Amazon Dynamo

DS 5110/CS 5501: Big Data Systems

Spring 2024

Lecture 10b

Yue Cheng

University of Virginia

Some material taken/derived from:

- Princeton COS-418 materials created by Michael Freedman.
- Wisconsin CS 544 by Tyler Caraza-Harter.

@ 2024 released for use under a CC BY-SA license.

Learning objectives

- Identify strengths and weaknesses of different data partitioning techniques
- Interpret a token ring to assign a data to a Dynamo storage node
- Describe how gossip protocol can be used to replicate (meta)data across nodes in a cluster, without need for a centralized metadata server

Dynamo impact

Motivation: Workloads like shopping cart do not need the SQL level of complexity and transaction guarantee!

Dynamo
(2007 paper)

Goal: Highly available when things are failing

Partitioning + replication

Customer demands

Cassandra
(2008 release)

DynamoDB
(2012 release)

Amazon's infrastructure (circa 2007)

- **Tens of thousands** of servers in globally-distributed data centers
- Peak load: Tens of millions of customers
- Tiered service-oriented architecture
  - **Stateless** web page rendering servers, atop
  - **Stateless** aggregator servers, atop
  - **Instances of stateful** data stores (**e.g.** Dynamo instances)
    - **`put()`, `get()`**: values "usually less than 1 MB"

Each instance contains **a few hundred** servers

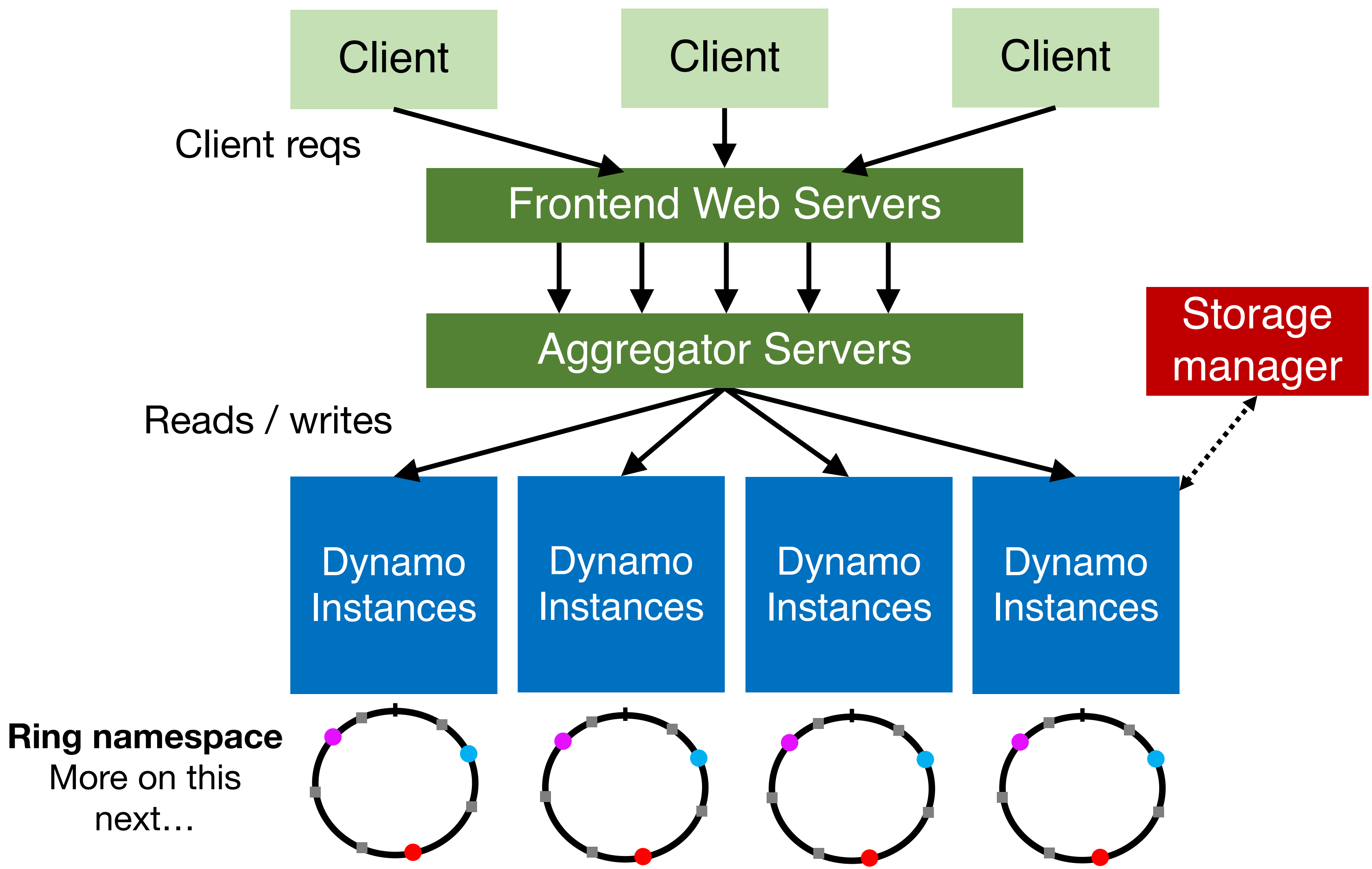
How does Amazon use Dynamo?

- Shopping cart
- Need a data storage to store lots of states
  - Product list (mostly read-only, replicated for high throughput)
  - Recently visited products
  - Orders

Dynamo requirements

- Highly available writes despite failures
  - Despite disks failing, network routes flapping, "data centers destroyed by tornadoes"
  - Always respond quickly, even during failures → replication
- Low request-response latency: focus on **99.9%** SLA
- Incrementally scalable as servers grow to workload
  - Adding "nodes" should be seamless
- Comprehensible conflict resolution
  - High availability in above sense implies conflicts

Amazon Dynamo architecture

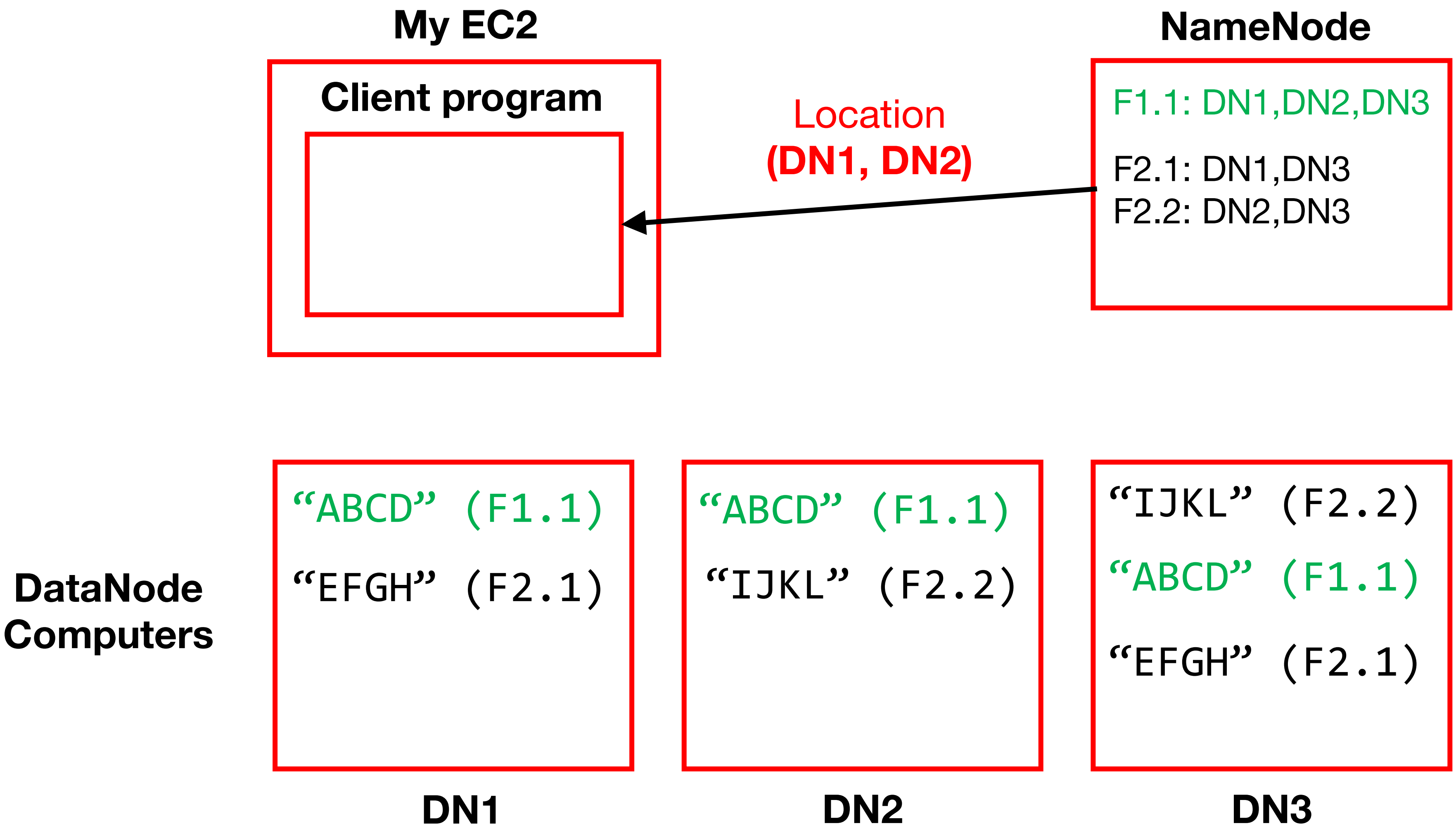


Partitioning approaches

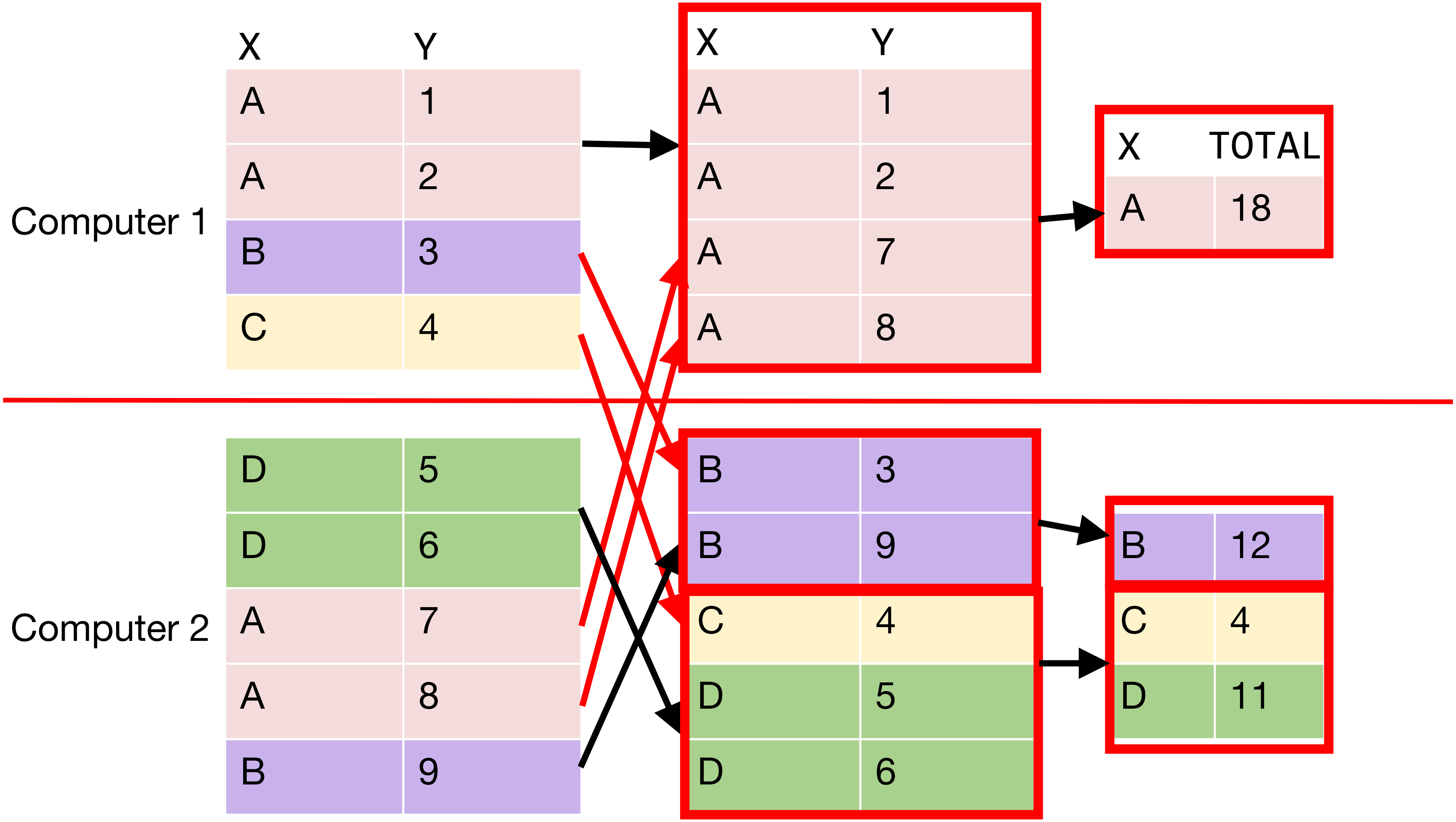
Given many machines and a piece of data, how do we decide where it should live?

- Mapping table
    - Location = {"fileA-block0": [datanode 1, …], …}
    - HDFS NameNode uses this
- Hash partitioning
    - Partition = hash(key) % partition_count
    - Spark shuffle uses this (for joining, grouping, etc.)
- Consistent hashing
    - **Dynamo** uses this

Review: HDFS partitioning



Review: Spark hash partitioning




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

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

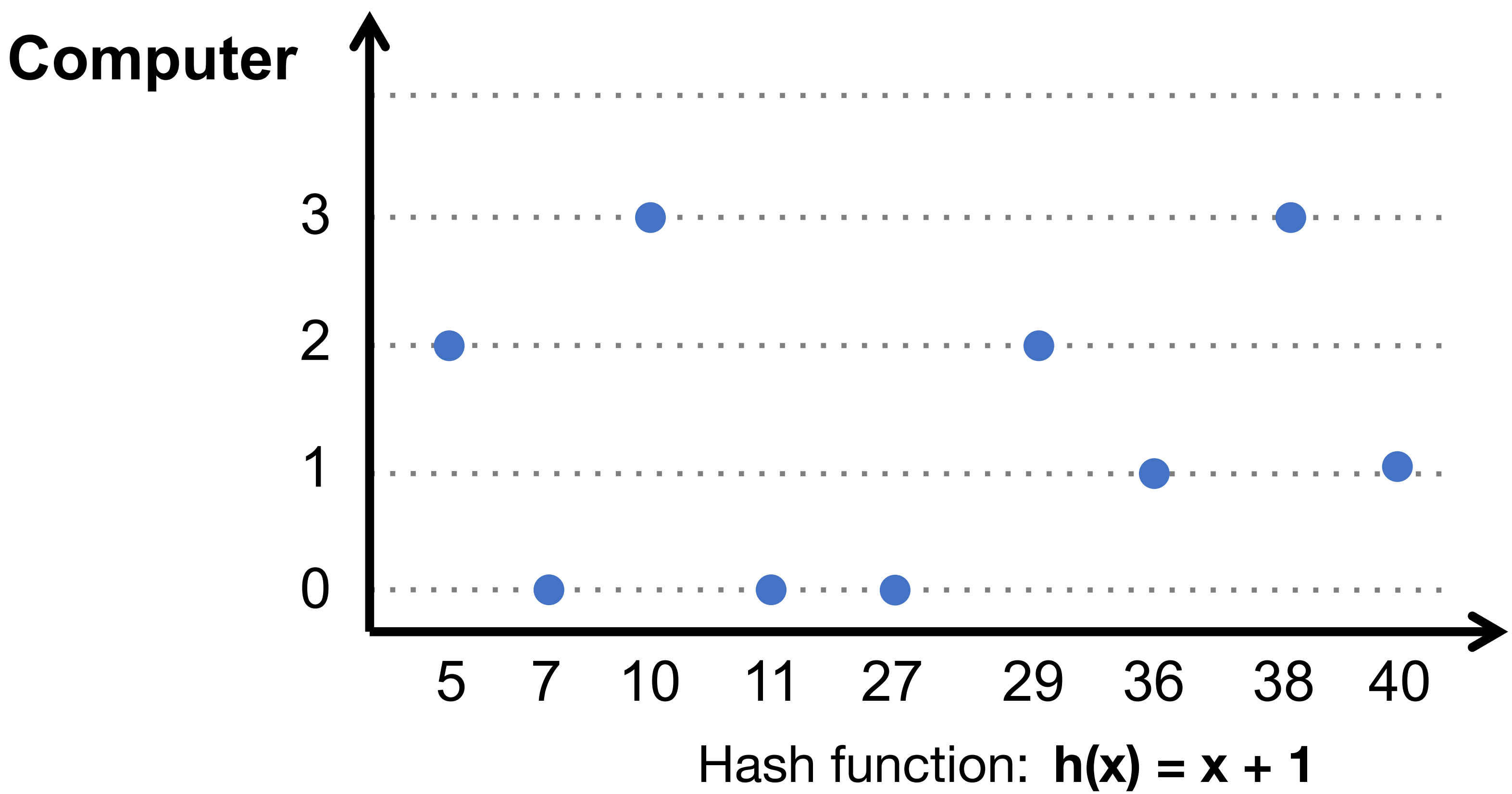
- Incrementally scaling: Can we efficiently add more machines to an already large cluster?

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

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

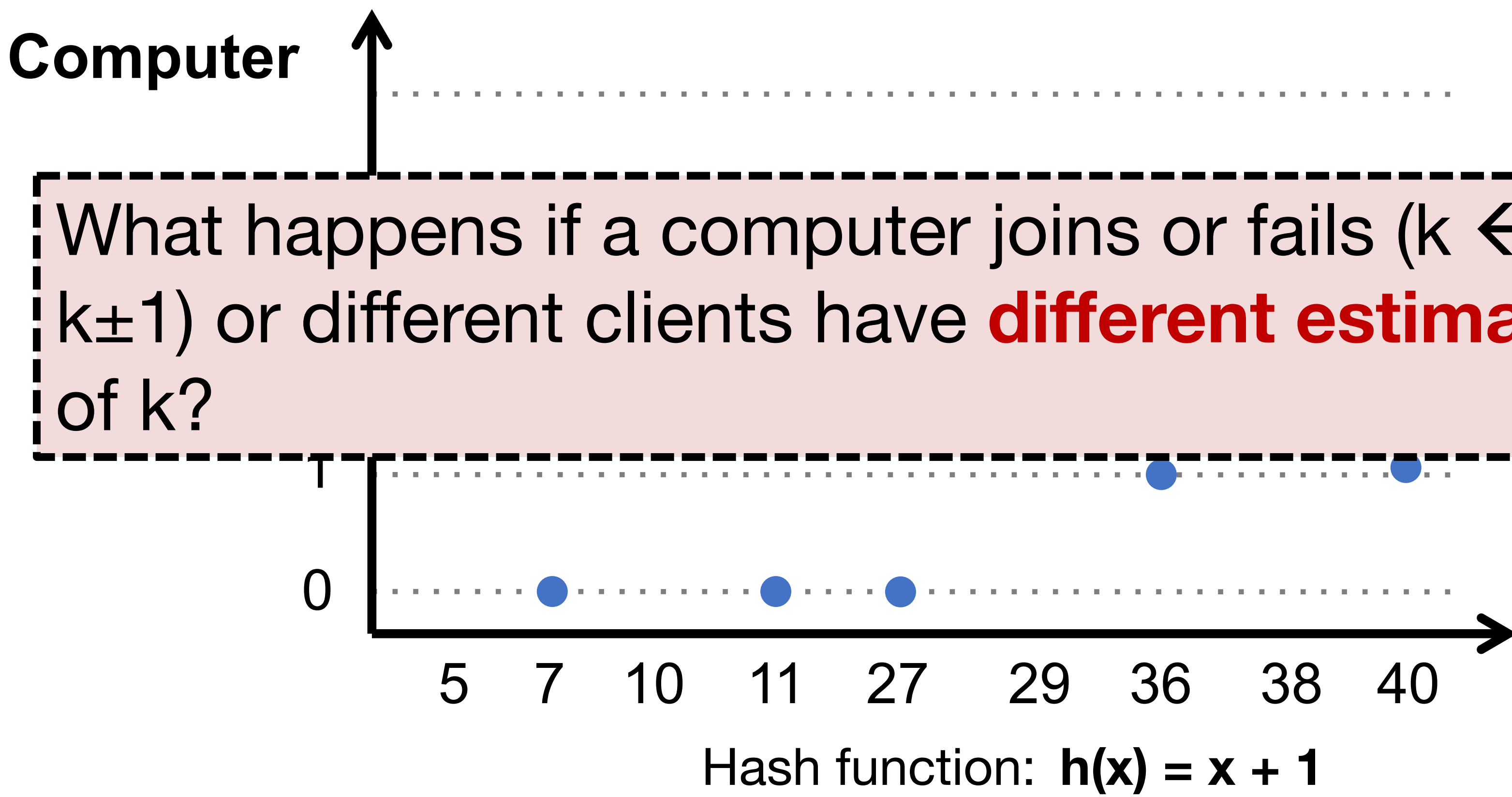
Problem for hash partitioning: Changing number of computers

$i = h(x) \% 4$



Problem for hash partitioning: Changing number of computers

i = h(*x*) % 4

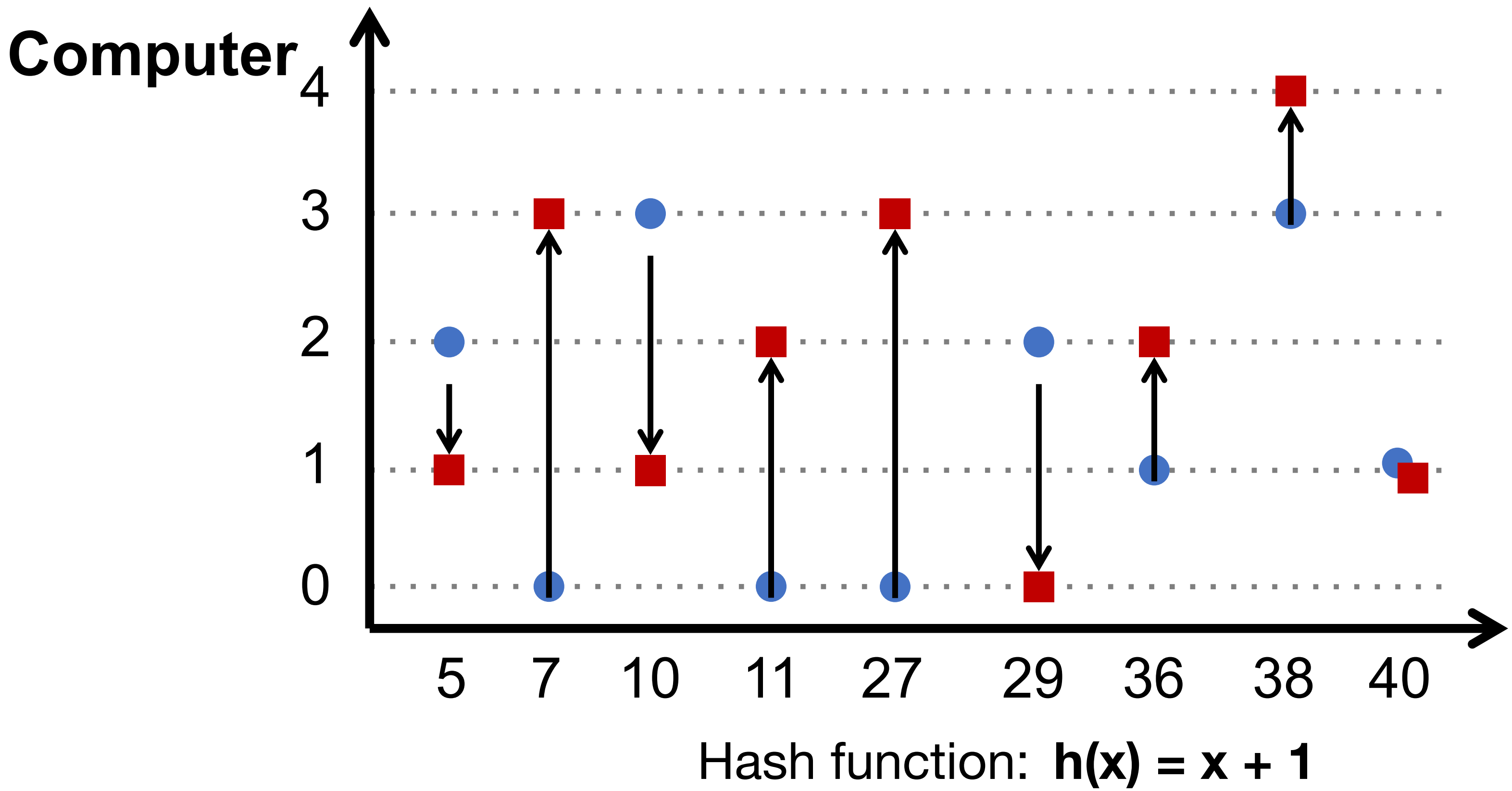


What happens if a computer joins or fails (k ← k±1) or different clients have **different estimate** of k?

Problem for hash partitioning: Changing number of computers

i = h(*x*) % 4

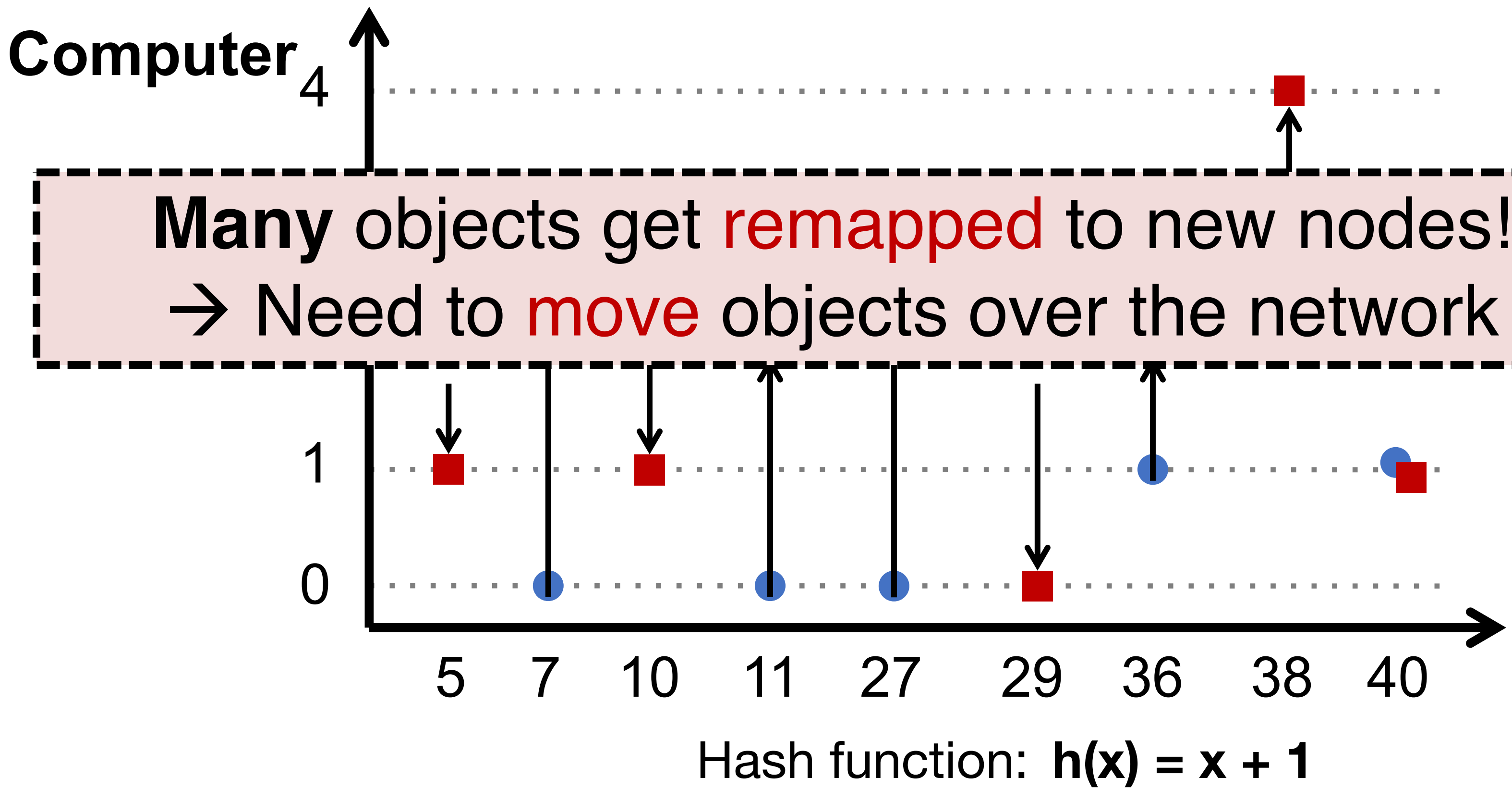
Add one machine: i = h(*x*) % 5



Problem for hash partitioning: Changing number of computers

i = h(*x*) % 4

Add one machine: i = h(*x*) % 5



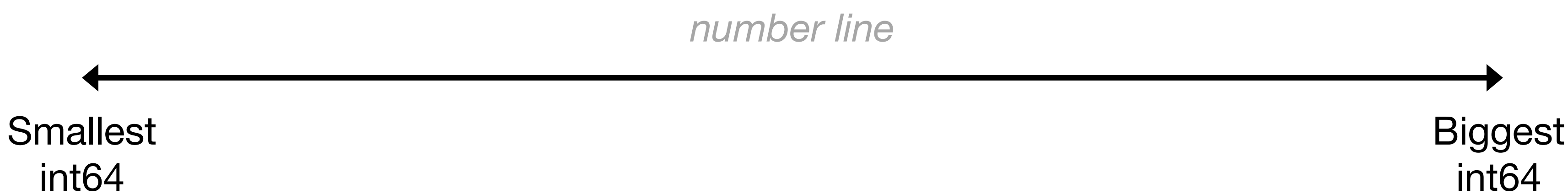
Many objects get remapped to new nodes!
→ Need to move objects over the network

Partitioning approaches

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

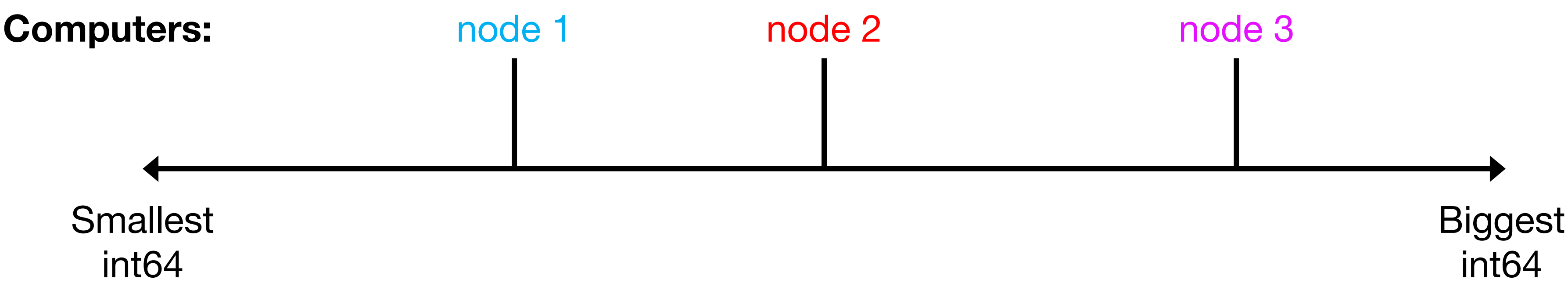
- Mapping table
  - Location = {"fileA-block0": [datanode 1, …], …}
  - HDFS NameNode uses this
- Hash partitioning
  - Partition = hash(key) % partition_count
  - Spark shuffle uses this (for joining, grouping, etc.)
- **Consistent hashing**
  - **Dynamo** uses this
  - token = hash(key) # every token is in a range, indicating the machine
  - location = {range(0,10): "machine1", range(10,20): "machine2", …}

Consistent hashing



Consistent hashing

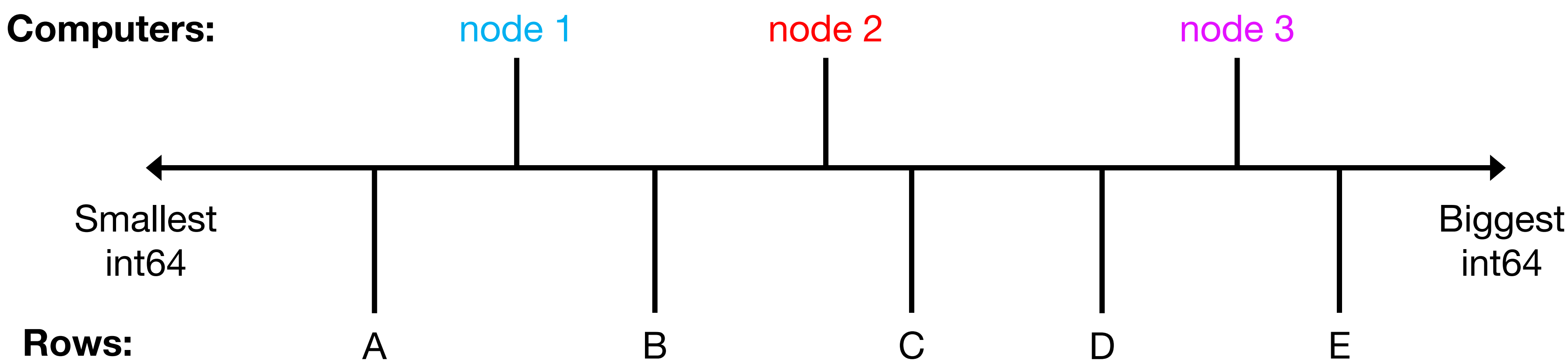
Token map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something



Assign every **computer** a point on the number line.
Could be random (though newer approaches are cleverer).
No hashing needed, yet!

Consistent hashing

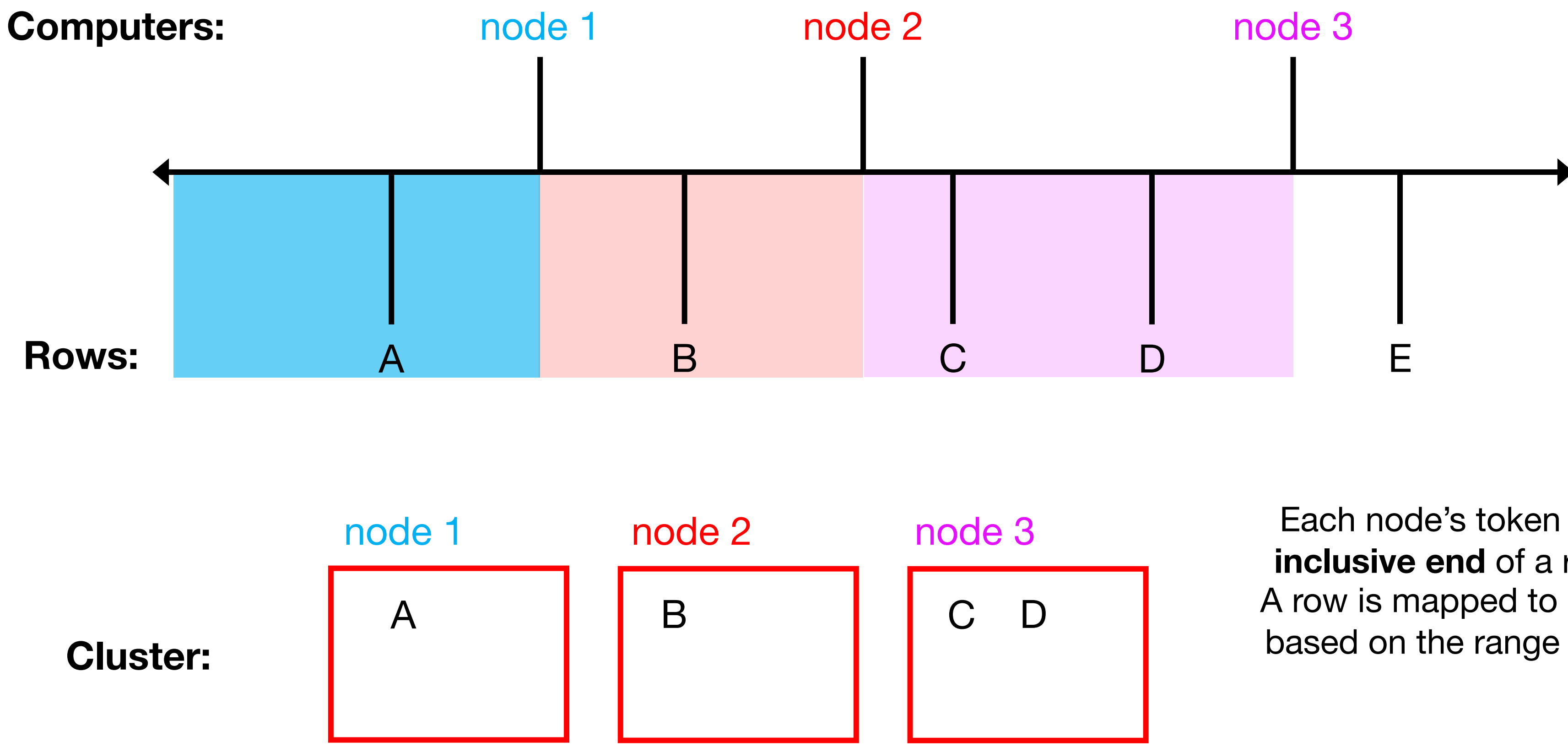
Token map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something



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

Consistent hashing

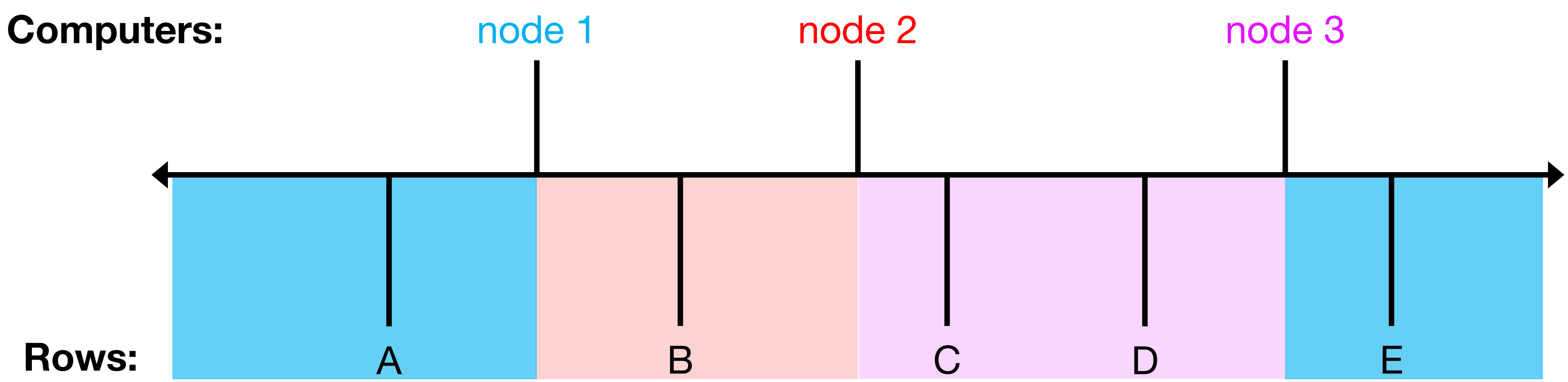
Token map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something



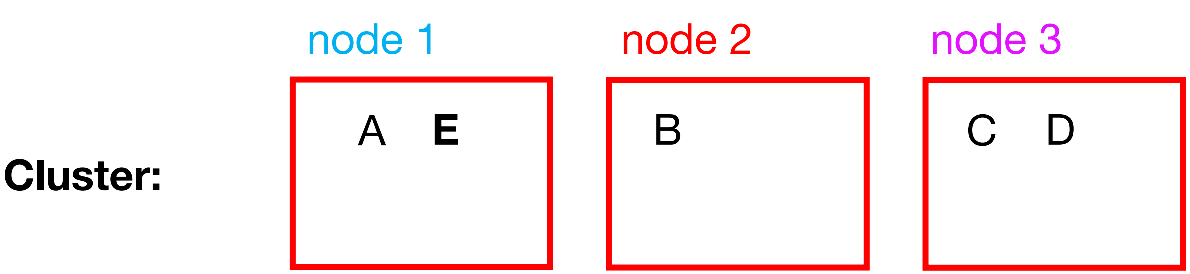
Each node's token is the **inclusive end** of a range. A row is mapped to a node based on the range it is in.

Consistent hashing

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

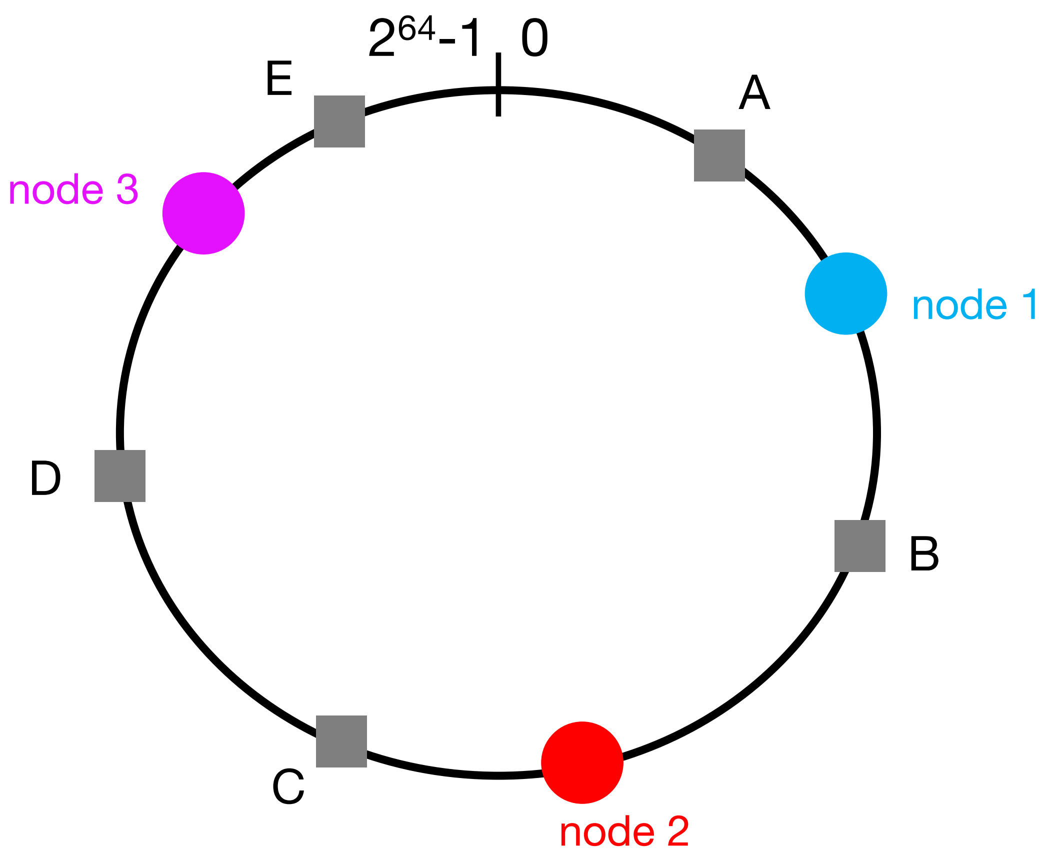


tokens > biggest node token are in the **wrapping range**. Rows in this region go to the node with the smallest token (**i.e., a ring space**).



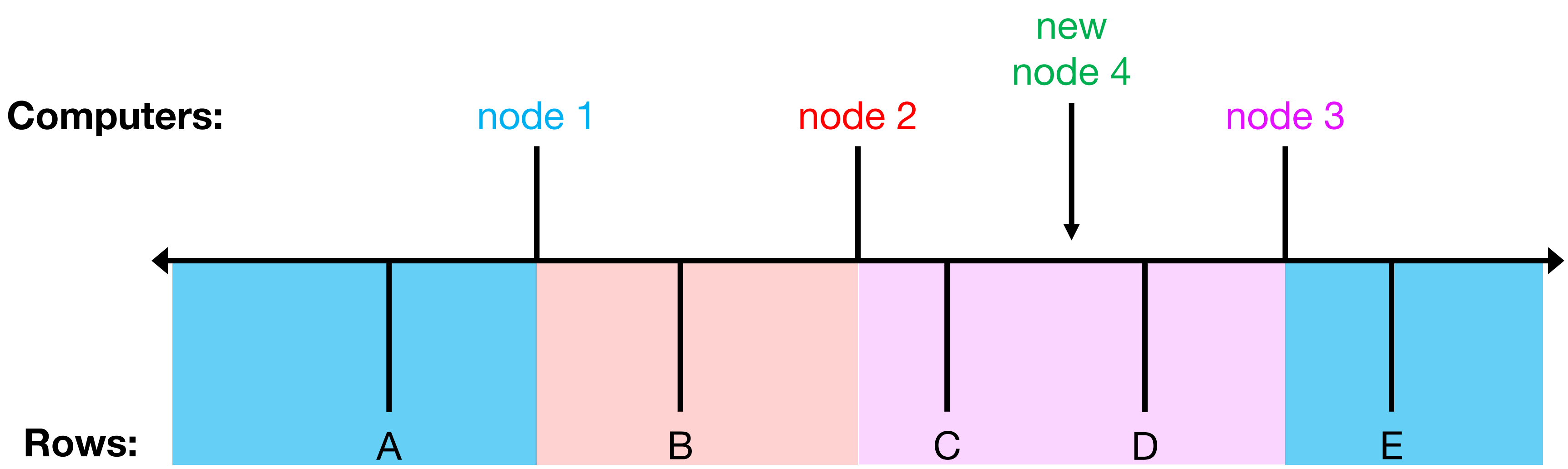
Alternate ring-based visualization

Given the wrapping, clusters using consistent hashing form a "**token ring**"

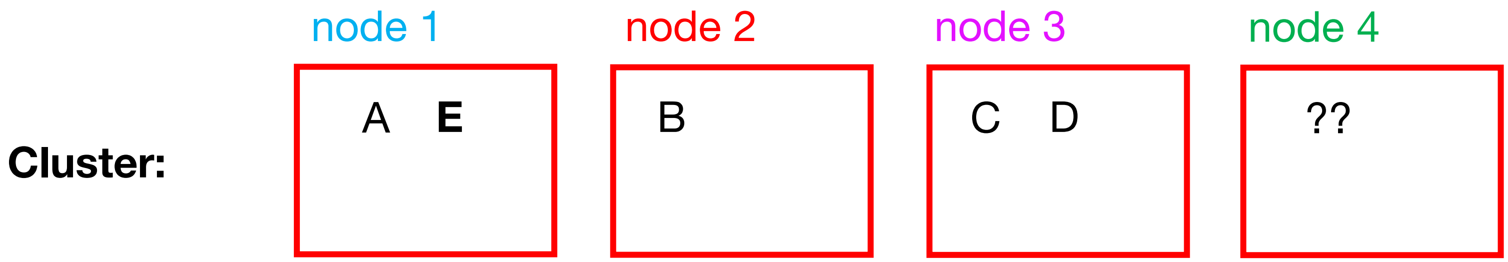


Adding a node

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

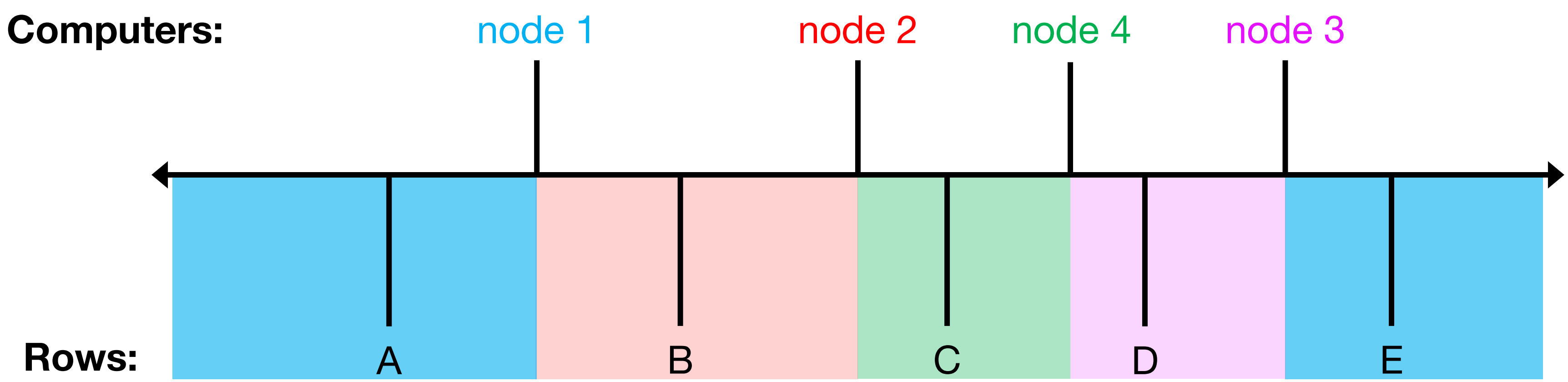


Which rows will have to move?
Which nodes will be involved?

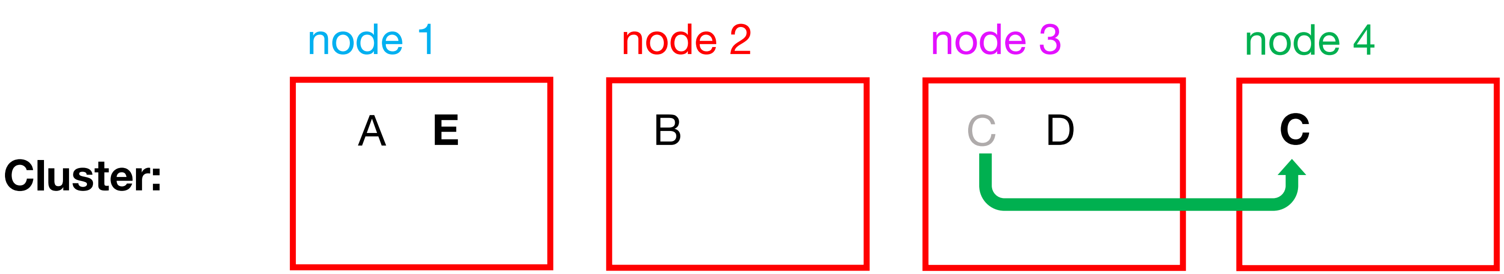


Adding a node

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

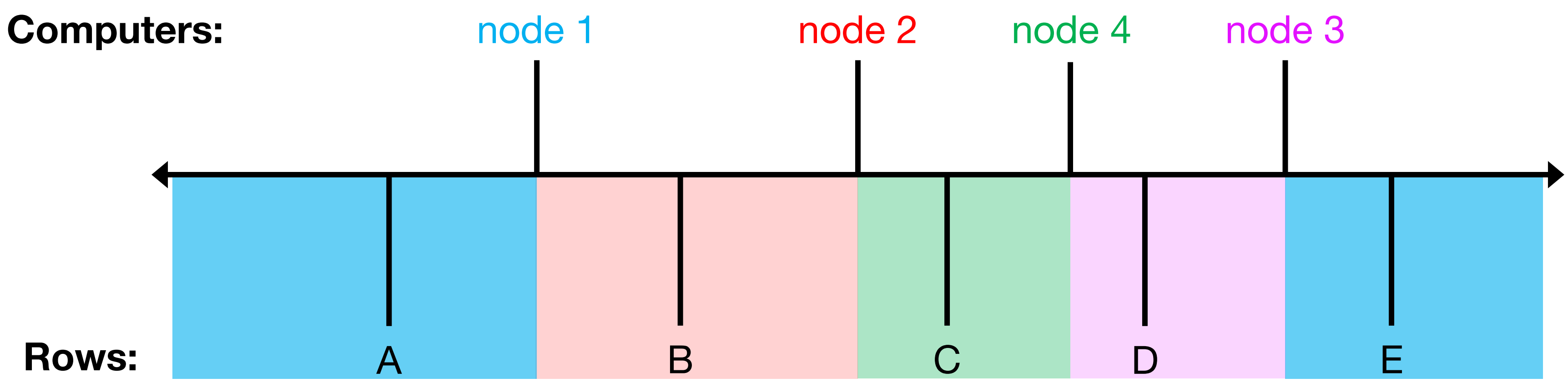


Which rows will have to move? **Only C**
Which nodes will be involved? Only node 3 and node 4



Adding a node

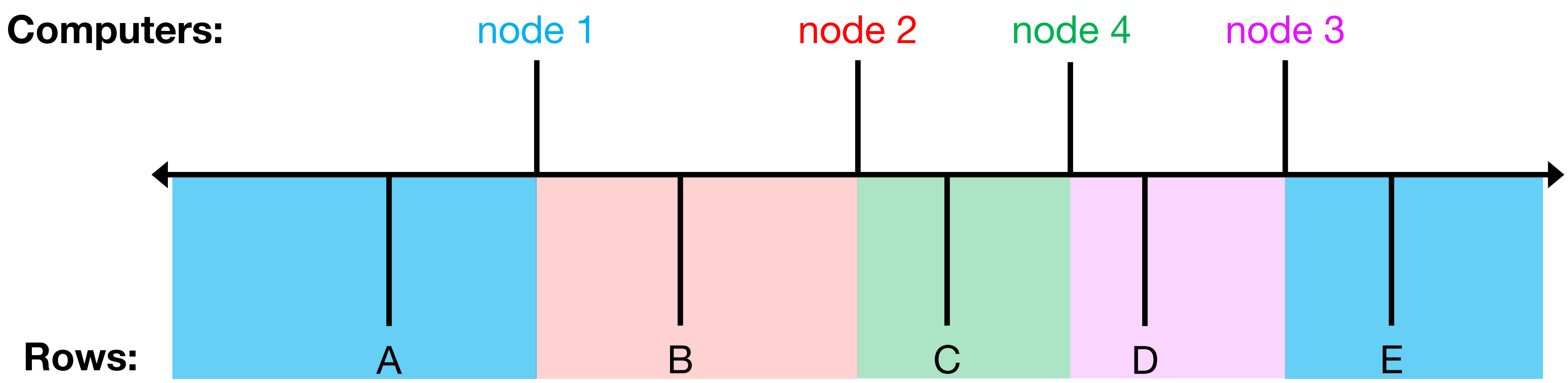
Token map:
token(node1) = pick something
token(node2) = pick something
token(node3) = pick something
token(node4) = pick something



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

Problems

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



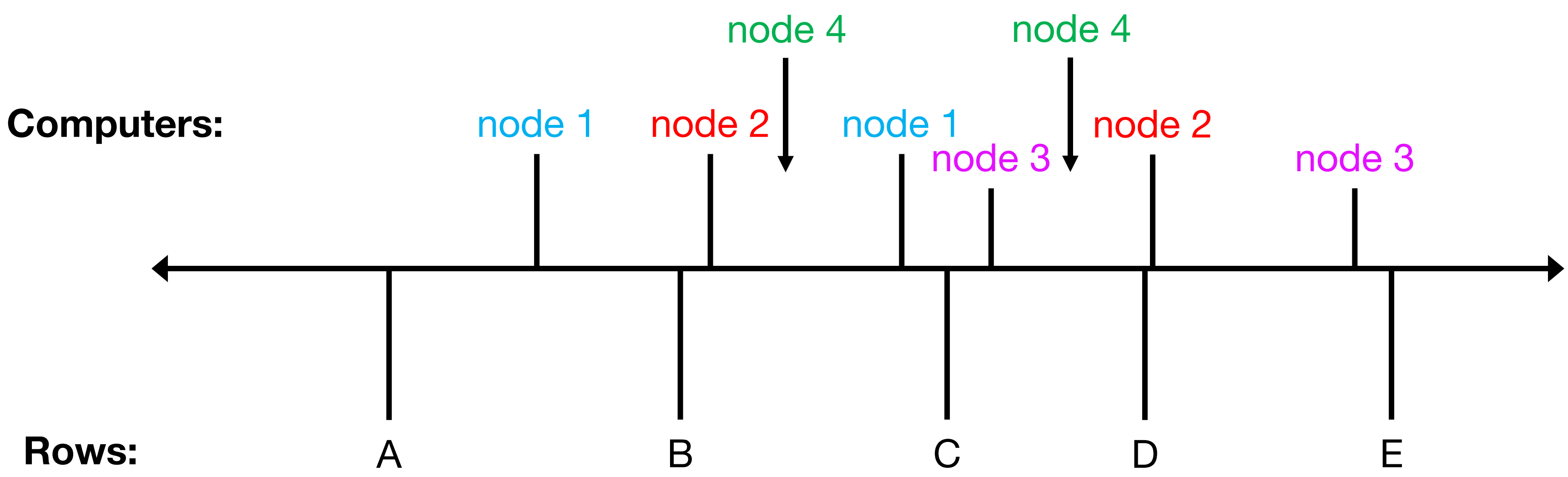
Problems with adding node 4:

- **Long term**: Only load of node 3 is alleviated
- **Short term**: Node 3 bears all the burden of transferring data to node 4

Solution: introducing vnodes (virtual nodes)

Virtual nodes (vnodes)

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

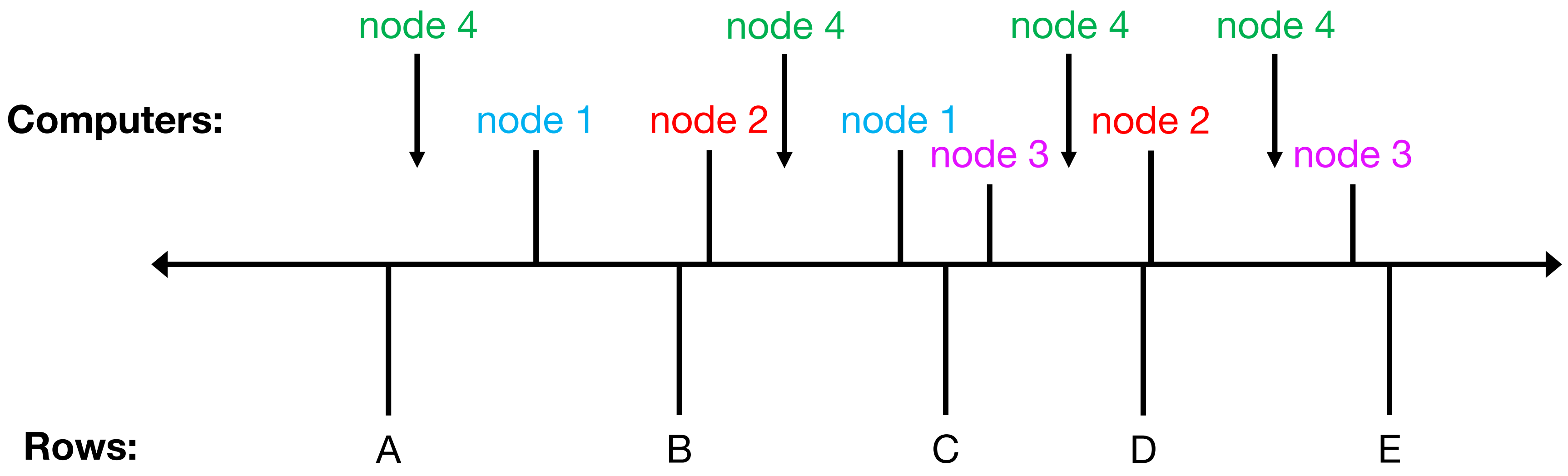


Each (physical) node is responsible for multiple ranges (in this case, 2)

- How many vnodes per node is configurable
- Node 4 will share some load off node 1 and node 2
- Achieves better (short-term and long-term) load balancing with a larger number of 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 capacity, thus more vnodes
- Probabilistically, 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}
token(node4) = {t7, t8}

Where should this live?

We don't want **a single point of failure** (like an HDFS NameNode).

Token map storage

node 1

table rows
…lots of data…

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

node 2

table rows
…lots of data…

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

node 3

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 global token map.

They should all get updated when new nodes join.

Adding nodes: Bad approach

node 1

table rows
…lots of data…

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

node 2

table rows
…lots of data…

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

node 3

table rows
…lots of data…

rebooting…

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

node 4

table rows
…lots of data…

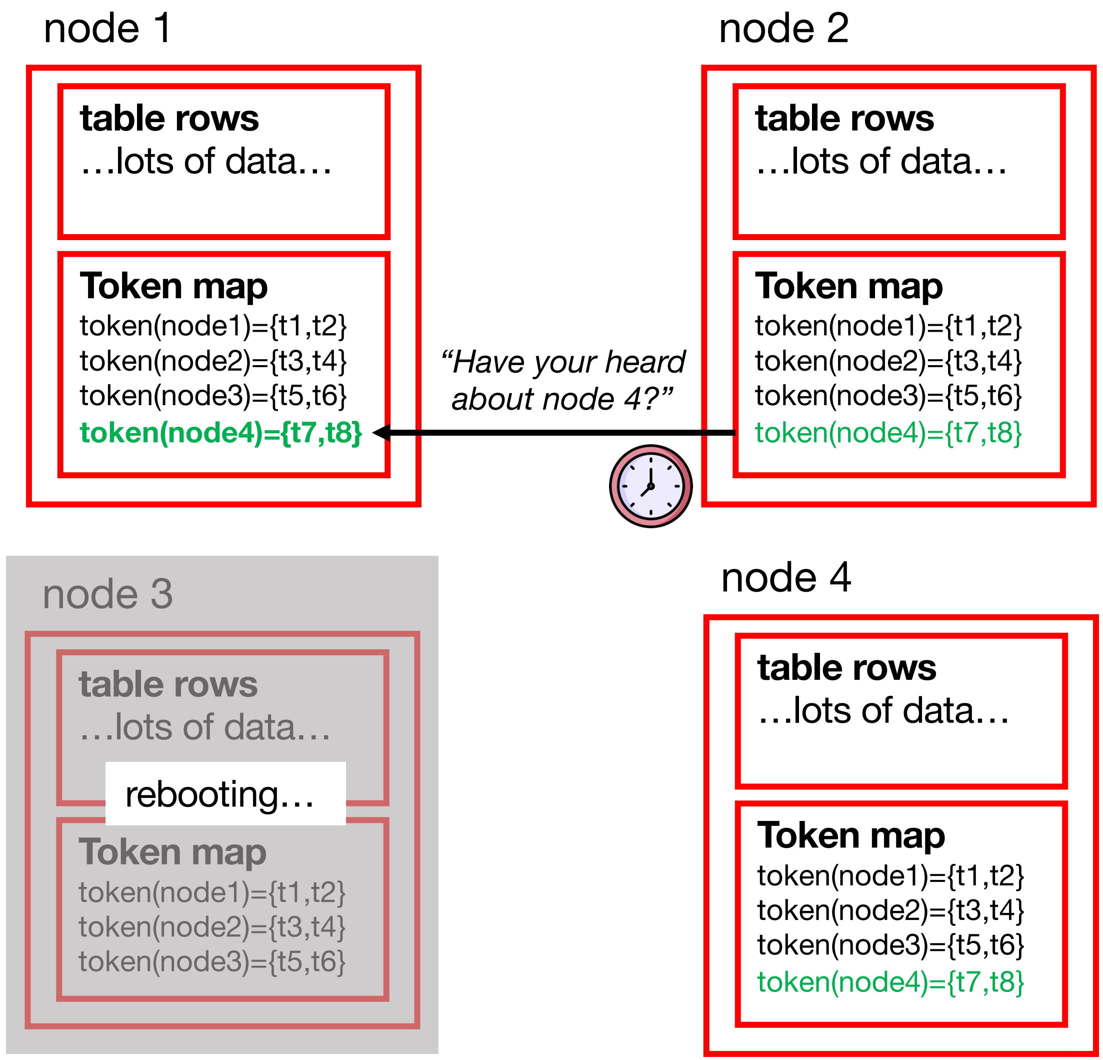
Token map
token(node1)={t1,t2}
token(node2)={t3,t4}
token(node3)={t5,t6}
token(node4)={t7,t8}

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

Better approach: Gossip

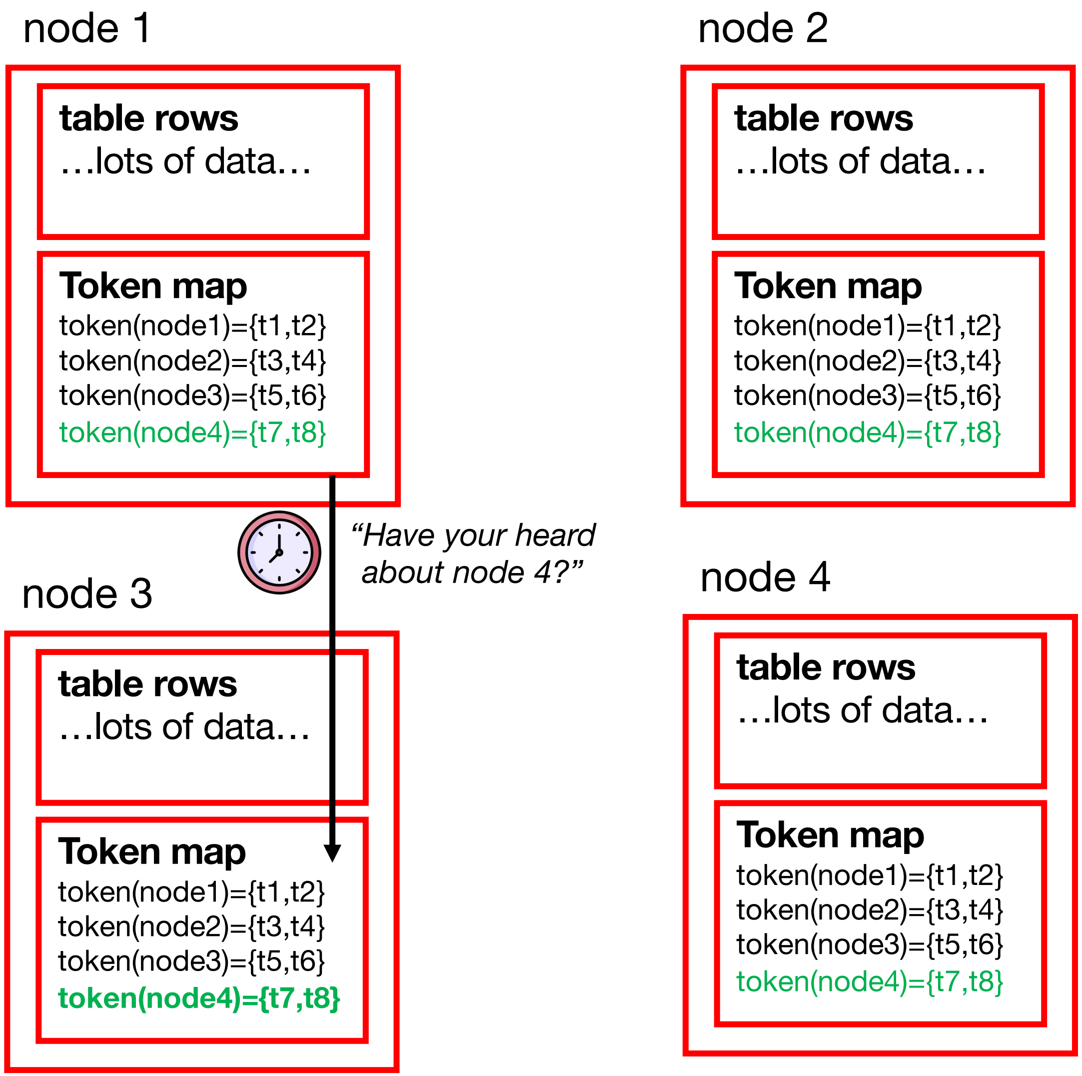
Just inform one or a few nodes about the new node

Better approach: Gossip



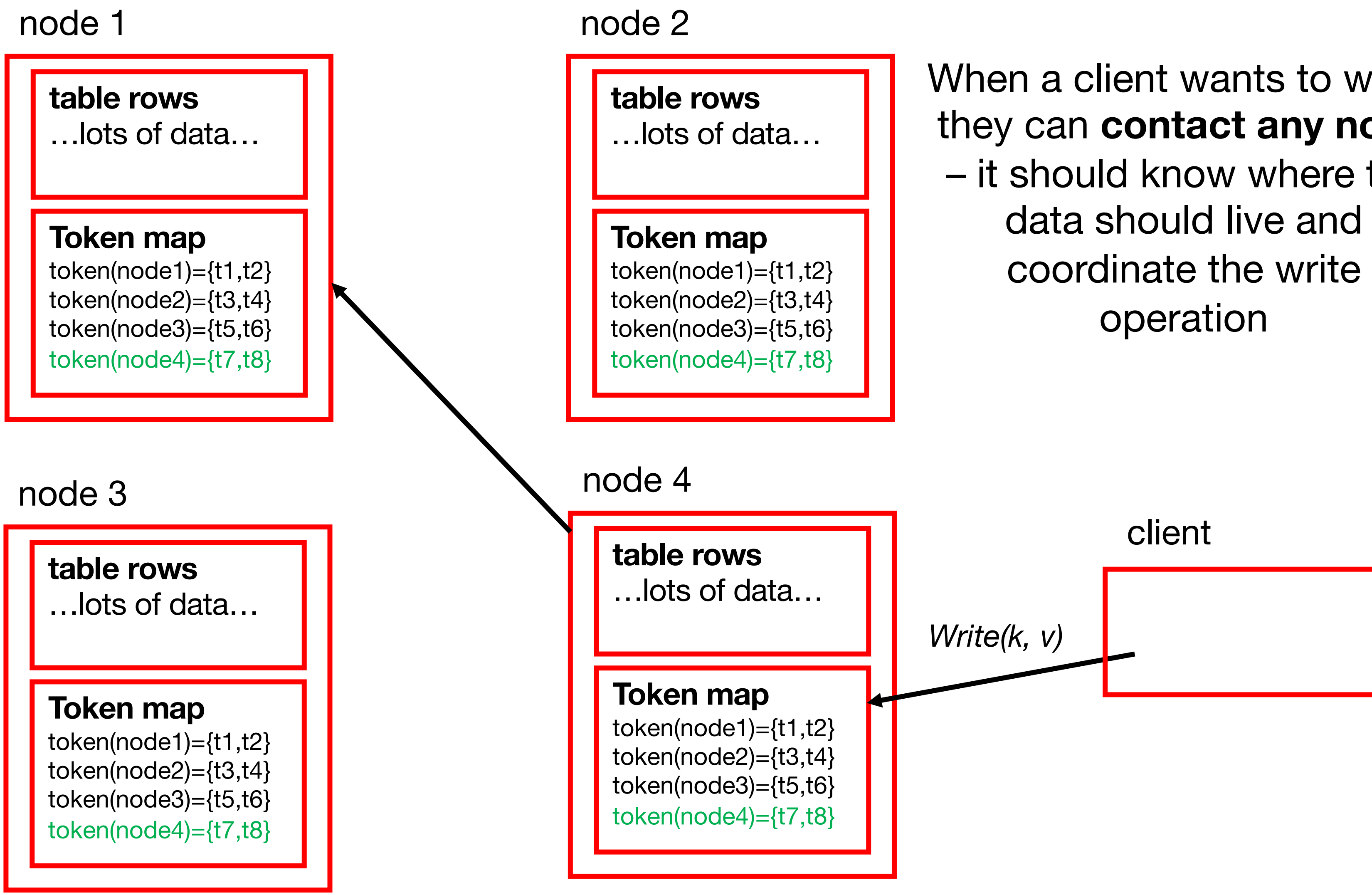
Once per second:
Choose a random friend,
Gossip about new nodes

Better approach: Gossip



Eventually, every node should find out…

Client can contact any node



When a client wants to write, they can **contact any node** – it should know where the data should live and coordinate the write operation