flatten a list of siblings at the same time?
- What if a netsplit occurs?

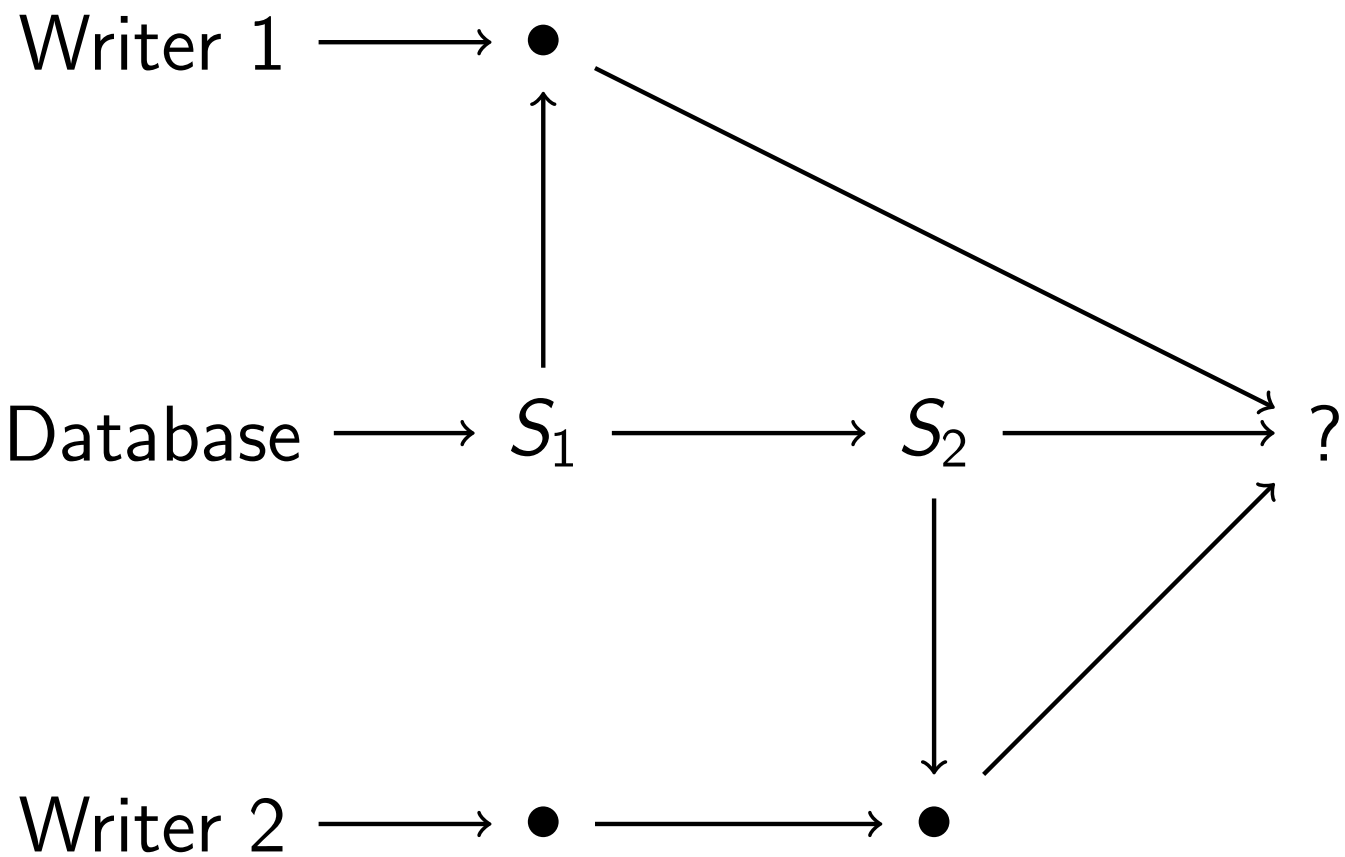
resolve_2

When attempting to flatten siblings, additional care must be taken.

- What if multiple writers attempt to flatten a list of siblings at the same time?
- What if a netsplit occurs?

W should be at least $\lceil Q/2 \rceil$ to prevent conflicts during a netsplit

Handling a `resolve_2` conflict



When writes collide

If multiple writers flatten siblings and save it to the database, there are a few options

When writes collide

If multiple writers flatten siblings and save it to the database, there are a few options

- Arbitrarily ignore all of them but one

When writes collide

If multiple writers flatten siblings and save it to the database, there are a few options

- Arbitrarily ignore all of them but one
- Enforce that flattening can only occur from one writer

When writes collide

If multiple writers flatten siblings and save it to the database, there are a few options

- Arbitrarily ignore all of them but one
- Enforce that flattening can only occur from one writer

The tradeoff, as always, is availability over consistency.

What about when...

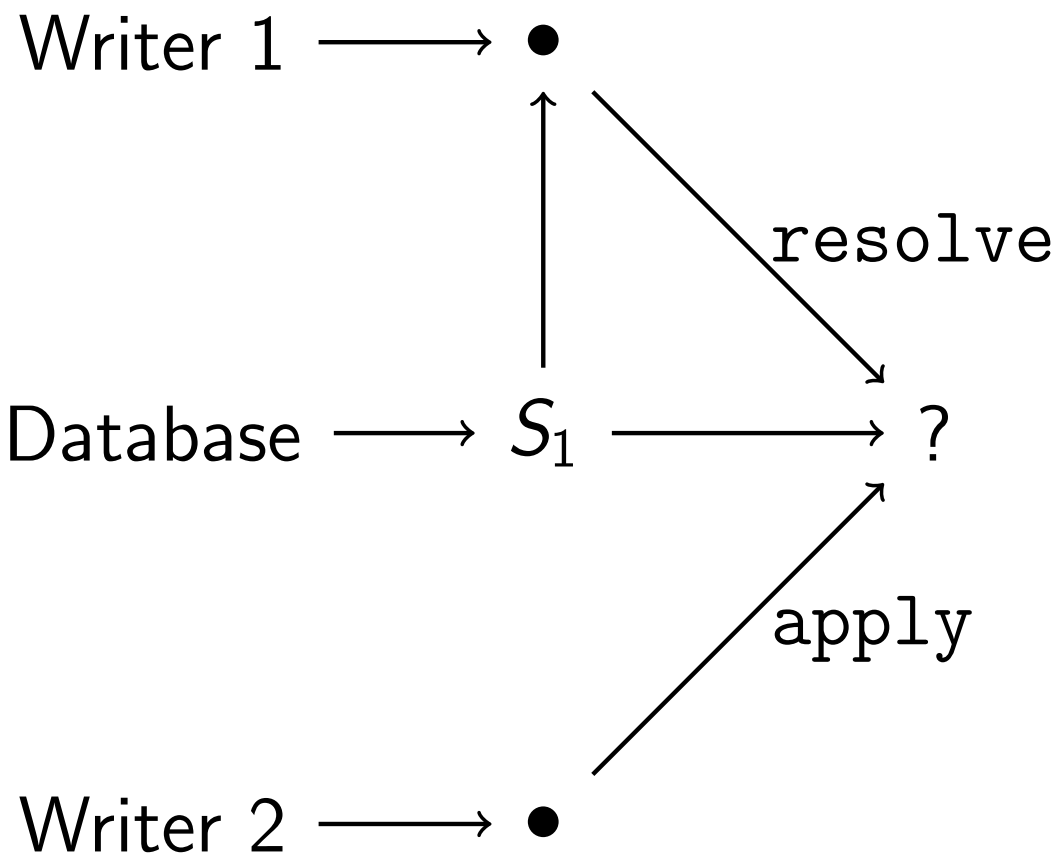
...the result of a flatten operation conflicts with a normal apply?

What about when...

...the result of a flatten operation conflicts with a normal apply?

Assuming all writers are well-behaved and read first with the `resolve` interface, it's safe to ignore the result of the flatten operation.

Another kind of conflict



Handling all conflicts

Many strategies exist for merging siblings.

Important takeaways:

Handling all conflicts

Many strategies exist for merging siblings.

Important takeaways:

- Make sure you can handle all cases, or some sets of siblings will be stuck (your `resolve` function should not have an exit path that doesn't result in a valid write).

Handling all conflicts

Many strategies exist for merging siblings.

Important takeaways:

- Make sure you can handle all cases, or some sets of siblings will be stuck (your `resolve` function should not have an exit path that doesn't result in a valid write).
- The best strategy isn't always the most complicated. With low conflict intensity, the majority of cases will be simple

Writer hierarchy

One option suitable for some applications is a writer hierarchy.

Writer hierarchy

One option suitable for some applications is a writer hierarchy.

Given a list of siblings $\{s_1, \ldots, s_n\}$, first sort them in order by the priority of their sources before composing the embedded functions.

Writer hierarchy

One option suitable for some applications is a writer hierarchy.
Given a list of siblings $\{s_1, \ldots, s_n\}$, first sort them in order by the priority of their sources before composing the embedded functions.
e.g. In an implementation of an email browser client, we may want to prioritize one tab over another.

Note about multi-key transactions

This is hard, but you can...

Note about multi-key transactions

This is hard, but you can...

...Changed my mind, this is very hard. Just denormalize more or use indices

Note about multi-key transactions

This is hard, but you can...

...Changed my mind, this is very hard. Just denormalize more or use indices

If you absolutely must impose a strict transactional relationship between two keys, there is a scheme to ensure that if an op on one key of the transaction fails, the op should fail on the other side (through the usage of inverse operators). Outside the scope.

Possible shifts to existing database usage

- Tighter integration between application code and database code (active app specific conflict resolution)

Possible shifts to existing database usage

- Tighter integration between application code and database code (active app specific conflict resolution)
- Pub/sub api to allow an application to listen for changes to a given key

Possible shifts to existing database usage

- Tighter integration between application code and database code (active app specific conflict resolution)
- Pub/sub api to allow an application to listen for changes to a given key
- Fully featured db client API to allowing setting op, precondition, and other metadata

Possible shifts to existing database usage

- Tighter integration between application code and database code (active app specific conflict resolution)
- Pub/sub api to allow an application to listen for changes to a given key
- Fully featured db client API to allowing setting op, precondition, and other metadata

Conclusion

- If you can reduce conflict intensity in other areas of the system, do it.
- This approach will not provide guaranteed consistency like a canonical CRDT, but is better than LWW.
- Consistency is a spectrum and the developer can lean in either direction. You must choose!