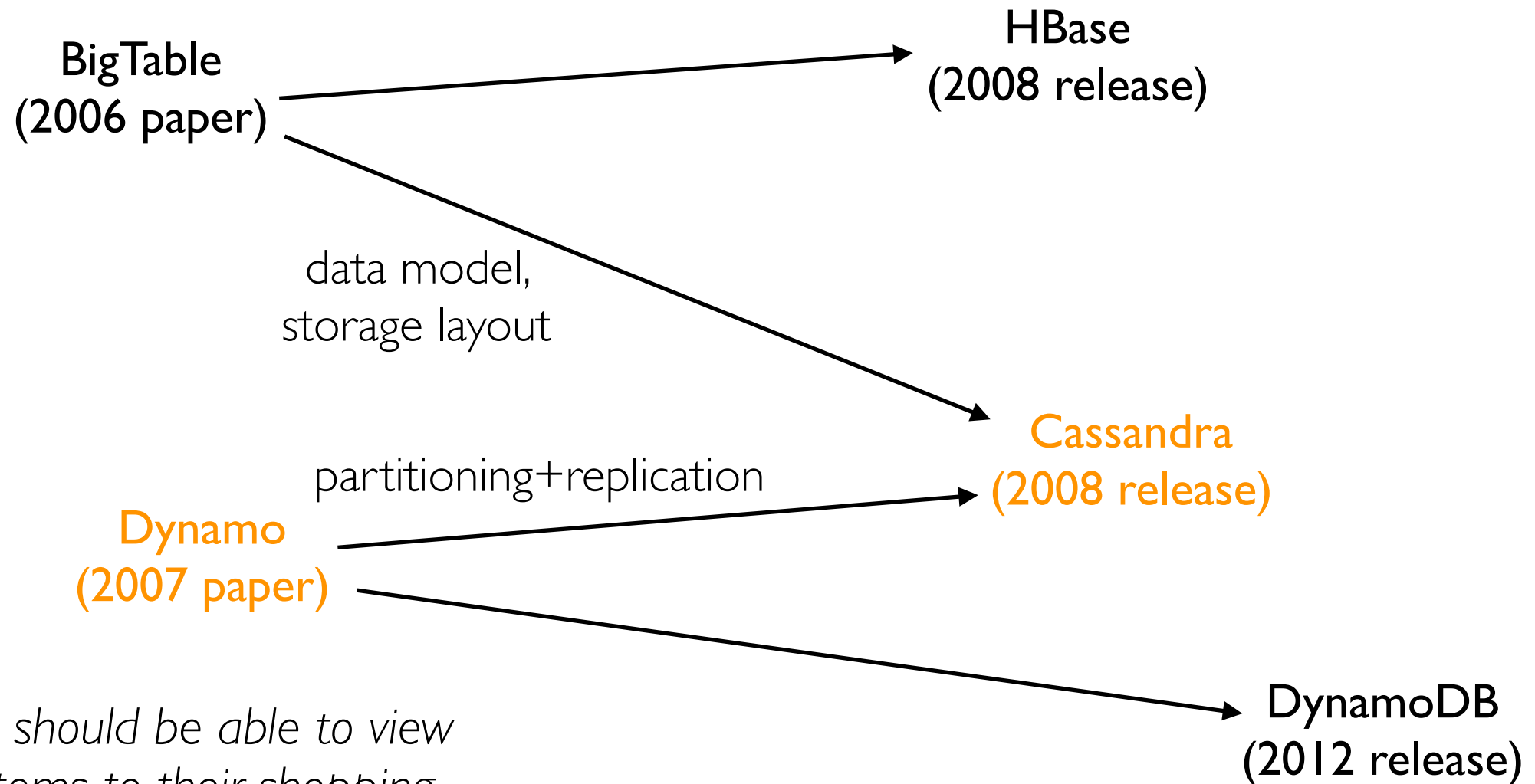


[544] Cassandra Partitioning+Replication

Tyler Caraza-Harter

Cassandra Influences



*"customers should be able to view
and add items to their shopping
cart even if disks are failing, network
routes are flapping, or data centers
are being destroyed by tornados"*
~ authors of first Dynamo paper

goal: highly available when things are failing

Outline: Cassandra Partitioning+Replication

Partitioning

Replication

Quorum Reads/Writes

Conflict Resolution

Cassandra Demos

Partitioning Approaches

Given many machines and a partition of data, *how do we decide where it should live?*

Mapping Data Structure

- locations = {"fileA-block0": [datanode1, ...], ...}
- **HDFS** NameNode uses this

Hash Partitioning

- partition = $\text{hash}(\text{key}) \% \text{partition_count}$
- **Spark** shuffle uses this (for grouping, joining, etc); data structures associate partitions with worker machines

Consistent Hashing

- **Dynamo** and **Cassandra** uses this

Review: HDFS Partitioning

My Laptop

client program

NameNode

F1.1: nodes 1, 2, 3

F2.1: nodes 1, 2

F2.2: nodes 2, 3

**block
locations**

"ABCD" (F1.1)

"EFGH" (F2.1)

"ABCD" (F1.1)

"EFGH" (F2.1)

"IJKL" (F2.2)

"IJKL" (F2.2)

"ABCD" (F1.1)

**DataNode
Computers**

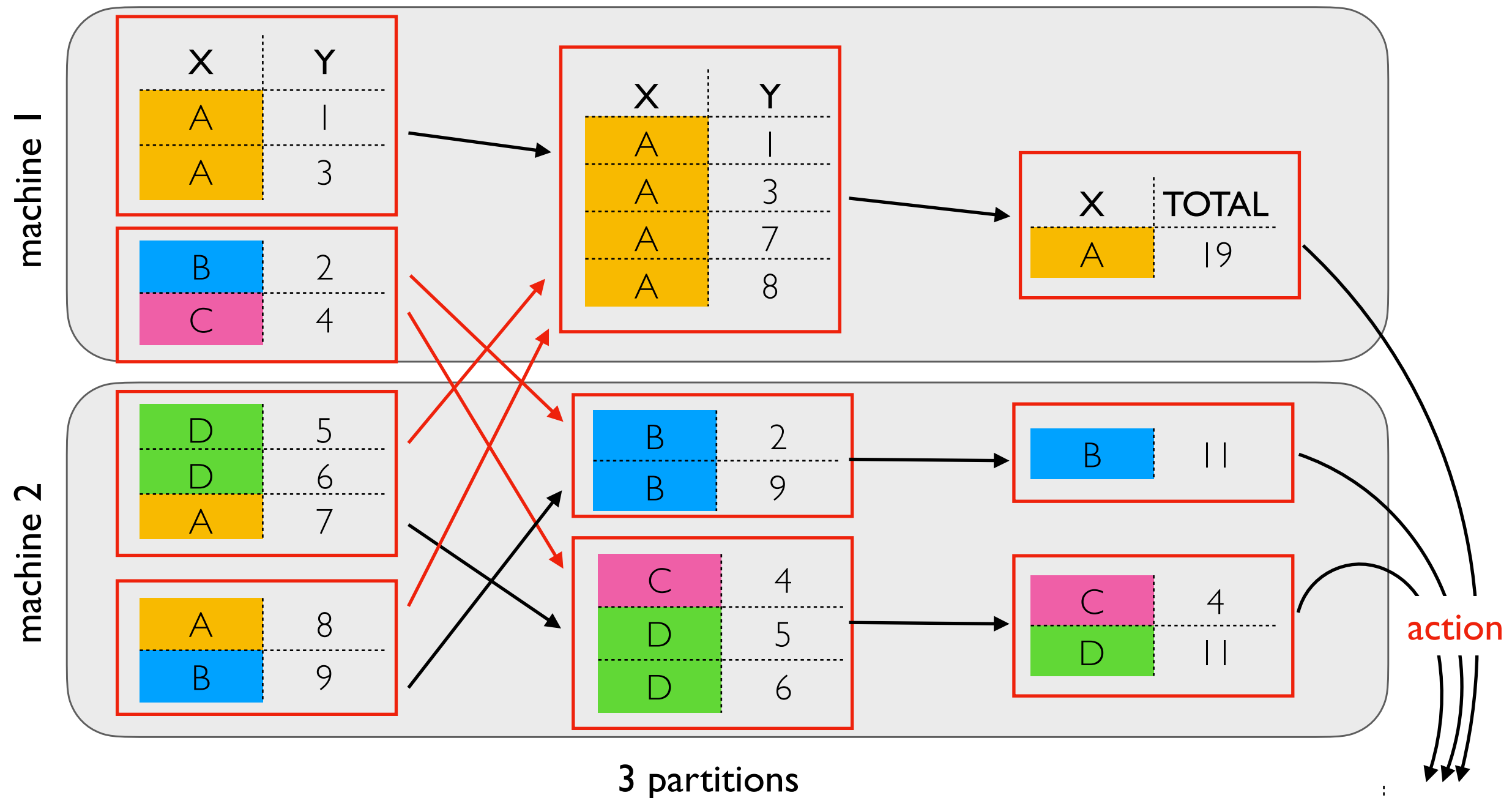
DN1

DN2

DN3

Review: Spark Hash Partitioning

□ partition



3 partitions

```
row = Row(X=D, Y=5)
partition = hash(row.X) % 3 # partition=2
```

file or
Pandas DF

X	TOTAL
A	19
B	11
C	4
D	11

Discuss Scalability: HDFS and Spark

Scalability: we can make efficient use of many machines for big data

Two ways we can have big data:

- **very many** file blocks or rows
- files or tables containing **many bytes of data**

Will HDFS struggle with either kind of big data? Spark?

Elasticity: Easily Growing/Shrinking Clusters

Incremental Scalability: can we efficiently add more machines to an already large cluster?

What happens when we **add a new DataNode** to an HDFS cluster?

What would need to happen if we able to **add an RDD partition** in the middle of a Spark hash-partitioned shuffle?

Elasticity: Easily Growing/Shrinking Clusters

Incremental Scalability: can we efficiently add more machines to an already large cluster?

What happens when we **add a new DataNode** to an HDFS cluster?

What would need to happen if we able to **add an RDD partition** in the middle of a Spark hash-partitioned shuffle?

Demo: hash partition 26 letters over 4 "machines".
Add a 5th machine. How many letters must move?

Partitioning Approaches

Given many machines and a partition of data, *how do we decide where it should live?*

Mapping Data Structure

- locations = {"fileA-block0": [datanode1, ...], ...}
- **HDFS** NameNode uses this

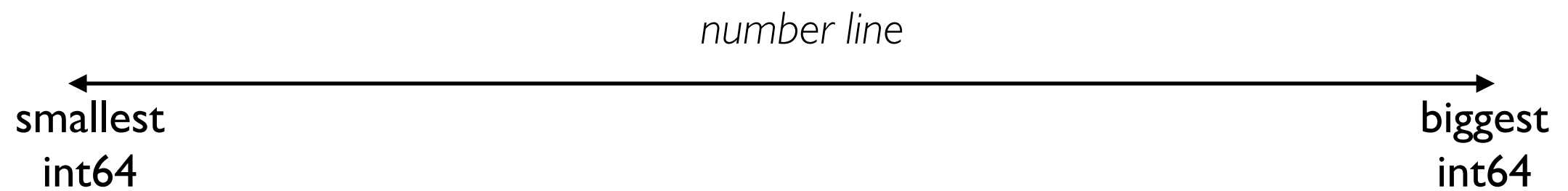
Hash Partitioning

- partition = $\text{hash}(\text{key}) \% \text{partition_count}$
- **Spark** shuffle uses this (for grouping, joining, etc); data structures associate partitions with worker machines

Consistent Hashing

- **Dynamo** and **Cassandra** uses this
- token = $\text{hash}(\text{key})$ # every token is in a range, indicating the worker
- locations = {range(0,10): "worker1", range(10,20): "worker2", ...}

Consistent Hashing



Consistent Hashing

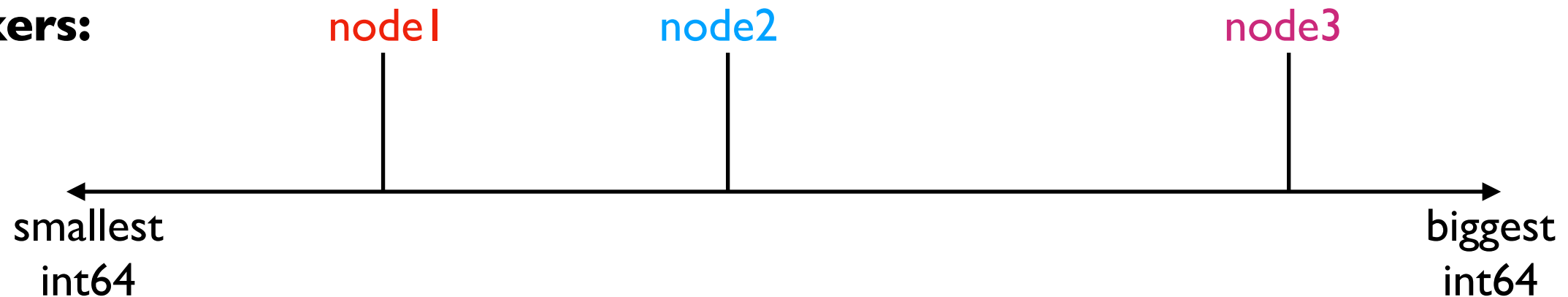
Token Map:

token(node1) = pick something

token(node2) = pick something

token(node3) = pick something

workers:



assign every **worker** a point on the number line.
Could be random (though newer approaches are more clever).
No hashing needed yet!

Consistent Hashing

Token Map:

$\text{token}(\text{node1}) = \text{pick something}$

$\text{token}(\text{node2}) = \text{pick something}$

$\text{token}(\text{node3}) = \text{pick something}$

workers:

node1

node2

node3

← smallest
int64

→ biggest
int64

rows:

A

B

C

D

E

assign every **row** a point on the number line.

$\text{token}(\text{row}) = \text{hash}(\text{row's partition key})$

Consistent Hashing

Token Map:

token(node1) = pick something

token(node2) = pick something

token(node3) = pick something

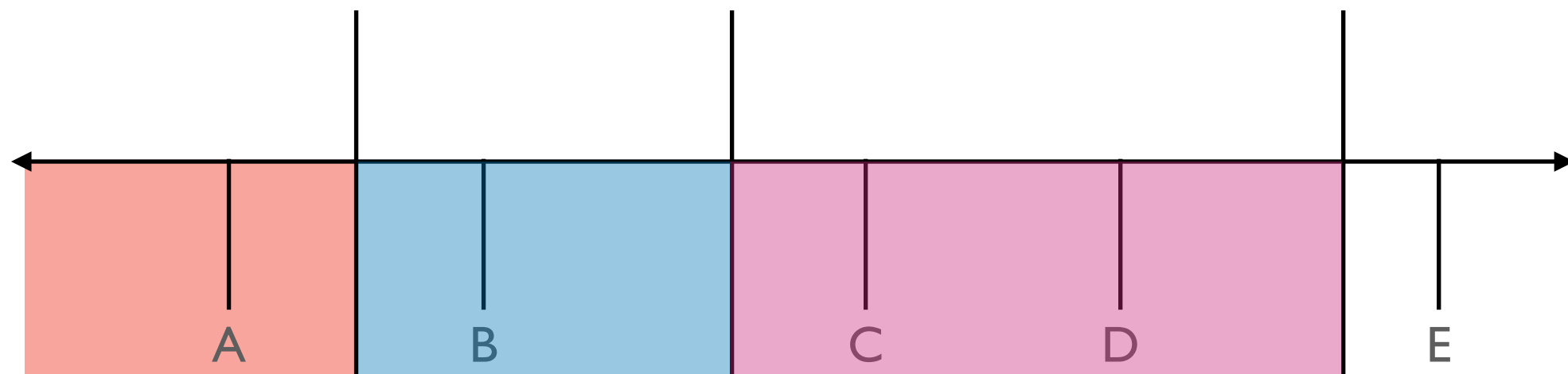
workers:

node1

node2

node3

rows:



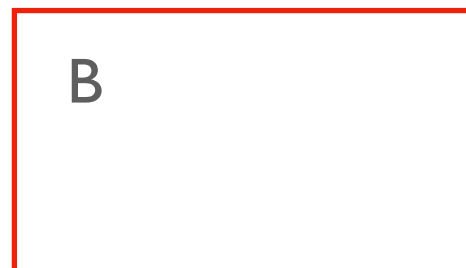
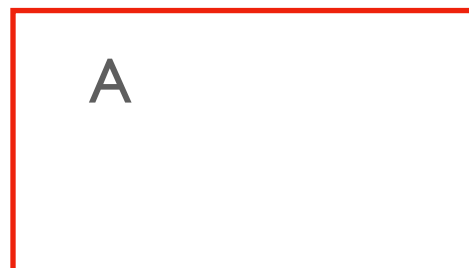
each node's token is the **inclusive end** of a range.
A row is mapped to a node based on the range it is in.

cluster:

node1

node2

node3



Consistent Hashing

Token Map:

token(node1) = pick something

token(node2) = pick something

token(node3) = pick something

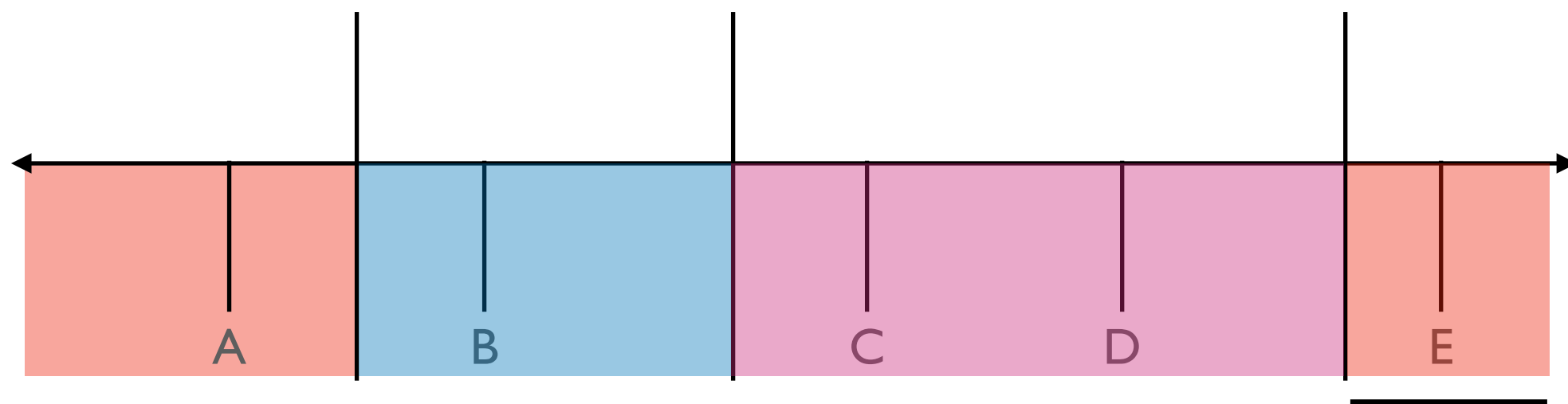
workers:

node1

node2

node3

rows:



tokens > biggest node token are in the *wrapping range*. Rows in this region go to the node with the smallest token.

cluster:

node1

node2

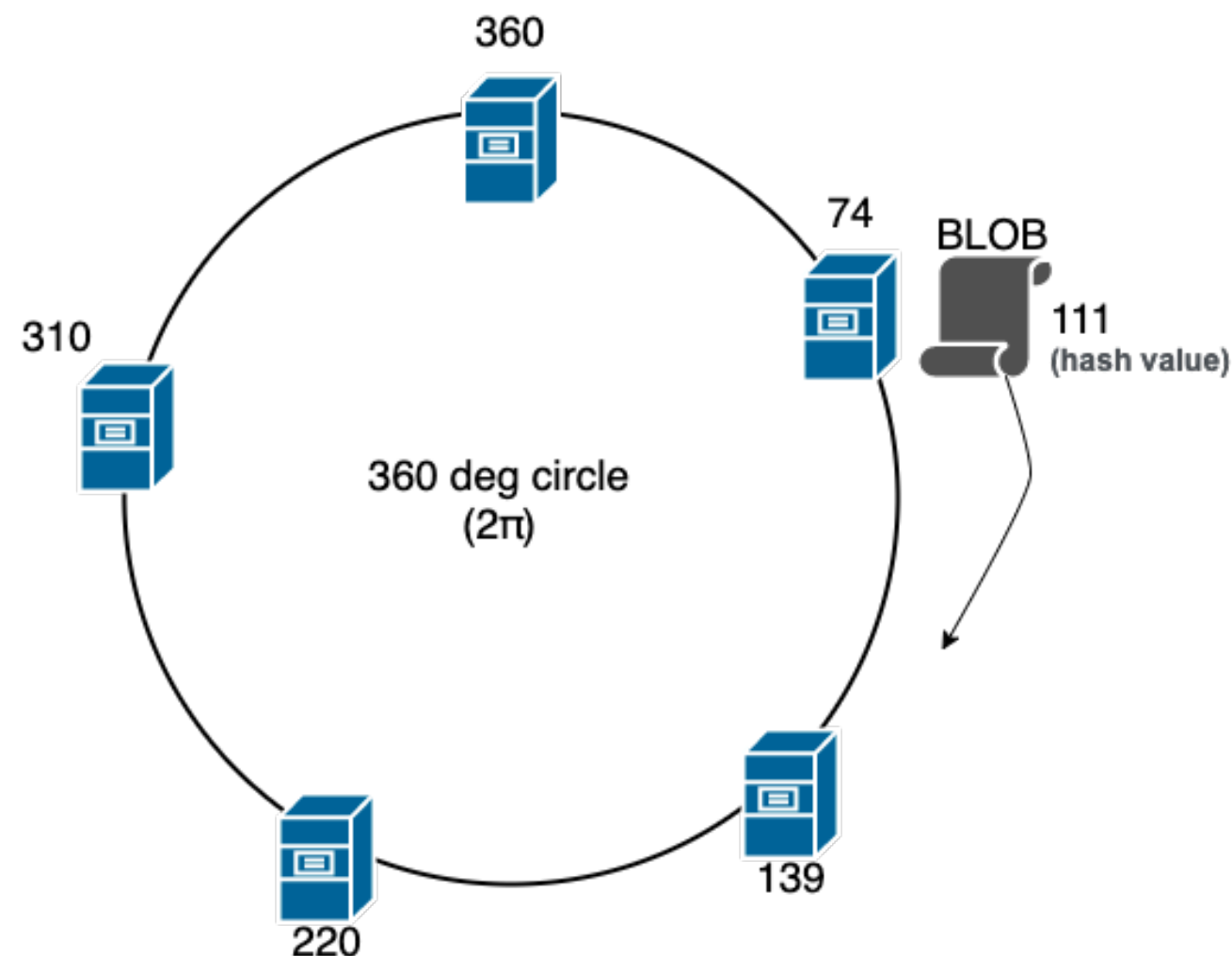
node3



Alternate Visualization

Given the wrapping, clusters using consistent hashing are called "token rings"

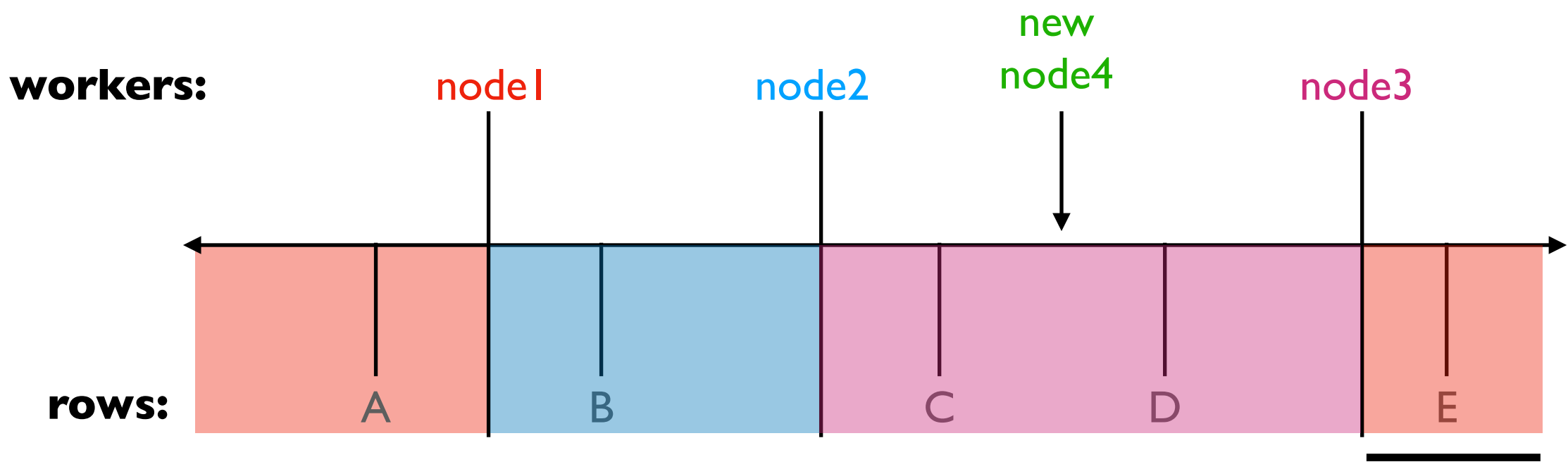
Common visuazilation (e.g., from Wikipedia)



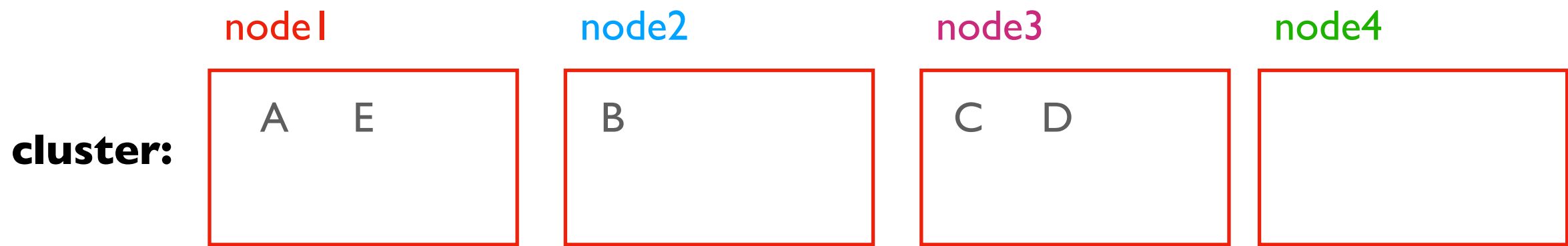
Adding a Node

Token Map:

token(node1) = pick something
token(node2) = pick something
token(node3) = pick something
token(node4) = pick something



which rows will have to move?
which nodes will be involved?



Adding a Node

Token Map:

token(node1) = pick something

token(node2) = pick something

token(node3) = pick something

token(node4) = pick something

workers:

node1

node2

node4

node3

rows:

A

B

C

D

E

which rows will have to move?
which nodes will be involved?

cluster:

node1

node2

node3

node4

A

E

B

D

C

Adding a Node

Token Map:

token(node1) = pick something

token(node2) = pick something

token(node3) = pick something

token(node4) = pick something

workers:

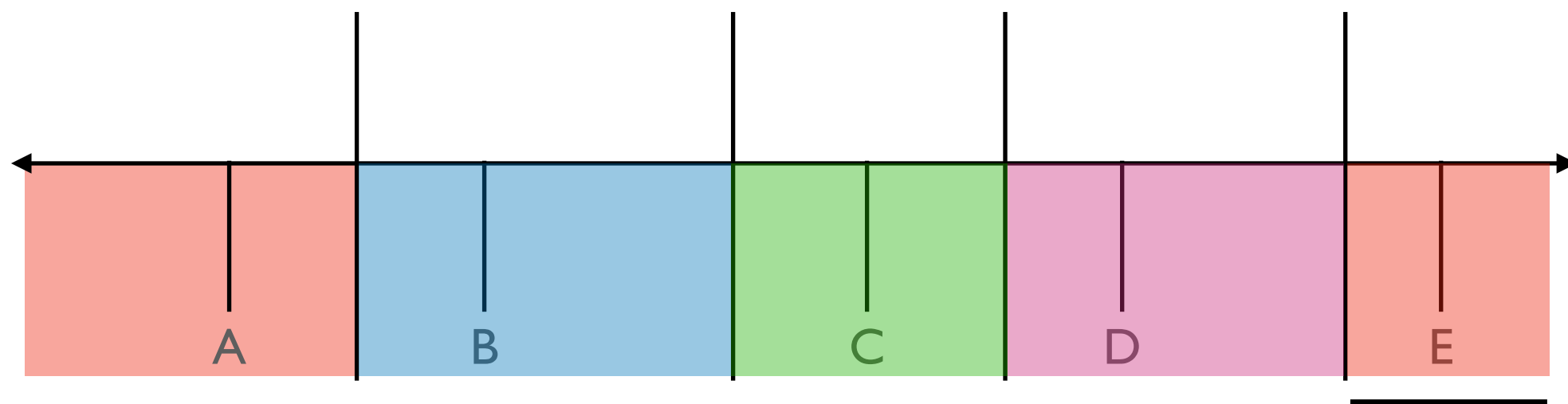
node1

node2

node4

node3

rows:



Typically, what fraction of the data must move when we scale from $N-1$ to N ?

Hash partitioning: about $(N-1)/N$ of the data

Consistent hashing: about $(\text{size of new range})/(\text{size of combined range})$ of the data must move.

Adding a Node

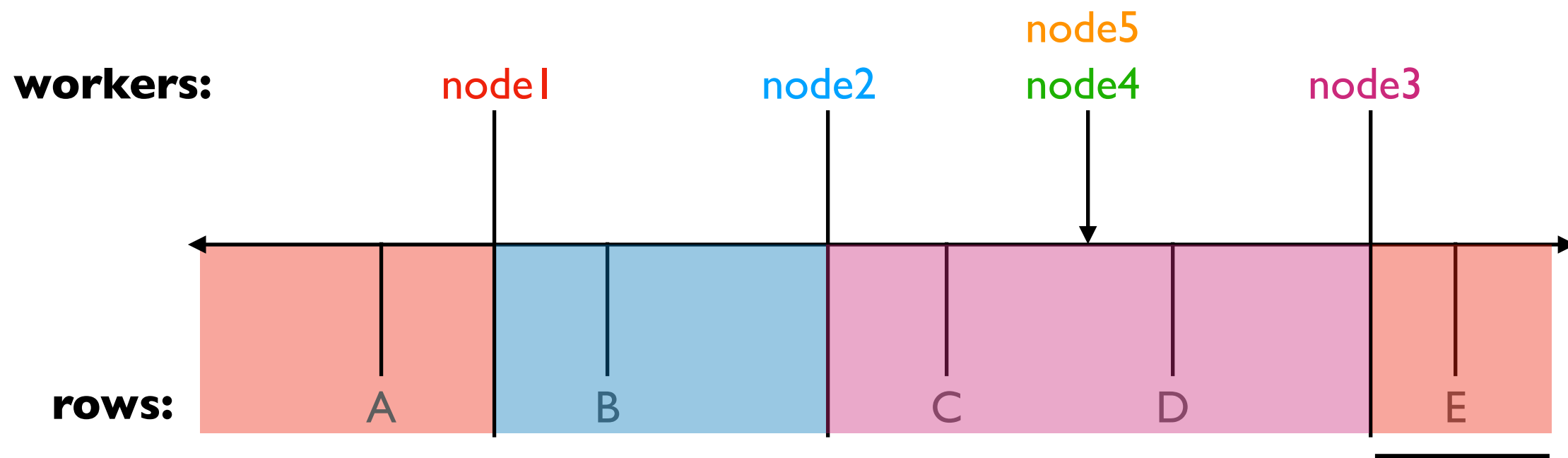
Token Map:

token(node1) = pick something

token(node2) = pick something

token(node3) = pick something

token(node4) = pick something



Problem: latest Cassandra versions by default try to choose new node tokens to split big ranges for better balance (instead of randomly picking). Adding multiple nodes simultaneously can lead to collisions, preventing nodes from joining.

Solution: add one at a time (after initial "seed" nodes)

Adding a Node

Token Map:

token(node1) = pick something

token(node2) = pick something

token(node3) = pick something

token(node4) = pick something

workers:

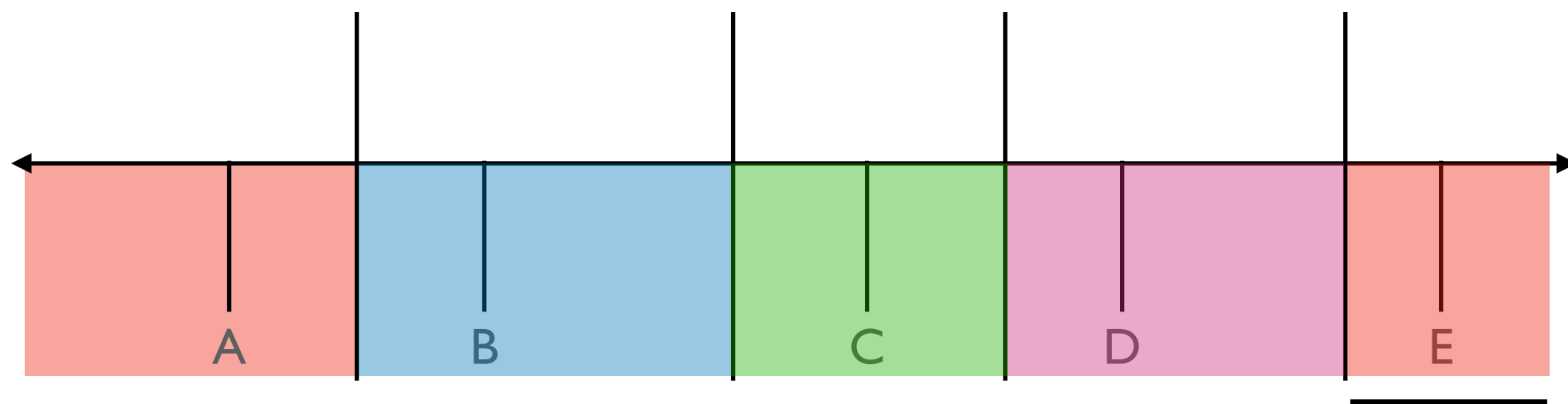
node1

node2

node4

node3

rows:



Other problems with adding node 4

- **long term:** only load of node 3 is alleviated
- **short term:** node 3 bears all the burden of transferring data to node 4

Solution: "vnodes"

Virtual Nodes (vnodes)

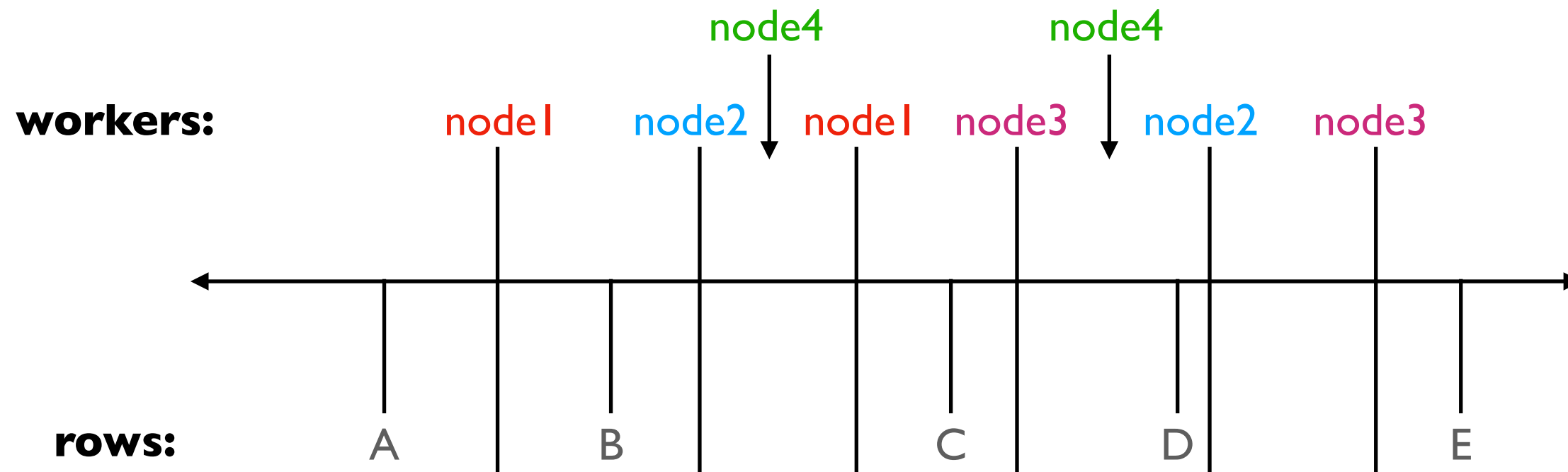
Token Map:

$\text{token}(\text{node1}) = \{t1, t2\}$

$\text{token}(\text{node2}) = \{t3, t4\}$

$\text{token}(\text{node3}) = \{t5, t6\}$

$\text{token}(\text{node4}) = \{t7, t8\}$



Each node is responsible for multiple ranges

- how many is configurable
- node 4 will take some load off nodes 1 and 2 (those to the right of its vnodes)

Token Map Storage

Token Map:

$\text{token}(\text{node1}) = \{t1, t2\}$

$\text{token}(\text{node2}) = \{t3, t4\}$

$\text{token}(\text{node3}) = \{t5, t6\}$

where should this live?



we don't want a single point of failure
(like an HDFS NameNode)

Token Map Storage

node1

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

node2

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

node3

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

every node has a copy of the token map

they should all get updated when new nodes join

Adding Nodes: Bad Approach

uh oh, node 3 won't know about node 4 when it comes back

node1

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

node2

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

node3

table rows

...lots of data...

rebooting...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

node4

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

Better Approach: Gossip

just inform one or a few nodes
about the new one

node1

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

node2

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

node3

table rows

...lots of data...

rebooting...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

node4

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

Better Approach: Gossip

once per second:
choose a random friend
gossip about new nodes

node1

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

node2

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

"have you heard
about node 4?"



node3

table rows

...lots of data...

rebooting...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

node4

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

Better Approach: Gossip

eventually, every node should find out

node1

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}



node2

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

node3

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

node4

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

Better Approach: Gossip

when a client wants to write a row,
they can contact any node -- it should
know where the data should live and
coordinate the operation

node1

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

node2

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

node3

table rows

...lots of data...

Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

node4 (coordinator)

table rows

...lots of data...

Token Map:

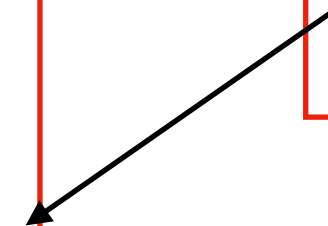
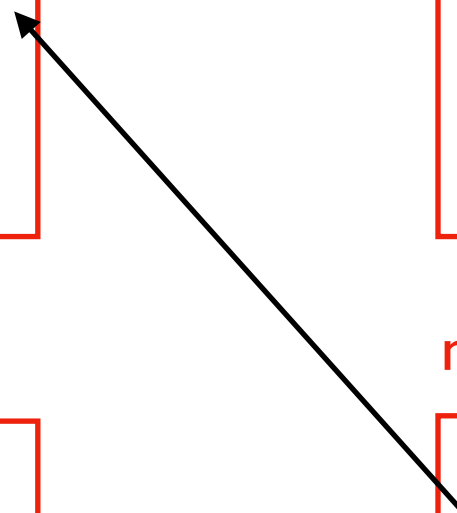
token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}

client



TopHat, Worksheet

Outline: Cassandra Partitioning+Replication

Partitioning

Replication

Quorum Reads/Writes

Conflict Resolution

Cassandra Demos

Replication

We *replicate* (create multiple copies on different nodes) to improve *durability* – meaning we don't want data to be lost when nodes die.

Cassandra lets us choose a different **RF** (replication factor) for each keyspace:

```
create keyspace X
with replication={'class': 'SimpleStrategy',
                  'replication_factor': 2};
```

```
create keyspace Y
with replication={'class': 'SimpleStrategy',
                  'replication_factor': 3};
```


Replication

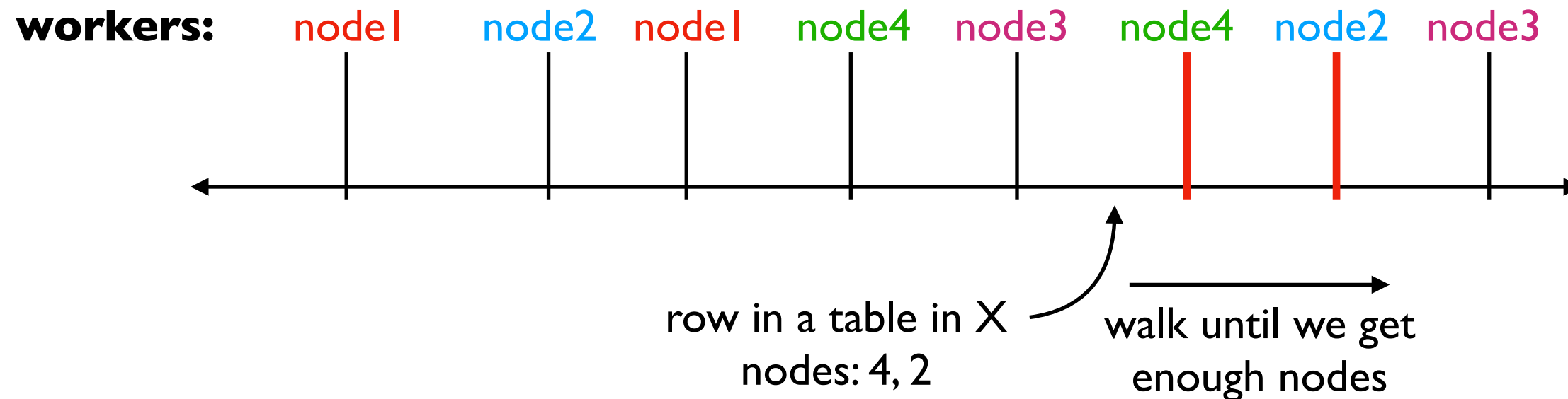
Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}



```
create keyspace X
with replication={'class': 'SimpleStrategy',
                  'replication_factor': 2};
```

```
create keyspace Y
with replication={'class': 'SimpleStrategy',
                  'replication_factor': 3};
```

Replication

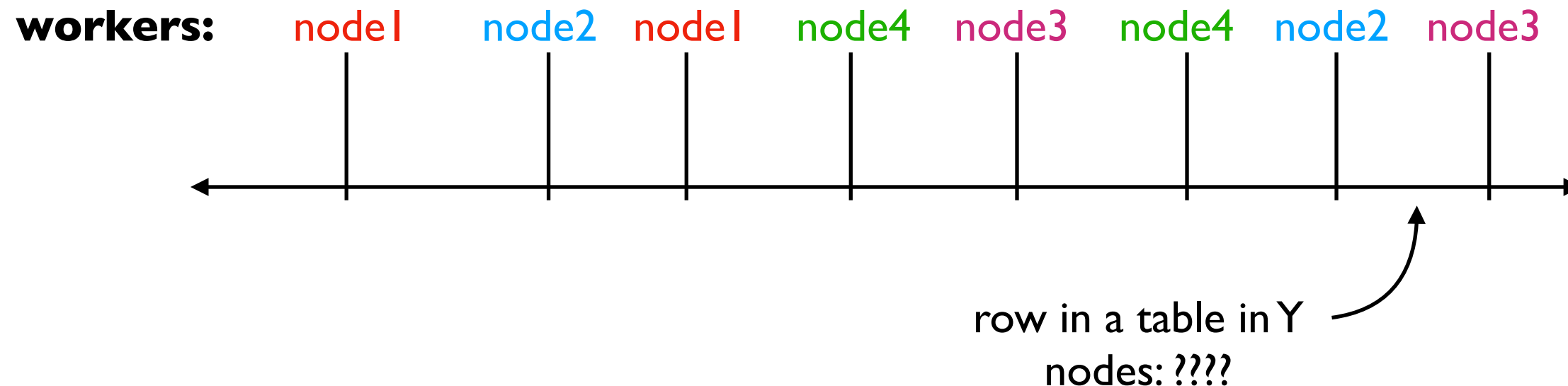
Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}



```
create keyspace X
with replication={'class': 'SimpleStrategy',
                  'replication_factor': 2};
```

```
create keyspace Y
with replication={'class': 'SimpleStrategy',
                  'replication_factor': 3};
```

Replication

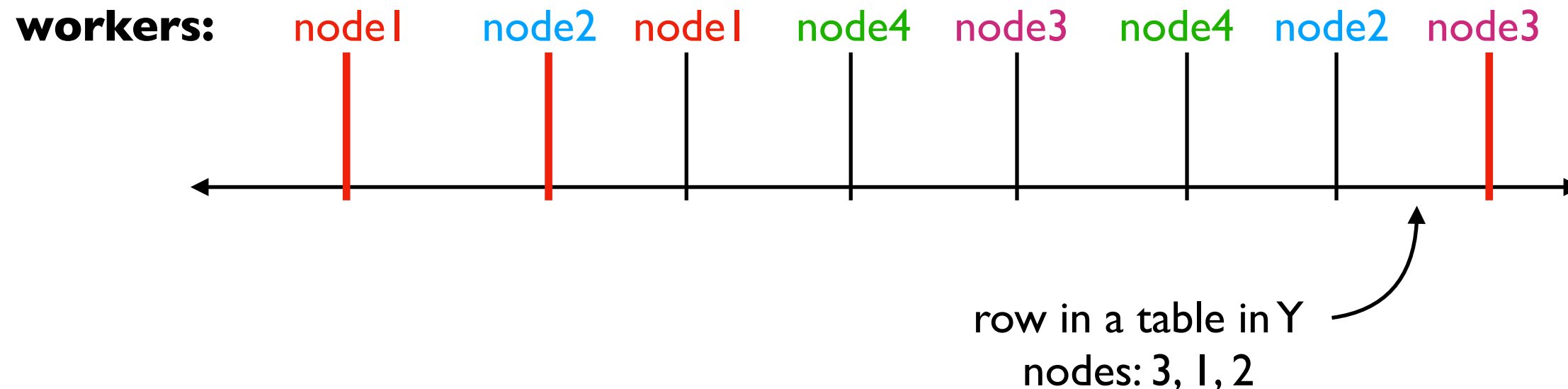
Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}



```
create keyspace X
with replication={'class': 'SimpleStrategy',
                  'replication_factor': 2};
```

```
create keyspace Y
with replication={'class': 'SimpleStrategy',
                  'replication_factor': 3};
```

Replication

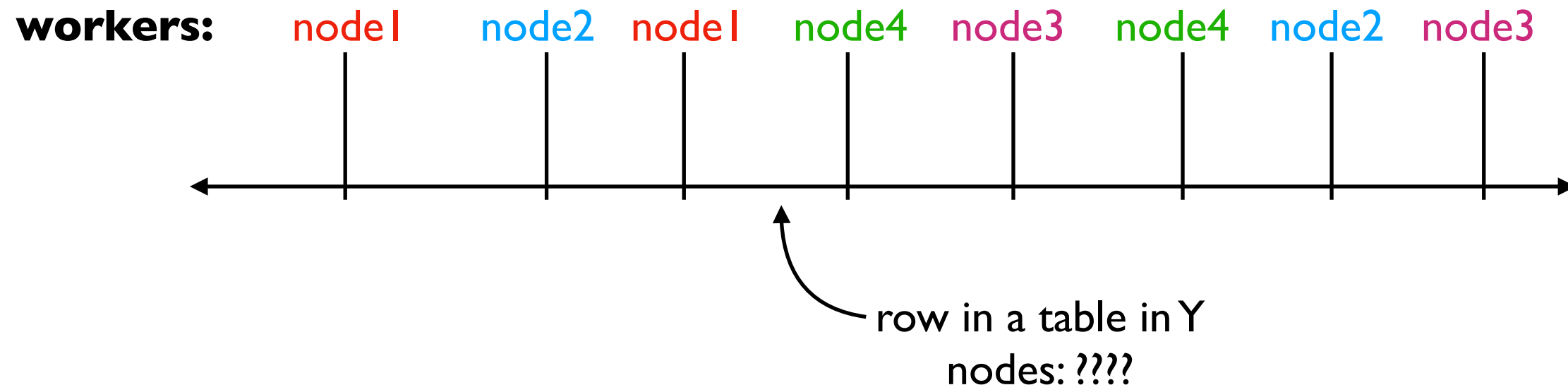
Token Map:

token(node1) = {t1, t2}

token(node2) = {t3, t4}

token(node3) = {t5, t6}

token(node4) = {t7, t8}



```
create keyspace X
with replication={'class': 'SimpleStrategy',
                  'replication_factor': 2};
```

```
create keyspace Y
with replication={'class': 'SimpleStrategy',
                  'replication_factor': 3};
```

Replication

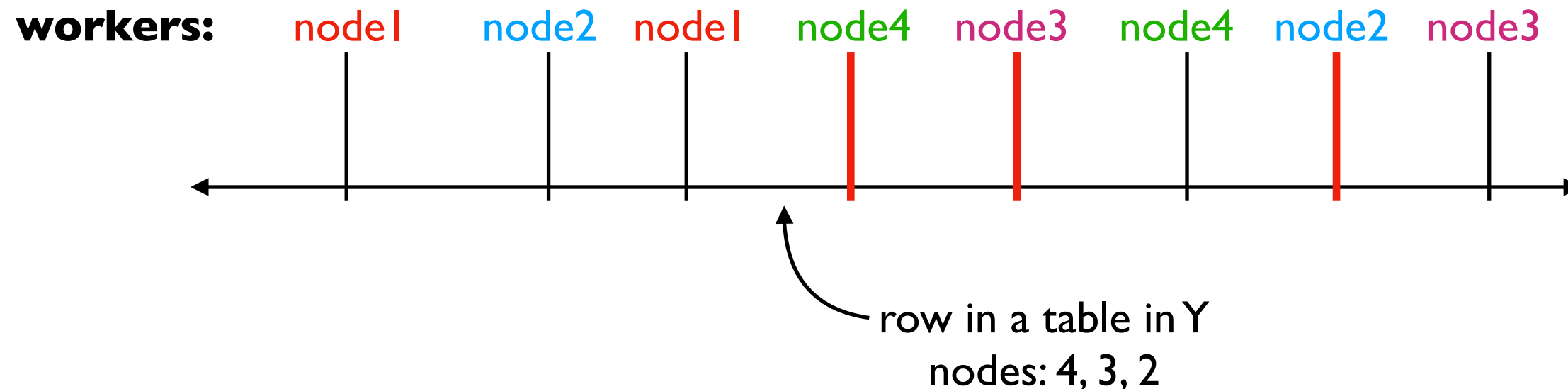
Token Map:

$\text{token}(\text{node1}) = \{t1, t2\}$

$\text{token}(\text{node2}) = \{t3, t4\}$

$\text{token}(\text{node3}) = \{t5, t6\}$

$\text{token}(\text{node4}) = \{t7, t8\}$



Important! Keeping multiple copies on vnodes on the same node provides little safety (when a node dies, all its vnodes die).

Cassandra can skip nodes as it "walks the ring".

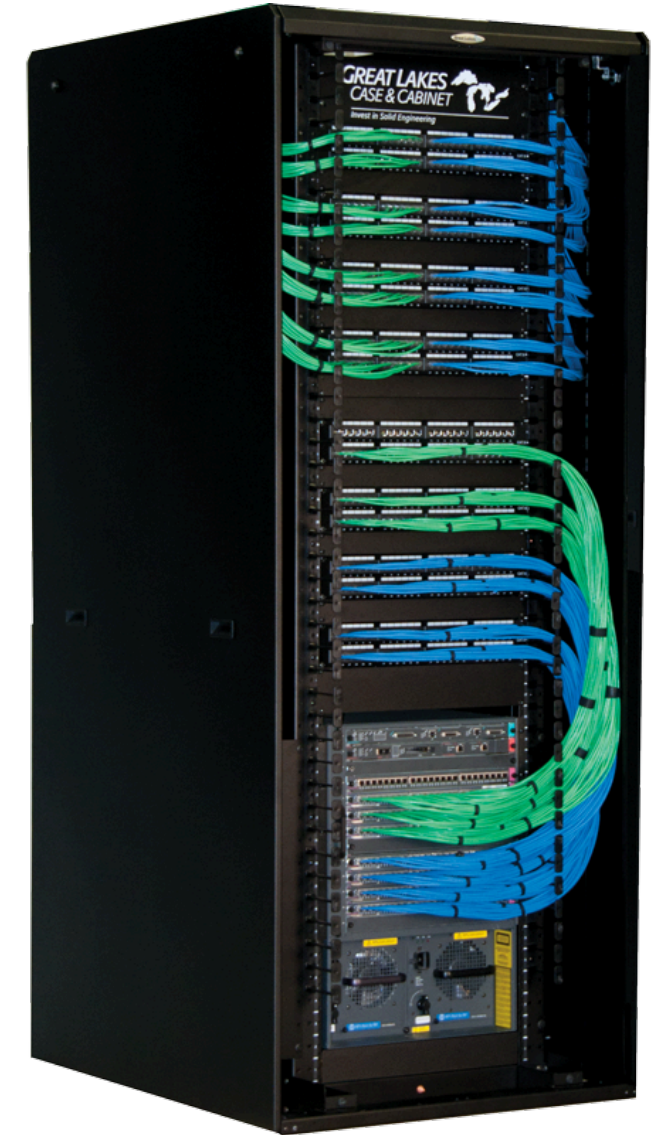
Network Infrastructure



Server



Data Center



Rack

<https://www.dotmagazine.online/issues/digital-infrastructure-and-transforming-markets/data-center-models>

https://buyhpe.com/us/en/servers/proliant-dl-servers/proliant-dl10-servers/proliant-dl20-server/hpe-proliant-dl20-gen10-plus-e-2336-2-9ghz-6-core-1p-16gb-u-4sff-500w-rps-server/p/p44115-b21?ef_id=Cj0KCQiAt66eBhCnARIsAKf3ZNFJsg49UV6Zm33R7lkRqi-XOd_JECmdyqNMAm2CKLSm_F-z6JTYDTQaAgMTEALw_wcB:G:s&s_kwid=AL!13472131331628972784!!!g!318267171339!!1707918369!67076417419&gclid=Cj0KCQiAt66eBhCnARIsAKf3ZNFJsg49UV6Zm33R7lkRqi-XOd_JECmdyqNMAm2CKLSm_F-z6JTYDTQaAgMTEALw_wcB

[https://www.server-rack-online.com/gl910ent-4048sss.html?](https://www.server-rack-online.com/gl910ent-4048sss.html?utm_medium=shoppingengine&utm_source=googlebase&utm_source=google&utm_medium=cpc&adpos=&scid=scplpgl910ent-4048sss&sc_intid=gl910ent-4048sss&gclid=Cj0KCQiAt66eBhCnARIsAKf3ZNEMYINPAA0RFGQIF0DsieCM6oh7i3kujVJlpnmJAIOPAJ3RWT11QMAaAqRnEALw_wcB)

[utm_medium=shoppingengine&utm_source=googlebase&utm_source=google&utm_medium=cpc&adpos=&scid=scplpgl910ent-4048sss&sc_intid=gl910ent-4048sss&gclid=Cj0KCQiAt66eBhCnARIsAKf3ZNEMYINPAA0RFGQIF0DsieCM6oh7i3kujVJlpnmJAIOPAJ3RWT11QMAaAqRnEALw_wcB](https://www.server-rack-online.com/gl910ent-4048sss.html?utm_medium=shoppingengine&utm_source=googlebase&utm_source=google&utm_medium=cpc&adpos=&scid=scplpgl910ent-4048sss&sc_intid=gl910ent-4048sss&gclid=Cj0KCQiAt66eBhCnARIsAKf3ZNEMYINPAA0RFGQIF0DsieCM6oh7i3kujVJlpnmJAIOPAJ3RWT11QMAaAqRnEALw_wcB)

Correlated Failures

One server goes down, all of its vnodes are gone.



Server

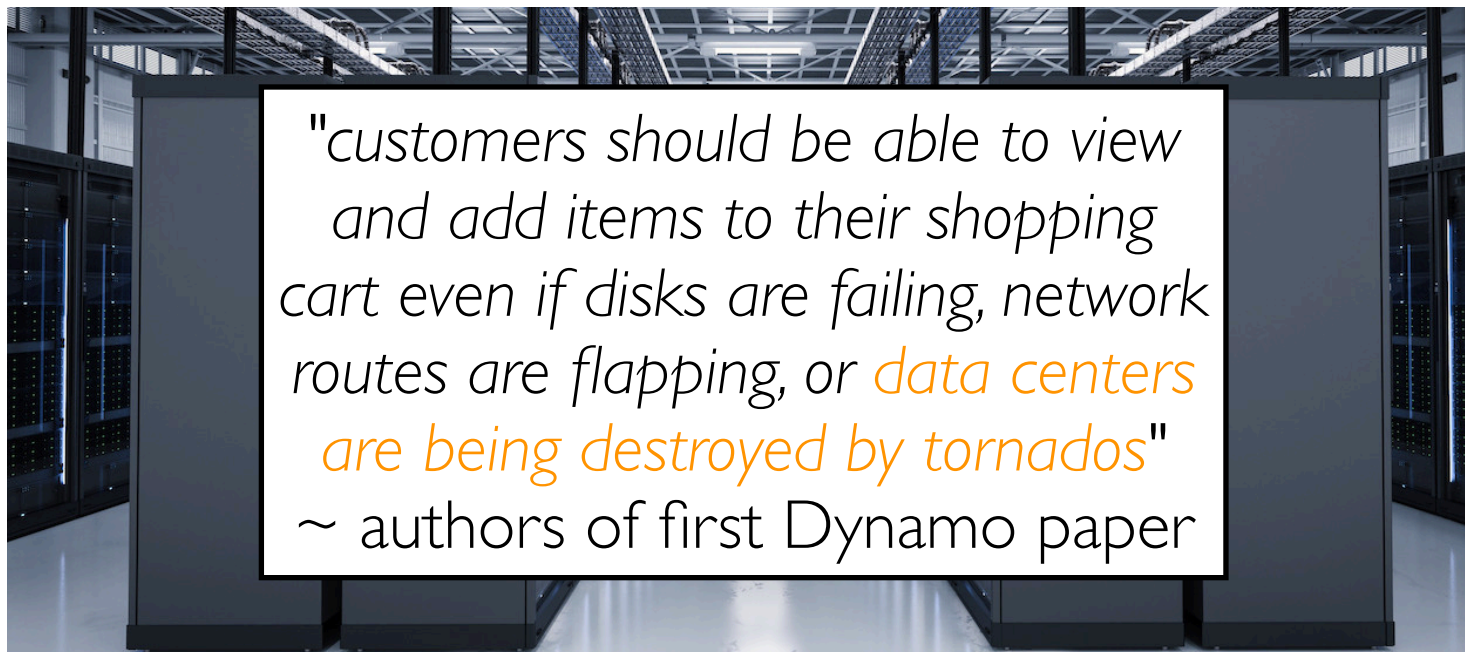
Whole-rack problems:

- top-of-rack switch fails
- rack's power supply fails



Rack

"customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados"
~ authors of first Dynamo paper



Data Center

<https://www.dotmagazine.online/issues/digital-infrastructure-and-transforming-markets/data-center-models>

https://buyhpe.com/us/en/servers/proliant-dl-servers/proliant-dl10-servers/proliant-dl20-server/hpe-proliant-dl20-gen10-plus-e-2336-2-9ghz-6-core-1p-16gb-u-4sff-500w-rps-server/p/p44115-b21?ef_id=Cj0KCQiAt66eBhCnARIsAKf3ZNFjsG49UV6Zm33R7lkRqi-XOd_JECmdyqNMAm2CKLSm_F-z6JTYDTQaAgMTEALw_wcB:G:s&s_kwid=AL113472131331628972784!!!g1318267171339!!1707918369!67076417419&gclid=Cj0KCQiAt66eBhCnARIsAKf3ZNFjsG49UV6Zm33R7lkRqi-XOd_JECmdyqNMAm2CKLSm_F-z6JTYDTQaAgMTEALw_wcB

[https://www.server-rack-online.com/gl910ent-4048sss.html?](https://www.server-rack-online.com/gl910ent-4048sss.html?utm_medium=shoppingengine&utm_source=googlebase&utm_source=google&utm_medium=cpc&adpos=&scid=scplpgl910ent-4048sss&sc_intid=gl910ent-4048sss&gclid=Cj0KCQiAt66eBhCnARIsAKf3ZNFjsG49UV6Zm33R7lkRqi-XOd_JECmdyqNMAm2CKLSm_F-z6JTYDTQaAgMTEALw_wcB)

[utm_medium=shoppingengine&utm_source=googlebase&utm_source=google&utm_medium=cpc&adpos=&scid=scplpgl910ent-4048sss&sc_intid=gl910ent-4048sss&gclid=Cj0KCQiAt66eBhCnARIsAKf3ZNFjsG49UV6Zm33R7lkRqi-XOd_JECmdyqNMAm2CKLSm_F-z6JTYDTQaAgMTEALw_wcB](https://www.server-rack-online.com/gl910ent-4048sss.html?utm_medium=shoppingengine&utm_source=googlebase&utm_source=google&utm_medium=cpc&adpos=&scid=scplpgl910ent-4048sss&sc_intid=gl910ent-4048sss&gclid=Cj0KCQiAt66eBhCnARIsAKf3ZNFjsG49UV6Zm33R7lkRqi-XOd_JECmdyqNMAm2CKLSm_F-z6JTYDTQaAgMTEALw_wcB)

Replication Policy

Cassandra replication strategies are "pluggable", with a couple built-in options.

SimpleStrategy

- all nodes are considered equal
- ignores rack and data center placement
- used in CS 544

NetworkTopologyStrategy

- considers data centers and racks
- when walking the ring, some vnodes may be skipped to protect against correlated failures

Worksheet

Outline: Cassandra Partitioning+Replication

Partitioning

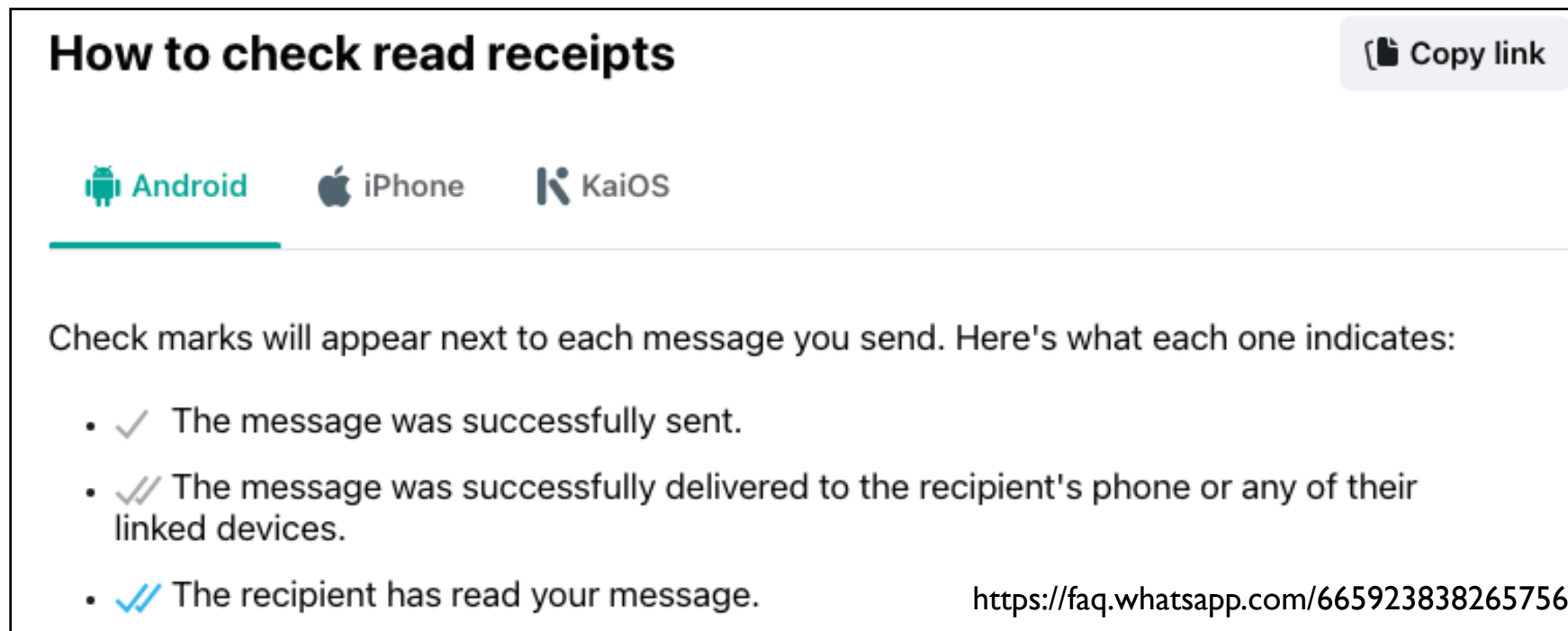
Replication

Quorum Reads/Writes

Conflict Resolution

Cassandra Demos

Write Acks: WhatsApp Example



these are examples of "acks" (acknowledgements)

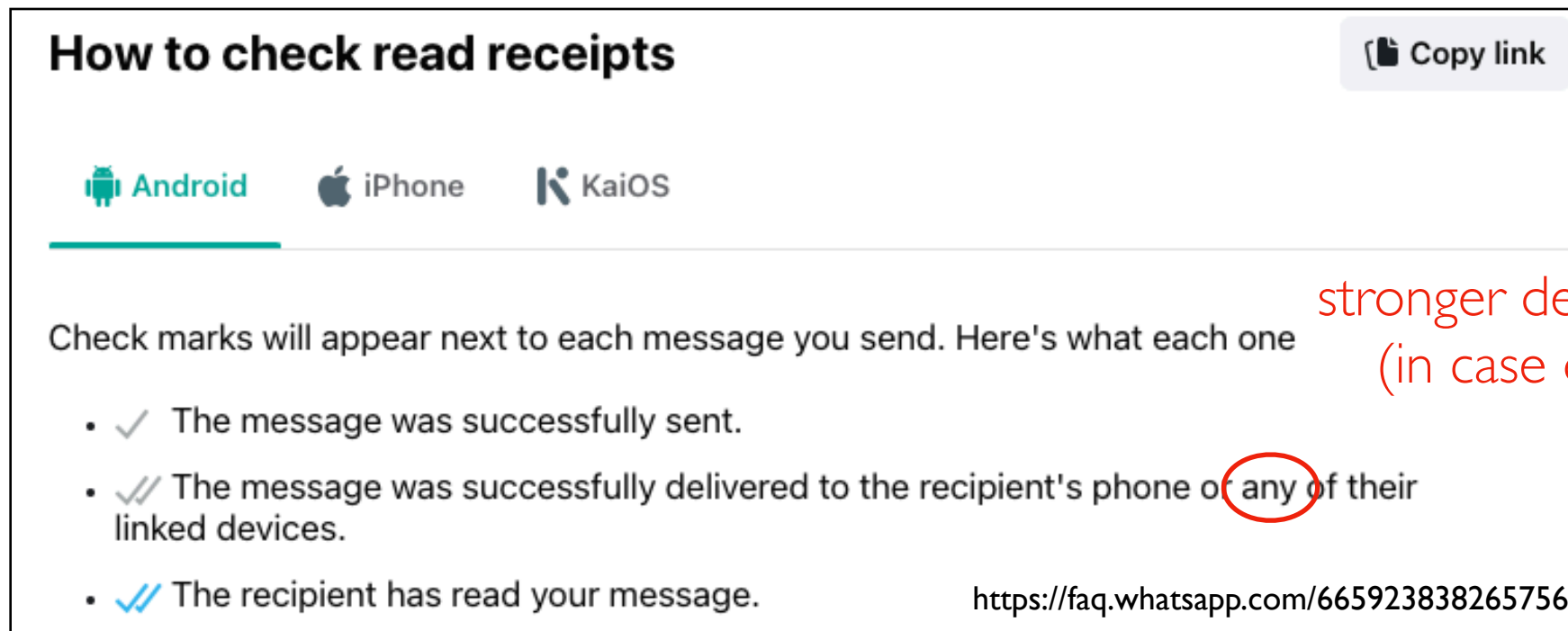
In distributed storage systems/databases, an ack means our data is committed.

"Committed" means our data is "safe", even if bad things happen. The definition varies system to system, based on what bad things are considered. For example:

- a node could hang until rebooted; a node's disk could permanently fail
- a rack could lose power; a data center could be destroyed

Obviously, no data is ever completely safe against any circumstance (e.g., comet strikes earth, leading to destruction of every data center).

Write Acks: WhatsApp Example



stronger definition: **all** devices
(in case one device fails)

these are examples of "acks" (acknowledgements)

In distributed storage systems/databases, an ack means our data is committed.

"Committed" means our data is "safe", even if bad things happen. The definition varies system to system, based on what bad things are considered. For example:


- a node could hang until rebooted; a node's disk could permanently fail
- a rack could lose power; a data center could be destroyed


Obviously, no data is ever completely safe against any circumstance (e.g., comet strikes earth, leading to destruction of every data center).


Write Acks: WhatsApp Example

How to check read receipts

[Copy link](#)

 Android

 iPhone

 KaiOS

Check marks will appear next to each message you send. Here's what each one indicates:

- ✓ The message was successfully sent.
- ✓✓ The message was successfully delivered to the recipient's phone or any of their linked devices.
- ✓✓ The recipient has read your message.

<https://faq.whatsapp.com/665923838265756>

these are examples of "acks" (acknowledgements)

two checks (in WhatsApp) mean the message reached the destination.

Does only one check mean the message has NOT reached the destination?

Write Acks: WhatsApp Example

How to check read receipts

Copy link

Android

iPhone

KaiOS

Check marks will appear next to each message you send. Here's what each one indicates:

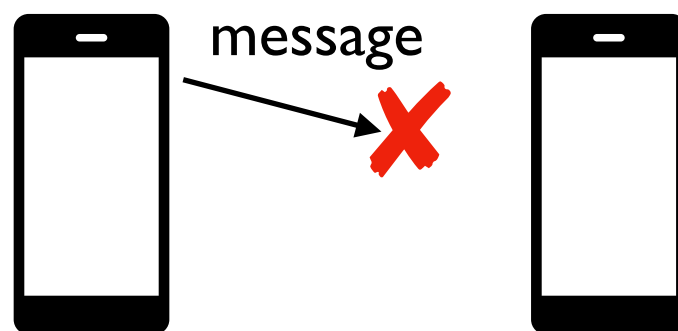
- ✓ The message was successfully sent.
- ✓✓ The message was successfully delivered to the recipient's phone or any of their linked devices.
- ✓✓ The recipient has read your message.

<https://faq.whatsapp.com/665923838265756>

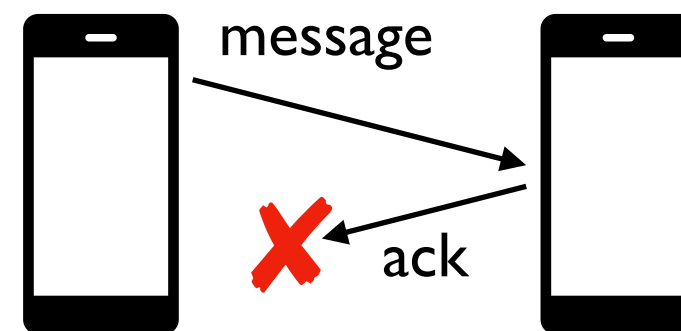
these are examples of "acks" (acknowledgements)

two checks (in WhatsApp) mean the message reached the destination.

Does only one check mean the message has NOT reached the destination?

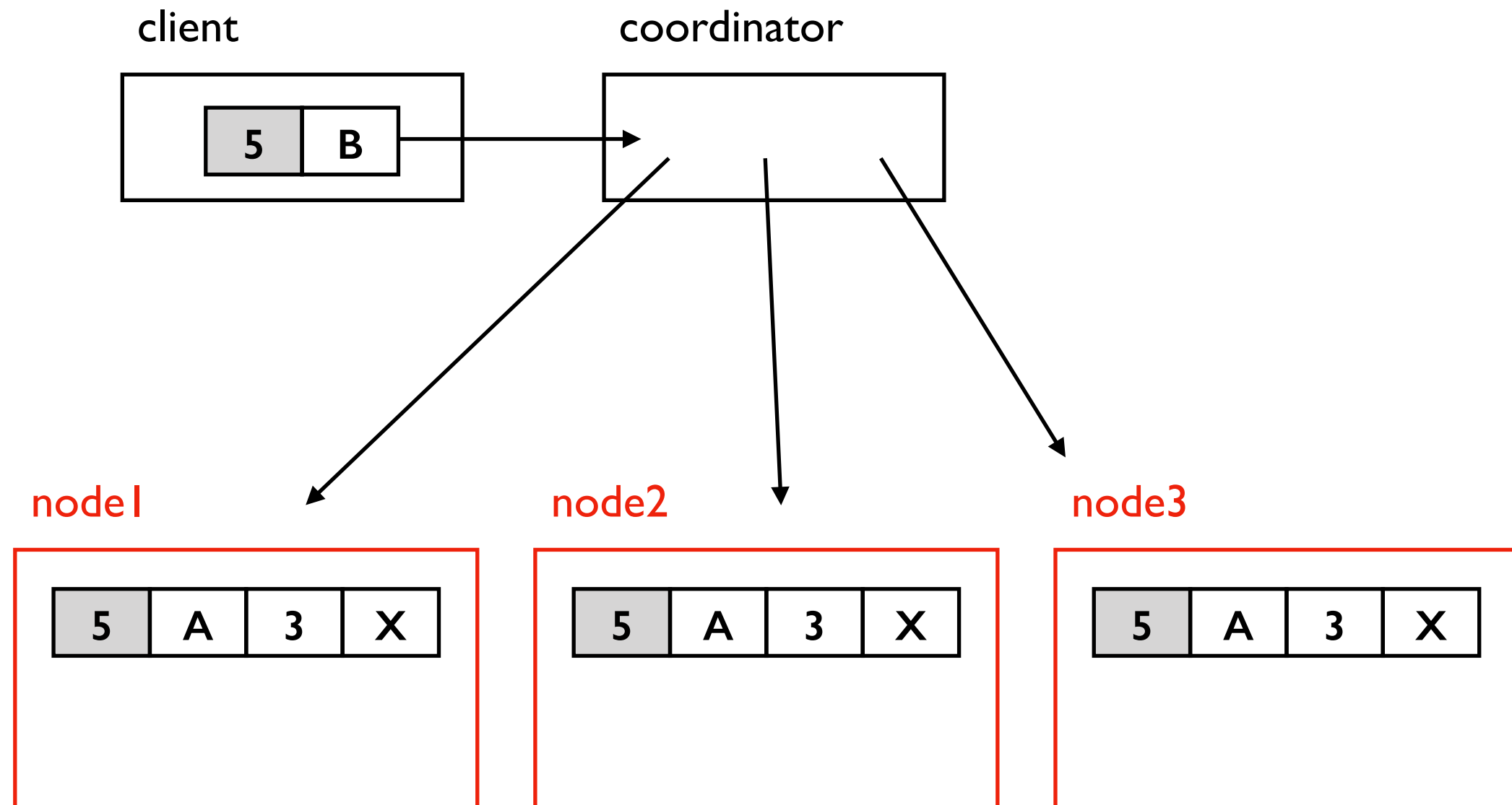


scenario 1



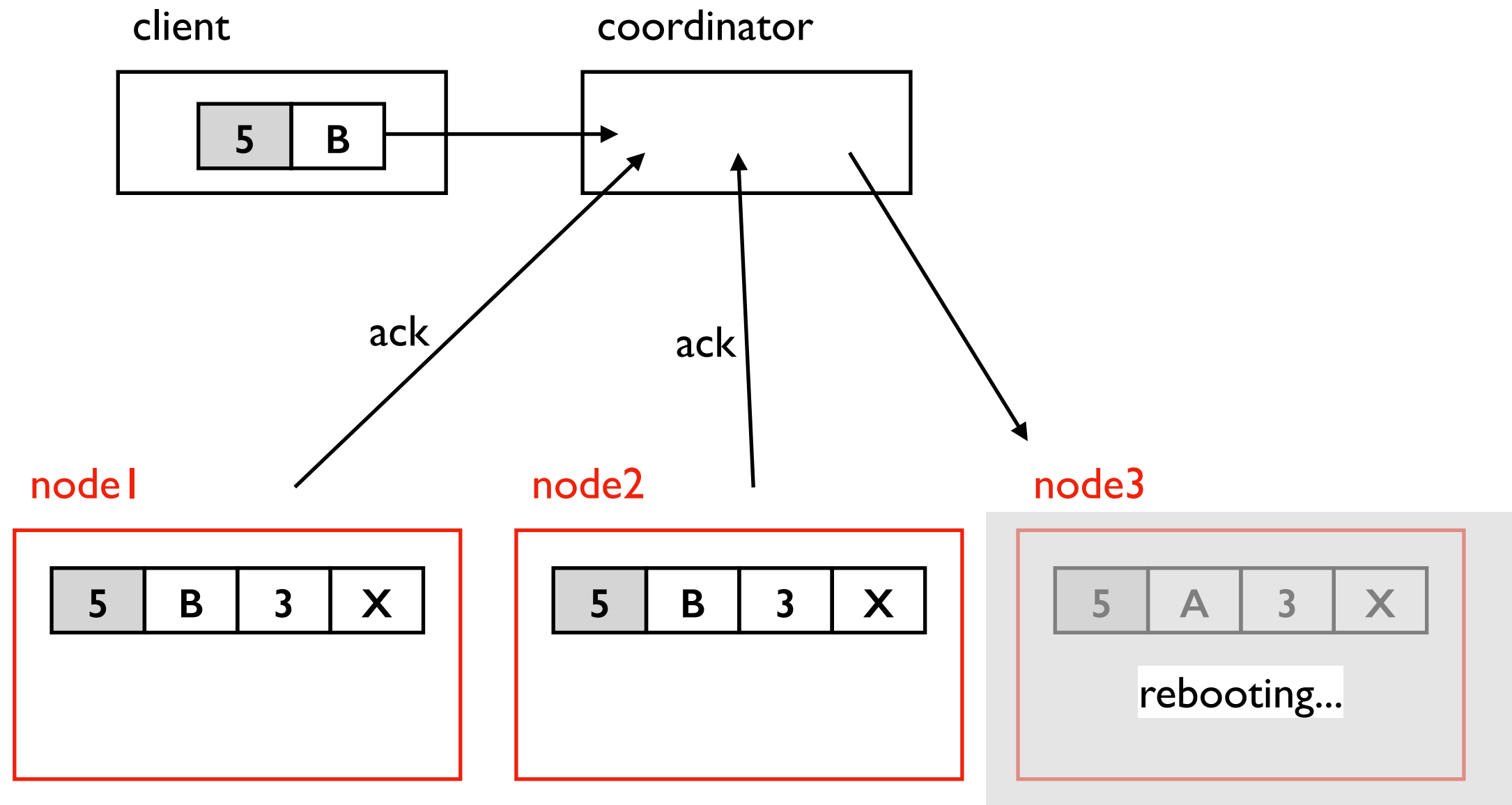
scenario 2

Cassandra Writes



Say RF=3. Coordinator will attempt to write data to all 3 replicas.

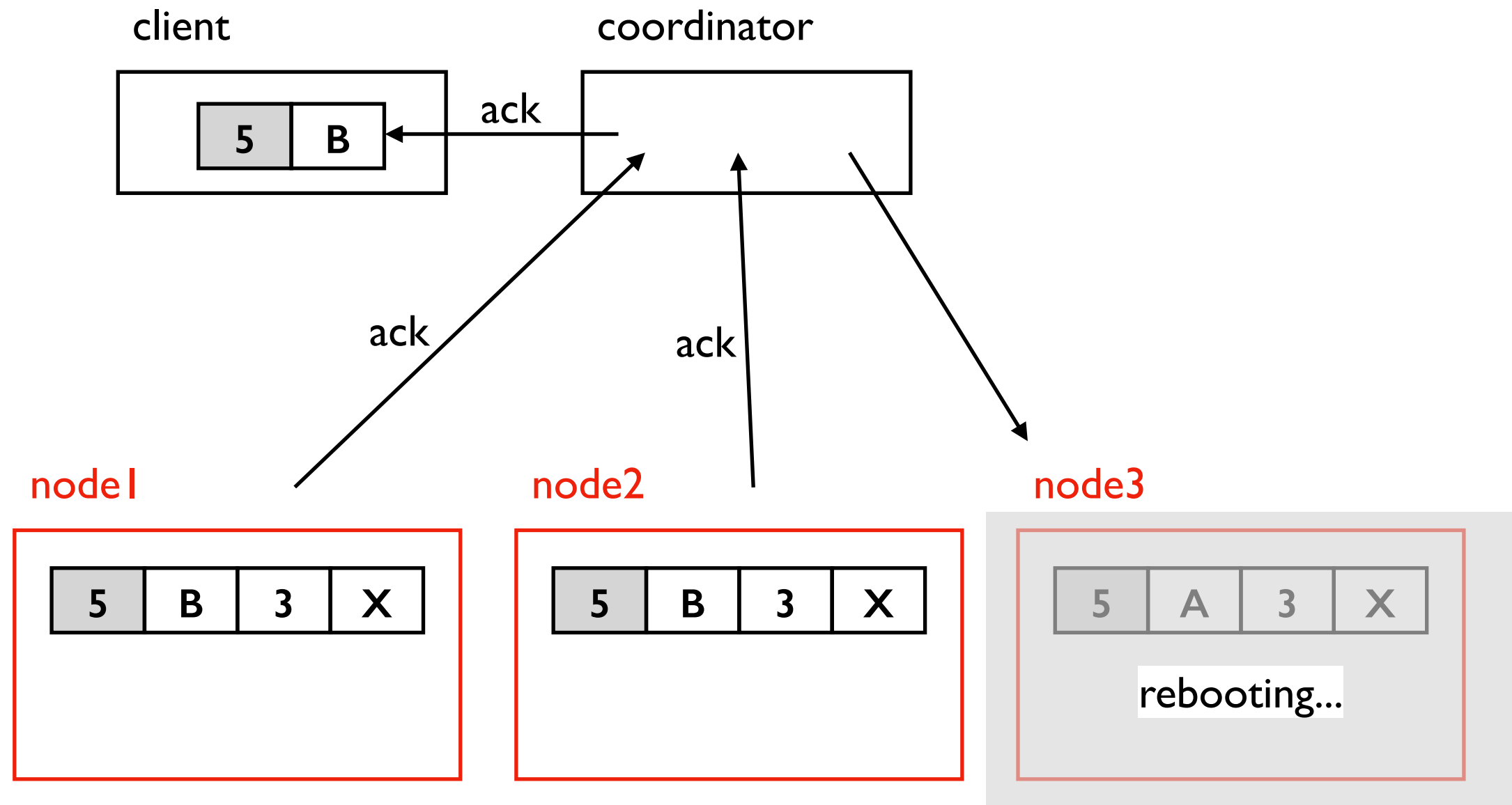
Cassandra Writes



Say RF=3. Coordinator will attempt to write data to all 3 replicas.

At what point should we send an ack to the client?

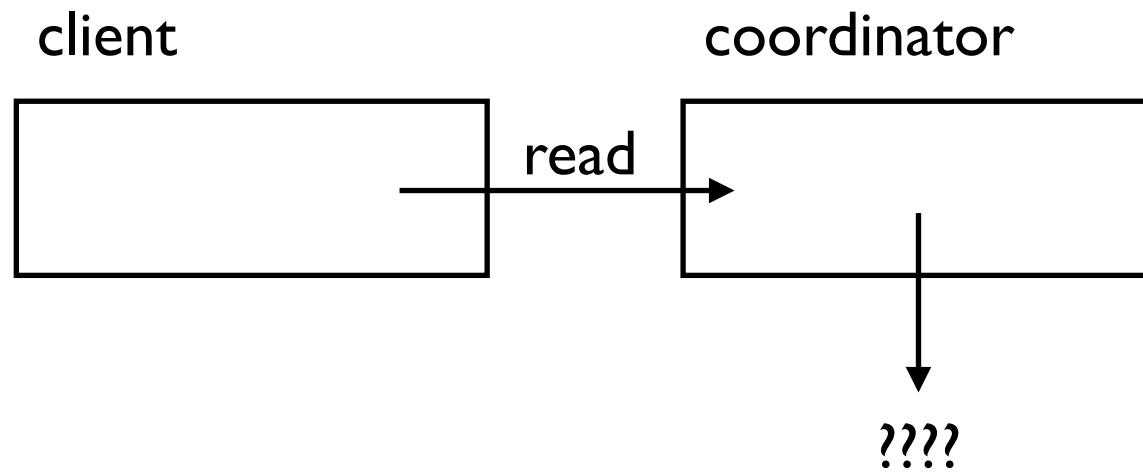
Cassandra Writes



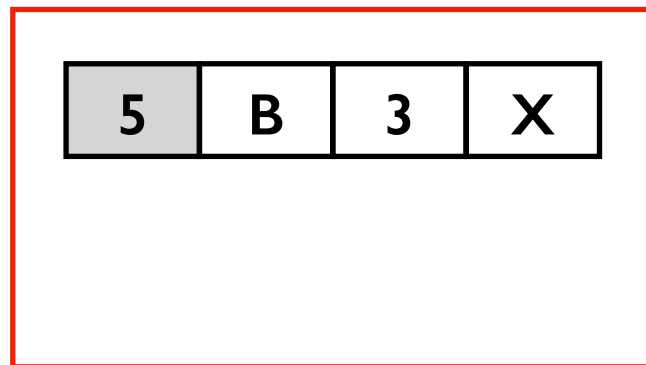
Say $RF=3$. Coordinator will attempt to write data to all 3 replicas.

At what point should we send an ack to the client?
Configurable. $W=2$ lets coordinator ack now, and data is fairly safe.

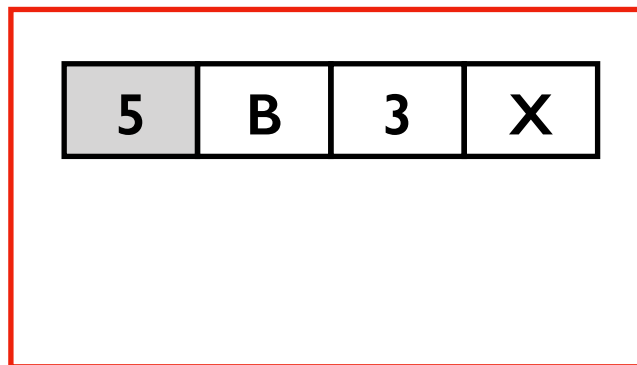
Cassandra Reads



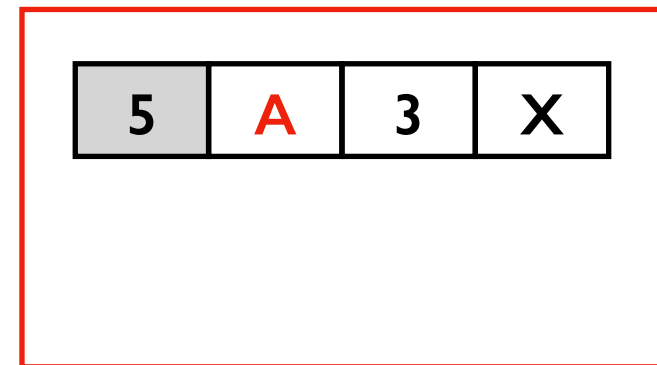
node1



node2

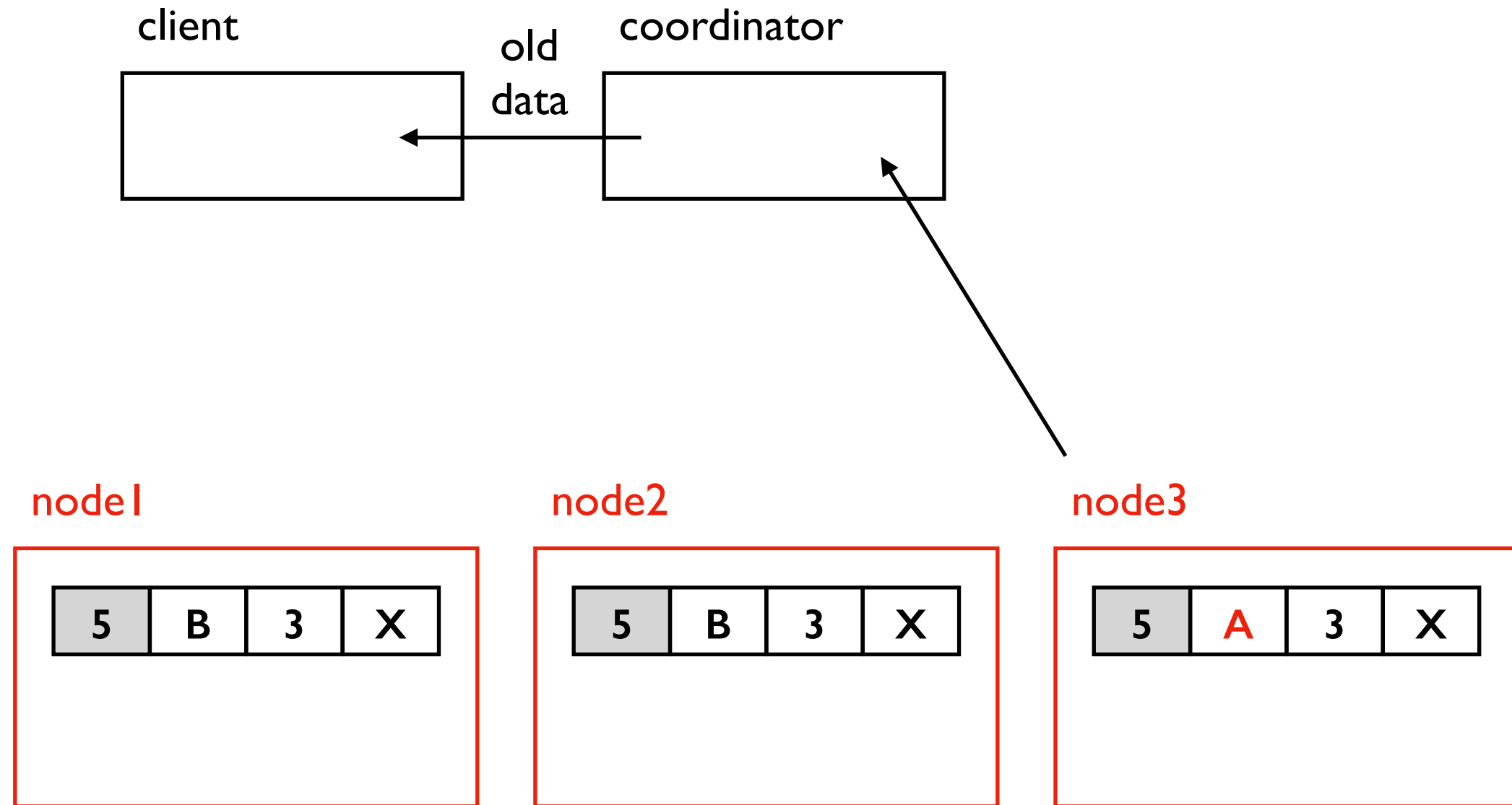


node3



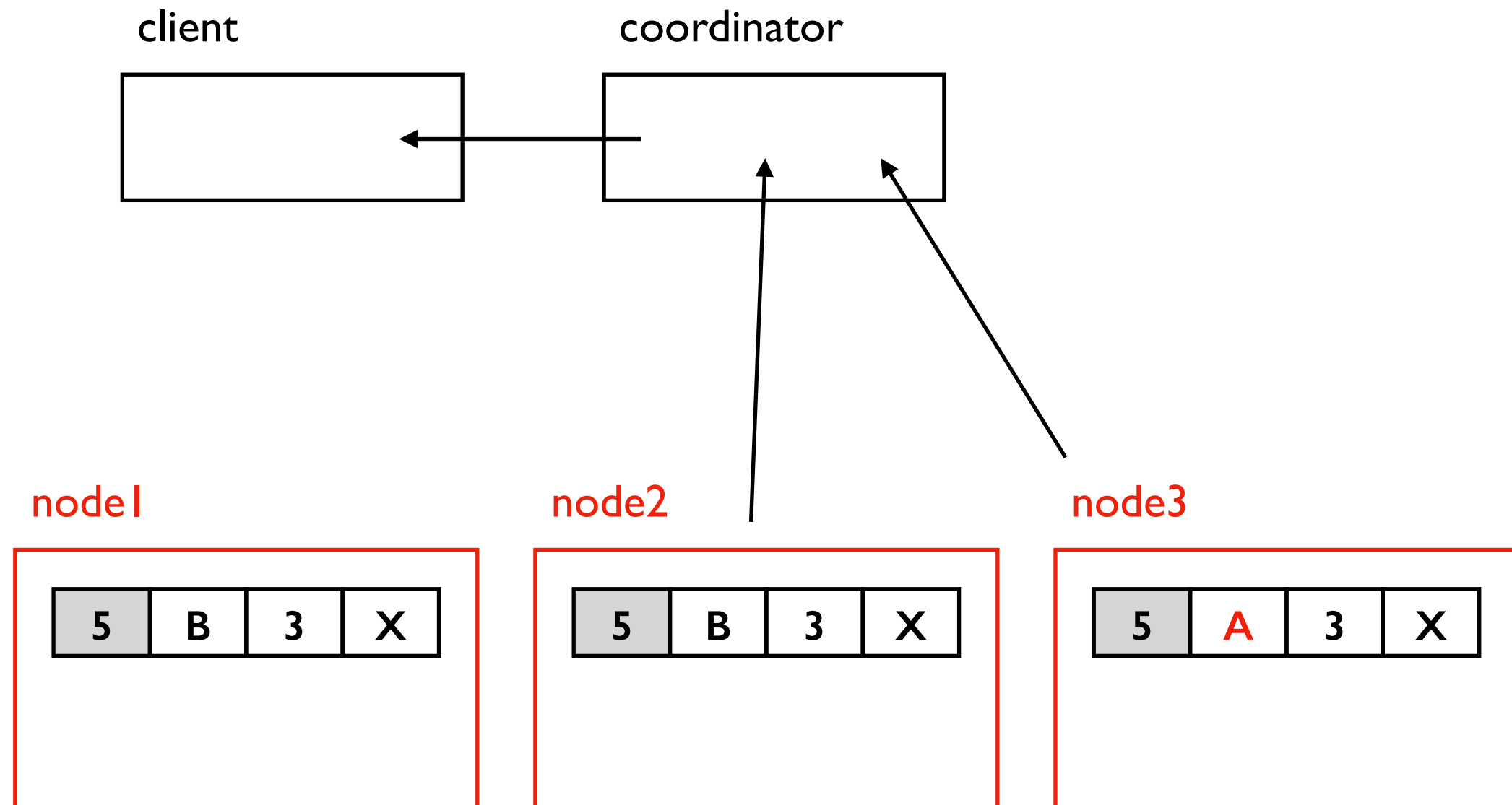
HDFS reads go to one replica. What if Cassandra tries that?

Cassandra Reads



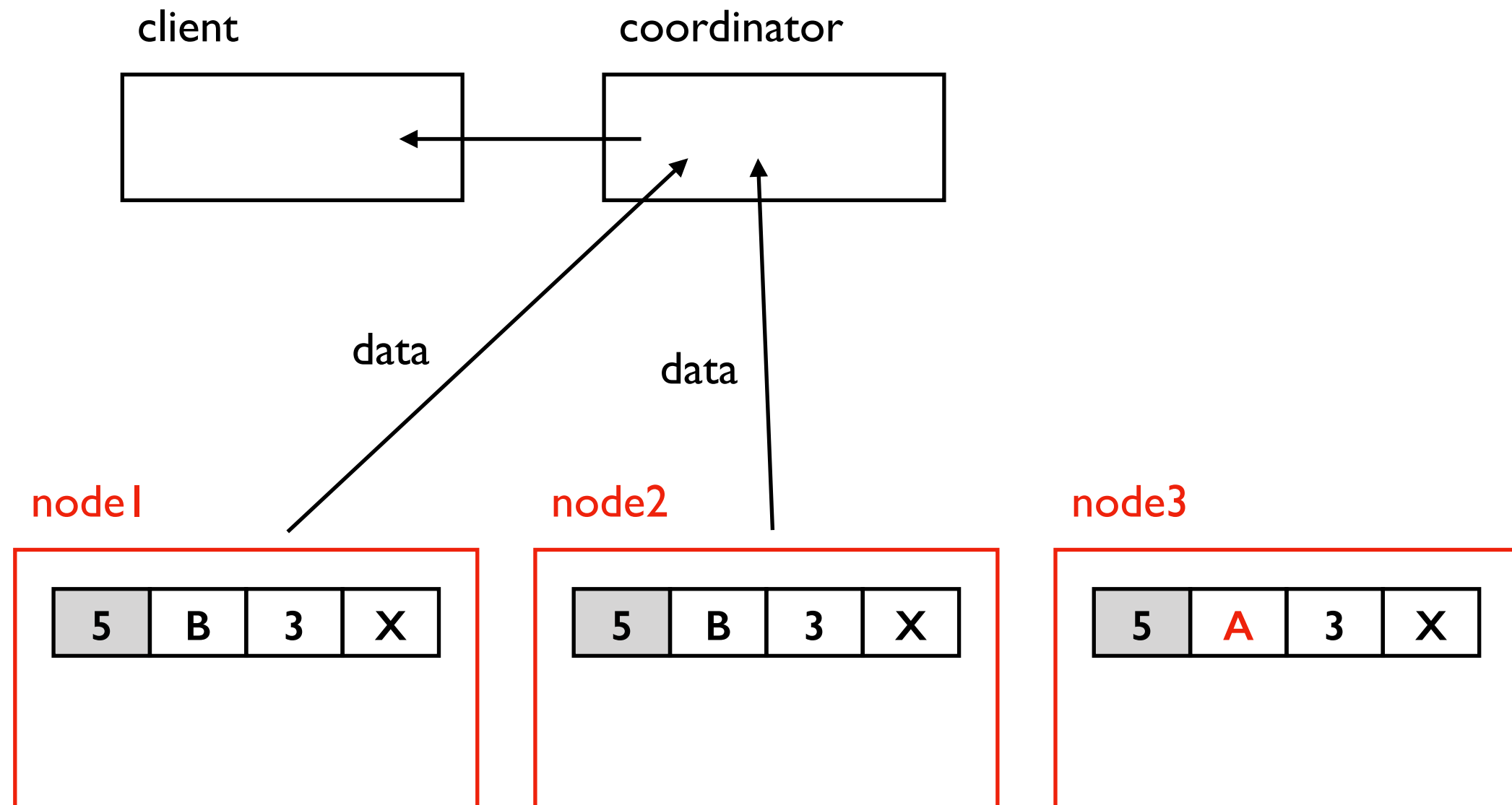
HDFS reads go to one replica. What if Cassandra tries that?

Cassandra Reads



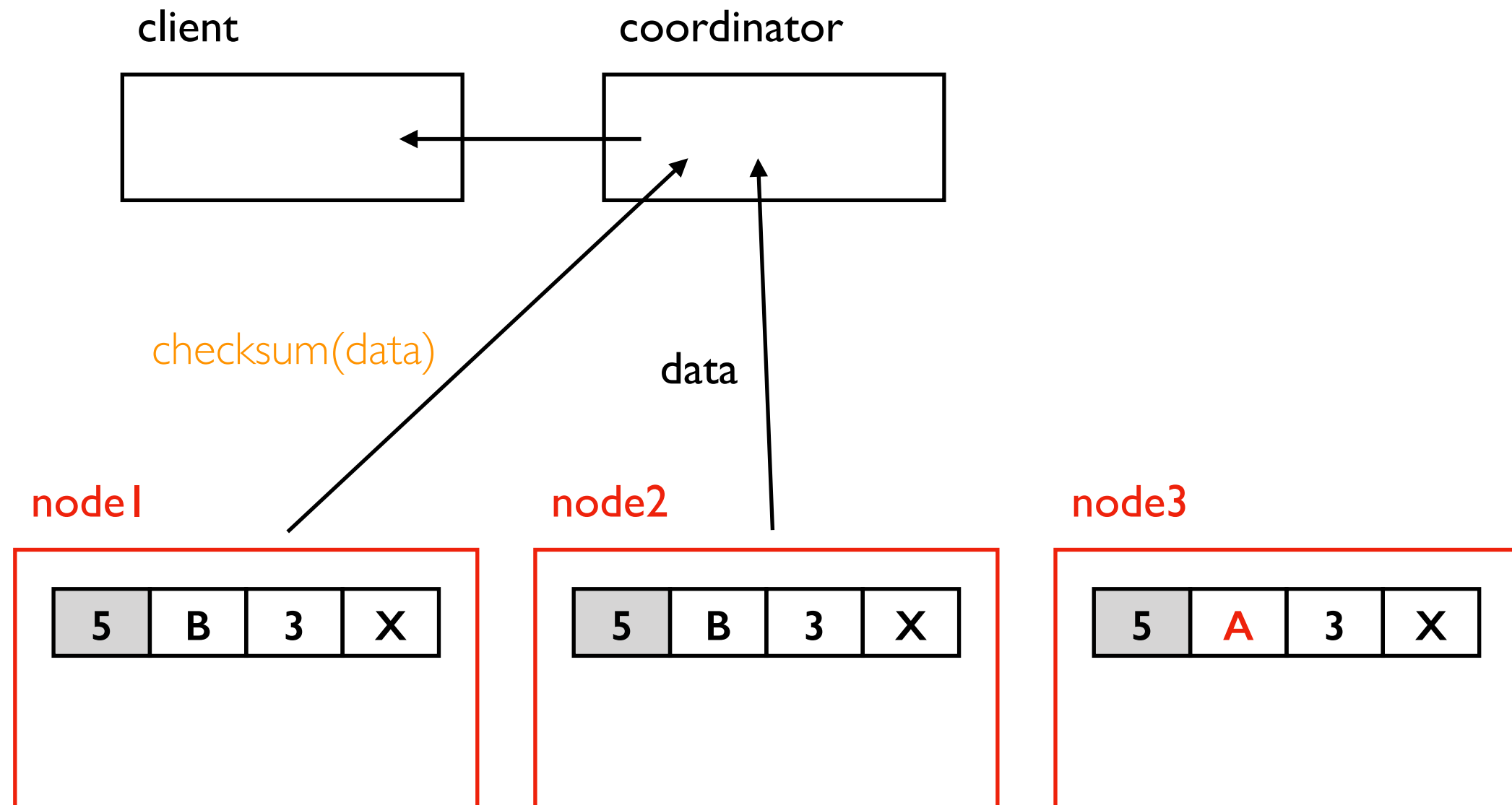
Read from R replicas (configurable). Here R=2.
Hopefully at least one of the replicas has new data.

Cassandra Reads



R=2 means we'll often read identical data from two replicas (wasteful!)

Cassandra Reads



R=2 means we'll often read identical data from two replicas (wasteful!)

Improvement: read one copy, and only request checksum from others.

A *checksum* (like md5) is a hash function where collisions are extremely rare and hard to find.

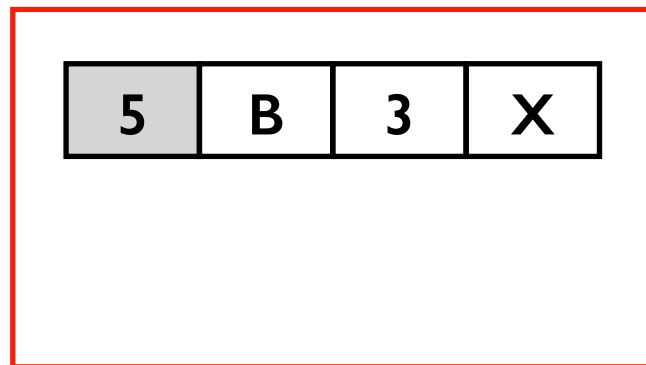
When $R+W > RF$

RF=3

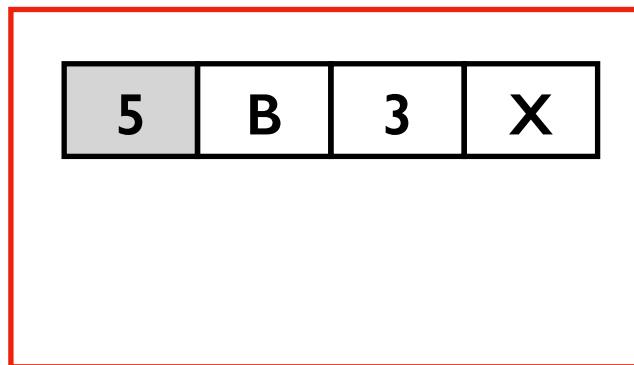
W=2

R=2

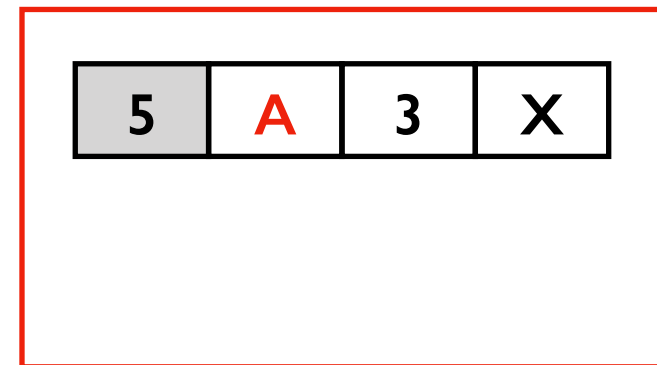
node1



node2



node3



When $R+W > RF$, the replicas read+written will overlap.

There are some caveats (related to ring membership and something called "hinted handoff") not covered here.

Tuning R and W

Say RF=3

W=3, R=1

- **reads are highly available** and fast -- only need one replica to respond before we can get back to the client!
- writes will not succeed (from the clients perspective) if even one node is down. But the data may still get recorded on some nodes.

W=1, R=3

- **writes are highly available** and fast -- only need one replica to respond before we can get back to the client!
- reads will not return data when even one node is down.
- risky: if the one node that took the write fails permanently, we'll lose committed data

W=2, R=2

- relatively balanced approach

W=1, R=1

- speed+availability more important than correct data

Worksheet

Outline: Cassandra Partitioning+Replication

Partitioning

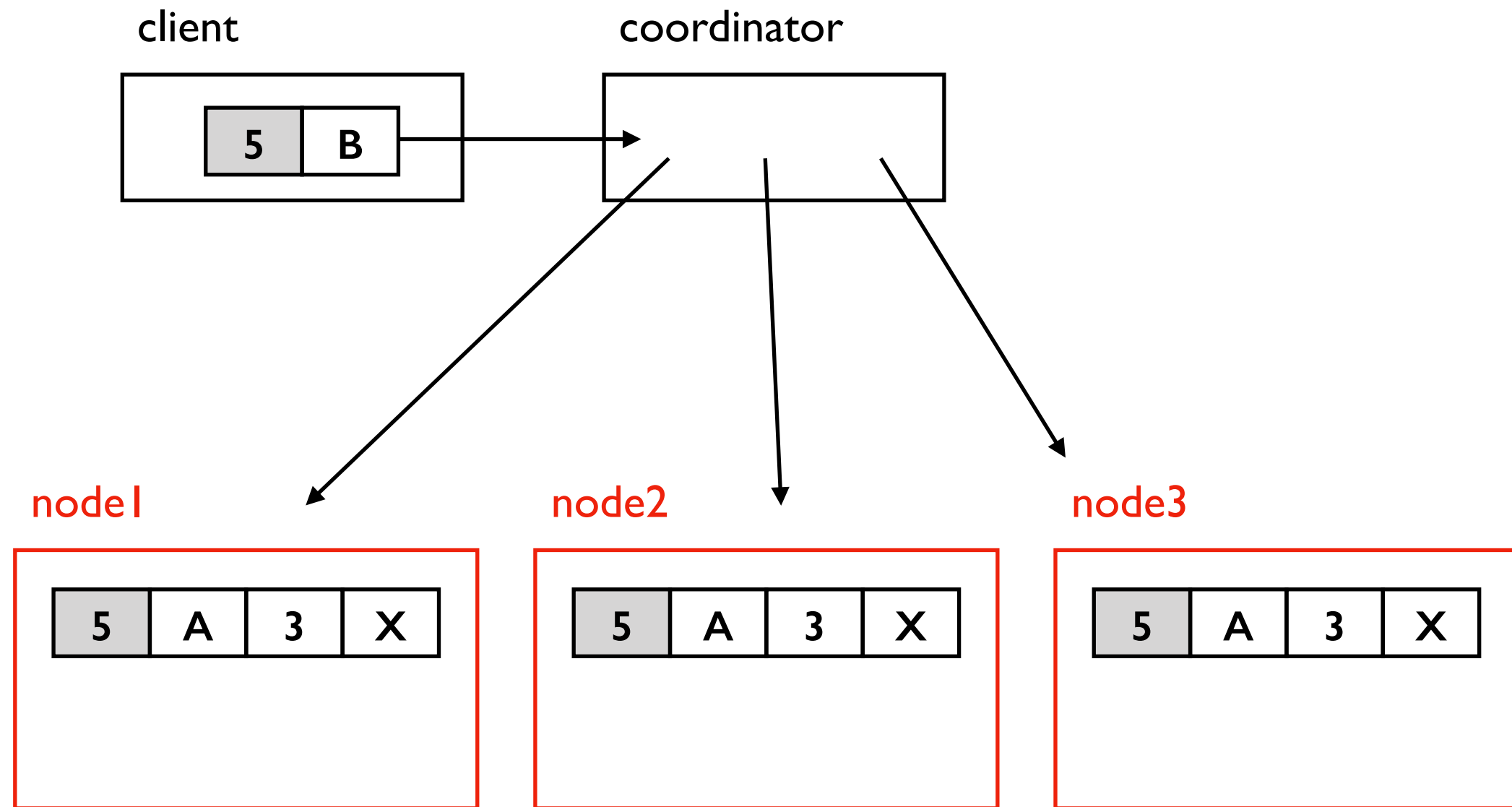
Replication

Quorum Reads/Writes

Conflict Resolution

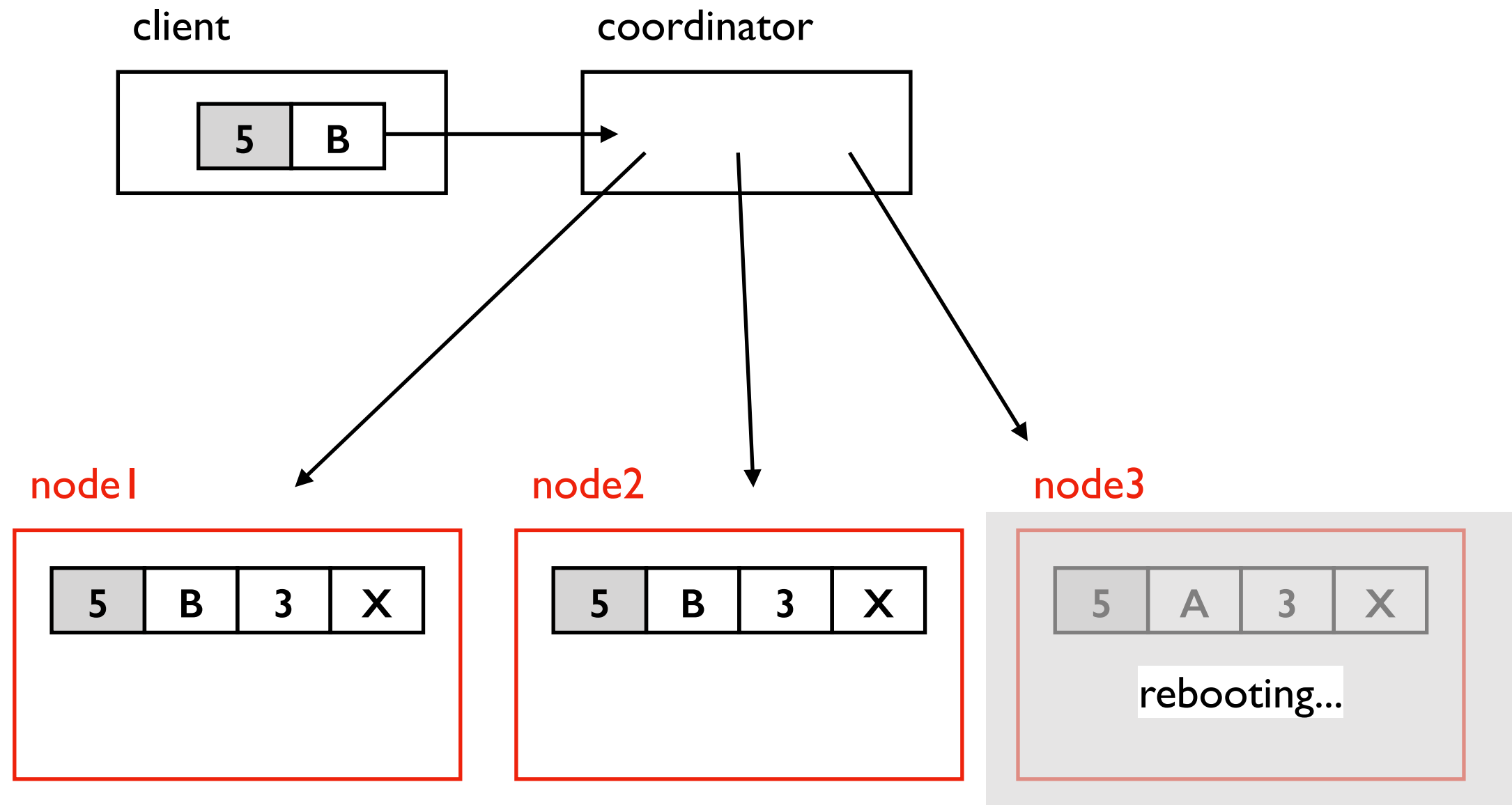
Cassandra Demos

Getting Conflicting Versions



Let $RF=3, R=2, W=2$

Getting Conflicting Versions



Let $RF=3, R=2, W=2$

Getting Conflicting Versions

client

--

coordinator

--

node1

5	B	3	X
---	---	---	---

node2

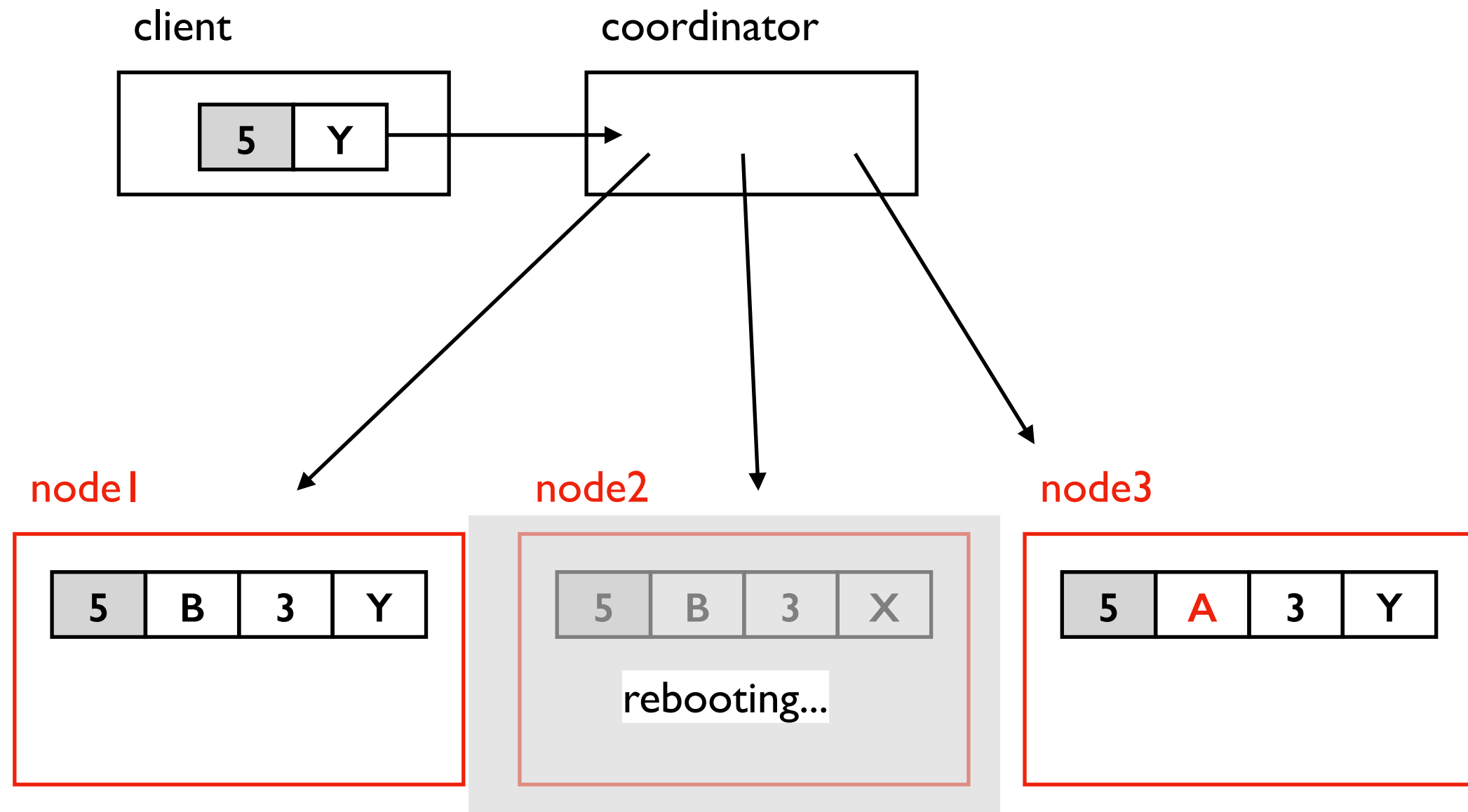
5	B	3	X
---	---	---	---

node3

5	A	3	X
---	---	---	---

Let $RF=3, R=2, W=2$

Getting Conflicting Versions



Let $RF=3, R=2, W=2$

Getting Conflicting Versions

client

--

coordinator

--

node1

5	B	3	Y
---	---	---	---

node2

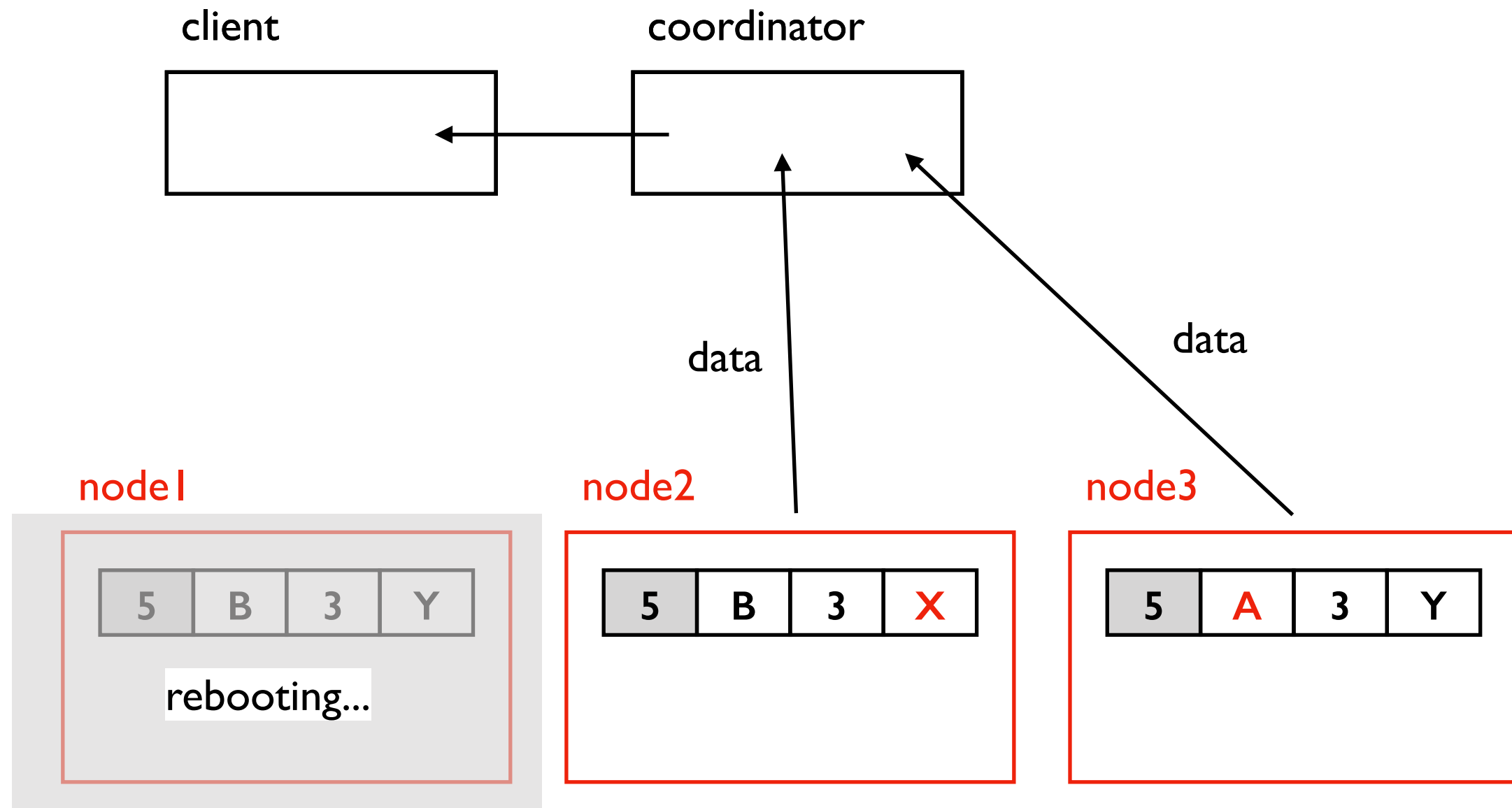
5	B	3	X
---	---	---	---

node3

5	A	3	Y
---	---	---	---

Let $RF=3, R=2, W=2$

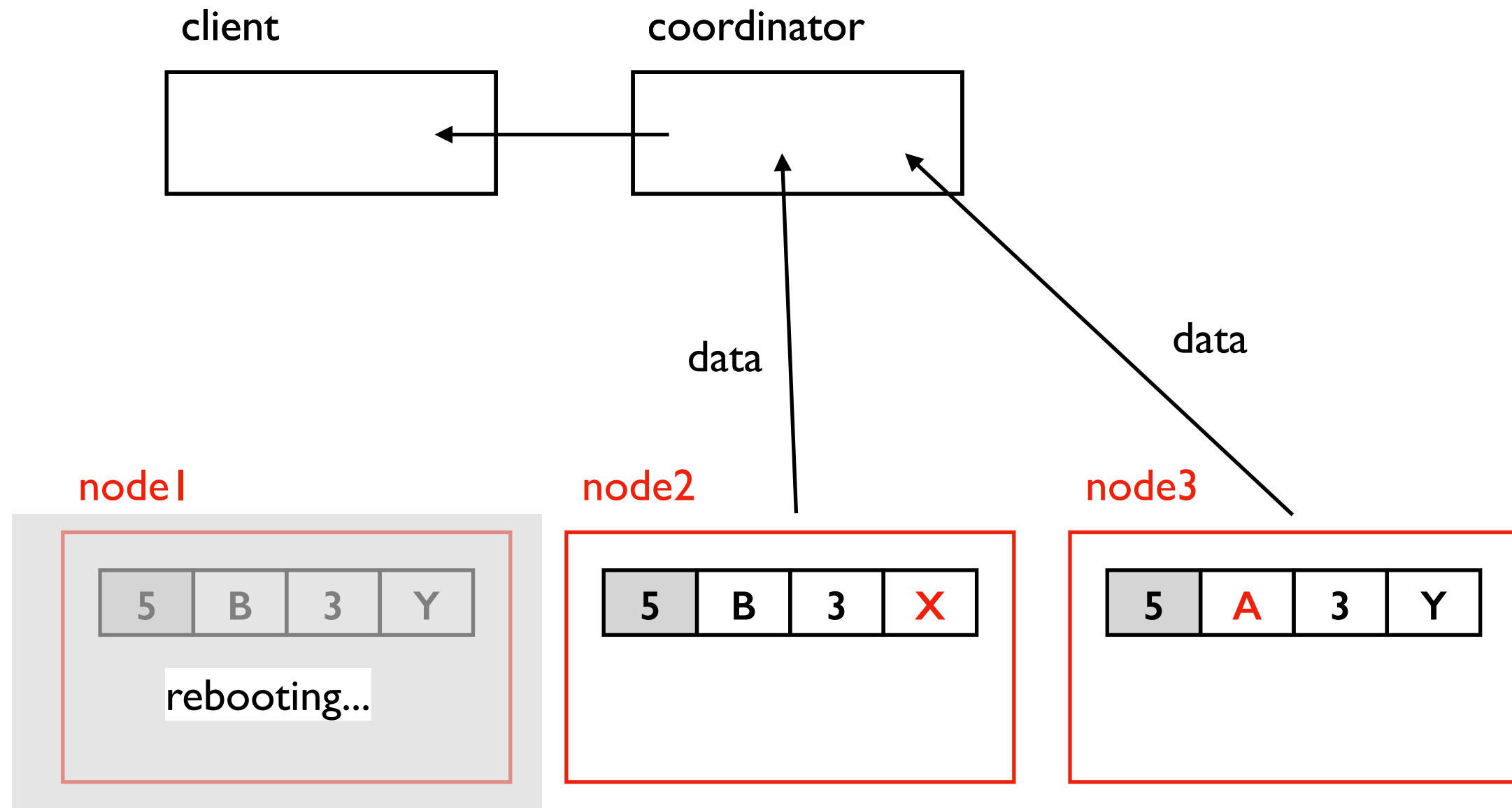
Getting Conflicting Versions



Which version of row 5 should be sent back?
Both contain some new data not contained by other.

Systems that allow conflicting versions to co-exist,
fixing it up later are "*eventually consistent*"

Getting Conflicting Versions

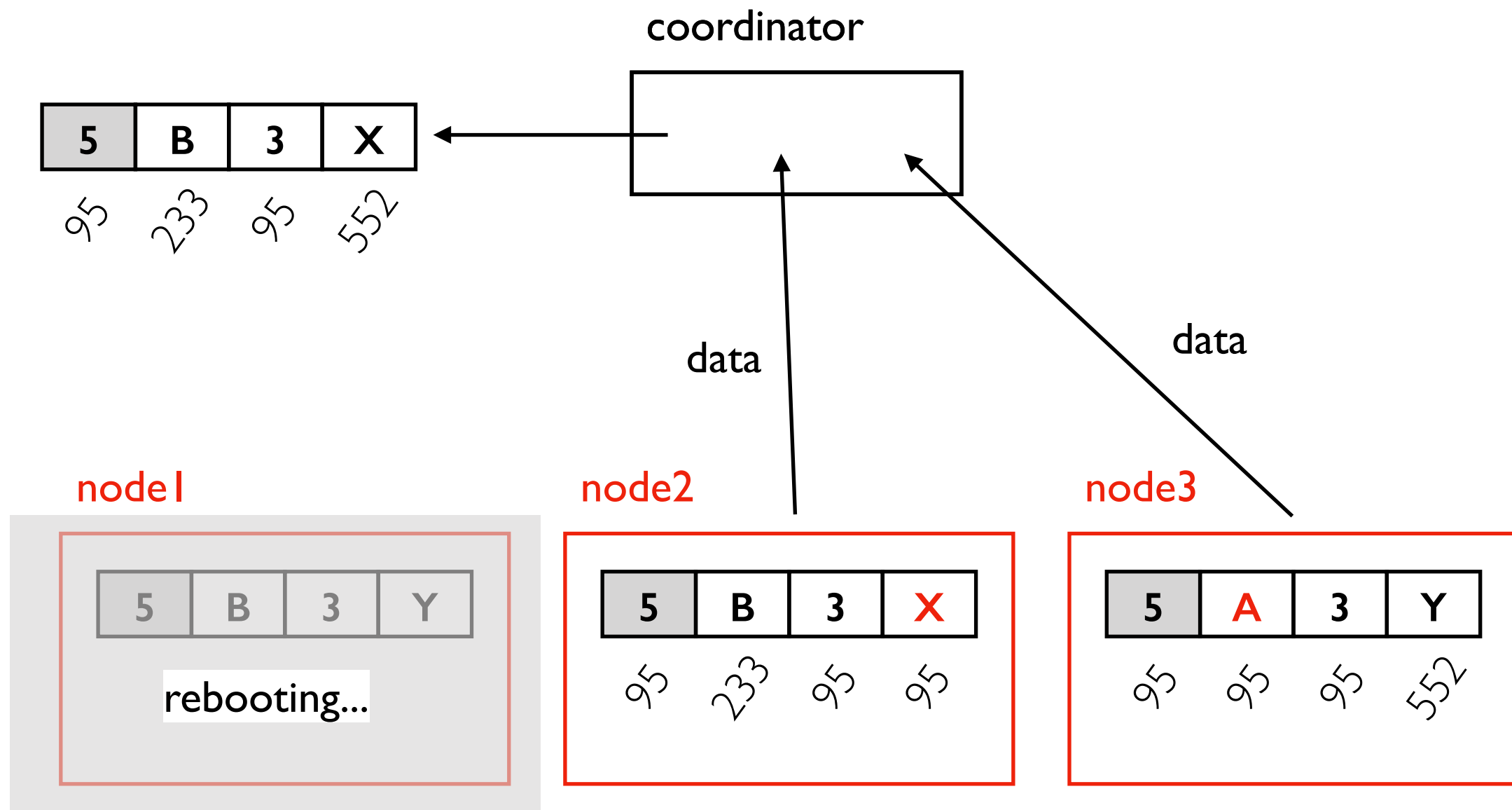


Approaches:

- send all version back to the client, which will need specialized conflict resolution code
- automatically combine them into a new row, and write that (if possible to all replicas)

Dynamo supports both. Cassandra uses second approach.

Timestamps



Every cell of every table has a timestamp:

- approximate (since clocks of nodes in a cluster are never perfectly in sync)
- policy is LWW (last writer wins), meaning prefer newer data
- Cassandra lets you query the timestamp of each cell

Outline: Cassandra Partitioning+Replication

Partitioning

Replication

Quorum Reads/Writes

Conflict Resolution

Cassandra Demos