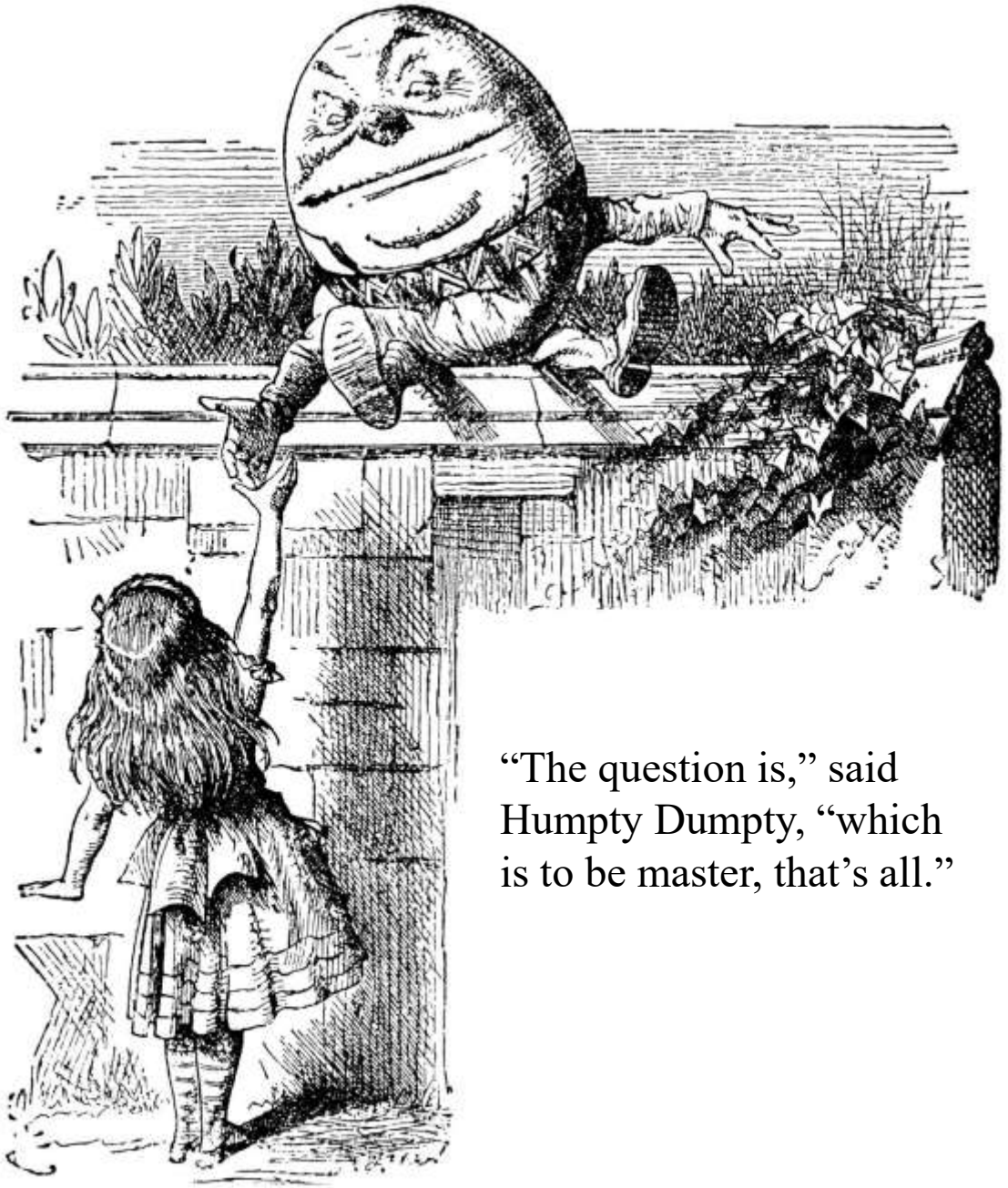


"The question is," said Humpty Dumpty, "which is to be master, that's all."

Leaderless

A Gentle Intro to ~~Multi master~~ Replication

Ewald Cress

SQL Saturday Cambridge

Thank you to our sponsors!

About me

@sqlOnIce

www.sqlonice.com

R&D engineer at bet365

Previously 100% database development

What goes on the C.V.

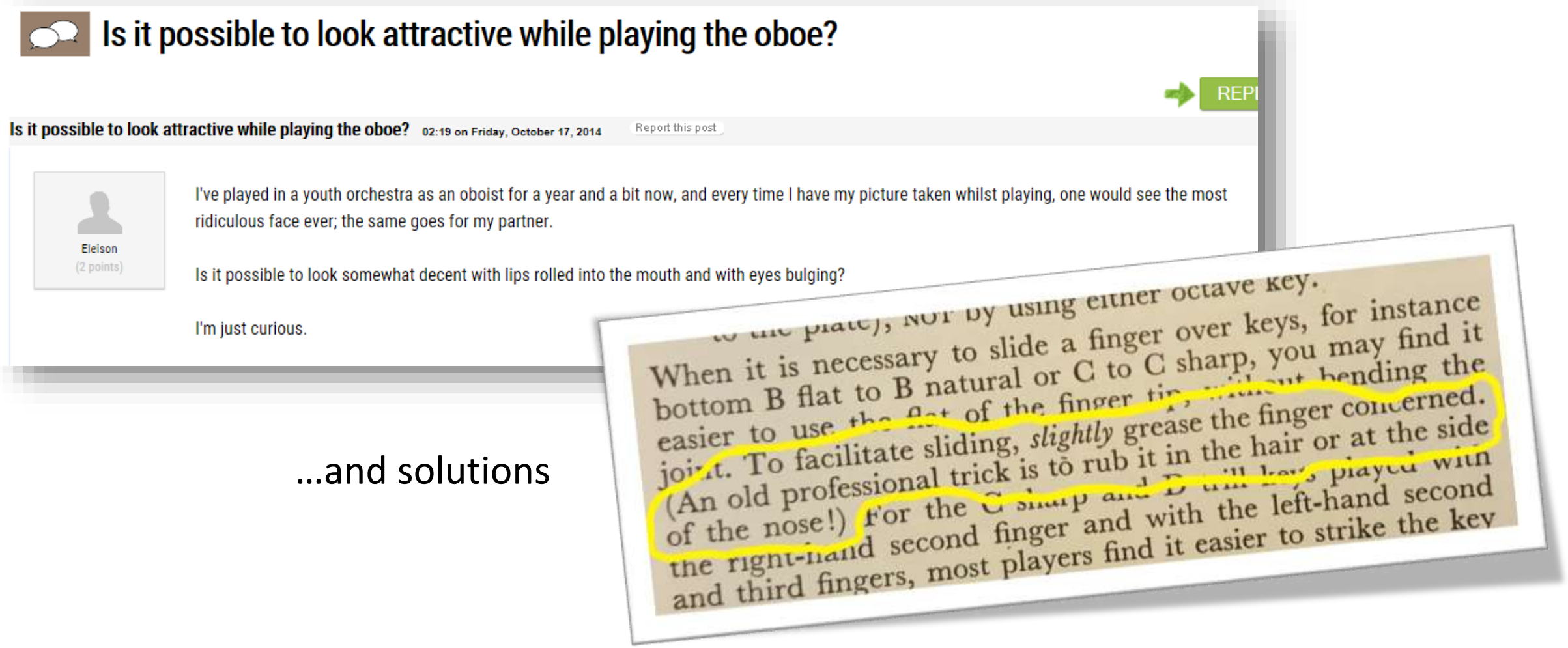
What stays in Vegas

Talk outline

- Why would we keep multiple copies of the same data item?
- How does this relate to the database engine we know and love?
- Conflict among concurrent updates is inevitable, right? RIGHT?
- What do conflict-free replicated data types look like?

Disclaimer: None of this directly applies to SQL Server, Azure SQL Database, although I'm touching on technology running behind the scenes in CosmosDB.
I'm not going to be demonstrating code, languages or frameworks. This is conceptual stuff.

Every domain has its share of exotic problems

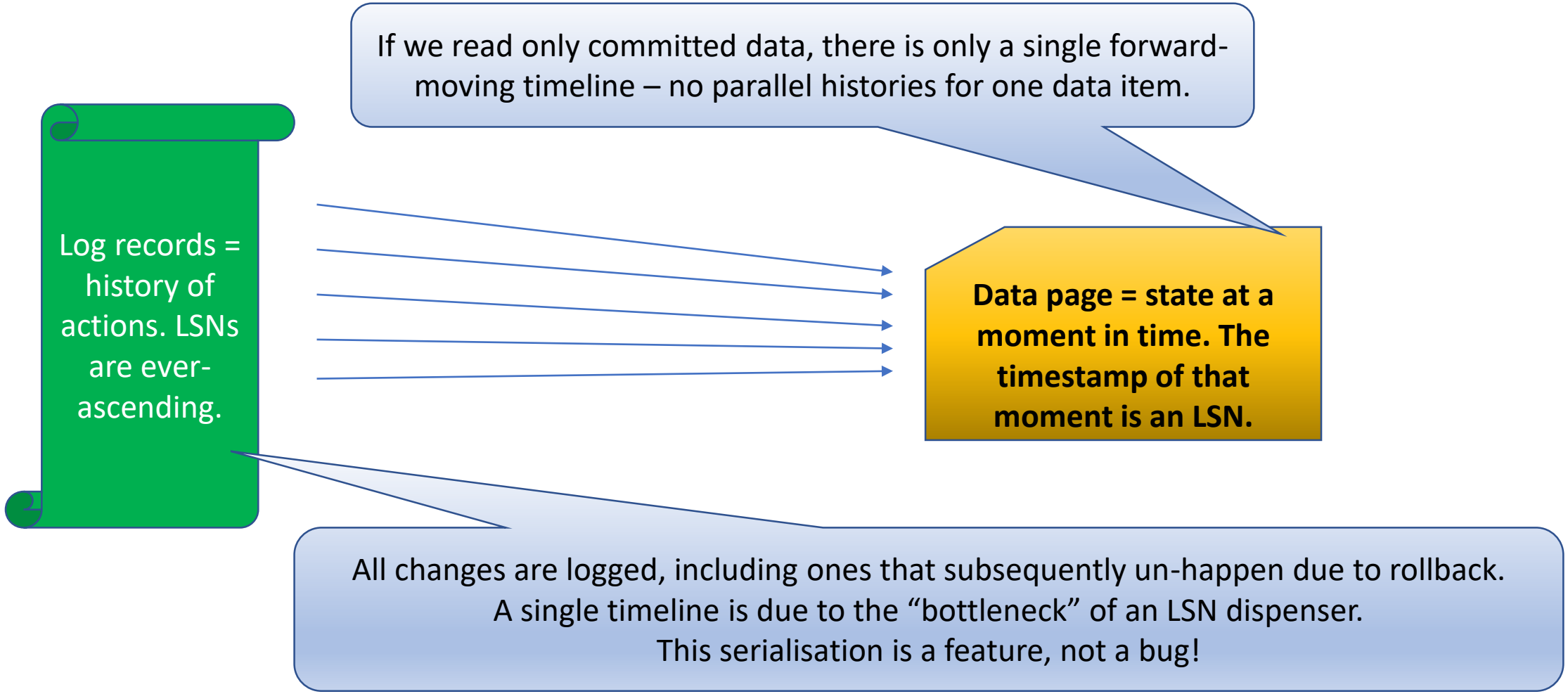


…and solutions

Why replicate data at all?

- Scaling out read-only data to remove a bottleneck
- Mitigate a single point of failure
- Disaster recovery
- Allow rolling upgrades
- Distribute around the globe = beat the speed of light (CosmosDB)
- This is a no-brainer as long as we only have one writable copy

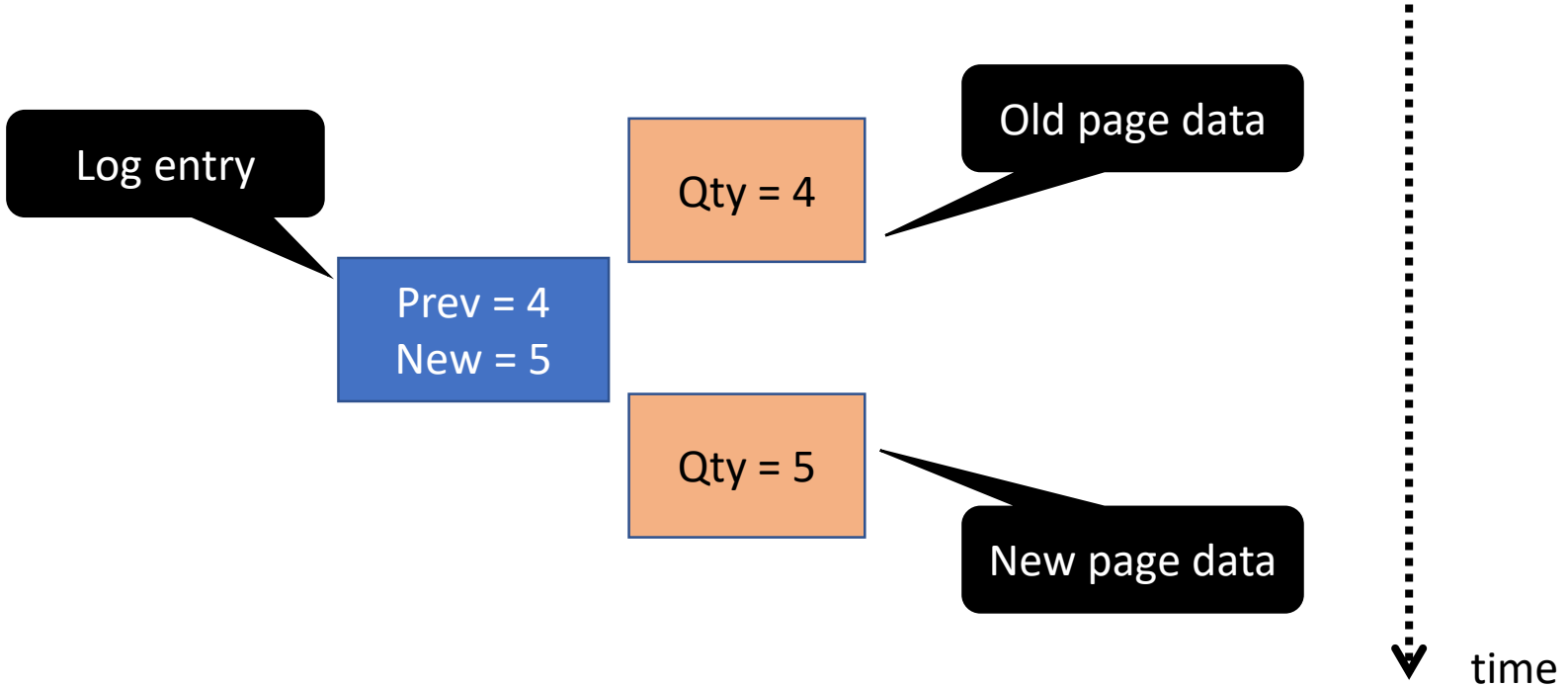
The saga of Paige and Alison



Schrödinger's stock level

```
BEGIN TRAN;

UPDATE Inventory
SET Quantity = 5
WHERE id = 999;
```



Meanwhile, a different transaction/client can choose to:

- see the old value (snapshot isolation), knowing it may be stale
- see the new value (read uncommitted), knowing it might get rolled back
- wait its turn for exclusive access

The CRDT challenge:

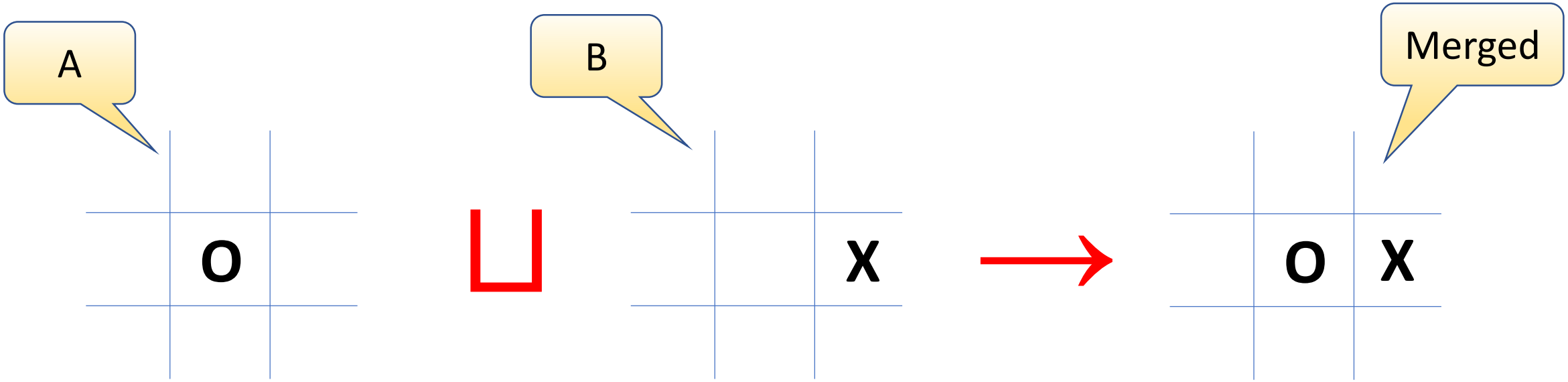
What if EVERYBODY could be right?

(even when they're wrong)

Conflict-free replicated data types cheat sheet

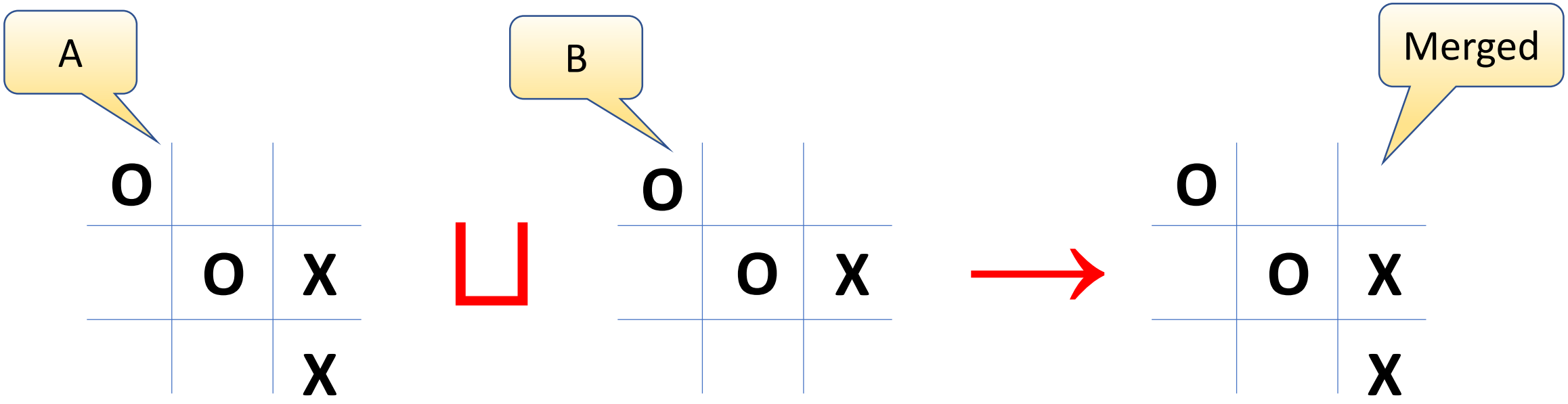
- Just a type/class (think integer, dictionary, shoppingBasket…)
- Internal representation up to the implementor
- Real CRDTs show common patterns, e.g. using logical clocks
- In addition to normal operations a consumer cares about, it must have a Merge method that is:
  - Commutative
  - Associative
  - Idempotent
- This Merge function must allow **ANY** two instances to be merged

A tic-tac-toe CRDT? Maybe…



A and B don’t step on each other’s toes, so their merge is a simple union.
They don’t disagree as much as contain non-overlapping fragments of the same picture.

Some more tic-tac-toe merge success



A > B, so B adds nothing to the merged result.
Put differently, B is an earlier point on the same timeline as A.

It's a perjury trap!

o		
	o	x
		x

⊔

x		
	o	x
		o

→

This is as bad as division by zero: we simply can't express such a merge.
These two aren't from different points in time, but from parallel universes;
alternative truths, as it were.

A jigsaw CRDT

The truth, the whole truth, and nothing but the truth

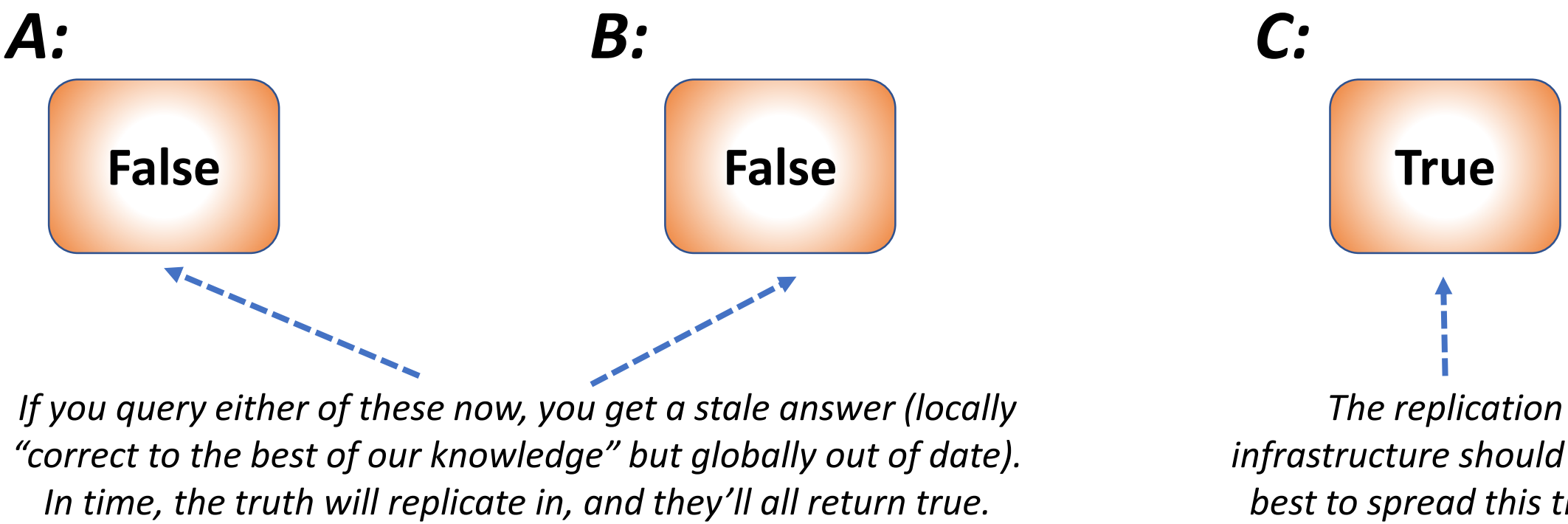
A tapestry of half-truths

Intuitively, merging is a form of set union here. Each puzzle piece is a set member with a deterministic target position, and pieces showing up in both replicas get deduplicated.

DEMO TIME:
Boolean wrestling

The enable-wins flag

A restricted form of Boolean; starts off as false, and once **any** replica has set it to true, it can't transition to false again.



The enable-wins flag: methods

Flag.Set()
Sets the value to True

Set() and Value() are exposed to the "end user" programming interface

Flag.Value()
Returns value like a normal Boolean, with False as default if not explicitly set

Flag.Merge(otherFlag)
Logical OR of the two values

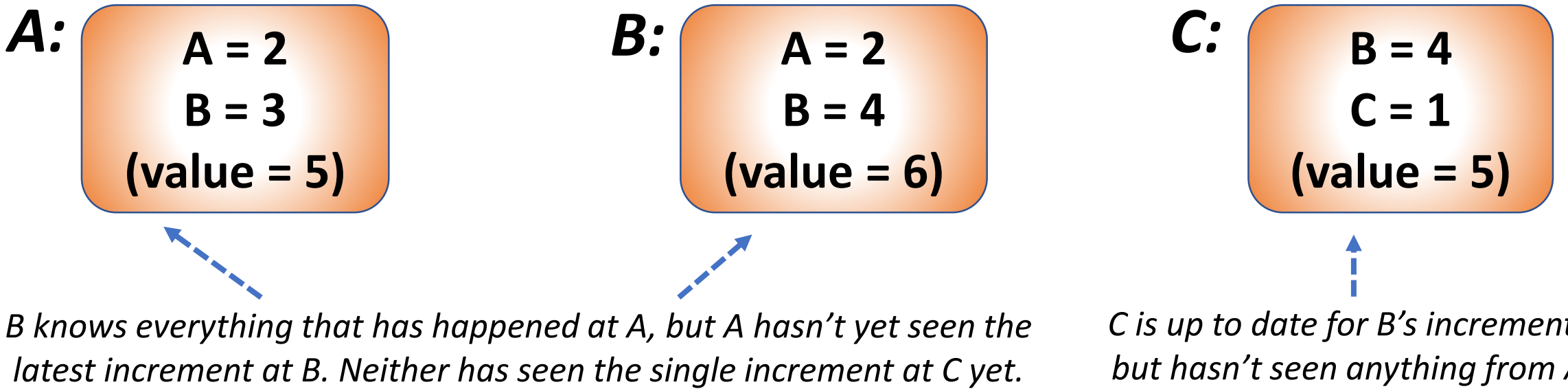
Merge() is only for internal use by replication gubbins

Merge() is purely pairwise. While the replication system may want to keep track of which replicas need merging into which, the data type has no need to deal with more than two instances at a time.

DEMO TIME:
Out for the count

The increment-only counter

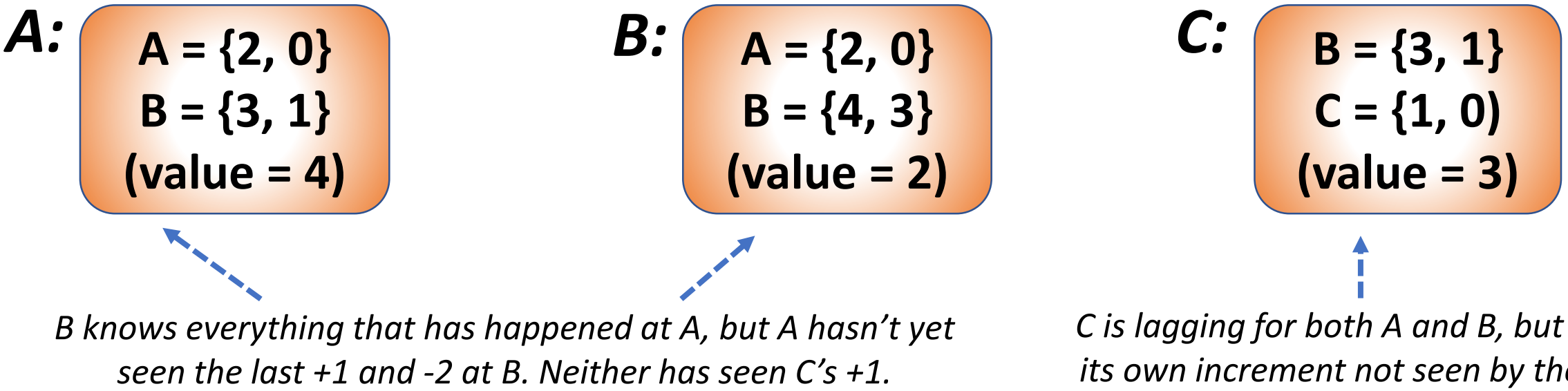
Increments are separately tracked per replica. No entry needs to be made for a replica until it services an increment.



Converged value 7 isn't in a single replica, but the system as a whole knows it.

The PN-counter

Exactly like the increment-only P-counter, but each actor gets separate count values for increments (P) and decrements (N).



Here B is ahead of A, but neither B nor C is ahead of each other – each holds a piece of truth that the other doesn't know.

DEMO TIME:
A bag of stuff

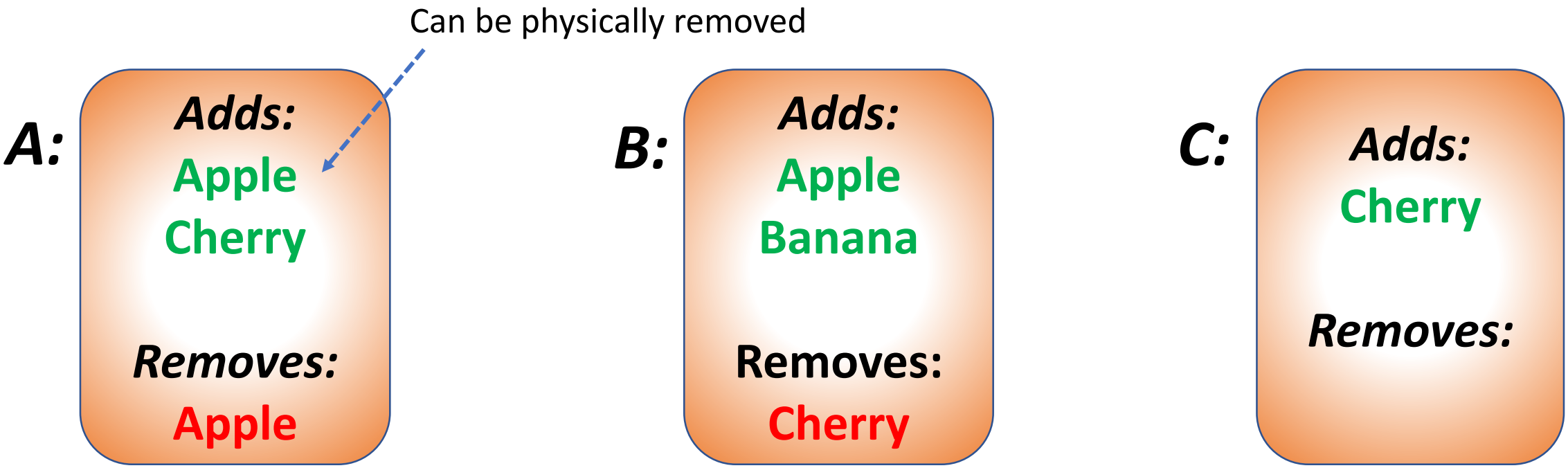
The grow-only set

Any replica can add any number of distinct things to the set, including in duplicate – being a set, deduplication is a natural operation

Pairwise merging through set union is trivial, but we can’t remove items.

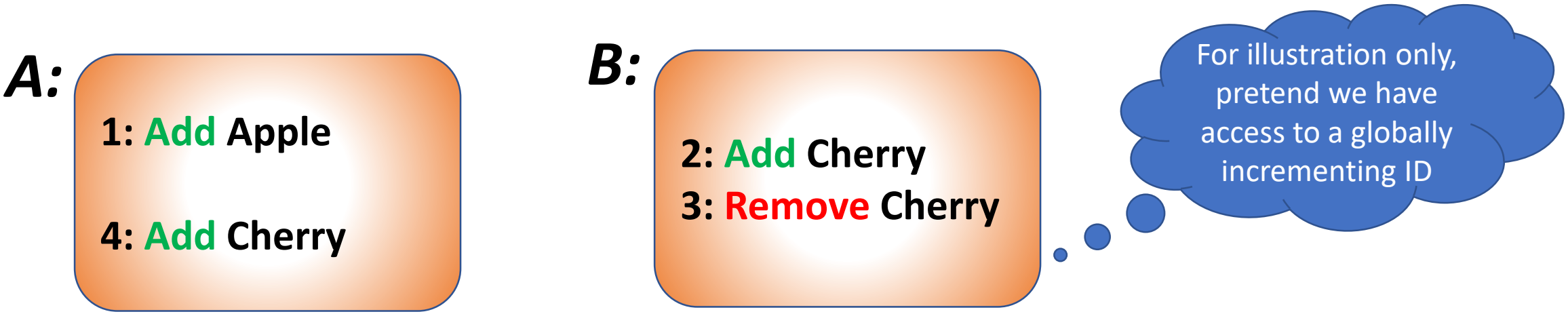
The 2-phase set

We can remove an item by tracking its removal as negative addition (antimatter!)



Converged set = {Banana}

Pomp and Metadata – more versatile sets



When merging these two, global ordering makes it obvious that **add** at 4 trumps the **remove** at 3. Also, since B knows that 3 trumps 2, we can compact things by deleting 2.

We can now re-**add** prior removes, unlike with the 2-phase set. But this implementation is pointless because it requires a central "Now dispenser" (ID generator/clock), aka serialisation point.

Must-read: Justin Sheehy, "There is No Now": https://queue.acm.org/detail.cfm?id=2745385

Now replace total order with partial order

A:

A1: Add Apple
A2: Add Cherry

B:

B1: Add Cherry
Remove Cherry B1
Remove Apple A1

No blank cheque removals

Per-replica history

Merging into…

A2: Cherry
A1,B1 don't exist

Compact form:

***Alison*!**

Clock = {A2, B1}
A2: Cherry

No future merge can reinstate A1 or B1 – clock knows they're gone

Result: the Observed-Remove Set

- A.K.A. the OR-set and AWORset (add-wins observed-remove)
- The optimised form stores removals implicitly in the combination of clock and element list
- In Riak the implementation is known as the ORSWOT: observed-remove set without tombstones
- Michael Owen “Using Erlang, Riak, and the ORSWOT CRDT at bet365” https://www.youtube.com/watch?v=WXmO1IvzIZY

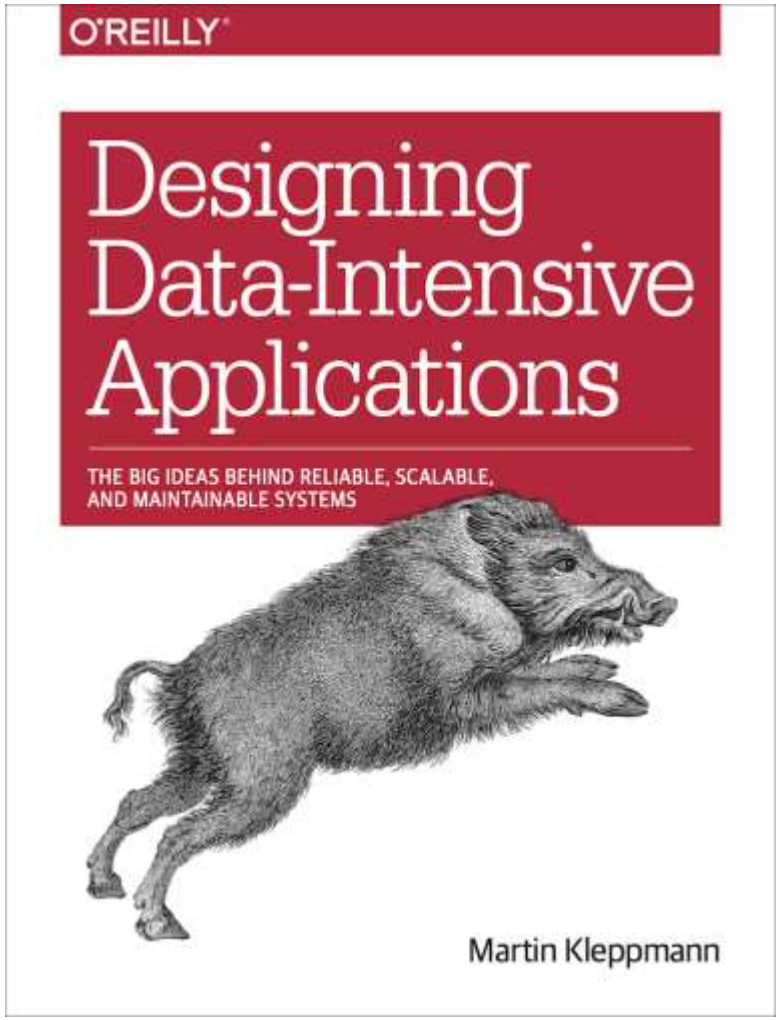
Where are we up to?

- Flags with one-way state transitions are trivial
- Increment-only counters must be internally partitioned by actor
- A PN-counter is a pair of increment-only counters
- Grow-only and 2-phase sets don’t require actor information
- Observed-remove sets need vector clocks, and each element gets a “surrogate key” in the form of an {actor, count} dot
- Sensible removal requires the client to state the baseline version they want to remove from

Wrapping up

- This coverage of sets papers over some subtle (but solved) problems
- More complicated CRDTs use dots and clocks as building blocks
- In fact, a vector clock is itself a CRDT!
- Maps, graphs, etc can be designed like this
- Sometimes it’s possible to abstract things away so they “just work”
- However, it always helps to understand your raw materials

If you want to dive deeper, two good pointers



Code Mesh London, 7-9 Nov 2018

Half-day tutorial by Annette Bieniusa and Nuno Preguiça:
"Just the right kind of consistency!"

Main conference includes a talk by Carlos Baquero, co-creator of CRDTs: *"CRDTs: From sequential to concurrent executions".*

This is a highly recommended conference for anybody wanting to look at non-mainstream technologies.

Two useful short reads

- A Bluffers Guide to CRDTs in Riak (Russell Brown): https://gist.github.com/russelldb/f92f44bdfb619e089a4d - this gives a good feel of using a specific database implementation, explained by one of its key implementors.
- CRDT Tutorial For Beginners (Lawrence Wagerfield): https://github.com/ljwagerfield/crdt

Both of these will serve as entry points into the literature.