CRDT Sets: Theory & Practice

Russell Brown
Basho Technologies

What?

- Why we need CRDTs
- What's a CRDT, anyway?
- To a general CRDT set

What?

- Riak Set Data Type
- Delta-Sets
- “Big”sets

This project is funded by the European Union,
7th Research Framework Programme, ICT call 10,
grant agreement n°609551.

Why CRDTs?

Scale Up

$$$Big Iron
(still fails)

Scale Out

Commodity Servers
Distributed Systems
Multi Datacenter

DISTRIBUTED DATABASE

Trade Off

CAP

http://aphyr.com/posts/288-the-network-is-reliable

C A

C A

C

A

CP

AP

Consistency

There must exist **a total order on all operations** such that each operation looks **as if it were completed at a single instant.** This is equivalent to requiring requests of the distributed shared memory to **act as if they were executing on a single node**, responding to operations one at a time.

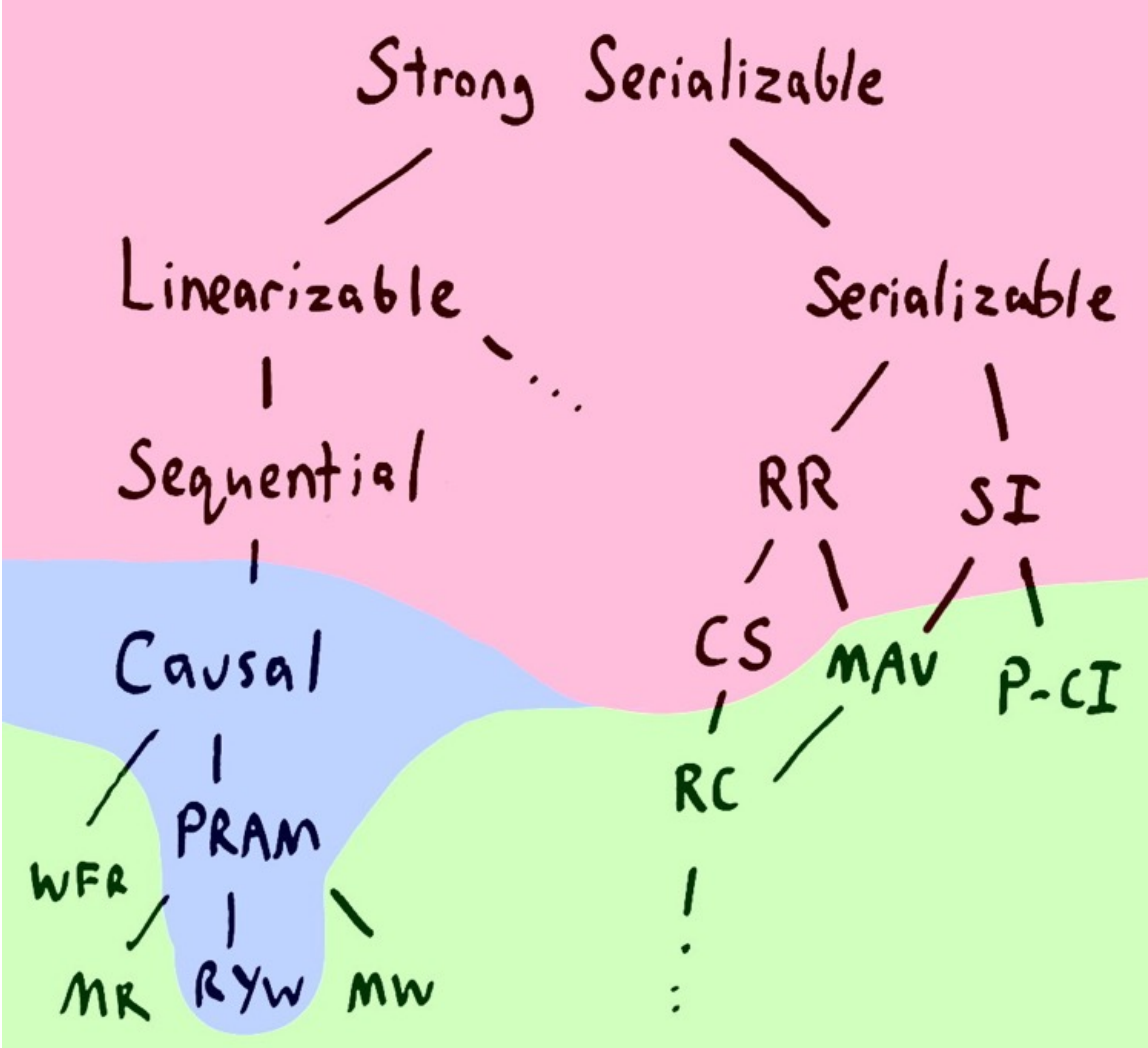
--Gilbert & Lynch

Consistency

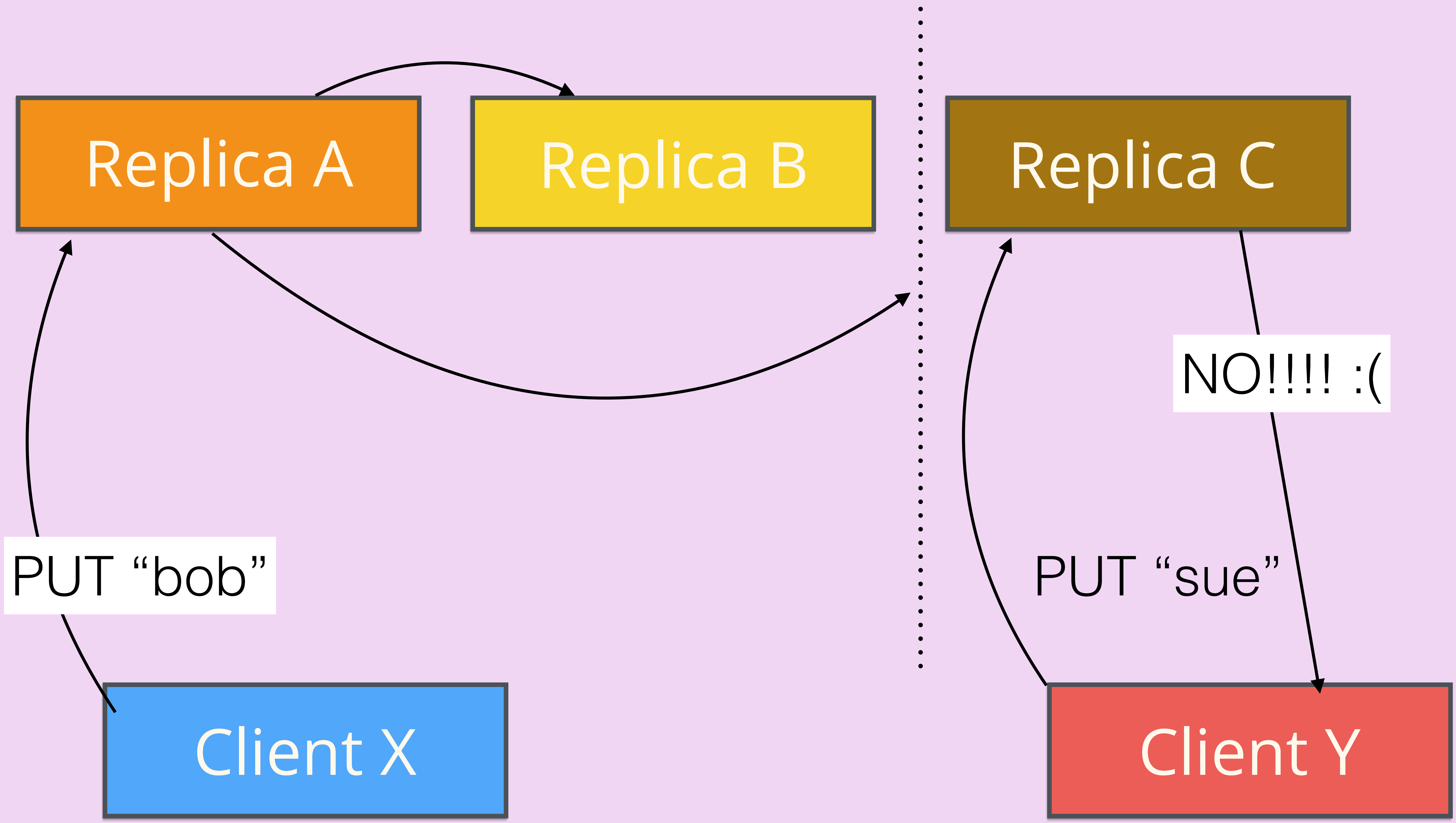
One important property of an atomic read/write shared memory is that **any read operation that begins after a write operation completes must return that value, or the result of a later write operation**. This is the consistency guarantee that generally provides **the easiest model for users to understand**, and is most convenient for those attempting to design a client application that uses the distributed service

riak

--Gilbert & Lynch



https://aphyr.com/posts/313-strong-consistency-models


Replica A
Replica B
Replica C
NO!!!! :(
PUT “bob”
PUT “sue”
Client X
Client Y

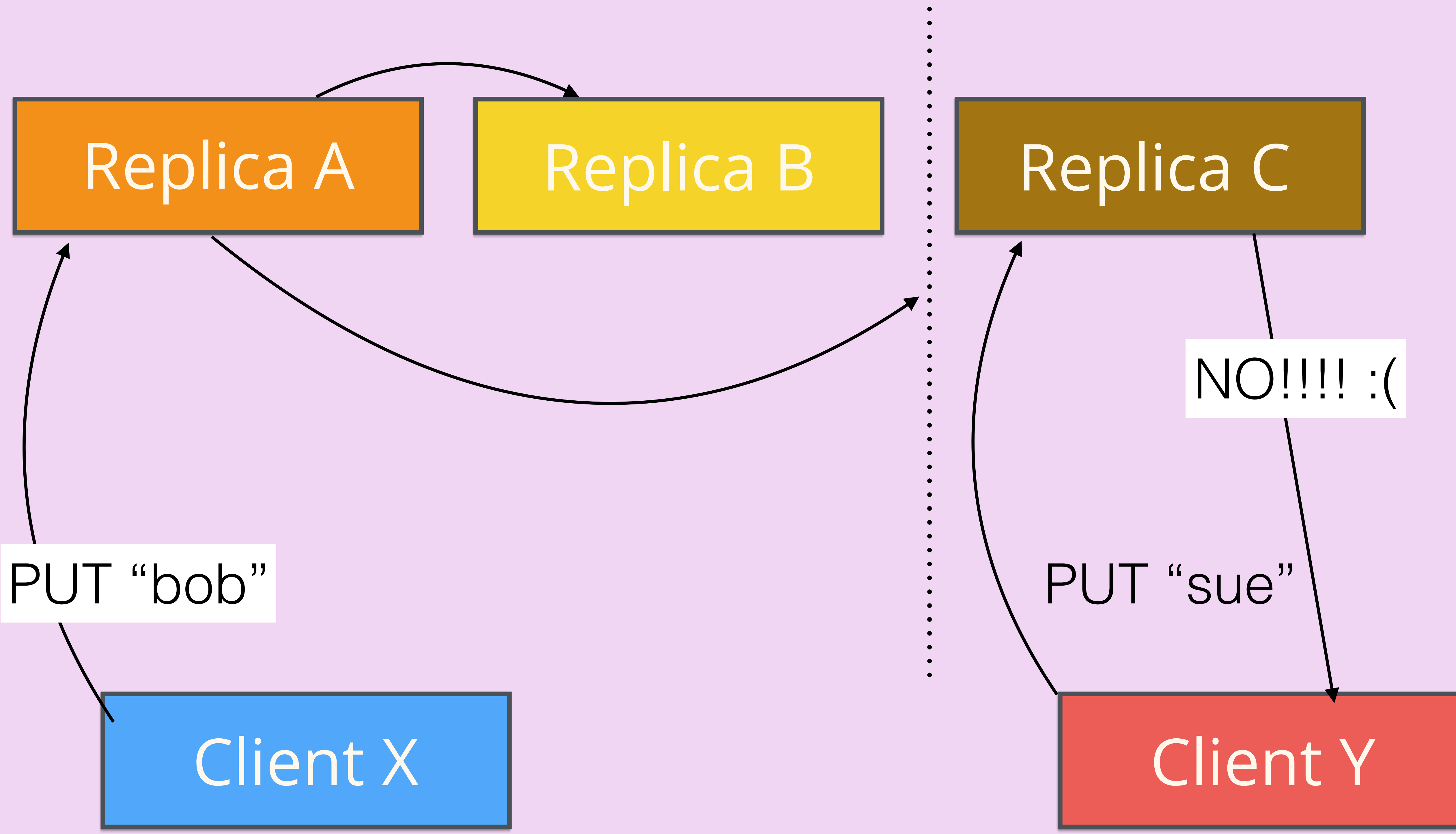
Consistent

Availability

Any non-failing node can respond to any request

--Gilbert & Lynch

riak

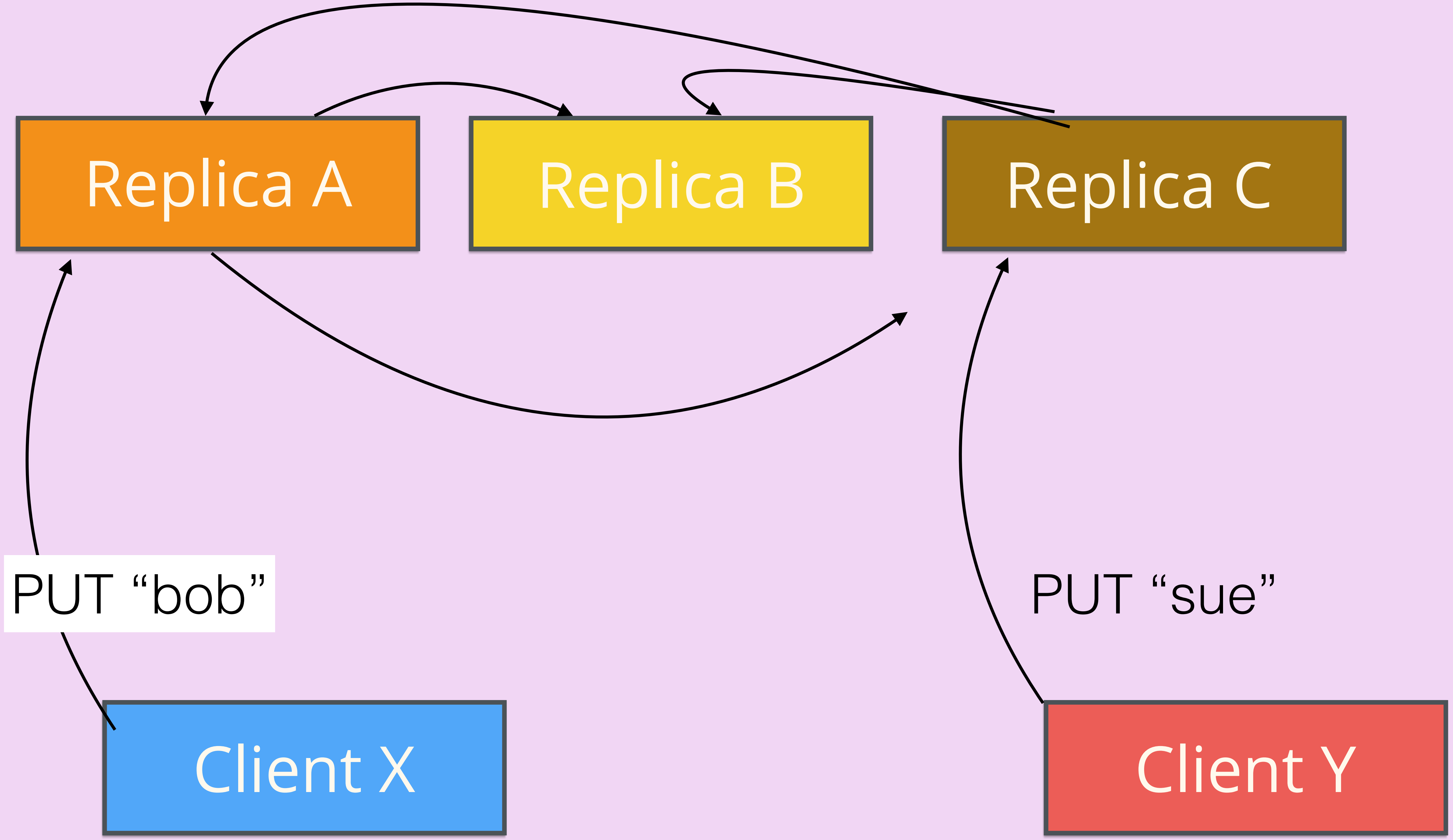

Replica A
Replica B
Replica C
NO!!!! :(
PUT “bob”
PUT “sue”
Client X
Client Y

Consistent

Consensus for a total order of events

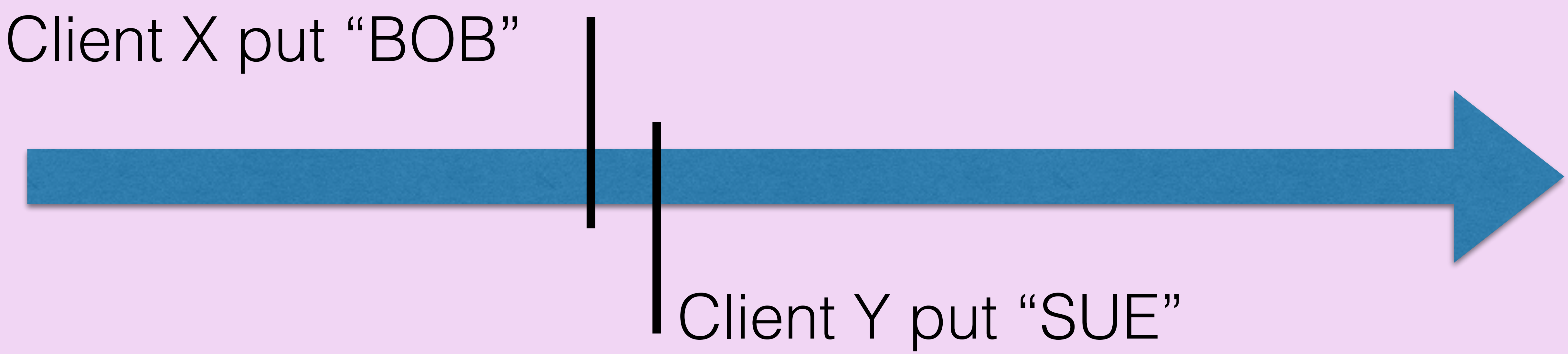
Requires a quorum

Coordination waits


Replica A
Replica B
Replica C
PUT “bob”
PUT “sue”
Client X
Client Y

Consistent

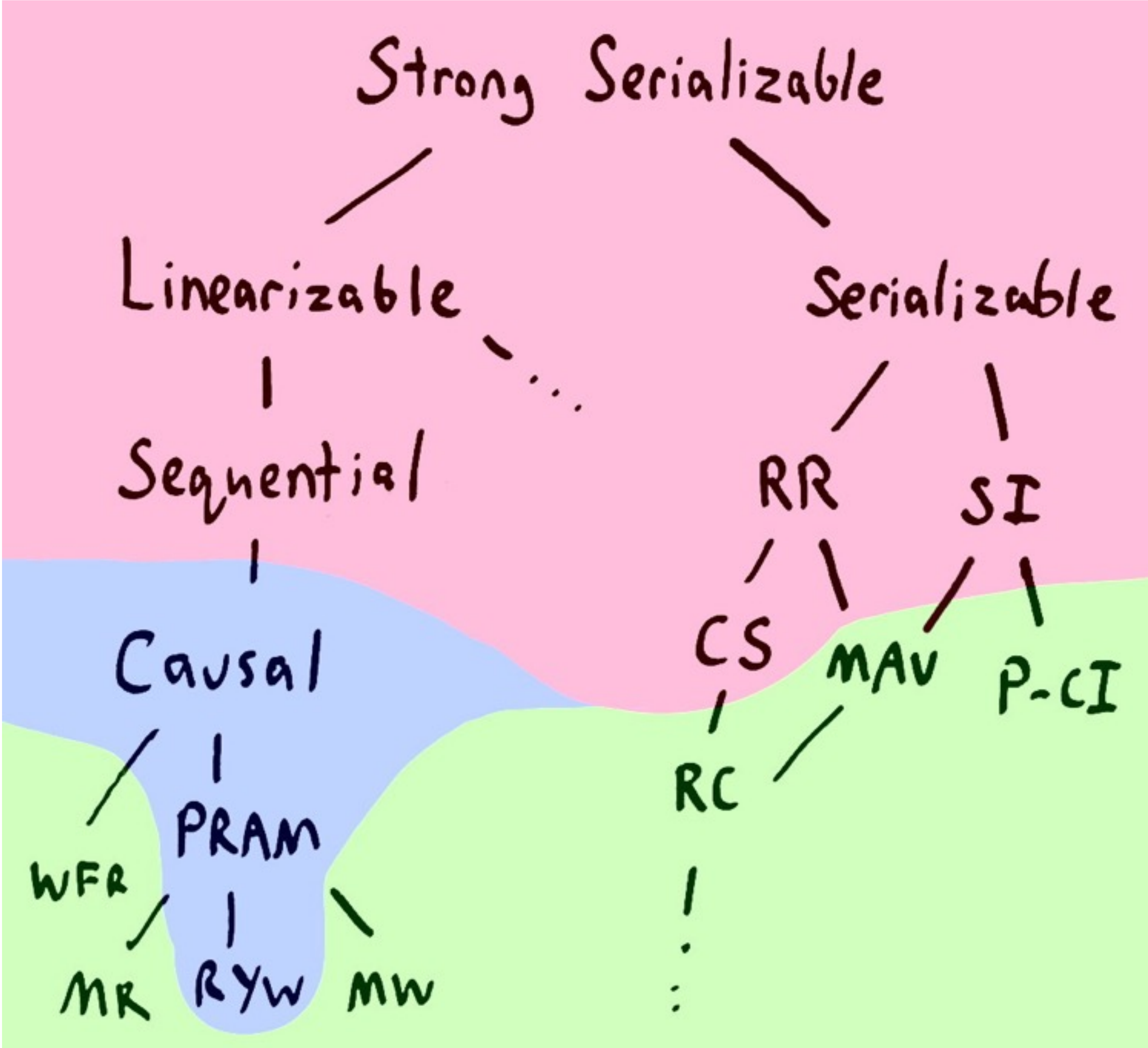
Events put in a TOTAL ORDER



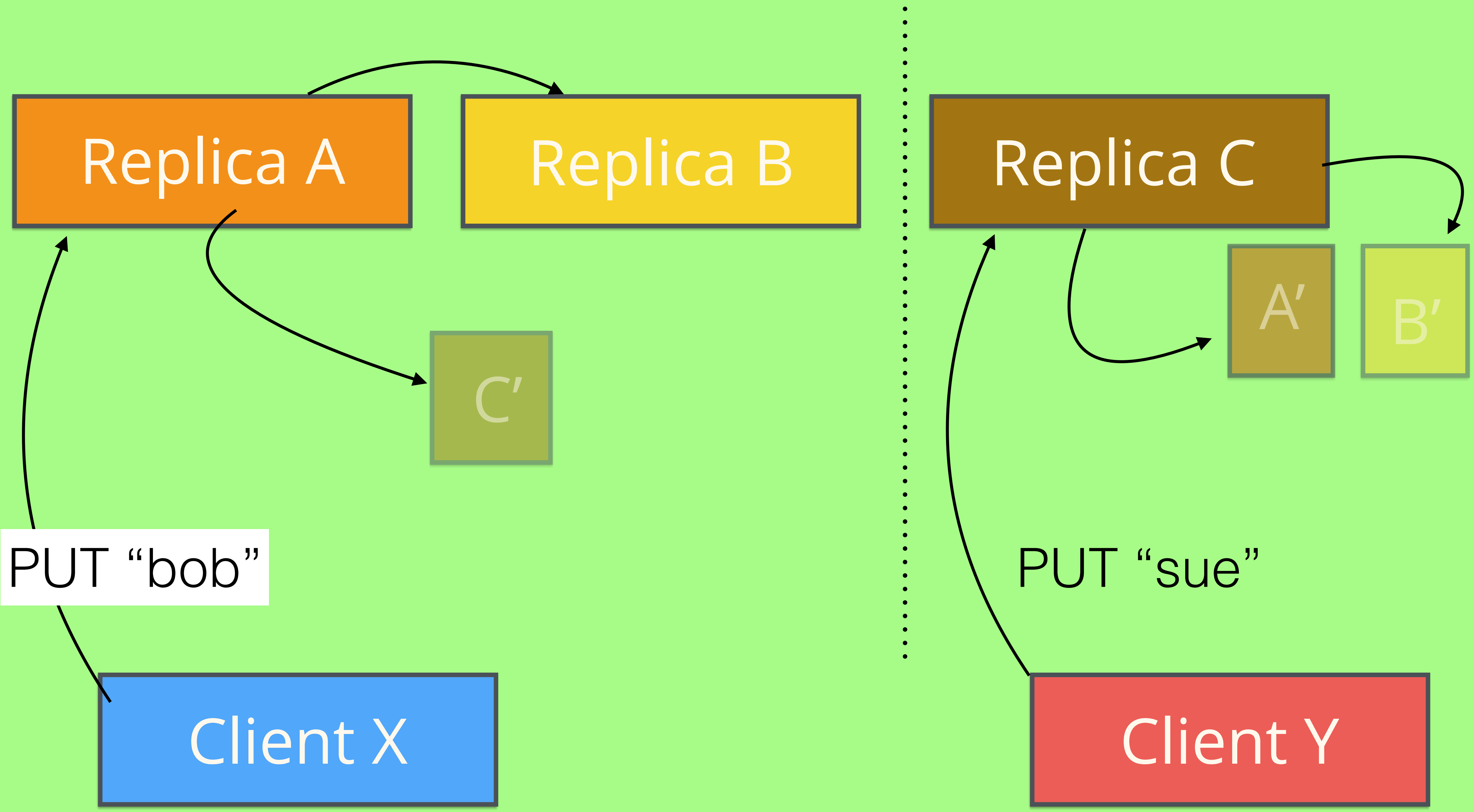
Eventual Consistency

Eventual consistency is a consistency model used in distributed computing that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

--Wikipedia



https://aphyr.com/posts/313-strong-consistency-models

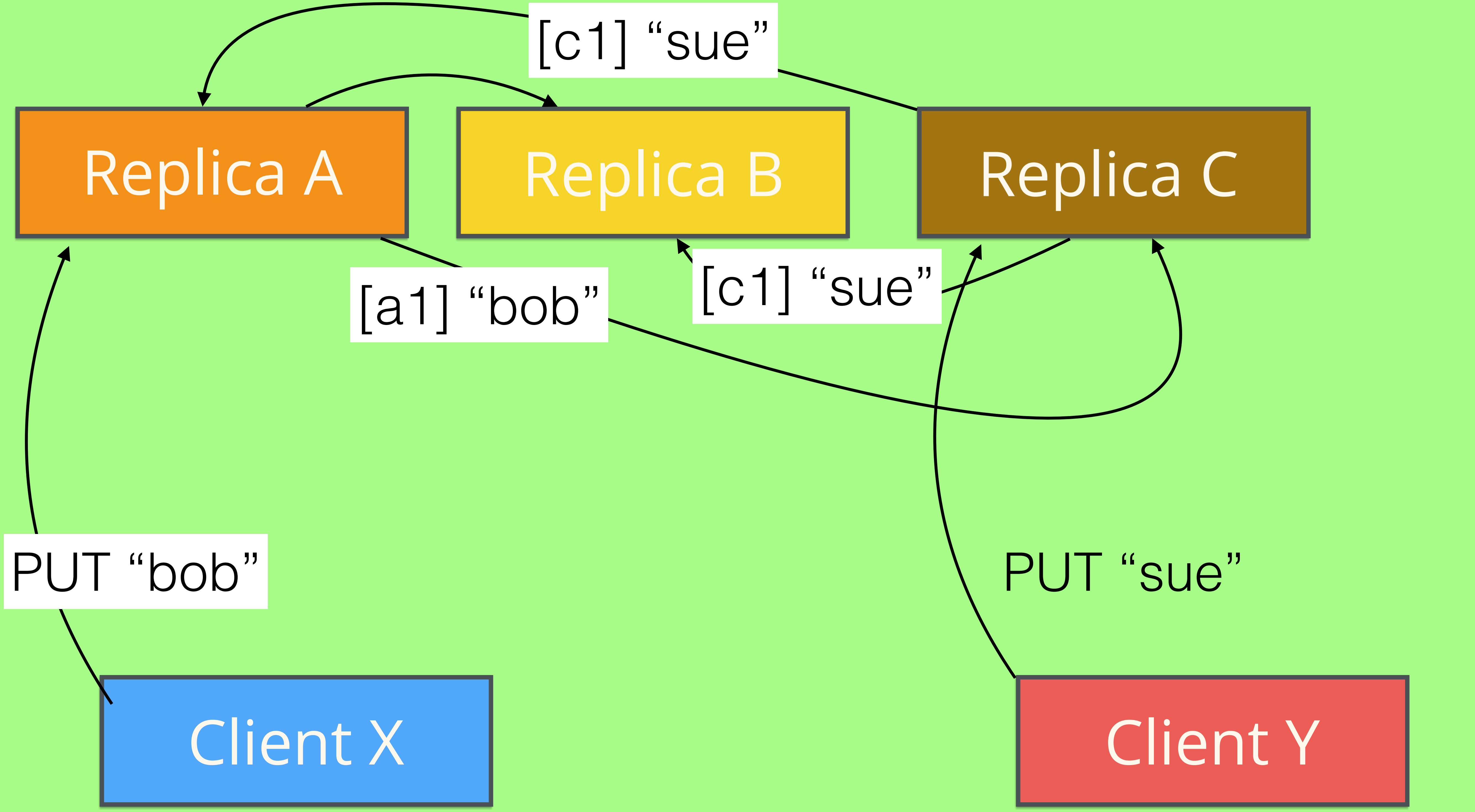

Replica A
Replica B
Replica C
C′
A′
B′
PUT "bob"
PUT "sue"
Client X
Client Y

Available

Optimistic replication (and logical clocks)

Reconcile concurrency on read

No coordination for lower latency


[c1] "sue"
Replica A
Replica B
Replica C
[a1] "bob"
[c1] "sue"
PUT "bob"
PUT "sue"
Client X
Client Y

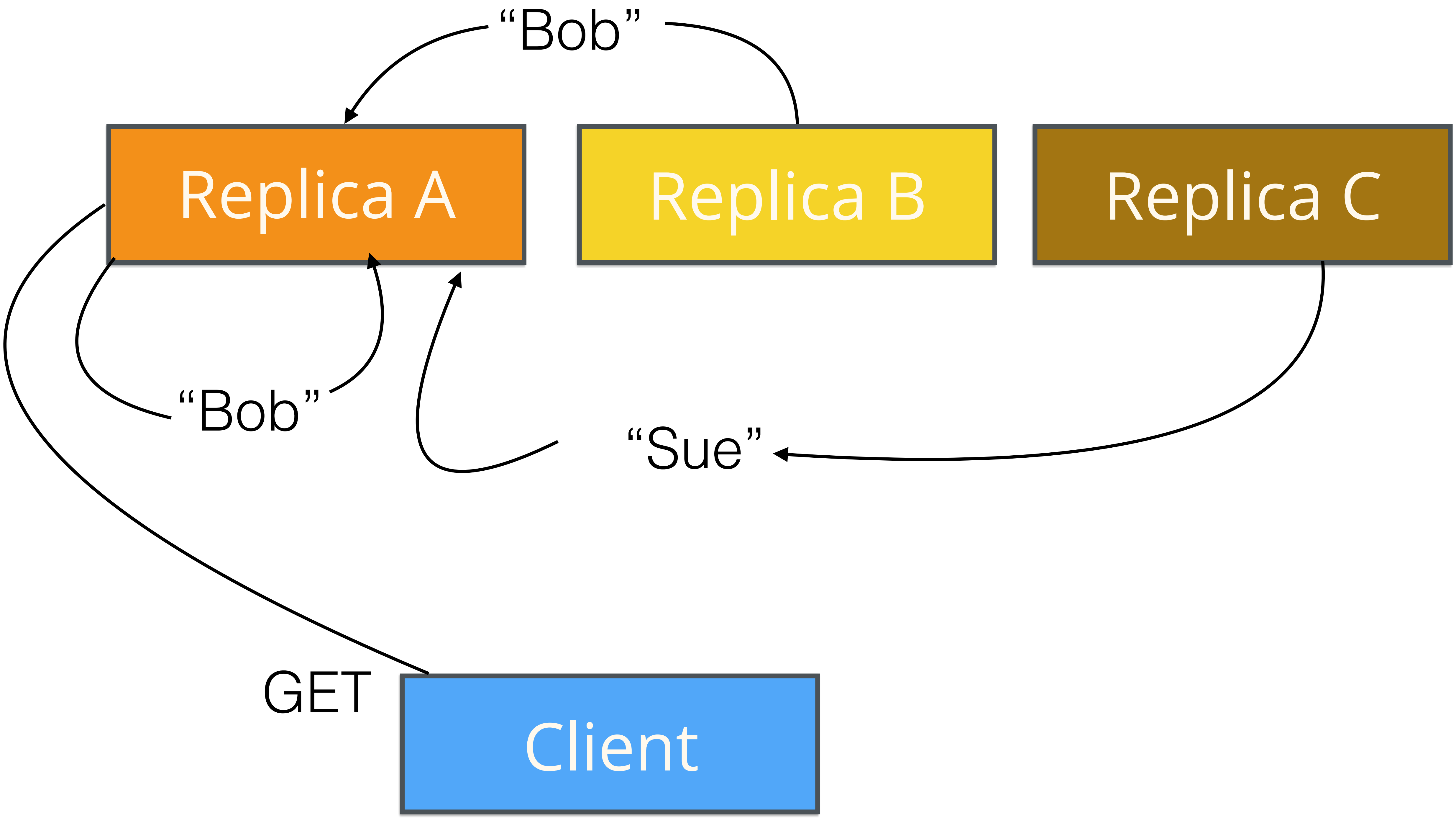
Low Latency

Problem?

Consistency

This is the consistency guarantee that generally provides the easiest model for users to understand, and is most convenient for those attempting to design a client application that uses the distributed service

--Gilbert & Lynch



Conflict!

Eventual Consistency

Eventual consistency is a consistency model used in distributed computing that informally guarantees that, if no new updates are made to a given data item, **eventually all accesses to that item will return the last updated value.**

--Wikipedia

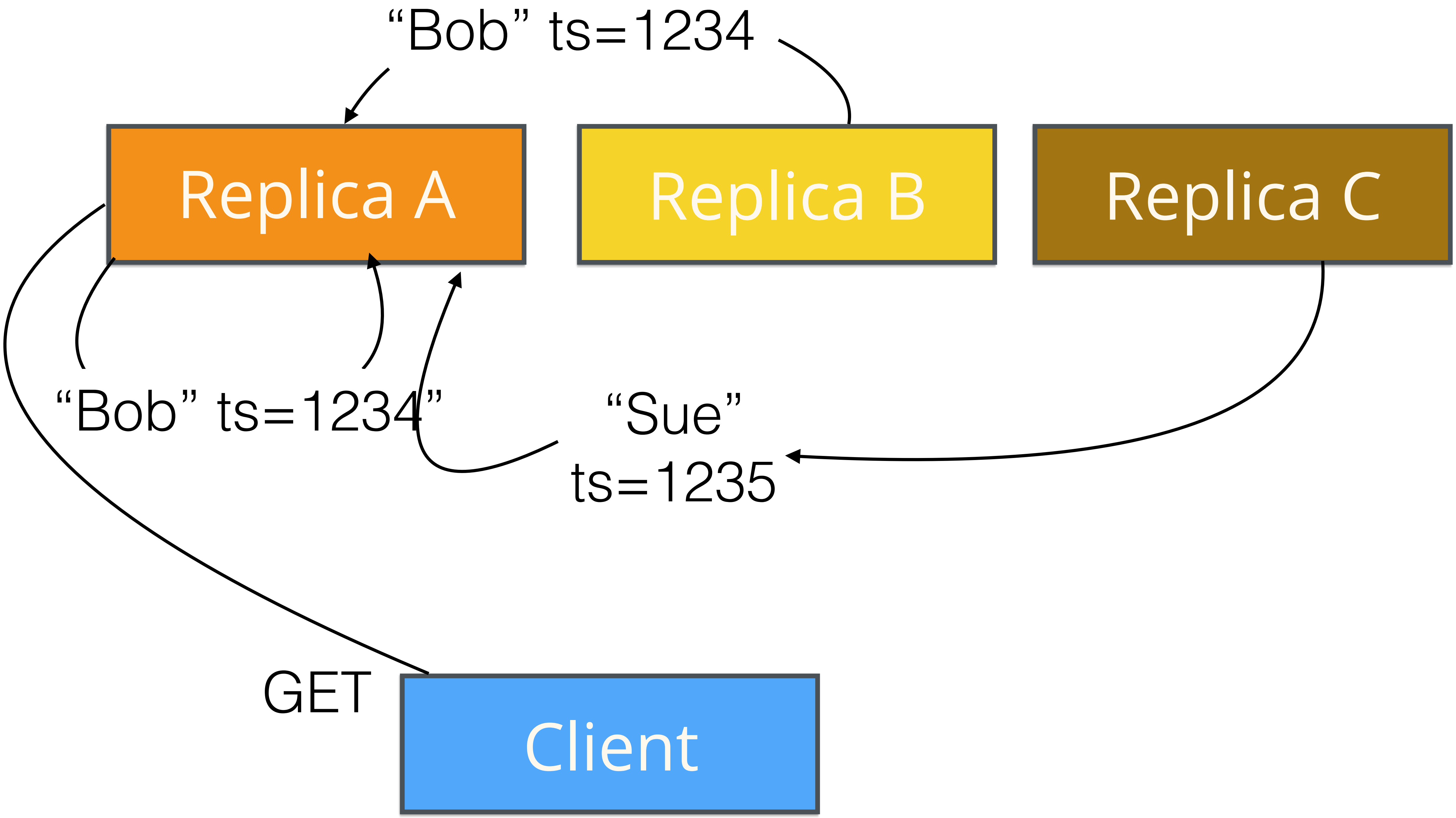
riak

Last Updated Value?

Convergence

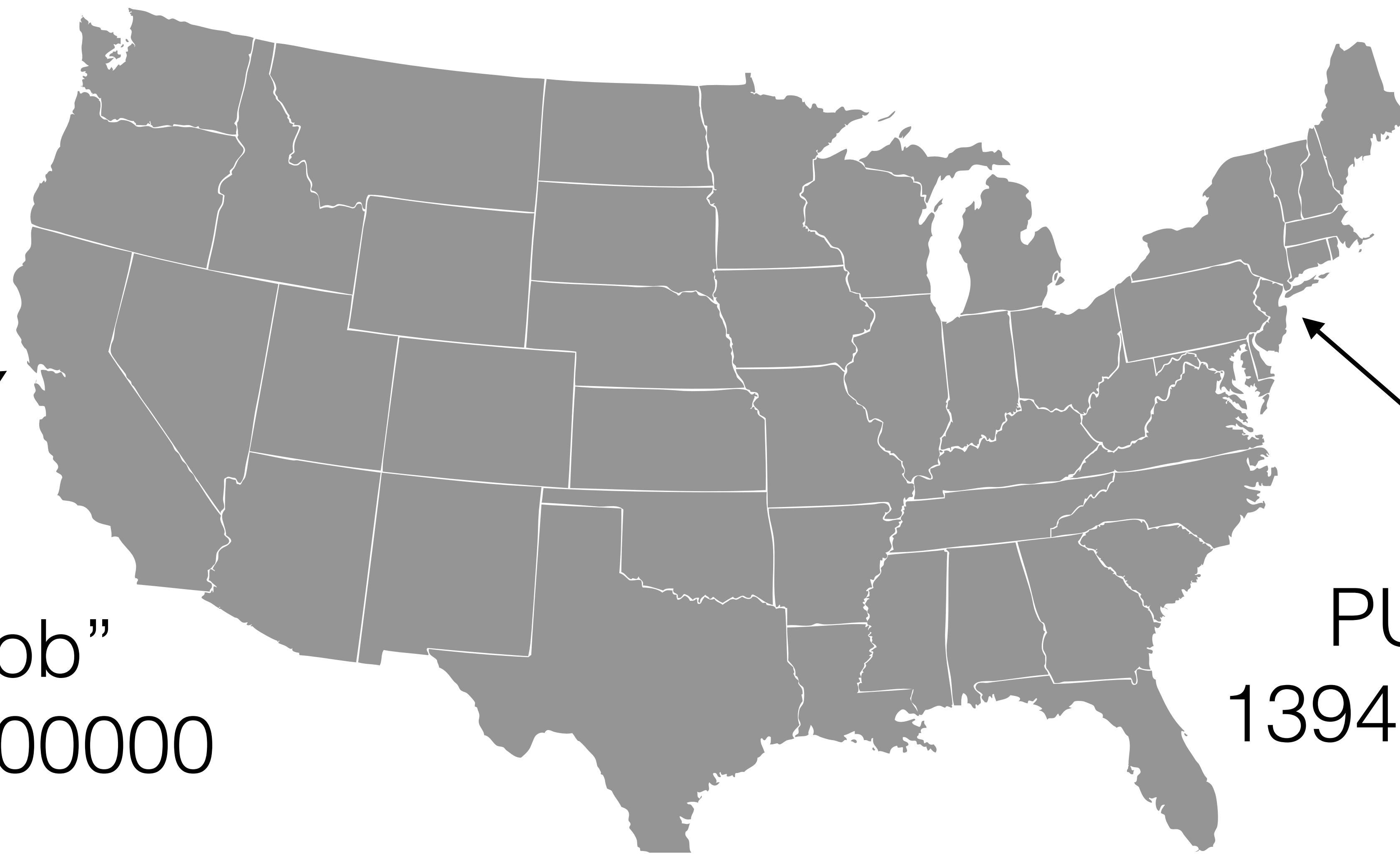
Availability is great - what’s my data?

It depends



Last Write Wins!

Physics Problem

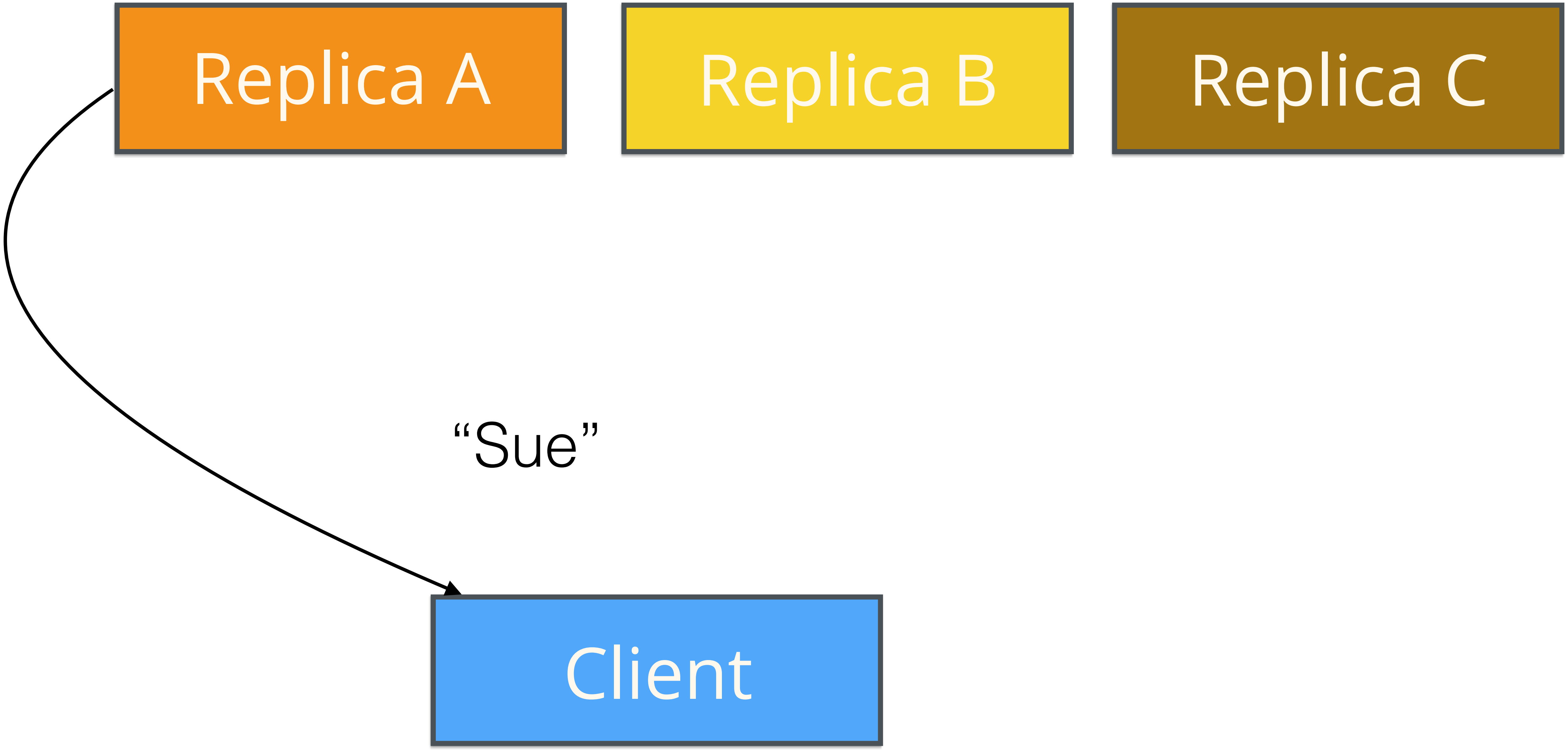


PUT “bob”
1394382600000

PUT “sue”
1394382600020

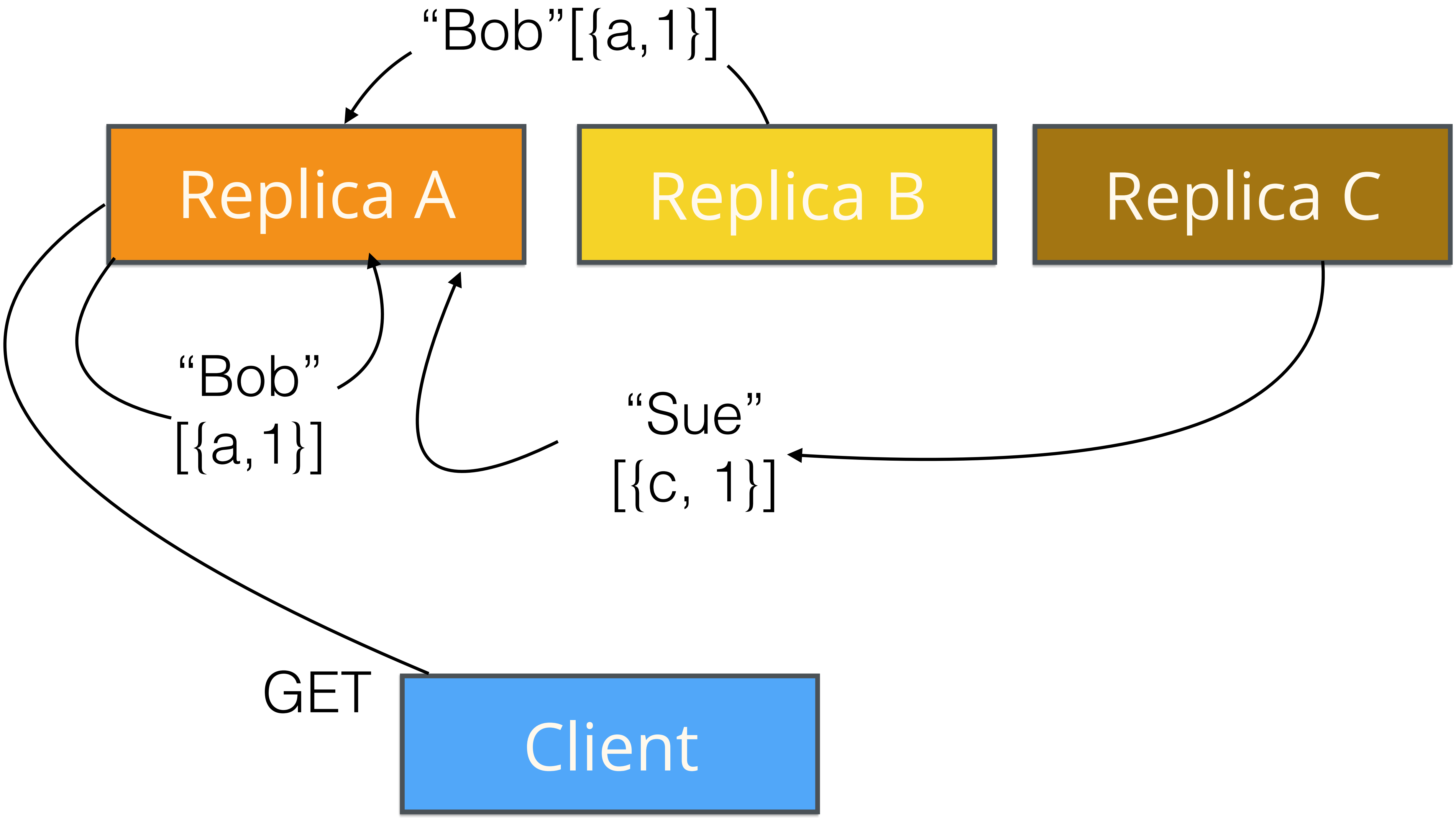
SF → NY

4,148 km
14 ms Light
21 ms fibre

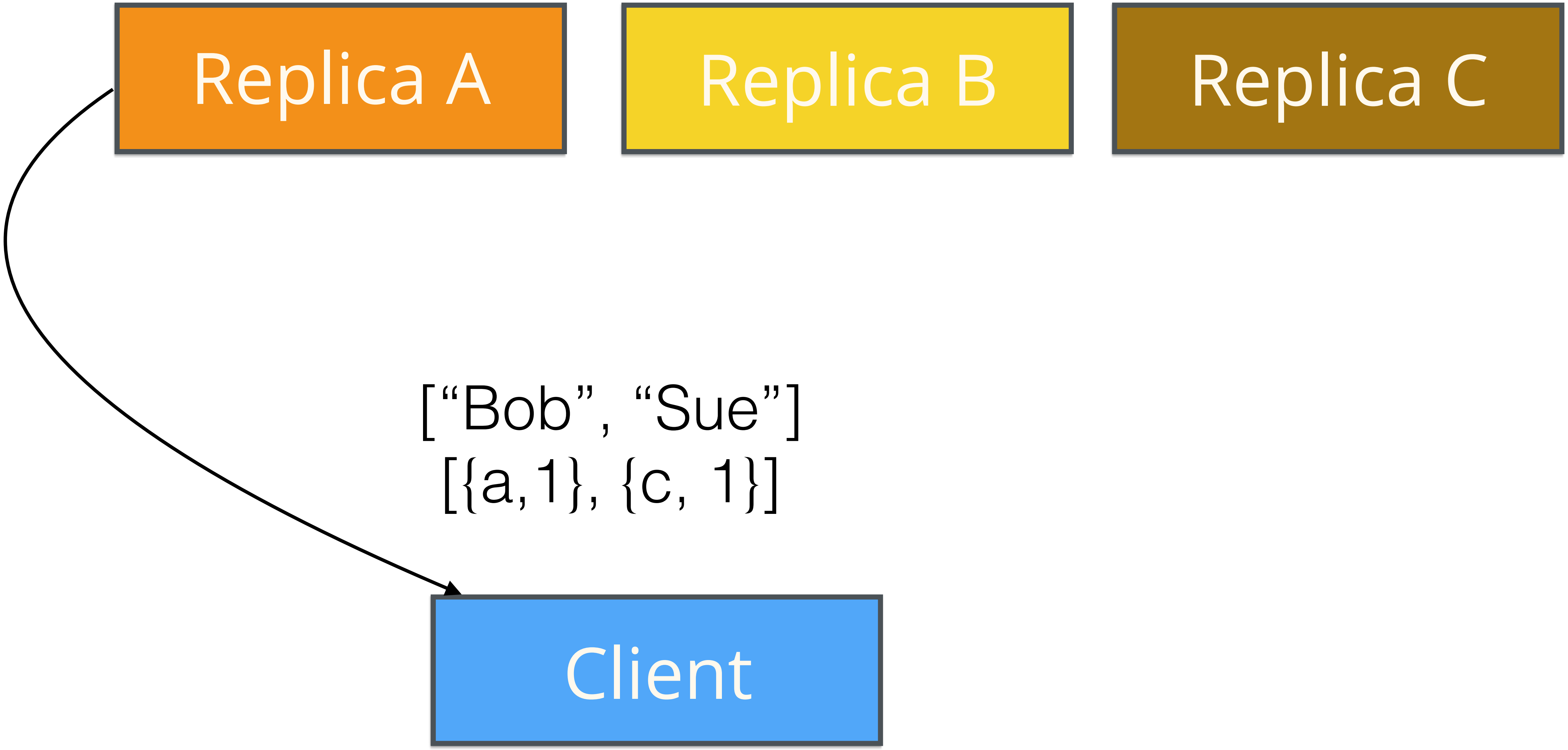


Last Write Wins

LWW - A Lossy Total Order



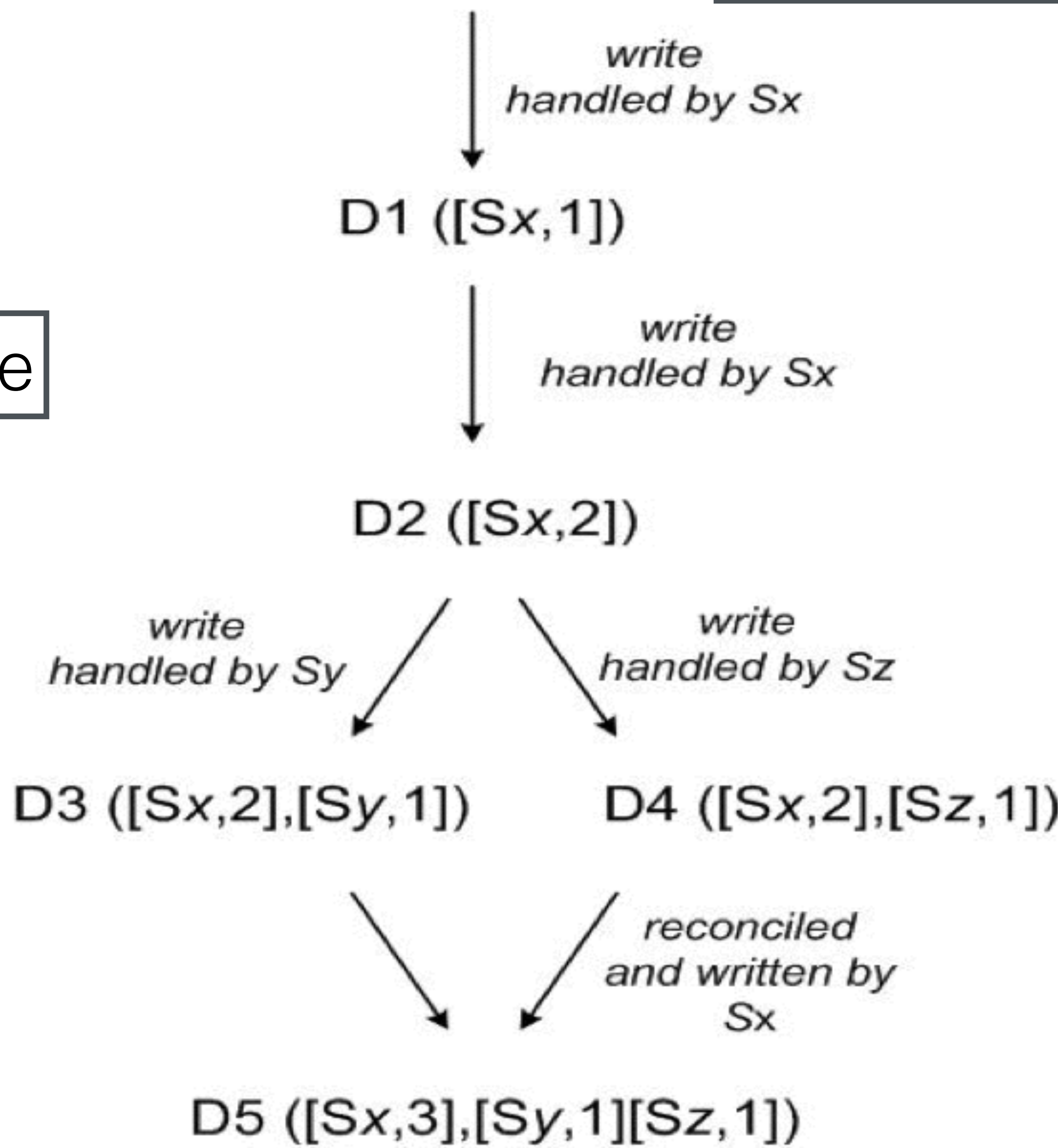
Conflict!



Multi-Value

MVR - A Partial Order

Logical Clocks



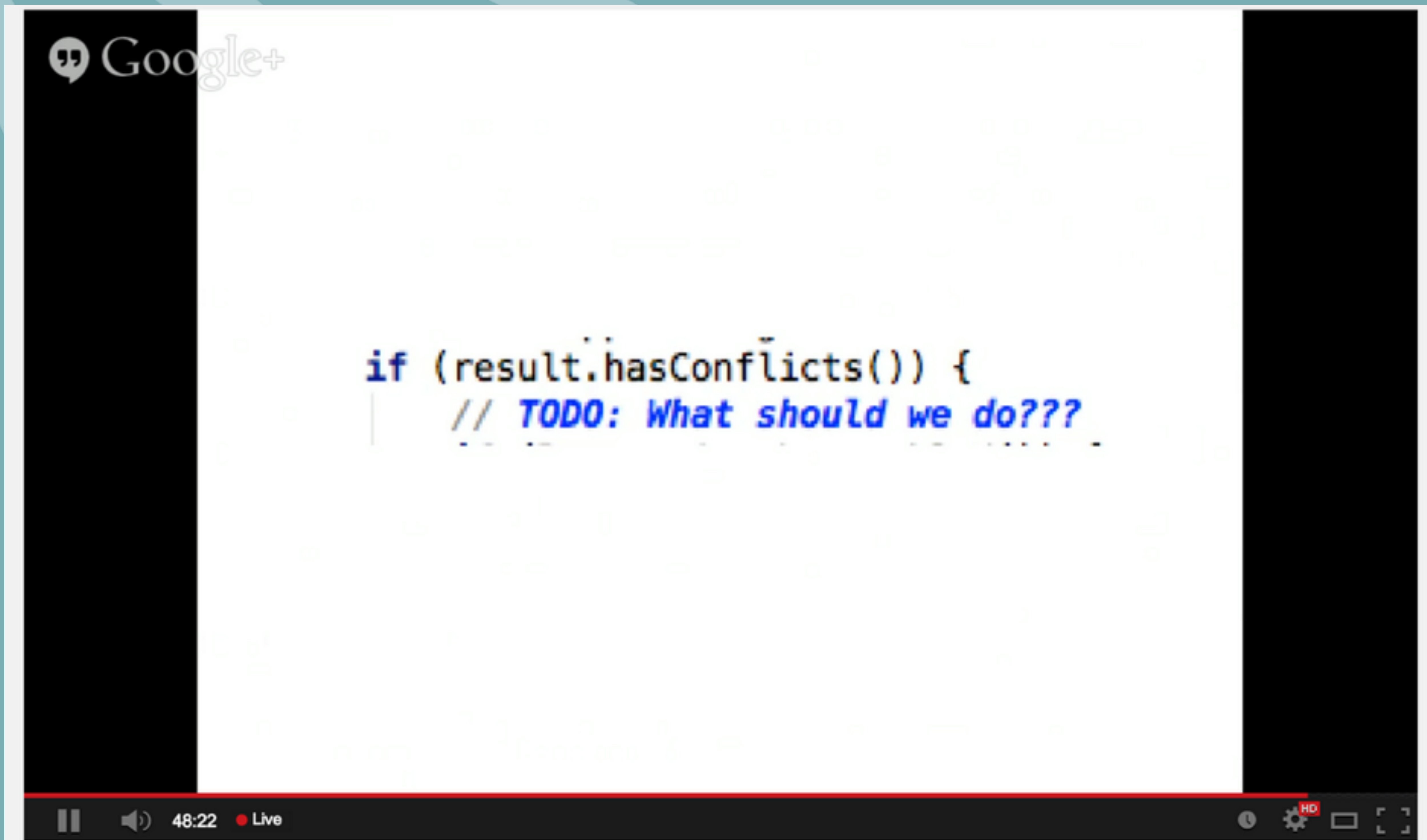
happens before

concurrent
divergent

convergent

Summary

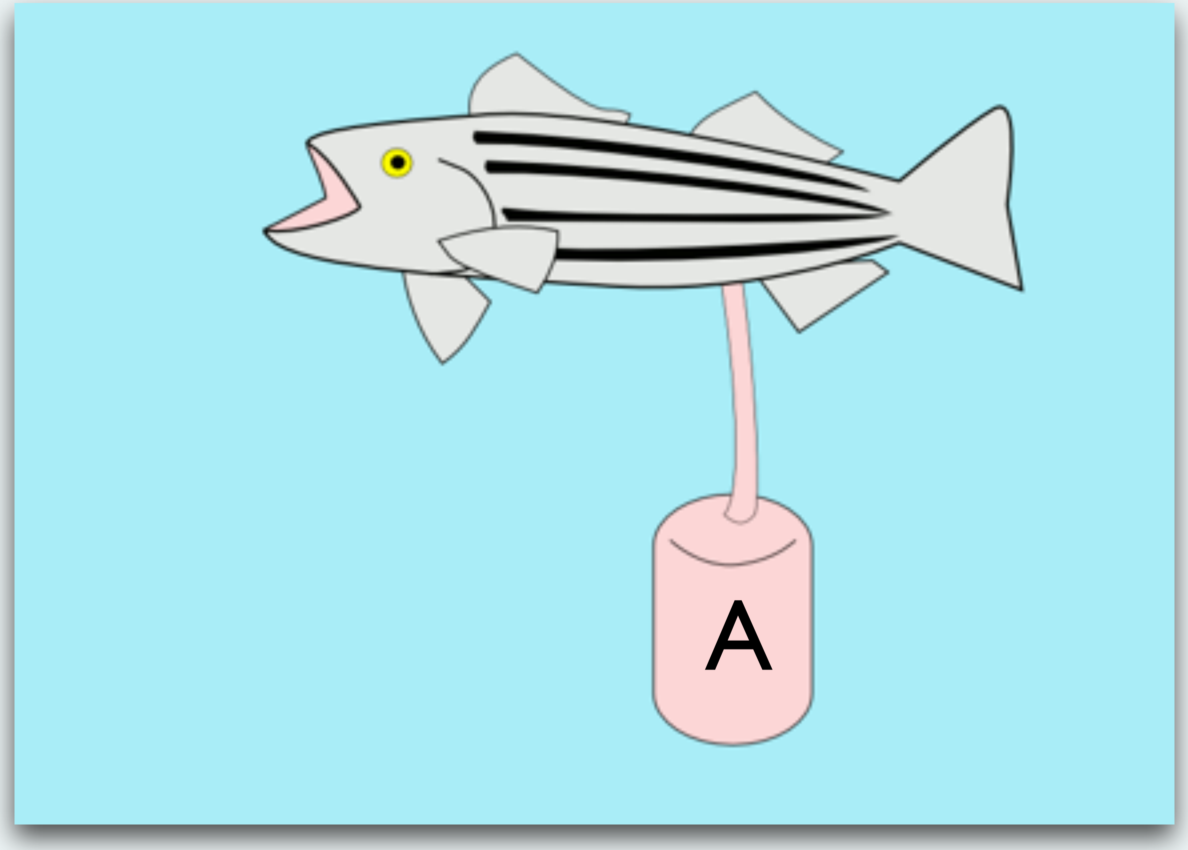
- Distributed systems for scale/fault tolerance/perf
- CAP trade-off
- Eventual consistency - concurrent writes

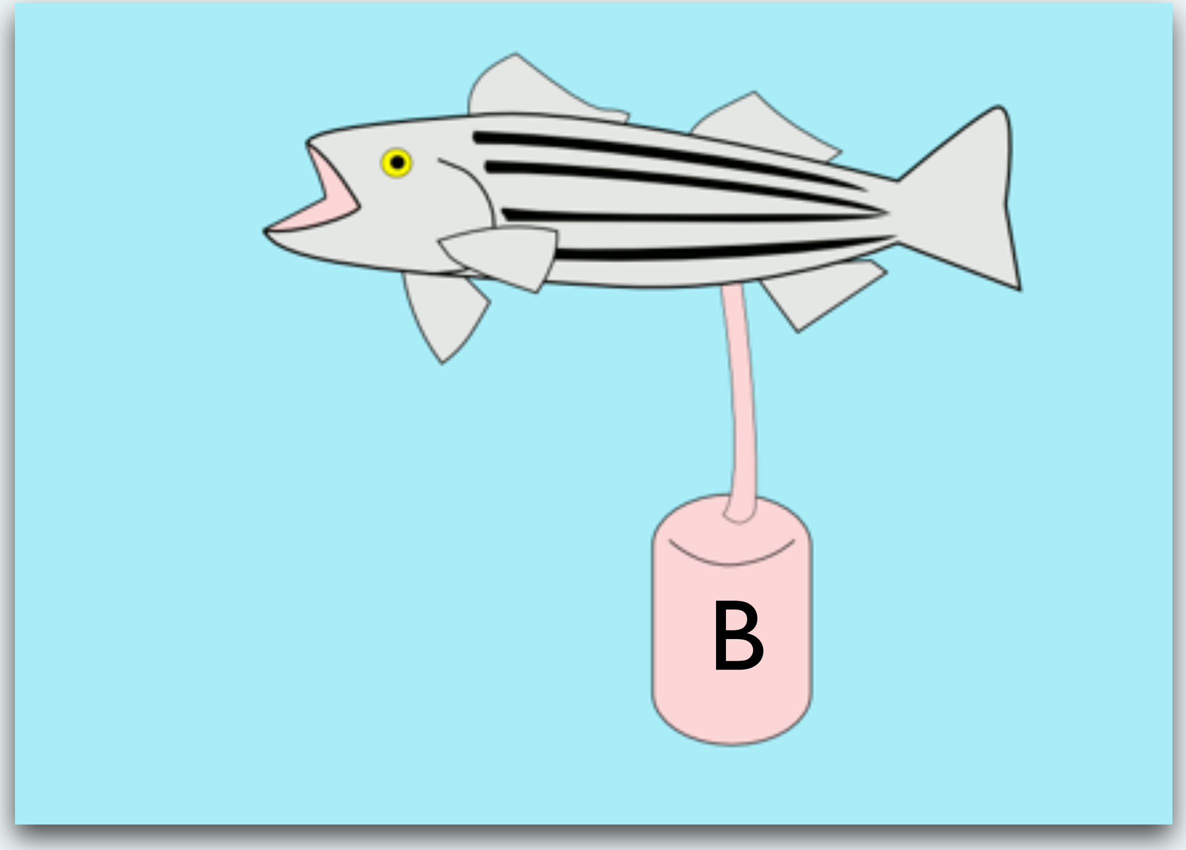

Google+
if (result.hasConflicts()) {
// TODO: What should we do???
48:22
Live
HD

Semantic Resolution

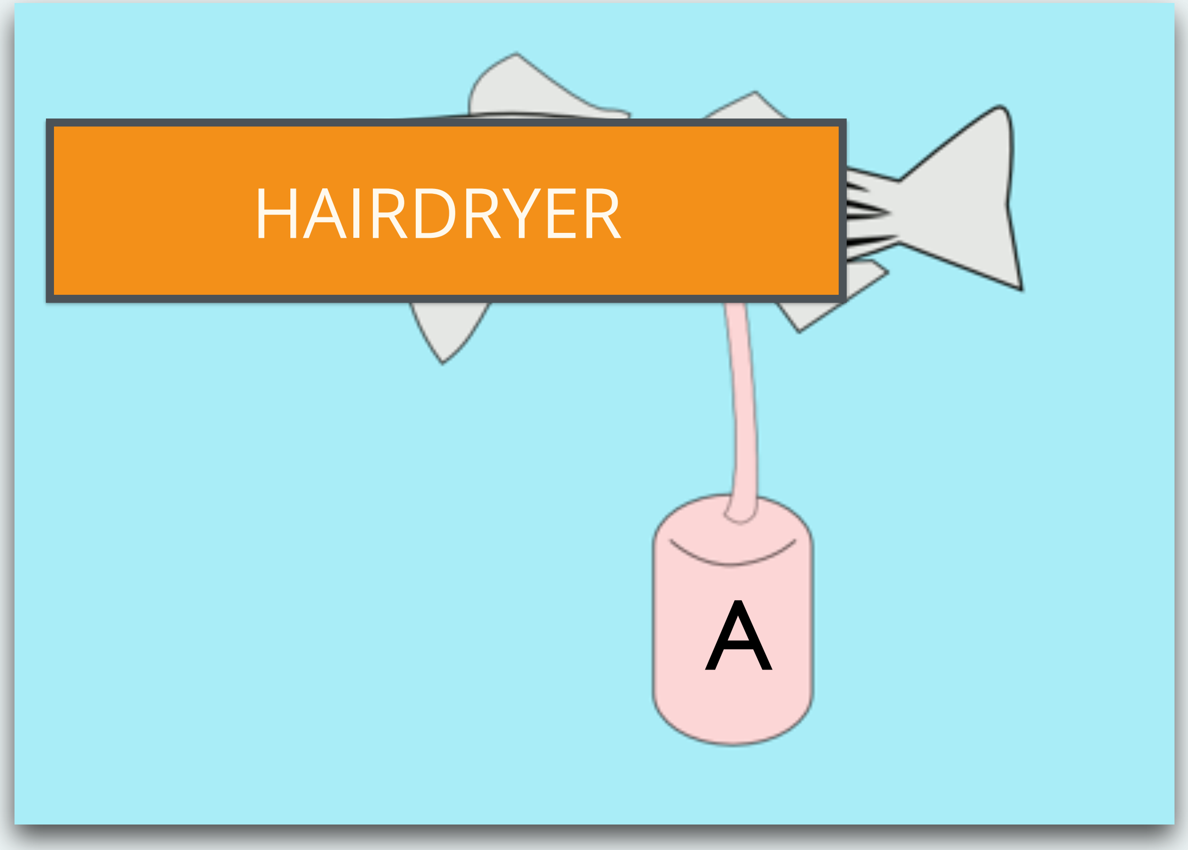
Dynamo

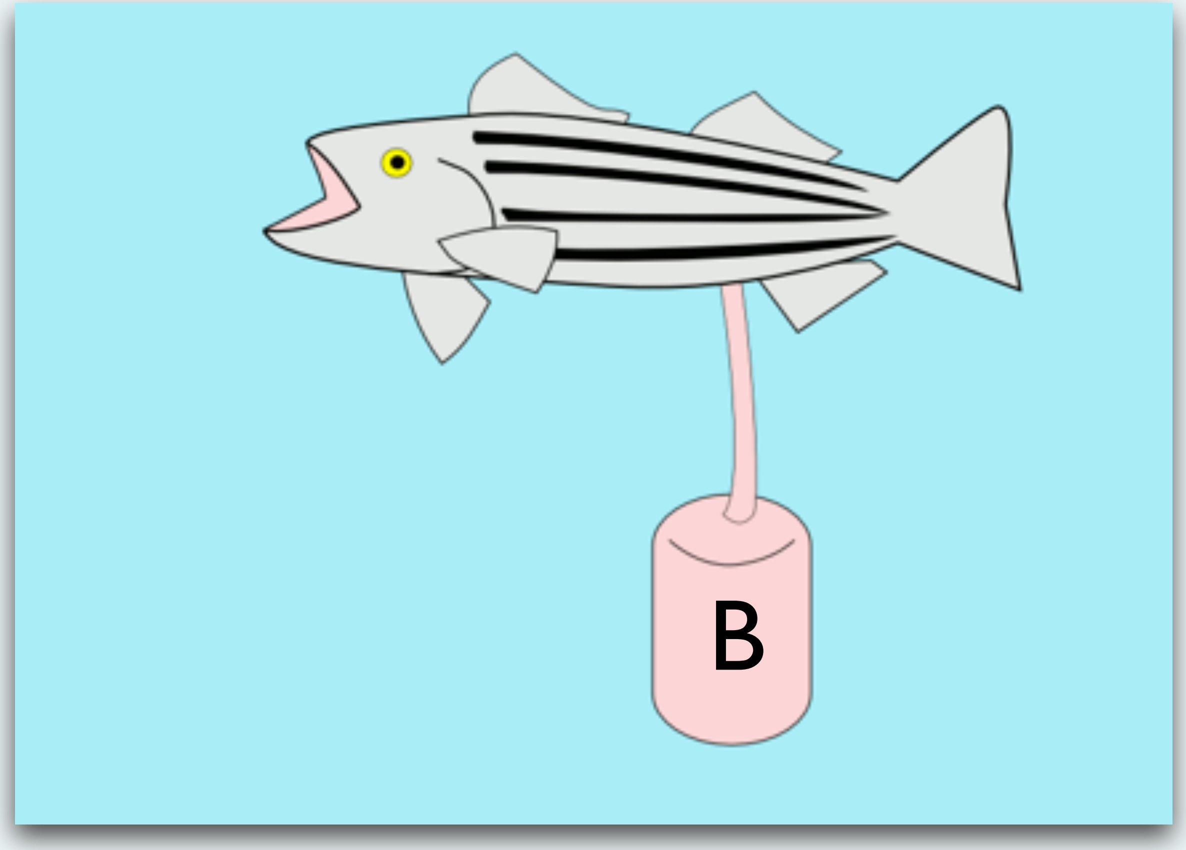
The Shopping Cart

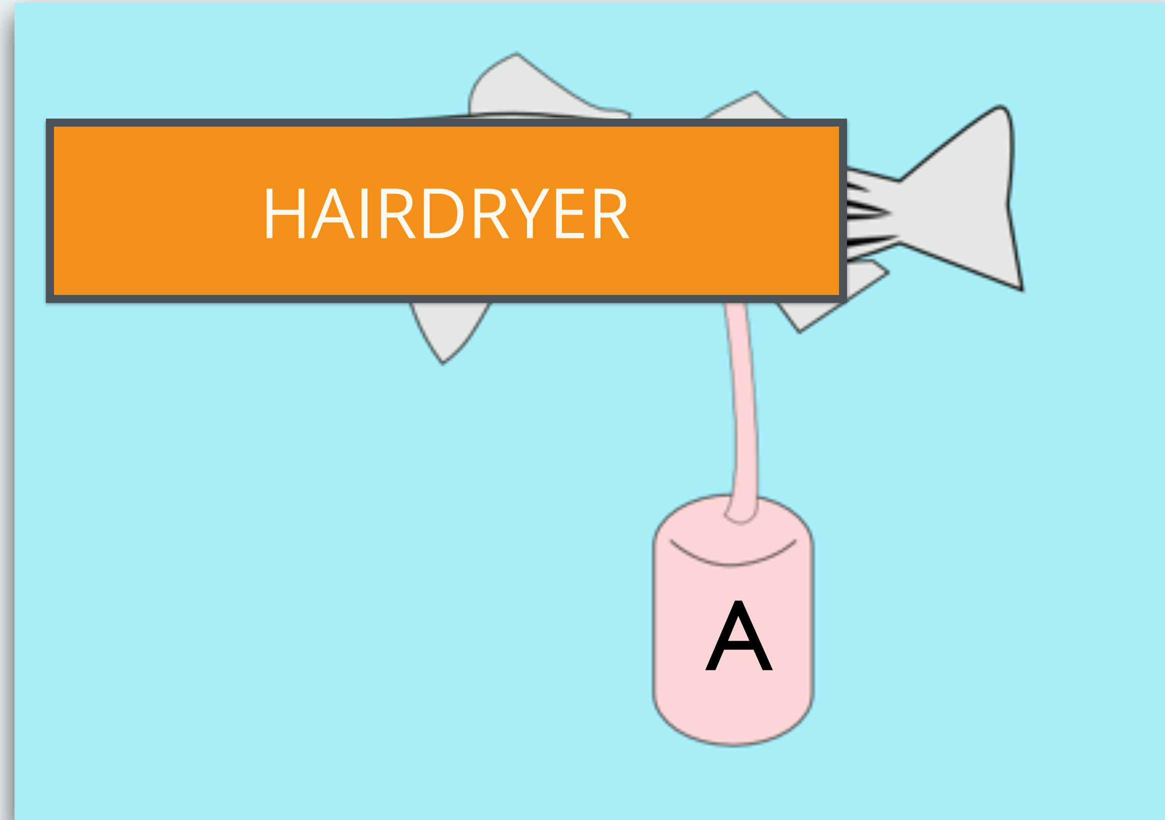


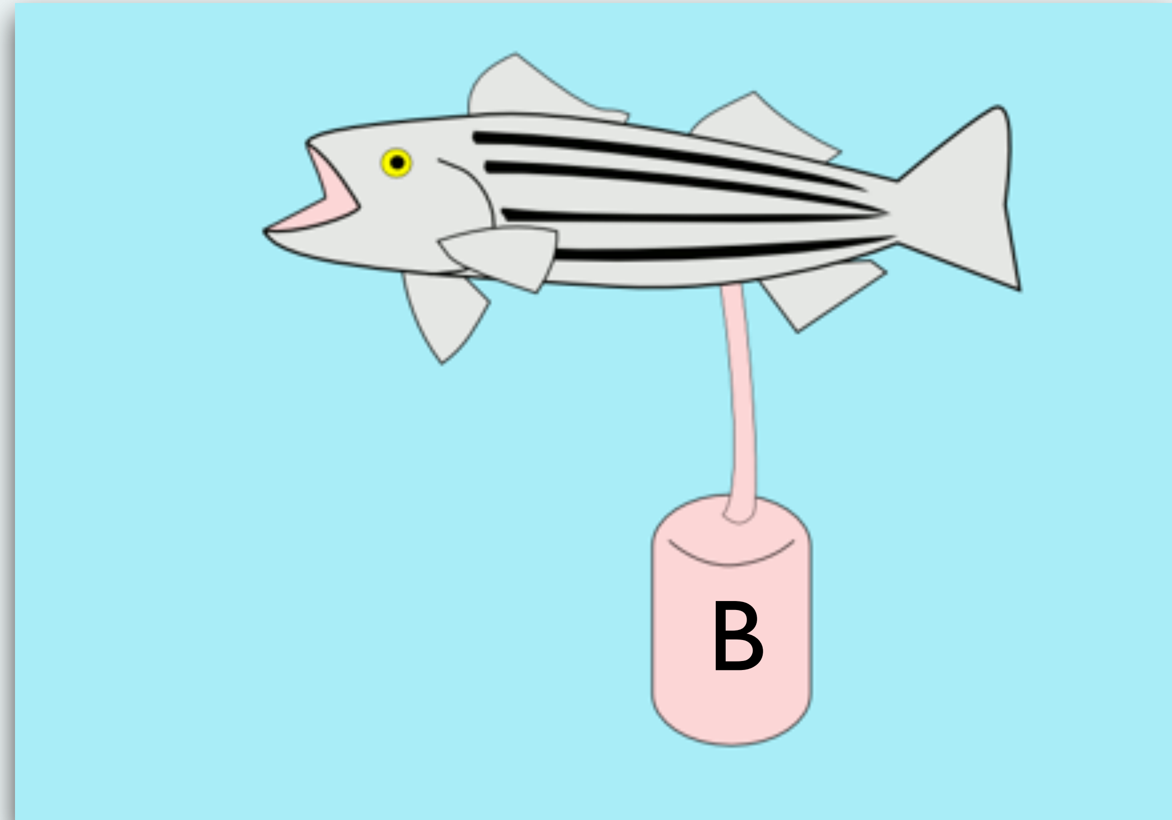


HAIRDRYER


HAIRDRYER
A

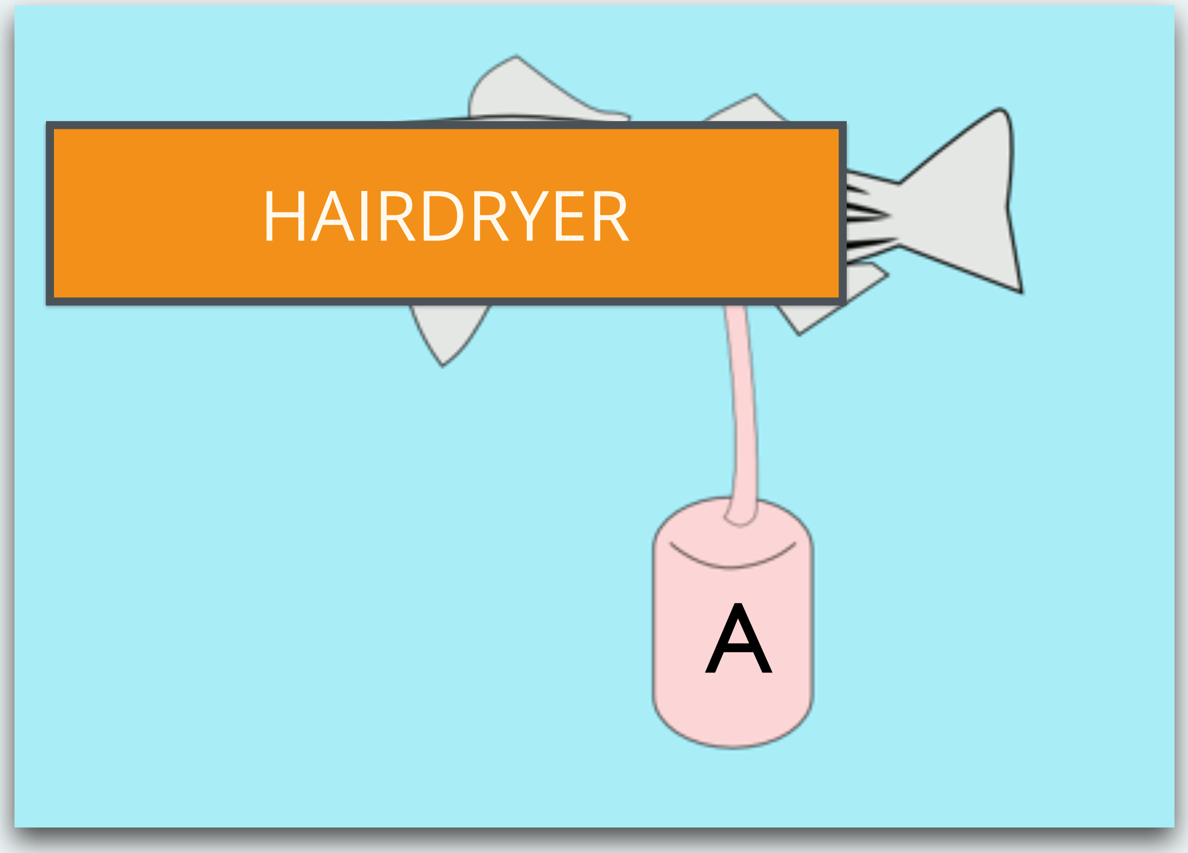

B

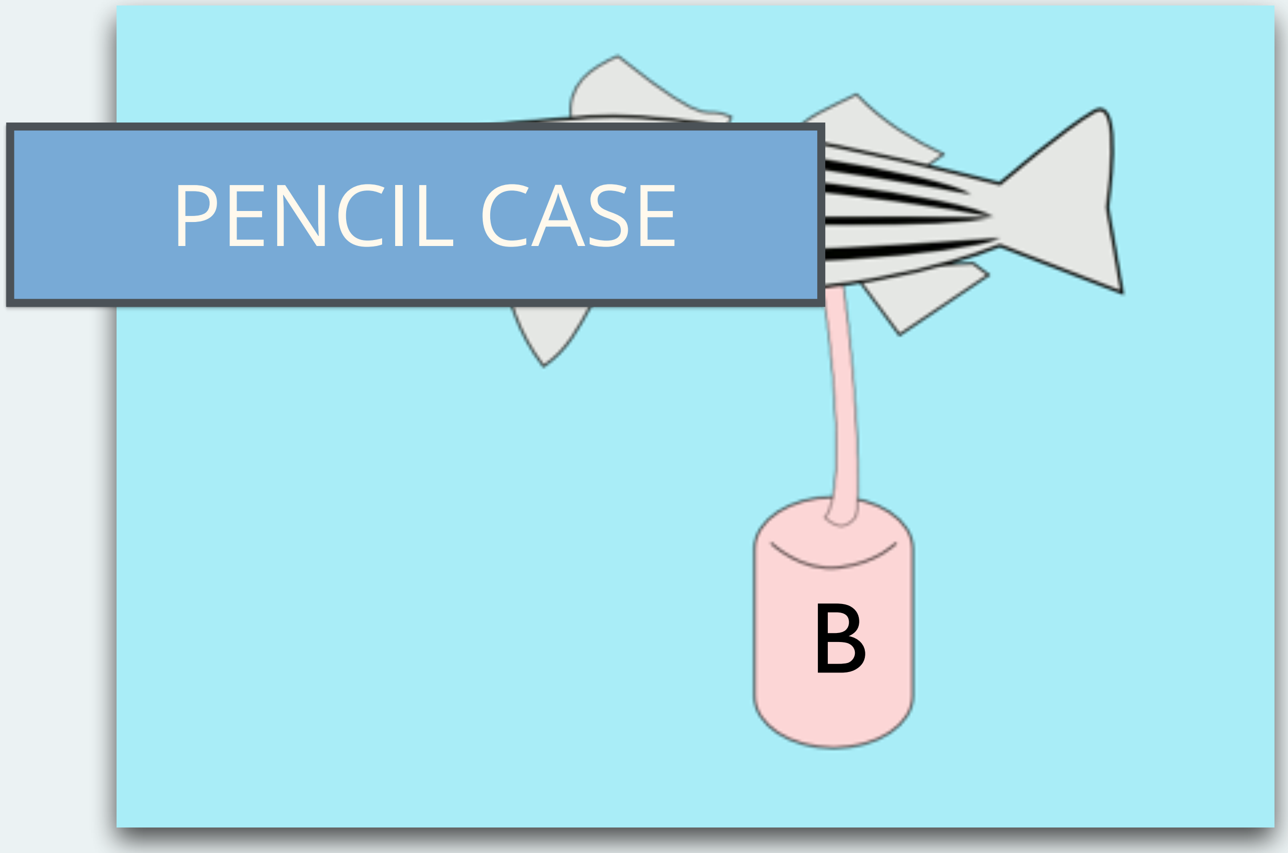


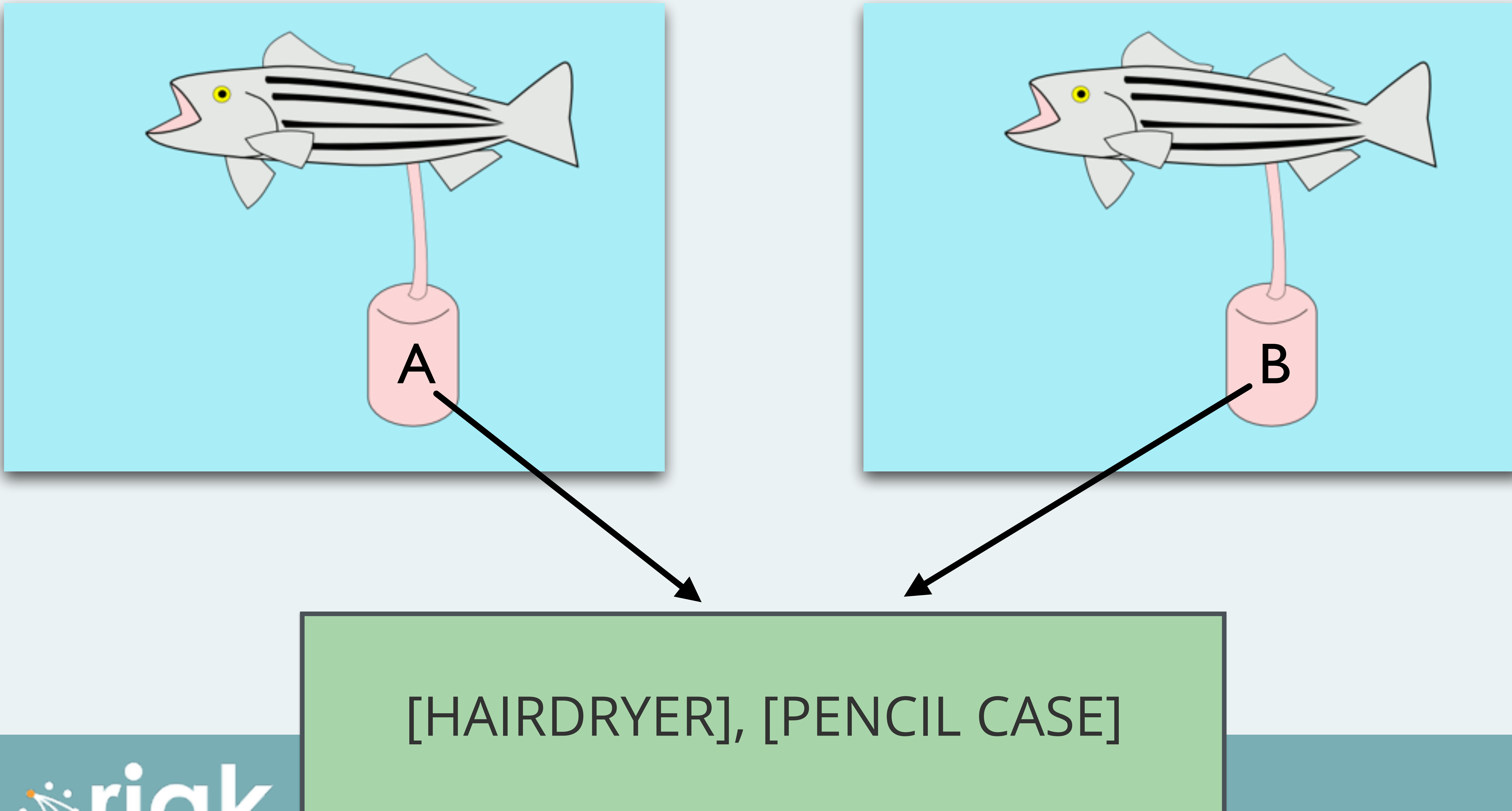


PENCIL CASE

riak


HAIRDRYER
A


PENCIL CASE
B


A
B
[HAIRDRYER], [PENCIL CASE]

Converge

Set Union of Values

Simples, right?

Removes?

Set Union?

“Anomaly”

Reappear

Google F1

“We have a lot of experience with eventual consistency systems at Google.”

“We find developers spend a significant fraction of their time building extremely complex and error-prone mechanisms to cope with eventual consistency”

Google F1

"Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains."

"…writing merge functions was likely to confuse the hell out of all our developers and slow down development…"

http://www.infoq.com/articles/key-lessons-learned-from-transition-to-nosql

Ad Hoc

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A comprehensive study of Convergent and Commutative Replicated Data Types

Marc Shapiro, INRIA & LIP6, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Carlos Baquero, Universidade do Minho, Portugal

Marek Zawirski, INRIA & UPMC, Paris, France

riak

What’s a CRDT?

State Based CRDT
Convergent

Join Semi-lattice

Join Semi-lattice

Partially ordered set; Bottom; least upper bound

$$\langle \mathbf{S}, \quad \bot, \quad \sqcup \rangle$$

Join Semi-lattice

Associativity: $(X \sqcup Y) \sqcup Z = X \sqcup (Y \sqcup Z)$

Join Semi-lattice

Commutativity: $X \sqcup Y = Y \sqcup X$

Join Semi-lattice

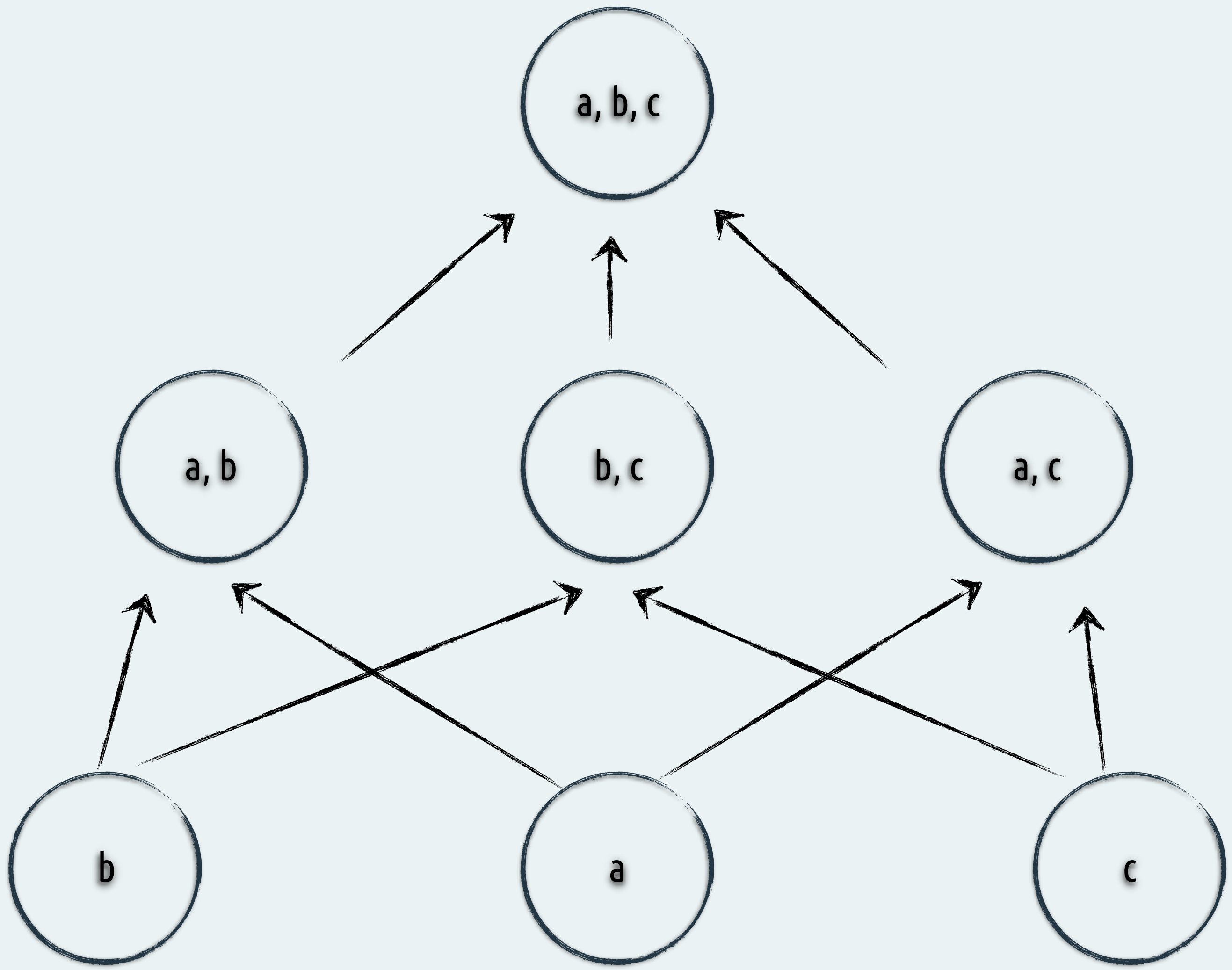
Idempotent: $X \sqcup X = X$

Join Semi-lattice

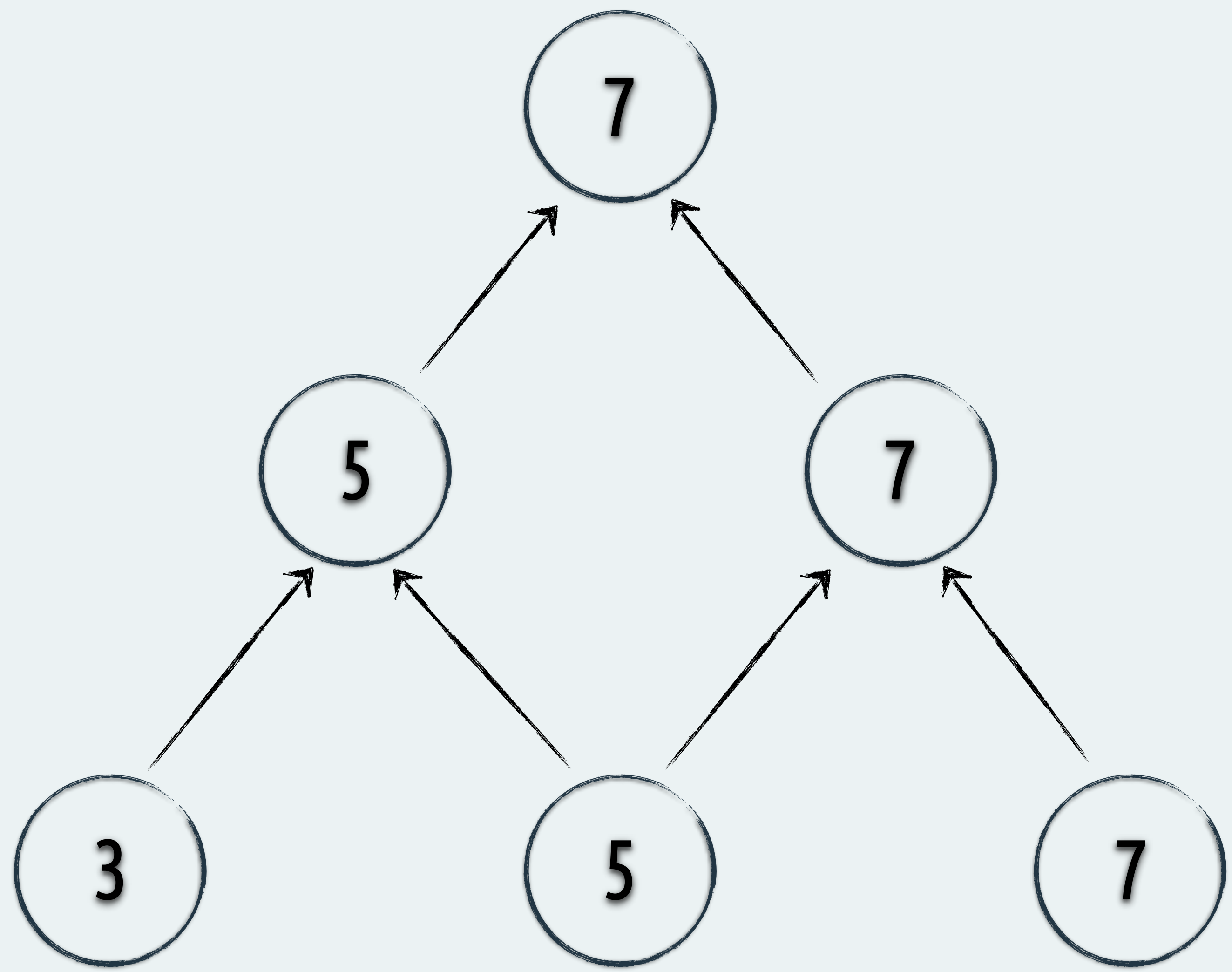
Objects grow over time; merge computes LUB

Join Semi-lattice

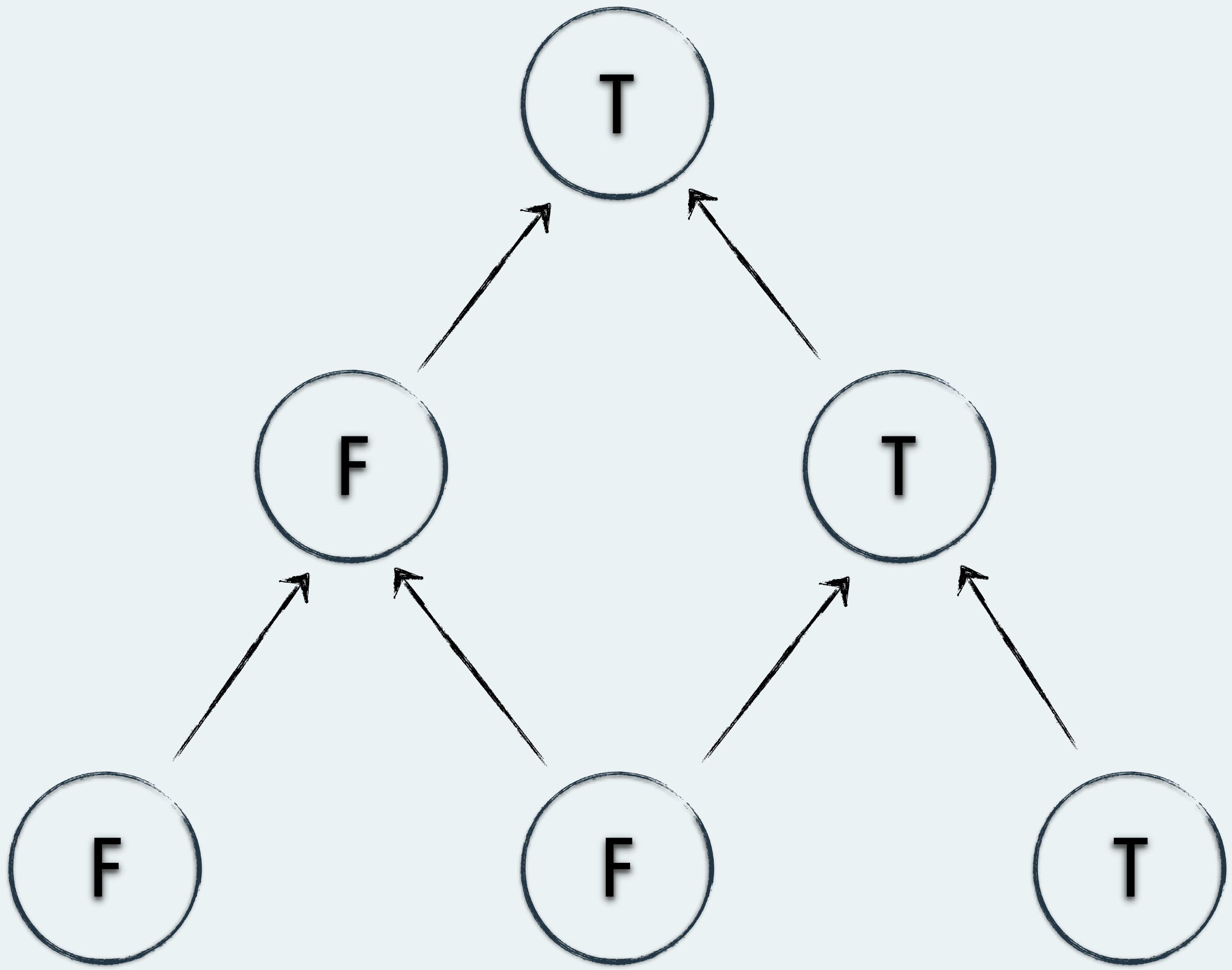
Examples



Set; merge function: union.



Increasing natural; merge function: max.



Booleans; merge function: or.

Merge

Deterministic

Idempotent

Associative

Commutative

Principled Merge

Reusable

Data Types

Defined Semantics

Evolution of a CRDT Set

“…after some analysis we found that much of our data could be modelled within sets so by leveraging CRDT’s our developers don't have to worry about writing bespoke merge functions for 95% of carefully selected use cases…”

riak

http://www.infoq.com/articles/key-lessons-learned-from-transition-to-nosql

Evolution of a CRDT Set

Evolution of a Set

G-SET

Evolution of a Set

G-SET

Shelly

Bob

Shelly

Bob

Pete

Shelly

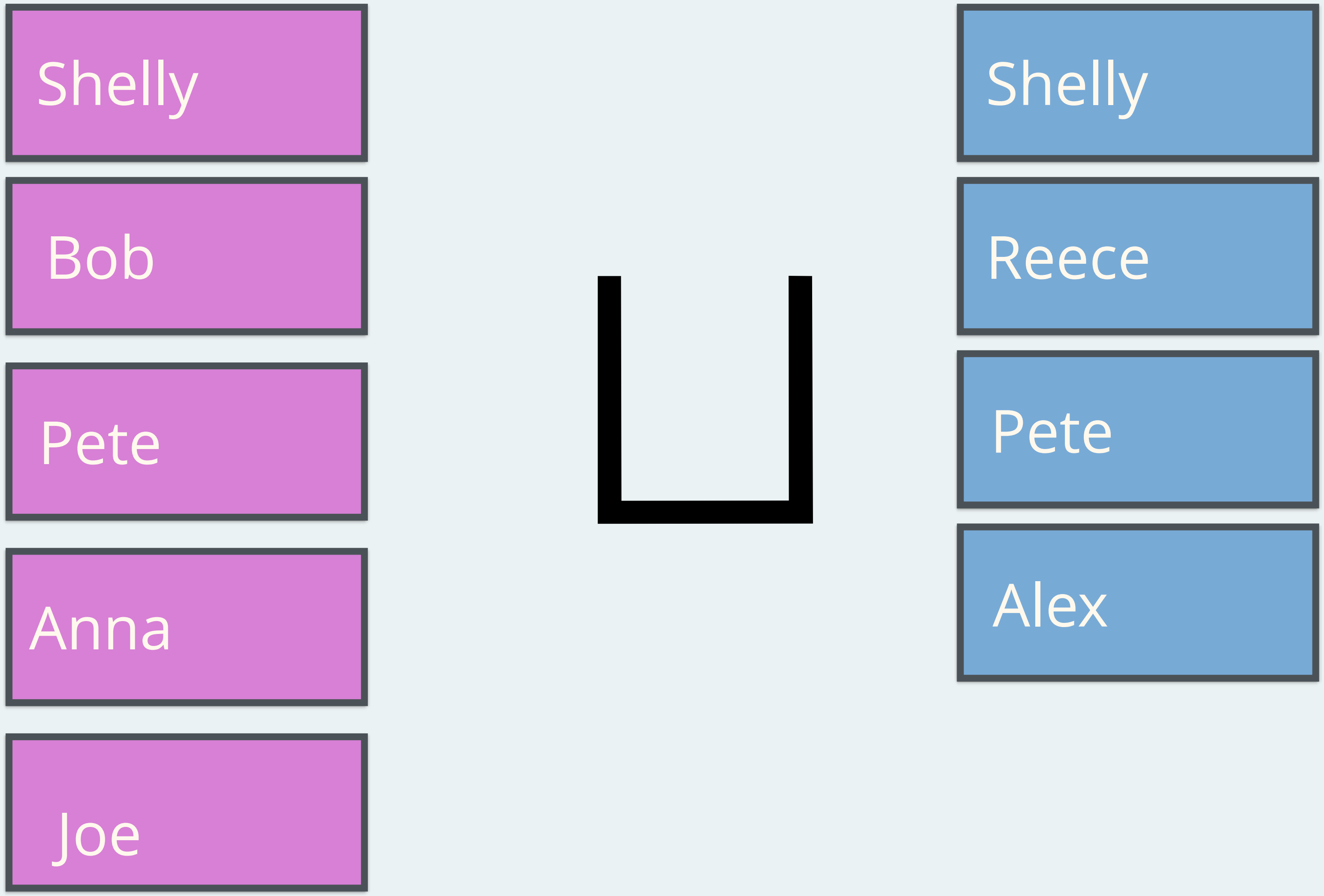
Bob

Pete

Anna

Joe

Shelly	Shelly
Bob	Reece
Pete	Pete
Anna	Alex
Joe	


Shelly
Bob
Pete
Anna
Joe
⊔
Shelly
Reece
Pete
Alex

Shelly

Bob

Pete

Anna

Joe

Reece

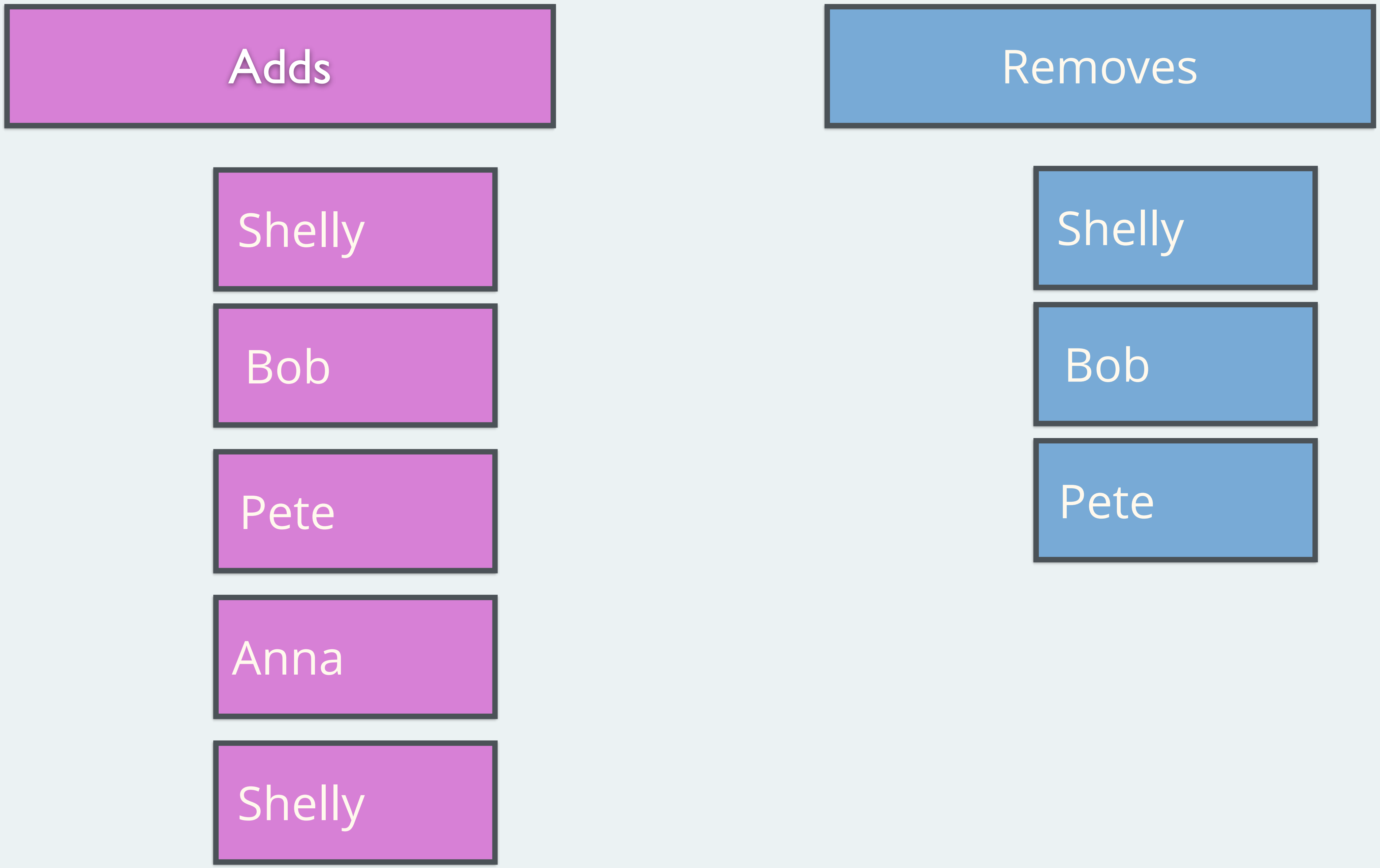
Alex

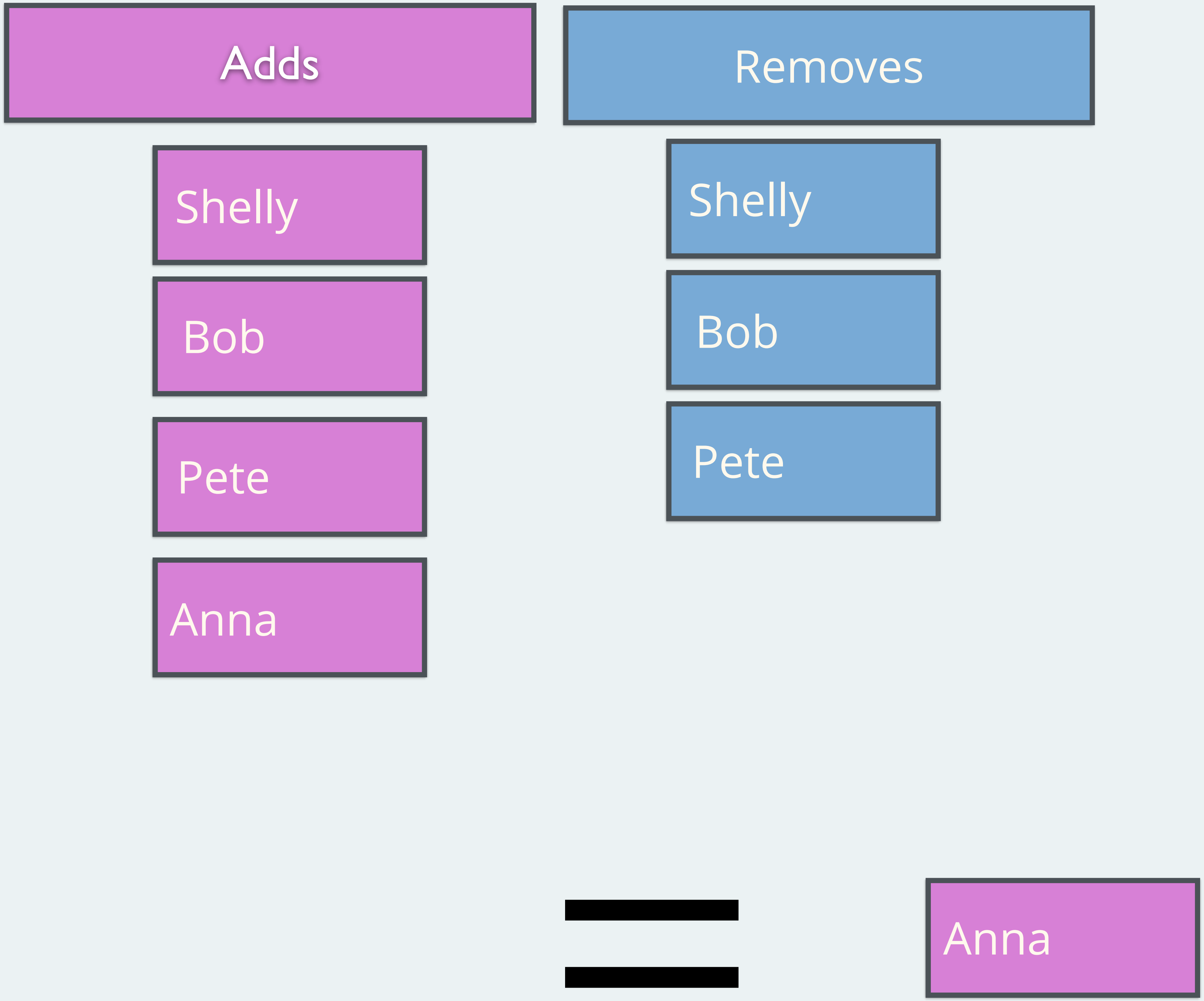
Removes?

Evolution of a Set

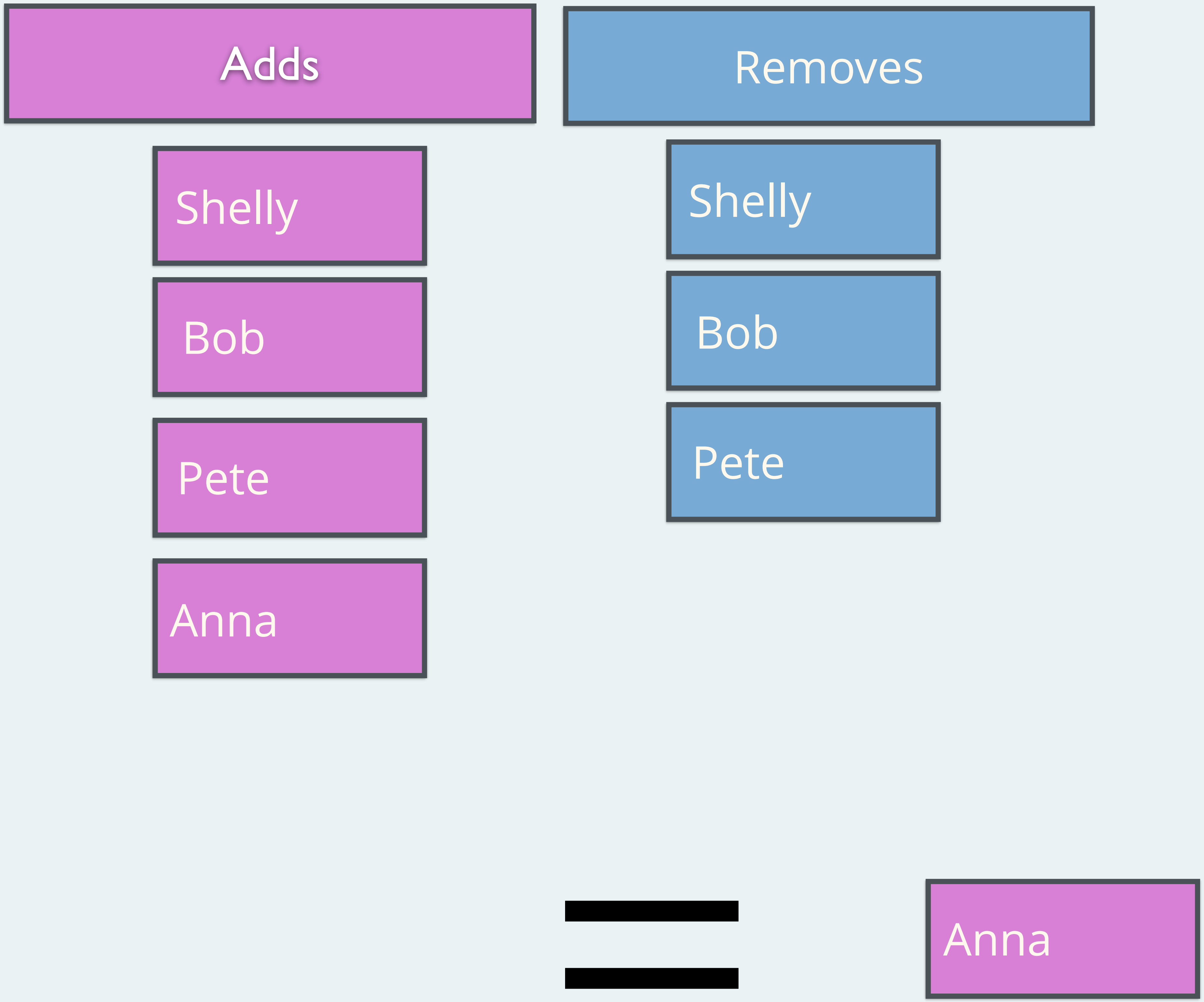
G-SET

2P-SET


Adds
Shelly
Bob
Pete
Anna
Shelly
Removes
Shelly
Bob
Pete


Adds
Removes
Shelly
Bob
Pete
Anna
Shelly
Bob
Pete
Anna

Value /= Structure


Adds
Removes
Shelly
Bob
Pete
Anna
Shelly
Bob
Pete
=
Anna

I changed my mind!

CRDT Sets

answers the question of "what is in the set?" when presented with siblings:

$$[x,y,z] \mid [w,x,y]$$

CRDT Sets

is w not added by A or removed by A?
is z not added be B or removed by B?

[x,y,z] | [w,x,y]

Evolution of a Set

U-SET

1	Shelly
2	Bob
3	Pete
4	Anna

$$\sqcup$$

5	Shelly
6	Jack
7	Jed

1,5	Shelly
2	Bob
3	Pete
4	Anna
6	Jack
7	Jed

Evolution of a Set

U-SET
OR-SET

Adds

1	Shelly
2	Bob
3	Pete
4	Anna

Removes

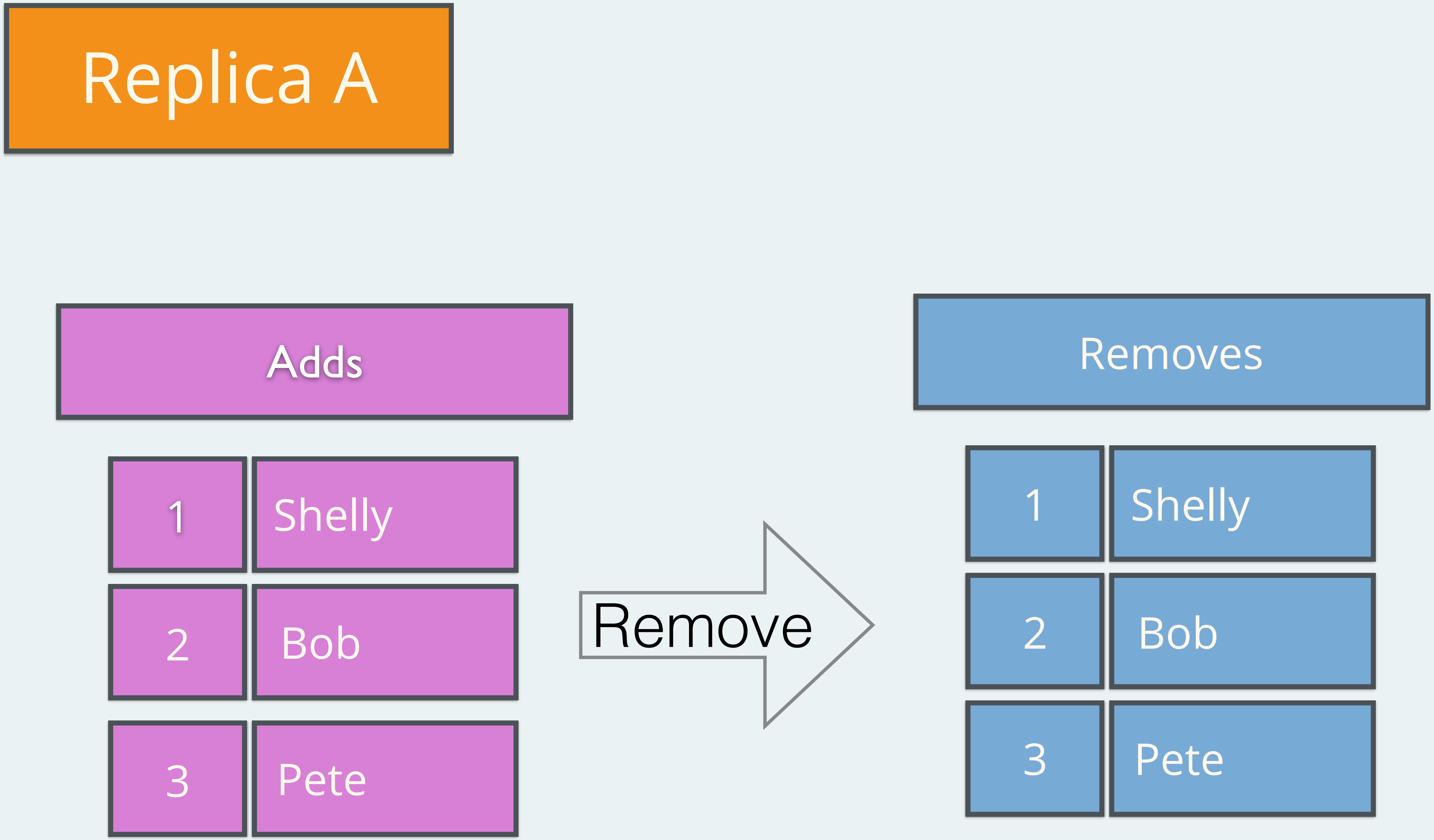
1	Shelly
2	Bob
3	Pete

Adds	
1	Shelly
2	Bob
3	Pete
4	Anna
5	Shelly

Removes	
1	Shelly
2	Bob
3	Pete

Replica A

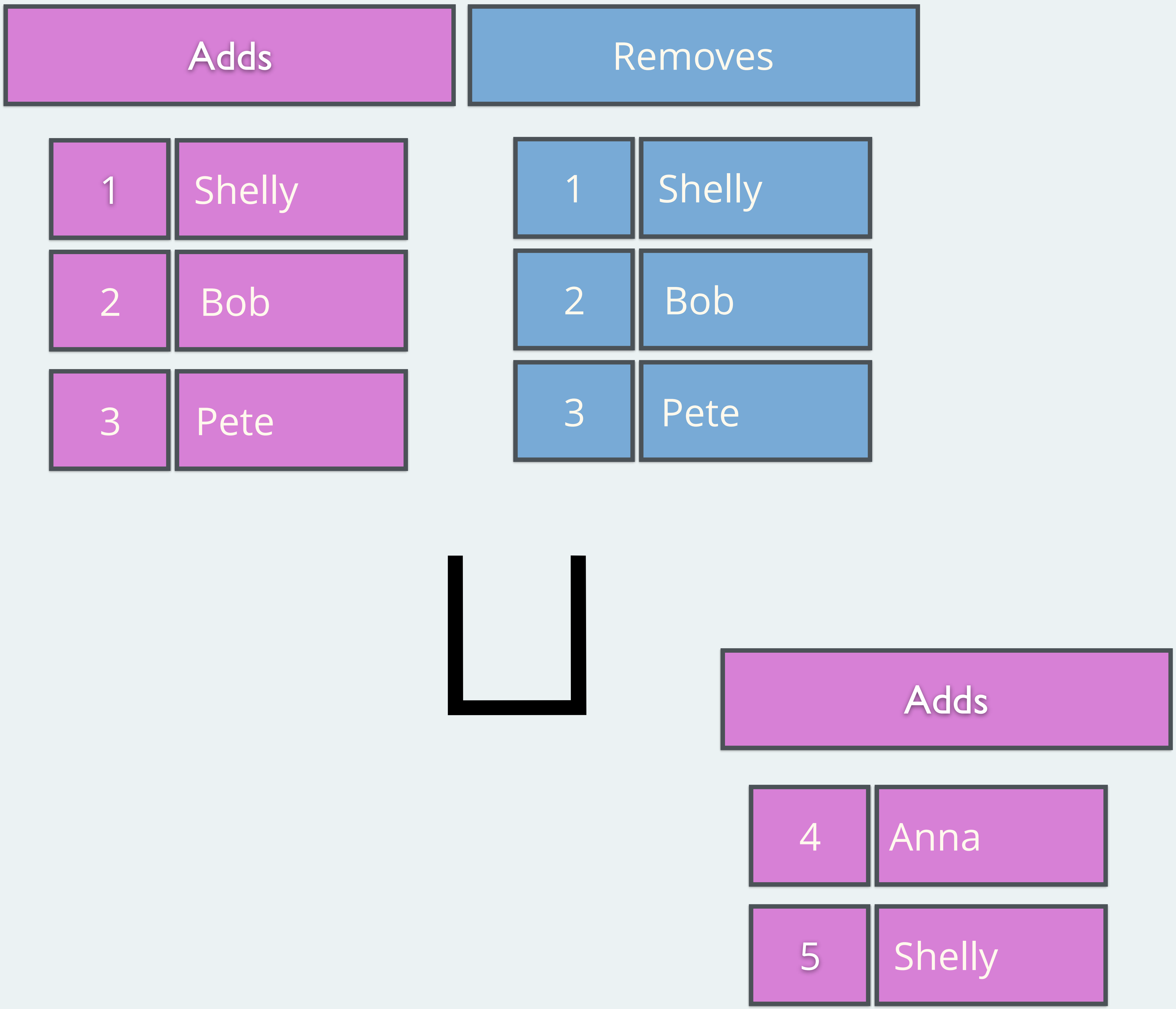
Adds	
1	Shelly
2	Bob
3	Pete


Replica A
Adds
1
Shelly
2
Bob
3
Pete
Remove
Removes
1
Shelly
2
Bob
3
Pete

Replica B

Adds

4	Anna
5	Shelly


Adds
Removes
1
Shelly
2
Bob
3
Pete
1
Shelly
2
Bob
3
Pete
⊔
Adds
4
Anna
5
Shelly

Adds	
1	Shelly
2	Bob
3	Pete
4	Anna
5	Shelly

Removes	
1	Shelly
2	Bob
3	Pete

=

Anna

Shelly

Observed Remove

Semantics

Add Wins

Evolution of a Set

U-SET

OR-SET

Adds	
1	Shelly
2	Bob
3	Pete
4	Anna
5	Shelly

Removes	
1	Shelly
2	Bob
3	Pete
4	Anna
5	Shelly

= Anna Shelly [ ]

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Optimized Conflict-free Replicated Set

Annette Bieniusa, INRIA & UPMC, Paris, France
Marek Zawirski, INRIA & UPMC, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Marc Shapiro, INRIA & LIP6, Paris, France
Carlos Baquero, HASLab, INESC TEC & Universidade do Minho, Portugal
Valter Balegas, CITI, Universidade Nova de Lisboa, Portugal

Sérgio Duarte CITI, Universidade Nova de Lisboa, Portugal

11 Oct 2012

Dotted Version Vectors: Logical Clocks for Optimistic Replication

Nuno Preguiça
CITI/DI
FCT, Universidade Nova de Lisboa
Monte da Caparica, Portugal
nmp@di.fct.unl.pt

Carlos Baquero, Paulo Sérgio Almeida,
Victor Fonte, Ricardo Gonçalves
CCTC/DI
Universidade do Minho
Braga, Portugal
{cbm,psa,vff}@di.uminho.pt, rtg@lsd.di.uminho.pt

Abstract

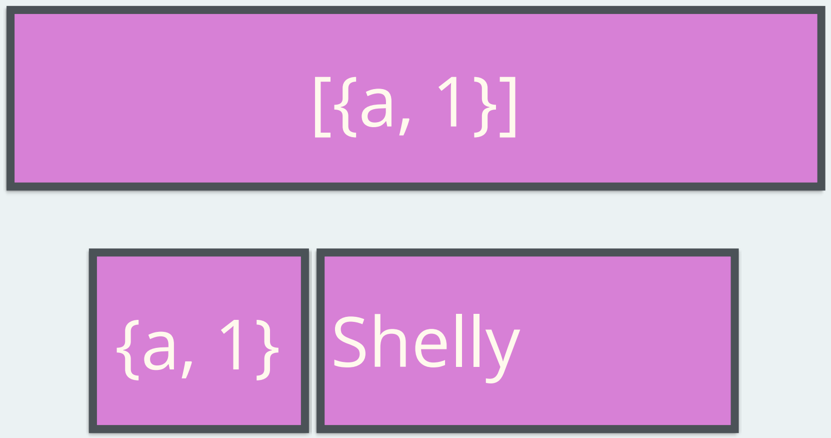
In cloud computing environments, a large number of users access data stored in highly available storage systems. To provide good performance to geographically disperse users and allow operation even in the presence of failures or network partitions, these systems often rely on optimistic replication solutions that guarantee only eventual consistency. In this scenario, it is important to be able to accurately and efficiently

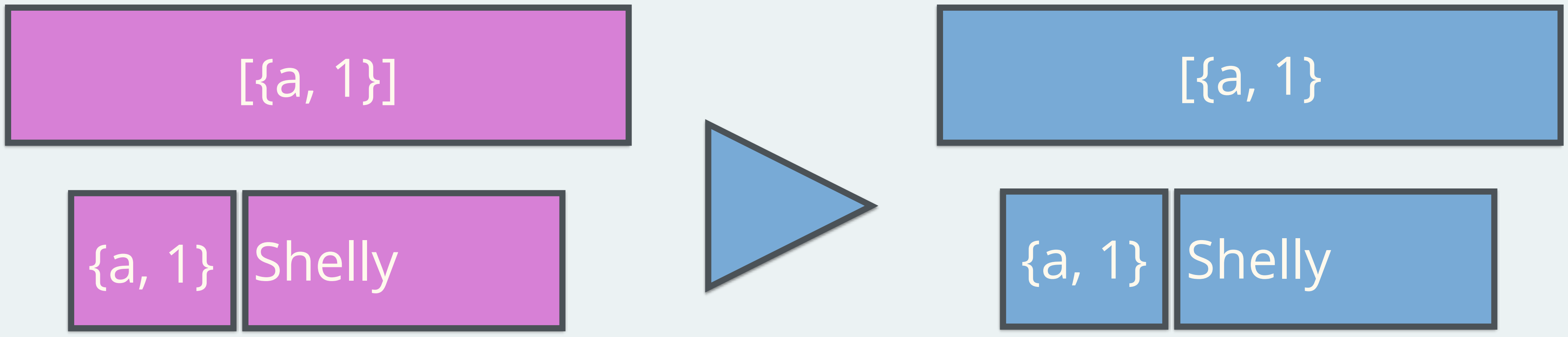
The mentioned systems follow a design where the data store is always writable. A consequence is that replicas of the same data item are allowed to diverge, and this divergence should later be repaired. Accurate tracking of concurrent data updates can be achieved by a careful use of well established causality tracking mechanisms [5], [6], [7], [8]. In particular, for data storage systems, version vectors [6] enables the system to compare any pair of replica versions and detect if

riak

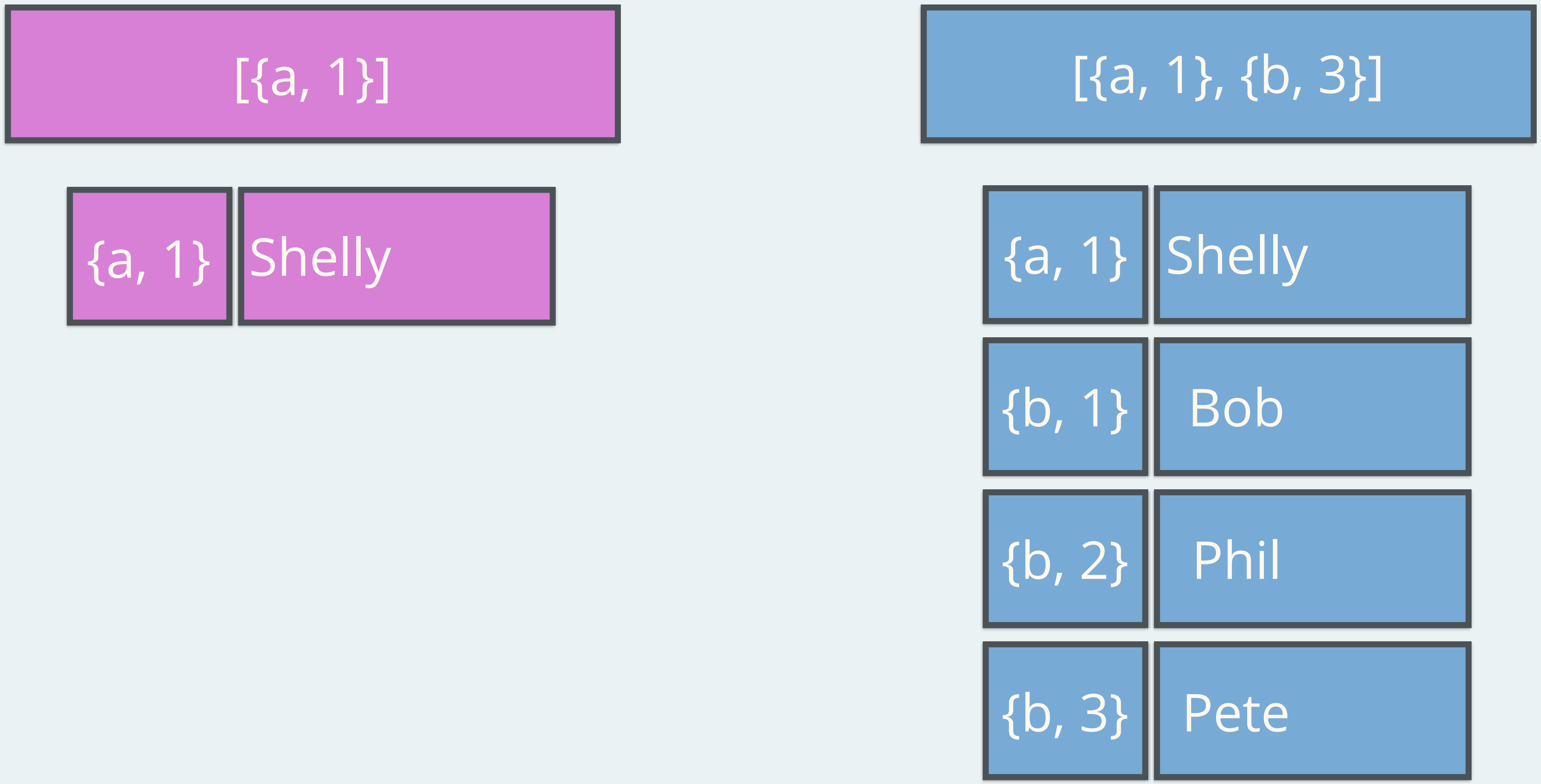
Evolution of a Set

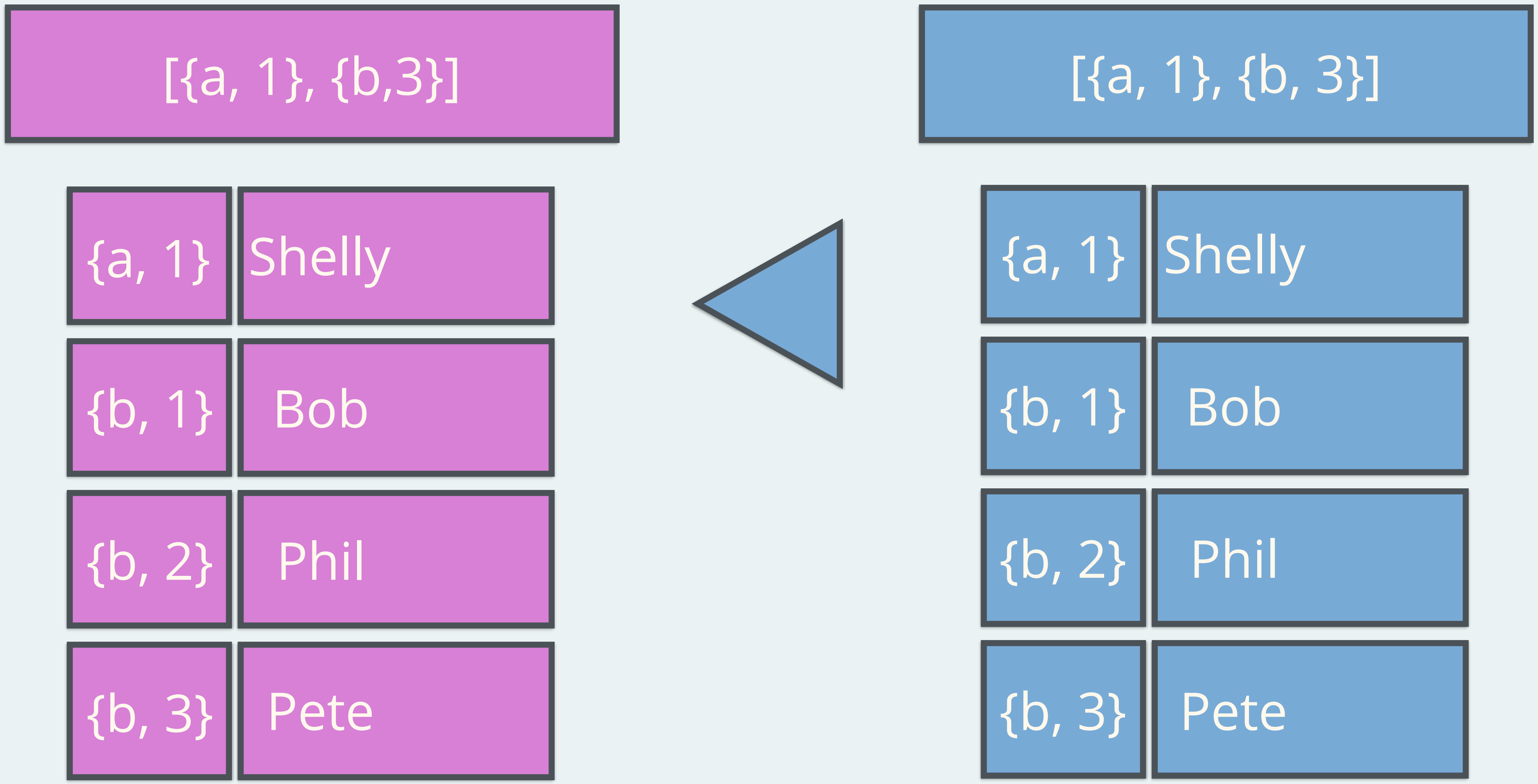
U-SET
OR-SET
OR-SWOT

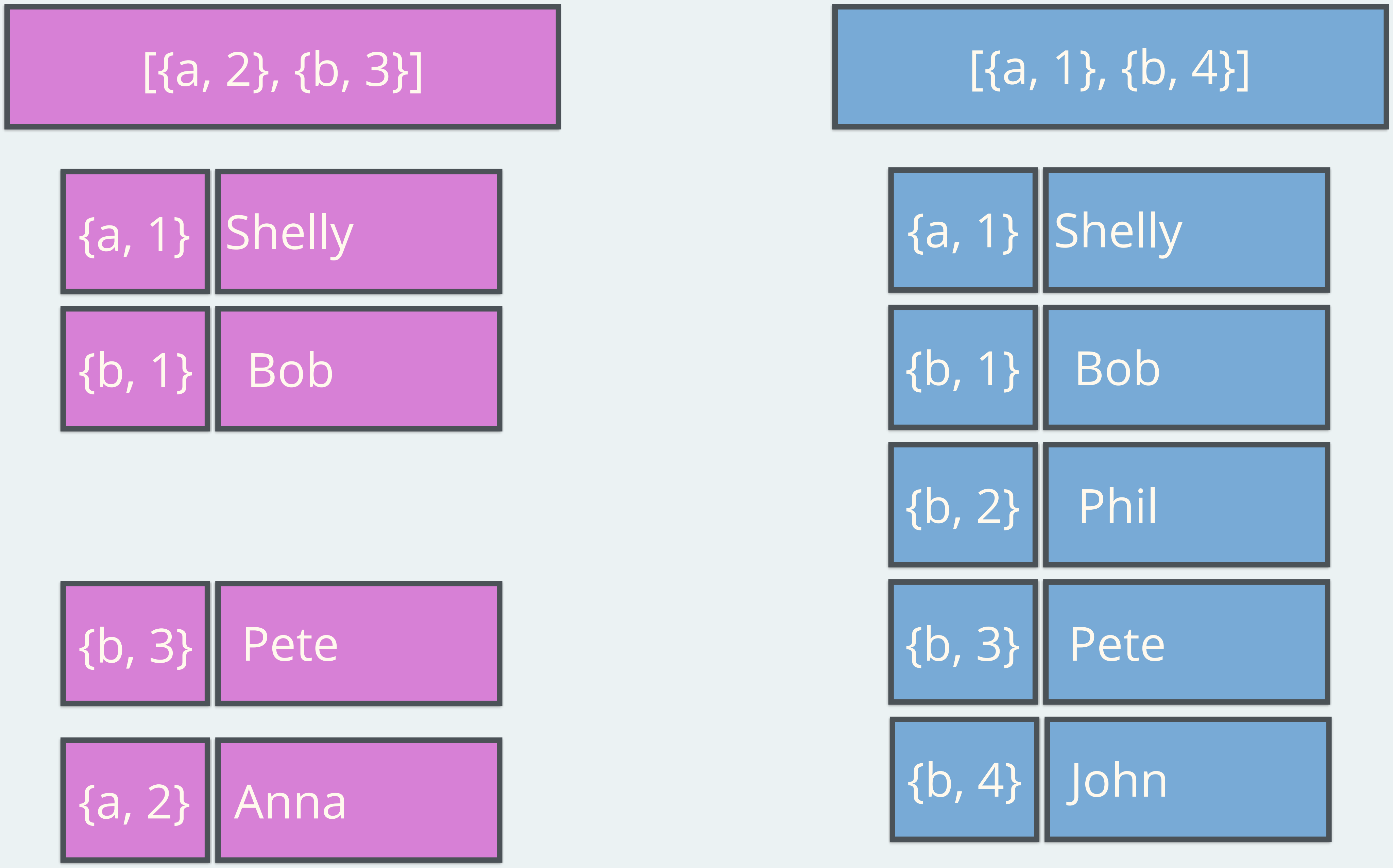

[{a, 1}]
{a, 1}
Shelly

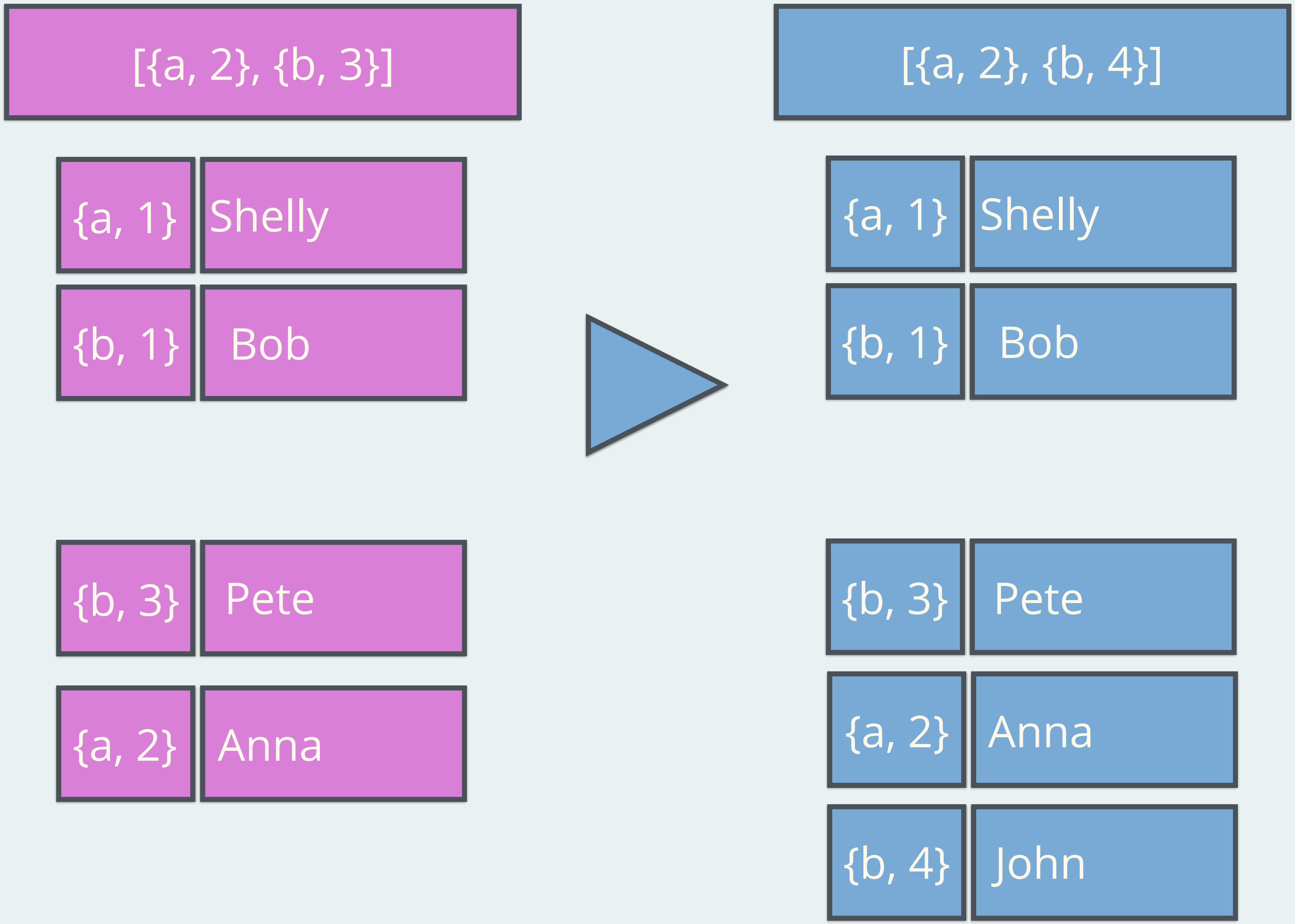


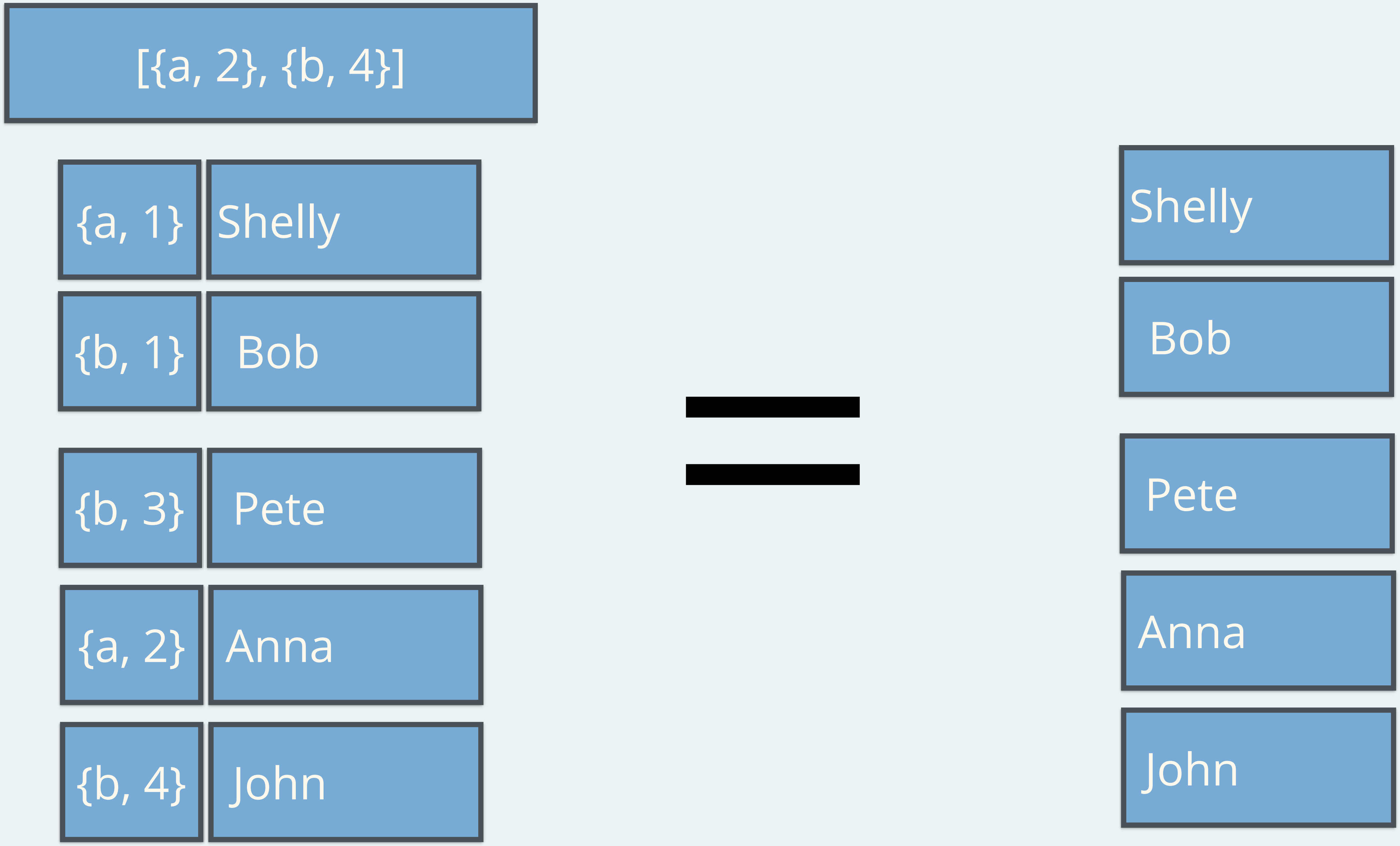
[{a, 1}]
{a, 1}
Shelly
[{a, 1}
{a, 1}
Shelly


[{a, 1}]
{a, 1}
Shelly
[{a, 1}, {b, 3}]
{a, 1}
Shelly
{b, 1}
Bob
{b, 2}
Phil
{b, 3}
Pete


[{a, 1}, {b,3}]
{a, 1}
Shelly
{b, 1}
Bob
{b, 2}
Phil
{b, 3}
Pete
[{a, 1}, {b, 3}]
{a, 1}
Shelly
{b, 1}
Bob
{b, 2}
Phil
{b, 3}
Pete


[{a, 2}, {b, 3}]
{a, 1}
Shelly
{b, 1}
Bob
{b, 3}
Pete
{a, 2}
Anna
[{a, 1}, {b, 4}]
{a, 1}
Shelly
{b, 1}
Bob
{b, 2}
Phil
{b, 3}
Pete
{b, 4}
John


[{a, 2}, {b, 3}]
{a, 1}
Shelly
{b, 1}
Bob
{b, 3}
Pete
{a, 2}
Anna
[{a, 2}, {b, 4}]
{a, 1}
Shelly
{b, 1}
Bob
{b, 3}
Pete
{a, 2}
Anna
{b, 4}
John


[{a, 2}, {b, 4}]
{a, 1}
Shelly
{b, 1}
Bob
{b, 3}
Pete
{a, 2}
Anna
{b, 4}
John
Shelly
Bob
Pete
Anna
John

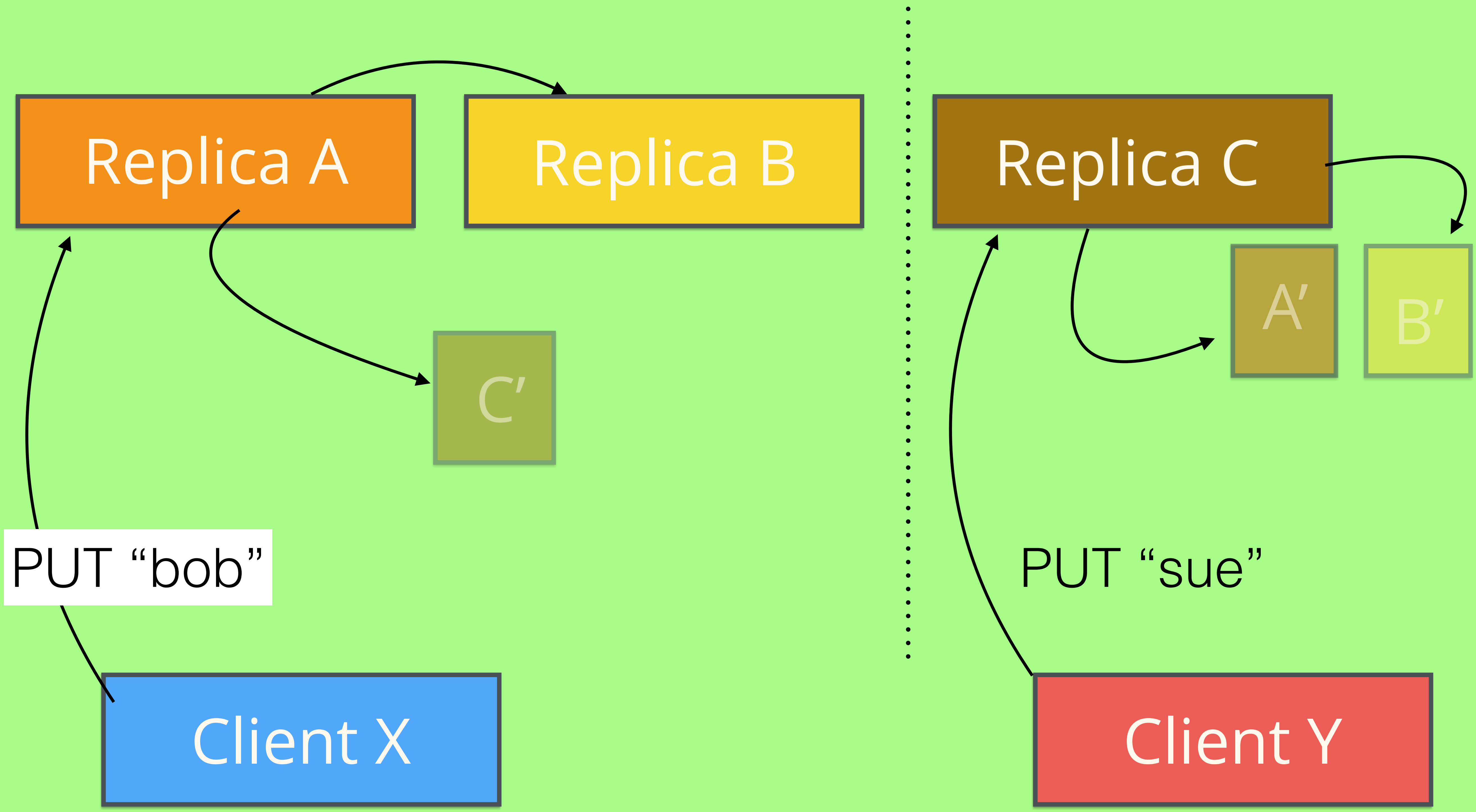
CRDT Sets

a semantic of “Add-Wins”
via
“Observed Remove”

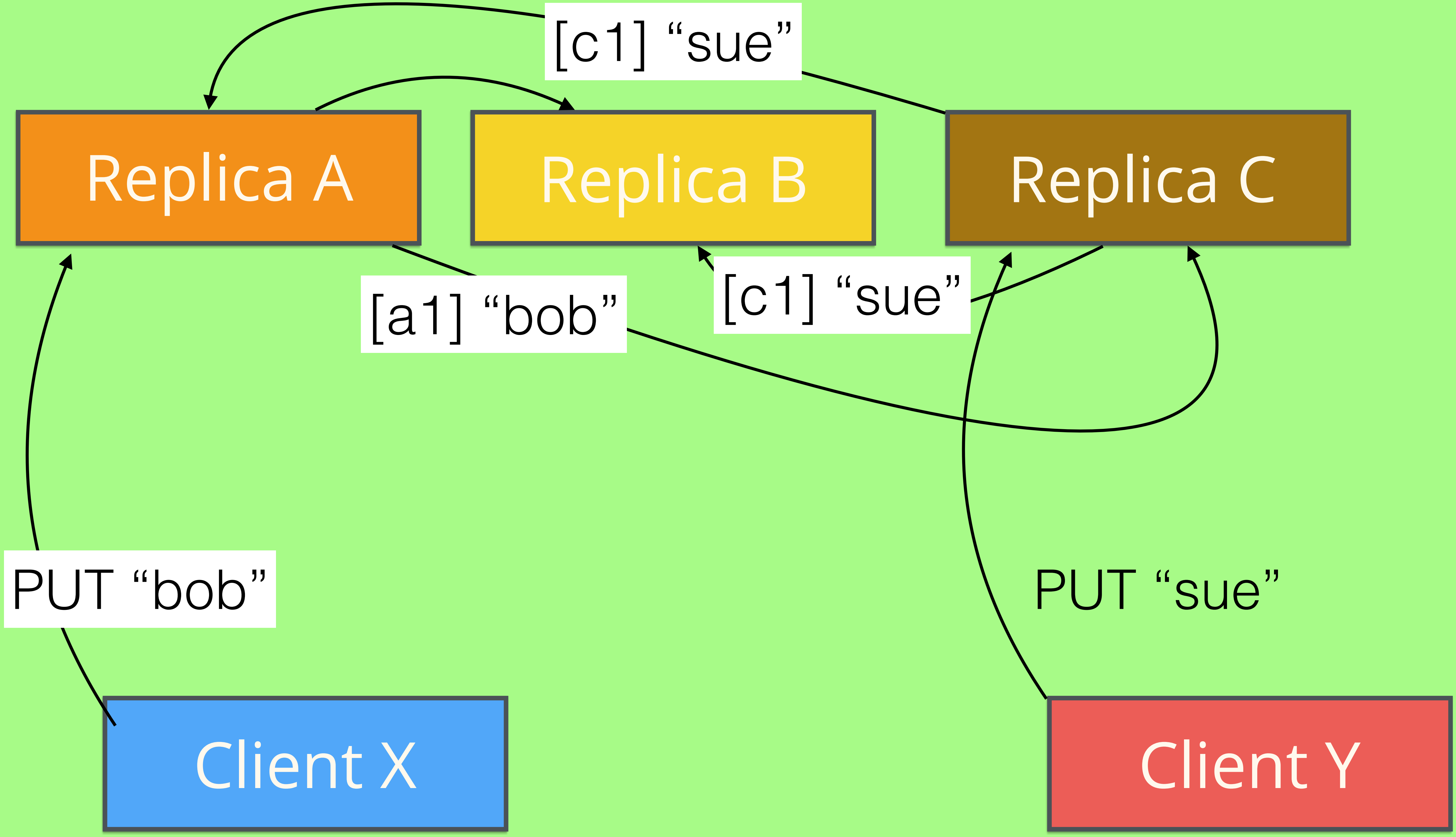
CRDTs

- Principled Merge
- Data Types with Defined Semantic
- Fine Grained Causality
  - minimal representation

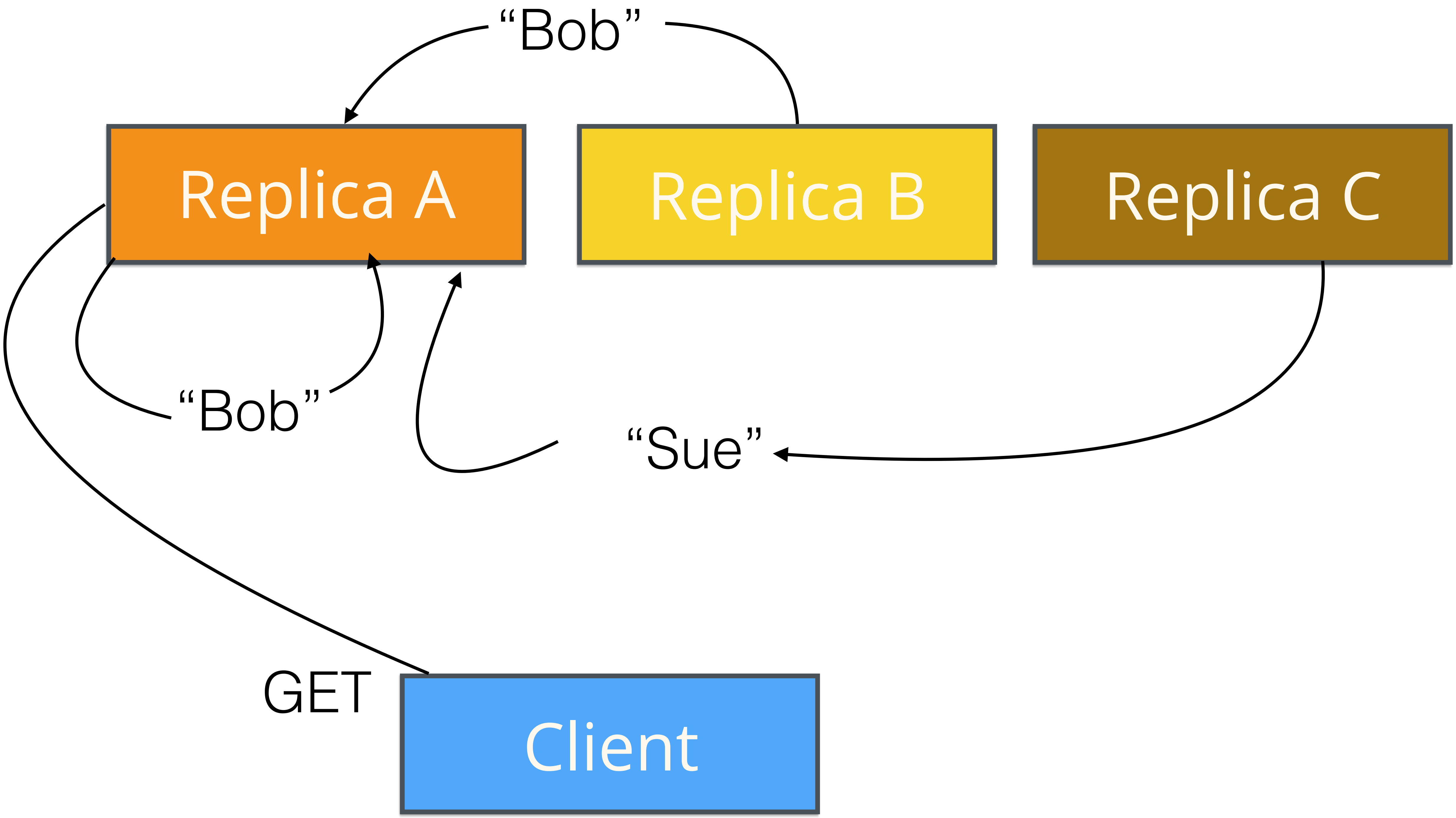
CRDTs IRL


Replica A
Replica B
Replica C
C′
A′
B′
PUT “bob”
PUT “sue”
Client X
Client Y

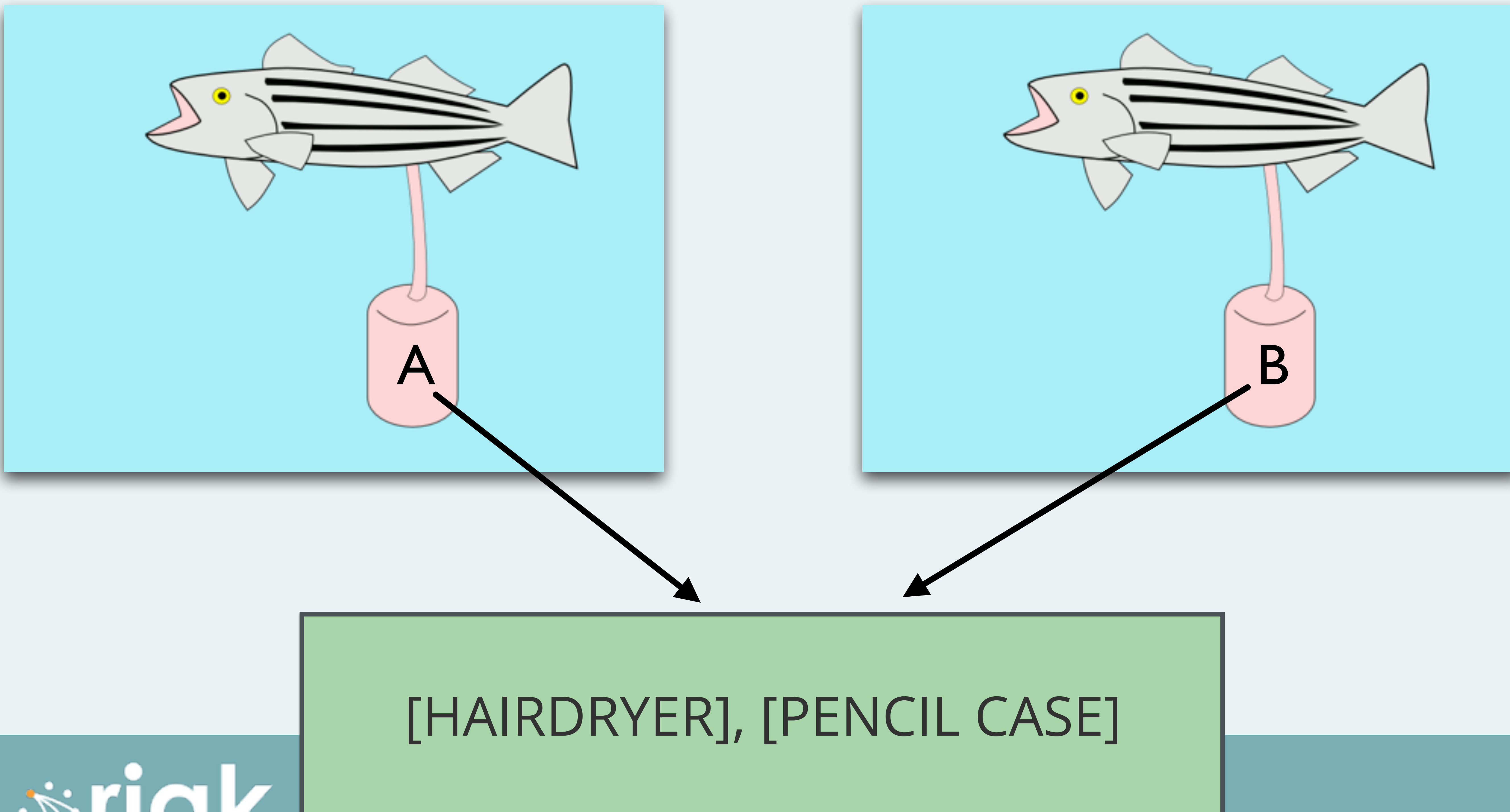
Available


[c1] "sue"
Replica A
Replica B
Replica C
[a1] "bob"
[c1] "sue"
PUT "bob"
PUT "sue"
Client X
Client Y

Low Latency



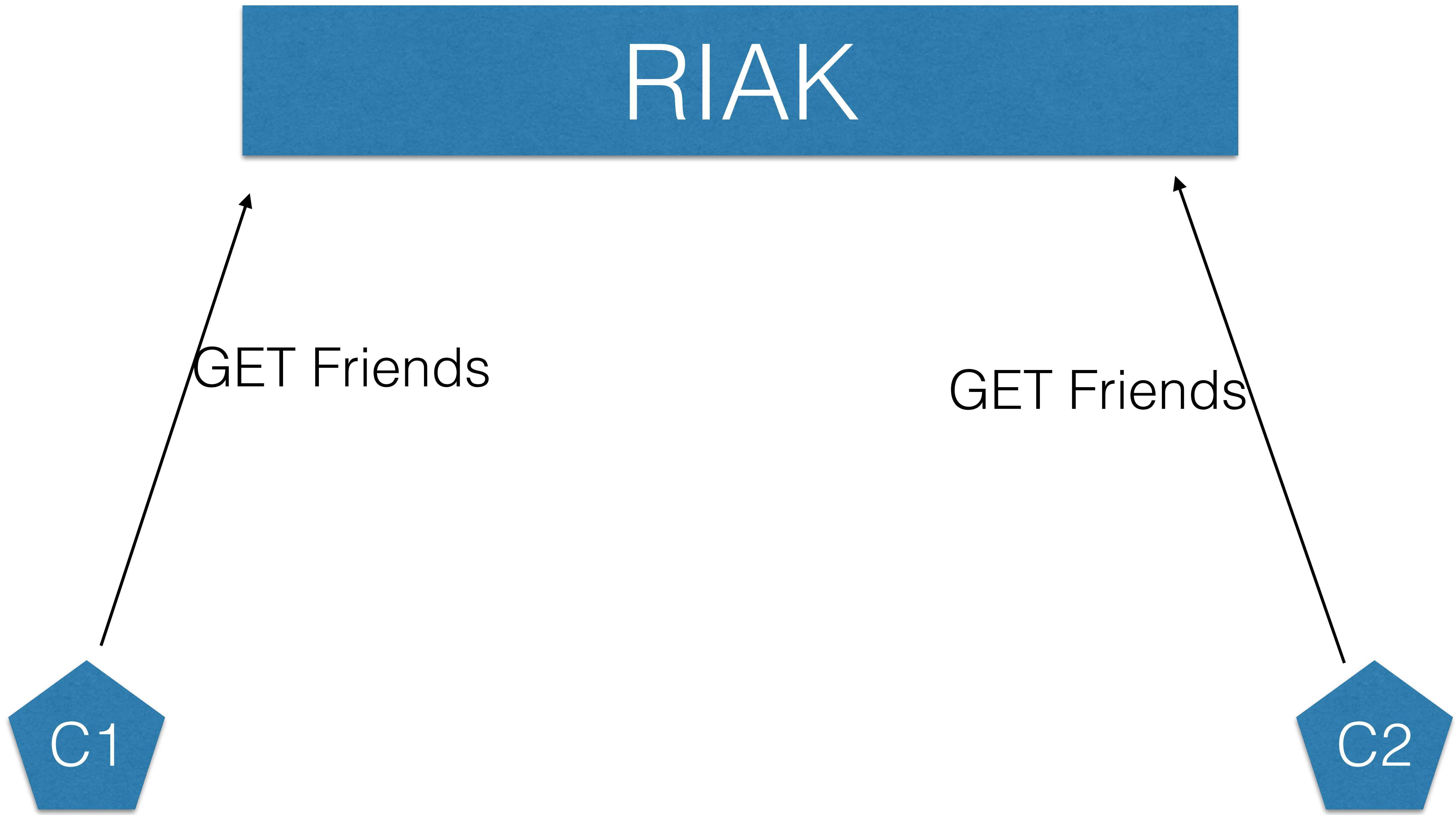
Conflict!


A
B
[HAIRDRYER], [PENCIL CASE]

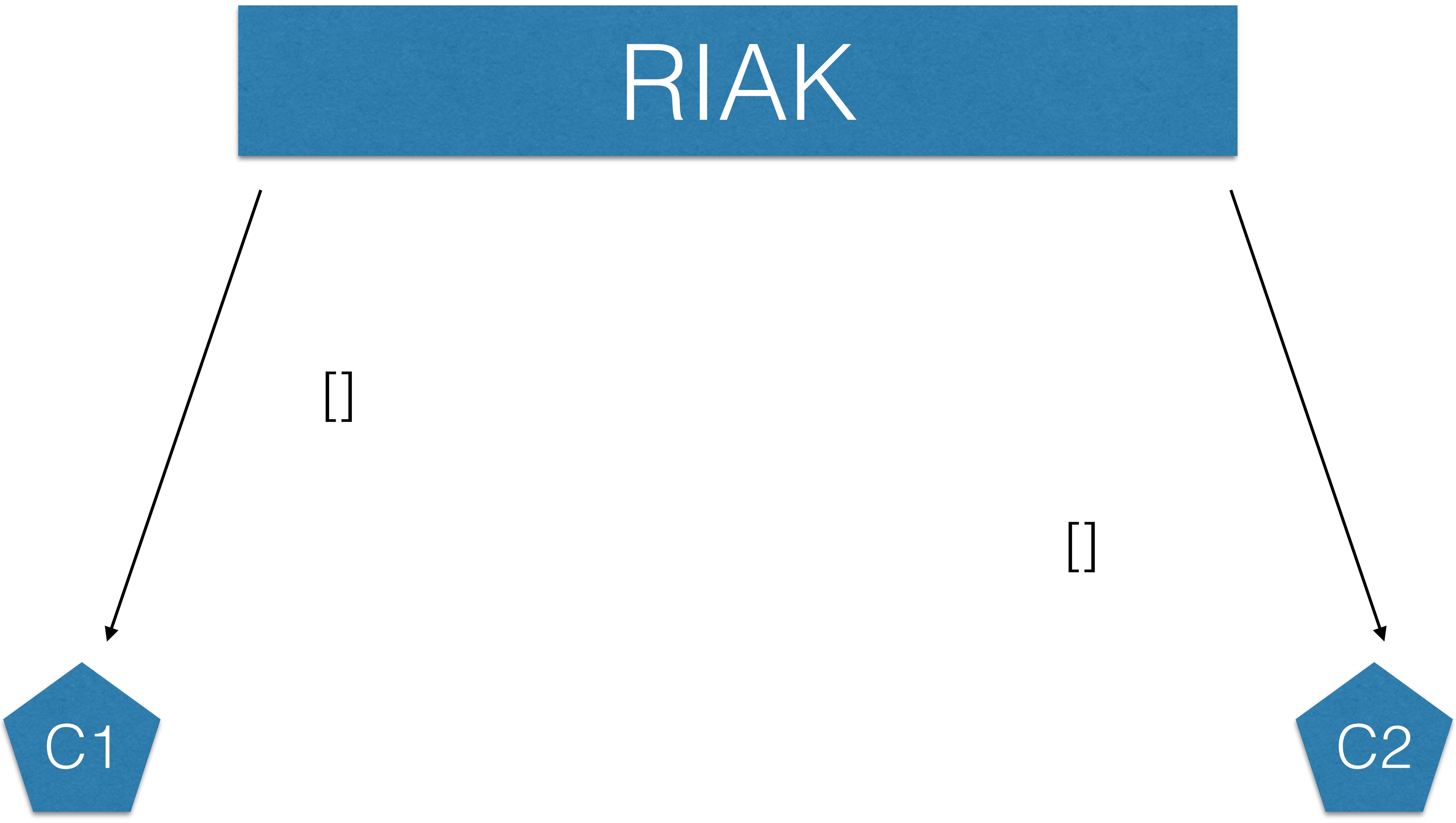
{"key": "value"}

riak

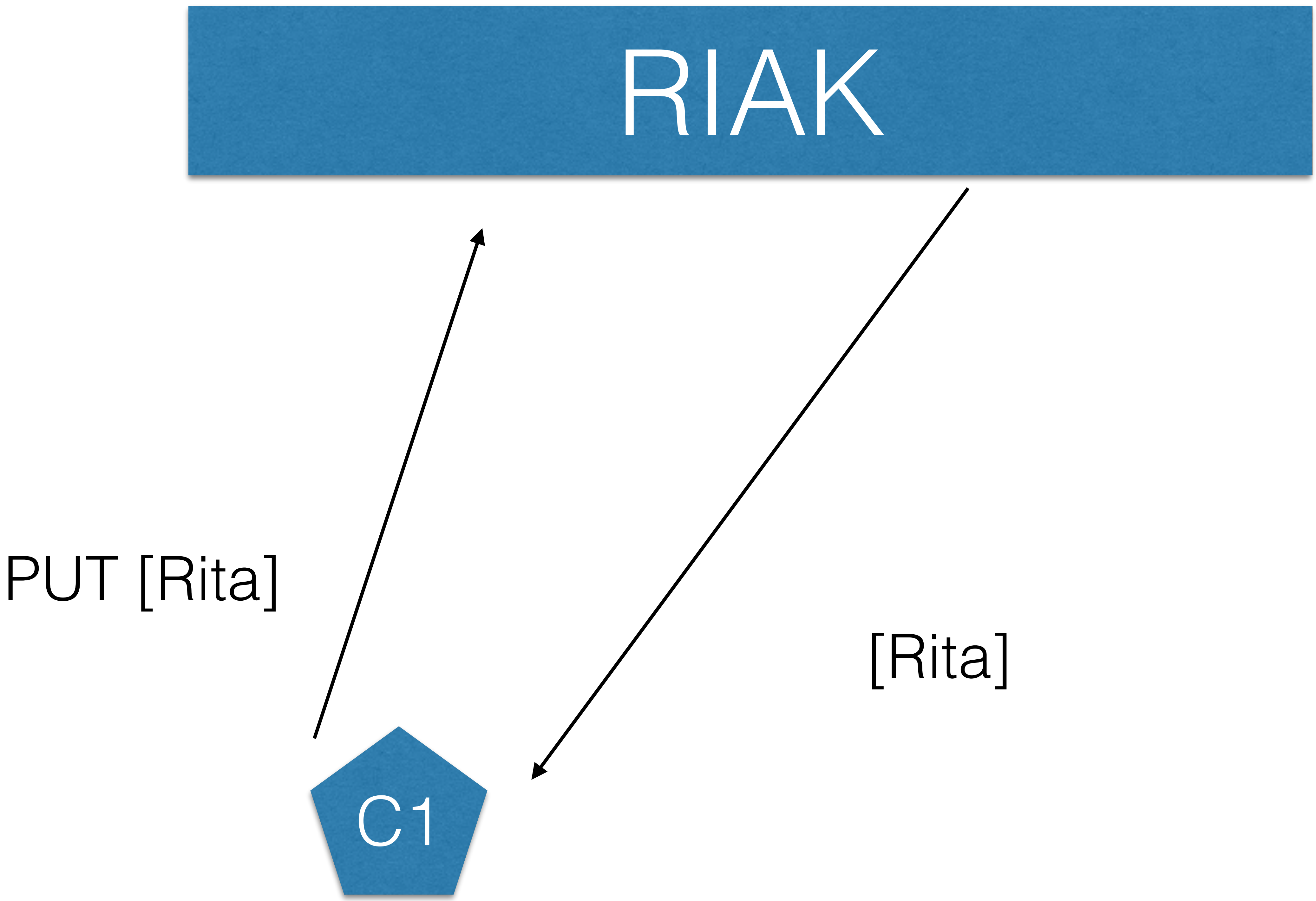
Sets



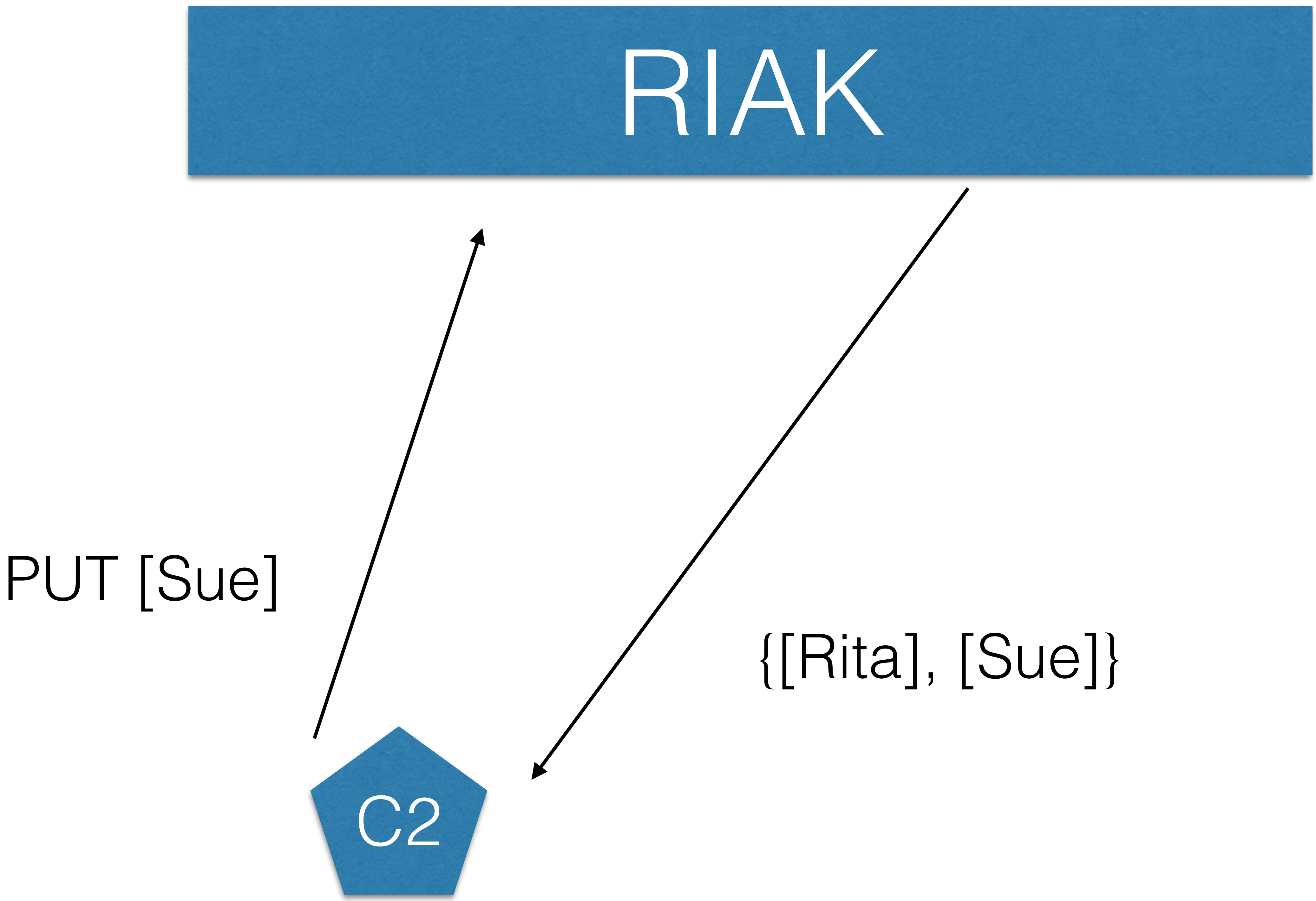
State To Client State



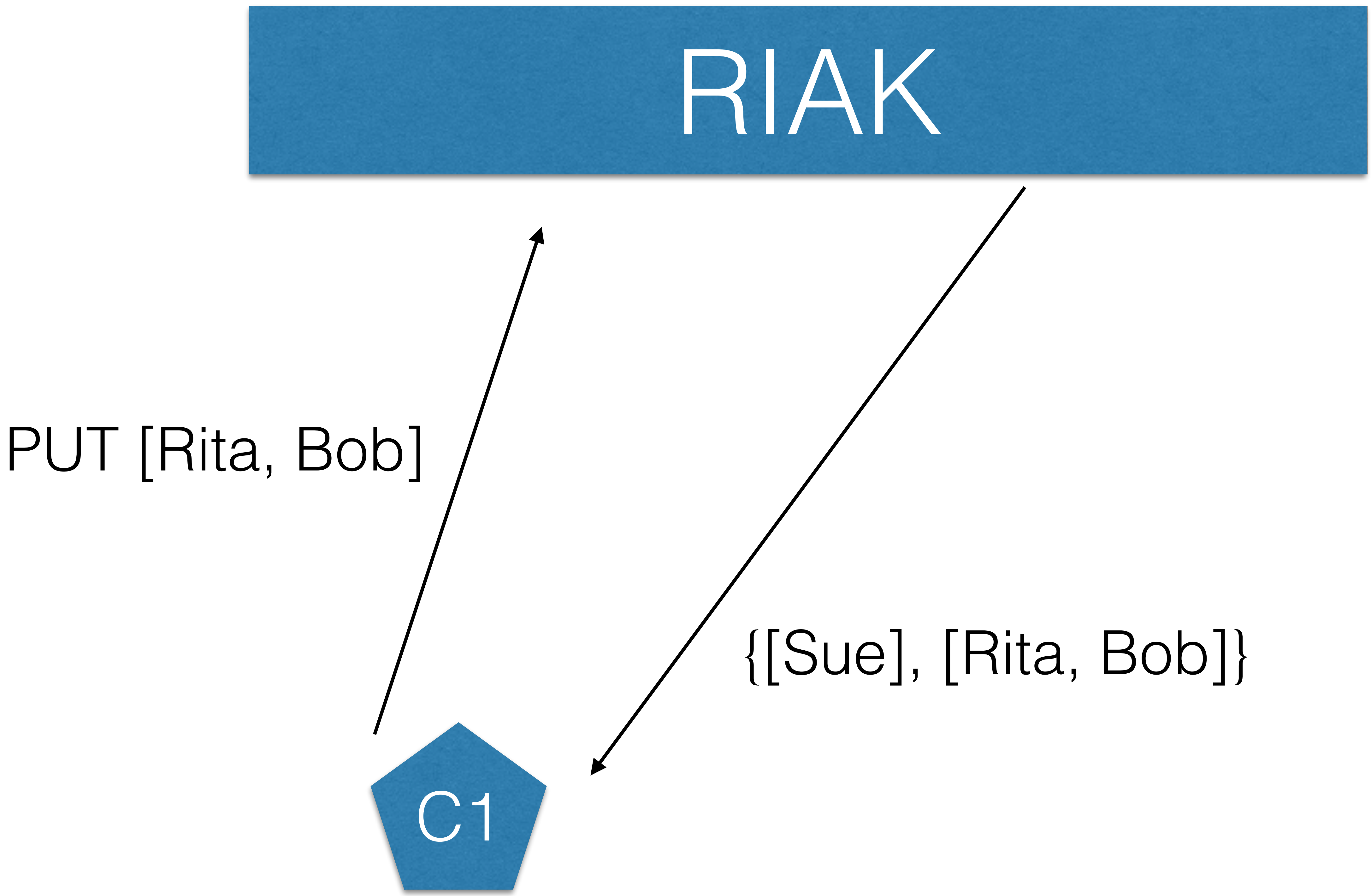
State To Client



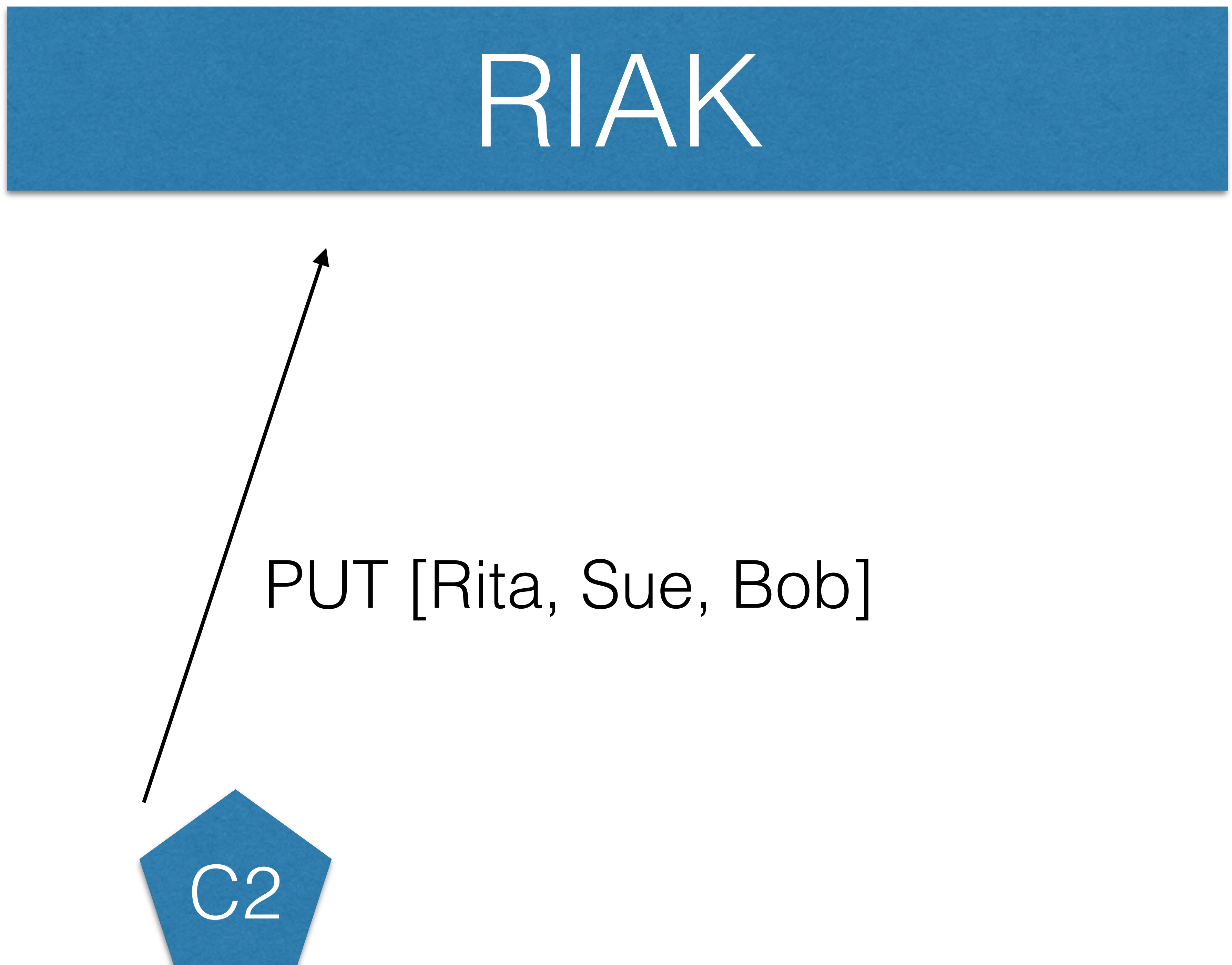
State To Client



State To Client



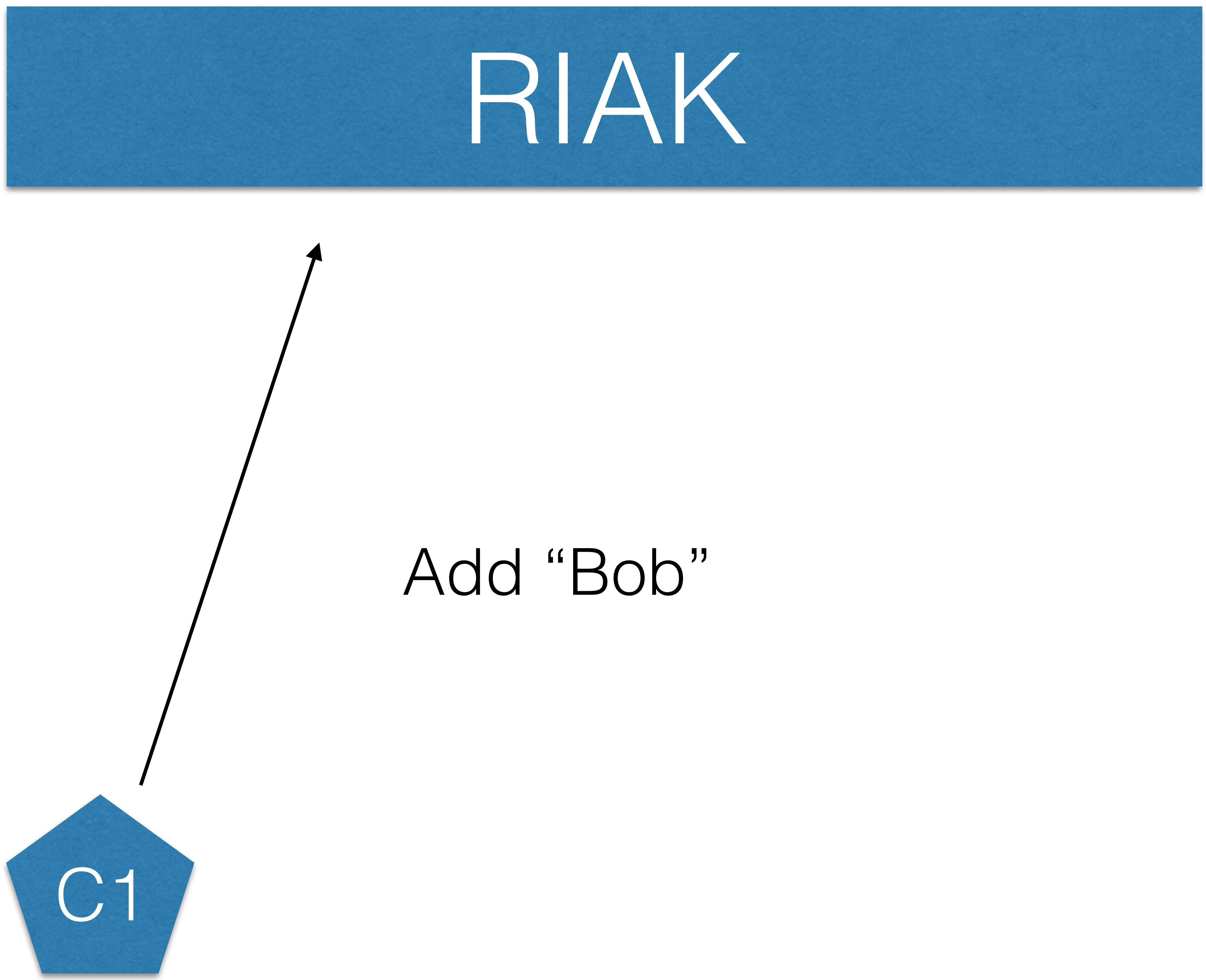
State To Client



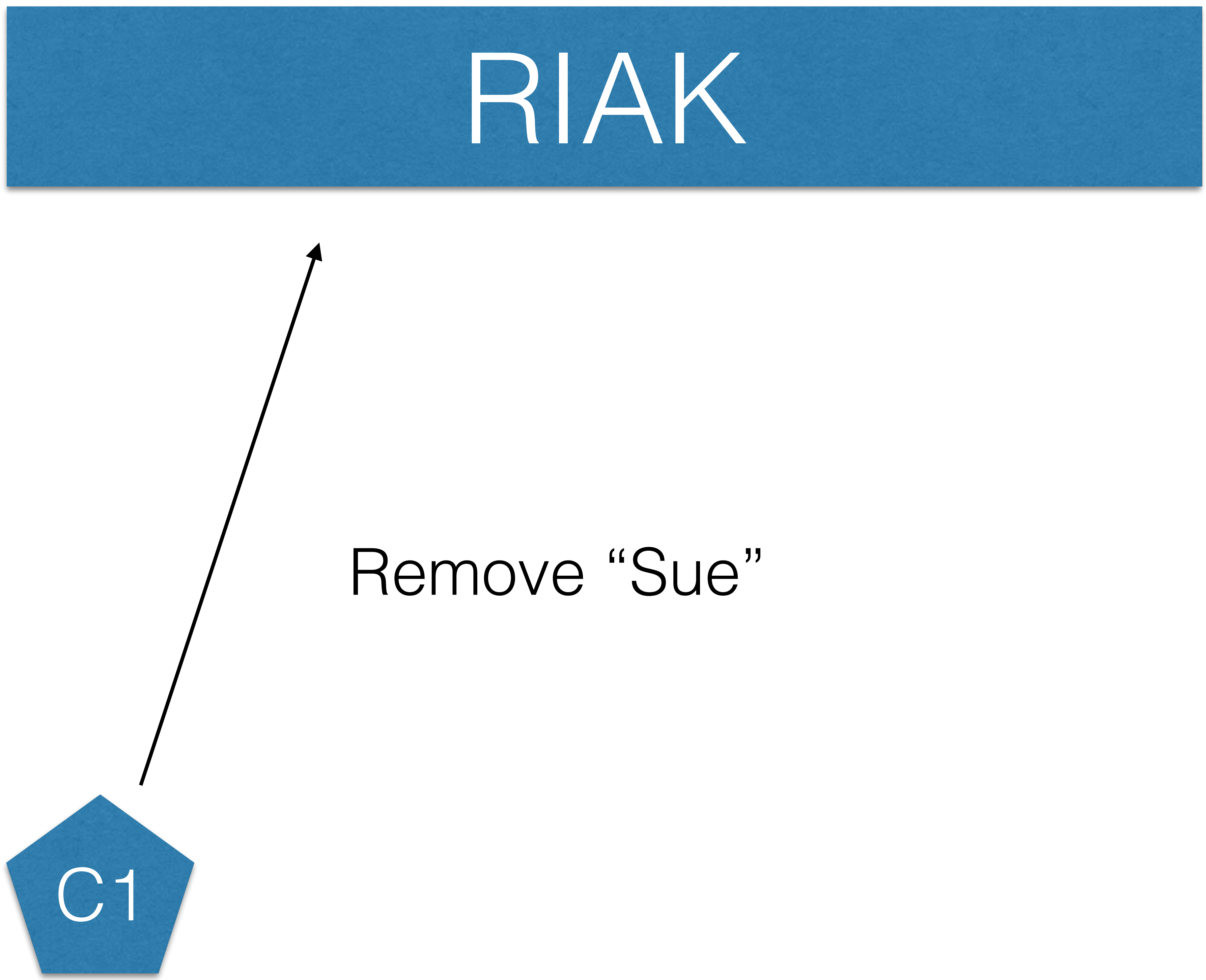
Problem?

- Requires Read Your Own Writes consistency
- Client must manage Actors in set's logical clock
  - Client ensures invariants
    - Serial actor, total order of events
- Read and Send all Data to Add/Remove an Element??

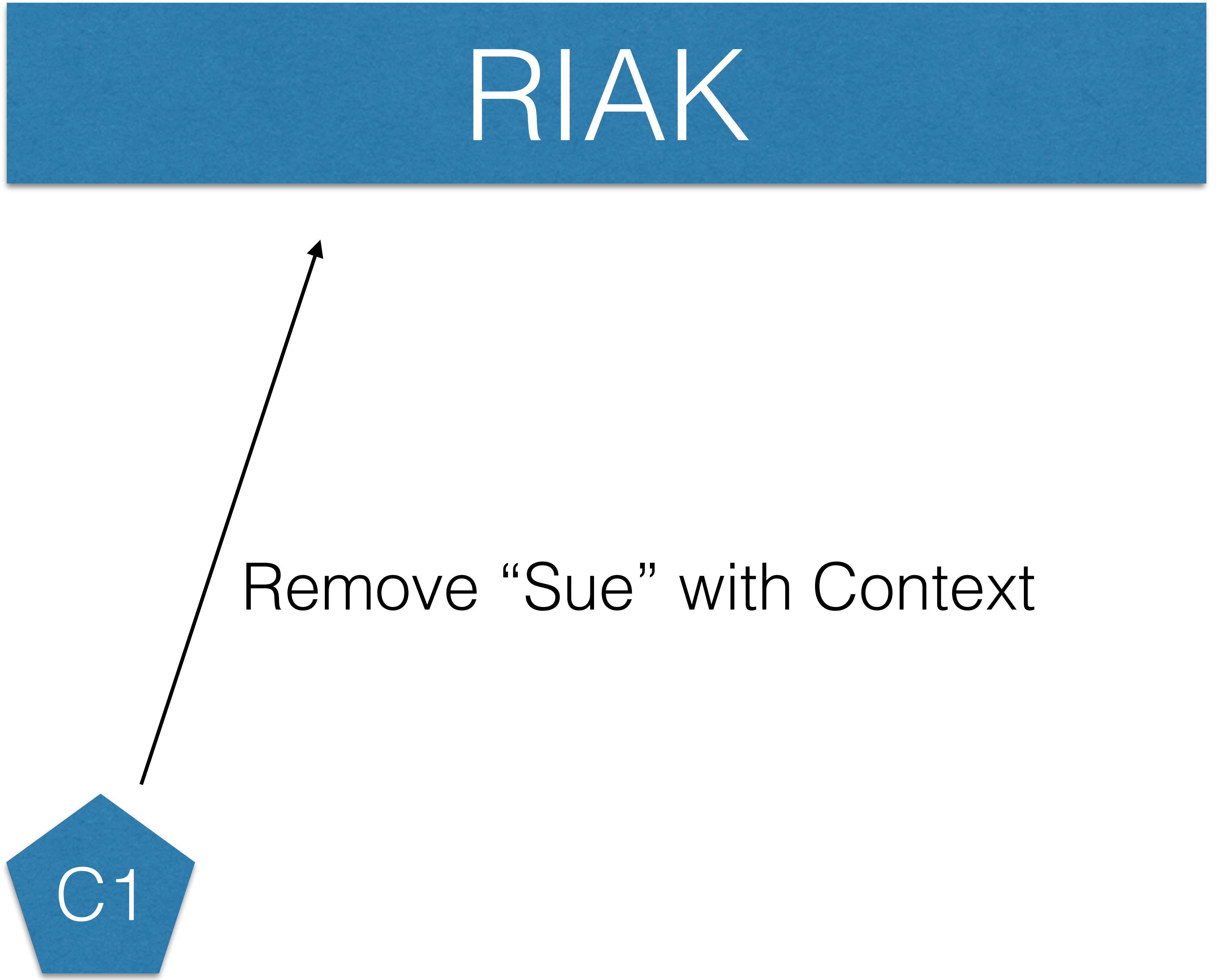
Operations!



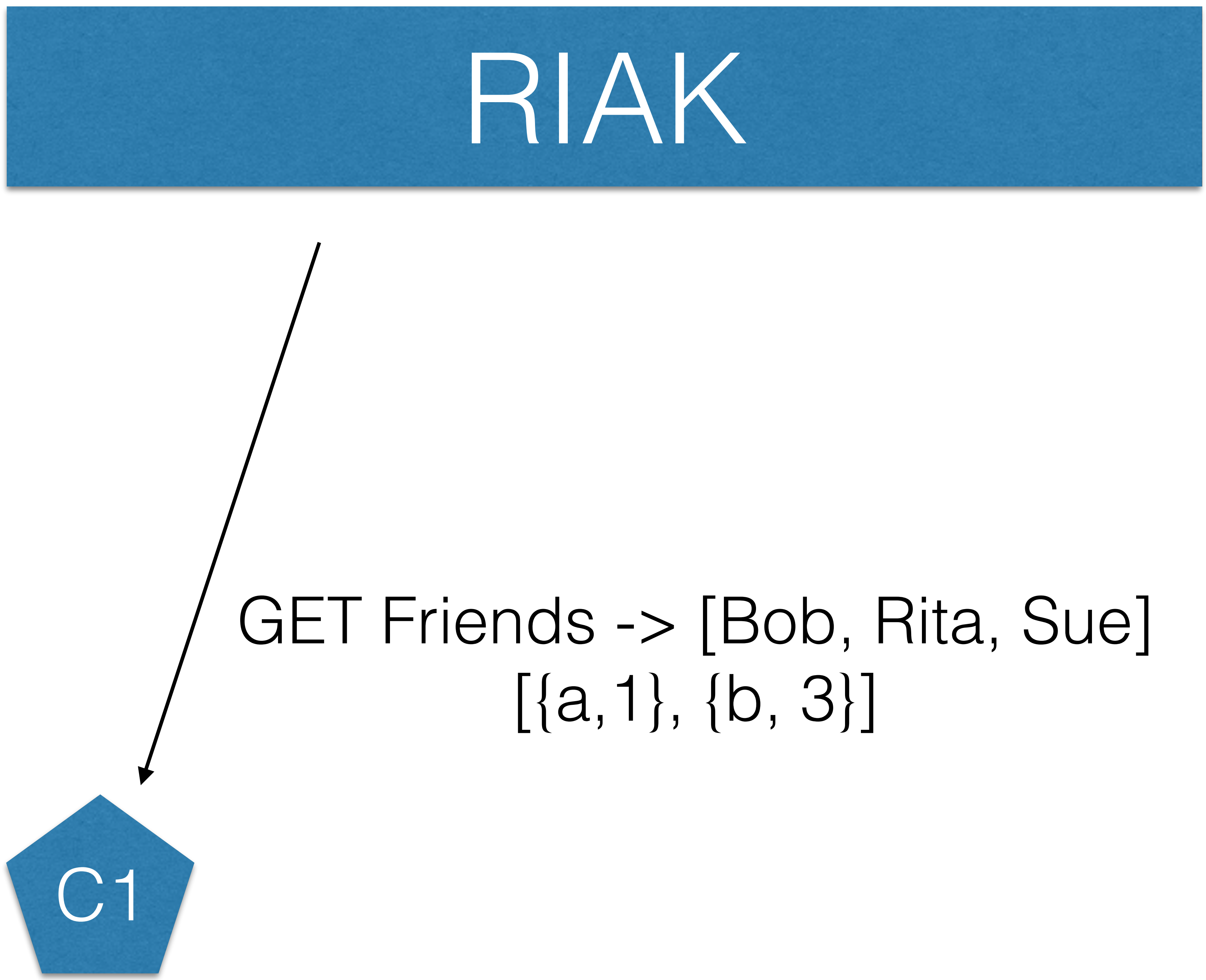
Operations!



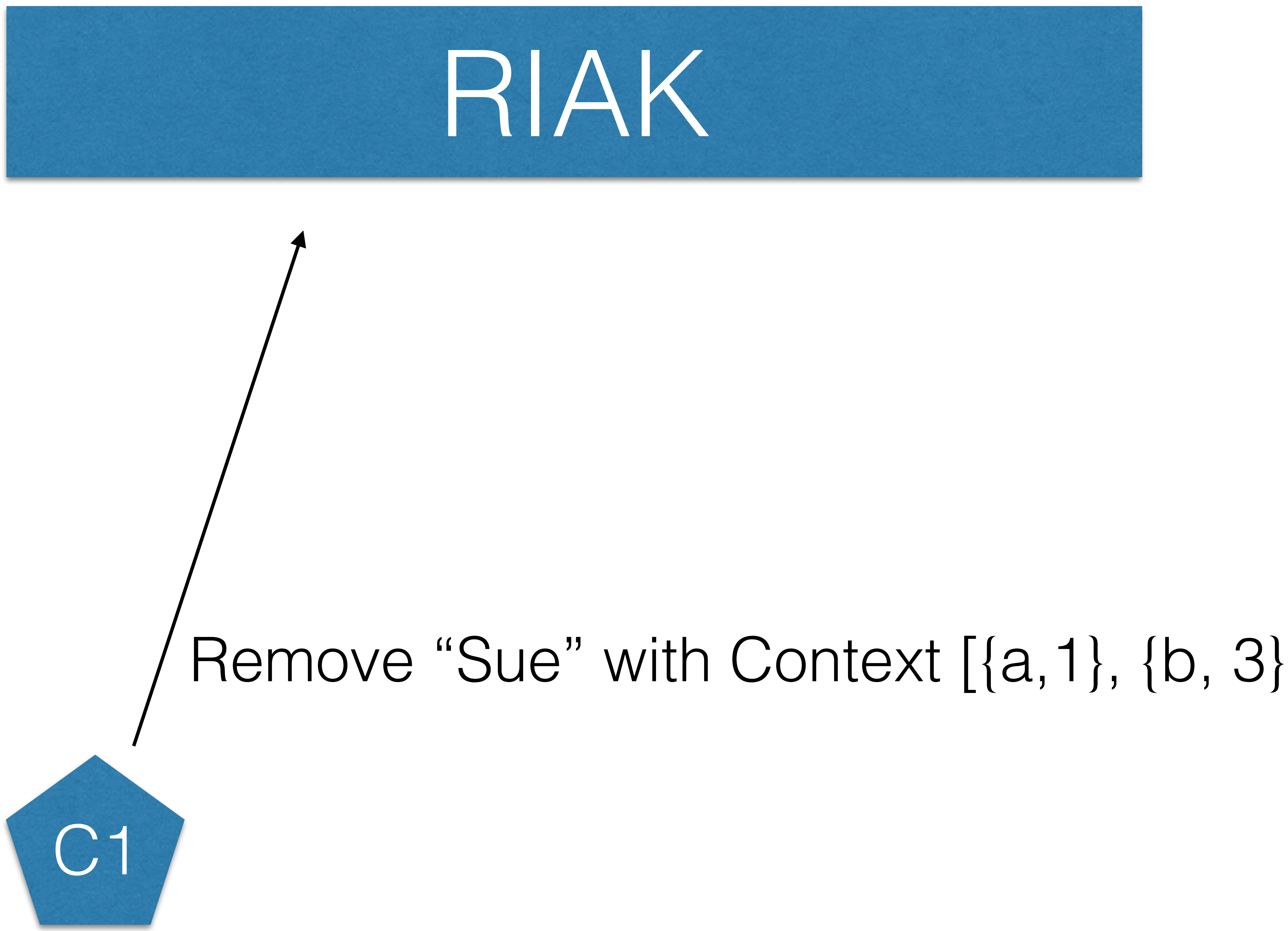
Operations!
Observed Remove



Operations!

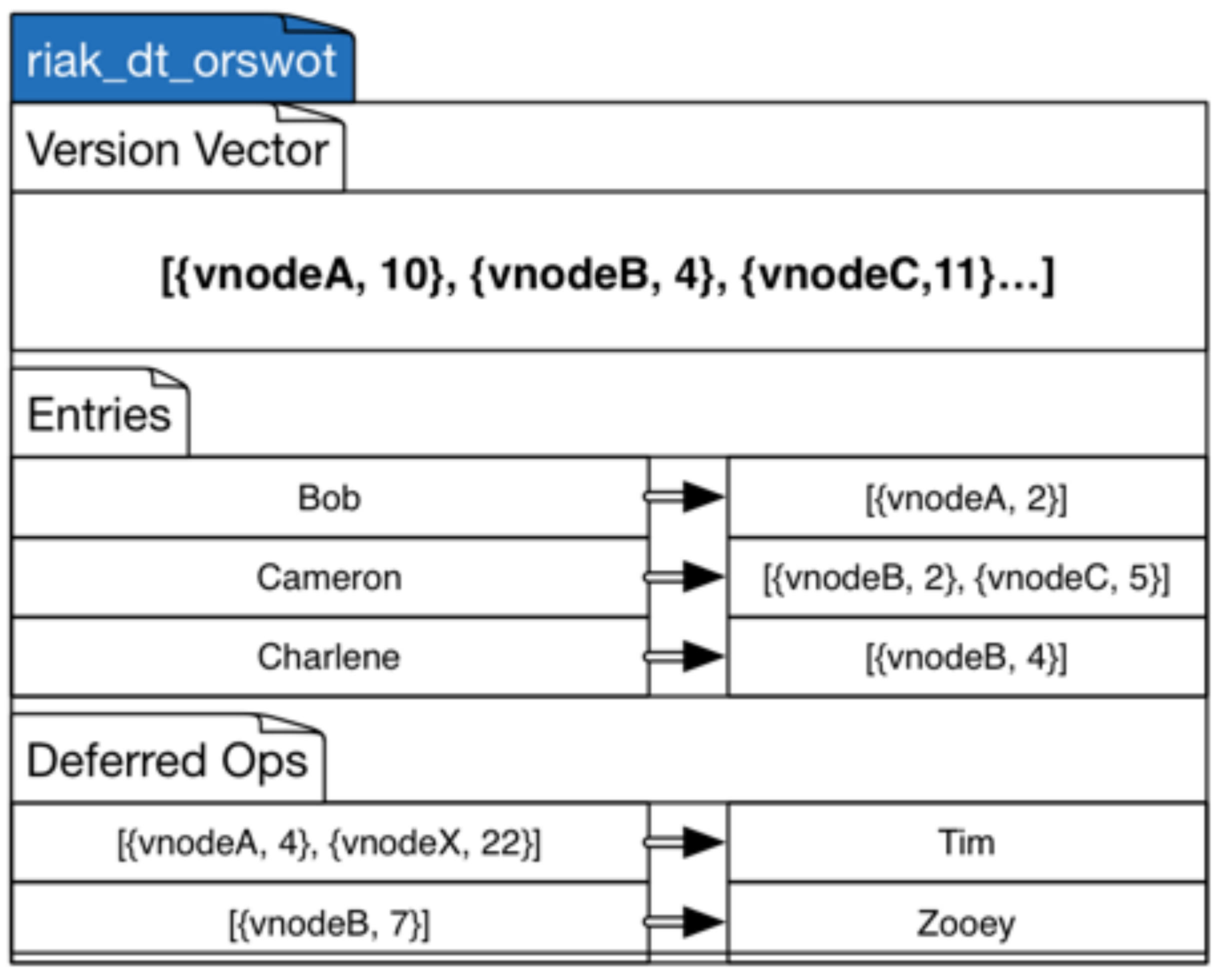


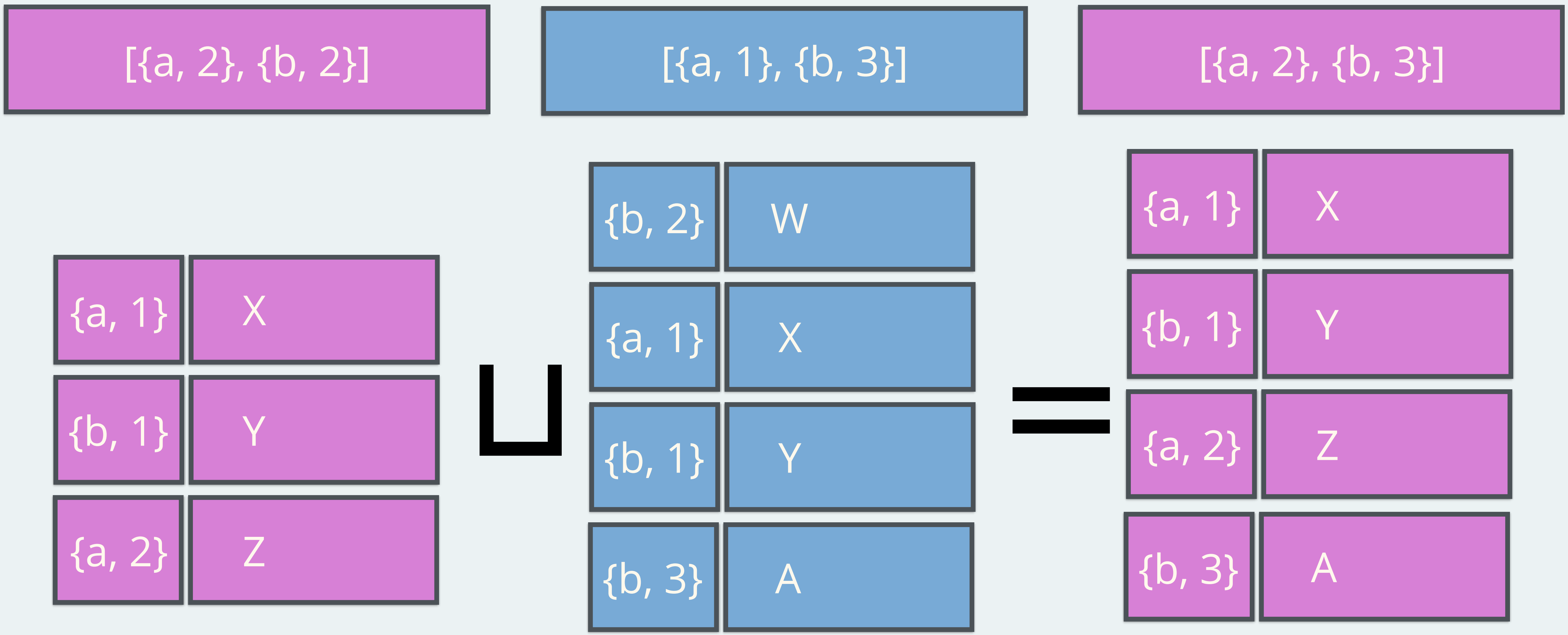
Operations!
Observed-Remove



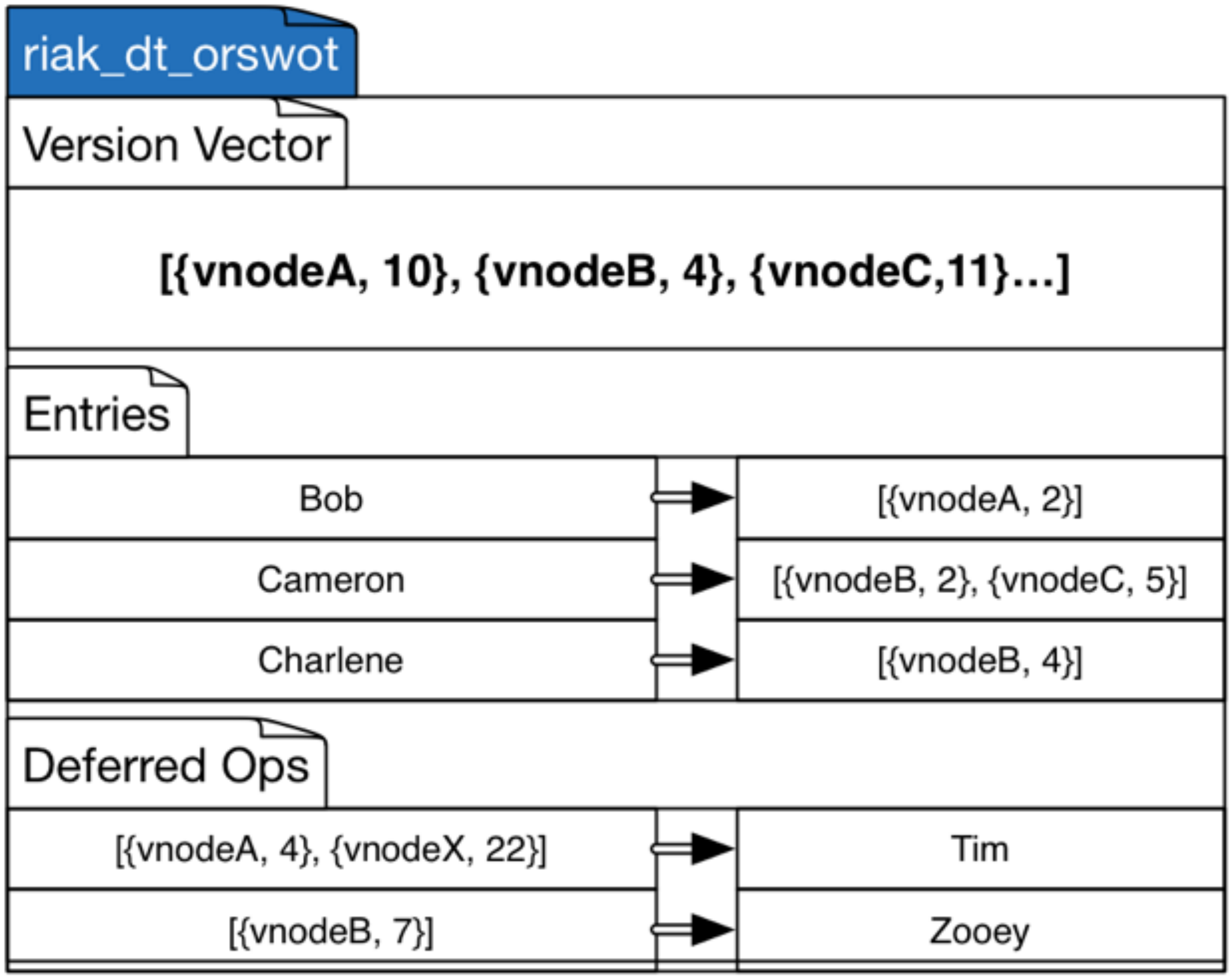
Riak 2.0

Riak Data Types
Riak_DT
CRDTs



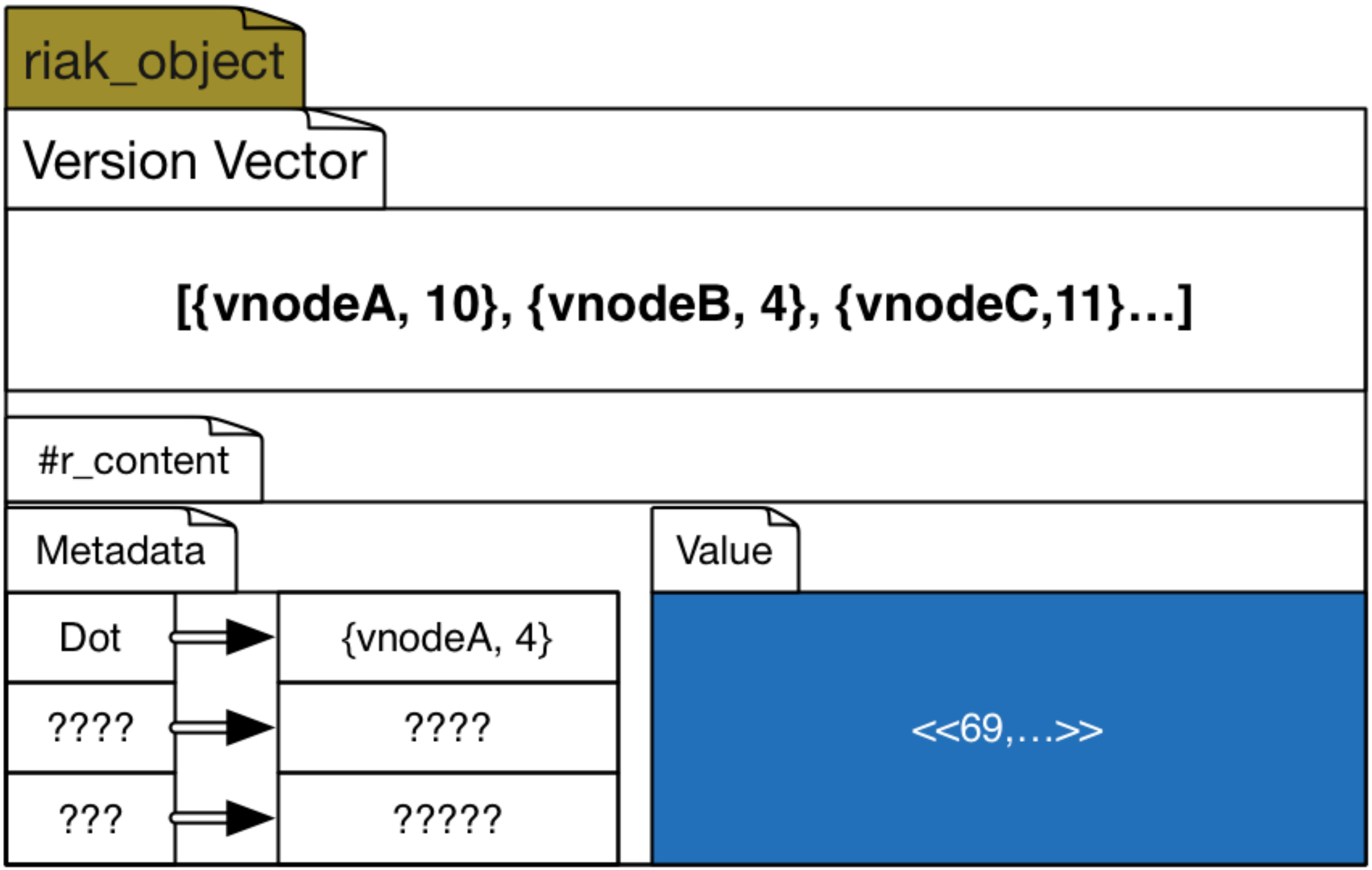

[{a, 2}, {b, 2}]
{a, 1} X
{b, 1} Y
{a, 2} Z
⊔
[{a, 1}, {b, 3}]
{b, 2} W
{a, 1} X
{b, 1} Y
{b, 3} A
=
[{a, 2}, {b, 3}]
{a, 1} X
{b, 1} Y
{a, 2} Z
{b, 3} A

Sets in Riak

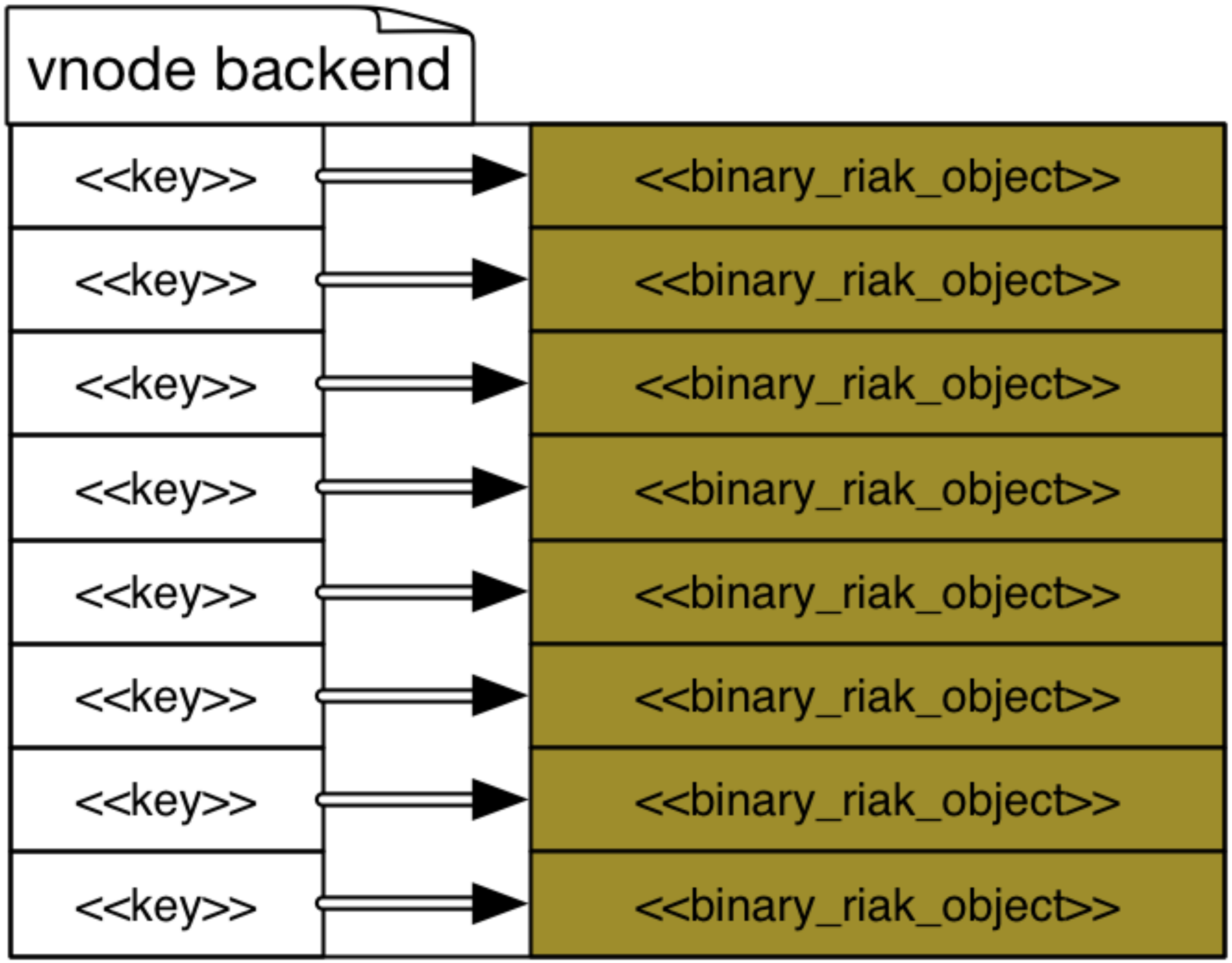


An optimized conflict-free replicated set
Annette Bieniusa et al
http://arxiv.org/abs/1210.3368

Sets in Riak


riak_object
Version Vector
[{vnodeA, 10}, {vnodeB, 4}, {vnodeC,11}…]
#r_content
Metadata
Value
Dot
{vnodeA, 4}
????
????
???
?????
<<69,…>>

Sets in Riak


vnode backend
<<key>>
<<binary_riak_object>>
<<key>>
<<binary_riak_object>>
<<key>>
<<binary_riak_object>>
<<key>>
<<binary_riak_object>>
<<key>>
<<binary_riak_object>>
<<key>>
<<binary_riak_object>>
<<key>>
<<binary_riak_object>>
<<key>>
<<binary_riak_object>>

Sets in Riak

PHOTO © 2011 J. RONALD LEE, CC ATTRIBUTION 3.0.
https://www.flickr.com/photos/jronaldlee/5566380424

Teach Riak about CRDTs

- API Boundary
- Syntactic merge riak_object:merge
  - Version Vector merge and sibling storage
  - CRDT == no siblings

Problem?

Use Case

- bet365 million pound customer
- Use CRDT sets for open bet tracking
- Partition Riak Sets
    - Performance - write speed
    - Size - cardinality

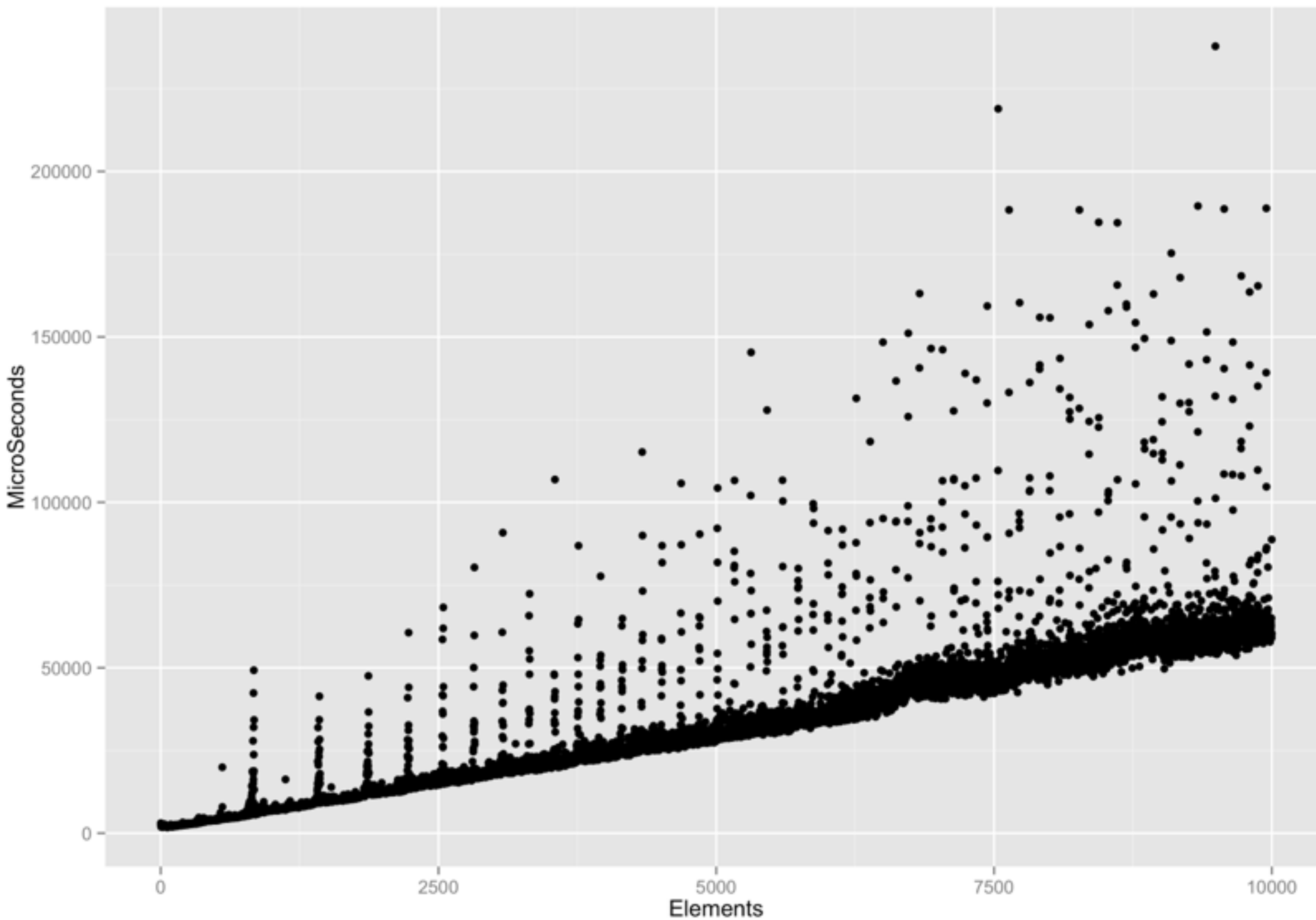
Use Case

- NHS England
- Use CRDT sets for mailbox app
- Truncate/archive older messages
    - Performance - write speed
    - Size - cardinality

Problem?

- Poor Write speed
- Can’t have “big” sets

Sets in Riak



200000
150000
100000
50000
0
MicroSeconds
0
2500
5000
7500
10000
Elements

10k sets, 100k elements, 50 workers - write

Sets in Riak

- read at replica
  - deserialise
  - mutate
  - serialise
  - write

Sets In Riak

- replicate FULL STATE
  - (read, deserialise? merge? serialise?, write?)
- ? riak_object.vv
  - Accidental Optimisation

Every time we change the set we read and write the whole set!

Delta-Sets

- Only replicate the Delta - the change
  - The delta is element + causal tag
  - Can be “Joined” like full state
    - Idempotent/Associate/Commutative
- Efficient State-based CRDTs by Delta-Mutation - Paulo Sérgio Almeida et al

Delta-Sets in Riak

- Still read whole set to generate delta
- Still read whole set to merge delta - in fact MUST
  - (read, deserialise! merge! serialise! write!)
    - Database - disk i/o is THE thing
- Delta is always concurrent/sibling
  - Save on the network, pay on the disk

Sets in Riak

Small : riak object

1MB limit

Disconnect

- Paper - minimal model to express innovation
    - A set Actors, each a replica
    - A single CRDT in memory
    - Reads are R=1

Disconnect

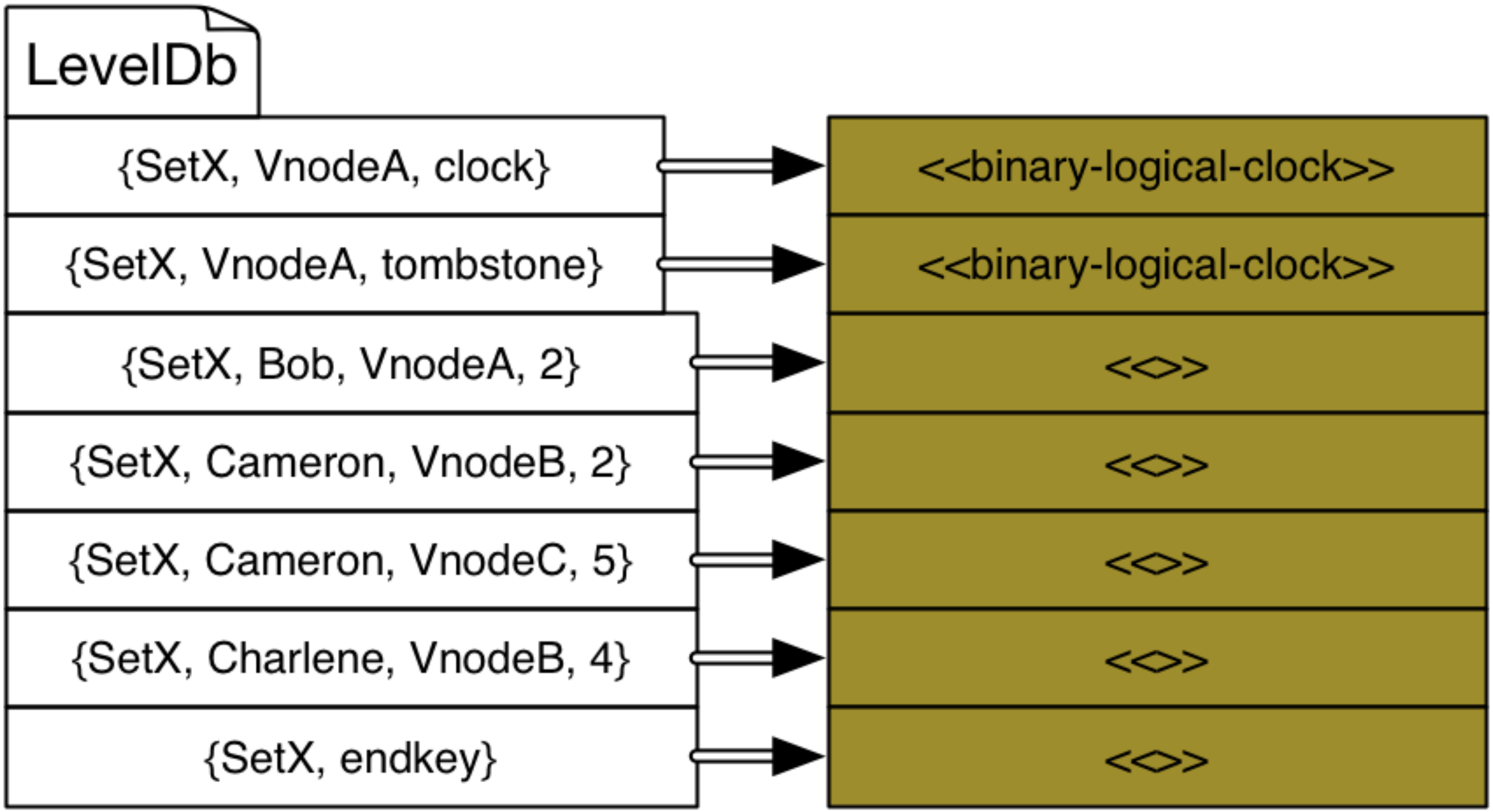
- Riak - A real world industrial database product
  - Many Keys, many CRDTs
  - Durably stored on Disk - serialisation
  - Clients act remotely on State
  - One Key, One Set O(n)

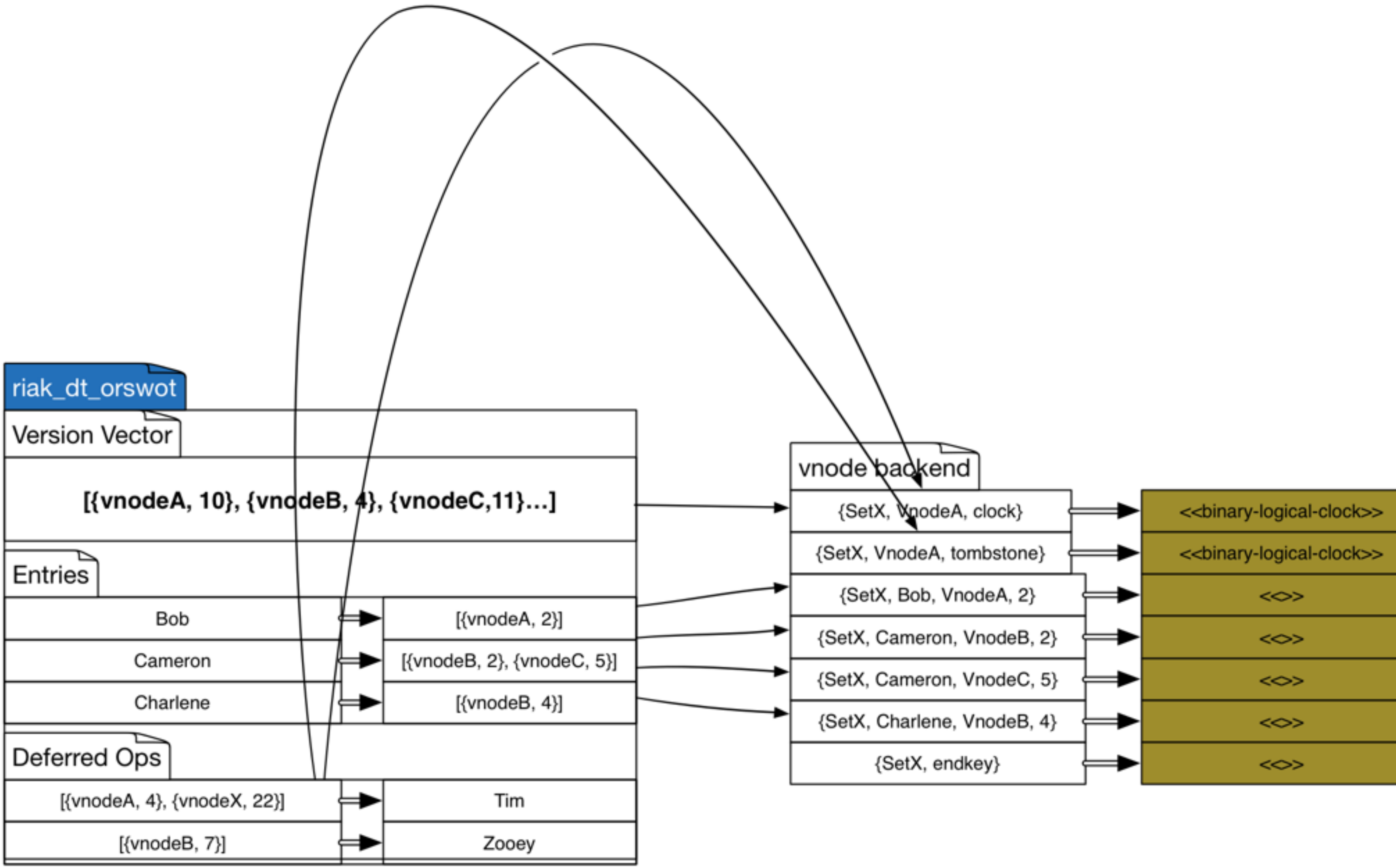
Problem Summary

- Join is expensive
- Serialisation/Deserialisation dance wasteful
- Disk i/o matters to a database!

Bigsets:
Make writes faster
and
sets bigger

Bigset Design: Overview

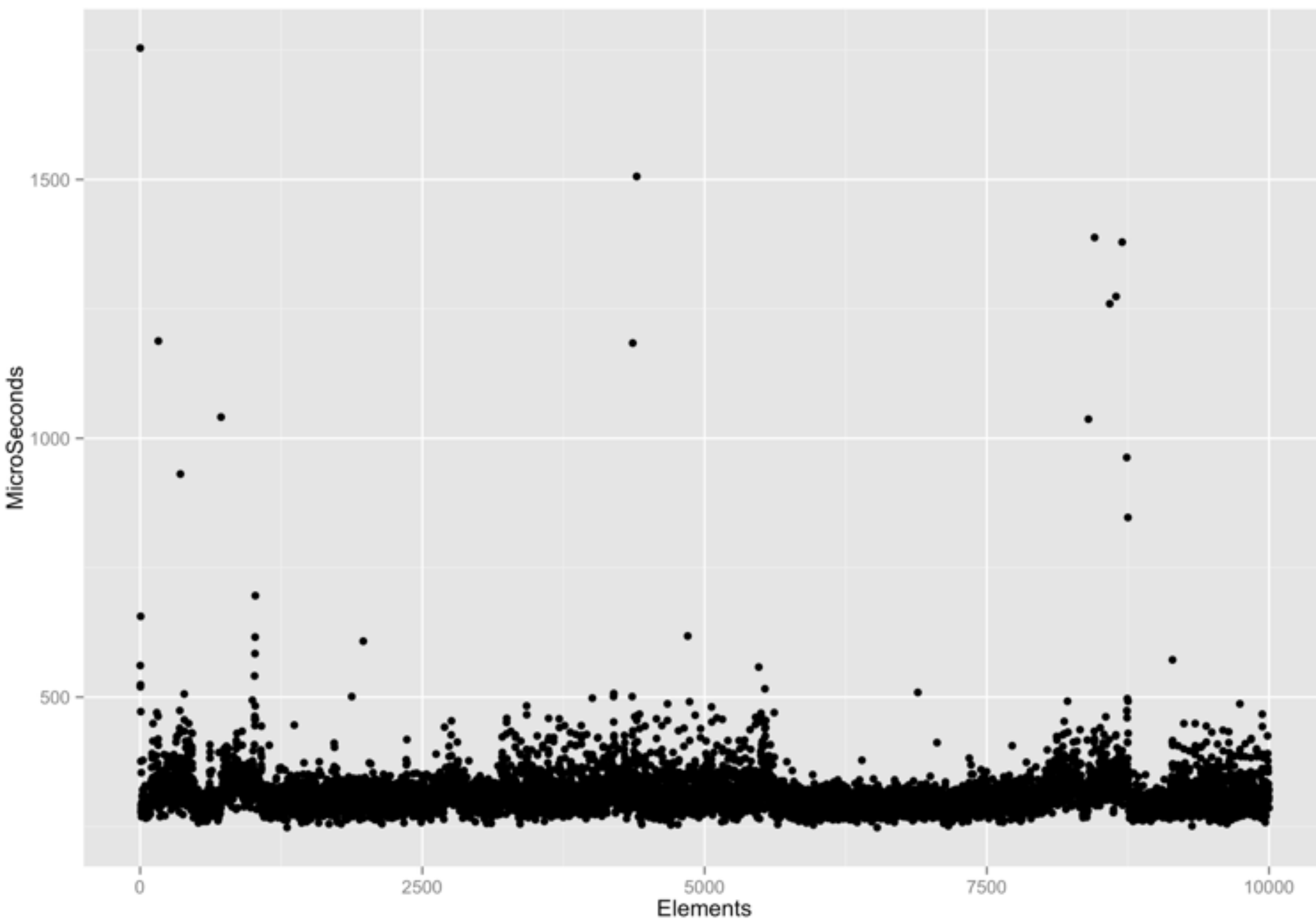


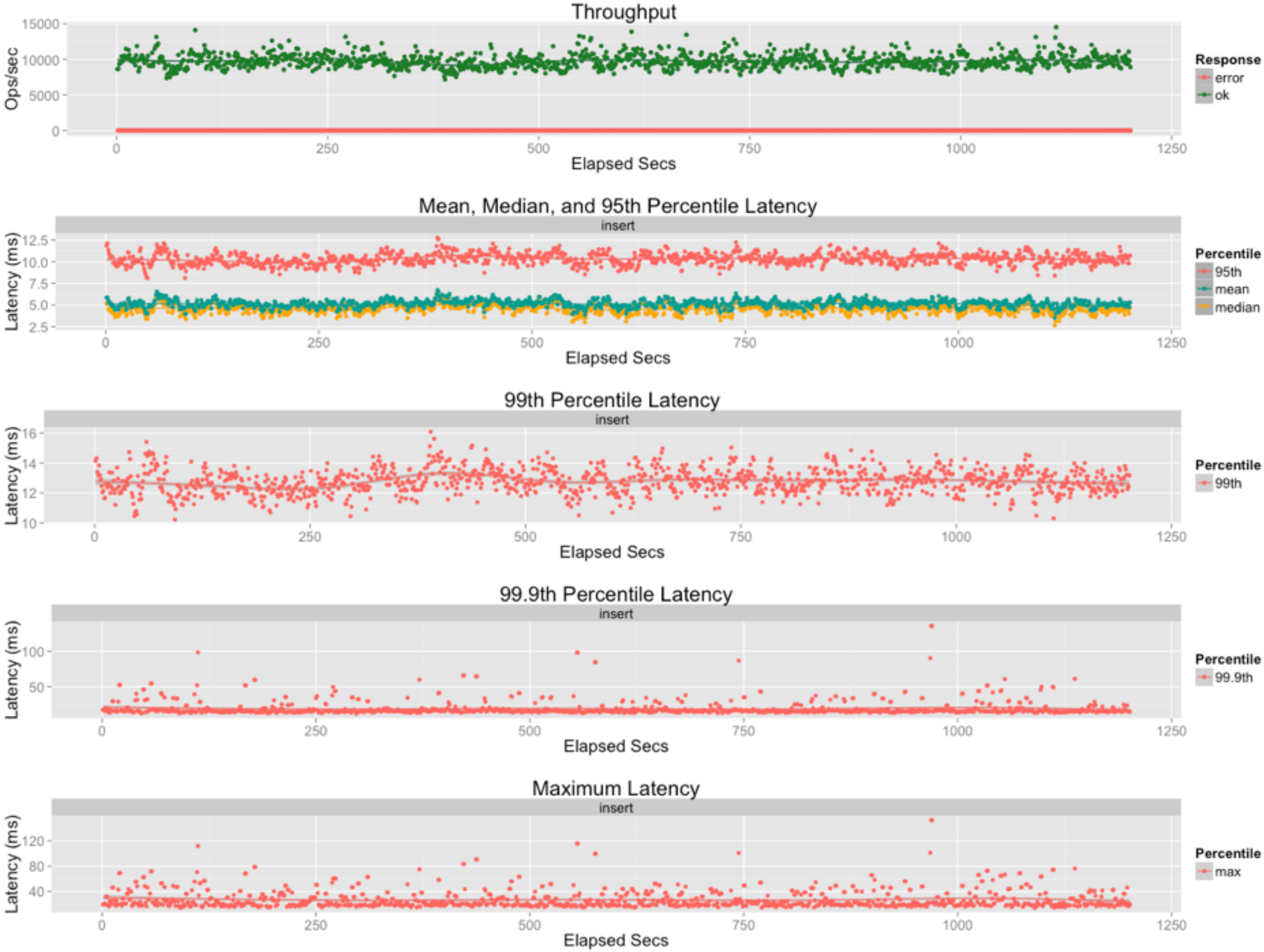

riak_dt_orswot
Version Vector
[{vnodeA, 10}, {vnodeB, 4}, {vnodeC,11}…]
Entries
Bob
[{vnodeA, 2}]
Cameron
[{vnodeB, 2}, {vnodeC, 5}]
Charlene
[{vnodeB, 4}]
Deferred Ops
[{vnodeA, 4}, {vnodeX, 22}]
Tim
[{vnodeB, 7}]
Zooey
vnode backend
{SetX, VnodeA, clock}
{SetX, VnodeA, tombstone}
{SetX, Bob, VnodeA, 2}
{SetX, Cameron, VnodeB, 2}
{SetX, Cameron, VnodeC, 5}
{SetX, Charlene, VnodeB, 4}
{SetX, endkey}
<<binary-logical-clock>>
<<binary-logical-clock>>
<<>>
<<>>
<<>>
<<>>
<<>>

Bigset Design: Overview

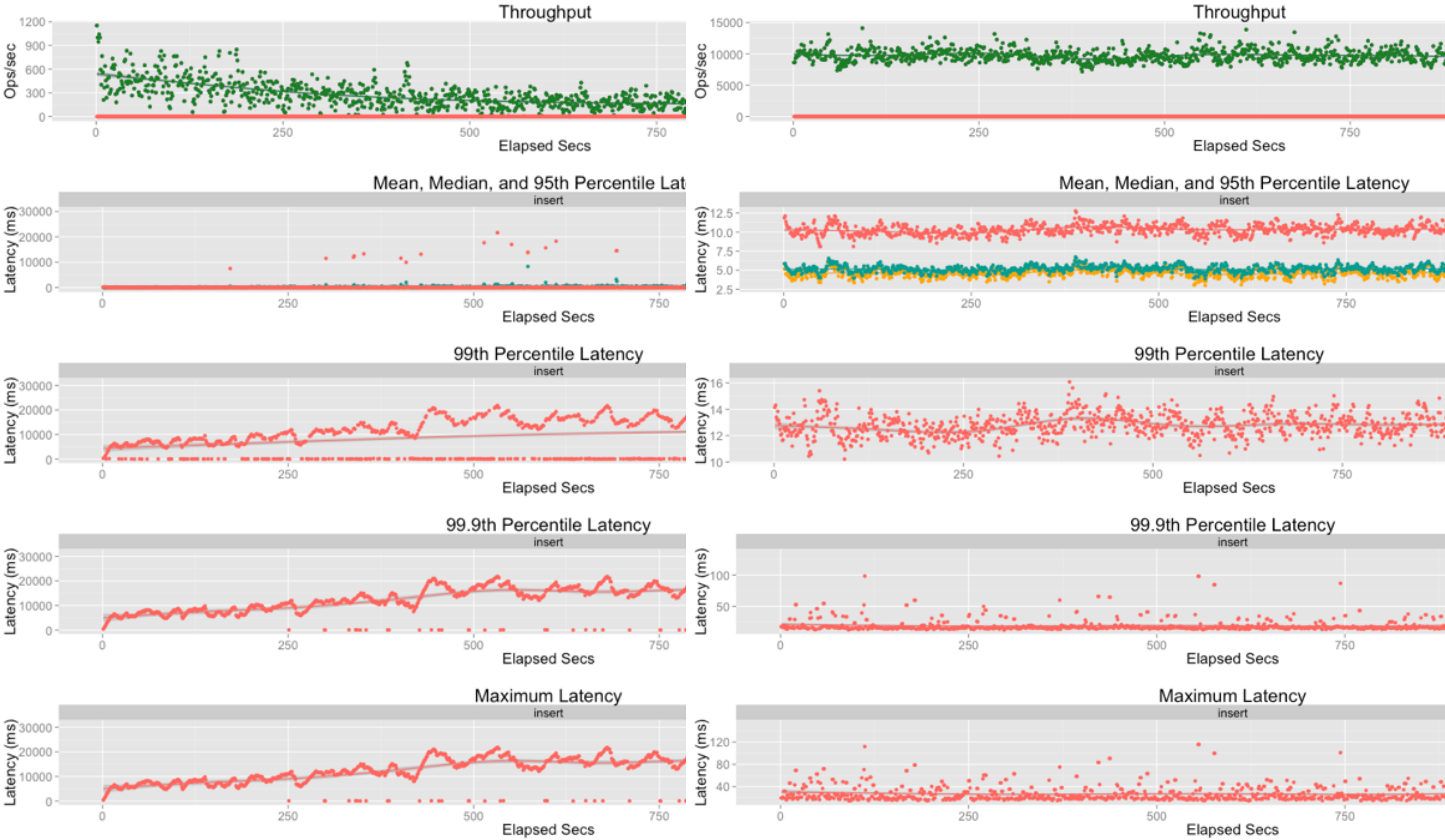
- Decomposed
- A clock, and some elements
  - each gets a key in leveldb

Initial Results





10k sets, 100k elements, 50 workers - write



10k sets, 100k elements, 50 workers - write

One small change

- Thinking from the bottom up
- Thinking about the disk and the database
- NOT a theoretical model

Bigset Design: write

- read clock
- increment
- assign dot to element
    - store clock+element
    - replicate delta

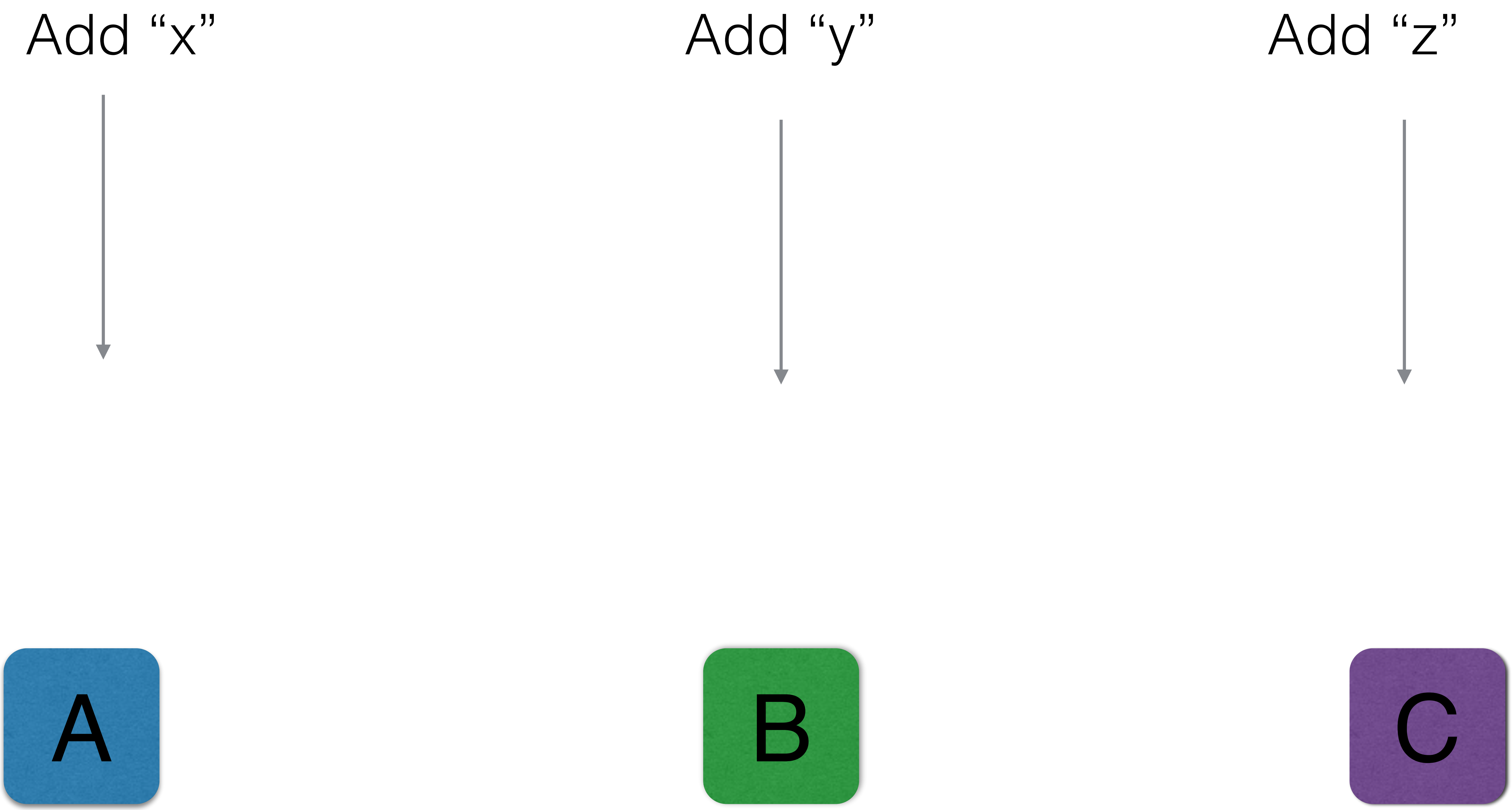
Bigset Design: write

- read clock
- if seen dot, ignore
- else add dot to clock
    - store clock+element

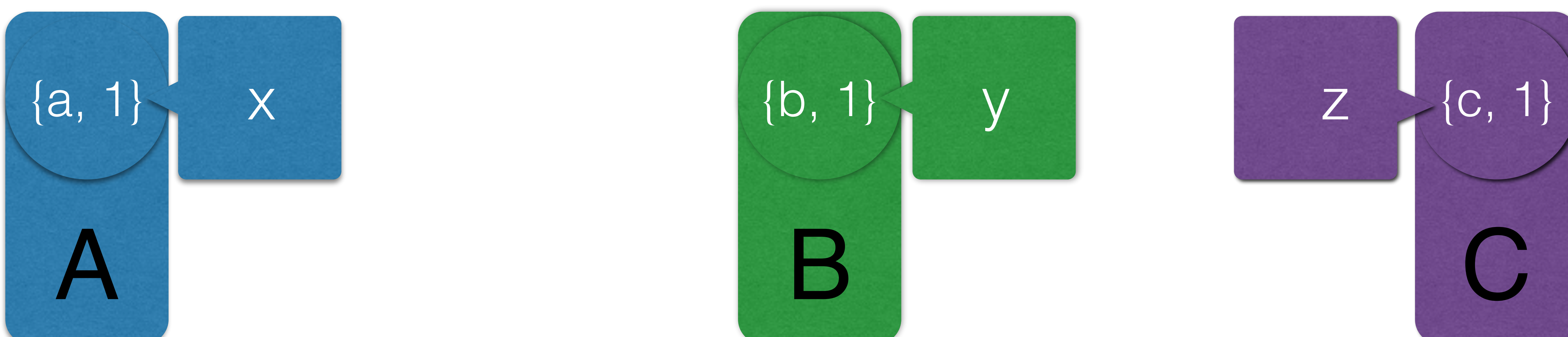
Bigset Design: Clock

- Base VV [{actor, counter}]
- “dot-cloud” [{actor, [counter]}]

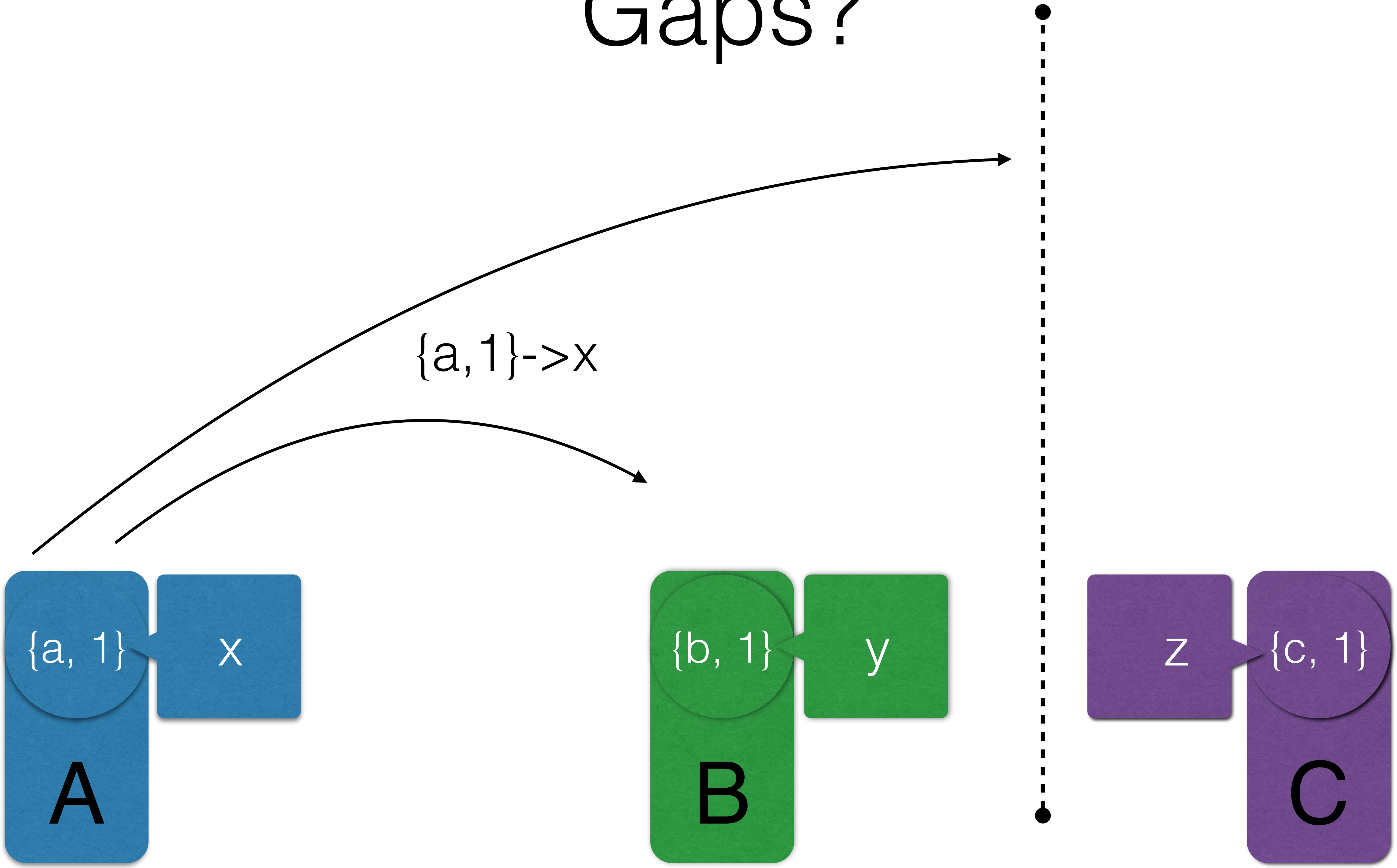
Bigset Design: Clock Gaps?



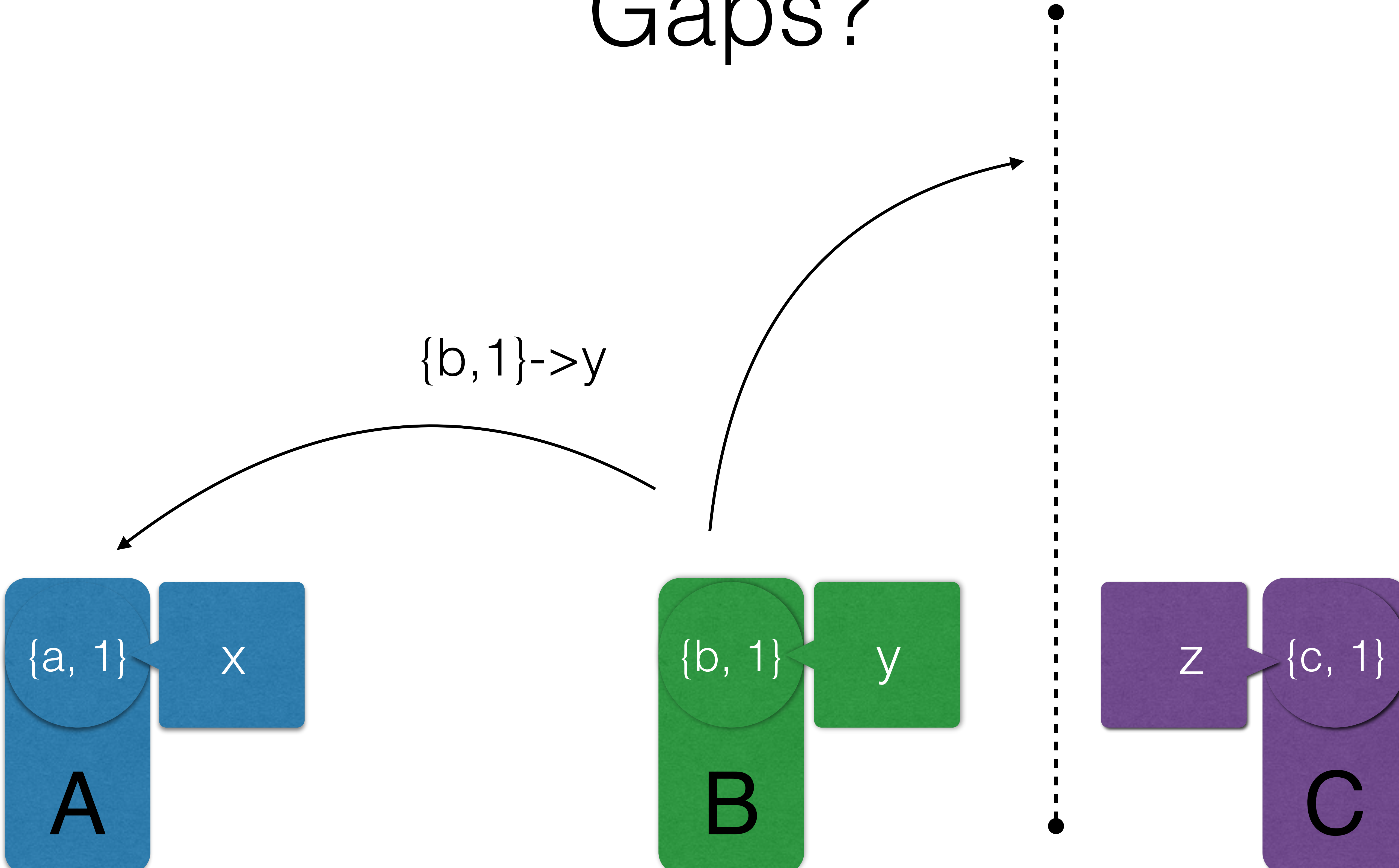
Bigset Design: Clock Gaps?



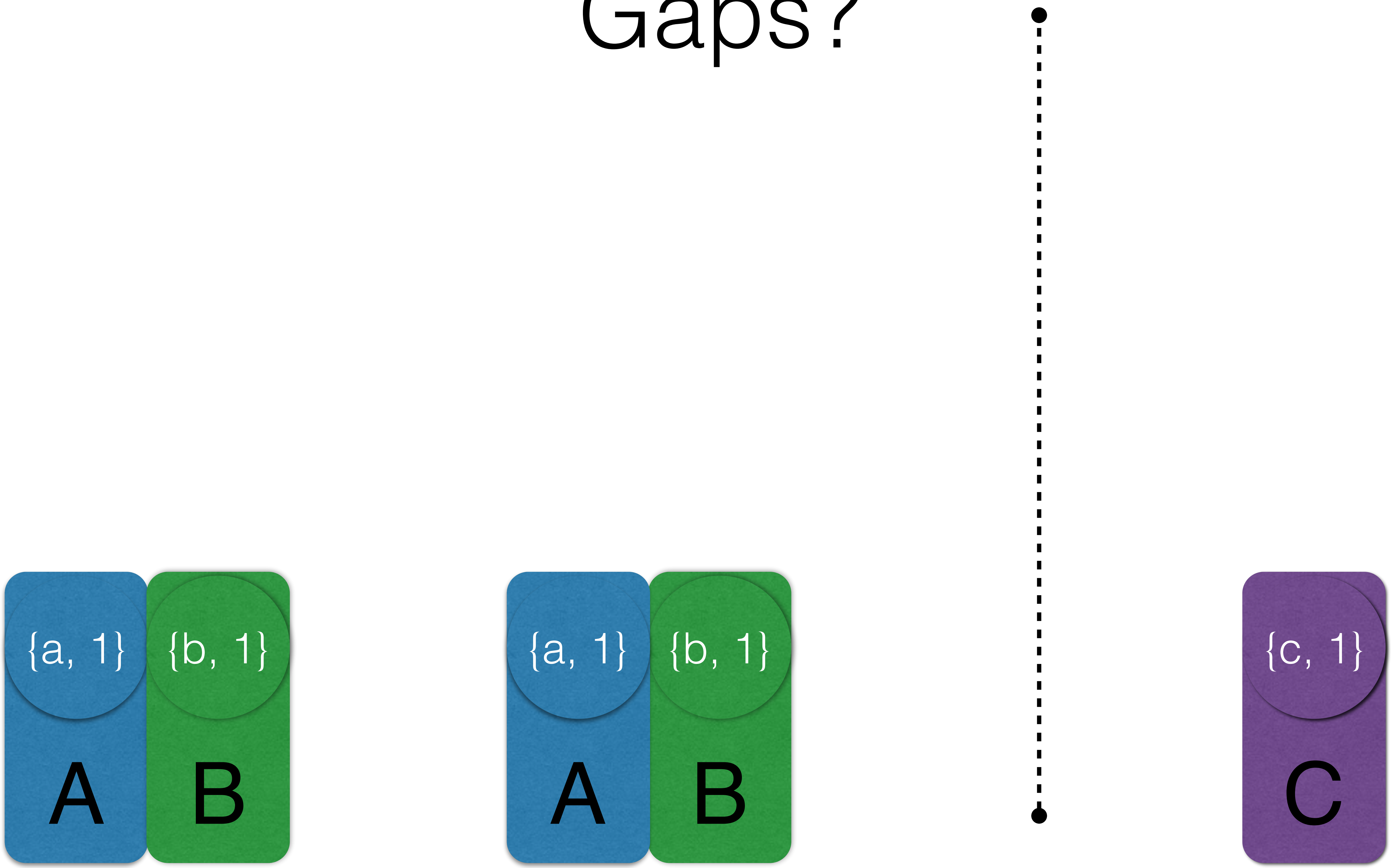
Bigset Design: Clock Gaps?



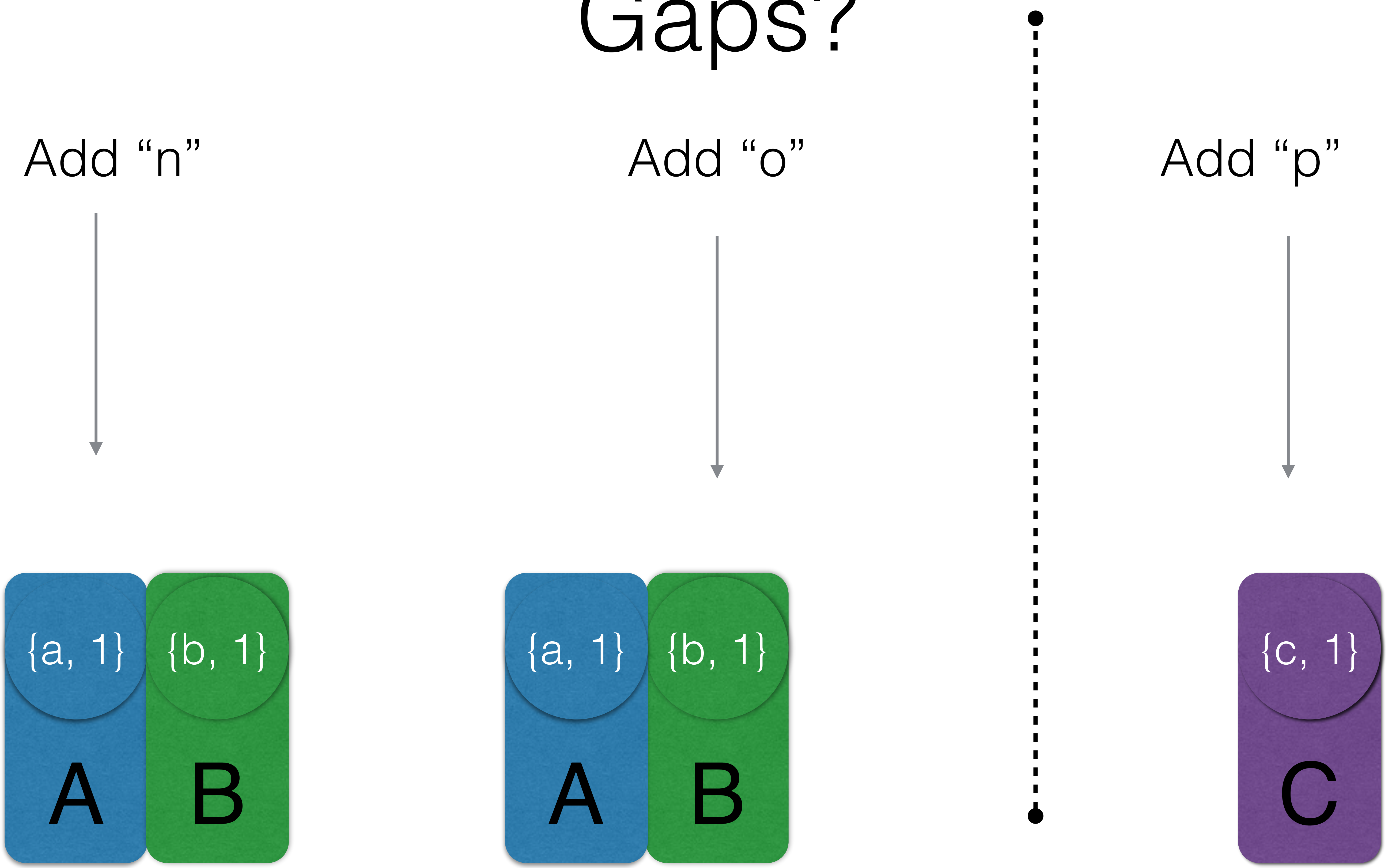
Bigset Design: Clock Gaps?



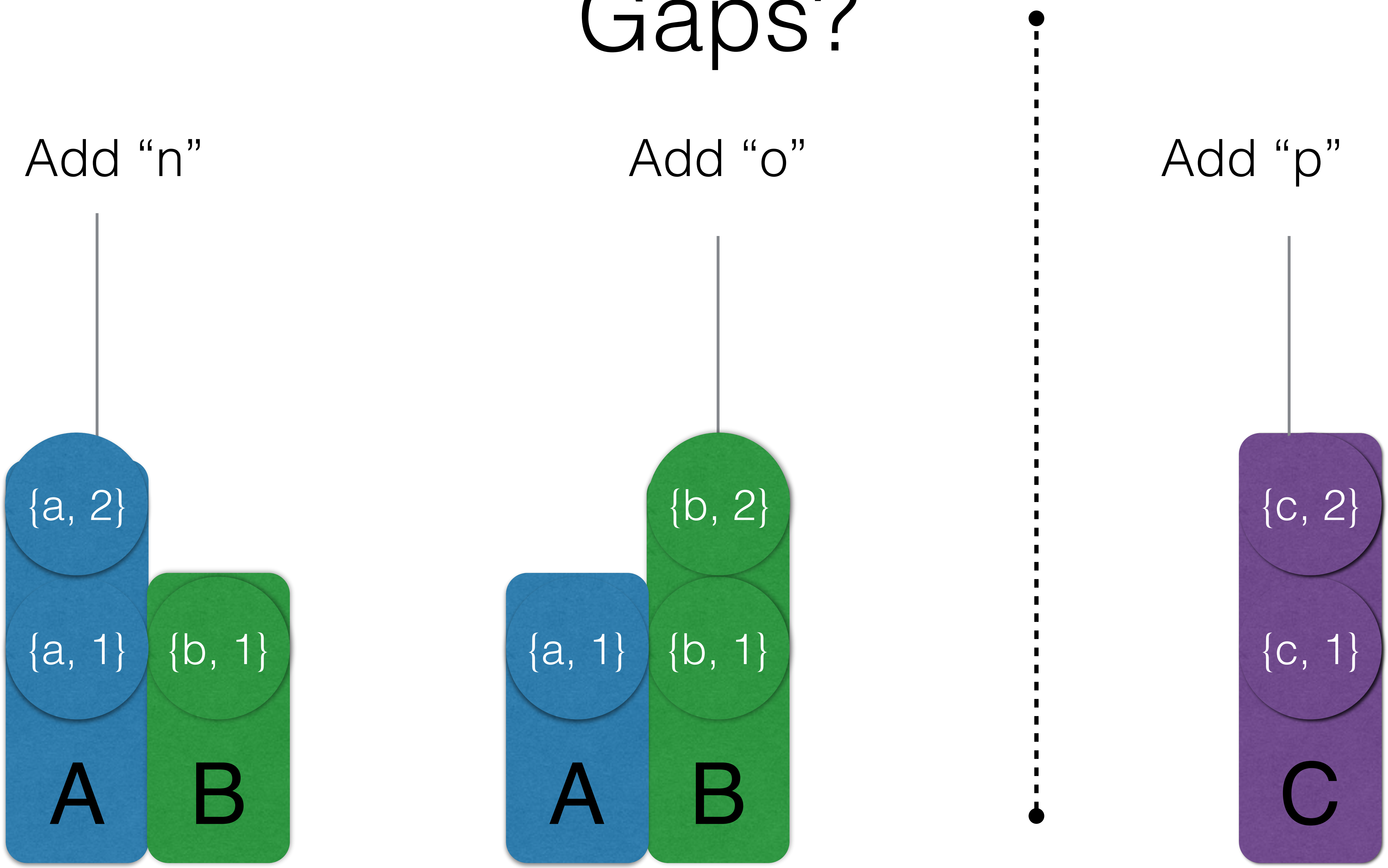
Bigset Design: Clock Gaps?



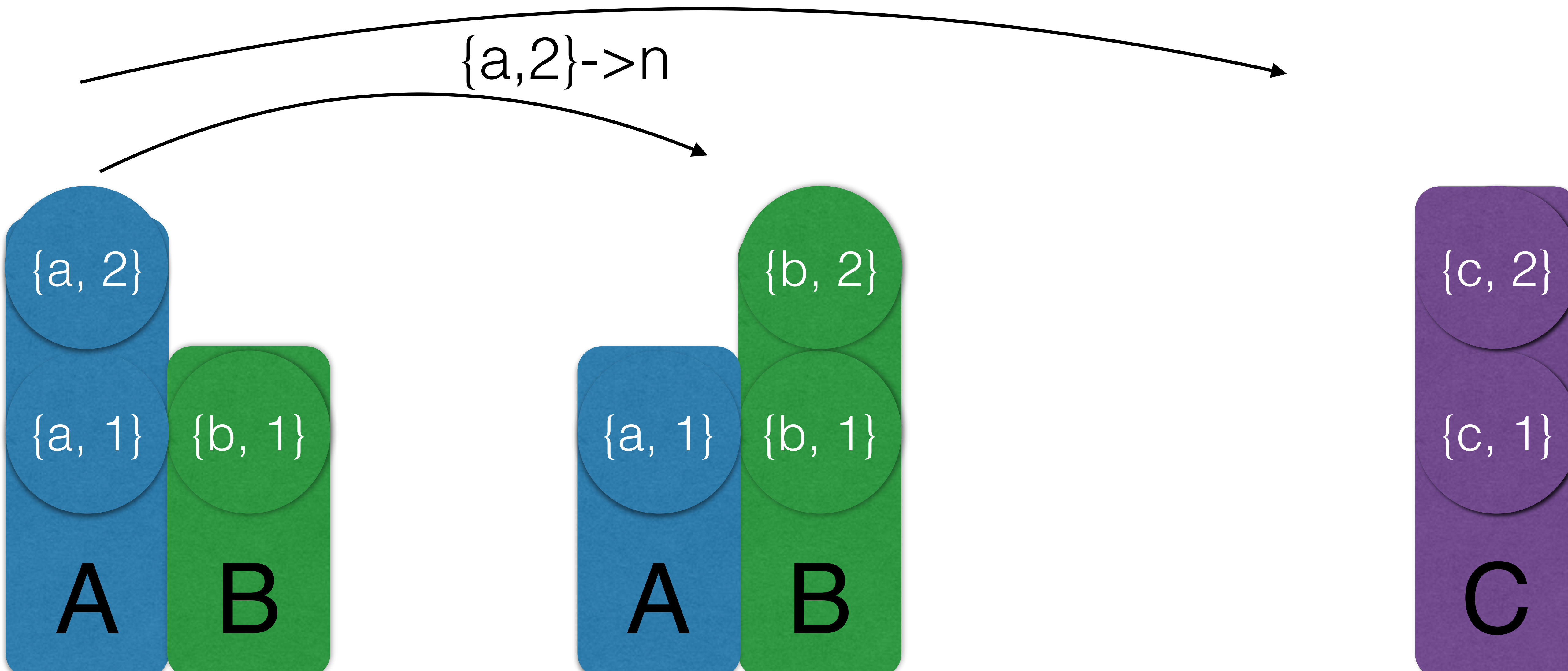
Bigset Design: Clock Gaps?



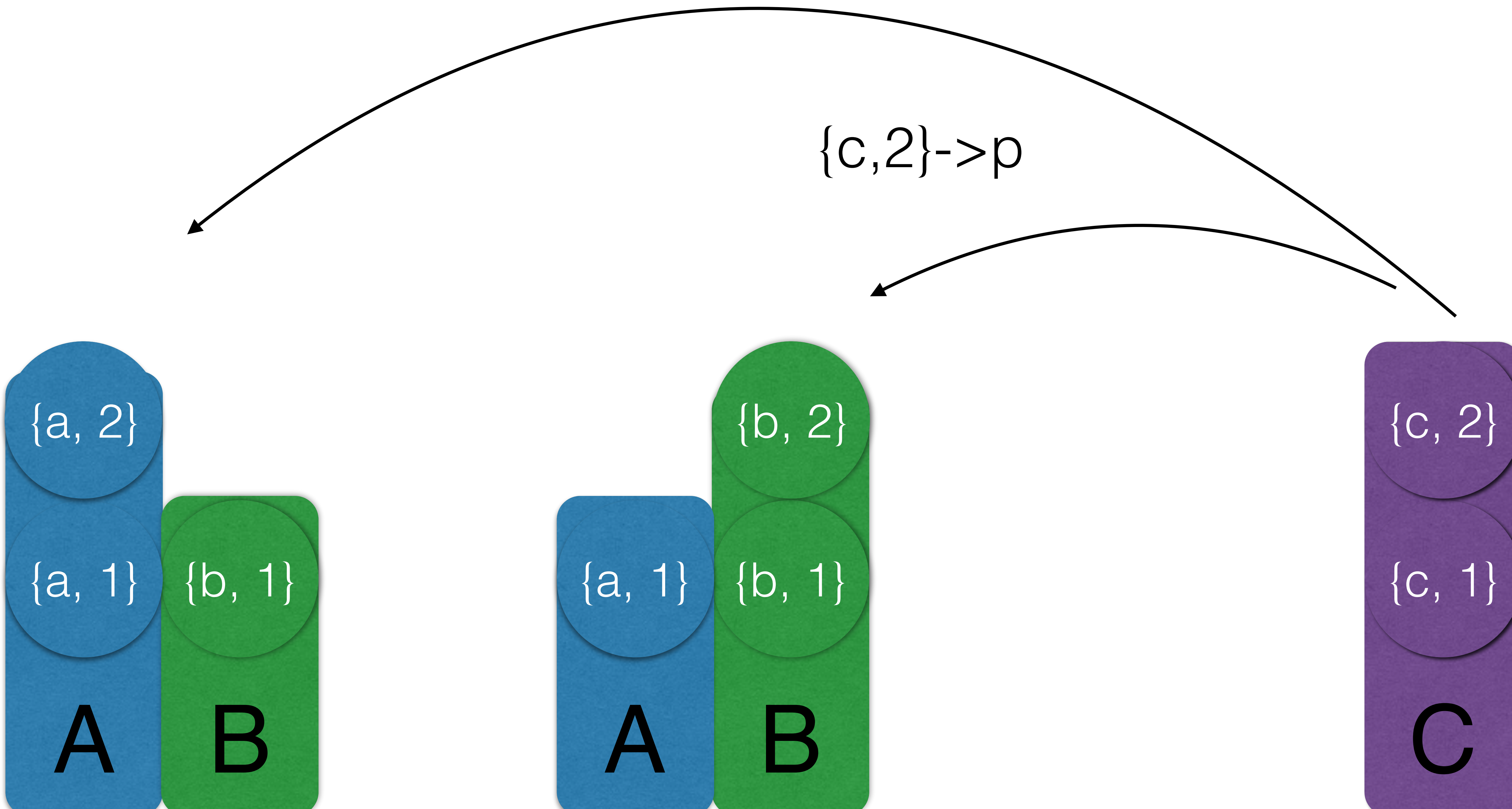
Bigset Design: Clock Gaps?



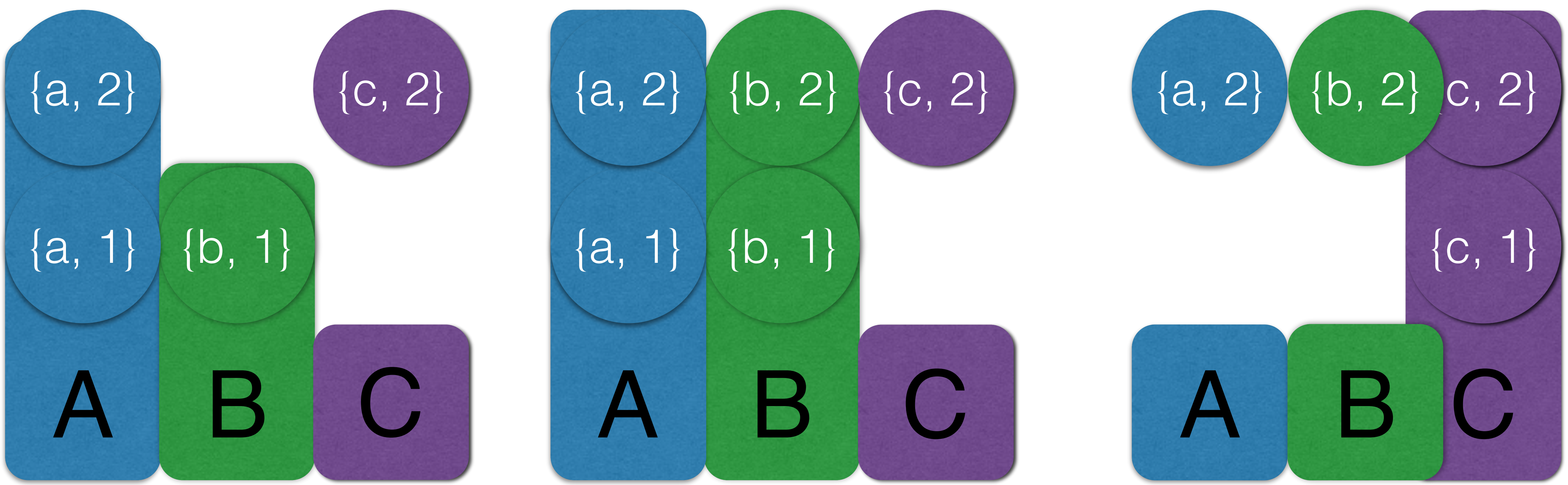
Bigset Design: Clock Gaps?



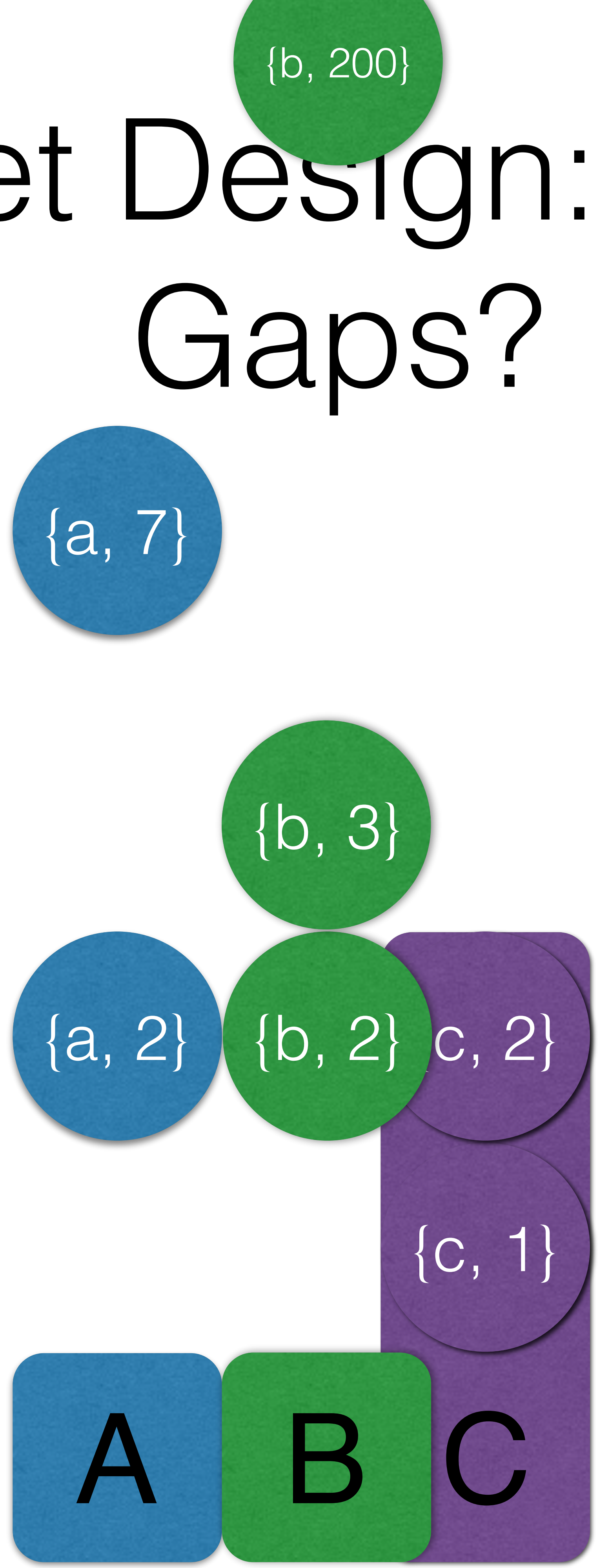
Bigset Design: Clock Gaps?



Bigset Design: Clock Gaps?



Bigset Design: Clock Gaps?



Bigset Design: elements

- <<Set, Element, Actor, Cnt>> so Actor,Cnt make a dot
  - Times/Space trade off for concurrent elements
- Ordered by Set, Element, Actor, Cnt
  - c++ key comparator for leveldb
  - No serialisation - fast writes

Bigset Design: End Key

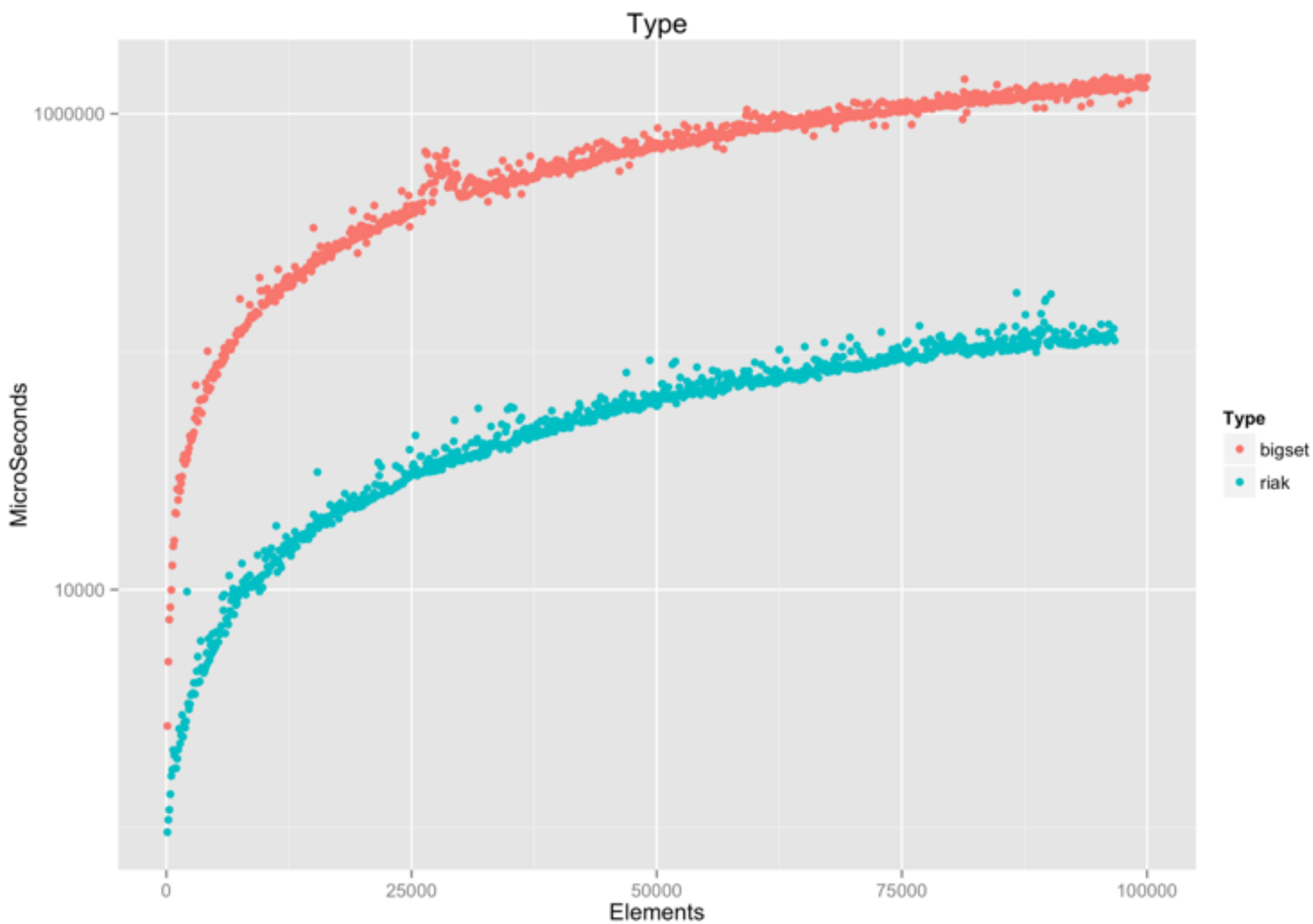
- <<Set, $z>>
- Sorts last

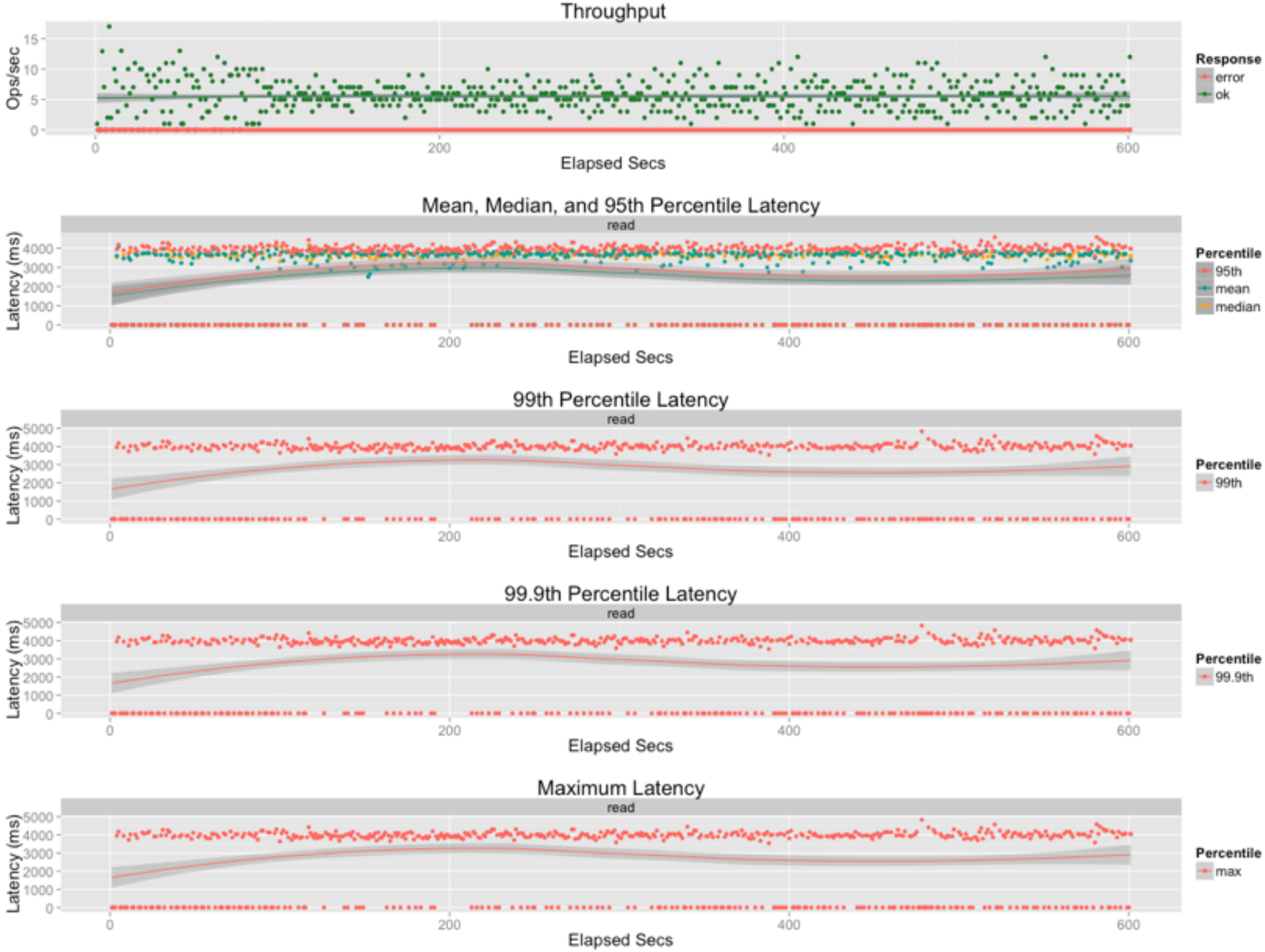
Bigset Design:Sorting

- Clock first, then elements, the end key
- For each set the keys are contiguous

Reads?

Initial Read Results





10k sets, 100k elements, 20 workers - read

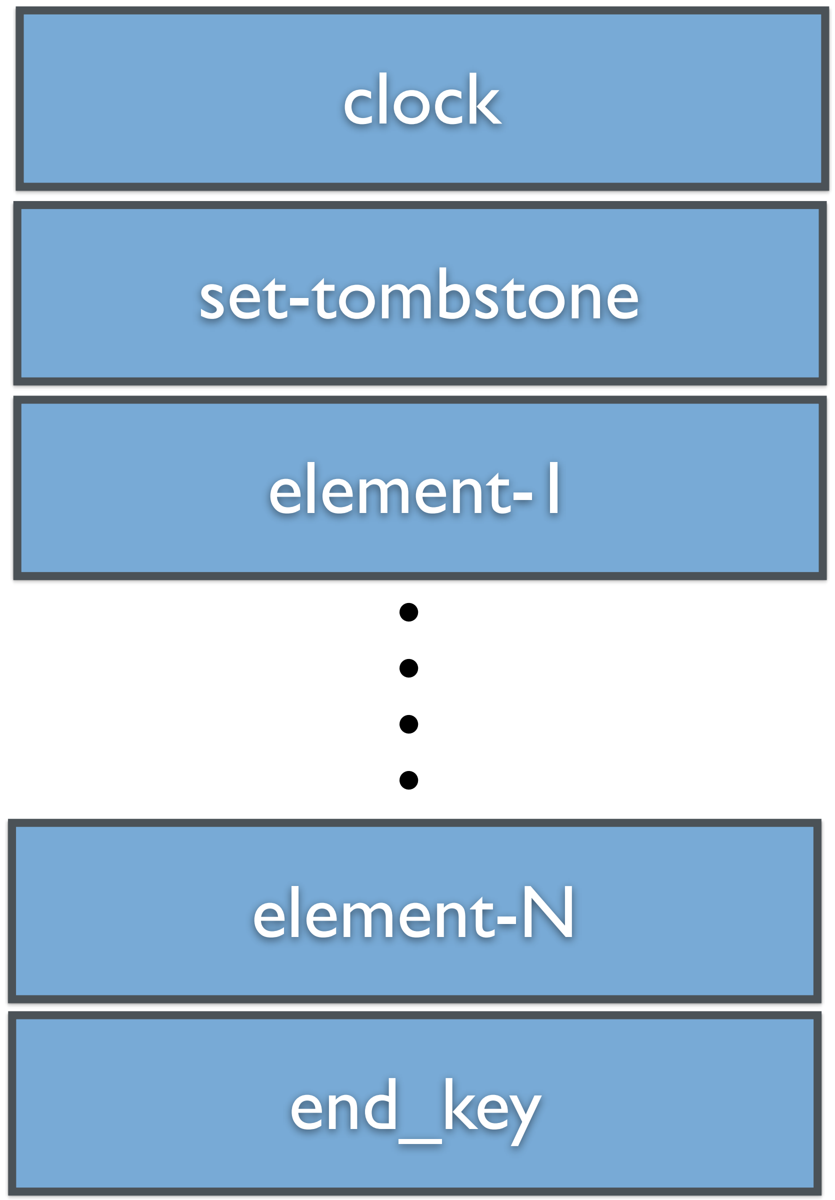
Bigset Design: Read

- Iterate over many keys
- Leveldb iterate -> erlang fold

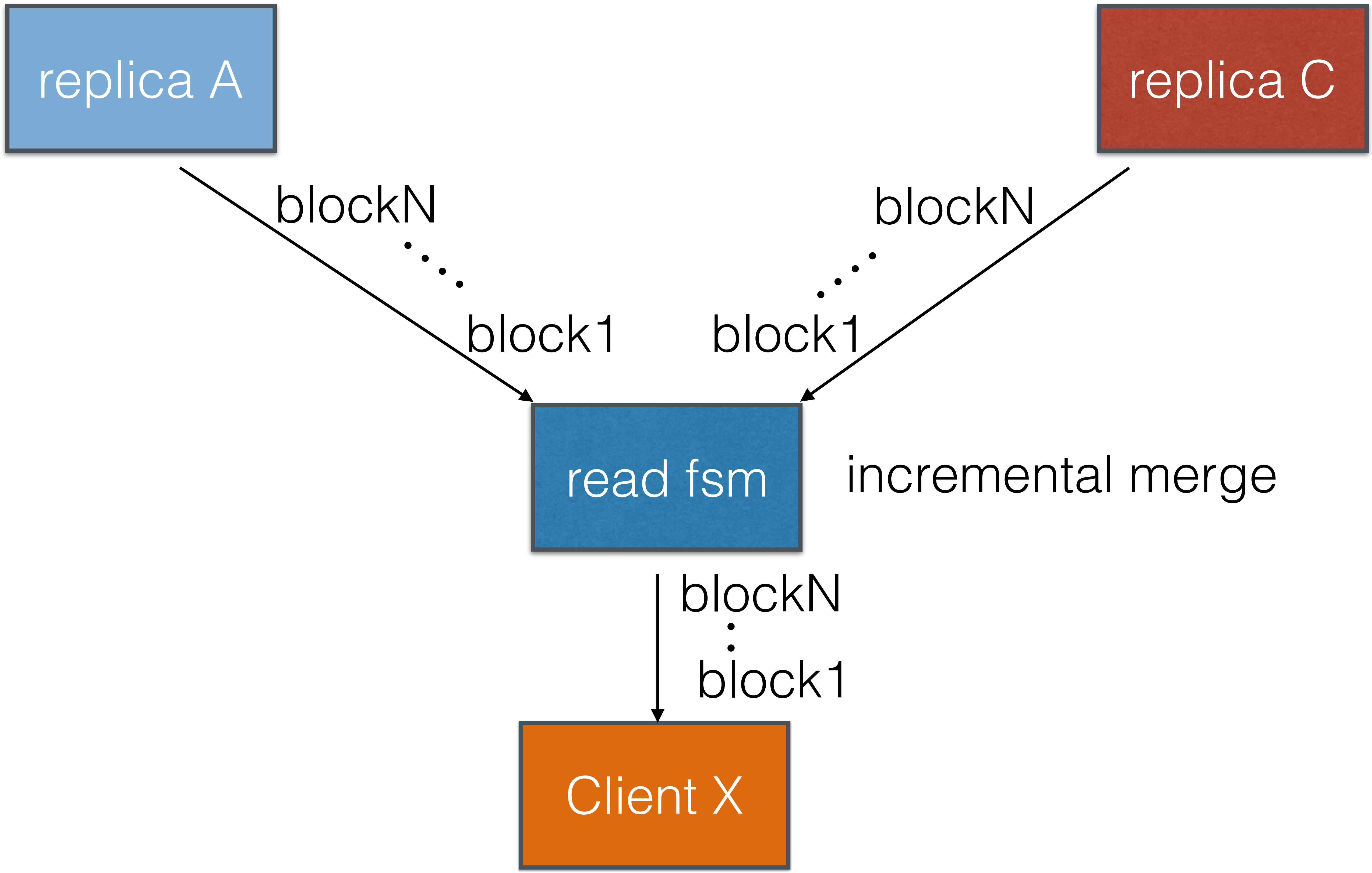
Bigset Design: Read

- "Streaming Fold" over Set (start-to-key-end key in a buffer)
  - Configurable chunks, say 100k elements
- Stream keys in batches to read_fsm - back pressure
- Read fsm incremental ORSWOT merge over R replicas
  - stream results to client

Bigset Design



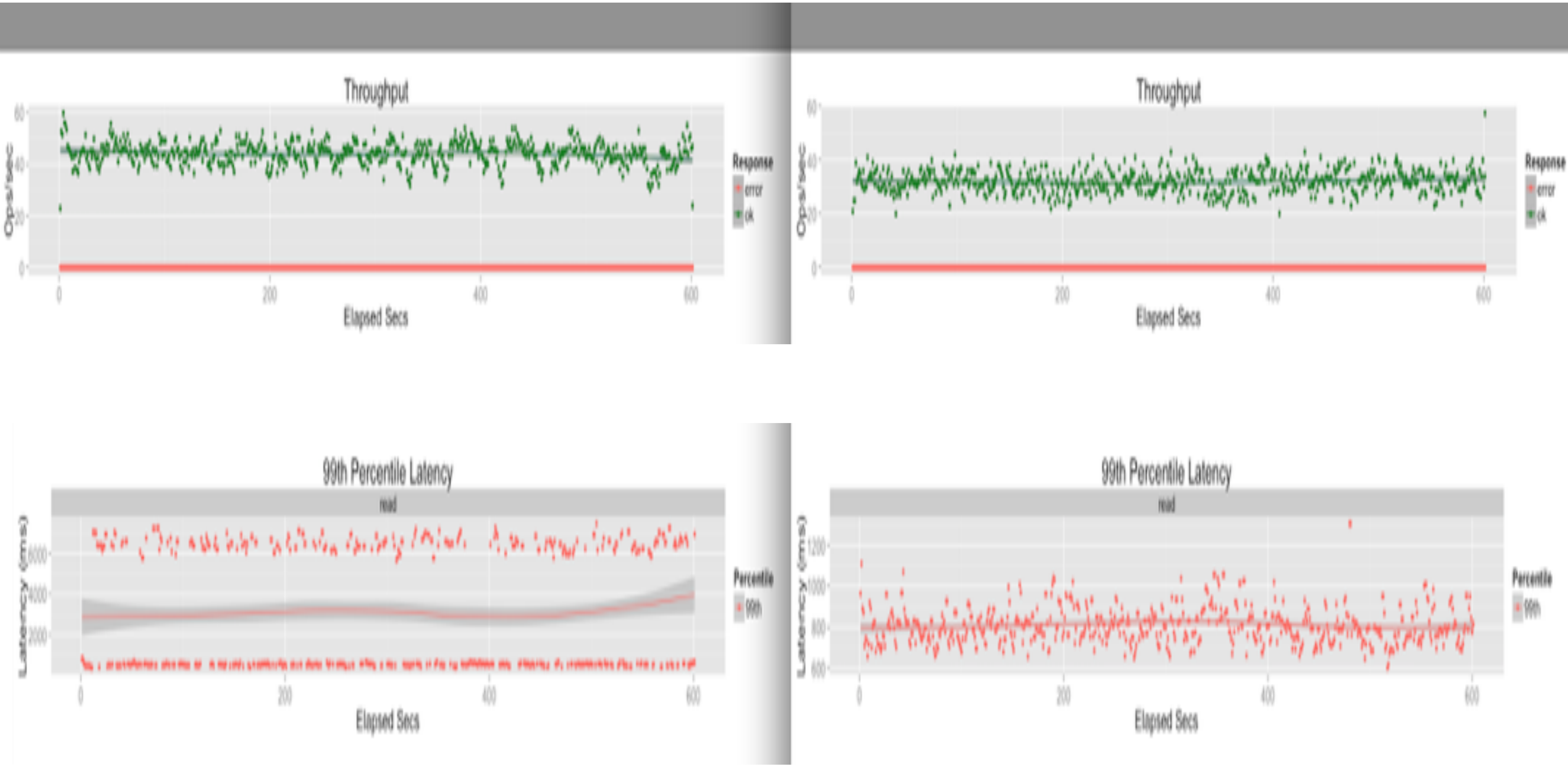
Bigset Design: read



Reads Today

10k sets, 100k elements, 20 workers - read

Reads Today

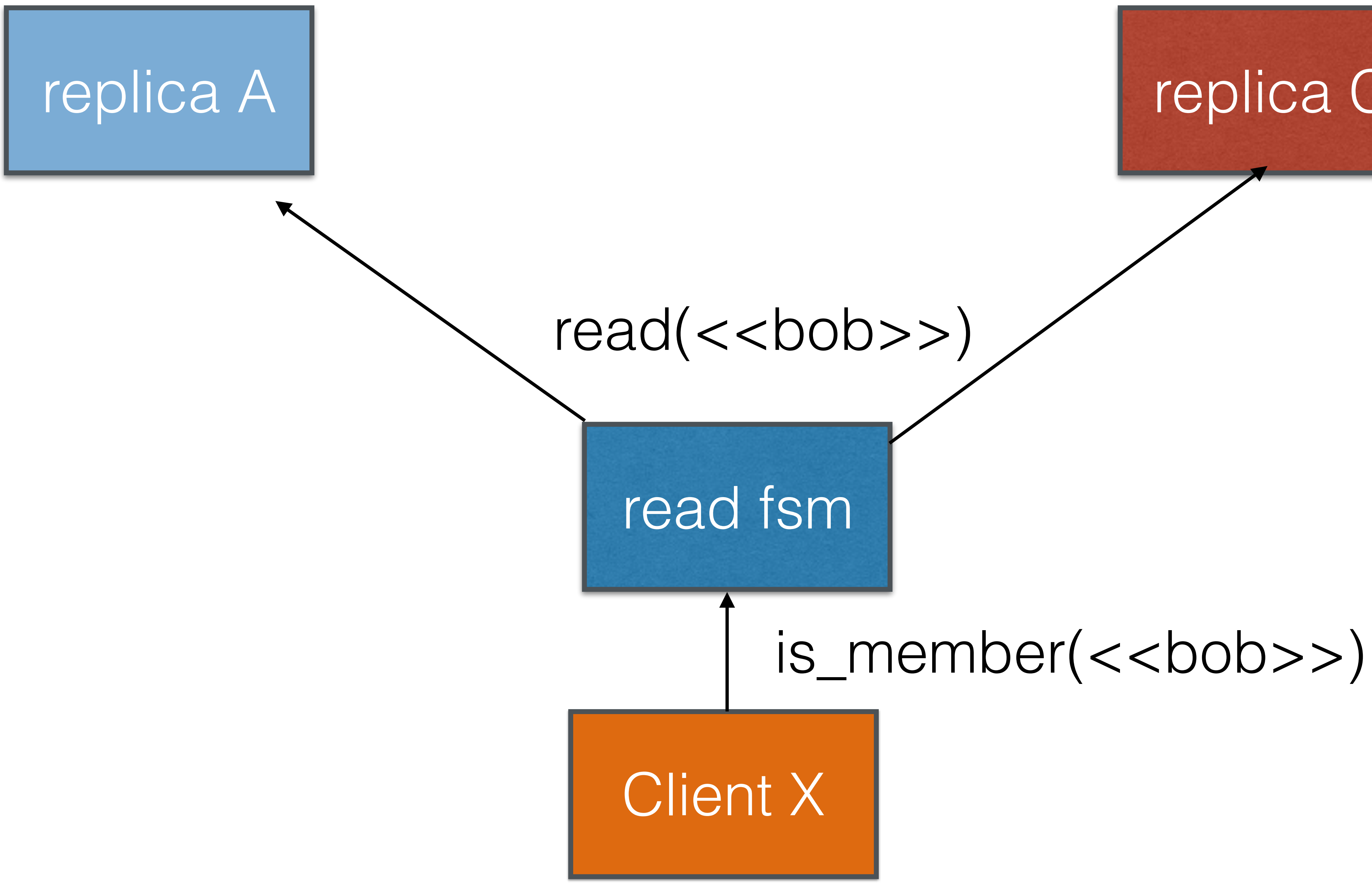


10k sets, 100k elements, 20 workers - read

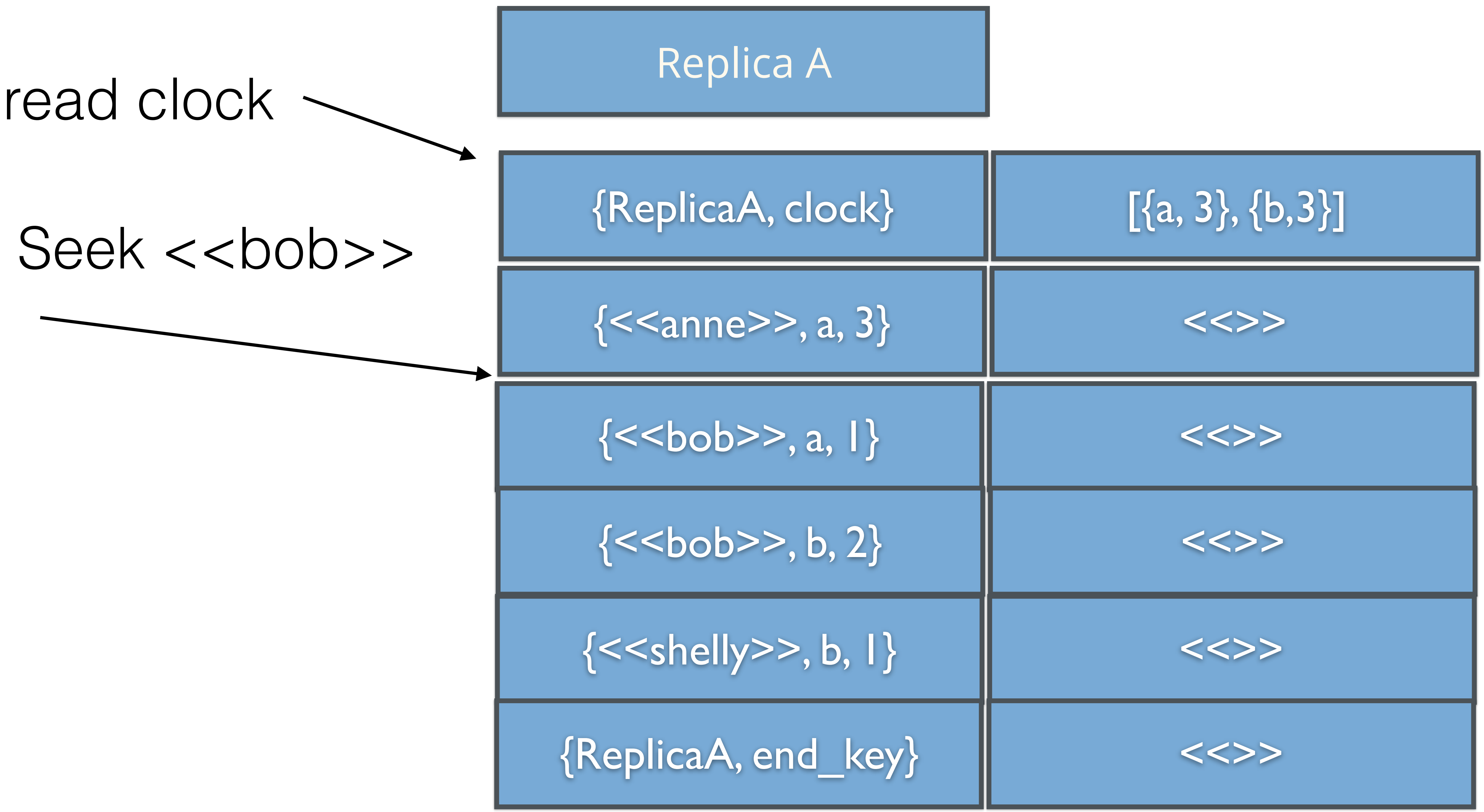
Full Set Read or Queries?

- Decomposed design enables queries
  - Is Member
  - subset queries - per vnode is c++
  - Range queries SORTED!
  - Pagination

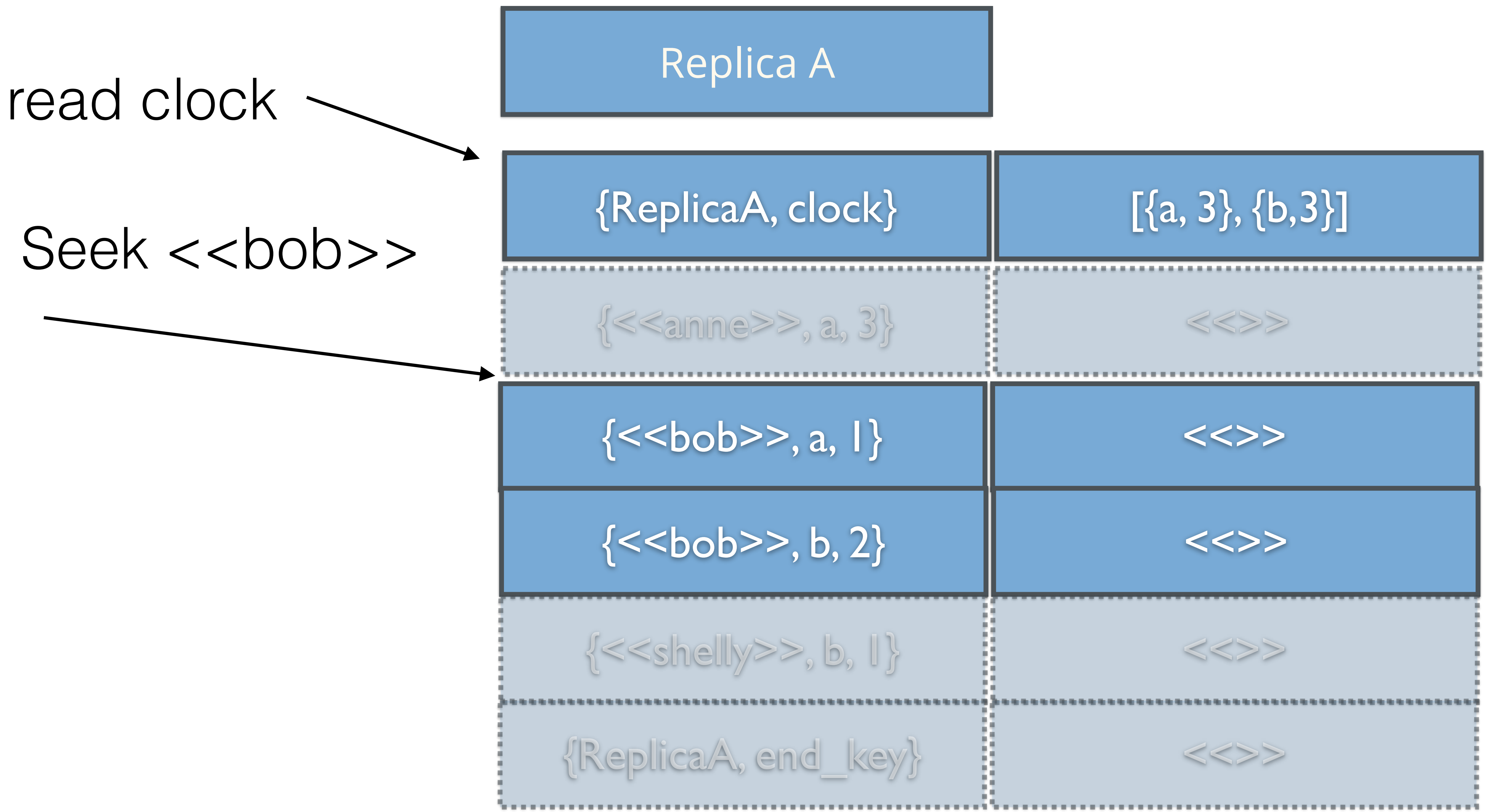
Is Member(X)



Is Member(X)

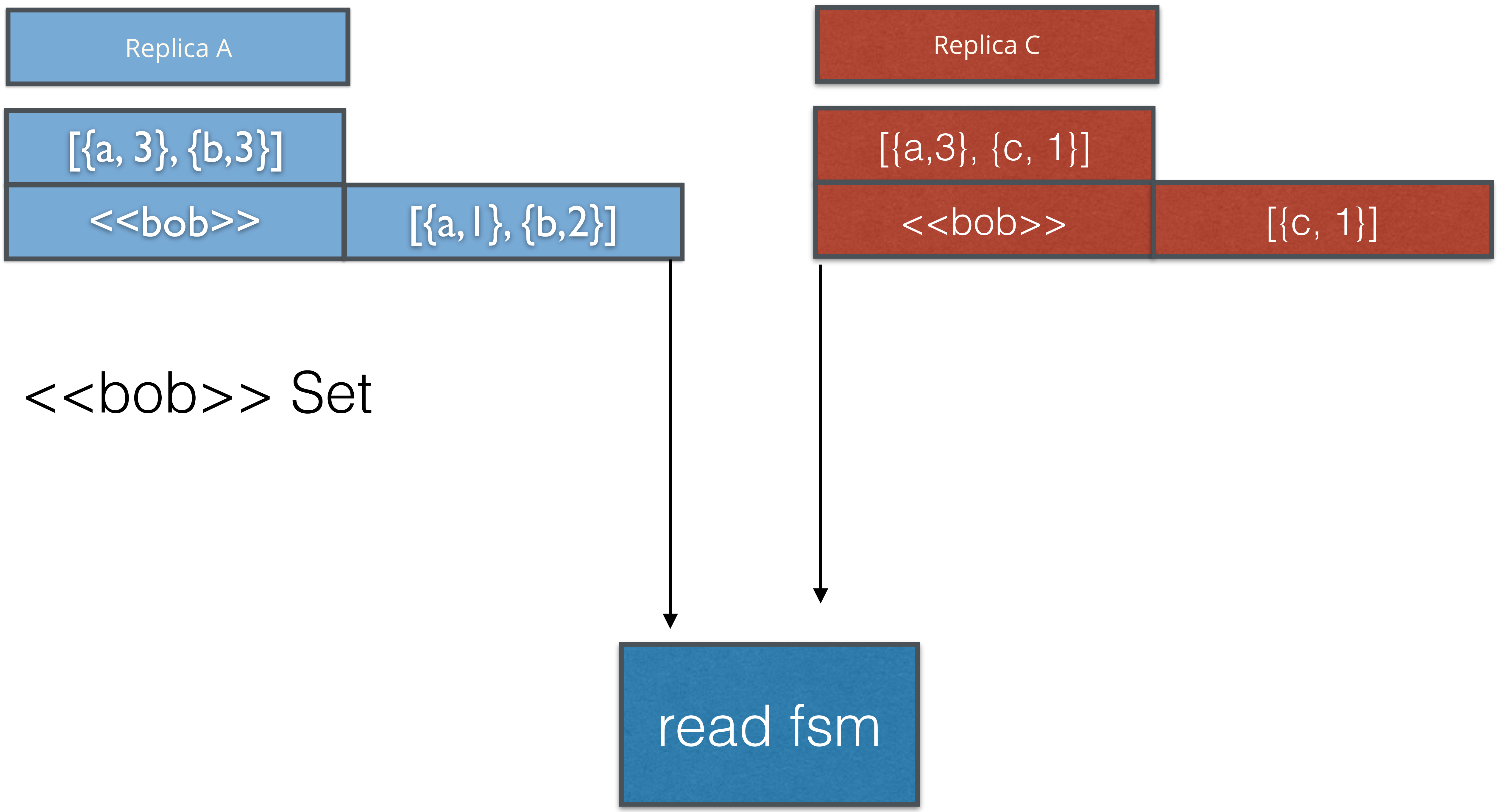

read clock
Seek <<bob>>
Replica A
{ReplicaA, clock}
[{a, 3}, {b,3}]
{<<anne>>, a, 3}
<<>>
{<<bob>>, a, 1}
<<>>
{<<bob>>, b, 2}
<<>>
{<<shelly>>, b, 1}
<<>>
{ReplicaA, end_key}
<<>>

Is Member(X)

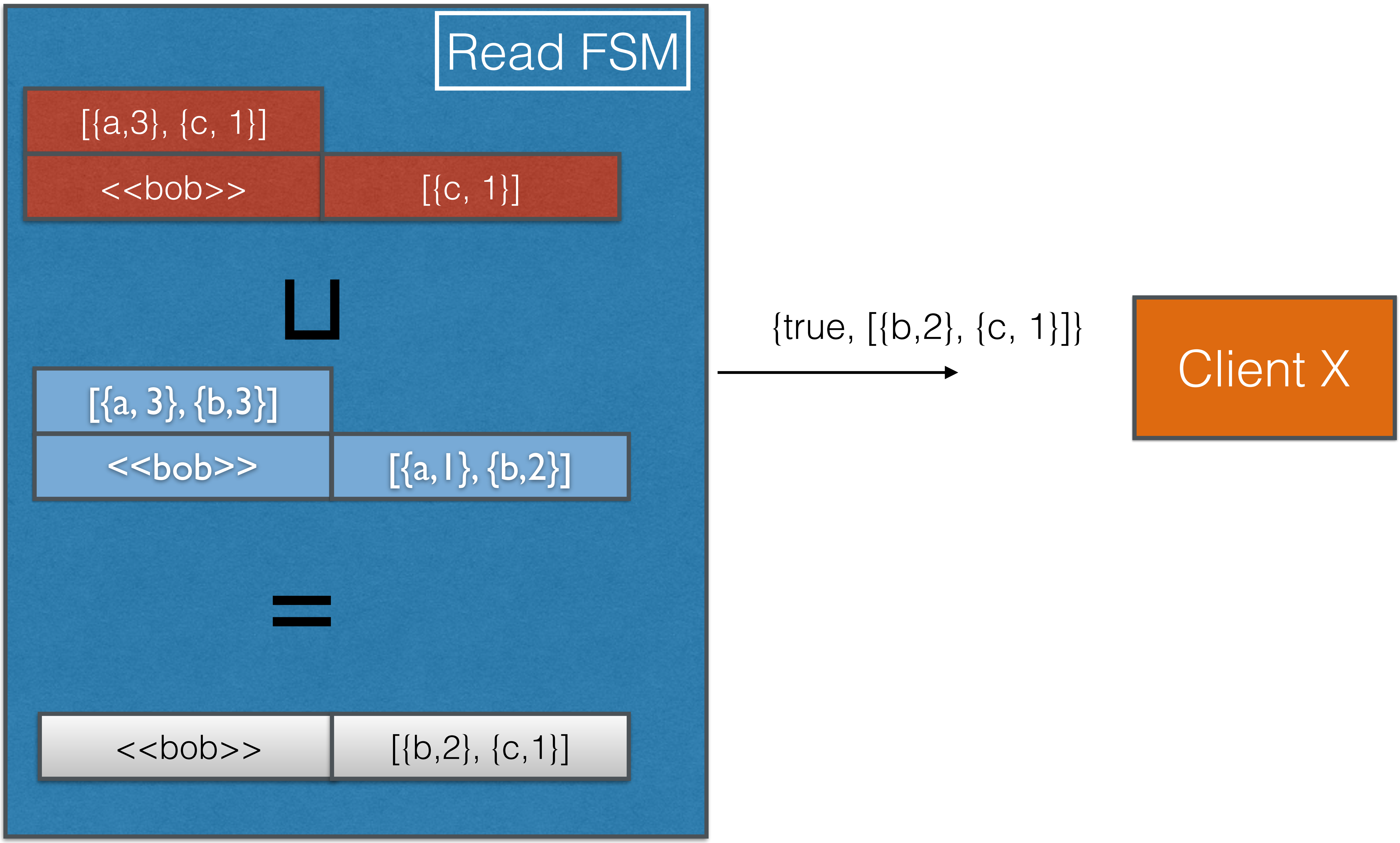


Replica A
read clock
{ReplicaA, clock}
[{a, 3}, {b,3}]
Seek <<bob>>
{<<anne>>, a, 3}
<<>>
{<<bob>>, a, 1}
<<>>
{<<bob>>, b, 2}
<<>>
{<<shelly>>, b, 1}
<<>>
{ReplicaA, end_key}
<<>>

Is Member(X)



Is Member(X)



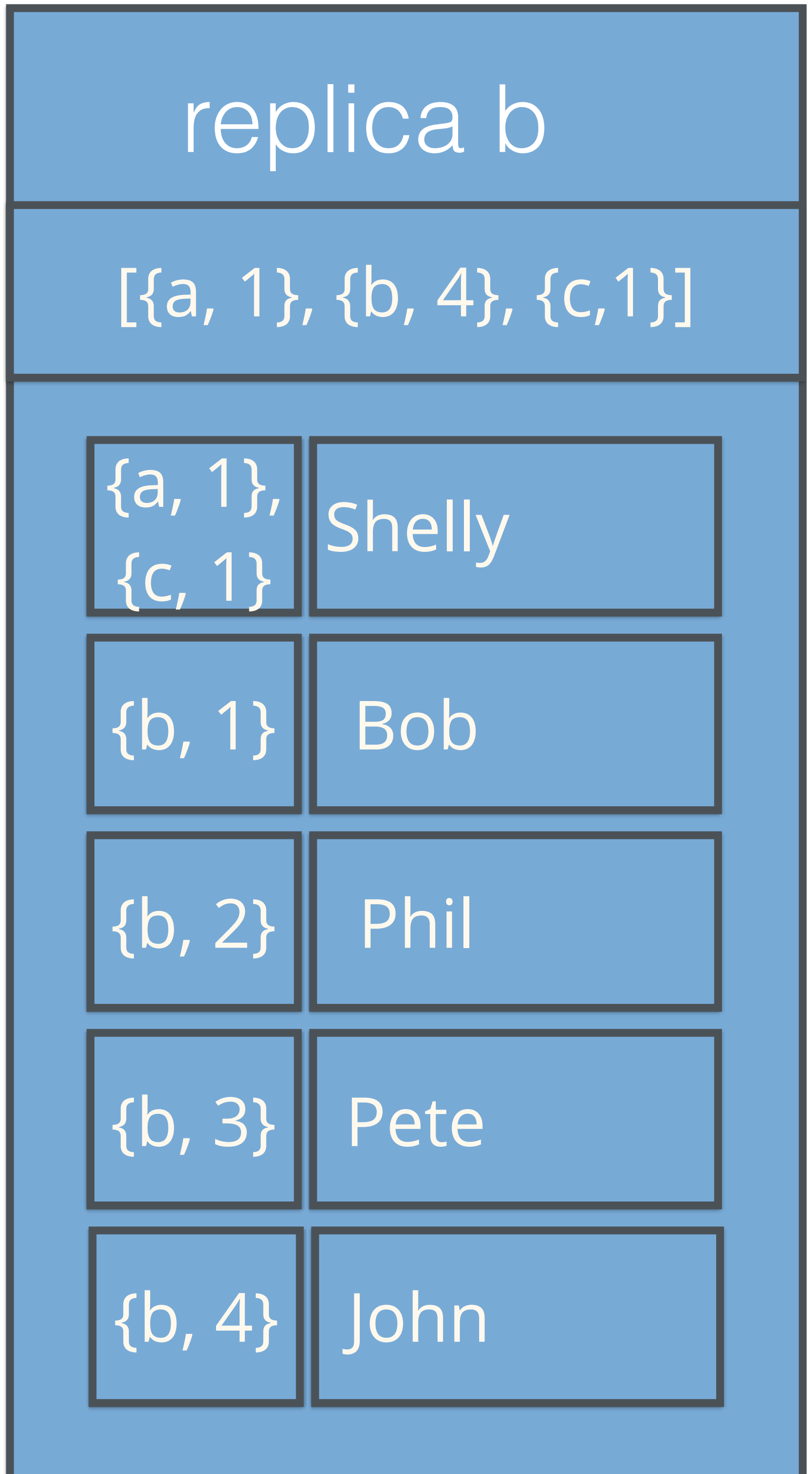
Removes

- Observed-Remove - context
- Requires _some kind_ of read
  - cheap membership check

Sets in Riak

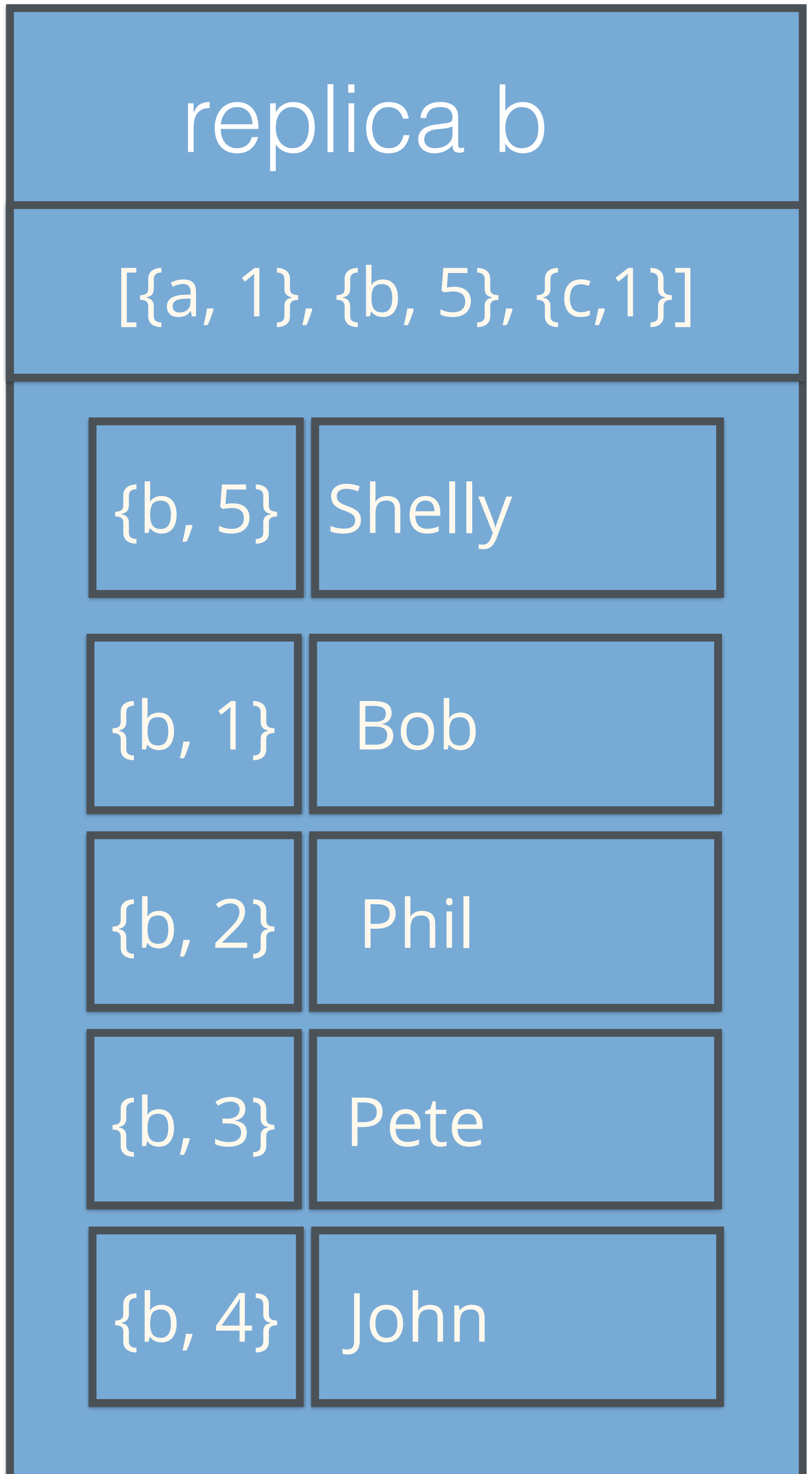
- Adds are removes!
- Action-at-a-distance!
  - Clients are NOT replicas

Adds are removes

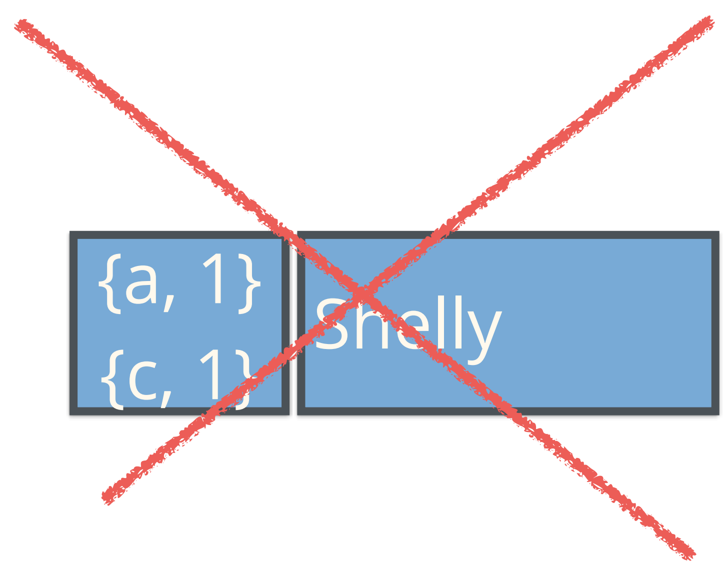


Add “Shelly”

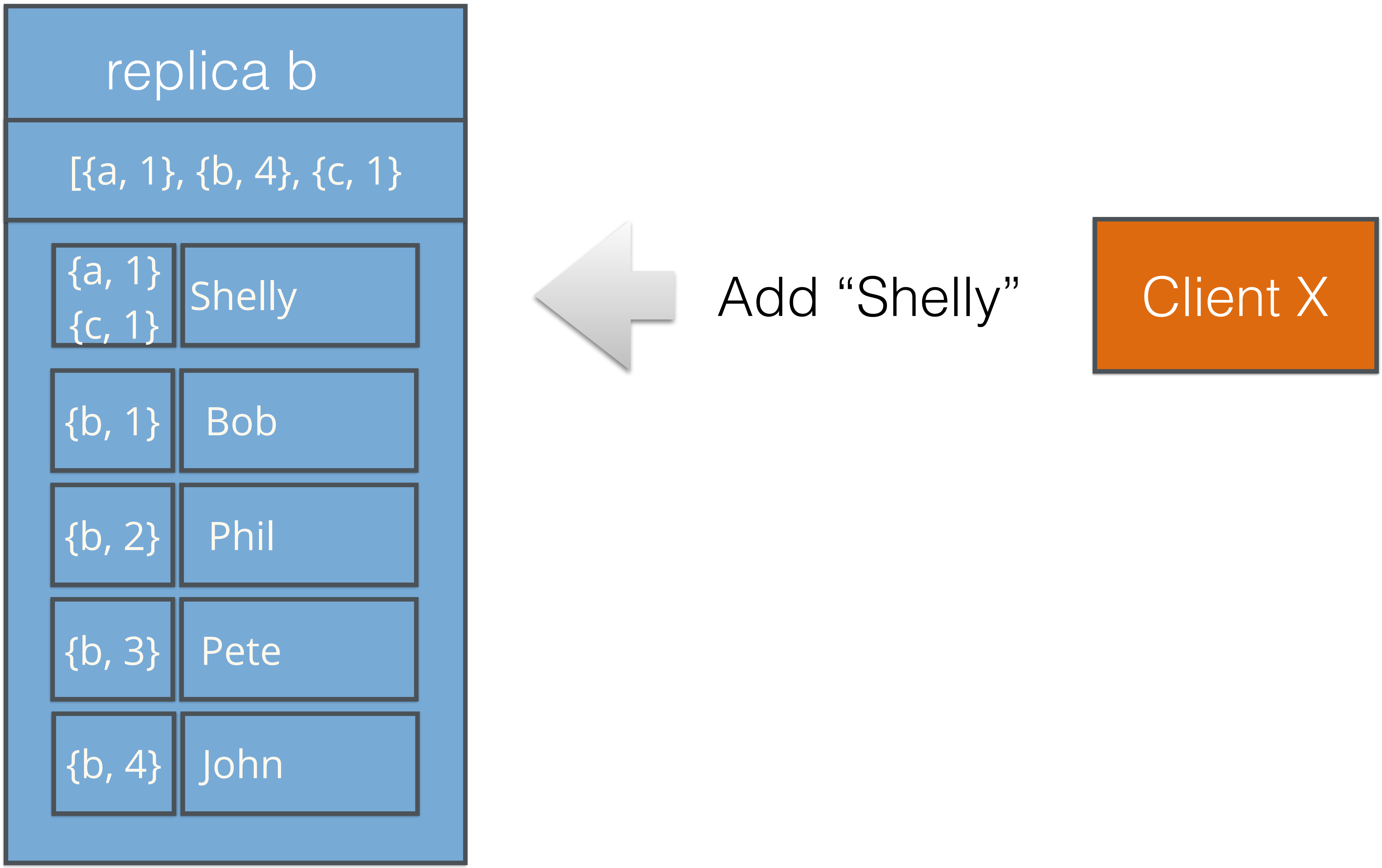
Adds are removes



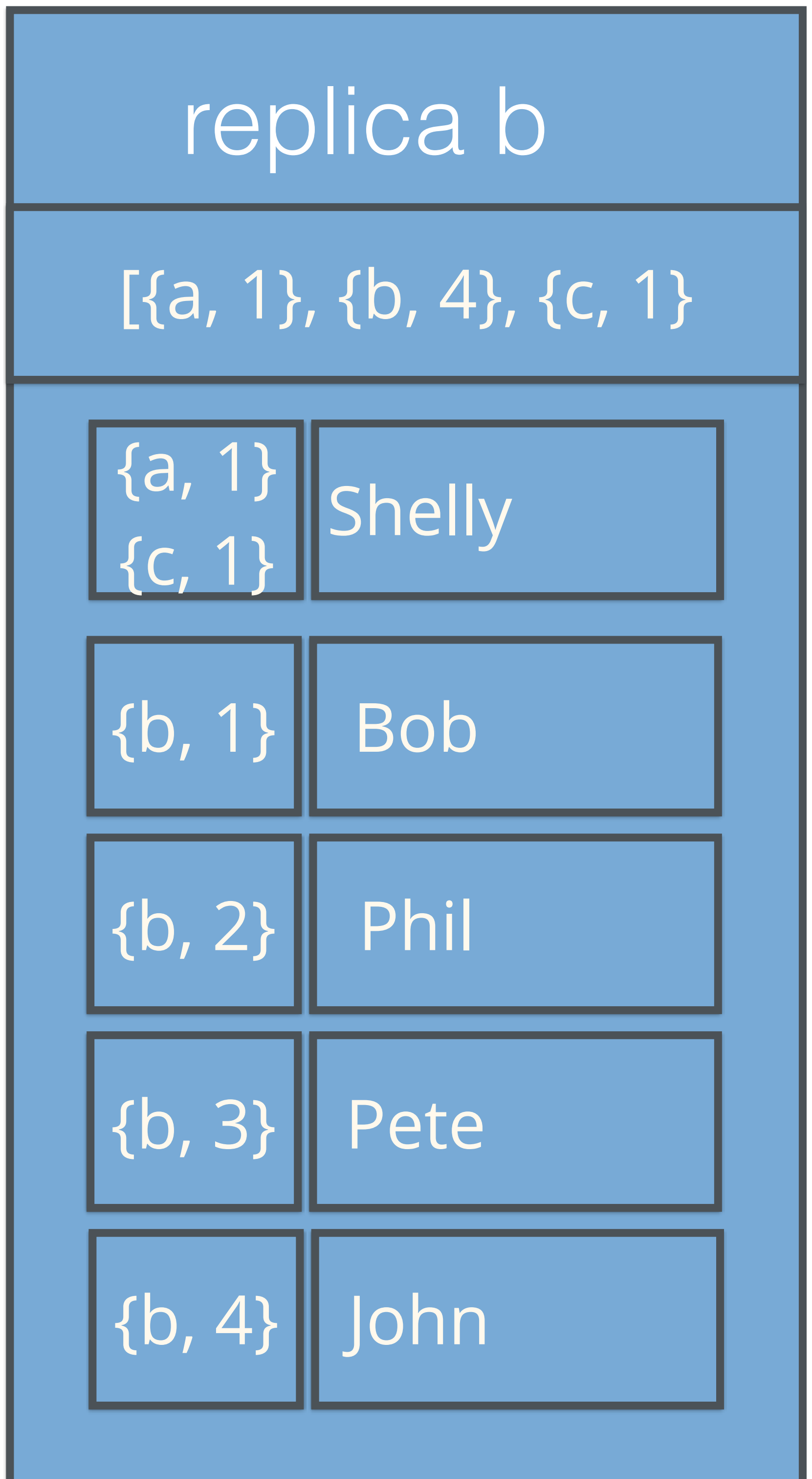
Not concurrent - Seen



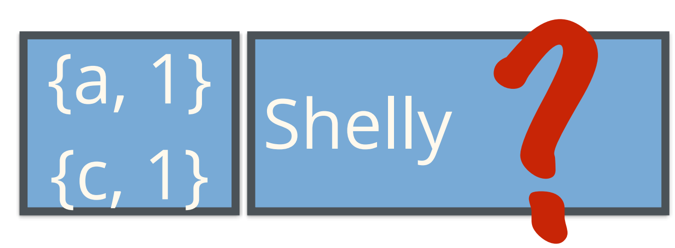
Action-at-a-Distance



Action-at-a-Distance



Are adds removes?
Concurrent?
Seen?



Contexts & Consistency

- add X no ctx
  - empty ctx - always concurrent (safe!)
  - local ctx - non-deterministic (could remove all, some, none other X)
    - depends on handling vnode’s state

Contexts & Consistency

- add X + per element ctx
  - only removes observed X regardless of handling vnode
  - mmmm, deterministic outcome

Bigset

- Adds are removes!
- They need a context
  - Cheap is_member(X)

Elided Complications

- Hand-Off
  - 1-way Full state merge
    - Without reading keys on receiving side!
    - Event Set Maths
  - Anti-Entropy
  - Read Repair
- Multi-Data-Center

Bigset Handoff

- MUST read full set at sender
- Read full state at receiver?
    - Would be bad

Bigset Handoff

- Sender sends keys
  - Keys receiver hasn’t seen - store
  - key receiver has seen - ignore
  - Only read clock!

Bigset Handoff

- Sender doesn't send keys it has removed
  - key receiver never saw - merge clocks
    - Just read clock!
  - key receiver saw - add to set-tombstone

Set-Tombstone

- (Compact) Set of causal tags of removed keys
- Stored like on disk in a key

Bigset Handoff

- handoff receiver state
  - On sender clock_key
    - C=clock, T=bigset_clock:fresh()
    - for each key received add dot to tracker

Bigset Handoff

- On end_key
  - C - T = Removed Dots
  - ToRemove = Receiver Clock intersected with Removed Dots
  - Receiver Tombstone + ToRemove

Compaction

- set-tombstone - logical clock/set of dots
- leveldb compaction
  - if dot < set-tombstone discard
  - set-tombstone = set-tombstone - dot
    - tell vnode after compaction “remove dots [dot()] from set-tombstone”

Next?

- Production level code for Riak 2.x
- Causal Consistency
- More types - “Big”Maps
- Tables of Maps or Sets
- SQL over Eventual Consistency

Summary

- Eventual Consistency buys you low latency & availability
- Conflicts can be hard for application developers
- CRDTs help
- A little engineering goes a long way
    - Decomposition brings complications too