LIVING IN EVENTUALLY CONSISTENT REALITY

INTRODUCTION

Bartosz Sypytkowski

@Horusiath

b.sypytkowski@gmail.com

bartoszsypytkowski.com

AGENDA

- Eventual consistency with CRDTs
- Existing solutions
- CRDT – basics and optimizations
- Different notions of time

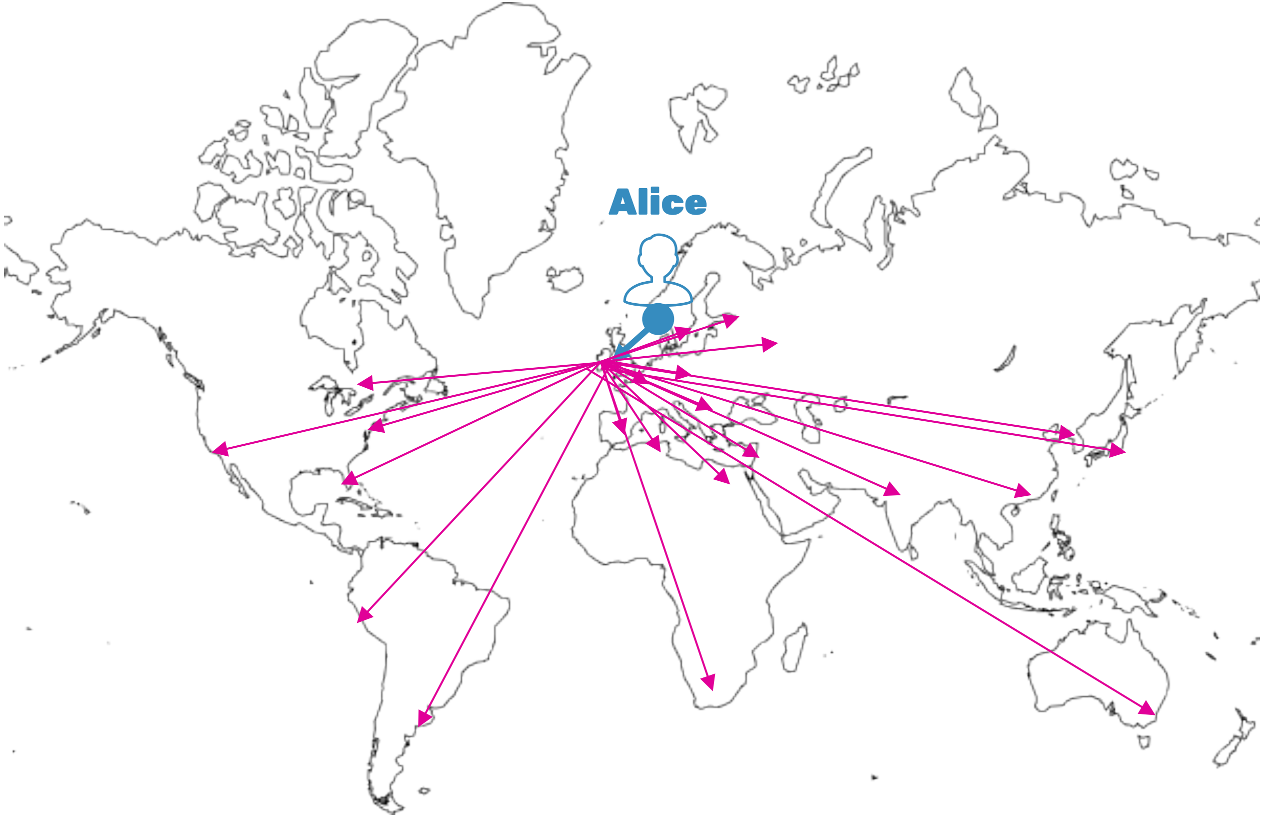
CASE STUDY

YouTube v2.0

VIDEO STREAMING

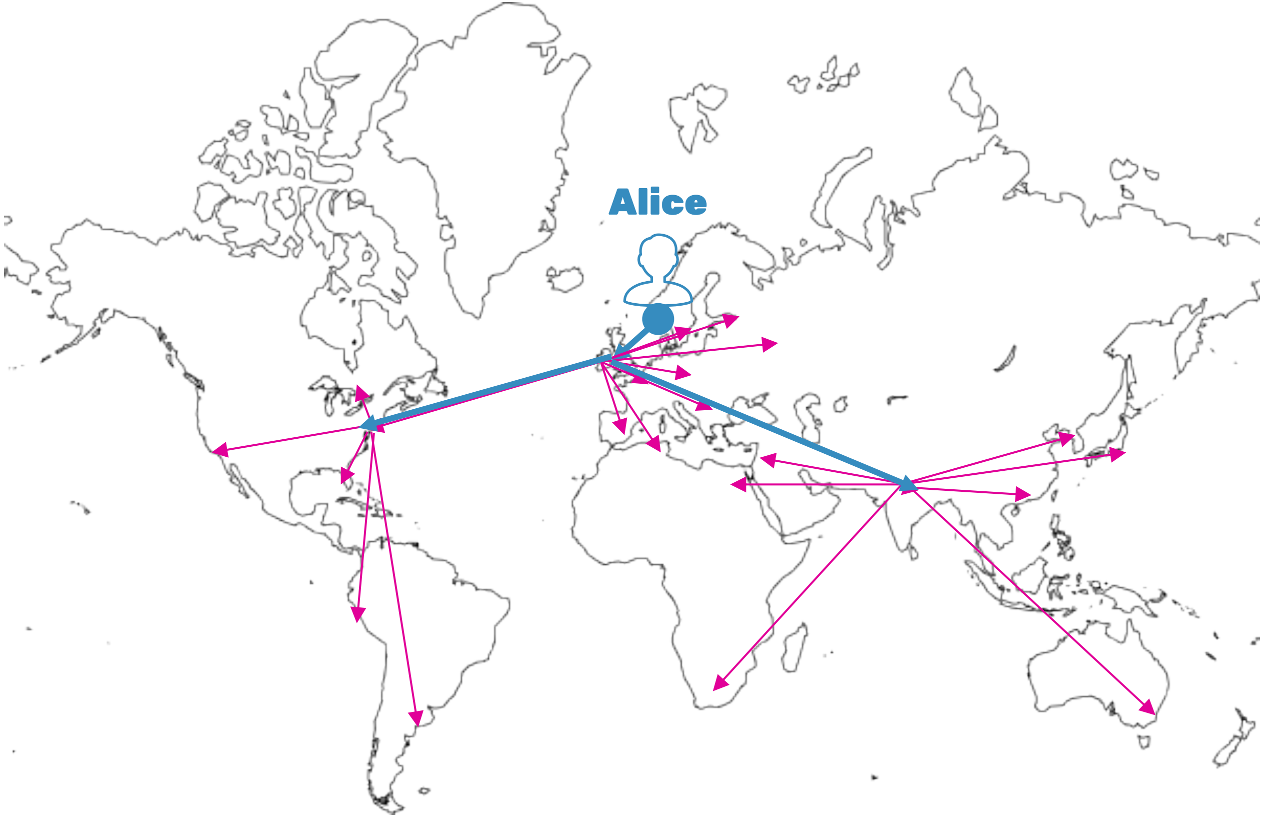
VIDEO STREAMING

VIDEO STREAMING



VIDEO STREAMING

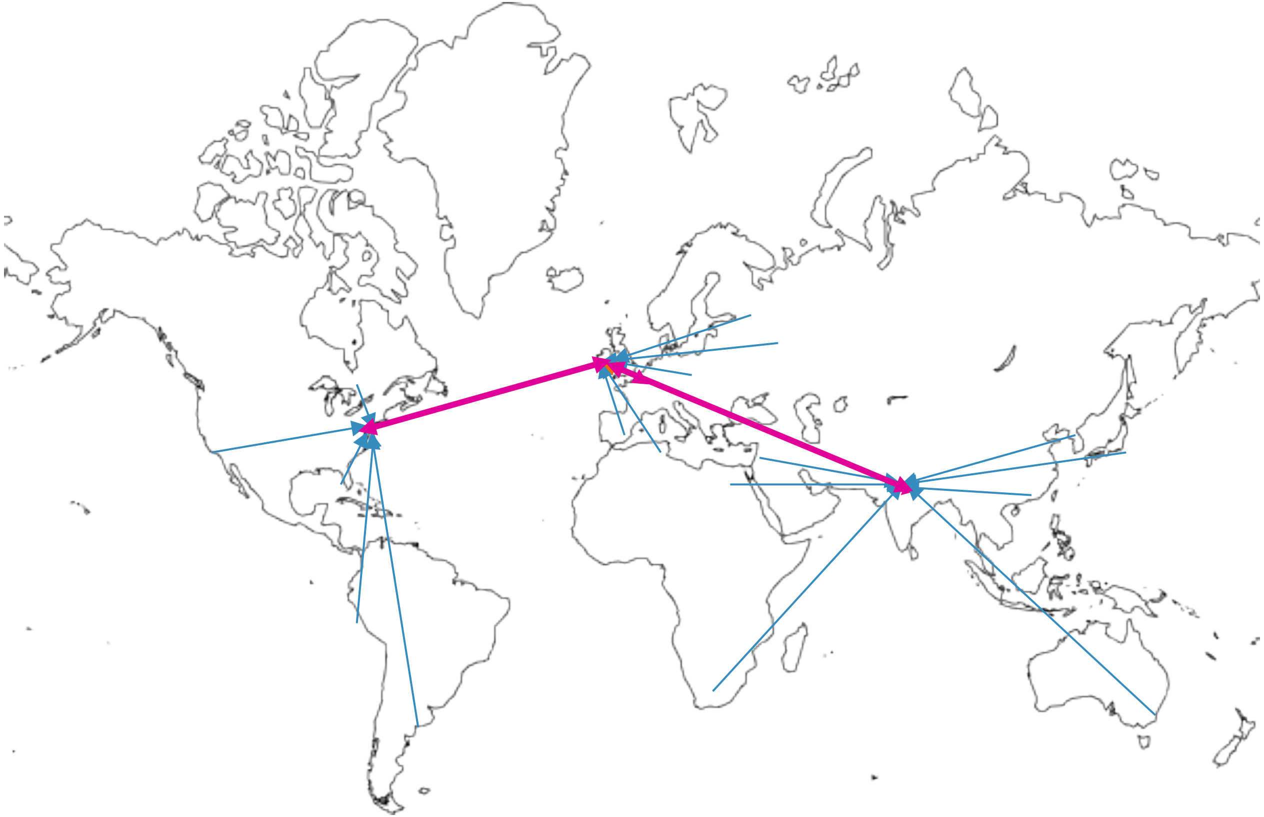
SINGLE WRITER REPLICATION



LET’S ADD A VIDEO VIEW COUNTER

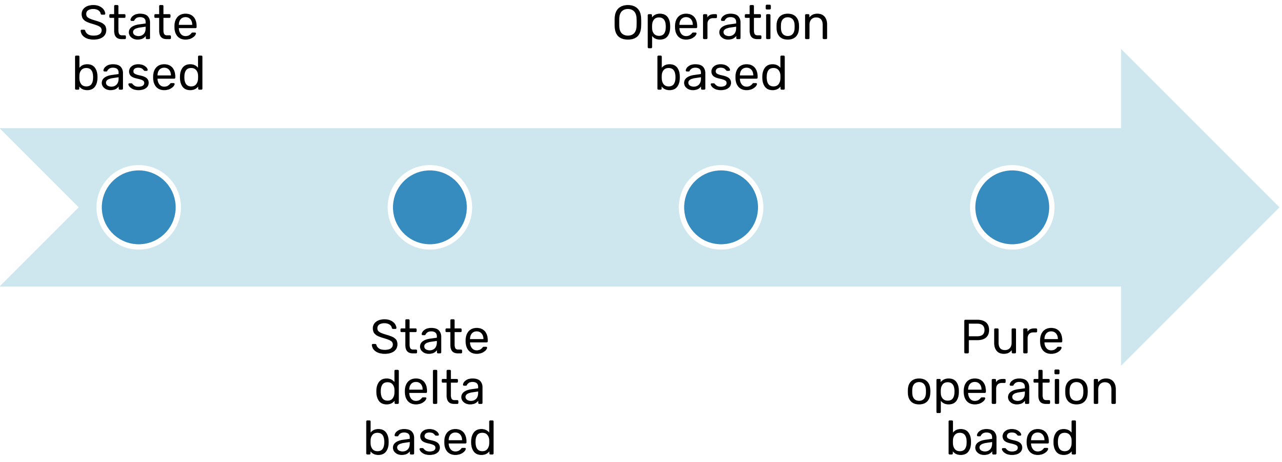
PAGE VIEWS

MULTI-MASTER REPLICATION



CAN WE SYNCHRONIZE DATA SAFELY WITHOUT A NEED FOR CONSENSUS?

CONFLICT-FREE REPLICATED DATA TYPES



Simple protocol

Complex structures

State based

State delta based

Operation based

Pure operation based

Complex protocol

Simple structures

USE CASES

1. *Sync data over the network with large latencies*
2. *Sync data between periodically disconnected devices*
3. *Navigation*
4. *Chat applications*
5. *Collaborative text editing*
6. *Mobile advertising*
7. *Edge computing*

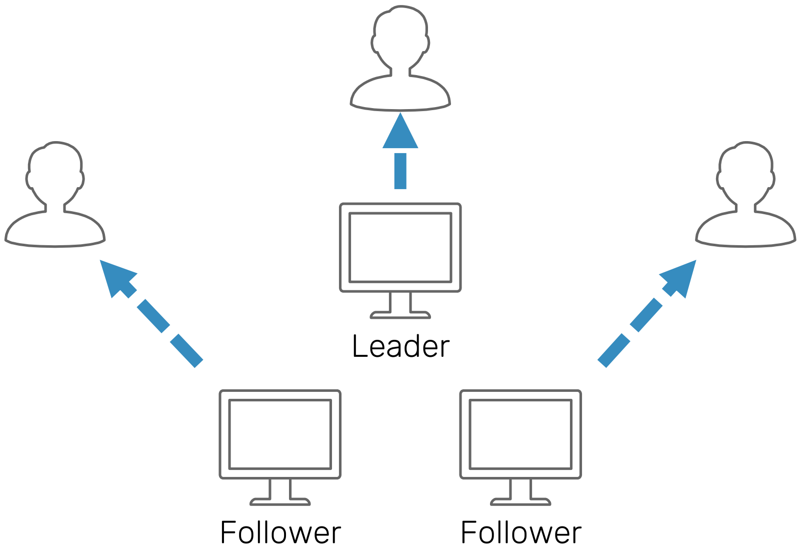
CRDT IN THE WILD

Riak	database	*operation based*
AntidoteDB	database	*operation based*
Amazon DynamoDB	database	
Azure CosmosDB	database (multi-master)	*custom (state based)*
Redis CRDB	database	*state based*
Lasp	Erlang library	*delta/state based*
Akka.DistributedData	JVM/.NET library	*delta/state based*
Eventuate	JVM library	*operation based*
Roshi	Go library	*state based*
Automerge	Javascript library	*operation based*

STATE BASED CRDT 101

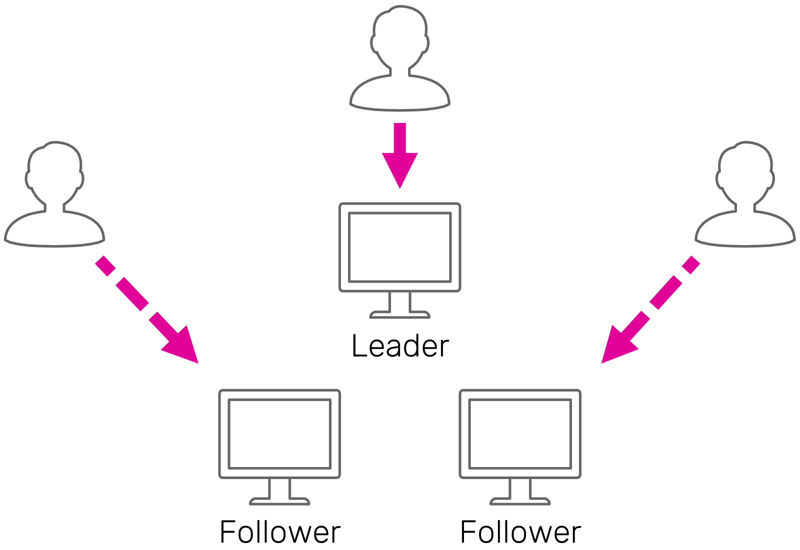
LEADER FOLLOWER REPLICATION

READS



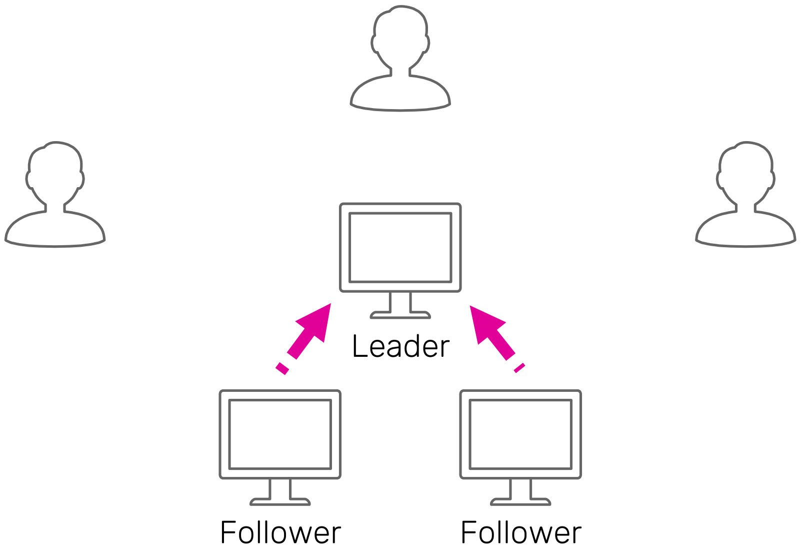
LEADER FOLLOWER REPLICATION

WRITES



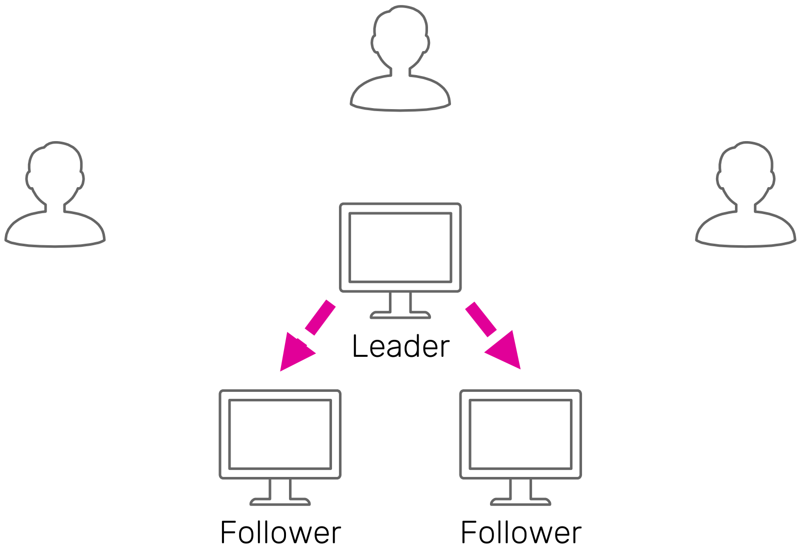
LEADER FOLLOWER REPLICATION

WRITES

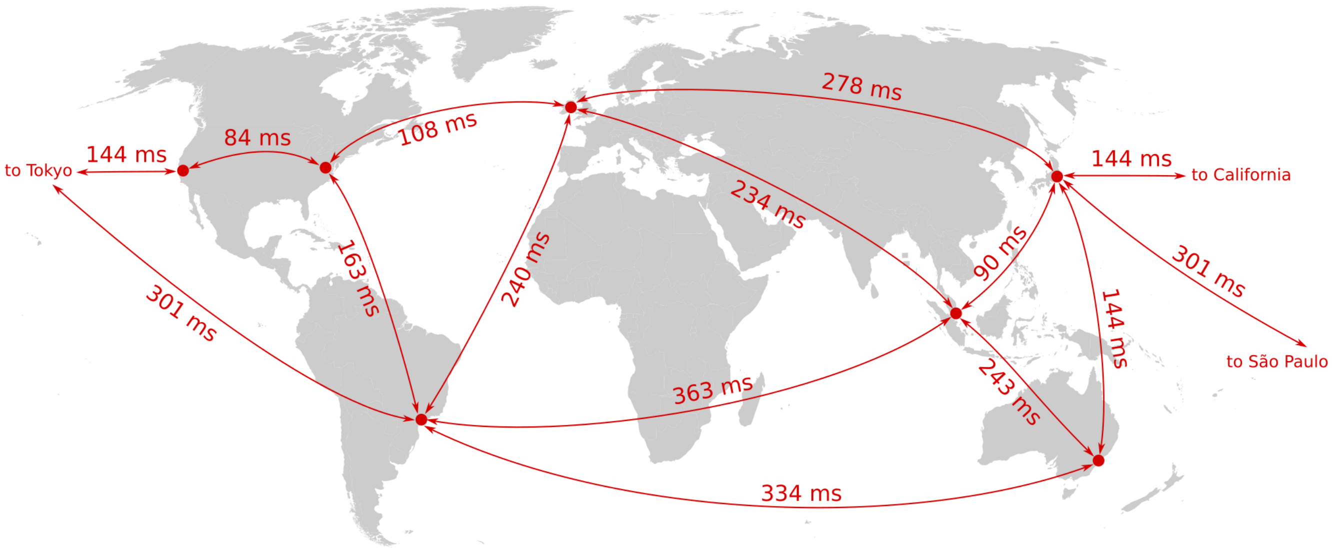


LEADER FOLLOWER REPLICATION

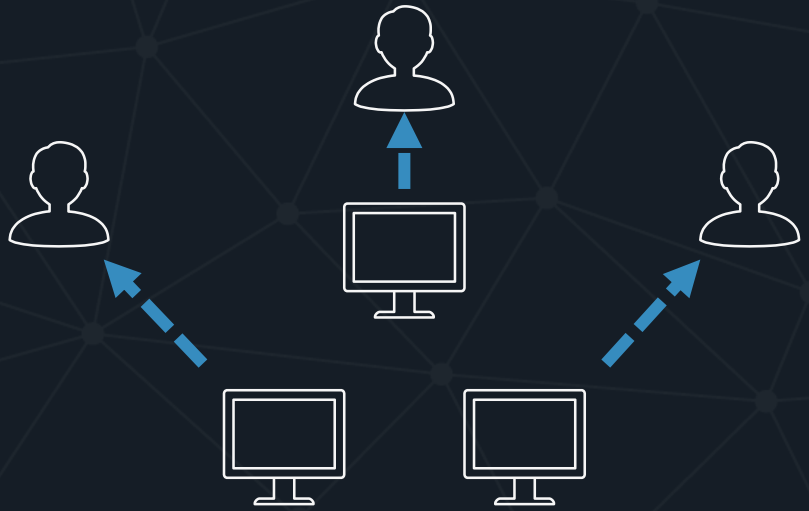
WRITES



ROUND TRIP TIMES

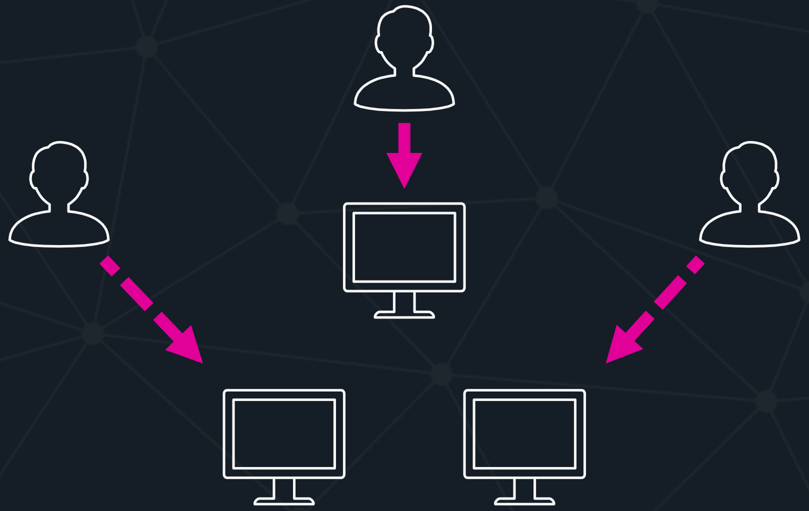


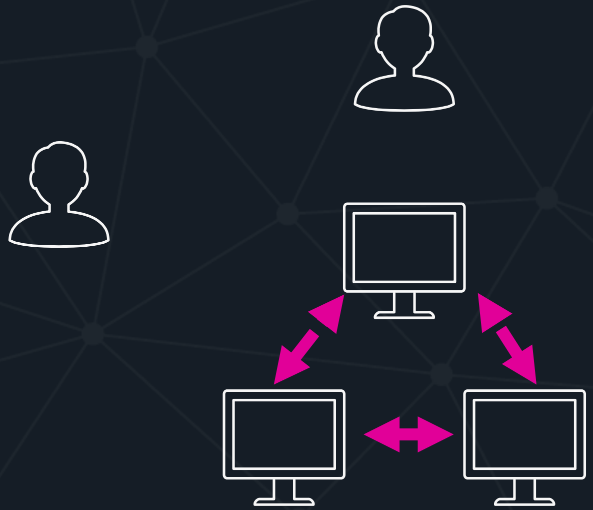
Source: https://speakerdeck.com/ept/local-first-software-returning-data-ownership-to-users



MASTERLESS REPLICATION

MASTERLESS REPLICATION





MASTERLESS REPLICATION

WE DON’T NEED CONSENSUS, IF INDIVIDUALLY WE ALWAYS REACH THE SAME CONCLUSION

CONVERGENCE

HOW TO KEEP THINGS IN SYNC

1. Commutative: $x \bullet y = y \bullet x$
2. Associative: $(x \bullet y) \bullet z = x \bullet (y \bullet z)$
3. Idempotent: $x \bullet x = x$

CASE #1

DISTRIBUTED VIEW COUNTER

G-COUNTER

GROWING ONLY COUNTER

```
const GCounter = {
  empty() {
    return {};
  },
  increment(counter, id) {
    counter[id] = (counter[id] || 0) + 1;
  },
  value(counter) {
    return Object.values(counter).reduce((sum, x) => sum + x, 0);
  },
  merge(existing, incoming) {
    Object.keys(incoming).forEach(id => {
      const incomingVal = incoming[id];
      const existingVal = existing[id] || 0;
      existing[id] = Math.max(incomingVal, existingVal);
    });
  }
};
```

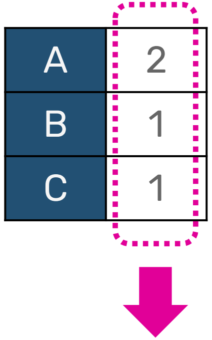
G-COUNTER

VALUE

A	2
B	1
C	1

G-COUNTER

VALUE



A	2
B	1
C	1

(2 + 1 + 1) => 4

G-COUNTER

GROWING ONLY COUNTER

```
const GCounter = {
  empty() {
    return {};
  },
  increment(counter, id) {
    counter[id] = (counter[id] || 0) + 1;
  },
  value(counter) {
    return Object.values(counter).reduce((sum, x) => sum + x, 0);
  },
  merge(existing, incoming) {
    Object.keys(incoming).forEach(id => {
      const incomingVal = incoming[id];
      const existingVal = existing[id] || 0;
      existing[id] = Math.max(incomingVal, existingVal);
    });
  }
};
```

G-COUNTER

MERGE

A	1
B	3

MAX

A	2
B	1
C	1

G-COUNTER

MERGE

A	1
B	3

MAX

A	2
B	1
C	1

MAX(1, 2)

MAX(3, 1)

MAX(0, 1)

A	2
B	3
C	1

G-COUNTER

GROWING ONLY COUNTER

```javascript
const GCounter = {
  empty() {
    return {};
  },
  increment(counter, id) {
    counter[id] = (counter[id] || 0) + 1;
  },
  value(counter) {
    return Object.values(counter).reduce((sum, x) => sum + x, 0);
  },
  merge(existing, incoming) {
    Object.keys(incoming).forEach(id => {
      const incomingVal = incoming[id];
      const existingVal = existing[id] || 0;
      existing[id] = Math.max(incomingVal, existingVal);
    });
  }
};
```

CASE #2

DISTRIBUTED “LIKE” COUNTER

PN-COUNTER

POSITIVE NEGATIVE COUNTER

```
const PNCounter = {
  empty() {
    return { inc: GCounter.empty(), dec: GCounter.empty() };
  },
  increment(counter, id) {
    GCounter.increment(counter.inc, id);
  },
  decrement(counter, id) {
    GCounter.increment(counter.dec, id);
  },
  value(counter) {
    return GCounter.value(counter.inc) - GCounter.value(counter.dec);
  },
  merge(existing, incoming) {
    GCounter.merge(existing.inc, incoming.inc);
    GCounter.merge(existing.dec, incoming.dec);
  }
};
```

PN-COUNTER

POSITIVE NEGATIVE COUNTER

```
const PNCounter = {
  empty() {
    return { inc: GCounter.empty(), dec: GCounter.empty() };
  },
  increment(counter, id) {
    GCounter.increment(counter.inc, id);
  },
  decrement(counter, id) {
    GCounter.increment(counter.dec, id);
  },
  value(counter) {
    return GCounter.value(counter.inc) - GCounter.value(counter.dec);
  },
  merge(existing, incoming) {
    GCounter.merge(existing.inc, incoming.inc);
    GCounter.merge(existing.dec, incoming.dec);
  }
};
```

CASE #3

VOTING SYSTEM – SURVEY PARTICIPATION TRACKING

G-SET

GROWING ONLY SET

```
const GSet = {
  empty() {
    return {};
  },
  add(set, item) {
    set[item] = 1;
  },
  value(set) {
    return Object.keys(set);
  },
  merge(existing, incoming) {
    Object.keys(incoming).forEach(item => existing[item] = 1);
  }
};
```

G-SET

GROWING ONLY SET

```
const GSet = {
  empty() {
    return {};
  },
  add(set, item) {
    set[item] = 1;
  },
  value(set) {
    return Object.keys(set);
  },
  merge(existing, incoming) {
    Object.keys(incoming).forEach(item => existing[item] = 1);
  }
};
```

SET UNION

CASE #4

DISTRIBUTED SHOPPING CART

G-MAP
+
PN-COUNTER

```
const ORCart = {
  empty() {
    return {};
  },
  add(cart, item, id) {
    var counter = cart[item] || PNCounter.empty();
    PNCounter.increment(counter, id);
    cart[item] = counter;
  },
  remove(cart, item, id) {
    var counter = cart[item] || PNCounter.empty();
    PNCounter.decrement(counter, id);
    cart[item] = counter;
  },
  value(cart) {
    return Object.keys(cart).reduce((result, item) => {
      result[item] = PNCounter.value(cart[item]);
      return result;
    }, {});
  },
  merge(existing, incoming) {
    Object.keys(incoming).forEach(item => {
      const x = existing[item] || PNCounter.empty();
      const y = incoming[item];
      PNCounter.merge(x, y);
      existing[item] = x;
    });
  }
};
```

G-MAP + PN-COUNTER

```
const ORCart = {
  empty() {
    return {};
  },
  add(cart, item, id) {
    var counter = cart[item] || PNCounter.empty();
    PNCounter.increment(counter, id);
    cart[item] = counter;
  },
  remove(cart, item, id) {
    var counter = cart[item] || PNCounter.empty();
    PNCounter.decrement(counter, id);
    cart[item] = counter;
  },
  value(cart) {
    return Object.keys(cart).reduce((result, item) => {
      result[item] = PNCounter.value(cart[item]);
      return result;
    }, {});
  },
  merge(existing, incoming) {
    Object.keys(incoming).forEach(item => {
      const x = existing[item] || PNCounter.empty();
      const y = incoming[item];
      PNCounter.merge(x, y);
      existing[item] = x;
    });
  }
};
```

G-MAP
+
PN-COUNTER

```
const ORCart = {
  empty() {
    return {};
  },
  add(cart, item, id) {
    var counter = cart[item] || PNCounter.empty();
    PNCounter.increment(counter, id);
    cart[item] = counter;
  },
  remove(cart, item, id) {
    var counter = cart[item] || PNCounter.empty();
    PNCounter.decrement(counter, id);
    cart[item] = counter;
  },
  value(cart) {
    return Object.keys(cart).reduce((result, item) => {
      result[item] = PNCounter.value(cart[item]);
      return result;
    }, {});
  },
  merge(existing, incoming) {
    Object.keys(incoming).forEach(item => {
      const x = existing[item] || PNCounter.empty();
      const y = incoming[item];
      PNCounter.merge(x, y);
      existing[item] = x;
    });
  }
};
```

G-MAP
+
PN-COUNTER

```
const ORCart = {
  empty() {
    return {};
  },
  add(cart, item, id) {
    var counter = cart[item] || PNCounter.empty();
    PNCounter.increment(counter, id);
    cart[item] = counter;
  },
  remove(cart, item, id) {
    var counter = cart[item] || PNCounter.empty();
    PNCounter.decrement(counter, id);
    cart[item] = counter;
  },
  value(cart) {
    return Object.keys(cart).reduce((result, item) => {
      result[item] = PNCounter.value(cart[item]);
      return result;
    }, {});
  },
  merge(existing, incoming) {
    Object.keys(incoming).forEach(item => {
      const x = existing[item] || PNCounter.empty();
      const y = incoming[item];
      PNCounter.merge(x, y);
      existing[item] = x;
    });
  }
};
```

CASE #5

TWITTER - LIST OF FOLLOWERS

G-SET

GROWING ONLY SET

1. Can only grow
2. No (safe) semantics for removing values

CAN WE COMPOSE G-SETS INTO REMOVE-AWARE SET?

2P-SET

TWO PHASE SET

```
const TwoPhaseSet = {
  empty() {
    return { add: GSet.empty(), rem: GSet.empty() };
  },
  add(set, item) {
    GSet.add(set.add, item);
  },
  remove(set, item) {
    GSet.add(set.rem, item);
  },
  value(set) {
    return Object.keys(set.add).reduce((result, item) => {
      if (!set.rem[item]) {
        result.push(item);
      }
      return result;
    }, []);
  },
  merge(existing, incoming) {
    GSet.merge(existing.add, incoming.add);
    GSet.merge(existing.rem, incoming.rem);
  }
};
```

2P-SET

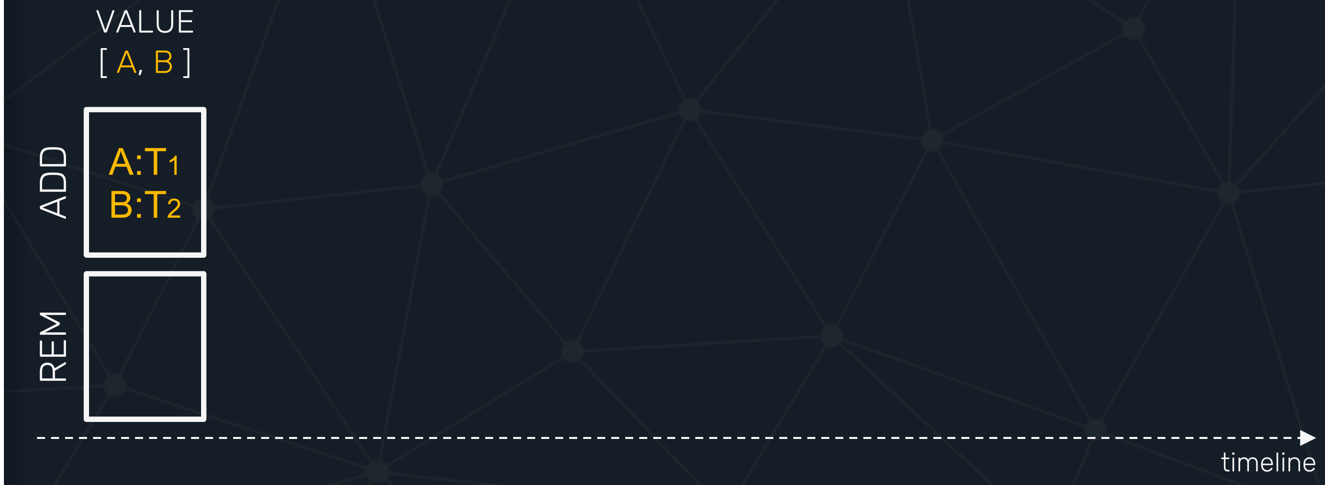
TWO PHASE SET

```
const TwoPhaseSet = {
  empty() {
    return { add: GSet.empty(), rem: GSet.empty() };
  },
  add(set, item) {
    GSet.add(set.add, item);
  },
  remove(set, item) {
    GSet.add(set.rem, item);
  },
  value(set) {
    return Object.keys(set.add).reduce((result, item) => {
      if (!set.rem[item]) {
        result.push(item);
      }
      return result;
    }, []);
  },
  merge(existing, incoming) {
    GSet.merge(existing.add, incoming.add);
    GSet.merge(existing.rem, incoming.rem);
  }
};
```

2P-SET

ISSUES

1. Once removed, element cannot be reinserted.
2. Requires tombstones to be always present.



VALUE

[A, B]

ADD

A:T_1

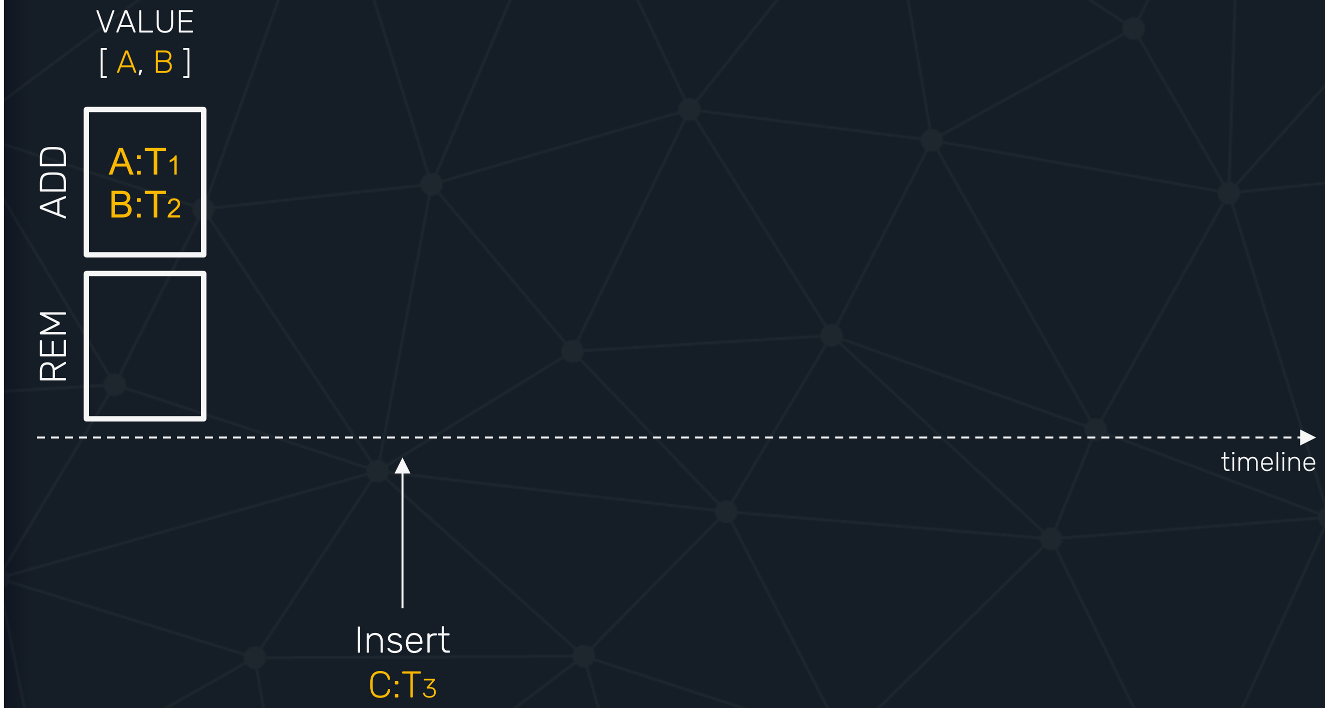
B:T_2

REM

timeline

OBSERVED REMOVE SET

OBSERVED REMOVE SET



VALUE

[A, B]

ADD

A:T_1

B:T_2

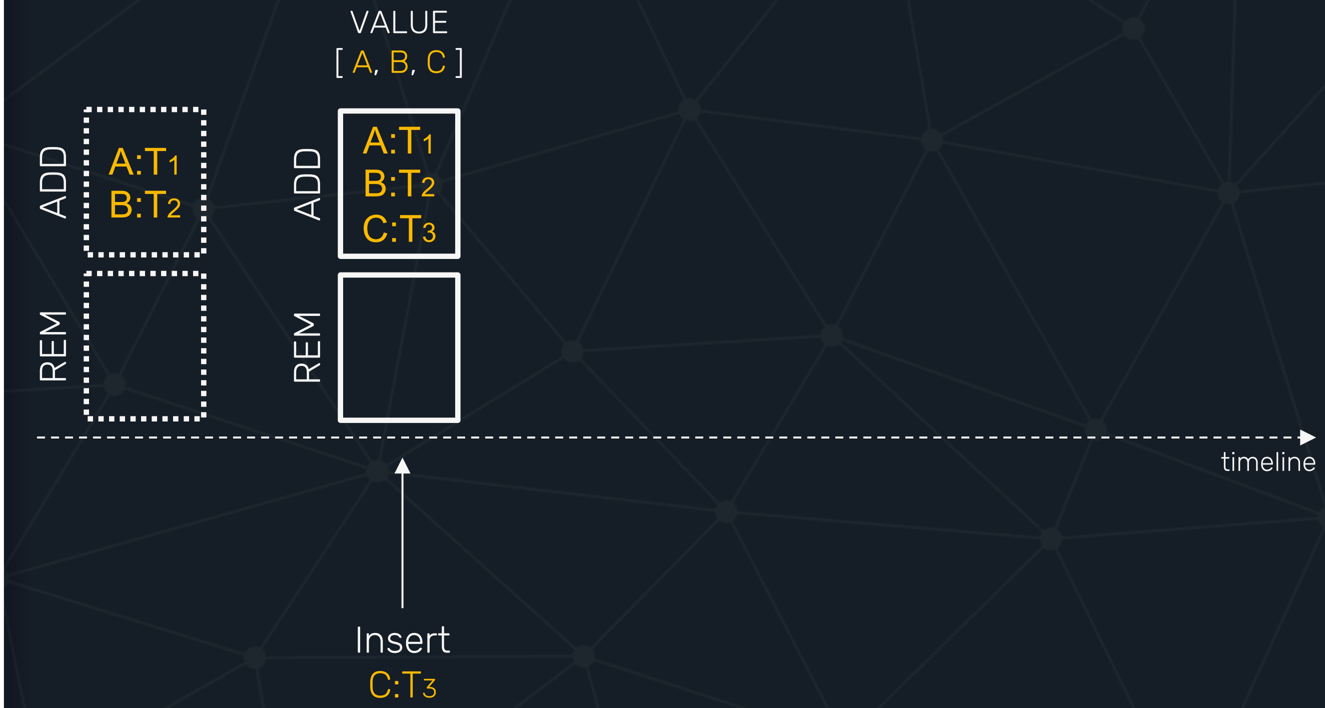
REM

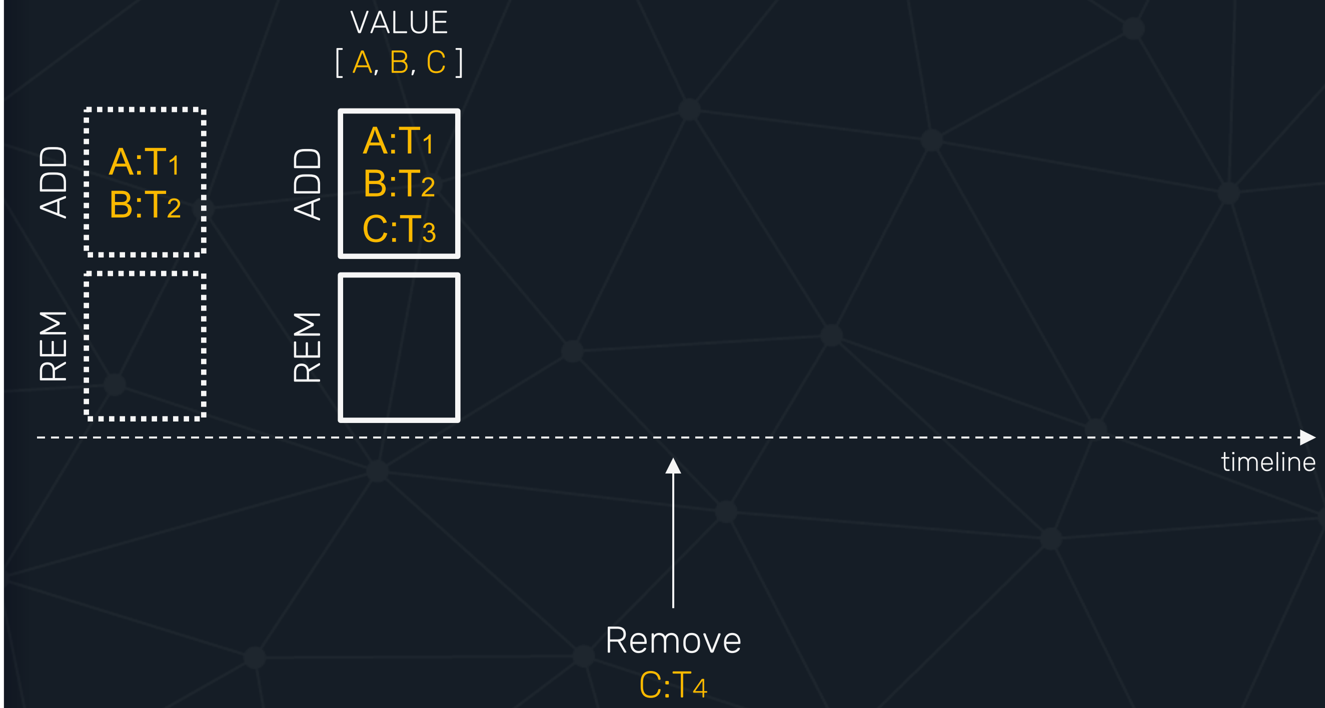
timeline

Insert

C:T_3

OBSERVED REMOVE SET





VALUE

[A, B, C]

ADD

$A:T_1$

$B:T_2$

REM

ADD

$A:T_1$

$B:T_2$

$C:T_3$

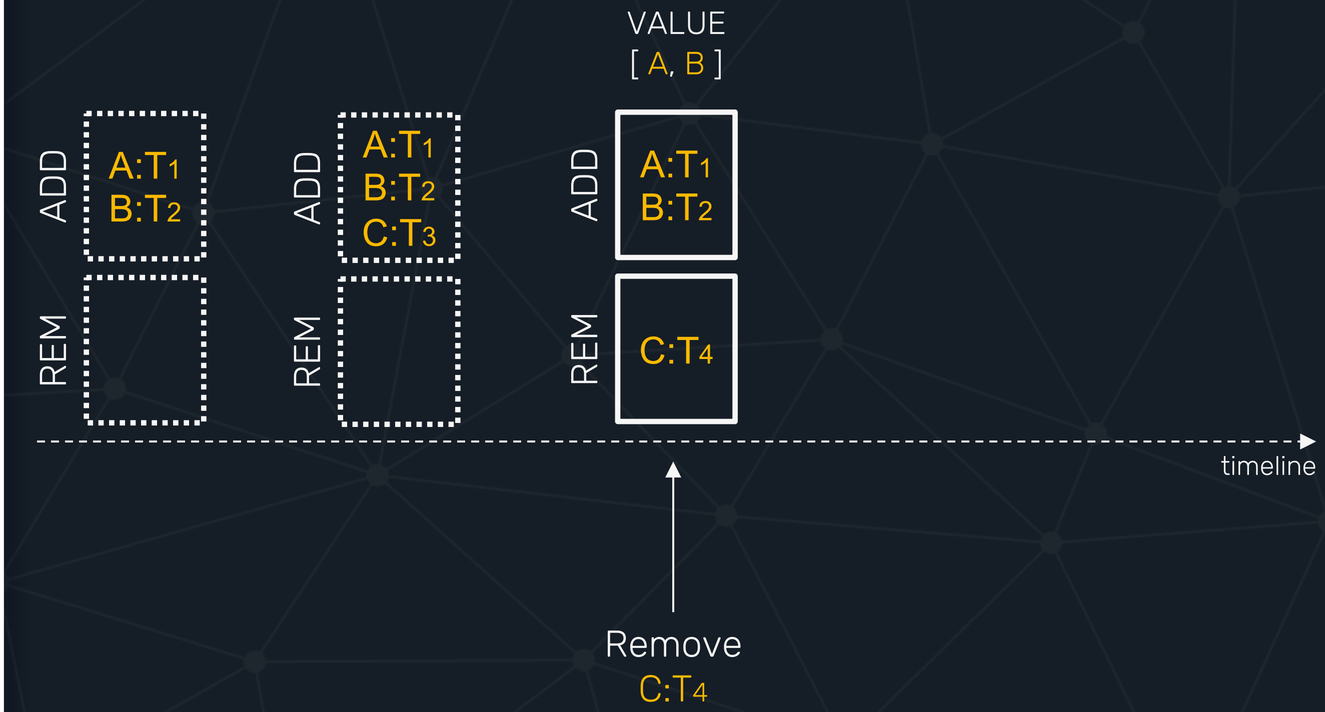
REM

timeline

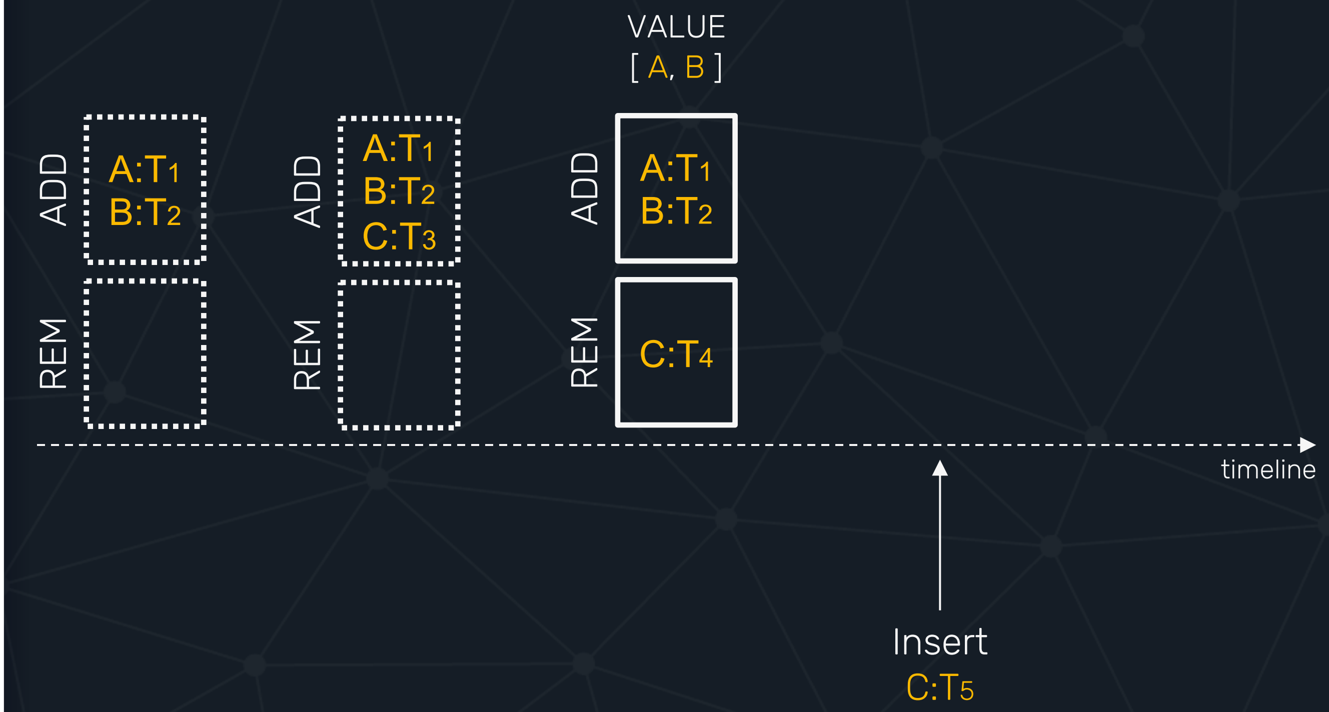
Remove

$C:T_4$

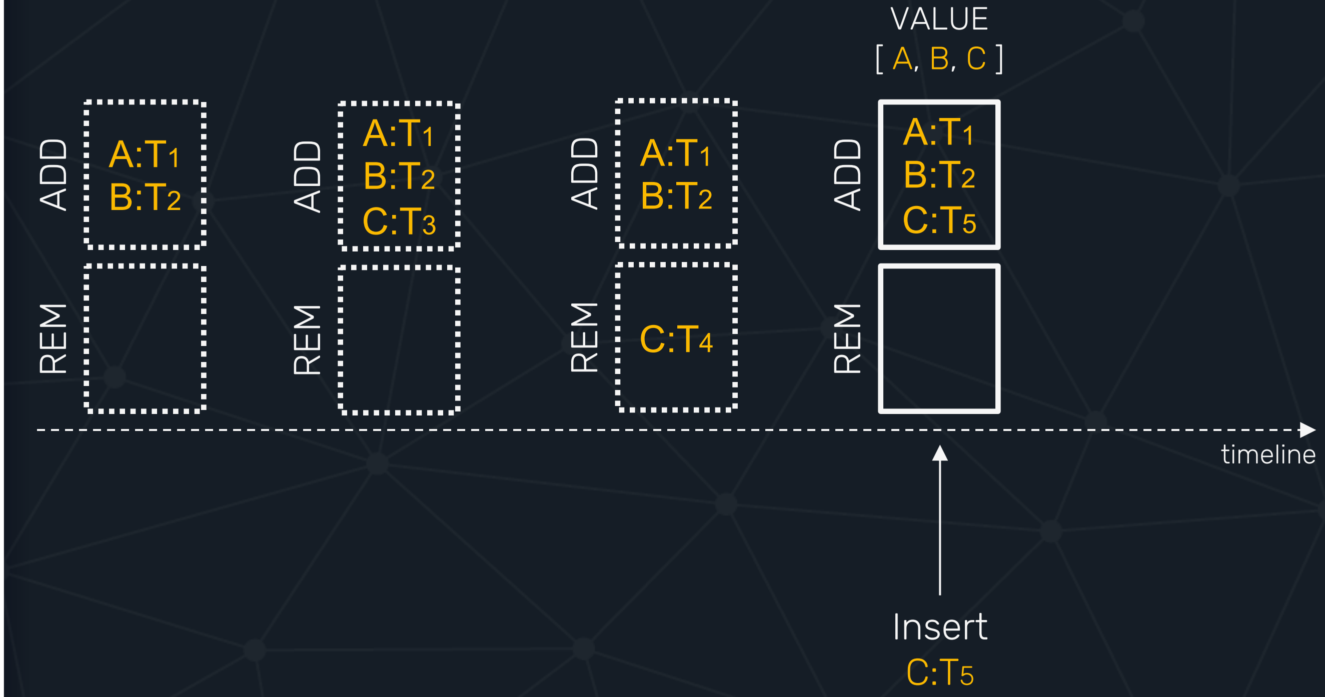
OBSERVED REMOVE SET



OBSERVED REMOVE SET



OBSERVED REMOVE SET



ADD

A:T_1
B:T_2

REM

ADD

A:T_1
B:T_2
C:T_3

REM

ADD

A:T_1
B:T_2

REM

C:T_4

VALUE
[A, B, C]

ADD

A:T_1
B:T_2
C:T_5

REM

timeline

Insert
C:T_5

OBSERVED REMOVE SET

LAST WRITE WINS SET

```
const LWWSet = {
  empty() {
    return { add: {}, rem: {} };
  },
  add(set, item) {
    set.add[item] = (new Date()).getTime();
    set.rem[item] = undefined;
  },
  remove(set, item) {
    set.add[item] = undefined;
    set.rem[item] = (new Date()).getTime();
  },
  value(set) {
    return Object.keys(set.add).reduce((result, item) => {
      const addedAt = set.add[item];
      const removedAt = set.rem[item] || 0;
      if (addedAt >= removedAt) {
        result.push(item);
      }
      return result;
    }, []);
  },
  merge(existing, incoming) {
    function partialMerge(a, b) {
      Object.keys(b).forEach(item => {
        a[item] = Math.max(b[item], a[item] || 0);
      });
    }
    partialMerge(existing.add, incoming.add);
    partialMerge(existing.rem, incoming.rem);
  }
};
```

LAST WRITE WINS SET

```
const LWWSet = {
  empty() {
    return { add: {}, rem: {} };
  },
  add(set, item) {
    set.add[item] = (new Date()).getTime();
    set.rem[item] = undefined;
  },
  remove(set, item) {
    set.add[item] = undefined;
    set.rem[item] = (new Date()).getTime();
  },
  value(set) {
    return Object.keys(set.add).reduce((result, item) => {
      const addedAt = set.add[item];
      const removedAt = set.rem[item] || 0;
      if (addedAt >= removedAt) {
        result.push(item);
      }
      return result;
    }, []);
  },
  merge(existing, incoming) {
    function partialMerge(a, b) {
      Object.keys(b).forEach(item => {
        a[item] = Math.max(b[item], a[item] || 0);
      });
    }
    partialMerge(existing.add, incoming.add);
    partialMerge(existing.rem, incoming.rem);
  }
};
```

LAST WRITE WINS SET

```
const LWWSet = {
  empty() {
    return { add: {}, rem: {} };
  },
  add(set, item) {
    set.add[item] = (new Date()).getTime();
    set.rem[item] = undefined;
  },
  remove(set, item) {
    set.add[item] = undefined;
    set.rem[item] = (new Date()).getTime();
  },
  value(set) {
    return Object.keys(set.add).reduce((result, item) => {
      const addedAt = set.add[item];
      const removedAt = set.rem[item] || 0;
      if (addedAt >= removedAt) {
        result.push(item);
      }
      return result;
    }, []);
  },
  merge(existing, incoming) {
    function partialMerge(a, b) {
      Object.keys(b).forEach(item => {
        a[item] = Math.max(b[item], a[item] || 0);
      });
    }
    partialMerge(existing.add, incoming.add);
    partialMerge(existing.rem, incoming.rem);
  }
};
```

SYSTEM CLOCK TIMESTAMP

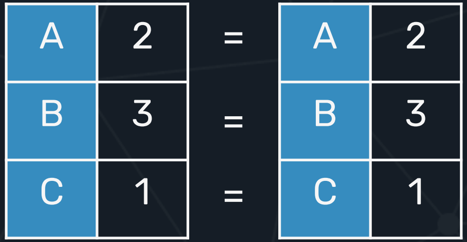
1. Requires clocks to be in sync.
2. Doesn’t say anything about concurrent updates.

VECTOR CLOCKS

TO THE RESCUE

PARTIAL ORDERING

Equals



A	2	=	A	2
B	3	=	B	3
C	1	=	C	1

Equals

A	2	=	A	2
B	3	=	B	3
C	1	=	C	1

Greater than

A	2	=	A	2
B	3	=	B	3
C	2	>	C	1

PARTIAL ORDERING

PARTIAL ORDERING

Equals

A	2	=	A	2
B	3	=	B	3
C	1	=	C	1

Greater than

A	2	=	A	2
B	3	=	B	3
C	2	>	C	1

Less than

A	2	=	A	2
B	2	<	B	3
C	1	=	C	1

Equals

A	2	=	A	2
B	3	=	B	3
C	1	=	C	1

Greater than

A	2	=	A	2
B	3	=	B	3
C	2	>	C	1

Less than

A	2	=	A	2
B	2	<	B	3
C	1	=	C	1

Concurrent

A	1	<	A	2
B	3	>	B	2
C	1	=	C	1

PARTIAL ORDERING

VECTOR CLOCK

```
const VectorClock = {
  merge: GCounter.merge,
  increment: GCounter.increment,
  compare(a, b) {
    function partialCompare(result, id) {
      if (result === null) return result;

      const aval = a[id] || 0;
      const bval = b[id] || 0;

      if (aval > bval) {
        if (result === -1) return null;
        else return 1;
      } else if (aval < bval) {
        if (result === 1) return null;
        else return -1;
      } else return result;
    }

    var result = Object.keys(a).reduce(partialCompare, 0);
    return Object.keys(b).reduce(partialCompare, result);
  }
};
```

ADD-WINS OBSERVED REMOVE SET

```
const AWORSet = {
  empty() {
    return { add: {}, rem: {} };
  },
  add(set, item, id) {
    var clock = set.add[item] || set.rem[item] || {};
    VectorClock.increment(clock, id);
    set.rem[item] = undefined;
    set.add[item] = clock;
  },
  remove(set, item, id) {
    var clock = set.add[item] || set.rem[item] || {};
    VectorClock.increment(clock, id);
    set.add[item] = undefined;
    set.rem[item] = clock;
  },
  value(set) {
    return Object.keys(set.add).reduce((result, item) => {
      var addClock = set.add[item] || {};
      var remClock = set.rem[item] || {};

      if (VectorClock.compare(addClock, remClock) !== -1) {
        result.push(item);
      }
      return result;
    }, []);
  },
  merge(existing, incoming) {
    function partialMerge(a, b) {
      Object.keys(b).forEach(item => {
        var aclock = a[item] || {};
        var bclock = b[item] || {};
        VectorClock.merge(aclock, bclock);
        a[item] = aclock;
      });
    }
    partialMerge(existing.add, incoming.add);
    partialMerge(existing.rem, incoming.rem);
  }
};
```

ADD-WINS OBSERVED REMOVE SET

```
const AWORSet = {
  empty() {
    return { add: {}, rem: {} };
  },
  add(set, item, id) {
    var clock = set.add[item] || set.rem[item] || {};
    VectorClock.increment(clock, id);
    set.rem[item] = undefined;
    set.add[item] = clock;
  },
  remove(set, item, id) {
    var clock = set.add[item] || set.rem[item] || {};
    VectorClock.increment(clock, id);
    set.add[item] = undefined;
    set.rem[item] = clock;
  },
  value(set) {
    return Object.keys(set.add).reduce((result, item) => {
      var addClock = set.add[item] || {};
      var remClock = set.rem[item] || {};

      if (VectorClock.compare(addClock, remClock) !== -1) {
        result.push(item);
      }
      return result;
    }, []);
  },
  merge(existing, incoming) {
    function partialMerge(a, b) {
      Object.keys(b).forEach(item => {
        var aclock = a[item] || {};
        var bclock = b[item] || {};
        VectorClock.merge(aclock, bclock);
        a[item] = aclock;
      });
    }
    partialMerge(existing.add, incoming.add);
    partialMerge(existing.rem, incoming.rem);
  }
};
```

ADD-WINS OBSERVED REMOVE SET

```
const AWORSet = {
  empty() {
    return { add: {}, rem: {} };
  },
  add(set, item, id) {
    var clock = set.add[item] || set.rem[item] || {};
    VectorClock.increment(clock, id);
    set.rem[item] = undefined;
    set.add[item] = clock;
  },
  remove(set, item, id) {
    var clock = set.add[item] || set.rem[item] || {};
    VectorClock.increment(clock, id);
    set.add[item] = undefined;
    set.rem[item] = clock;
  },
  value(set) {
    return Object.keys(set.add).reduce((result, item) => {
      var addClock = set.add[item] || {};
      var remClock = set.rem[item] || {};

      if (VectorClock.compare(addClock, remClock) !== -1) {
        result.push(item);
      }
      return result;
    }, []);
  },
  merge(existing, incoming) {
    function partialMerge(a, b) {
      Object.keys(b).forEach(item => {
        var aclock = a[item] || {};
        var bclock = b[item] || {};
        VectorClock.merge(aclock, bclock);
        a[item] = aclock;
      });
    }
    partialMerge(existing.add, incoming.add);
    partialMerge(existing.rem, incoming.rem);
  }
};
```

ADD-WINS OBSERVED REMOVE SET

```
const AWORSet = {
  empty() {
    return { add: {}, rem: {} };
  },
  add(set, item, id) {
    var clock = set.add[item] || set.rem[item] || {};
    VectorClock.increment(clock, id);
    set.rem[item] = undefined;
    set.add[item] = clock;
  },
  remove(set, item, id) {
    var clock = set.add[item] || set.rem[item] || {};
    VectorClock.increment(clock, id);
    set.add[item] = undefined;
    set.rem[item] = clock;
  },
  value(set) {
    return Object.keys(set.add).reduce((result, item) => {
      var addClock = set.add[item] || {};
      var remClock = set.rem[item] || {};

      if (VectorClock.compare(addClock, remClock) !== -1) {
        result.push(item);
      }
      return result;
    }, []);
  },
  merge(existing, incoming) {
    function partialMerge(a, b) {
      Object.keys(b).forEach(item => {
        var aclock = a[item] || {};
        var bclock = b[item] || {};
        VectorClock.merge(aclock, bclock);
        a[item] = aclock;
      });
    }
    partialMerge(existing.add, incoming.add);
    partialMerge(existing.rem, incoming.rem);
  }
};
```

WALL CLOCK TIMESTAMP

Overhead:
8 bytes

VECTOR CLOCKS

Overhead:

*Nr. of nodes * (Key size + seq. nr.)*

Examples:
3 × (4 + 8) = 36 bytes

10 × (16 + 8) = 240 bytes

DELTAS

COMPRESSING THE PAYLOAD

DELTAS

State

(G-Counter)

A	2
B	3
C	1

DELTAS

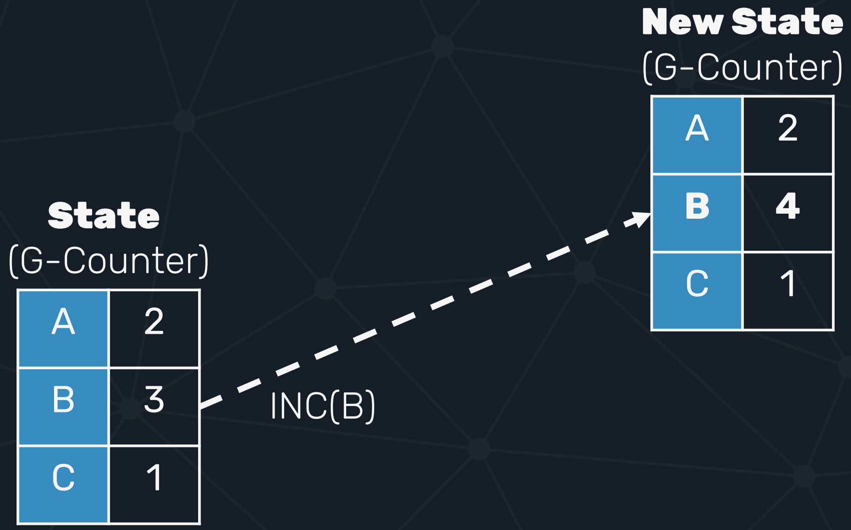
State

(G-Counter)

A	2
B	3
C	1

INC(B)

DELTAS



State
(G-Counter)

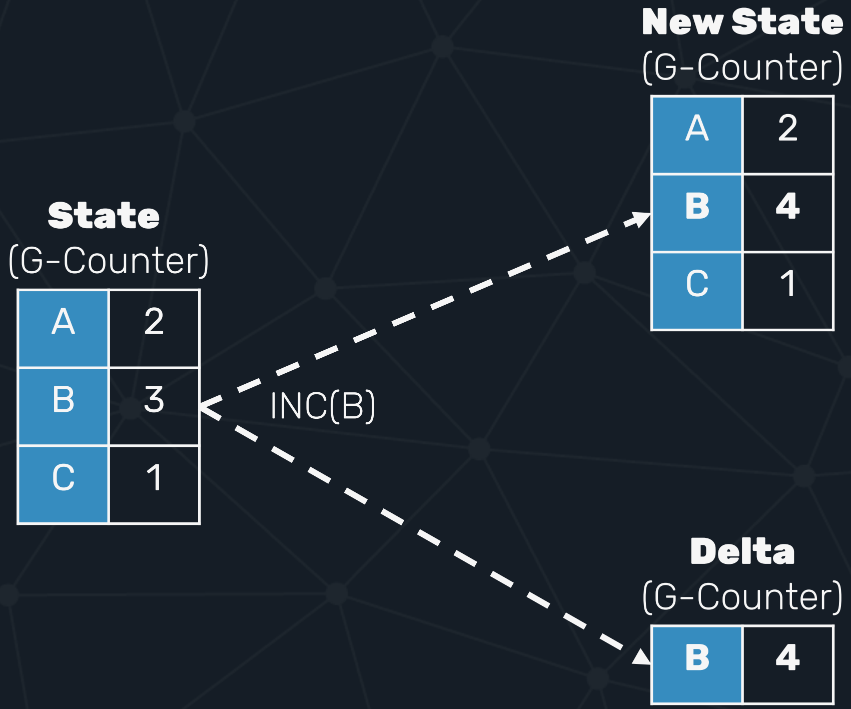
A	2
B	3
C	1

INC(B)

New State
(G-Counter)

A	2
B	**4**
C	1

DELTAS



State
(G-Counter)

A	2
B	3
C	1

INC(B)

New State
(G-Counter)

A	2
B	**4**
C	1

Delta
(G-Counter)

B	**4**

DELTA STATE CRDT

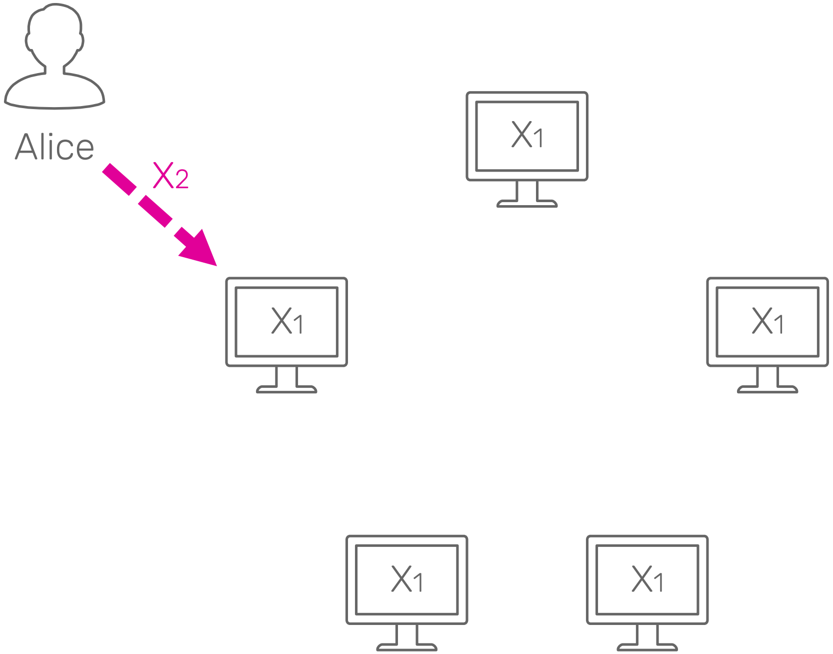
1. Propagate only change set of the state after update.
2. Make sure that all changes have been propagated and received.

APPENDIX #1

ENFORCING CONSISTENCY

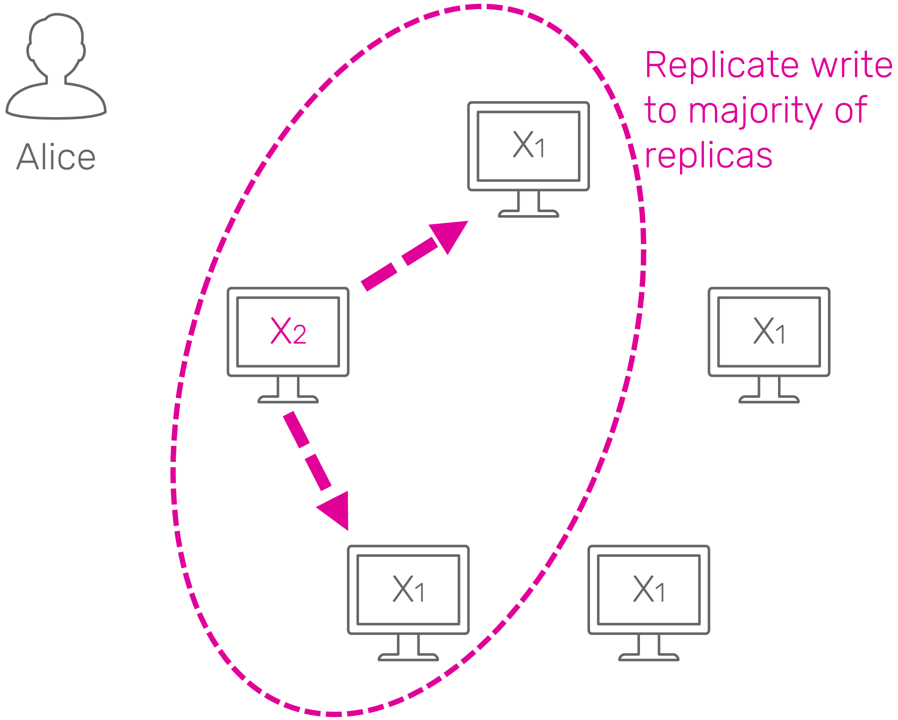
ENFORCING CONSISTENCY

WRITES



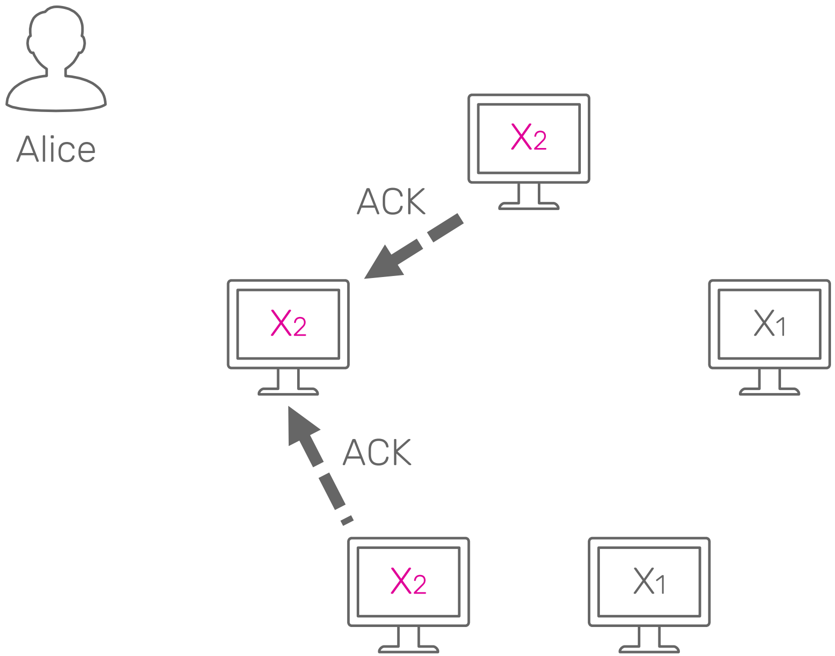
ENFORCING CONSISTENCY

WRITES



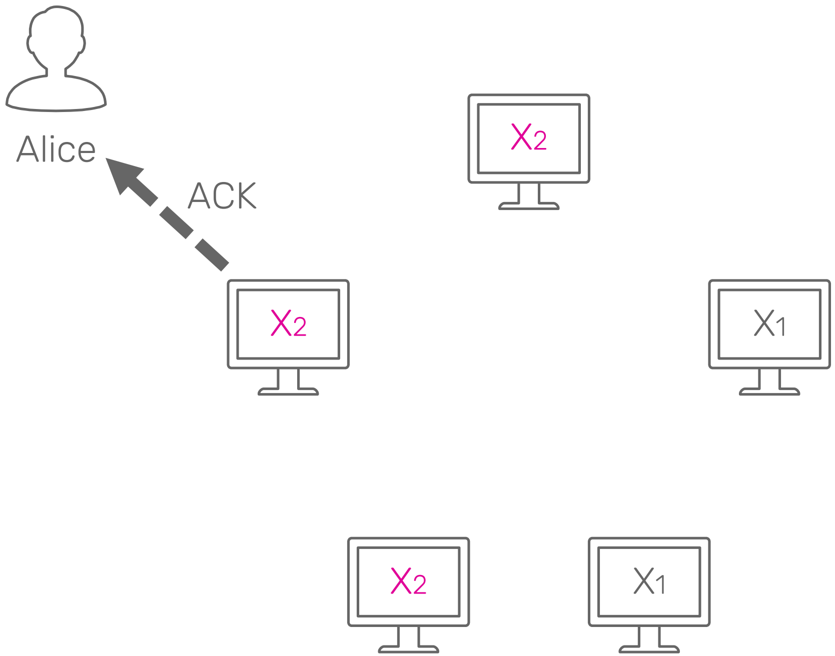
ENFORCING CONSISTENCY

WRITES



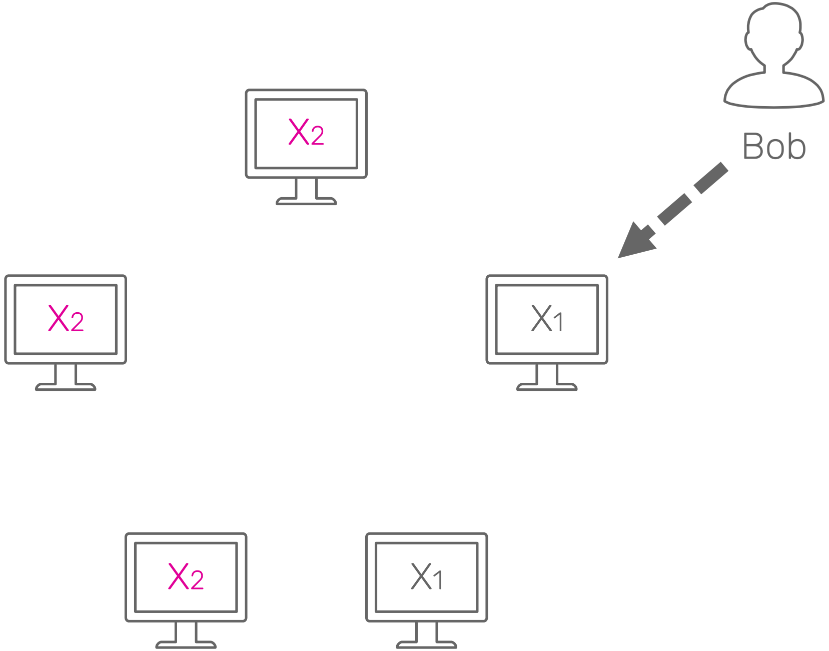
ENFORCING CONSISTENCY

WRITES



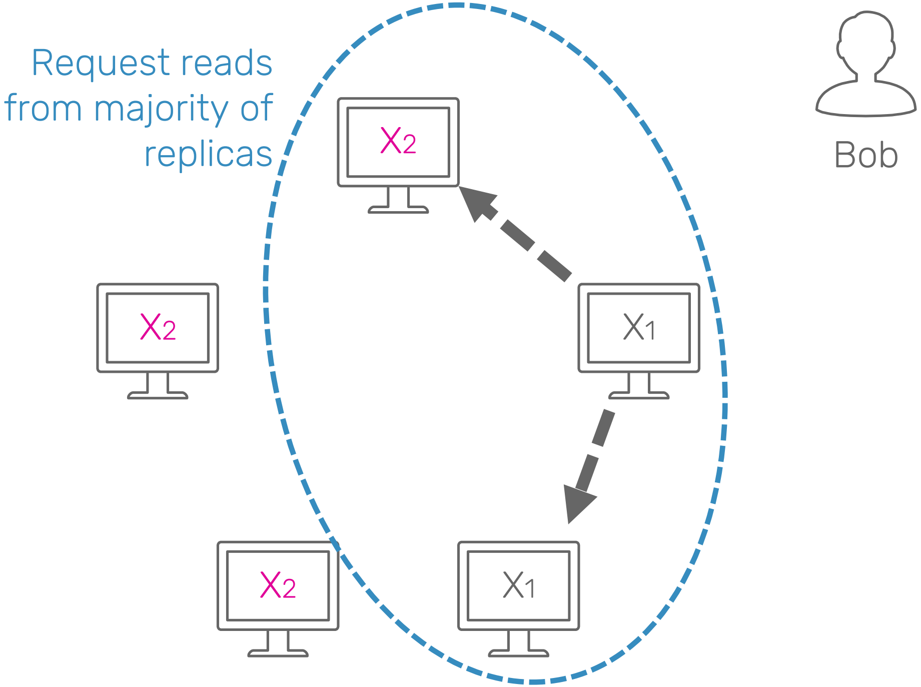
ENFORCING CONSISTENCY

READS



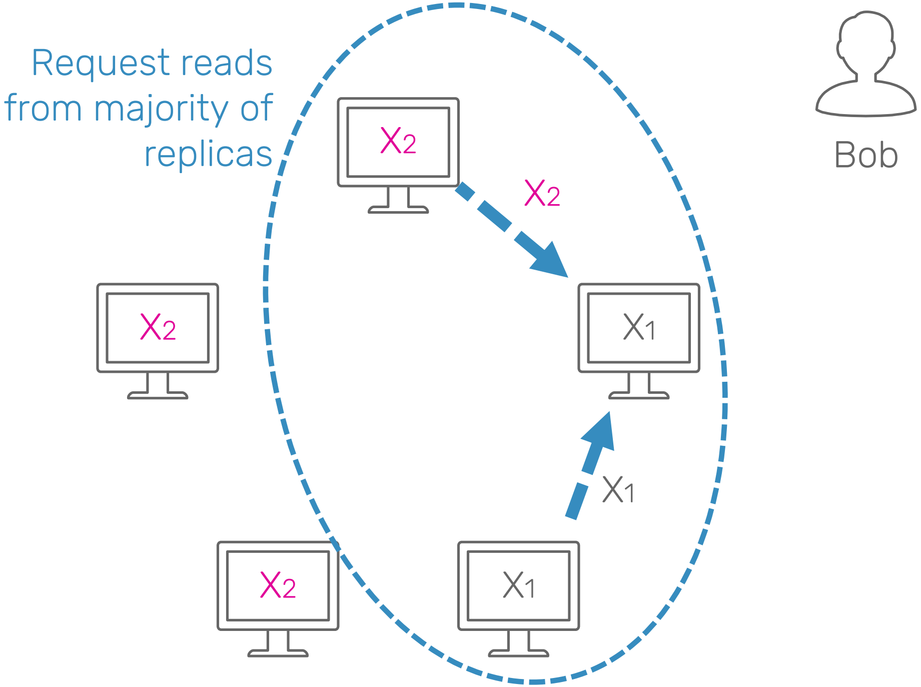
ENFORCING CONSISTENCY

READS



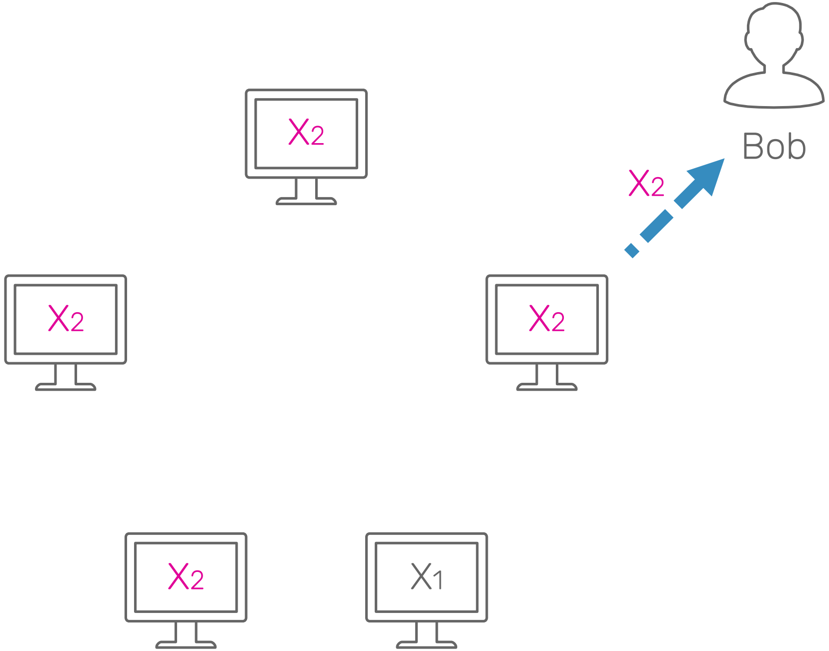
ENFORCING CONSISTENCY

READS



ENFORCING CONSISTENCY

READS



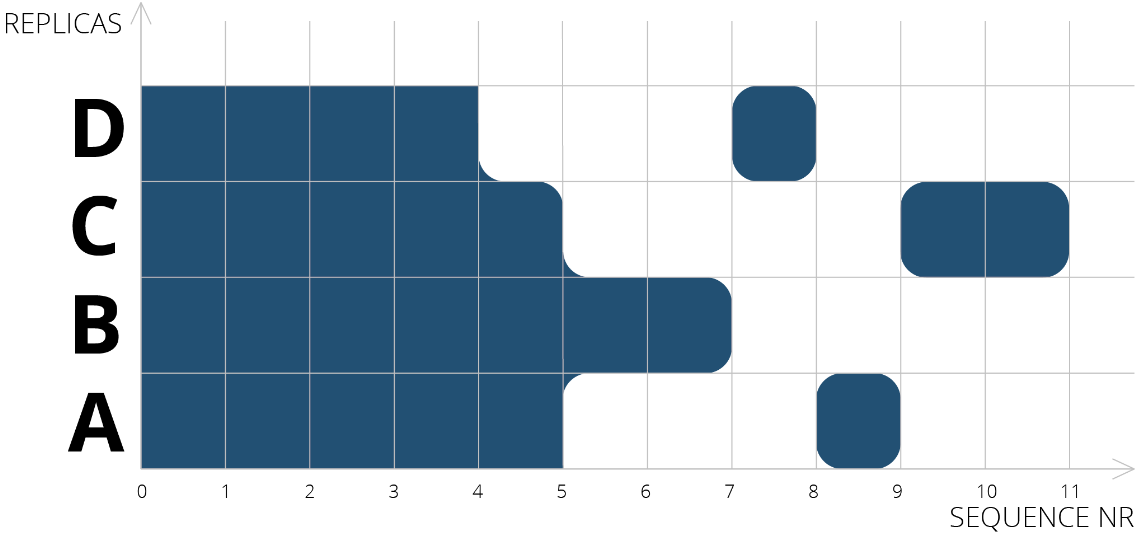
APPENDIX #2

DOTTED VERSION VECTORS

DOT

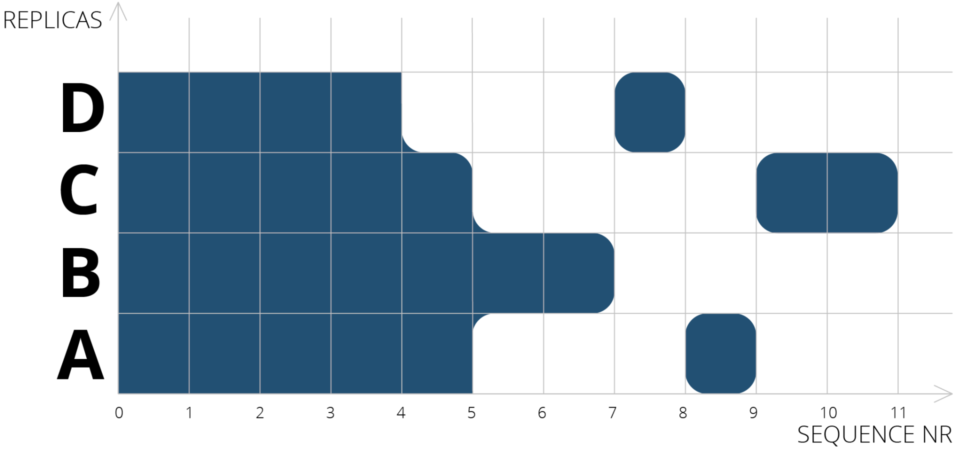
REPLICA ID **A:1** SEQUENCE NR.

DOTTED VERSION VECTORS



DOTTED VERSION VECTORS

REPRESENTATION



D	4
C	5
B	7
A	5

vector clock

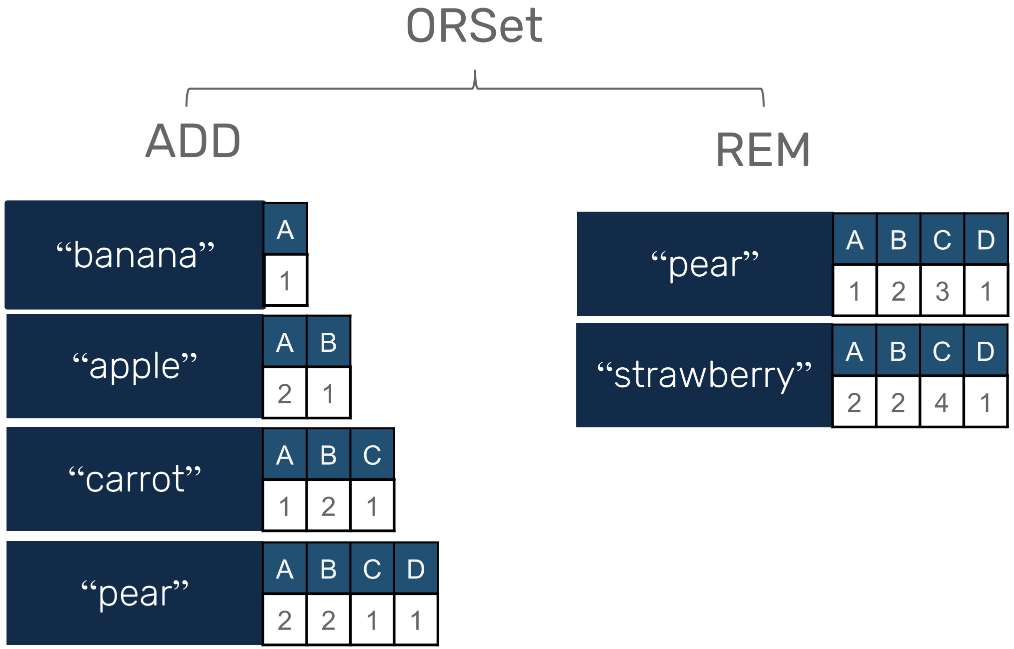
+

D	8
C	10
C	11
A	9

dot cloud

OR-SET

UNOPTIMIZED VERSION



ORSet

ADD

	A
“banana”	1

	A	B
“apple”	2	1

	A	B	C
“carrot”	1	2	1

	A	B	C	D
“pear”	2	2	1	1

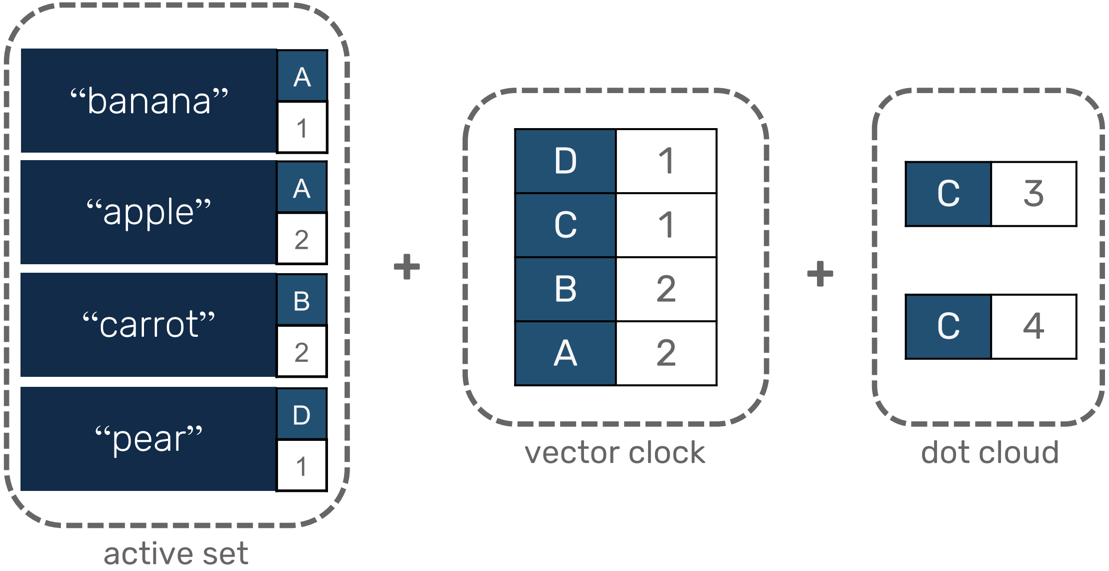
REM

	A	B	C	D
“pear”	1	2	3	1

	A	B	C	D
“strawberry”	2	2	4	1

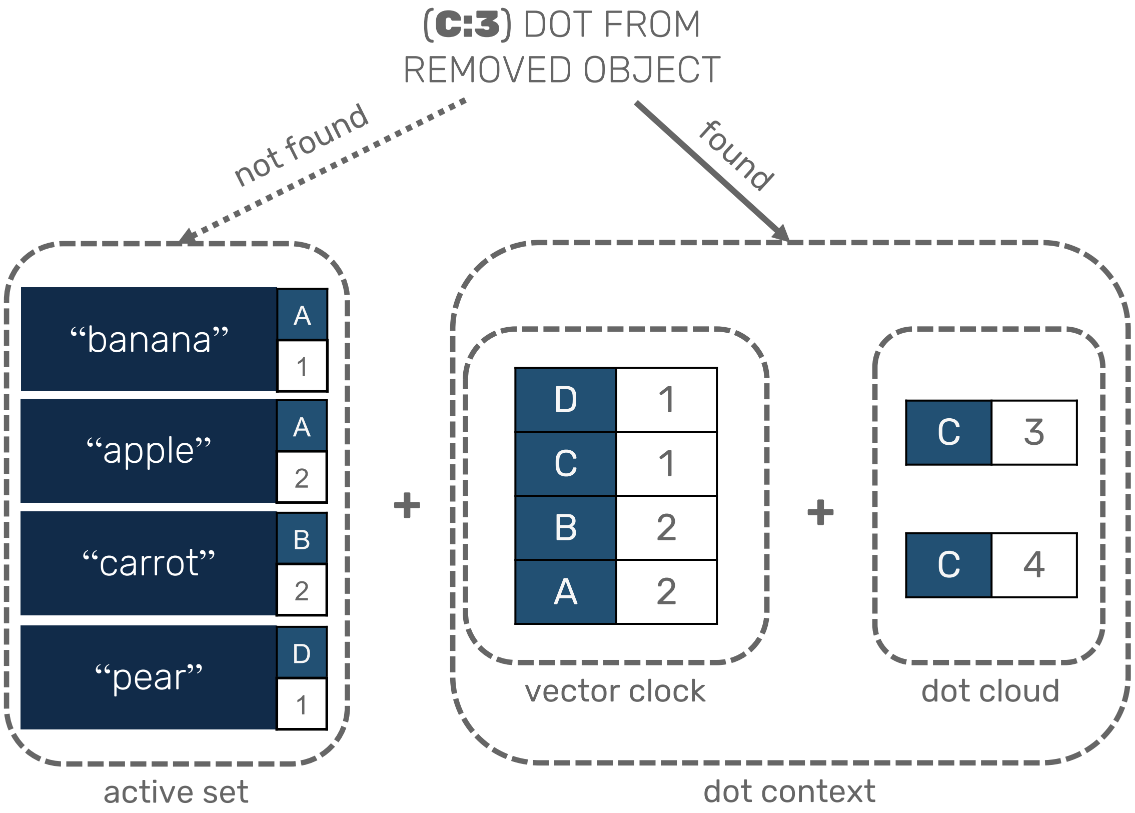
OR-SET

WITHOUT TOMBSTONES



OR-SET

TRACKING REMOVED VALUES



“banana”	A	1
“apple”	A	2
“carrot”	B	2
“pear”	D	1

D	1
C	1
B	2
A	2

C	3
C	4

WHAT'S NEXT?

1. JSON-like CRDTs
2. Distributed transactions
   a. RAMP
   b. CURE

SUMMARY

REFERENCES

- Consistency without consensus: https://www.infoq.com/presentations/crdt-soundcloud
- CRDTs and the Quest for Distributed Consistency: https://www.youtube.com/watch?v=B5NULPSiOGw
- CRDT blog posts: https://bartoszsypytkowski.com/tag/crdt/
- CRDT examples in F#: https://github.com/Horusiath/crdt-examples/
- Azure CosmosDB custom conflict resolution: https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-manage-conflicts#create-a-custom-conflict-resolution-policy-using-a-stored-procedure
- Redis Enterprise CRDB: https://docs.redislabs.com/latest/rs/administering/database-operations/create-crdb/

THANK YOU