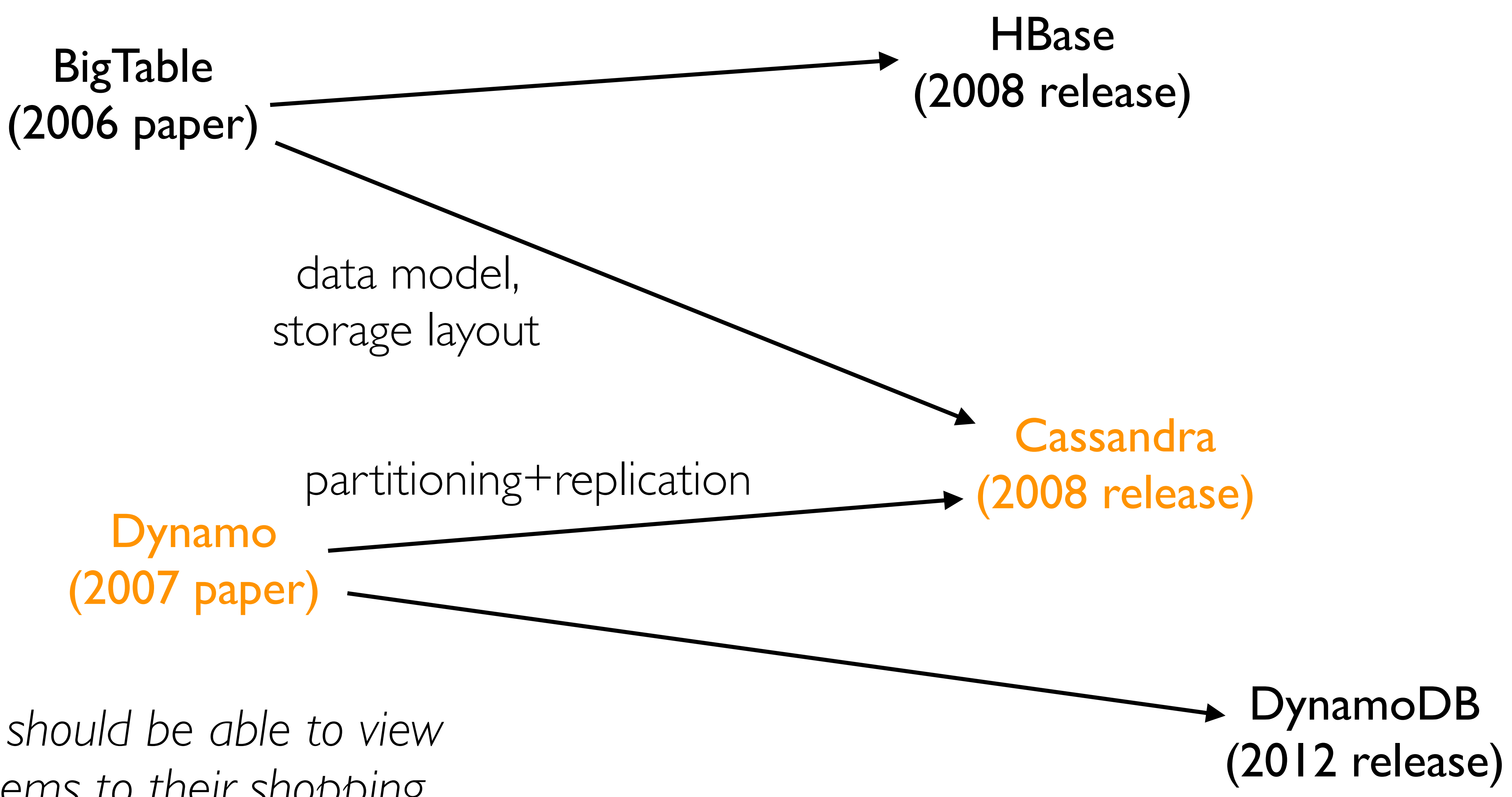
[544] Cassandra Partitioning

Tyler Caraza-Harter

Learning Objectives

- identify strengths and weaknesses of different partitioning techniques
- interpret a token ring to assign a row of data to a Cassandra worker (assume single replication for now)
- describe how gossip can be used to replicate data across workers in a cluster, without need for a centralized boss

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

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

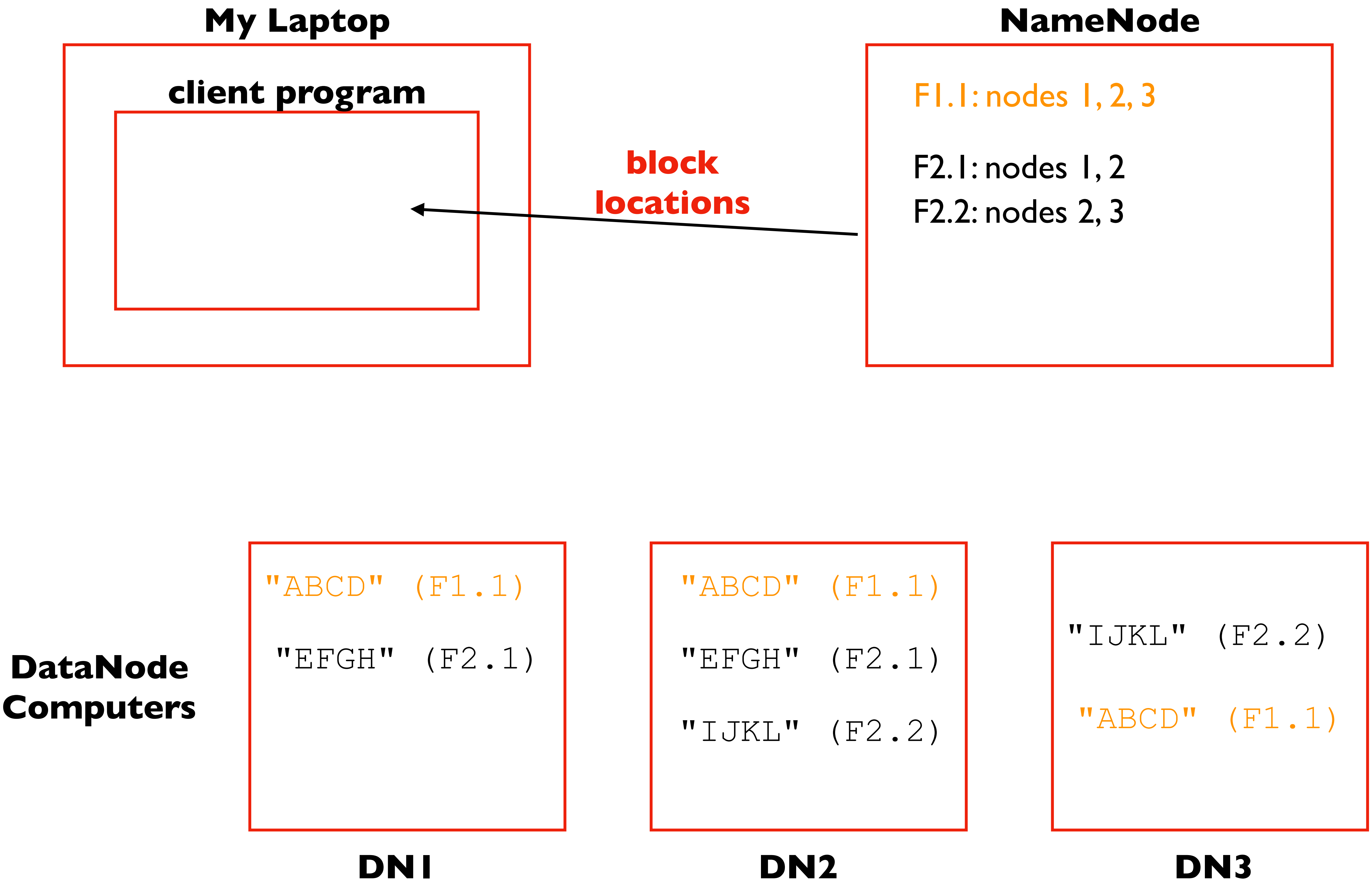
Hash Partitioning

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

Consistent Hashing

- Dynamo and Cassandra use this

Review: HDFS Partitioning



My Laptop

client program

block locations

NameNode

F1.1: nodes 1, 2, 3

F2.1: nodes 1, 2

F2.2: nodes 2, 3

DataNode Computers

DN1

"ABCD" (F1.1)

"EFGH" (F2.1)

DN2

"ABCD" (F1.1)

"EFGH" (F2.1)

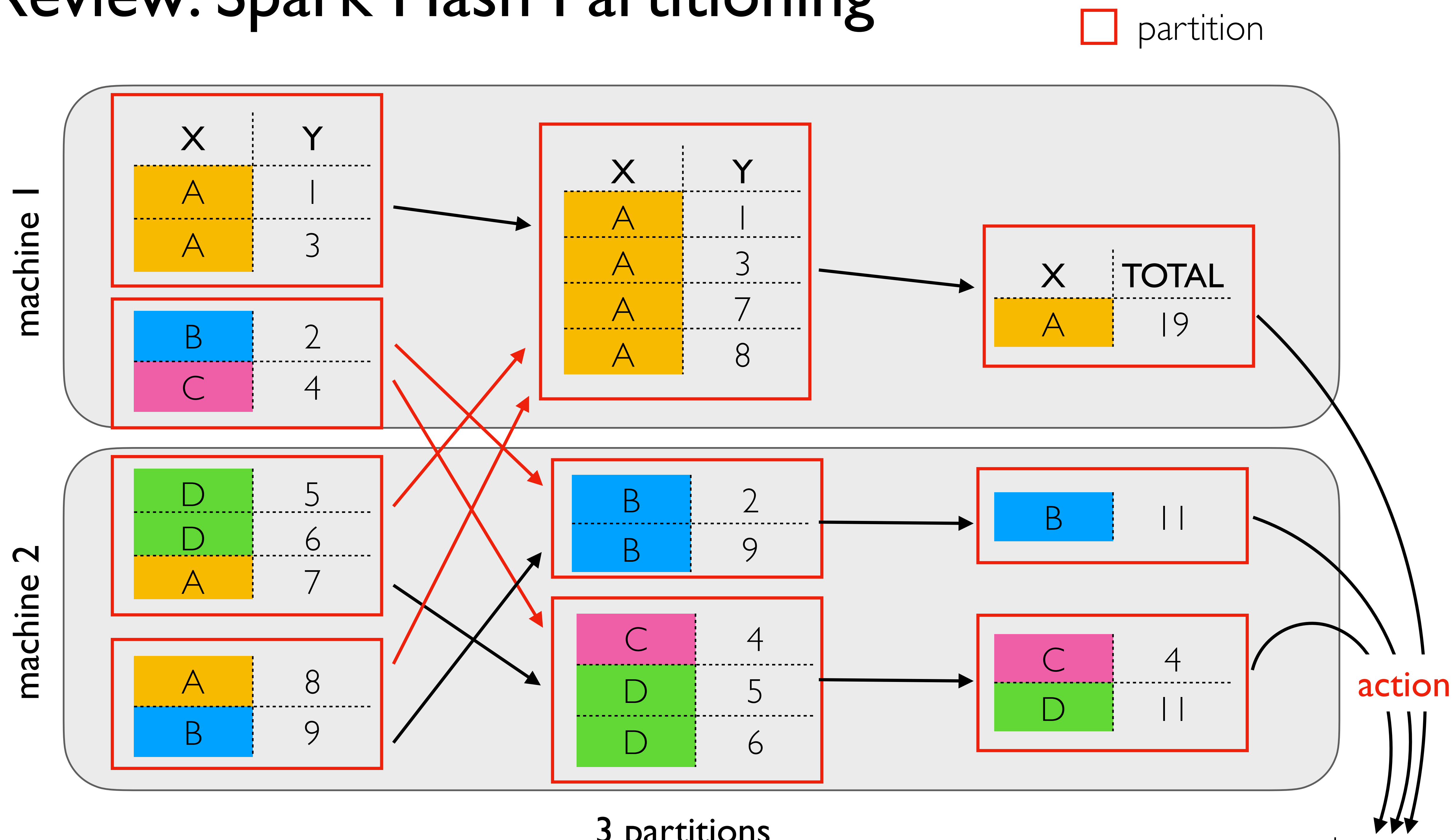
"IJKL" (F2.2)

DN3

"IJKL" (F2.2)

"ABCD" (F1.1)

Review: Spark Hash Partitioning



X	Y
A	1
A	3

B	2
C	4

D	5
D	6
A	7

A	8
B	9

X	Y
A	1
A	3
A	7
A	8

B	2
B	9

C	4
D	5
D	6

X	TOTAL
A	19

B	11

C	4
D	11

3 partitions

X	TOTAL
A	19
B	11
C	4
D	11

file or
Pandas DF

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

Discuss Scalability: HDFS and Spark

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

Some ways we can have big data:

- few large objects (files/tables)
- lots of small objects (files/tables)

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

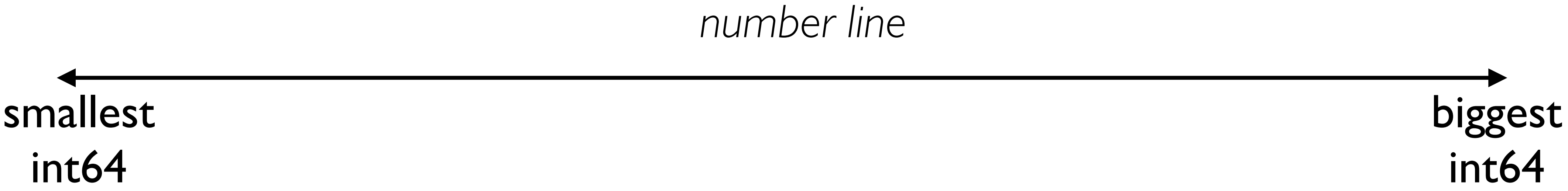
Hash Partitioning

- partition = hash(key) % 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 = hash(key)   *# every token is in a range, indicating the worker*
- locations = {range(0,10): "worker1", range(10,20): "worker2", ...}

Consistent Hashing



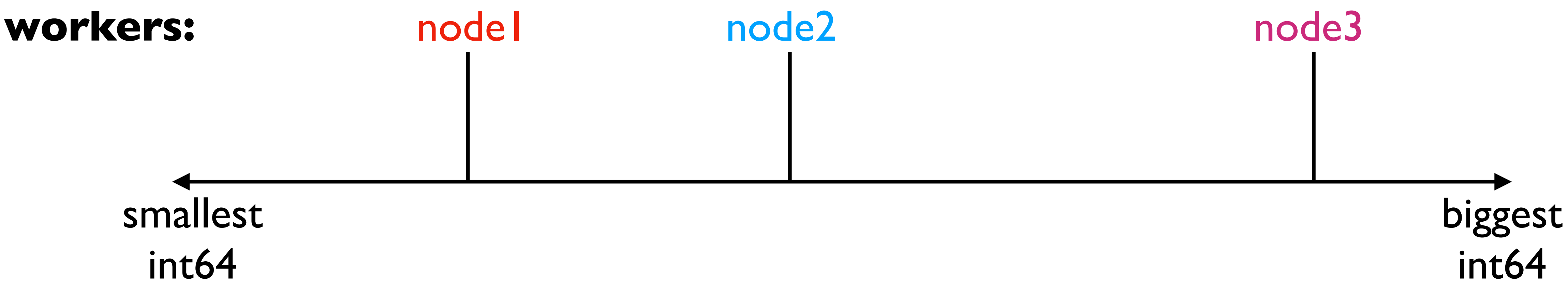
number line

smallest
int64

biggest
int64

Consistent Hashing

Token Map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something

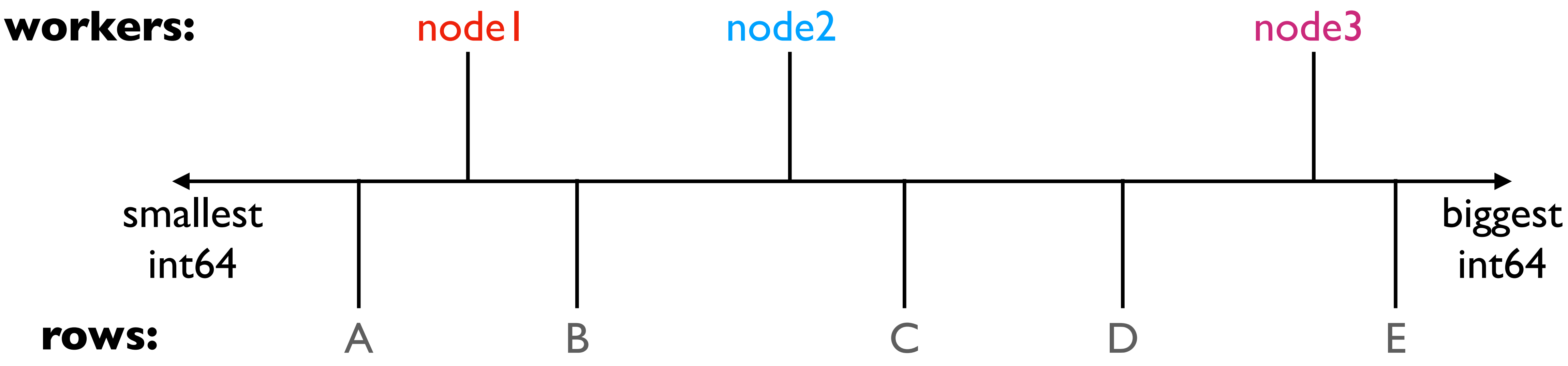


assign every **worker** a point on the number line.
Could be random (though newer approaches are more clever).
No hashing needed, *yet*!

Consistent Hashing

Token Map:

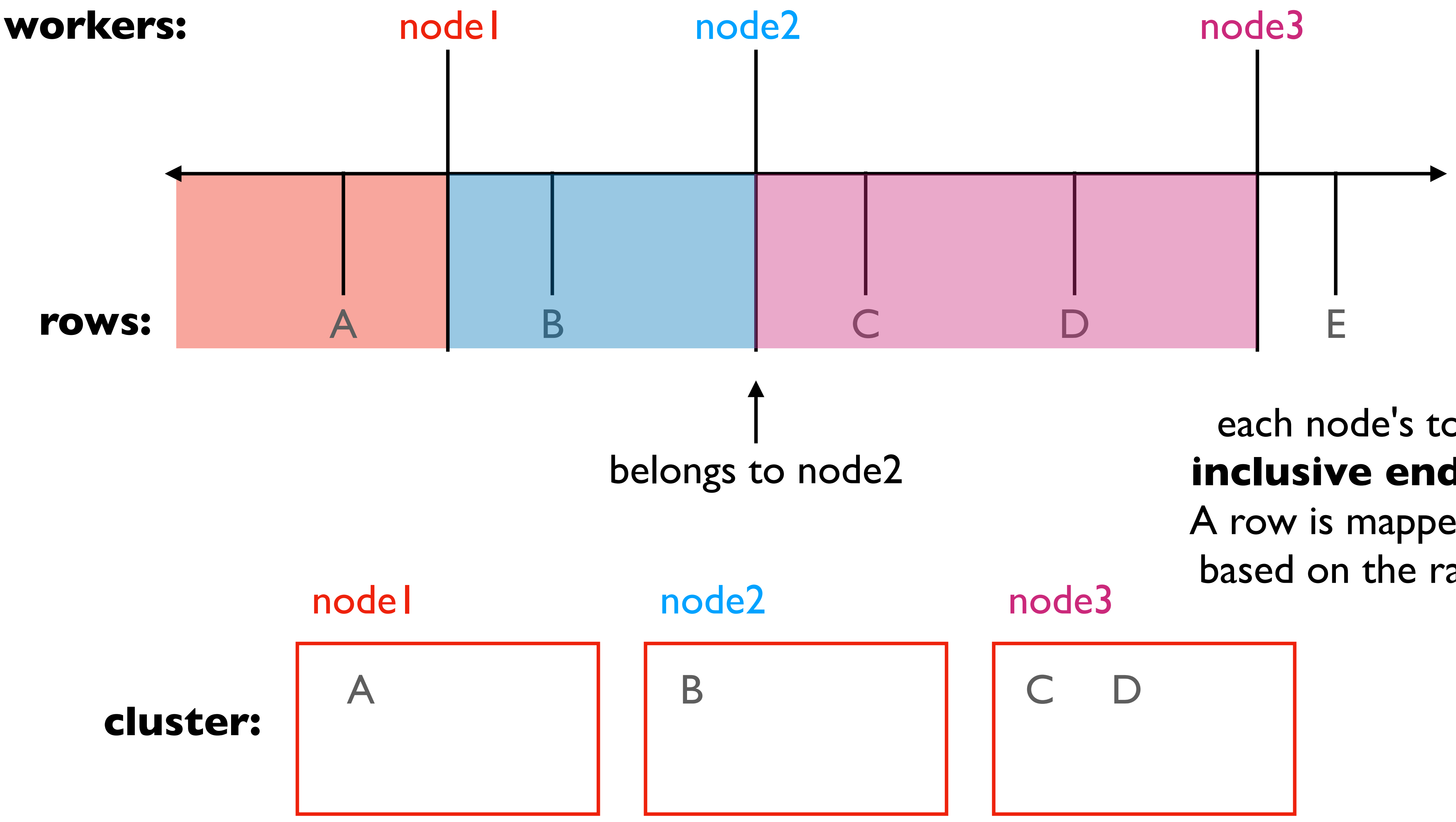
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something



assign every **row** a point on the number line.
token(row) = hash(row's **partition** key)

Consistent Hashing

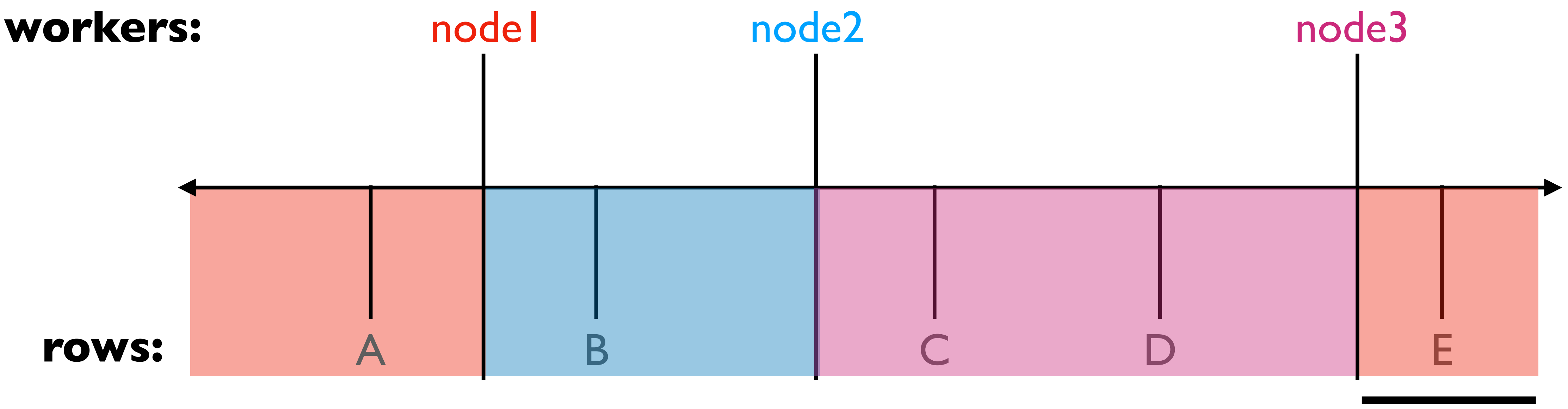
Token Map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something



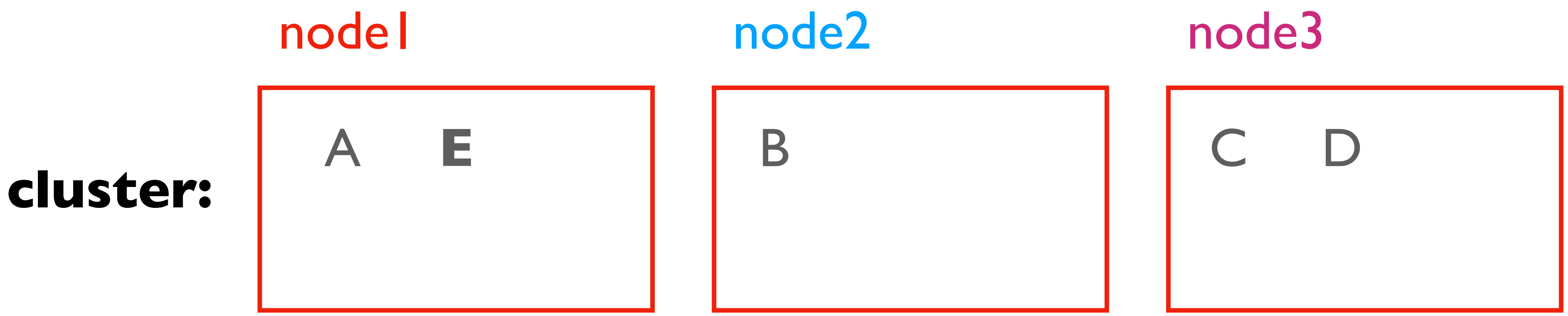
each node's token is the **inclusive end** of a range. A row is mapped to a node based on the range it is in.

Consistent Hashing

Token Map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something



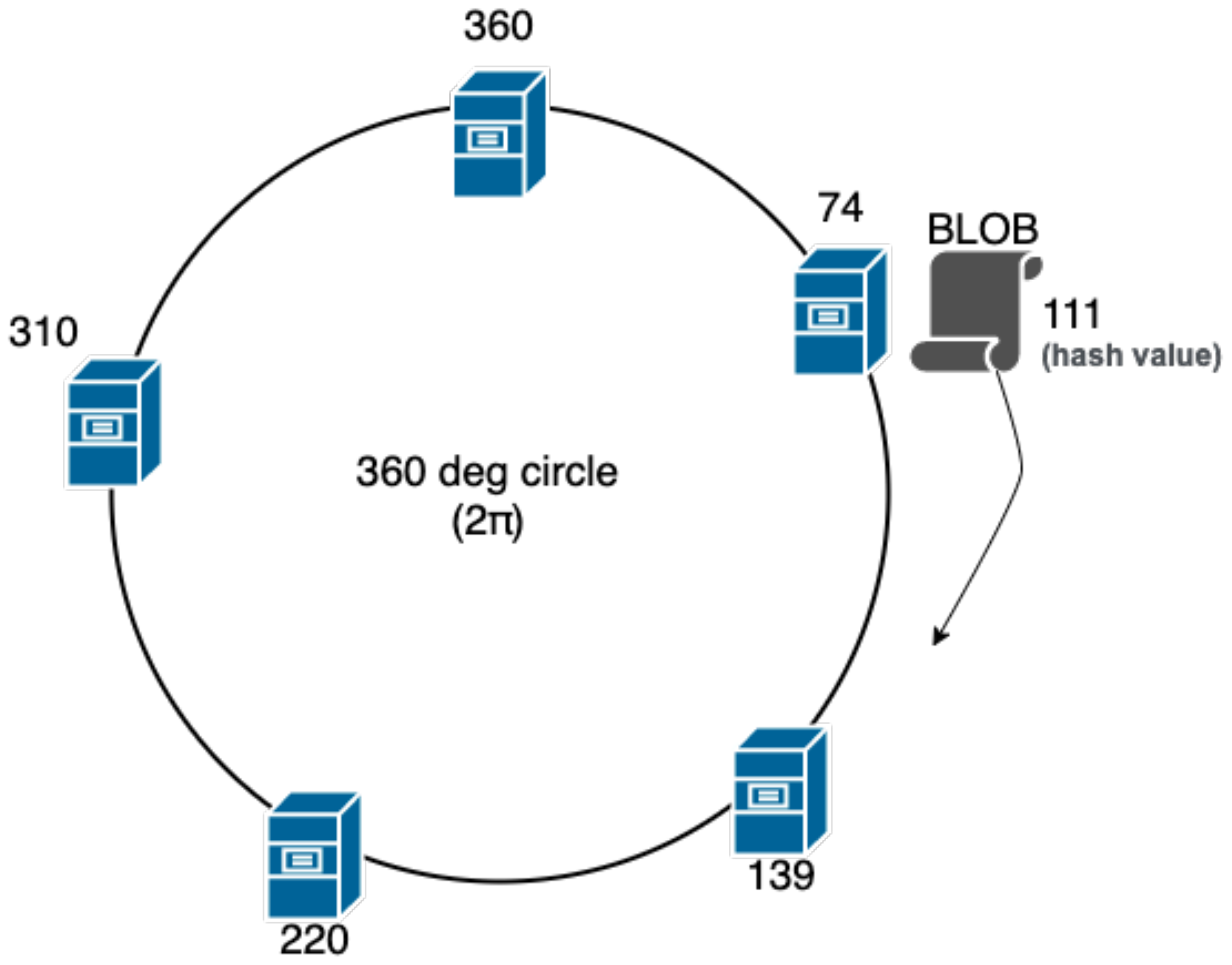
tokens > biggest node token are in the *wrapping range*. Rows in this region go to the node with the smallest token.



Alternate Visualization

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

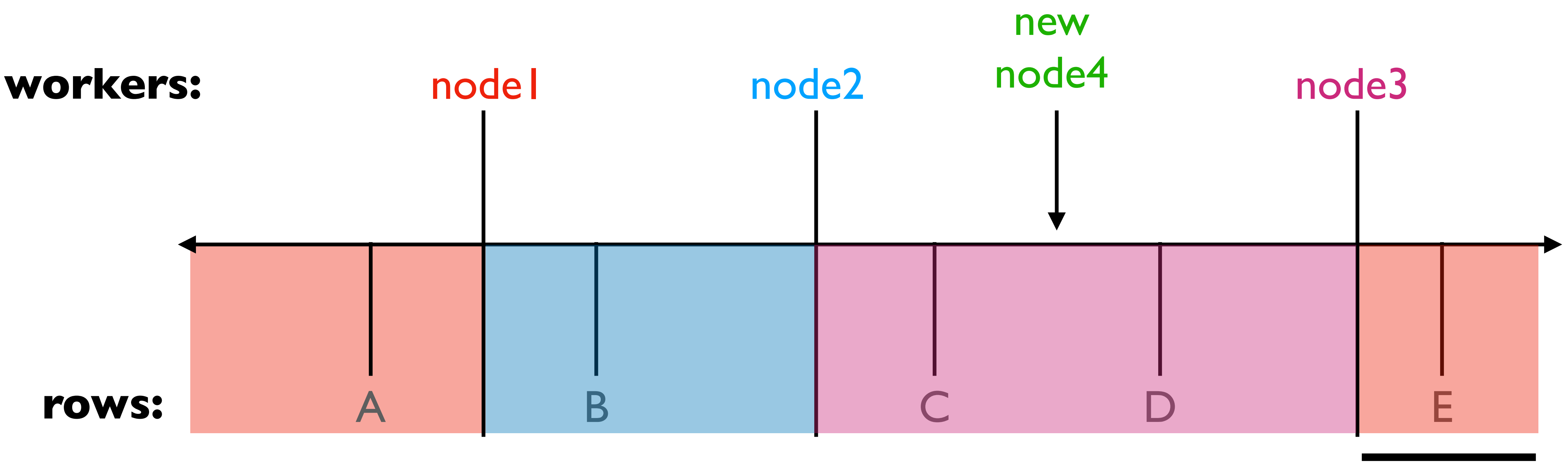
Common visuazilation (e.g., from Wikipedia)



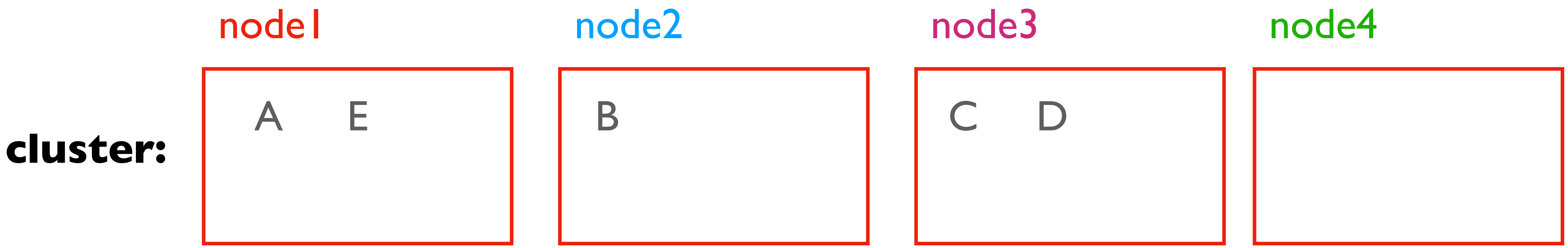
https://en.wikipedia.org/wiki/Consistent_hashing#/media/File:Consistent_Hashing_Sample_Illustration.png

Adding a Node

Token Map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something
token(node4) = pick something

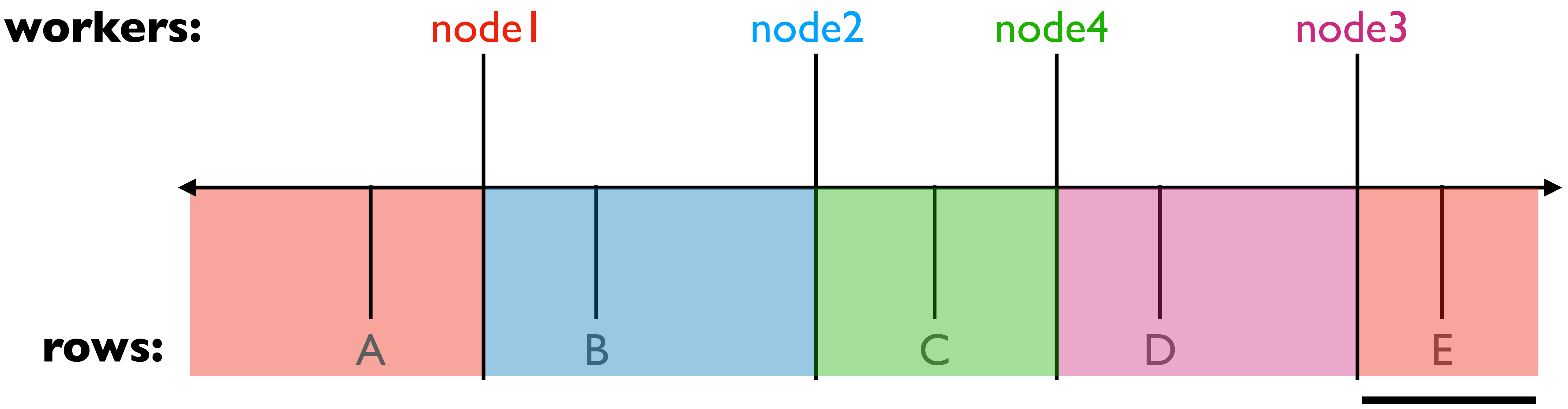


which rows will have to move?
which nodes will be involved?

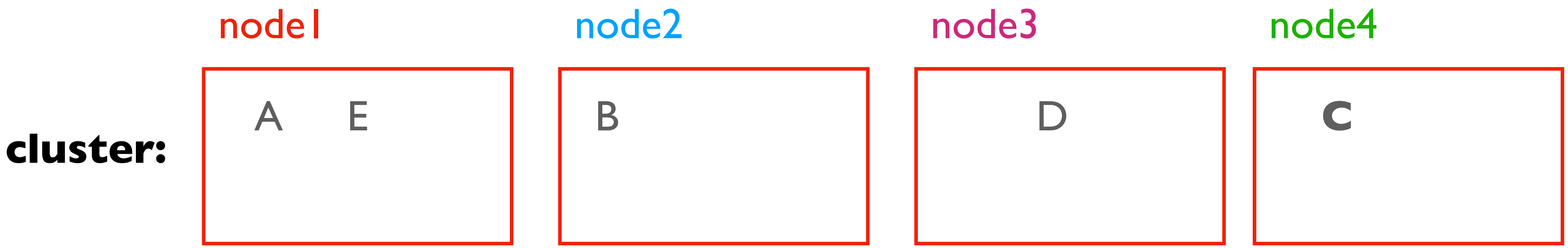


Adding a Node

Token Map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something
token(node4) = pick something

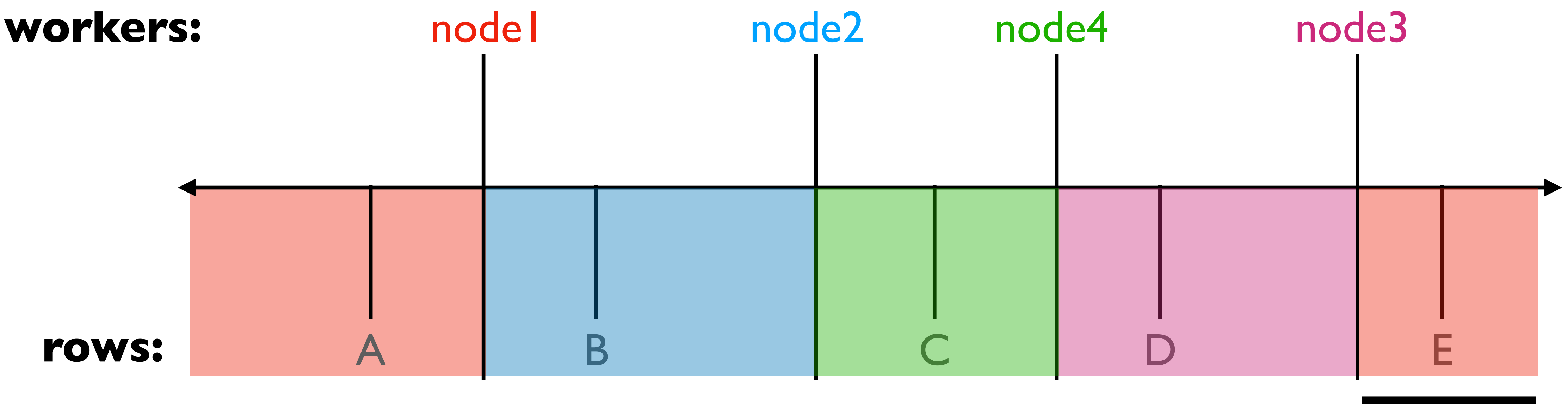


which rows will have to move? Only C
which nodes will be involved? Only node3 and node4



Adding a Node

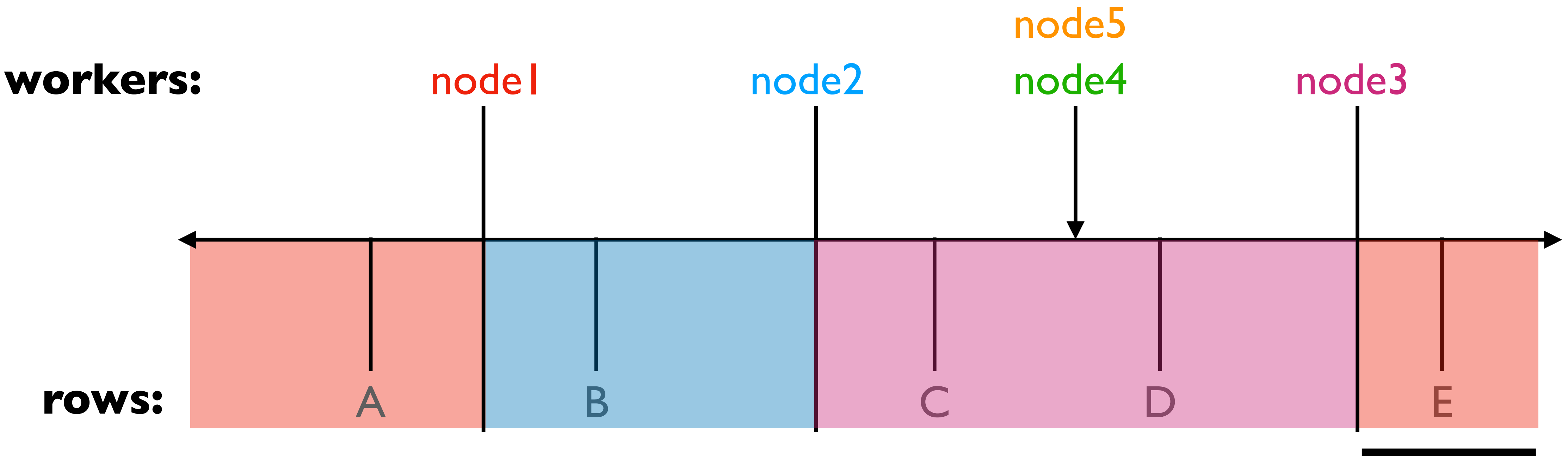
Token Map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something
token(node4) = pick something



Typically, what fraction of the data must move when we scale from N-1 to N?
Hash partinioning: about (N-1)/N of the data
Consistent hashing: about (size of new range)/(size of both ranges) of the data must move.

Collisions

Token Map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something
token(node4) = pick something

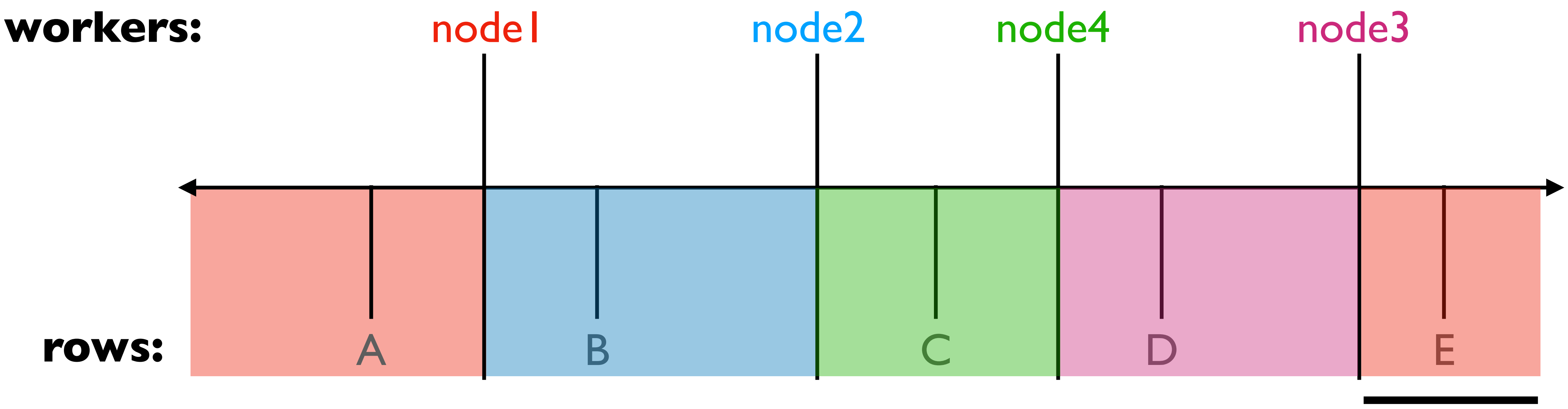


Problem: latest Cassandra versions by default try to choose new node tokens to split big ranges for better balance (instead of randomly picking). Adding multiple nodes simultaneously can lead to collisions, preventing nodes from joining.

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

Sharing the Work

Token Map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something
token(node4) = pick something



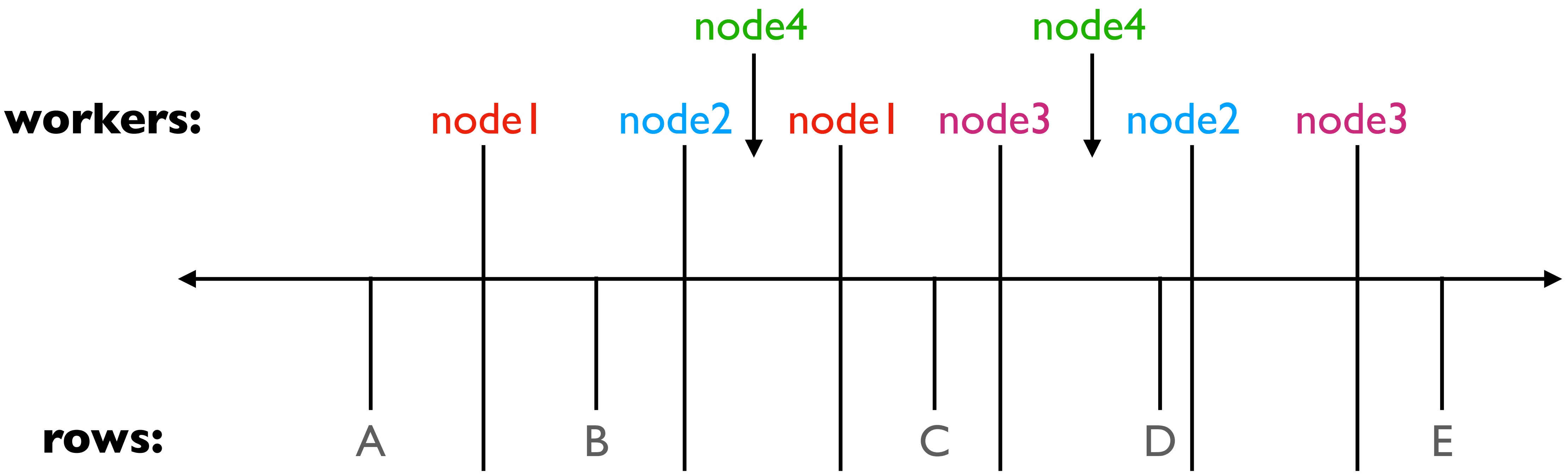
Other problems with adding node 4

- 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:
token(node1) = {t1, t2}
token(node2) = {t3, t4}
token(node3) = {t5, t6}
token(node4) = {t7, t8}

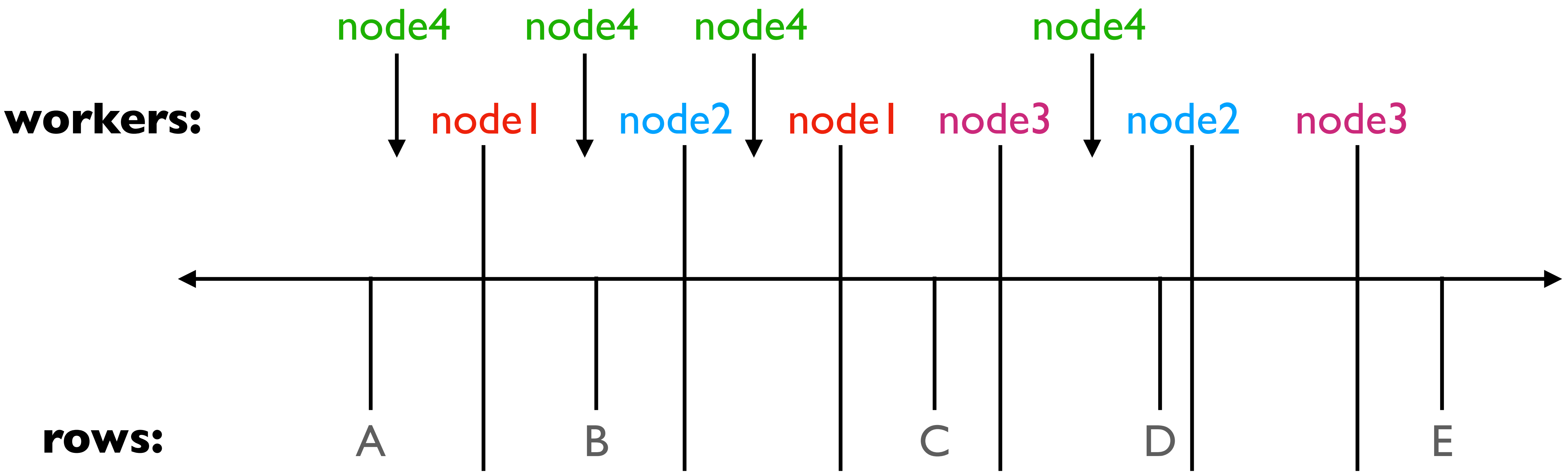


Each node is resonsible for multiple ranges

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

Heterogeneity

Token Map:
token(node1) = {t1, t2}
token(node2) = {t3, t4}
token(node3) = {t5, t6}
token(node4) = {t7, t8, t9, t10}



Heterogeneity: some machines (e.g., newer ones) have more resources

- more powerful nodes can have more vnodes
- probabalisticly, they'll do more work and store more data

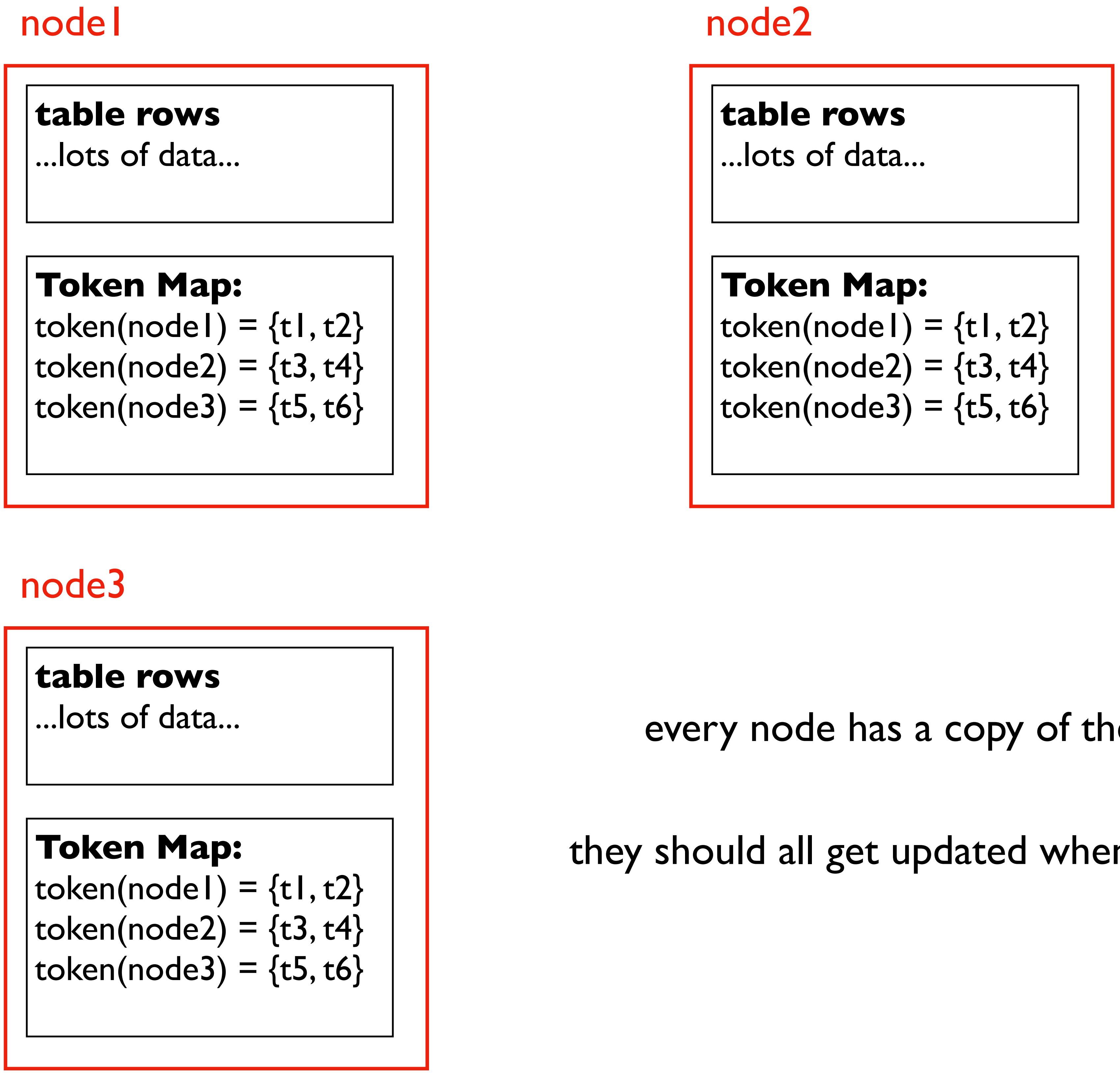
Token Map Storage

Token Map:
token(node1) = {t1, t2}
token(node2) = {t3, t4}
token(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...

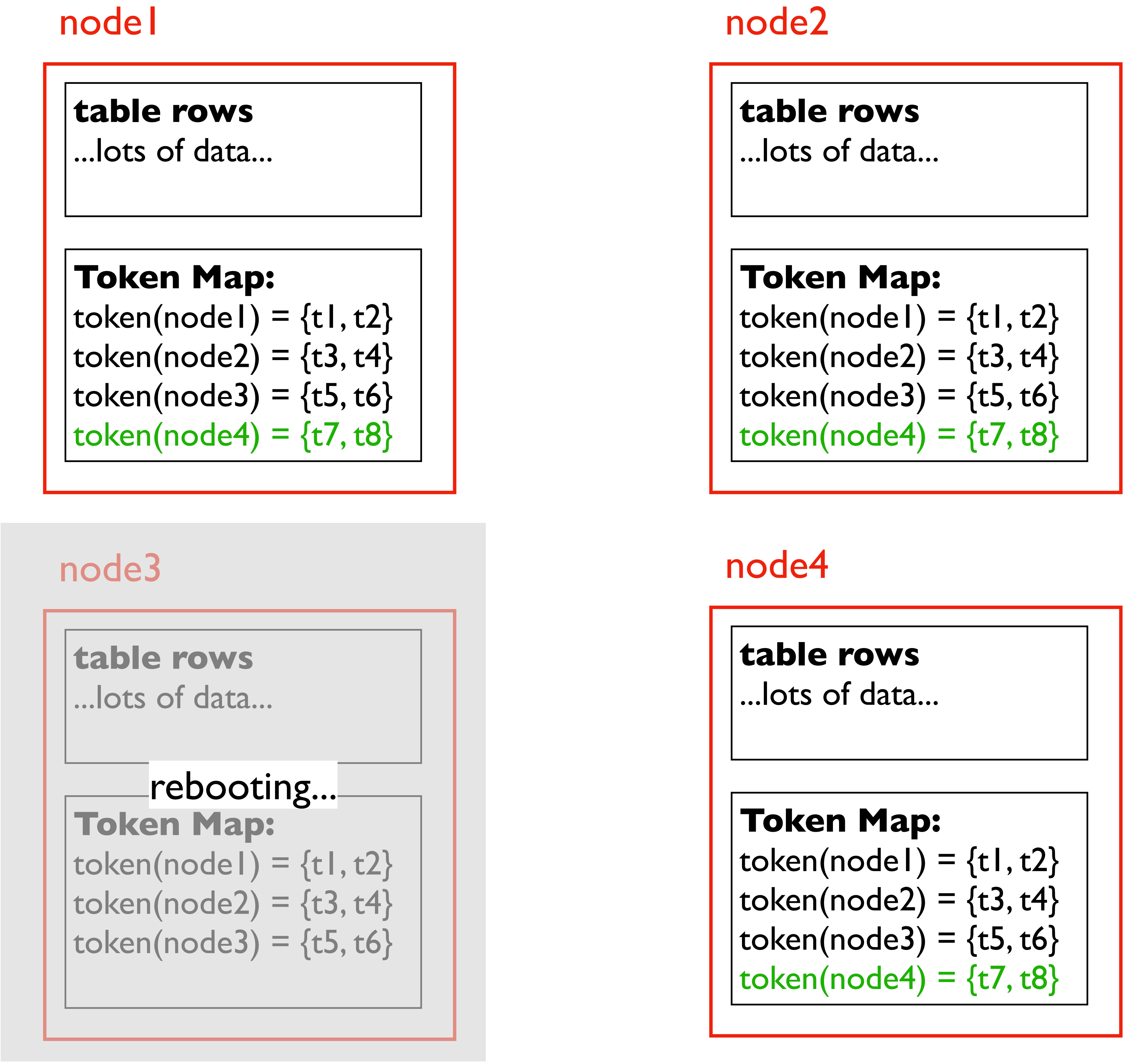
Token Map:
token(node1) = {t1, t2}
token(node2) = {t3, t4}
token(node3) = {t5, t6}

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}

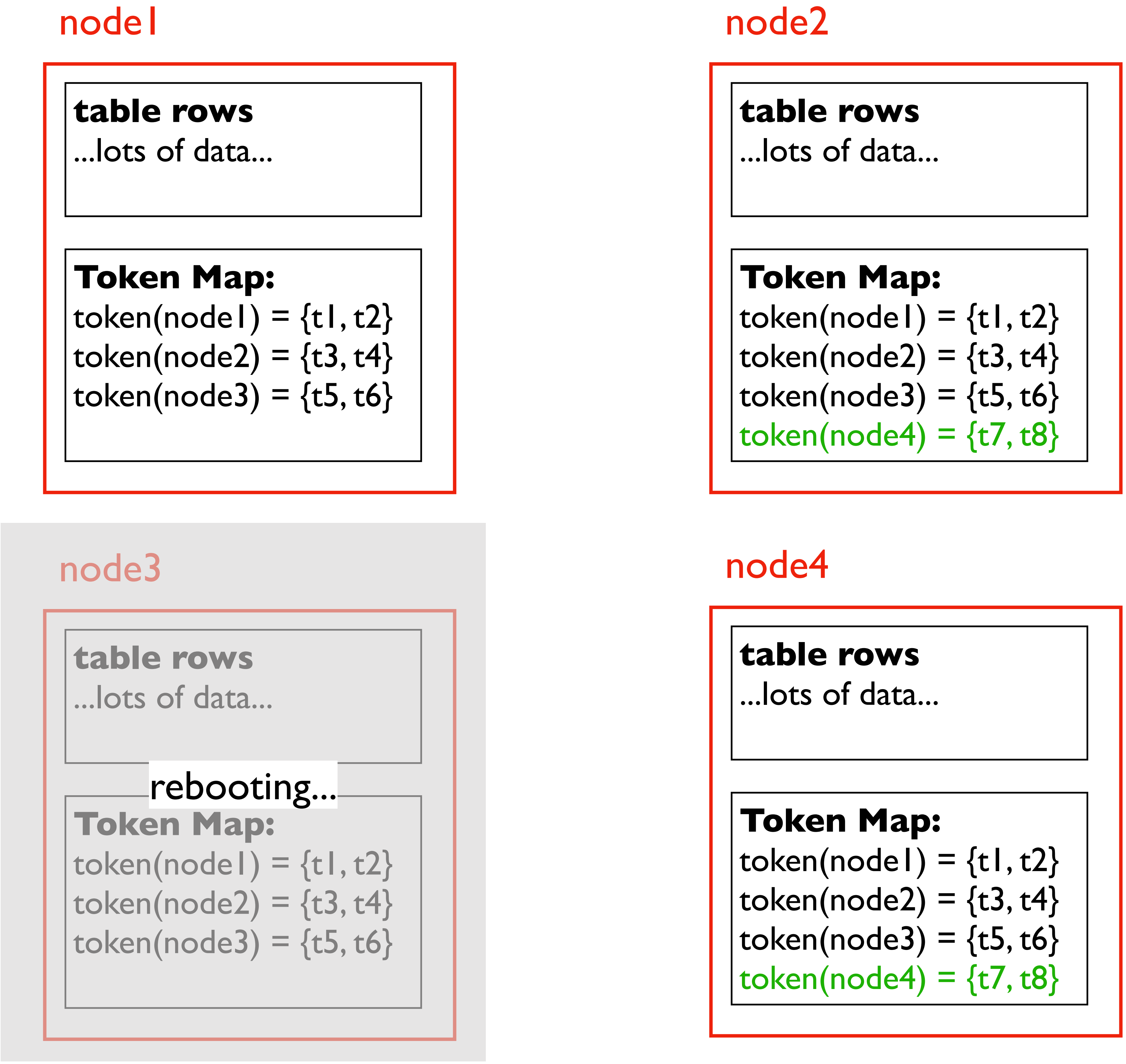
node4

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}

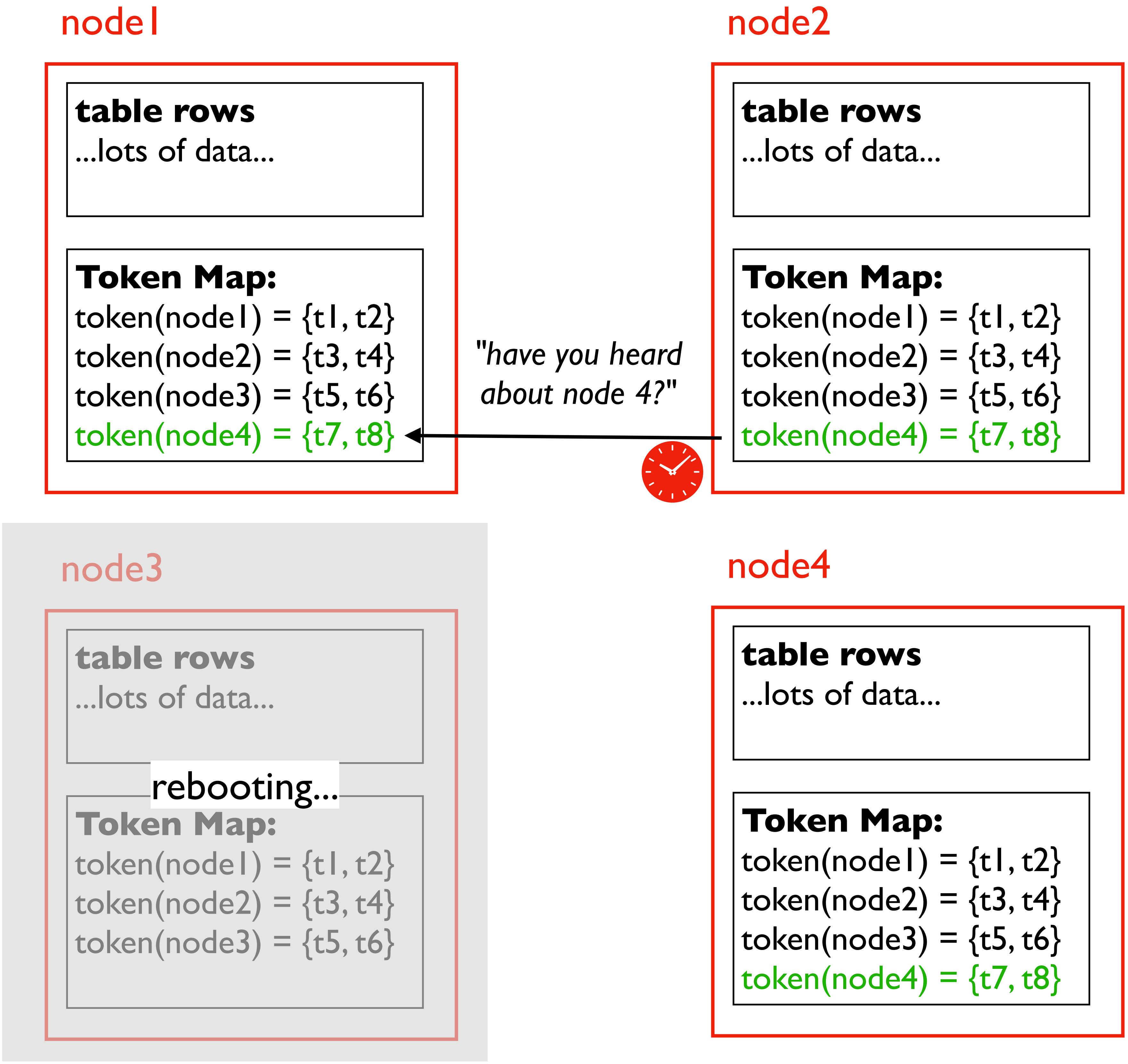
node4

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}

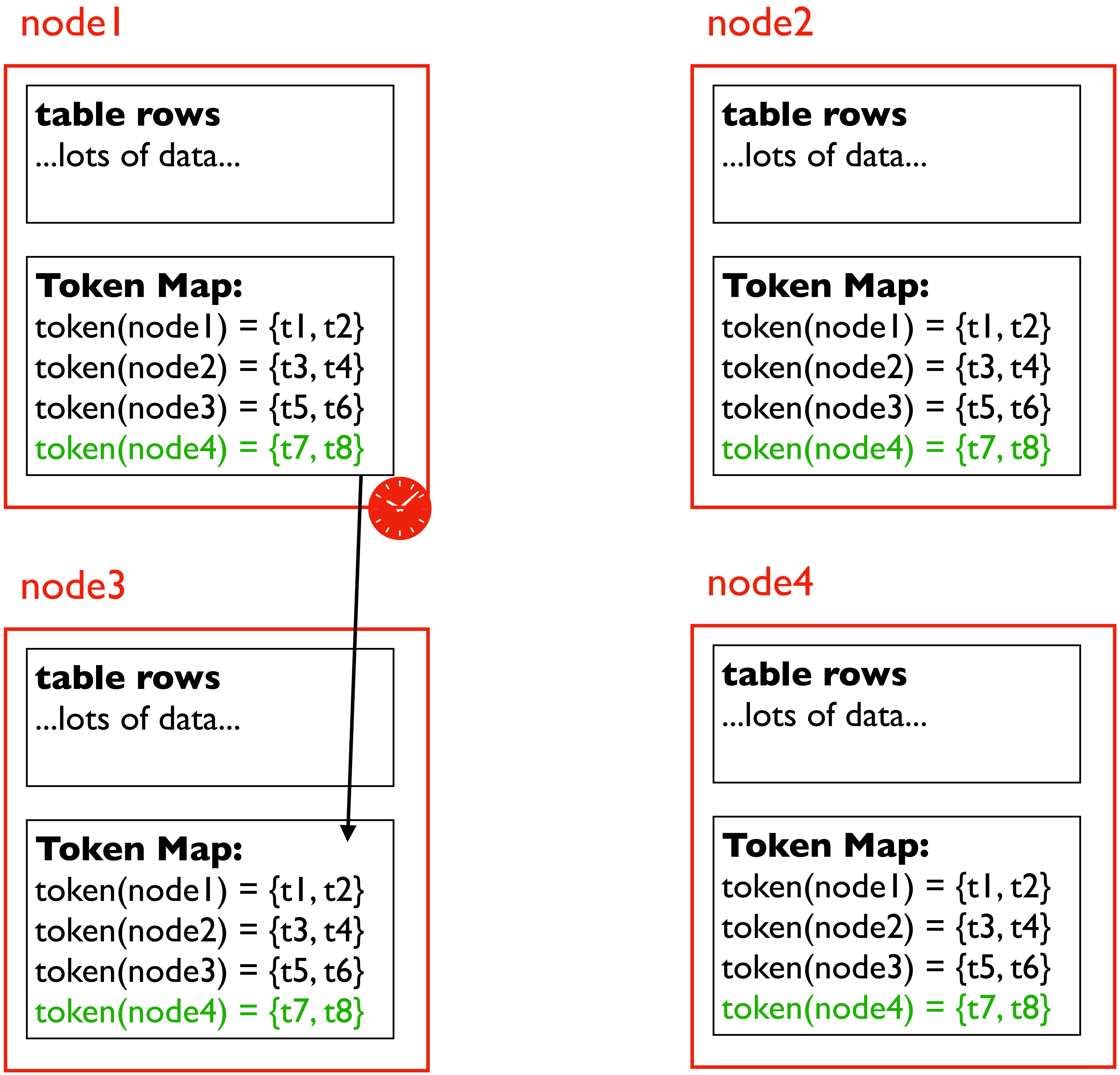
node4

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}

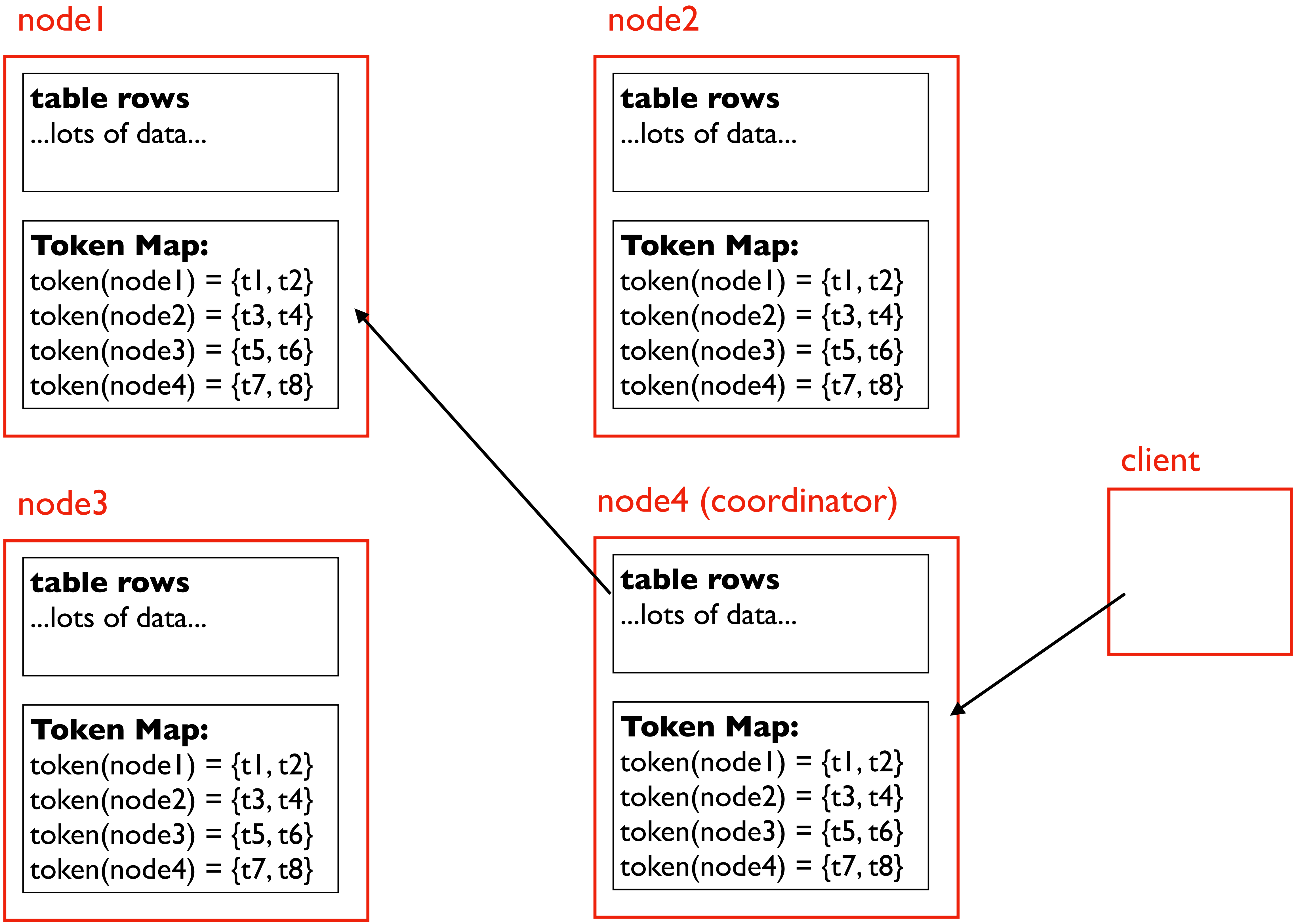
node4

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