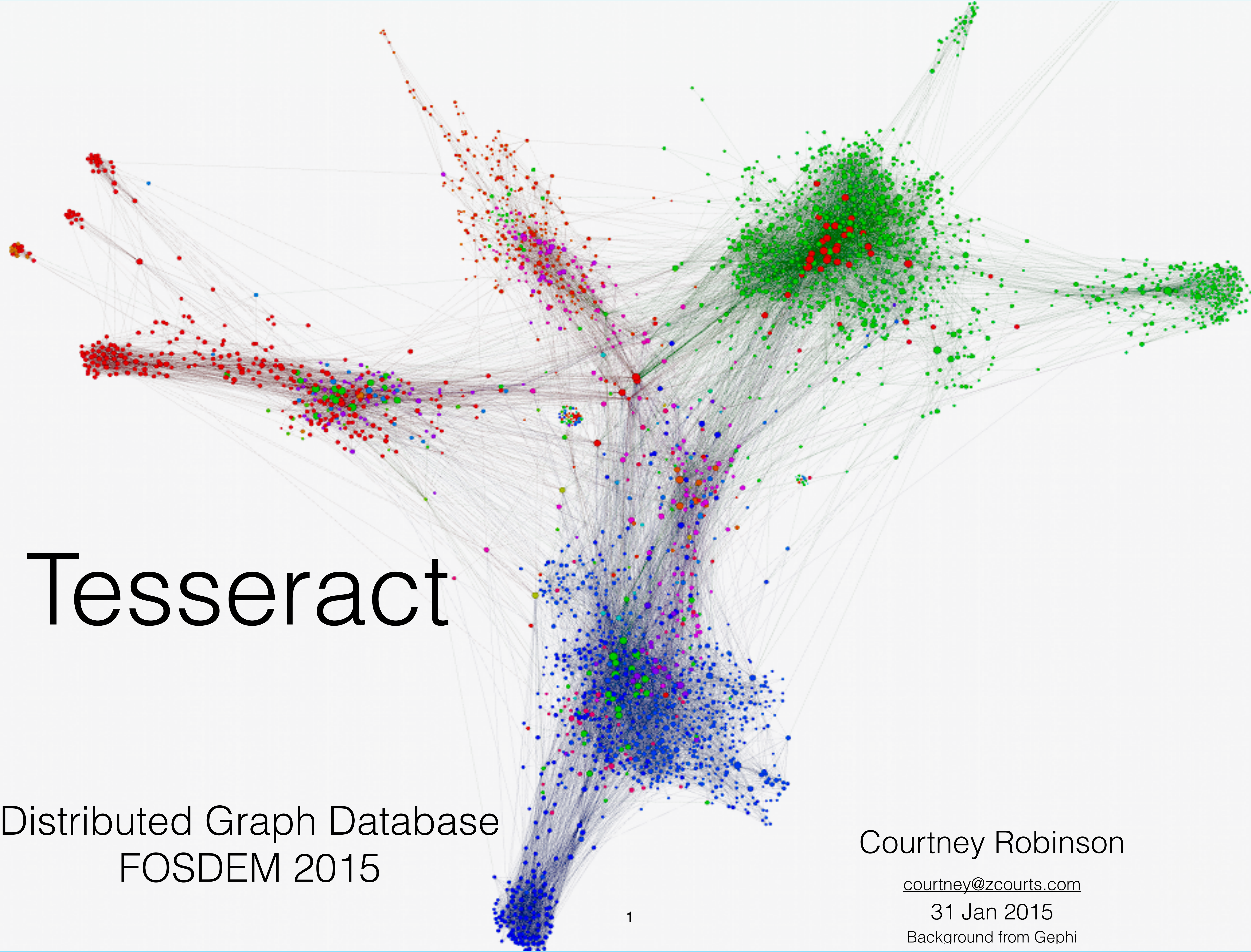


Tesseract

Distributed Graph Database
FOSDEM 2015

Courtney Robinson
courtney@zcourts.com

31 Jan 2015

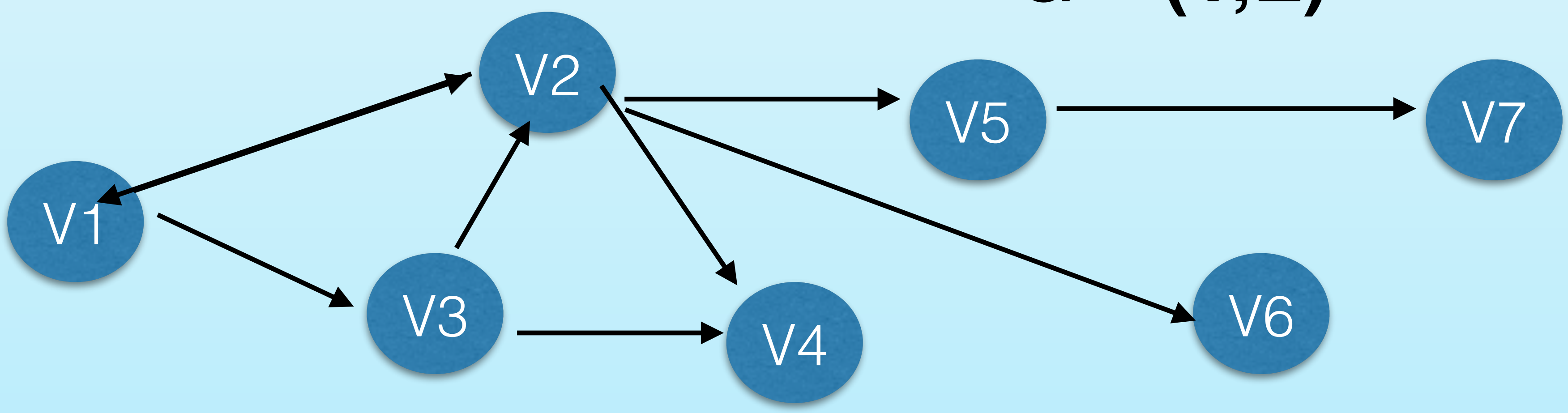
Background from Gephi

- I can be found around the web as “zcourts”, Google it…
- The web is one very prominent example of a graph
- Too big for a single machine
- So we must split or “partition” it over multiple
- Partitioning is hard…in fact, it has been shown to be np-complete
- All we can do is edge closer to more “optimal” solutions
- The Tesseract is an ongoing research project
- Its focus is on distributed graph partitioning
- The rest of this presentation is a series of solutions, which together, takes one step closer to faster distributed graph processing

Terminology

Graph - A graph G is made up of a set of vertices and edges,

G = (V,E)



Vertex - Smallest unit of user accessible datum

Edge - Connects two vertices, may have a direction

Property - Key value pair available on an Edge or Vertex

Aims of the Tesseract

1. Implement distributed eventually consistent graph database

2. Develop a distributed graph partitioning algorithm

3. Develop a computational model able to support both real time and batch processing on a distributed graph

Aims of the Tesseract

1. Implement distributed eventually consistent graph database
2. Develop a distributed graph partitioning algorithm
3. Develop a computational model able to support both real time and batch processing on a distributed graph

CRDTs…in one slide™

CRDTs…in one slide™

Conflict free replicated data types

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

(1+2) + 3 = 1 + (2+3)

Associative

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

Associative

(1+2) + 3 = 1 + (2+3)

Commutative

1 + 2 = 2 + 1

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

Associative

$(1+2) + 3 = 1 + (2+3)$

Commutative

$1 + 2 = 2 + 1$

$1 + 1 \neq 1$

Not Idempotent

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

$$(1+2) + 3 = 1 + (2+3)$$

Associative

Commutative

$$1 + 2 = 2 + 1$$

$$1 + 1 \neq 1$$

Not Idempotent

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

$$(1+2) + 3 = 1 + (2+3)$$

Associative

Commutative

$$1 + 2 = 2 + 1$$

$$1 + 1 \neq 1$$

Not Idempotent

Unfortunately addition isn't enough. The CIA properties are required to have a CRDT

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

$$(1+2) + 3 = 1 + (2+3)$$

Associative

$$1 + 2 = 2 + 1$$

Commutative

$$1 + 1 \neq 1$$

Not Idempotent

Unfortunately addition isn't enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… **Sets!**

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

$(1+2) + 3 = 1 + (2+3)$ — Associative

Commutative — $1 + 2 = 2 + 1$

$1 + 1 \neq 1$ — Not Idempotent

Unfortunately addition isn't enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… **Sets!**

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

$$(1+2) + 3 = 1 + (2+3)$$

Associative

Commutative

$$1 + 2 = 2 + 1$$

$$1 + 1 \neq 1$$

Not Idempotent

Unfortunately addition isn't enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… **Sets!**

Addition with sets is done using ∪

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

$$(1+2) + 3 = 1 + (2+3)$$

Associative

Commutative

$$1 + 2 = 2 + 1$$

$$1 + 1 \neq 1$$

Not Idempotent

Unfortunately addition isn't enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… **Sets!**

Addition with sets is done using ∪

$$(1 \cup 2) \cup 3 = 1 \cup (2 \cup 3)$$

Associative

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

Associative

$$(1+2) + 3 = 1 + (2+3)$$

Commutative

$$1 + 2 = 2 + 1$$

$$1 + 1 \neq 1$$

Not Idempotent

Unfortunately addition isn't enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… **Sets!**

Addition with sets is done using ∪

Associative

$$(1 \cup 2) \cup 3 = 1 \cup (2 \cup 3)$$

Commutative

$$1 \cup 2 = 2 \cup 1$$

CRDTs…in one slide™

Conflict free replicated data types

i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

Associative

$(1+2) + 3 = 1 + (2+3)$

Commutative

$1 + 2 = 2 + 1$

$1 + 1 \neq 1$

Not Idempotent

Unfortunately addition isn't enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… **Sets!**

Addition with sets is done using ∪

Associative

$(1 \cup 2) \cup 3 = 1 \cup (2 \cup 3)$

Commutative

$1 \cup 2 = 2 \cup 1$

$1 \cup 1 = 1$

Idempotent!

I lied, two slides…TM?

I lied, two slides…TM?

- Several types of CRDTs are available.

I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.

I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!

I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!

= insert = merge = replicate = remove

I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!

= insert = merge = replicate = remove

I lied, two slides…TM?

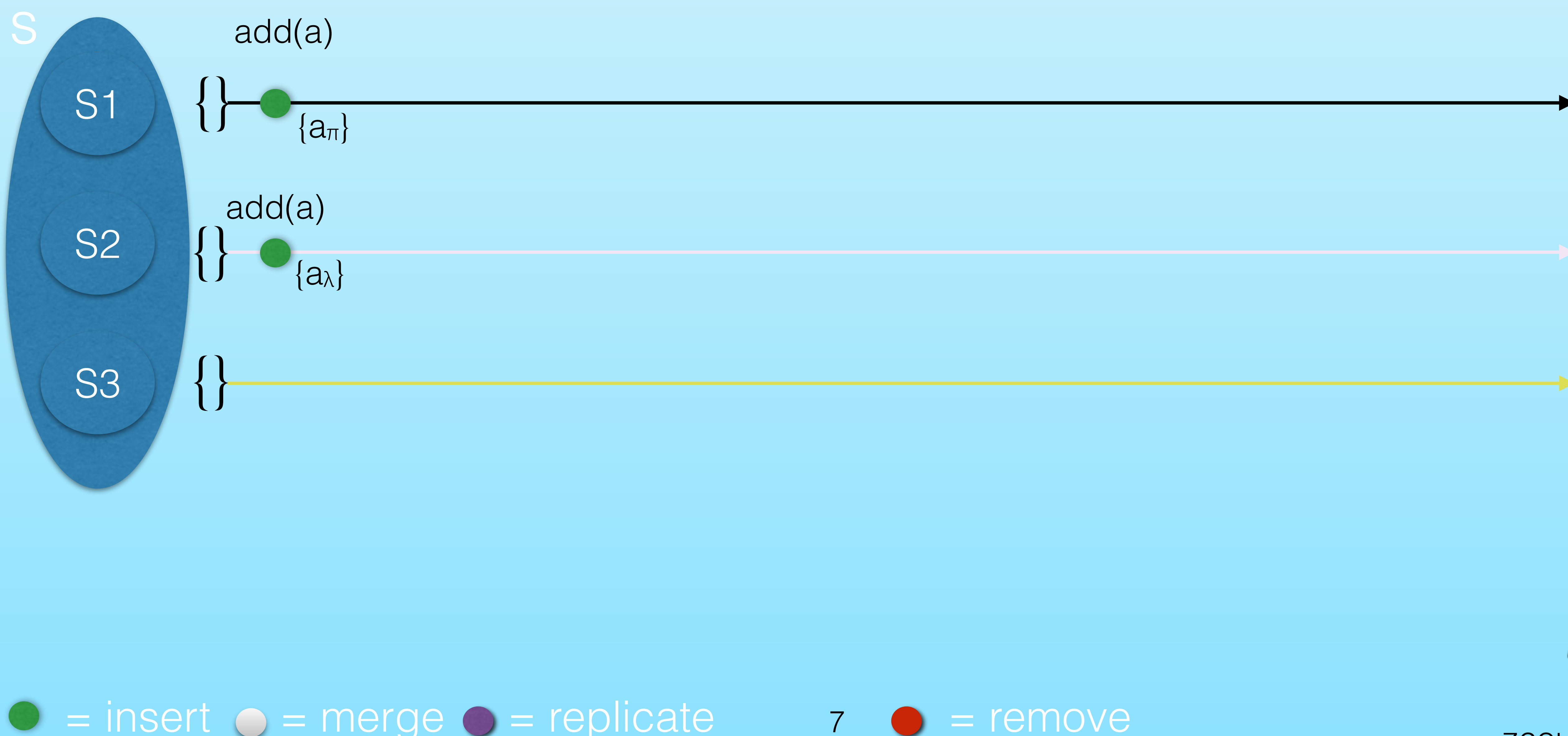
- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!

= insert = merge = replicate = remove

zcourts.com

I lied, two slides…TM?

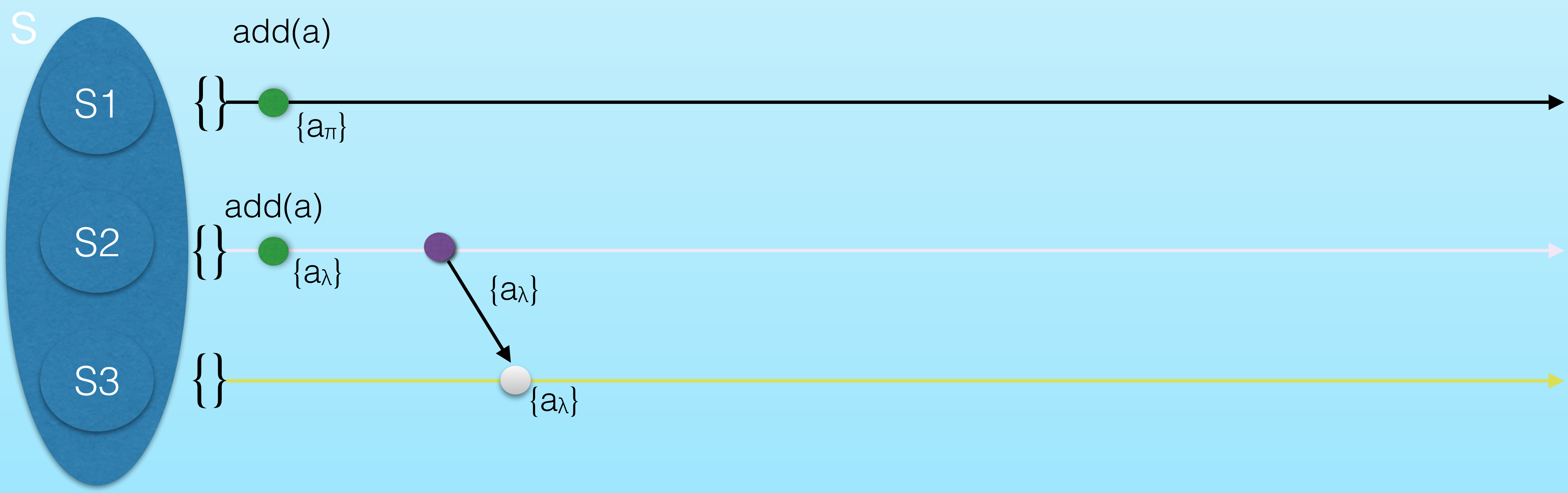
- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!



zcourts.com

I lied, two slides…TM?

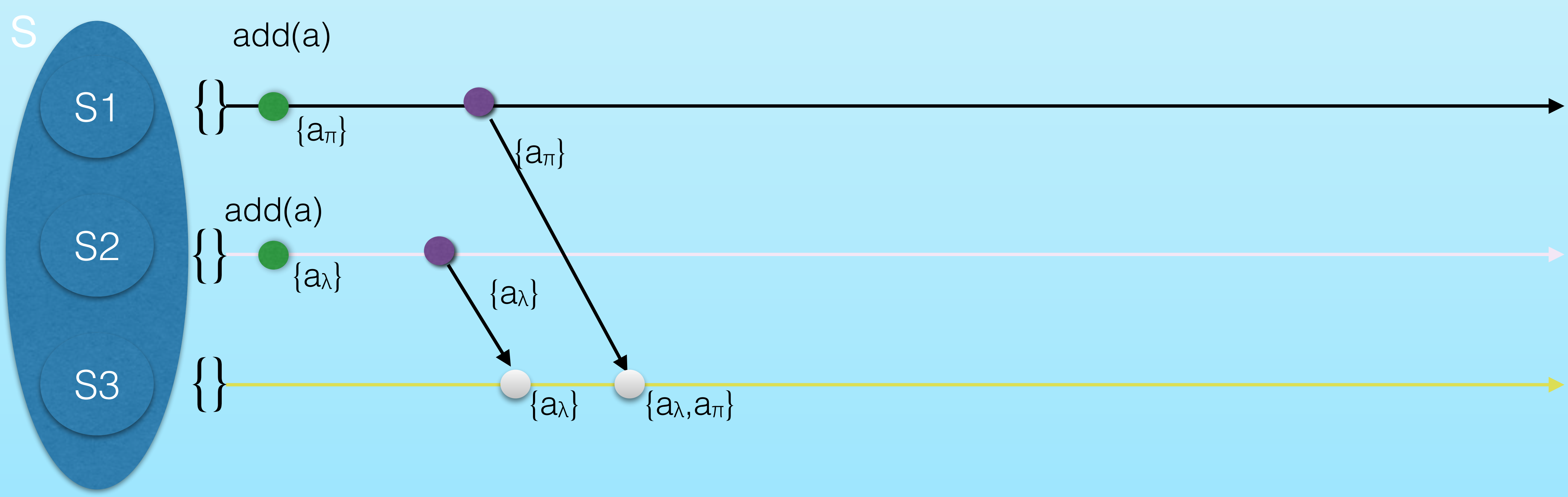
- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!



zcourts.com

I lied, two slides…TM?

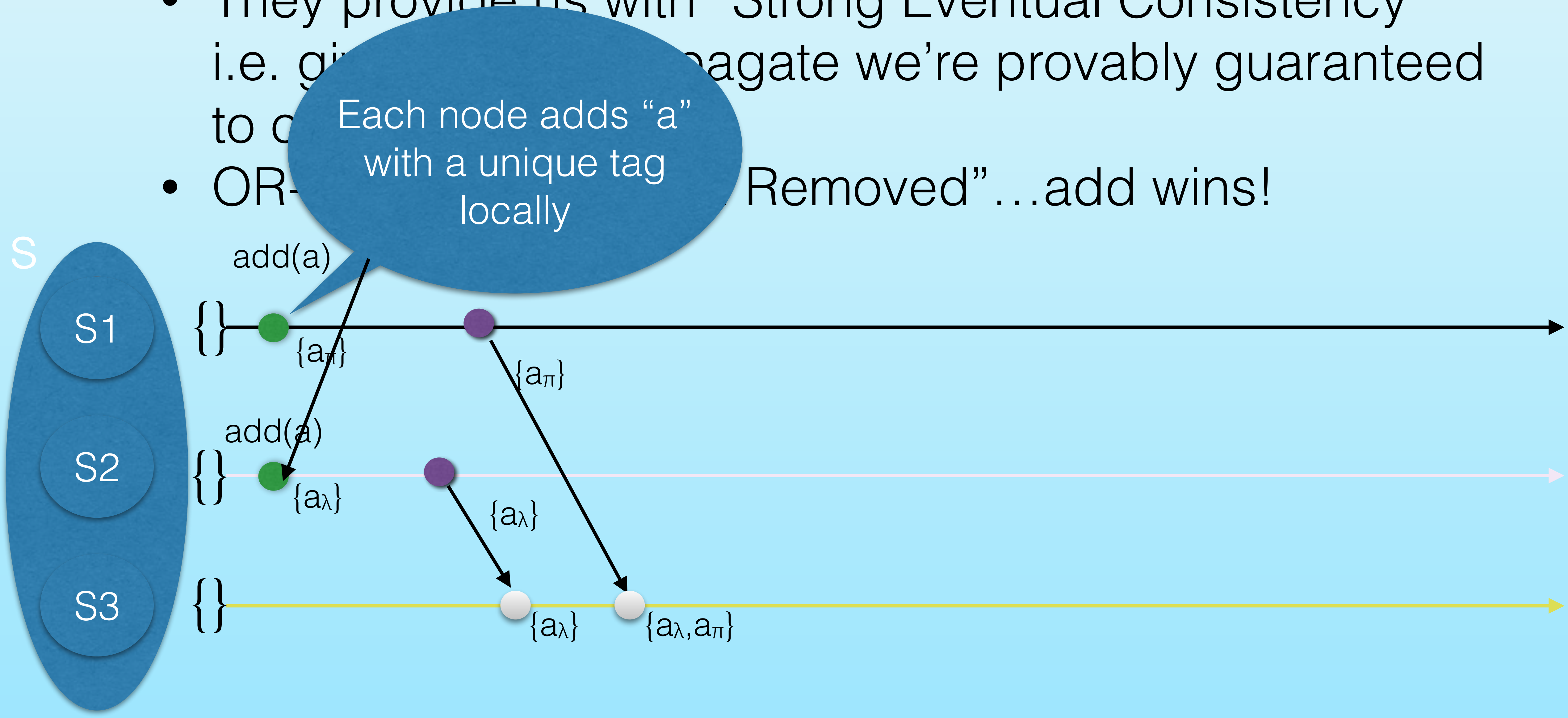
- Several types of CRDTs are available.
- They provide us with "Strong Eventual Consistency" i.e. given states propagate we're provably guaranteed to converge.
- OR-set i.e. "Observed Removed"…add wins!



= insert = merge = replicate = remove

I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. gi… …agate we’re provably guaranteed to c…
- OR-… Removed”…add wins!

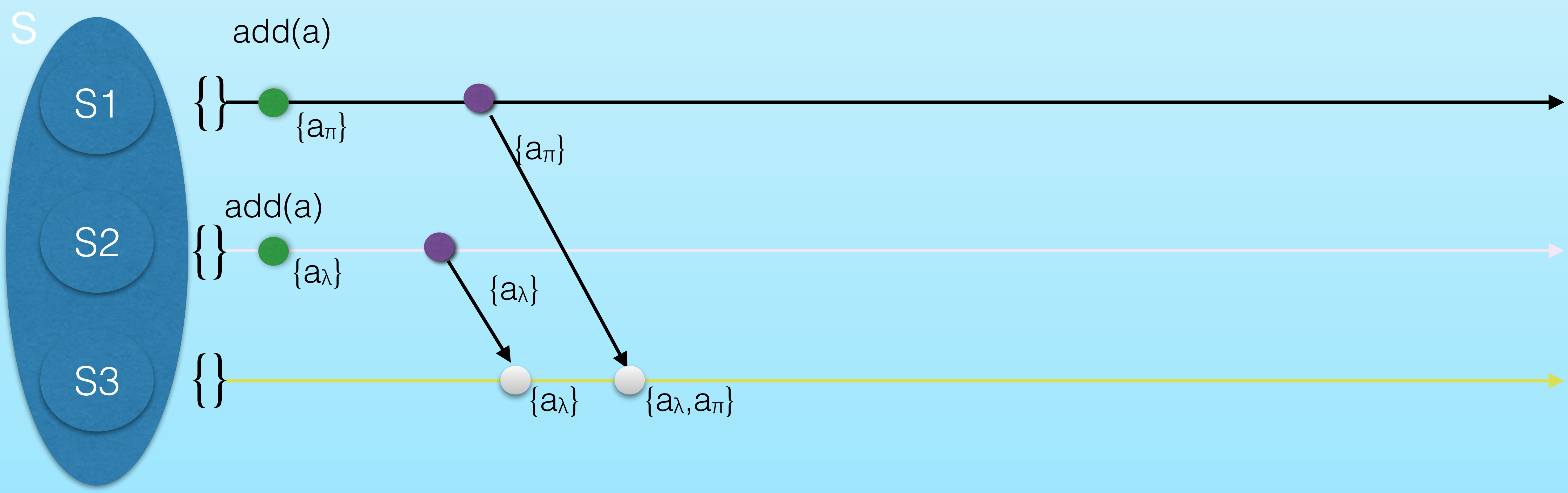


= insert = merge = replicate

= remove

I lied, two slides…TM?

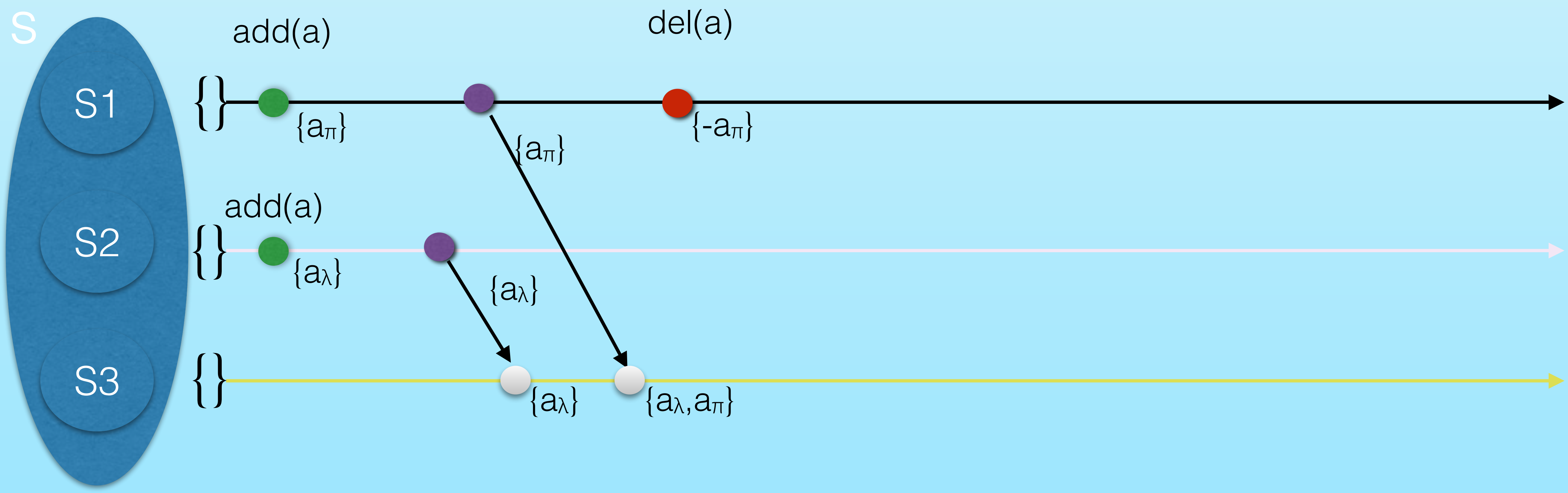
- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!



= insert = merge = replicate = remove

I lied, two slides…TM?

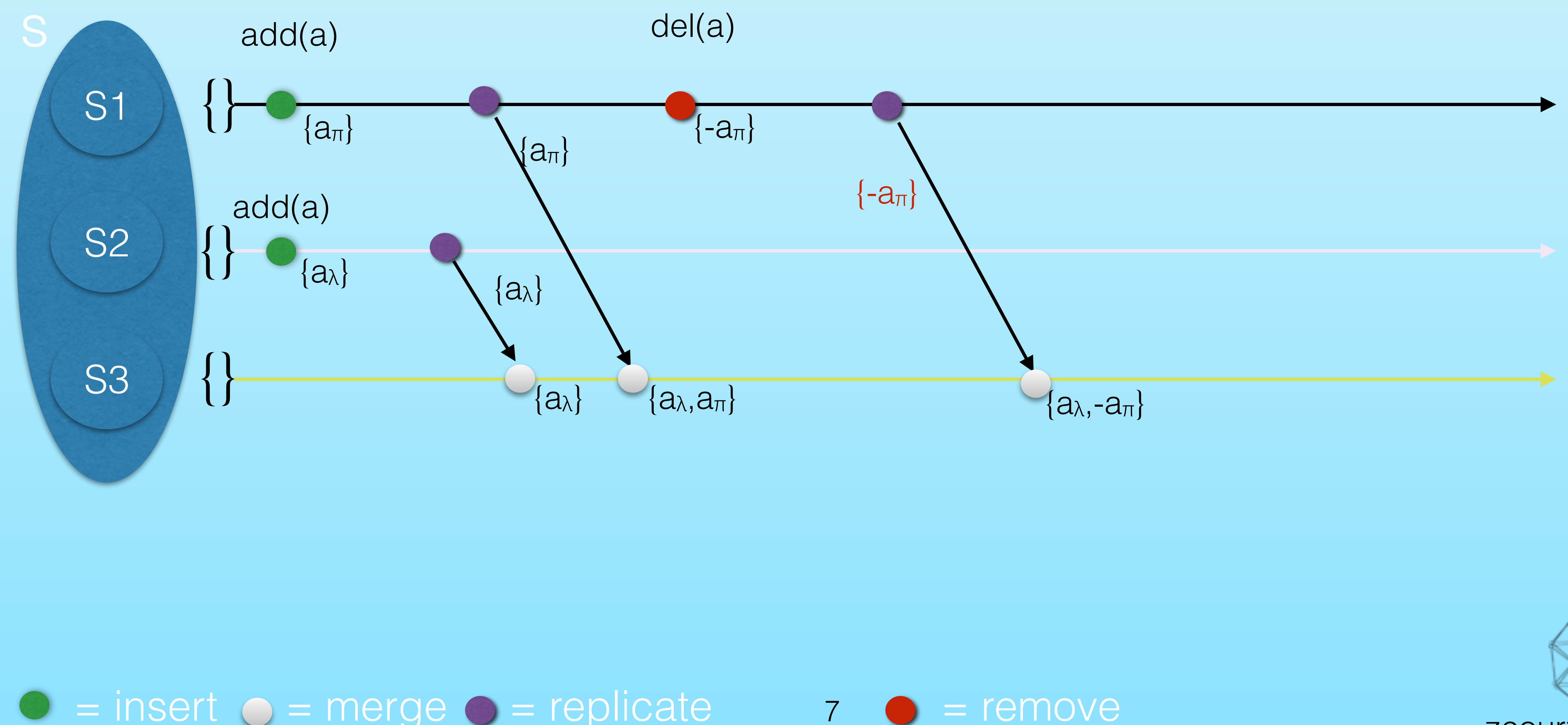
- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!



= insert = merge = replicate = remove

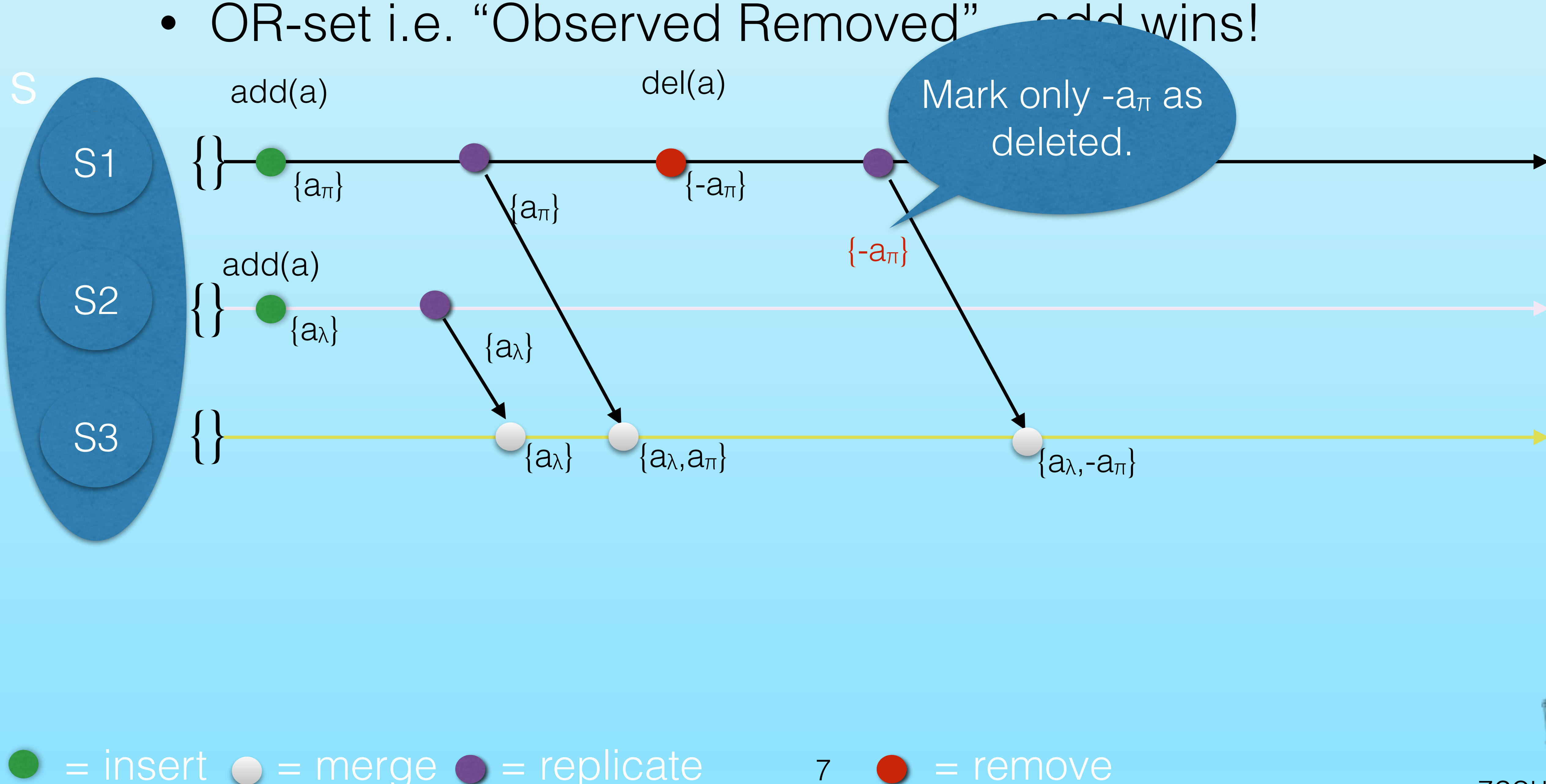
I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!



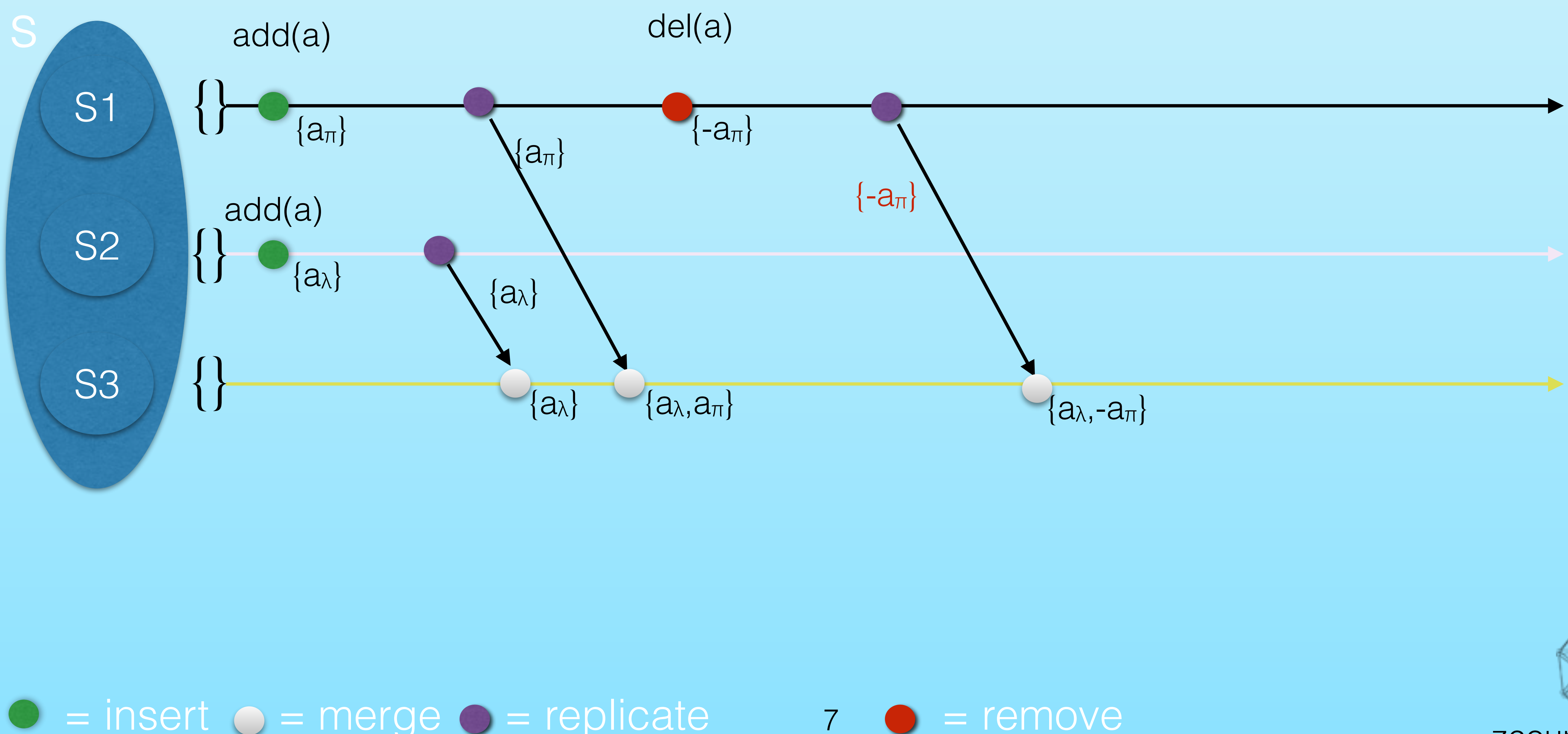
I lied, two slides…™?

- Several types of CRDTs are available.
- They provide us with "Strong Eventual Consistency" i.e. given states propagate we're provably guaranteed to converge.
- OR-set i.e. "Observed Removed" – add wins!



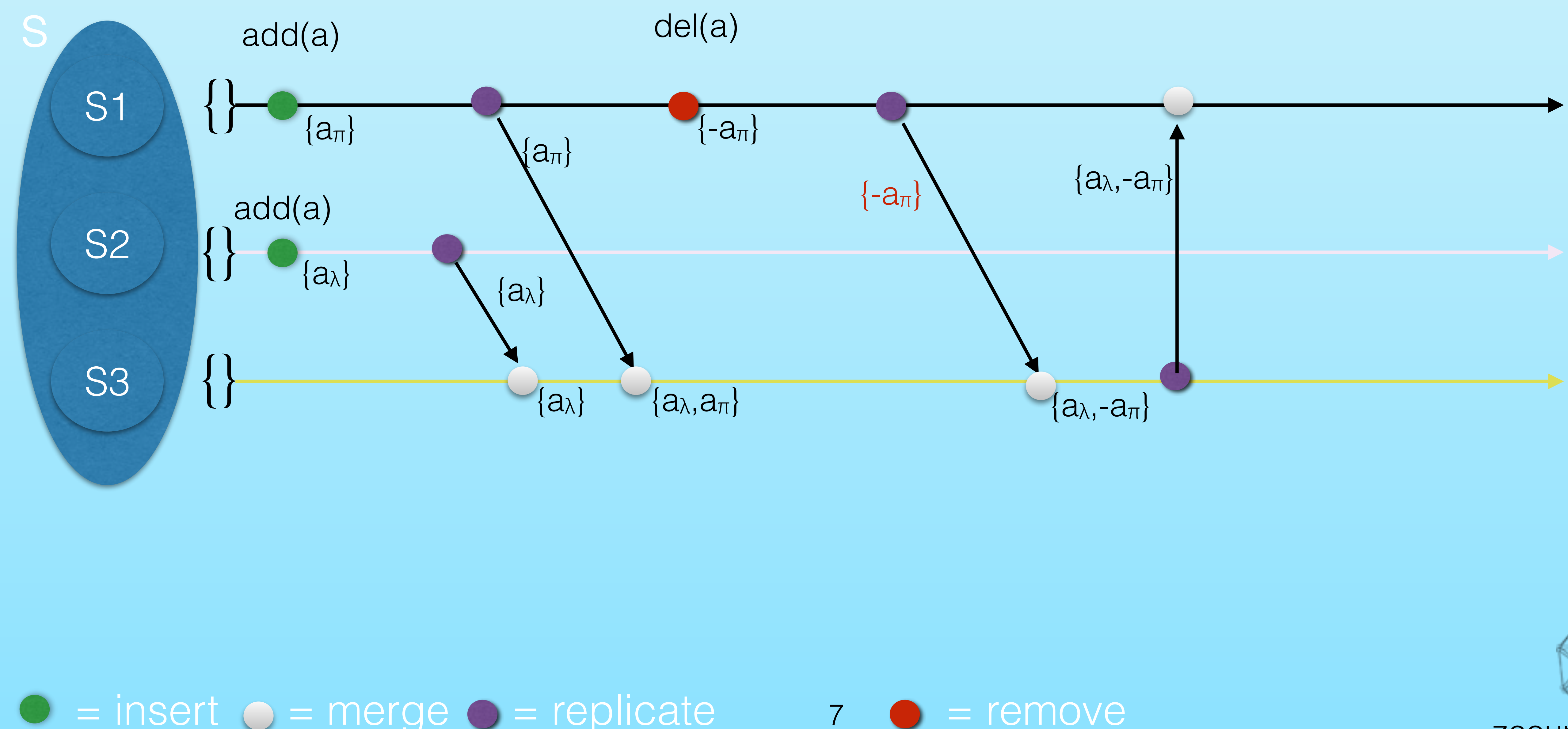
I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!



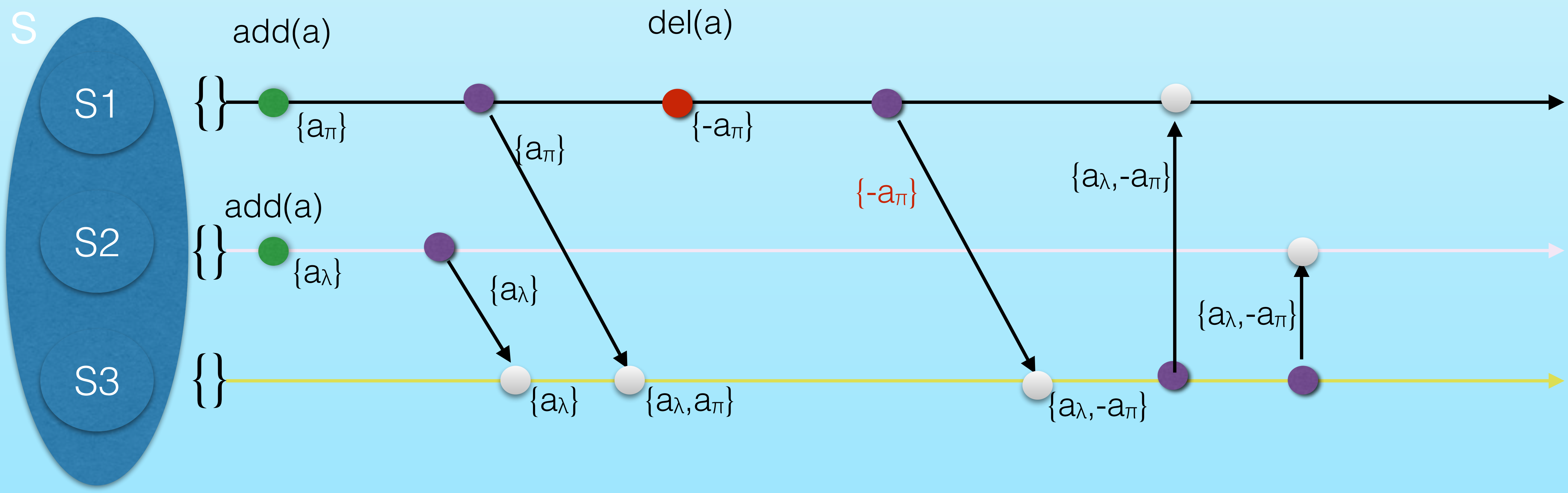
I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!



I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!

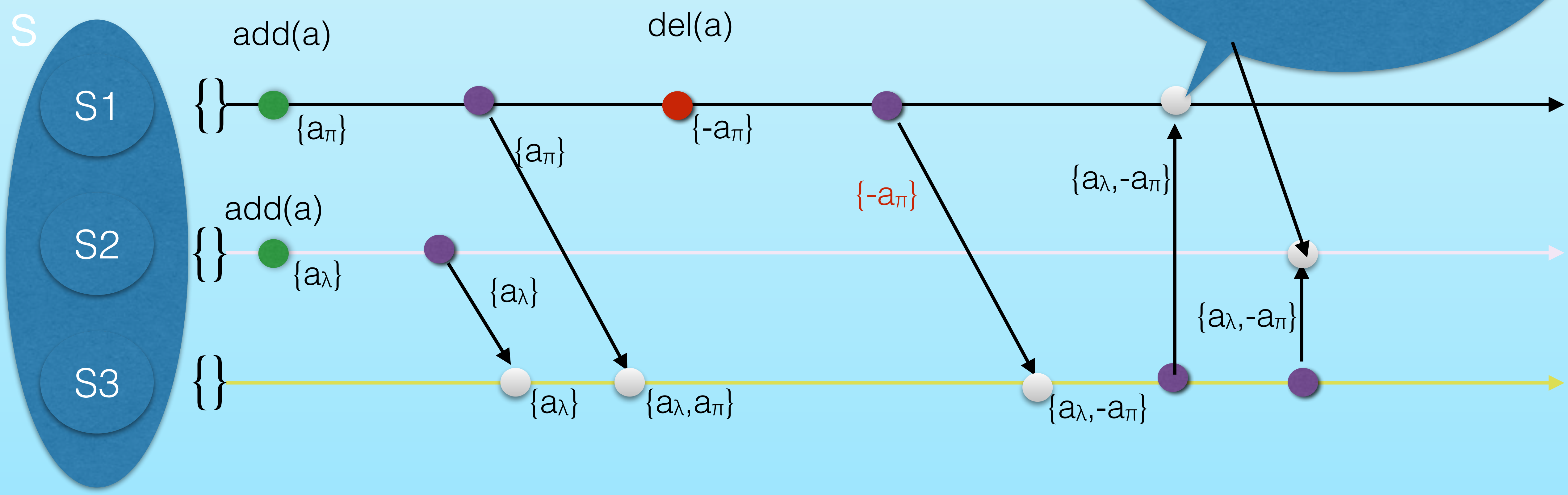


= insert = merge = replicate = remove

I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual
  i.e. given states propagate we’re prov
  to converge.
- OR-set i.e. “Observed Removed”…add

Merge takes symmetrical difference of the local and remote sets resulting in a_λ being in the set



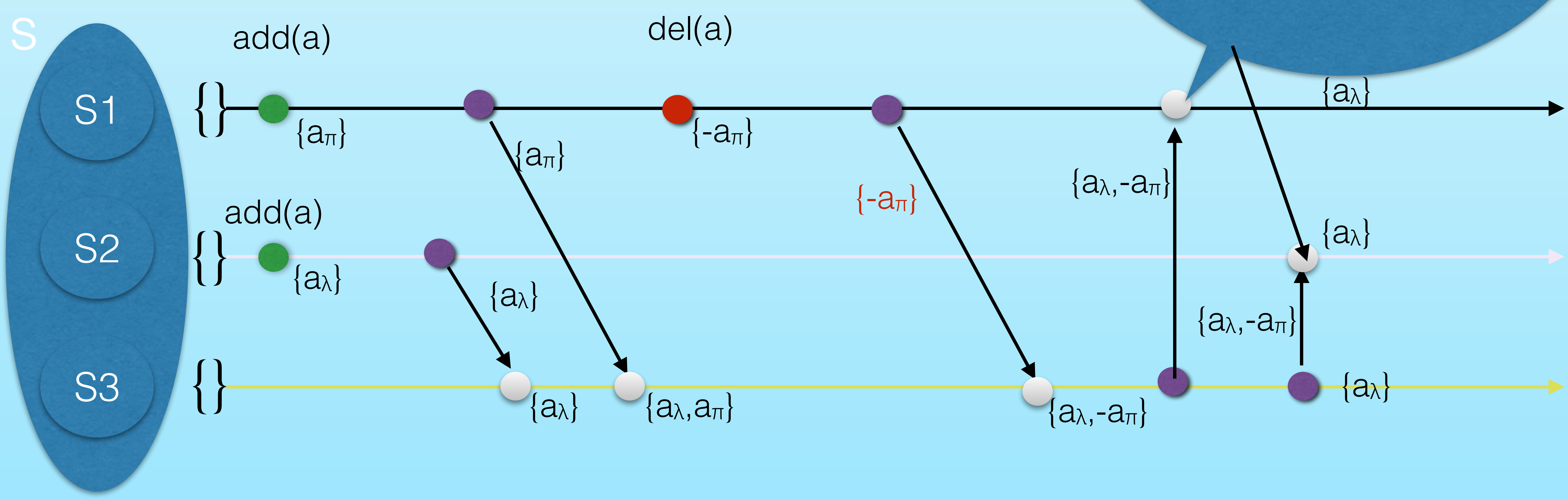
= insert = merge = replicate 7 = remove

zcourts.com

I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual
  i.e. given states propagate we’re prov
  to converge.
- OR-set i.e. “Observed Removed”…adc

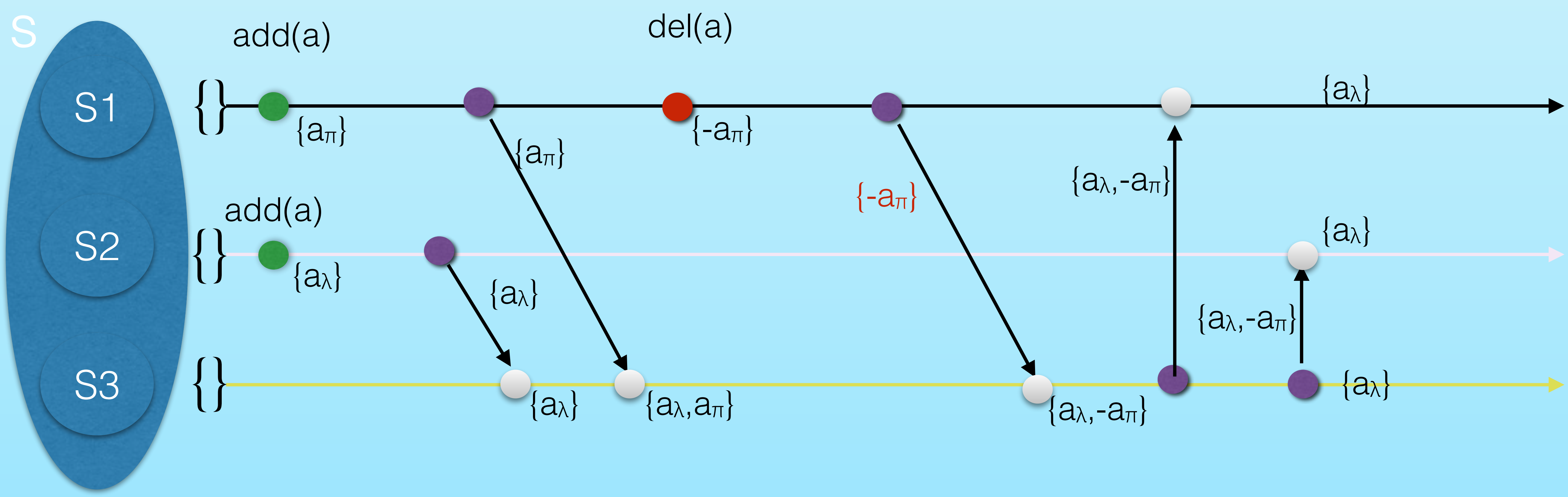
Merge takes symmetrical difference of the local and remote sets resulting in a_λ being in the set



= insert = merge = replicate = remove

I lied, two slides…™?

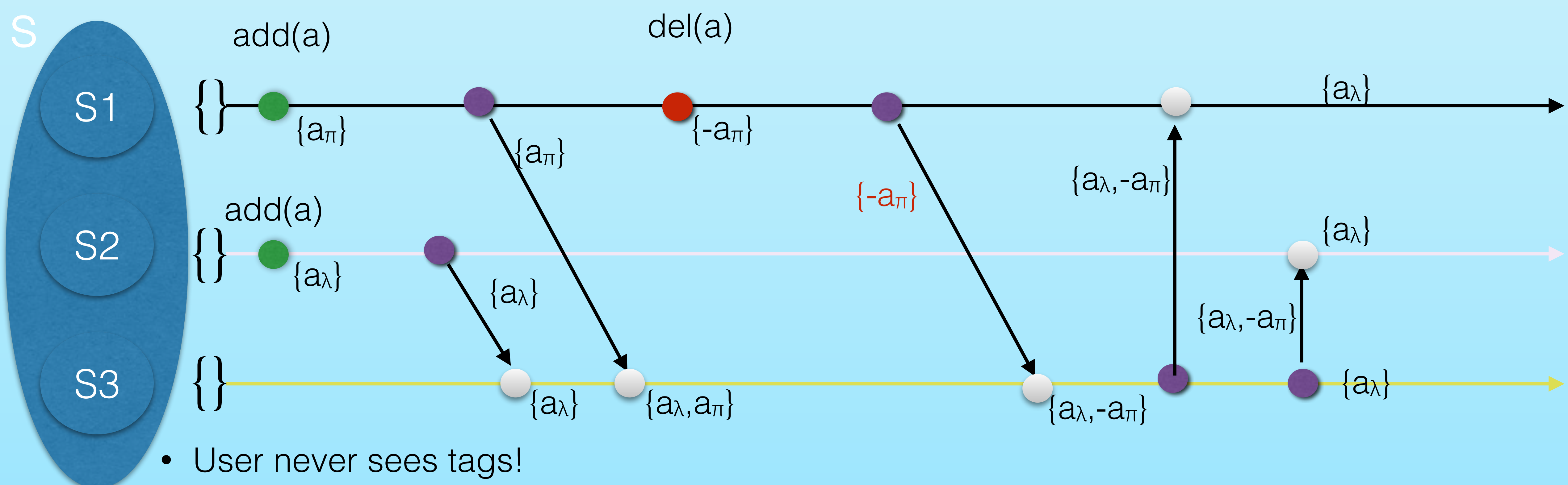
- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!



= insert = merge = replicate = remove

I lied, two slides…TM?

- Several types of CRDTs are available.
- They provide us with "Strong Eventual Consistency" i.e. given states propagate we're provably guaranteed to converge.
- OR-set i.e. "Observed Removed"…add wins!



- User never sees tags!
- Query time checks are used to enable DAGs (if violation of DAG constraint is detected then the runtime simply says the violating edge does not exist and triggers clean up)
- Note,the deleted "a" is optionally kept as a tombstone if the runtime is configured to support "snapshots"

= insert = merge = replicate = remove

Aims of the Tesseract

1. Implement distributed eventually consistent graph database

2. Develop a distributed graph partitioning algorithm

3. Develop a computational model able to support both real time and batch processing on a distributed graph

CRDTs again…because they're important

- One very important property of a CRDT is:

$$\{a,b,c,d\} :\Leftrightarrow \{a,b\} \cup \{c,d\}$$

- Those two sets being logically equivalent is a desirable property
- Enables partitioning (with rendezvous hashing for e.g.)

Naïve “cascading vertices”

- Naïve graph partitioning
- Depends on the query model to make up for its Naïvety
- Uses hashing to place data
- Two cascading algorithms formulated from:

  V = the vertex to cascade

  n = max nodes to cascade across

  ń = auto-determined value of n, using logistics growth model

  d = deg(v) = Degree of V

  e = $\langle \forall_{deg(v)} \in G \rangle$ i.e. average degree of all vertices in the graph

  lnVl = Max number of edges per node for a vertex

  i.e. cascading point (min number of edges before cascading occurs)

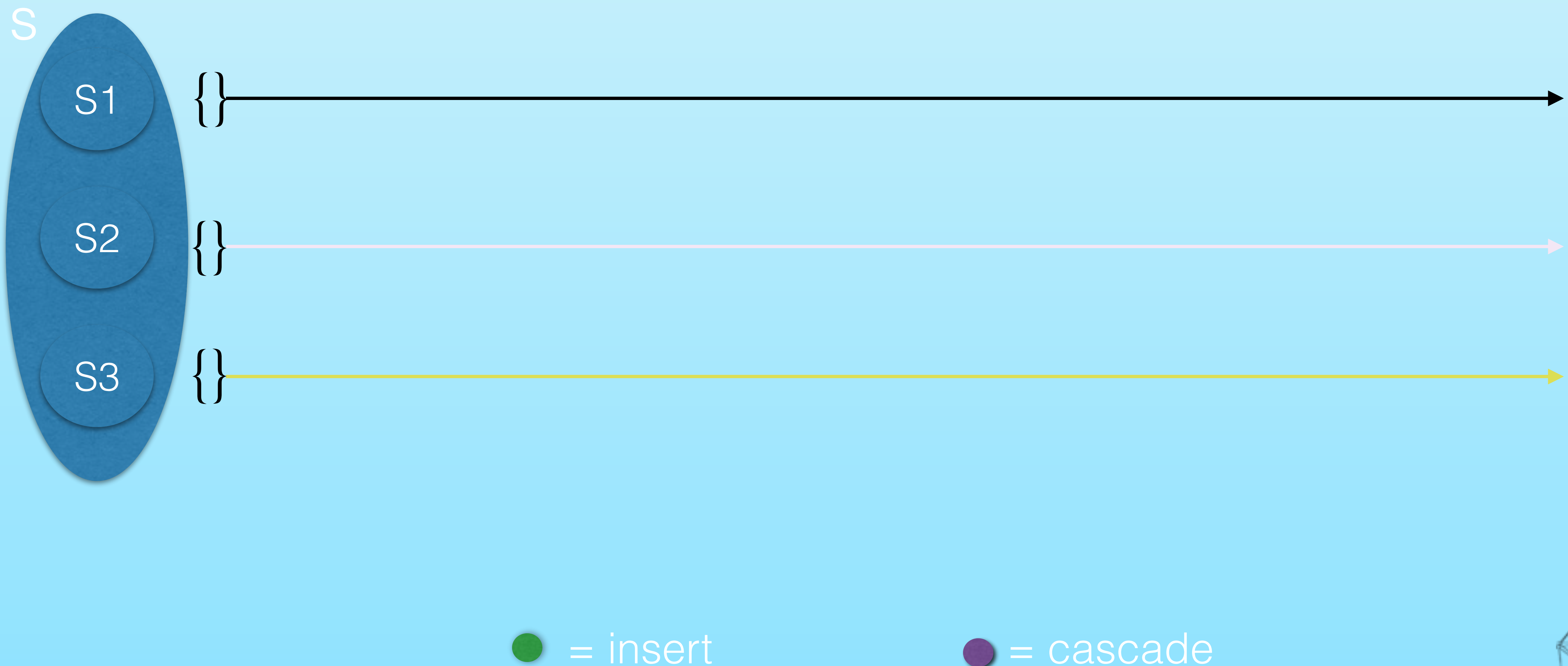
1. |nV| = d / n - user provides n, split evenly across nodes
2. |nV| = max(d,e) / n - user provides n, split evenly based on d or e if e is bigger

“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



zcourts.com

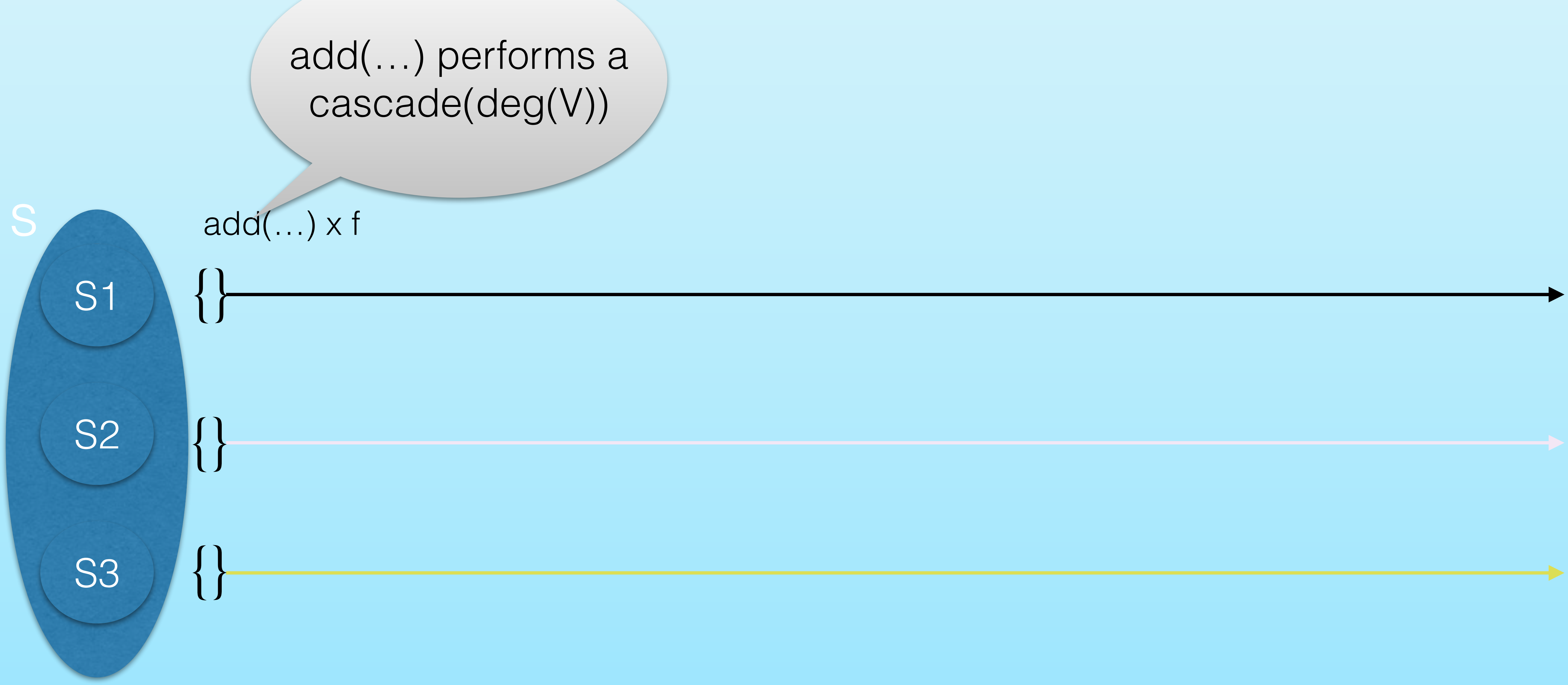
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

= insert   = cascade

“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



= insert = cascade

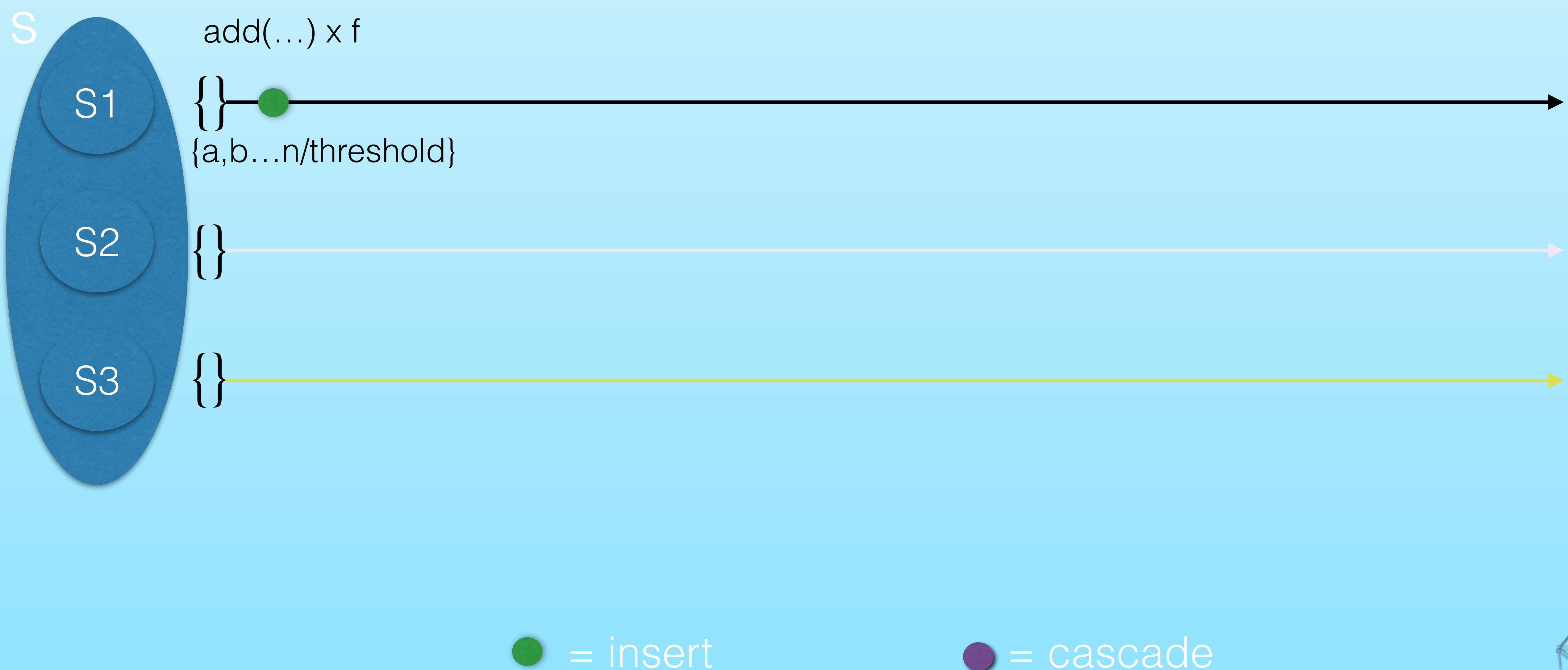
"Cascading vertices" by example

- Let's use Twitter followers as an example
- Each letter represents a unique follower

= insert = cascade

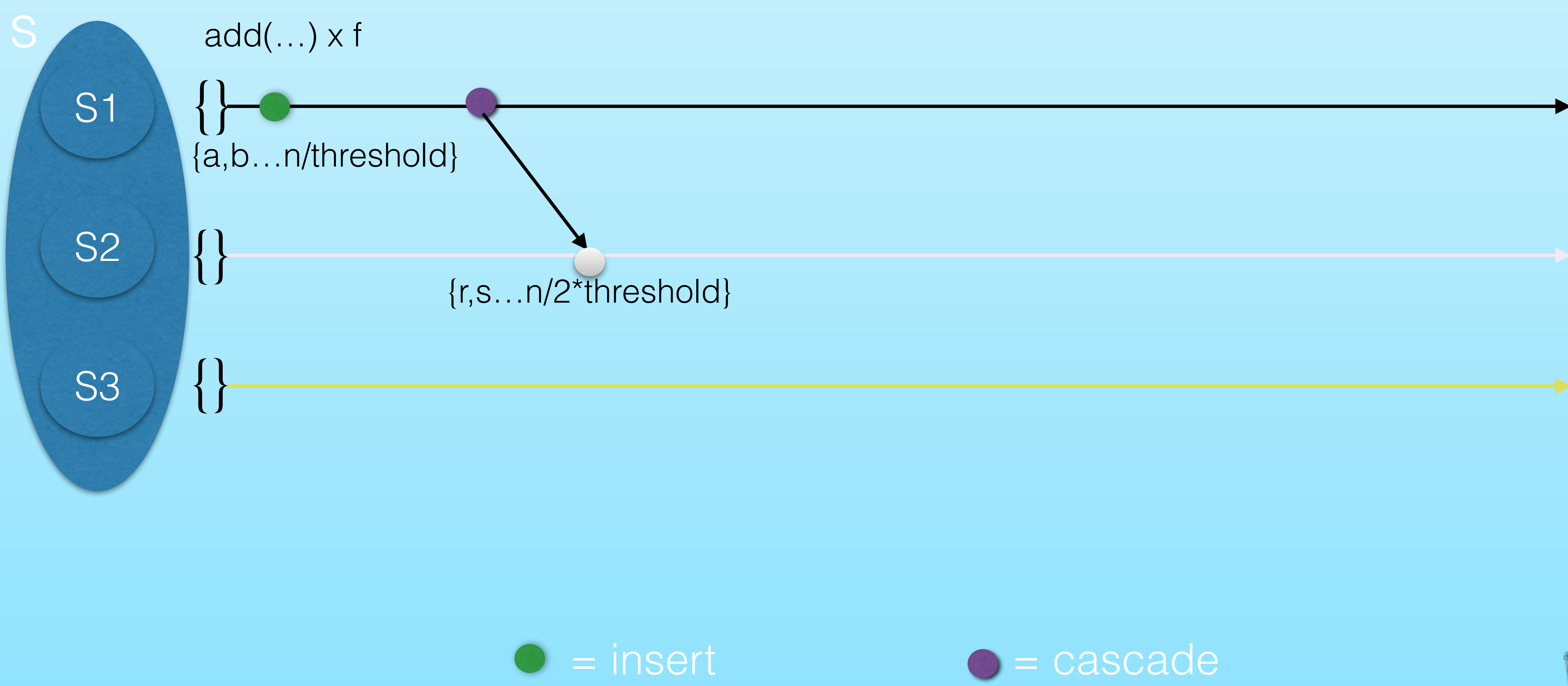
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



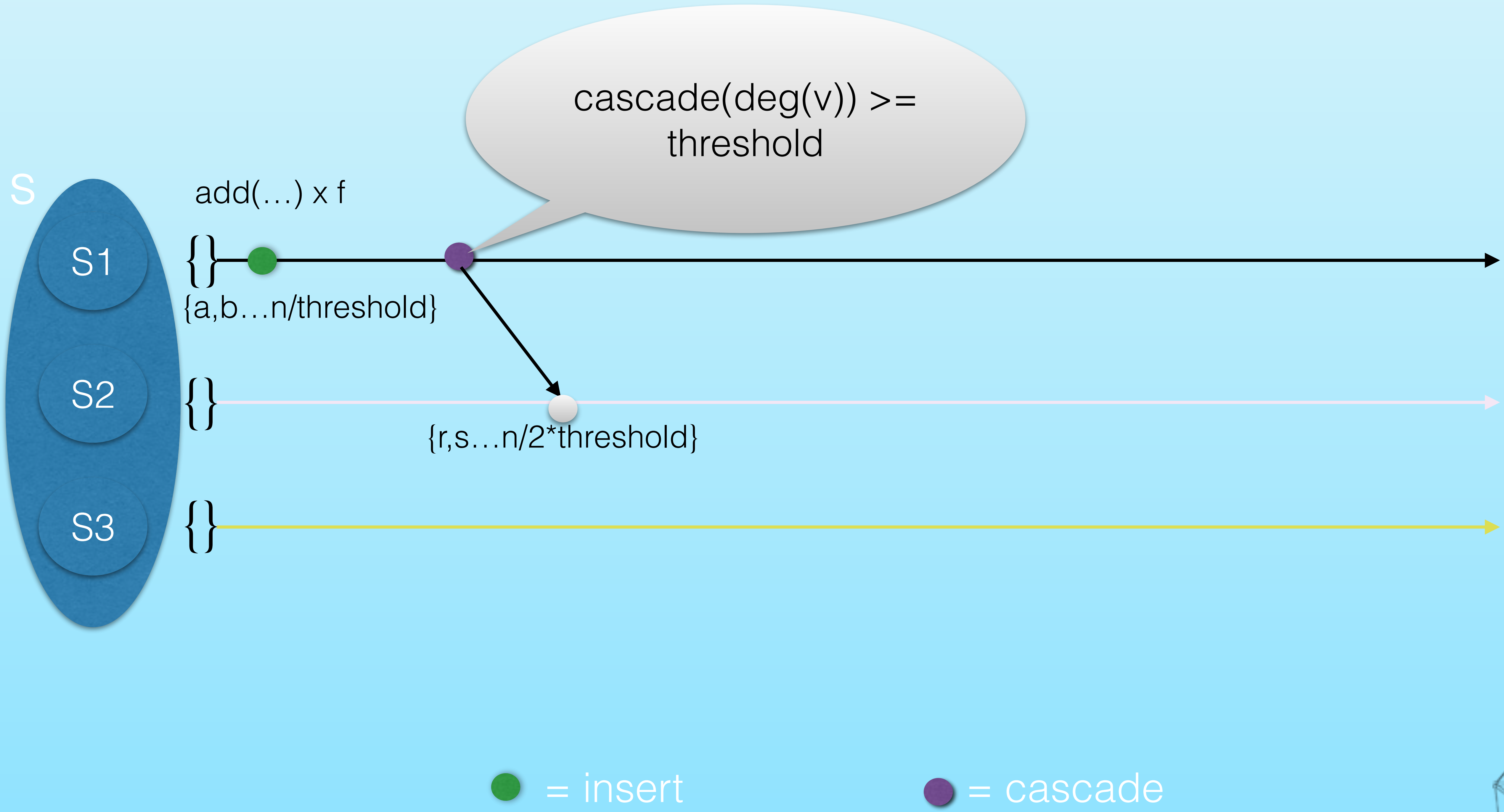
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



“Cascading vertices” by example

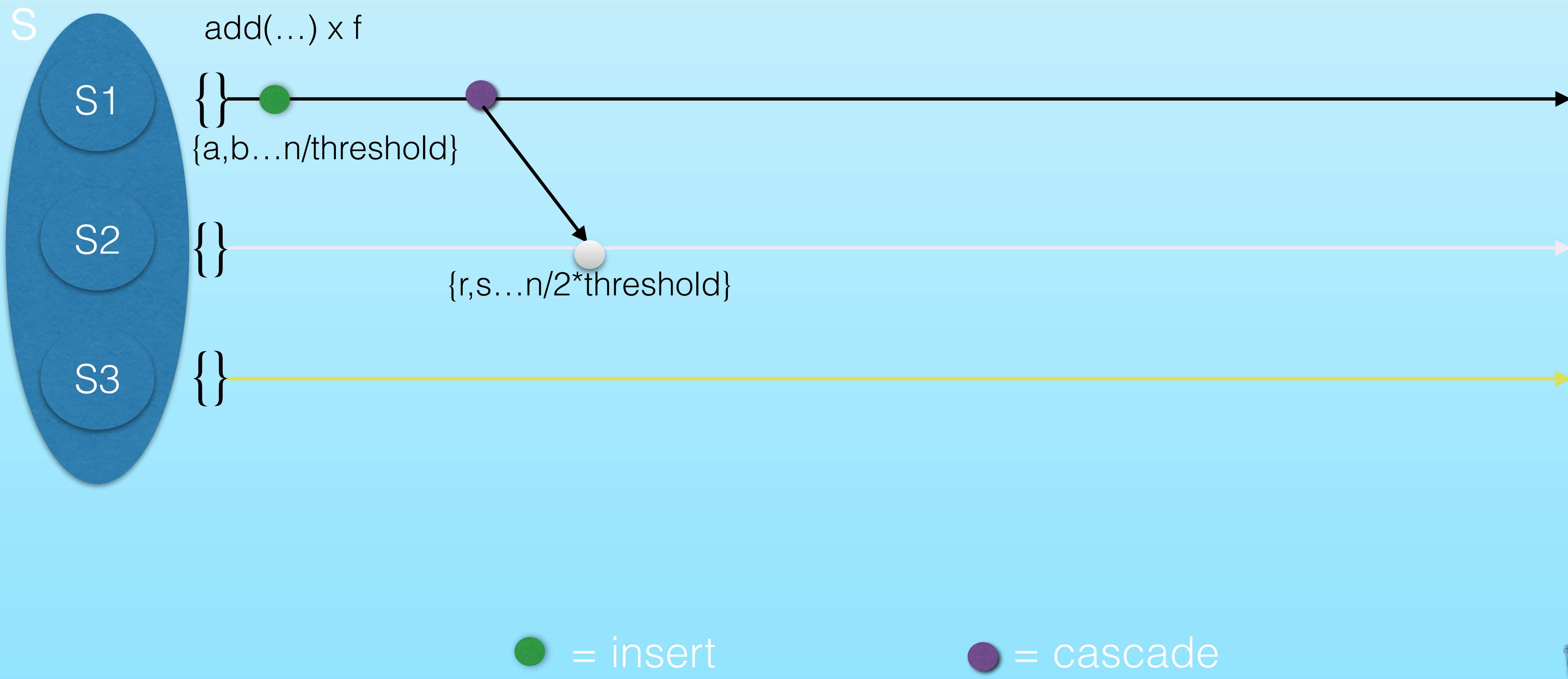
- Let’s use Twitter followers as an example
- Each letter represents a unique follower



zcourts.com

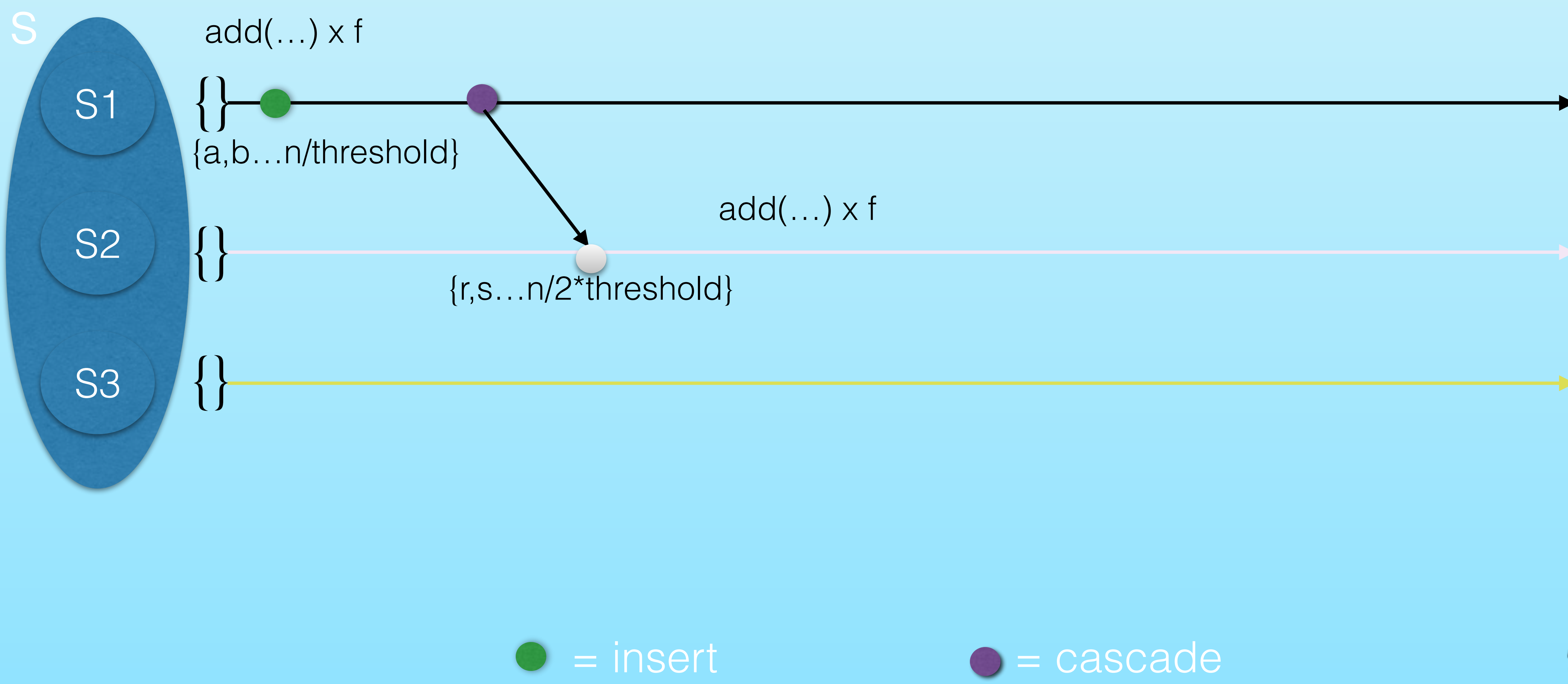
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



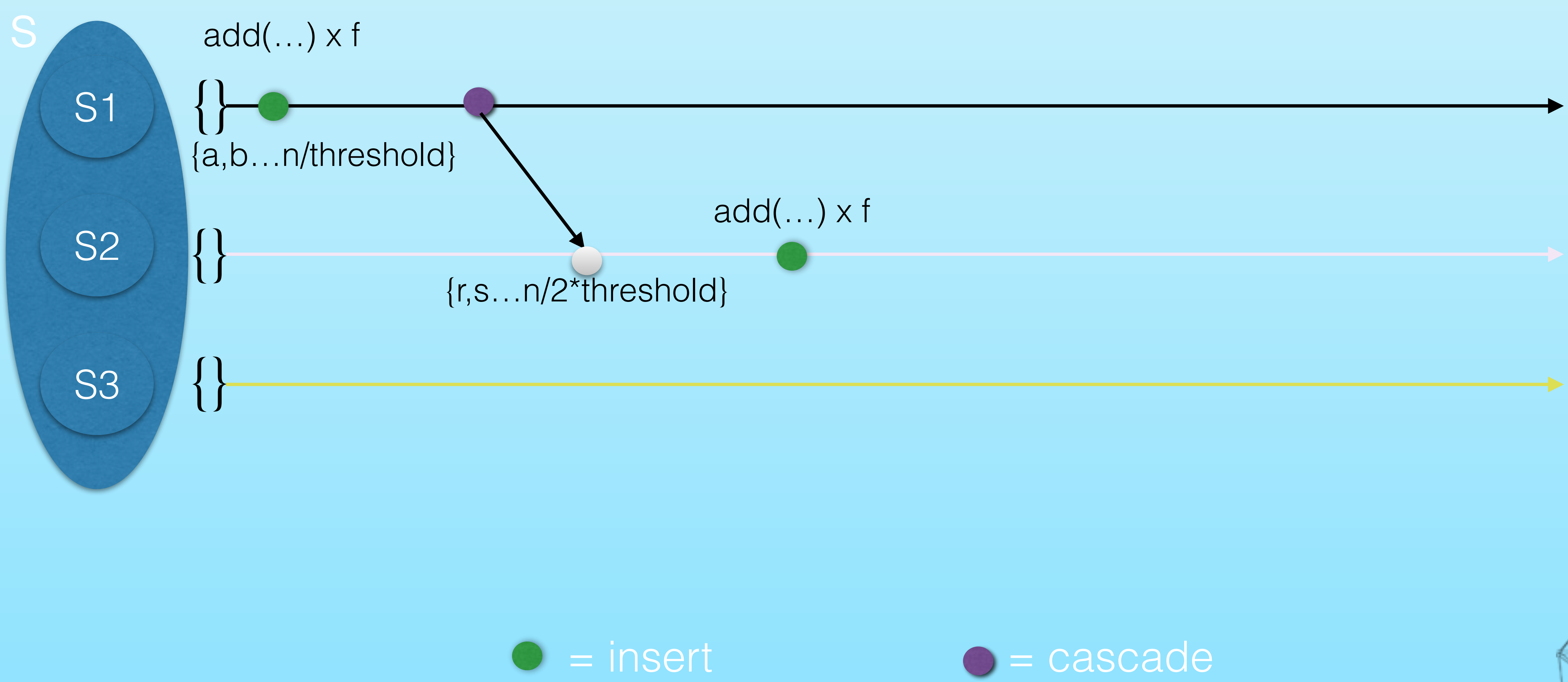
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



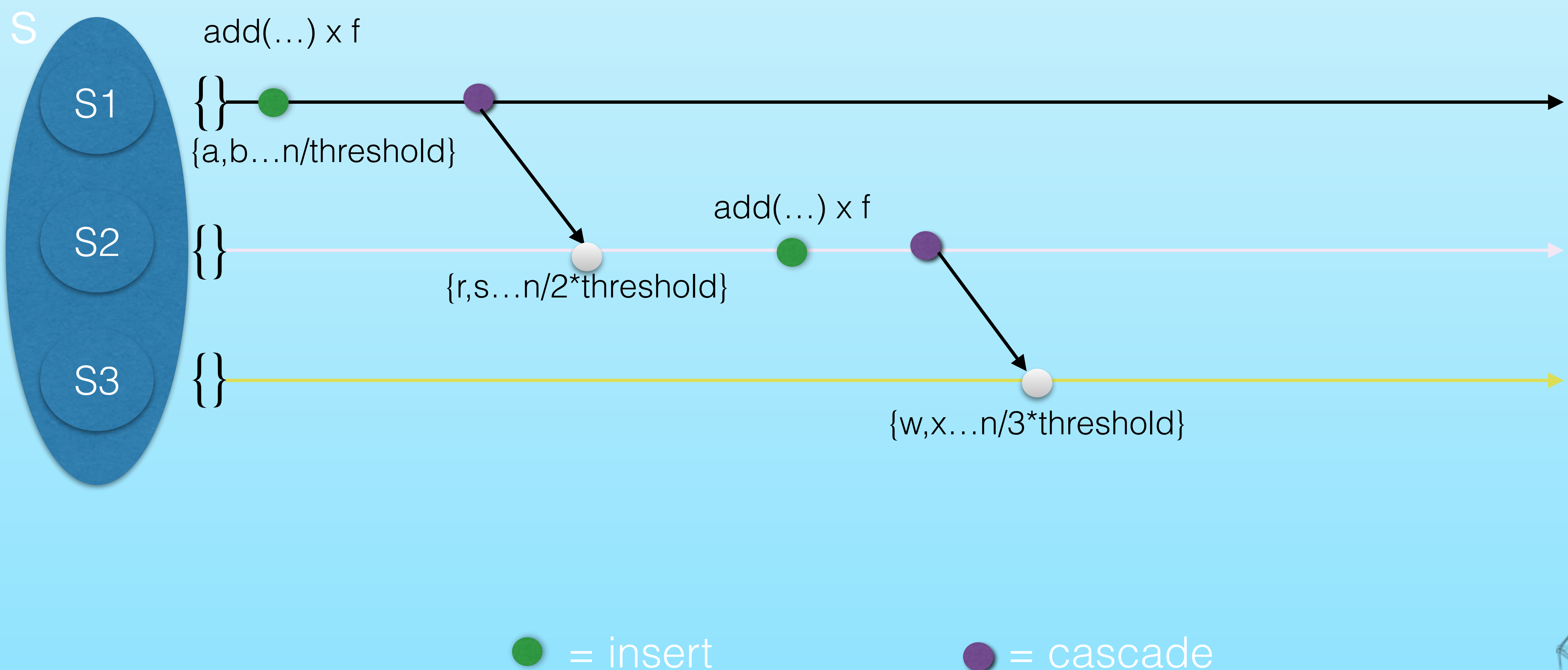
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



“Cascading vertices” by example

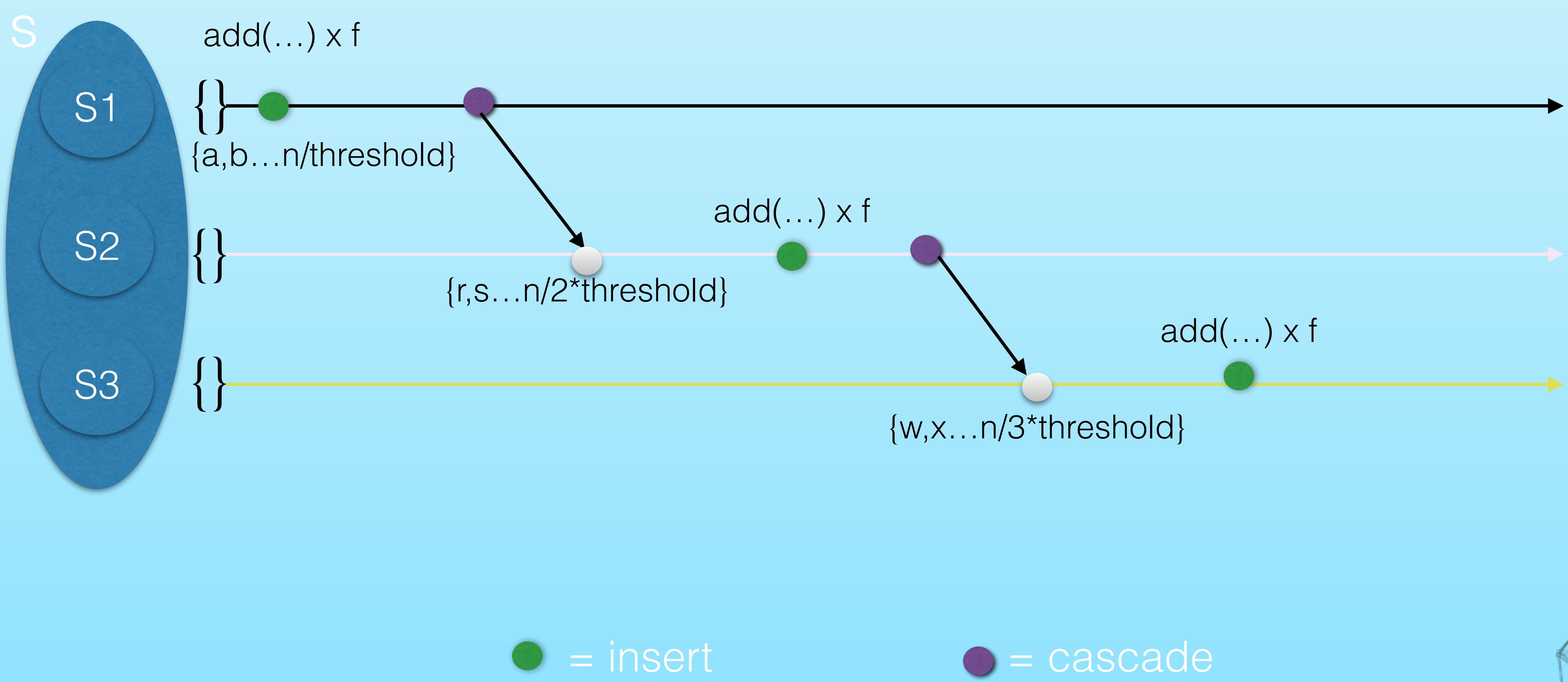
- Let’s use Twitter followers as an example
- Each letter represents a unique follower



zcourts.com

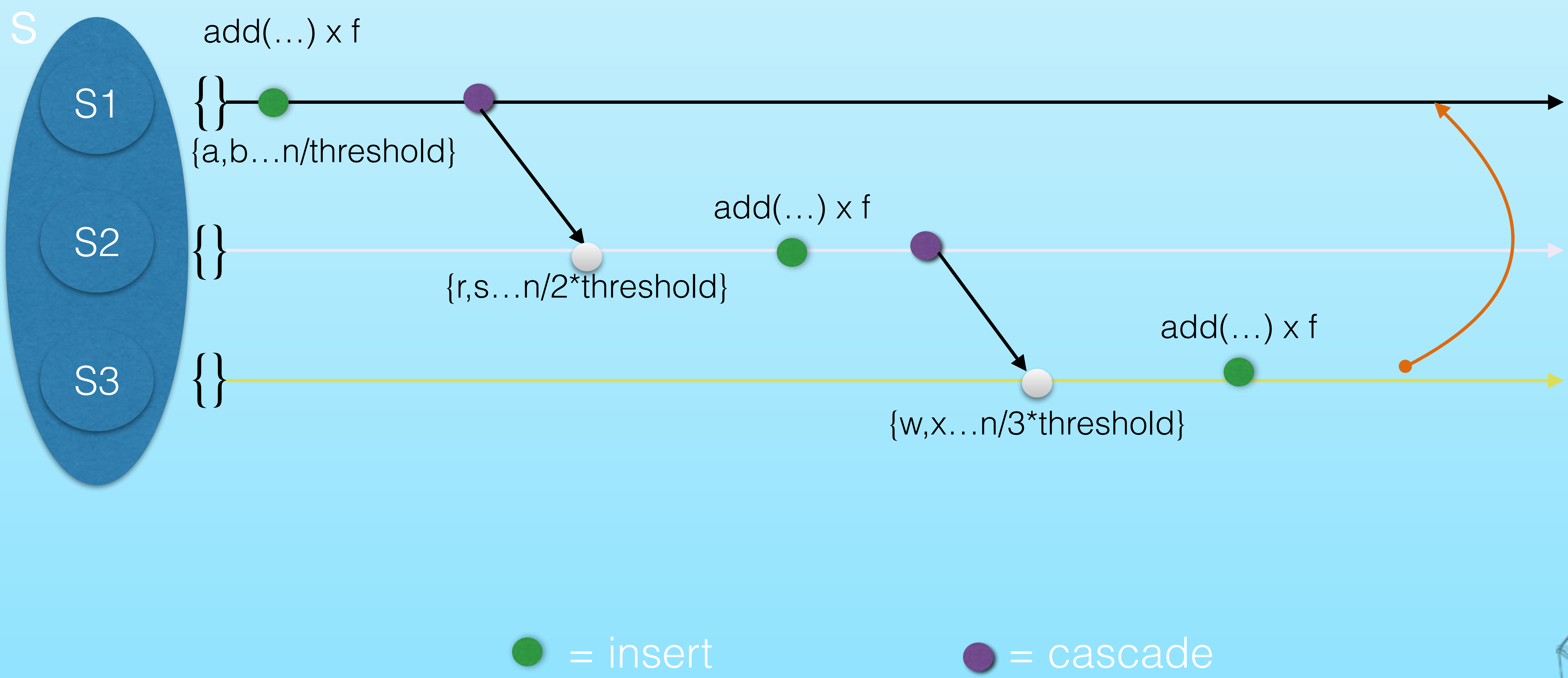
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



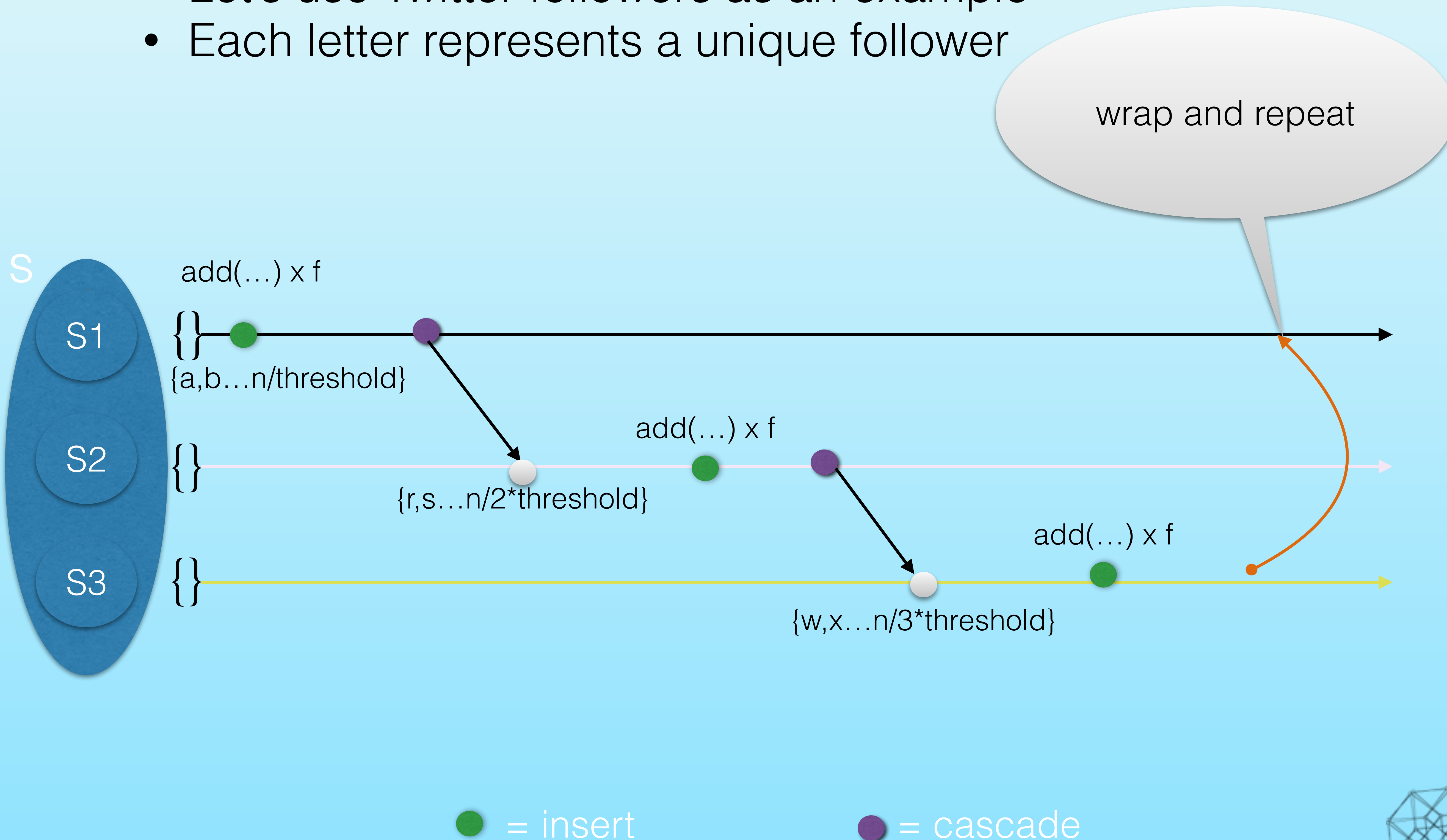
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



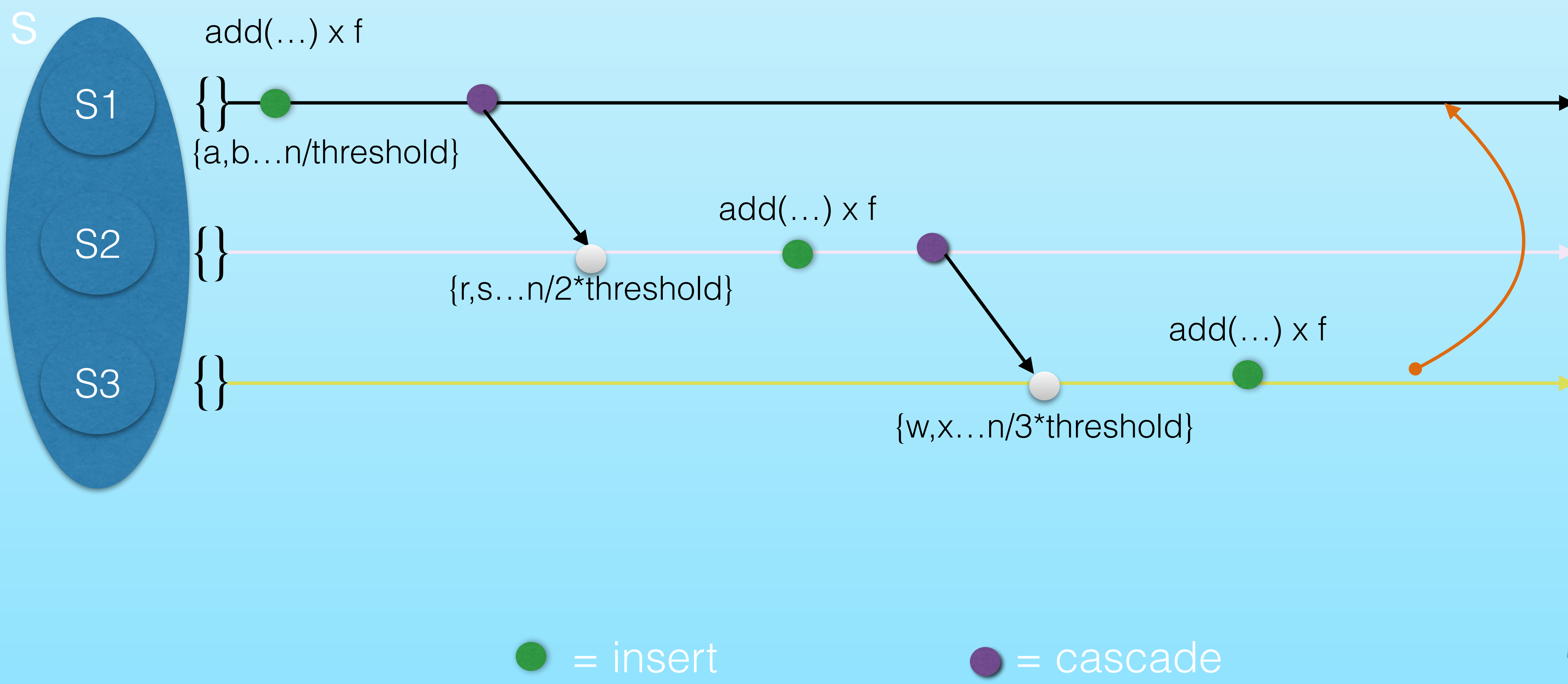
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower



Aims of the Tesseract

1. Implement distributed eventually consistent graph database

2. Develop a distributed graph partitioning algorithm

3. Develop a computational model able to support both real time and batch processing on a distributed graph

Distributed computation

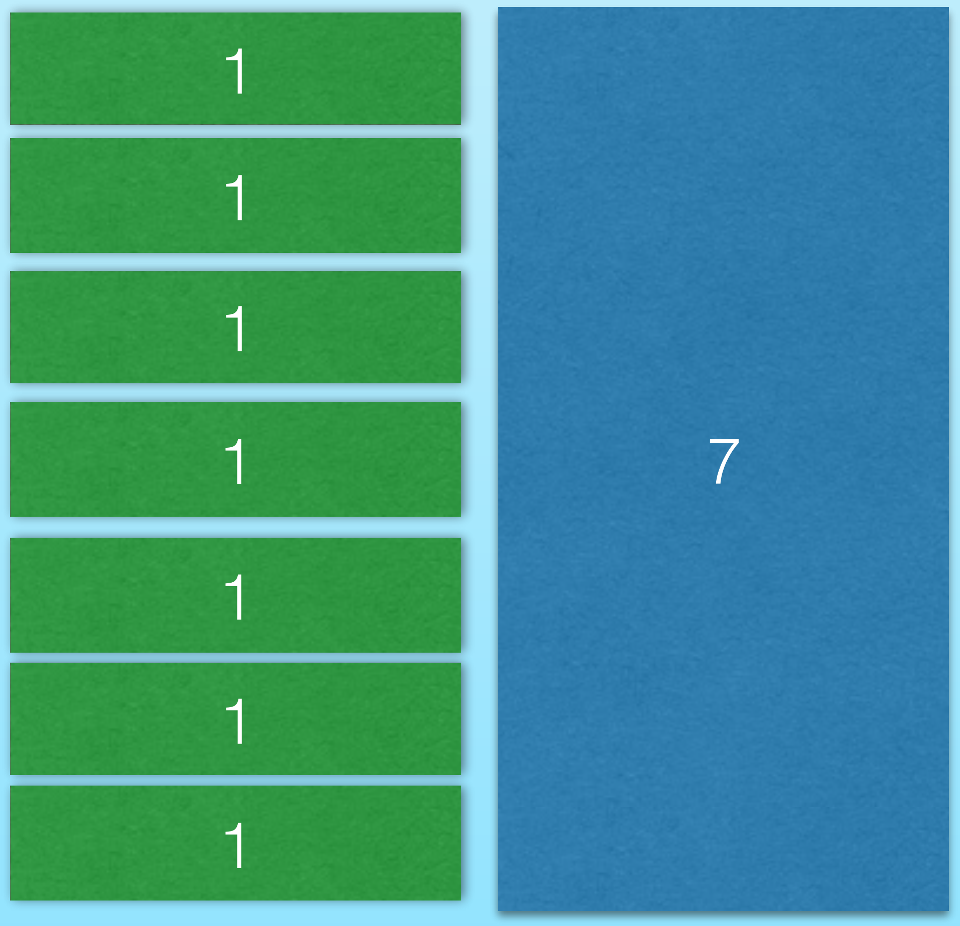
Localised calculations

Amortisation

Memoization

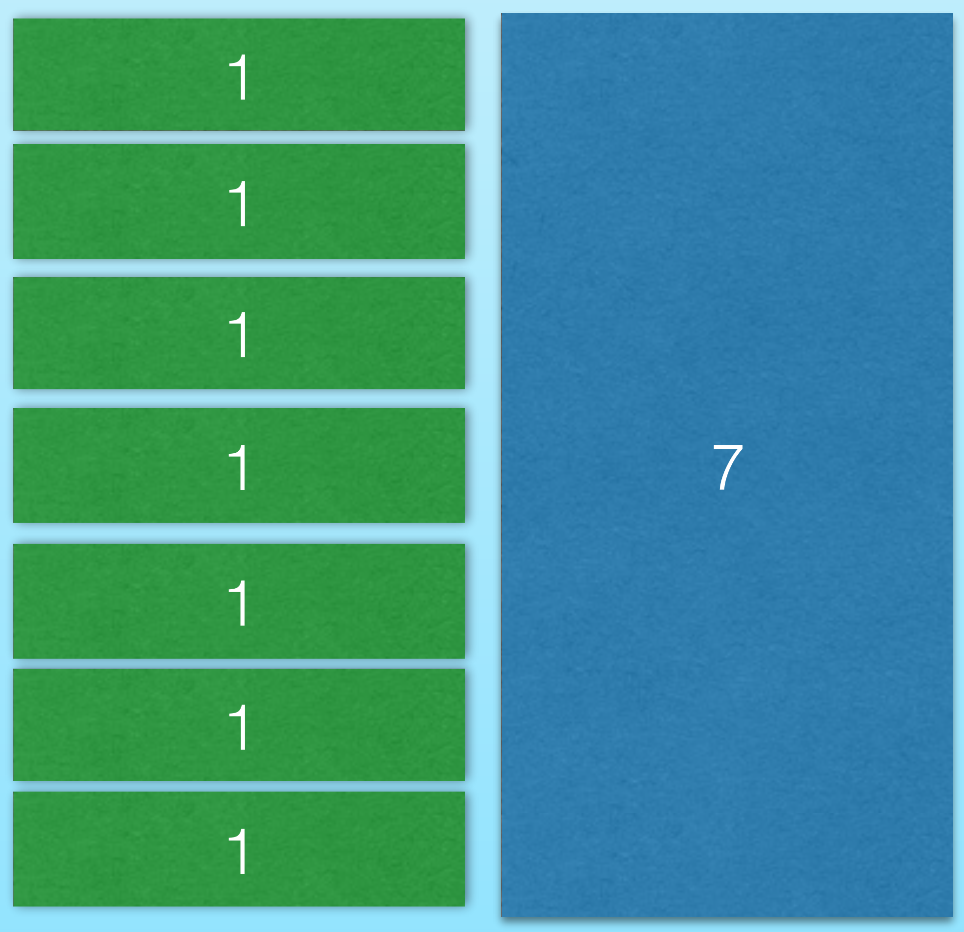
Amortisation

- Optimise to perform more “cheap” computations
- This allows us to occasionally pay the cost of more “expensive” operations such that they computationally balance out
- e.g. Checking data locally on a node vs querying over a network



Amortisation

- Optimise to perform more “cheap” computations
- This allows us to occasionally pay the cost of more “expensive” operations such that they computationally balance out
- e.g. Checking data locally on a node vs querying over a network



Memoization

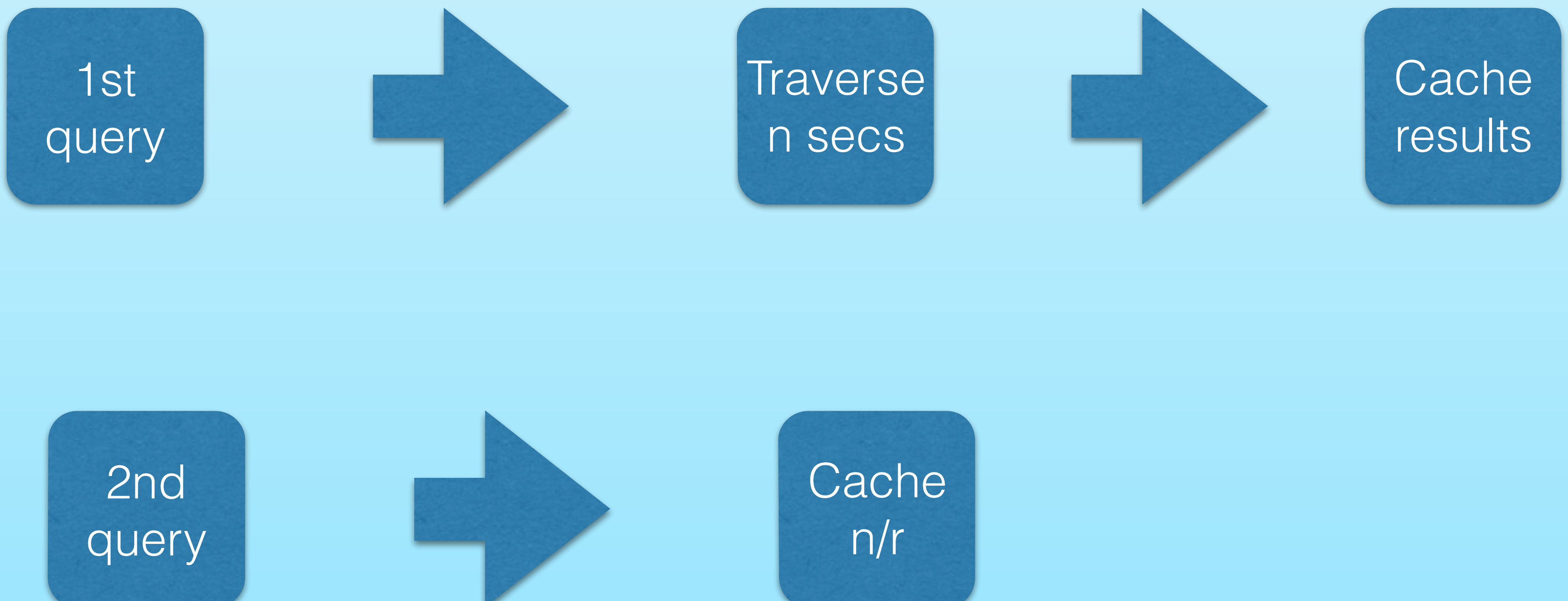
- Cache the results of computations
  - A luxury afforded by immutability
- Sacrifices disk space and memory
- Provides improved query performance

Memoization

- Cache the results of computations
  - A luxury afforded by immutability
- Sacrifices disk space and memory
- Provides improved query performance

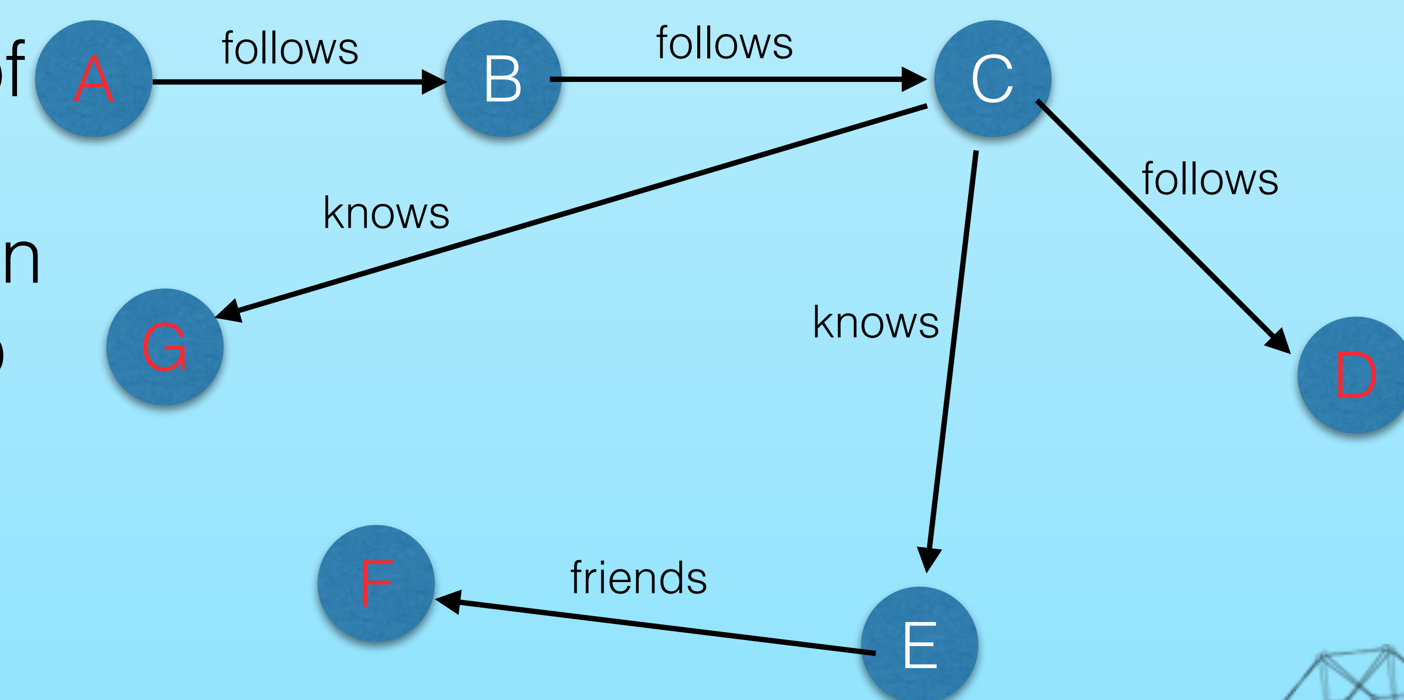
Memoization

- Cache the results of computations
    - A luxury afforded by immutability
- Sacrifices disk space and memory
- Provides improved query performance



Wormhole traversals

- Immutability offers guarantees
- Place markers at every n vertex intervals
- When traversing, don't visit every vertex, jump to markers instead.
- Markers at A, G, F, D
- By pass B,C,E during traversal, almost halving the time.
- The resulting data has any skipped vertex asynchronously fetched
- A key part of this is in the use of "Path summaries"
- Path summary is an optimisation that enables the runtime to skip network requests
- Allows traversal to continue locally and async request is made to gather the remote results



zcourts.com

Going functional

Going functional

- Early implementation was in Haskell

Going functional

- Early implementation was in Haskell
- Why? Because it did everything I wanted.

Going functional

- Early implementation was in Haskell
- Why? Because it did everything I wanted.
- Later realised it's not Haskell in particular I wanted
  - …but its semantics
    - Immutability
    - Purity
    - and some other stuff
    - and, well…functions!

Going functional

- Early implementation was in Haskell
- Why? Because it did everything I wanted.
- Later realised it's not Haskell in particular I wanted
  - …but its semantics
    - Immutability
    - Purity
    - and some other stuff
    - and, well…functions!
- The whole graph thing is an optimisation problem
  - The properties of a purely functional language enables a run time to make a lot of assumptions
  - These assumptions open possibilities not otherwise available (some times by allowing us to pretend a problem isn't there)

Distributed Query Model: TQL, Tesseract Query Language

- Haskell?
- …before you start sneaking out the back doors
- What would that even look like…?

Distributed Query Model: TQL, Tesseract Query Language

- Haskell?
- …before you start sneaking out the back doors
- What would that even look like…?

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                               E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                               E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

Distributed Query Model: TQL, Tesseract Query Language

- Haskell?
- …before you start sneaking out the back doors
- What would that even look like…?

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                                  E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                                  E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

- What you're looking at is TQL

Distributed Query Model: TQL, Tesseract Query Language

- Haskell?
- …before you start sneaking out the back doors
- What would that even look like…?

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                              E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                              E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

- What you're looking at is TQL
  - a pure

Distributed Query Model: TQL, Tesseract Query Language

- Haskell?
- …before you start sneaking out the back doors
- What would that even look like…?

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                              E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                              E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

- What you're looking at is TQL
  - a pure
  - functional language

Distributed Query Model: TQL, Tesseract Query Language

- Haskell?
- …before you start sneaking out the back doors
- What would that even look like…?

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                              E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                              E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

- What you're looking at is TQL
  - a pure
  - functional language
  - it has type inferencing and all the cool functional widgets!

Distributed Query Model: TQL pt2

Distributed Query Model: TQL pt2

- How was that functional?

Distributed Query Model: TQL pt2

- How was that functional?
- It employed use of:
  - Functions - relation between a set of input and a set of permissible outputs
  - Monads - structures that allow you to define computation in terms of the steps necessary to obtain the results of the computation.
  - Monoids - a set with a single associative (1+ 2) + 3 == 1 + (2+3) binary operation an identity element (an element where, when applied to any other in the set, the value of the other element remains unchanged. e.g. given * as the binary operation and the set S={1,2,3}, 1 is the identity element since 1 * 1 = 1, 2 * 1 = 2 and 3 * 1 = 3)
  - Currying - where a function which takes multiple arguments is converted into a series of functions which take a single argument, the currying technique produces partially applied functions.
  - Higher order functions - functions which takes other functions as its parameter
  - Function composition - the process of making the result of one function the argument of another

Distributed Query Model: TQL pt2

- How was that functional?
- It employed use of:
  - Functions - relation between a set of input and a set of permissible outputs
  - Monads - structures that allow you to define computation in terms of the steps necessary to obtain the results of the computation.
  - Monoids - a set with a single associative (1+ 2) + 3 == 1 + (2+3) binary operation an identity element (an element where, when applied to any other in the set, the value of the other element remains unchanged. e.g. given * as the binary operation and the set S={1,2,3}, 1 is the identity element since 1 * 1 = 1, 2 * 1 = 2 and 3 * 1 = 3)
  - Currying - where a function which takes multiple arguments is converted into a series of functions which take a single argument, the currying technique produces partially applied functions.
  - Higher order functions - functions which takes other functions as its parameter
  - Function composition - the process of making the result of one function the argument of another
- Don’t believe me? Let’s look at a definition for “INSERT” shown on the previous slide

Distributed Query Model: TQL pt3

Distributed Query Model: TQL pt3

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Distributed Query Model: TQL pt3

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Function name

Distributed Query Model: TQL pt3

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Function name

Graph namespace

Distributed Query Model: TQL pt3

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Function name

Graph namespace

Vertex type

Distributed Query Model: TQL pt3

Edge type

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Function name

Graph namespace

Vertex type

Distributed Query Model: TQL pt3

… = var-arg
+ Homogeneous

Edge type

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Function name

Graph namespace

Vertex type

Distributed Query Model: TQL pt3

… = var-arg
+ Homogeneous

Edge type

Result of “INSERT”

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Function name

Graph namespace

Vertex type

Distributed Query Model: TQL pt3

… = var-arg
+ Homogeneous

Edge type

Result of “INSERT”

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Result of lambda function

Function name

Vertex type

Graph namespace

Distributed Query Model: TQL pt3

… = var-arg
+ Homogeneous

Edge type

Result of “INSERT”

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Result of lambda function

Function name

Graph namespace

Vertex type

- Lambda function you say?

Distributed Query Model: TQL pt3

… = var-arg
+ Homogeneous

Edge type

Result of “INSERT”

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Result of lambda function

Function name

Vertex type

Graph namespace

- Lambda function you say?
- Where, where?

Distributed Query Model: TQL pt3

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

- Lambda function you say?
- Where, where?

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                              E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                              E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

- Lambda function you say?
- Where, where?

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                                 E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                                 E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                                E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                                E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

From here…

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                               E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                               E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

From here…

…to here!

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                                E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                                E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                              E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                              E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                               E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                               E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                                E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                                E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
- Built on top of LLVM

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                                E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                                E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
- Built on top of LLVM
- TQL comes with a useful “standard” library like most languages

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                                E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                                E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
- Built on top of LLVM
- TQL comes with a useful “standard” library like most languages
- An “Algorithms & machine learning” module will ship as an add-on module

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                                 E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                                 E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
- Built on top of LLVM
- TQL comes with a useful “standard” library like most languages
- An “Algorithms & machine learning” module will ship as an add-on module
- Ability to define new modules/add or override functions

Distributed Query Model: TQL pt3

```
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
                                 E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
                                 E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

INSERT :: ((String -> (V…) -> (E…) -> PartialTransform)) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
- Built on top of LLVM
- TQL comes with a useful “standard” library like most languages
- An “Algorithms & machine learning” module will ship as an add-on module
- Ability to define new modules/add or override functions
- Include additional modules (yours or a third party’s)

Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations
- Client interface remains consistent
- While on going research can improve the run time without major client changes

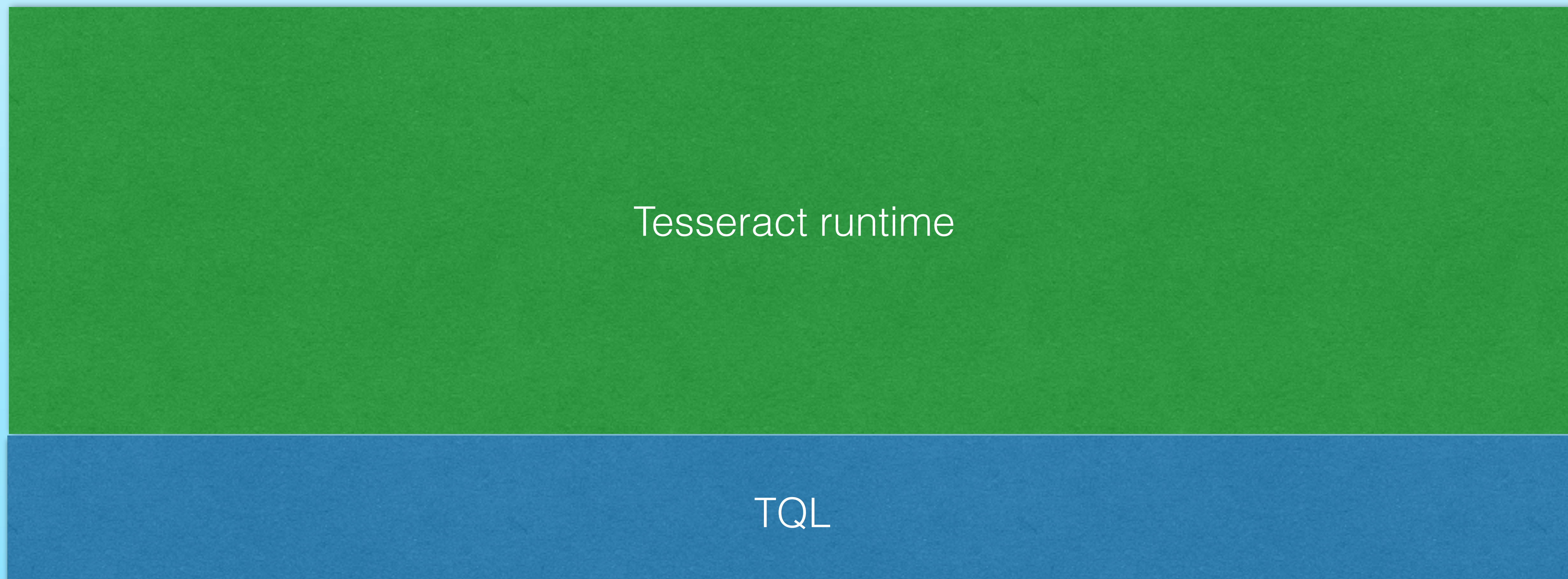
Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations
- Client interface remains consistent
- While on going research can improve the run time without major client changes

Tesseract runtime

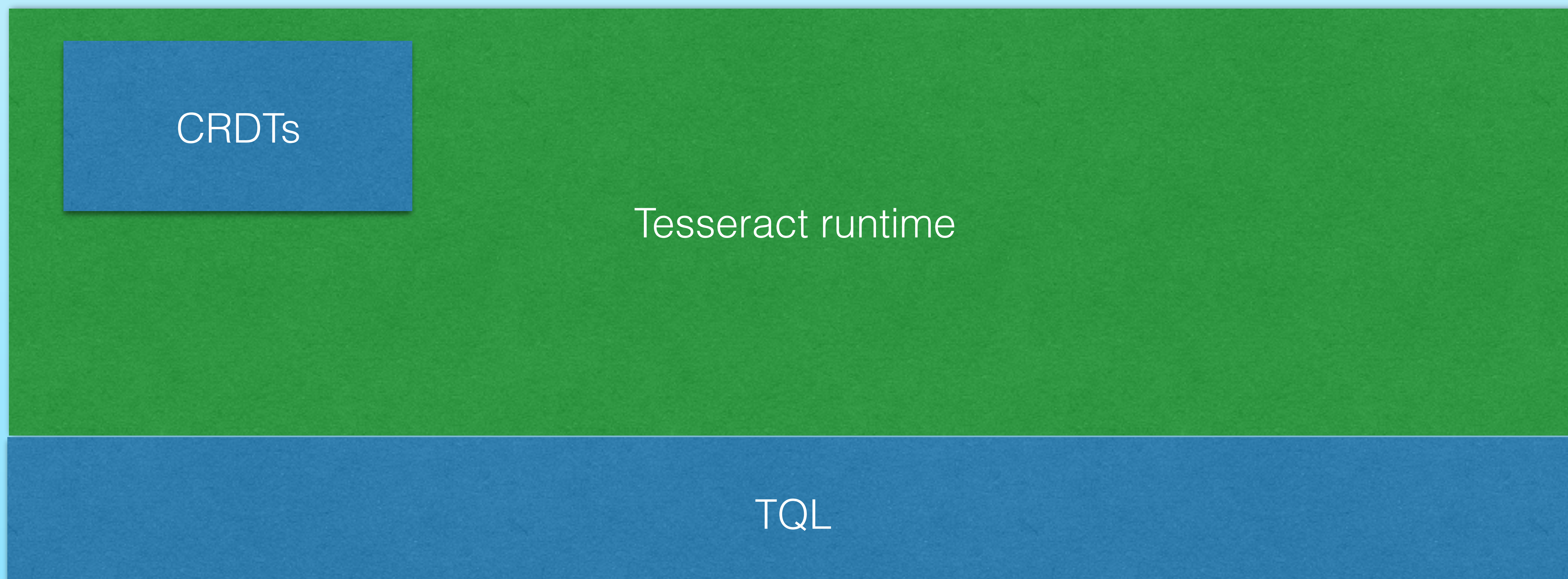
Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations
- Client interface remains consistent
- While on going research can improve the run time without major client changes



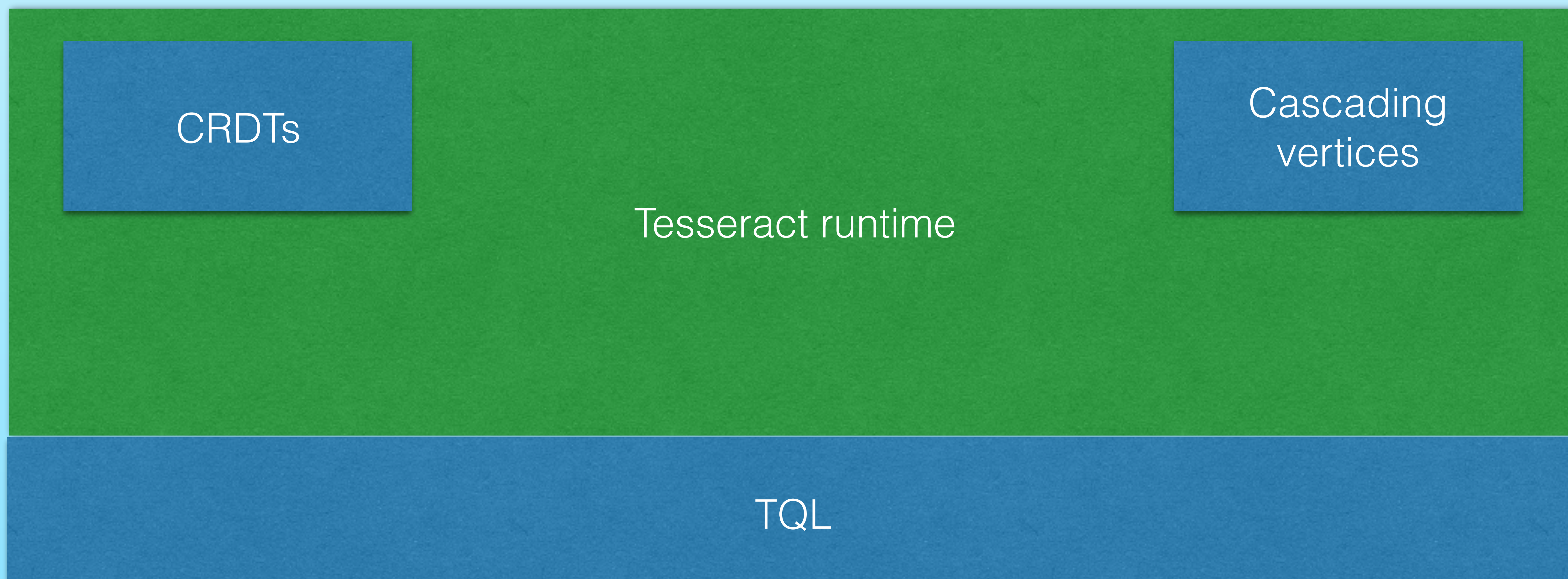
Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations
- Client interface remains consistent
- While on going research can improve the run time without major client changes



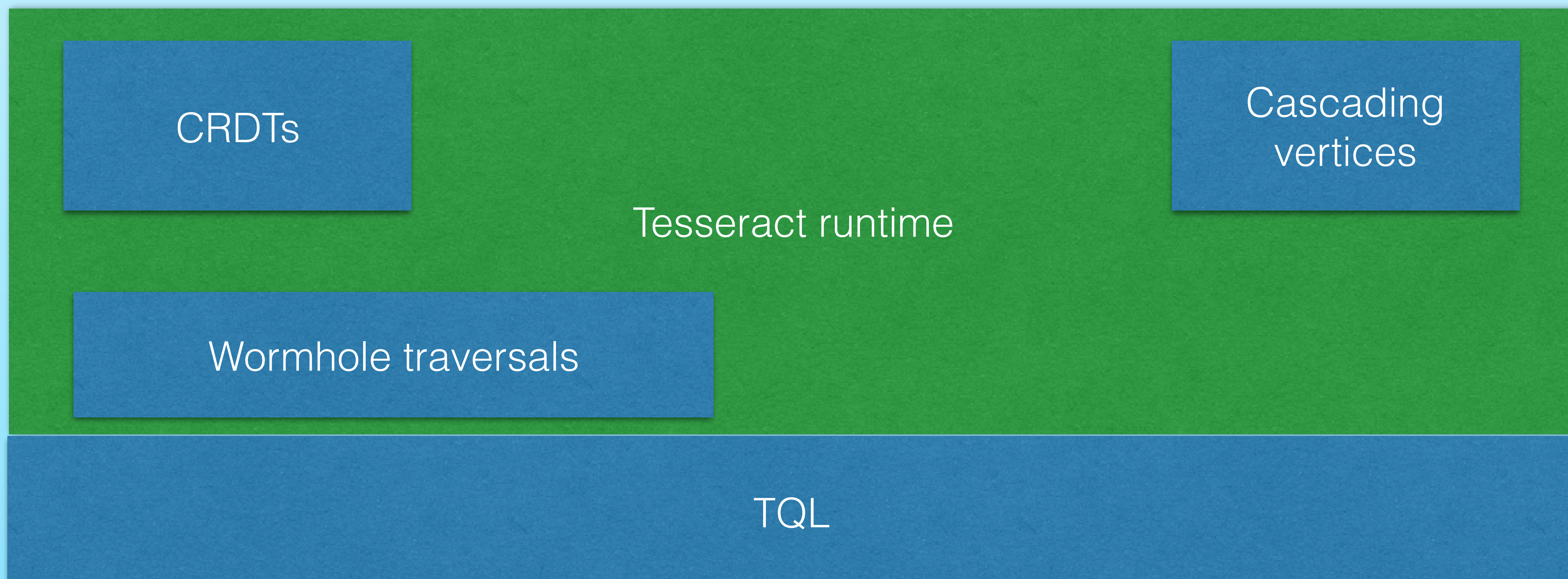
Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations
- Client interface remains consistent
- While on going research can improve the run time without major client changes



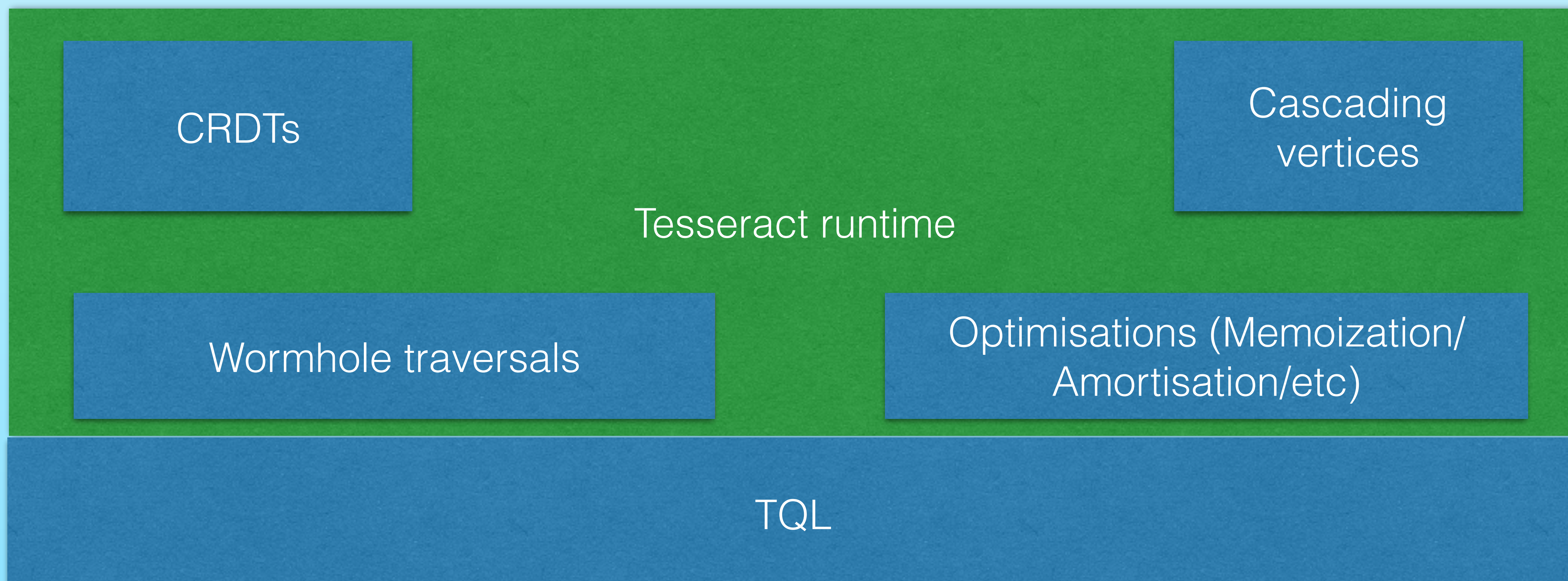
Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations
- Client interface remains consistent
- While on going research can improve the run time without major client changes



Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations
- Client interface remains consistent
- While on going research can improve the run time without major client changes



Compaction & Garbage collection

Compaction & Garbage collection

- Immutability means we store data that's no longer needed i.e. garbage

Compaction & Garbage collection

- Immutability means we store data that's no longer needed i.e. garbage
- CRDTs can accumulate a large amount of garbage
  - This can be avoided by not keeping tombstones at all
  - Without tombstones the system is unable to do a consistent snapshot
  - If snapshots are disabled, tombstones are not needed
  - Short synchronisation are used out of the query path to do some clean up (currently evaluating RAFT for GC consensus)

Compaction & Garbage collection

- Immutability means we store data that's no longer needed i.e. garbage
- CRDTs can accumulate a large amount of garbage
  - This can be avoided by not keeping tombstones at all
  - Without tombstones the system is unable to do a consistent snapshot
  - If snapshots are disabled, tombstones are not needed
  - Short synchronisation are used out of the query path to do some clean up (currently evaluating RAFT for GC consensus)
- Current work is modelled off of JVM's generational collectors

Compaction & Garbage collection

- Immutability means we store data that's no longer needed i.e. garbage
- CRDTs can accumulate a large amount of garbage
  - This can be avoided by not keeping tombstones at all
  - Without tombstones the system is unable to do a consistent snapshot
  - If snapshots are disabled, tombstones are not needed
  - Short synchronisation are used out of the query path to do some clean up (currently evaluating RAFT for GC consensus)
- Current work is modelled off of JVM's generational collectors
- Algorithm needs more investigation…

Compaction & Garbage collection

- Immutability means we store data that's no longer needed i.e. garbage
- CRDTs can accumulate a large amount of garbage
  - This can be avoided by not keeping tombstones at all
  - Without tombstones the system is unable to do a consistent snapshot
  - If snapshots are disabled, tombstones are not needed
  - Short synchronisation are used out of the query path to do some clean up (currently evaluating RAFT for GC consensus)
- Current work is modelled off of JVM's generational collectors
- Algorithm needs more investigation…
- Compaction also serves as an opportunity to optimise data location
  - Write only means vertex properties and edges aren't always next to each other in a data file
  - During compaction we re-arrange contents
  - Helps reduce the amount of work required by spindle disks to fetch a vertex's data

First release due in 2-3 months

Will be Apache v2 Licensed

github.com/zcourts/Tesseract

End…

Questions?

Courtney Robinson
Google “zcourts”
courtney@zcourts.com

github.com/zcourts/Tesseract