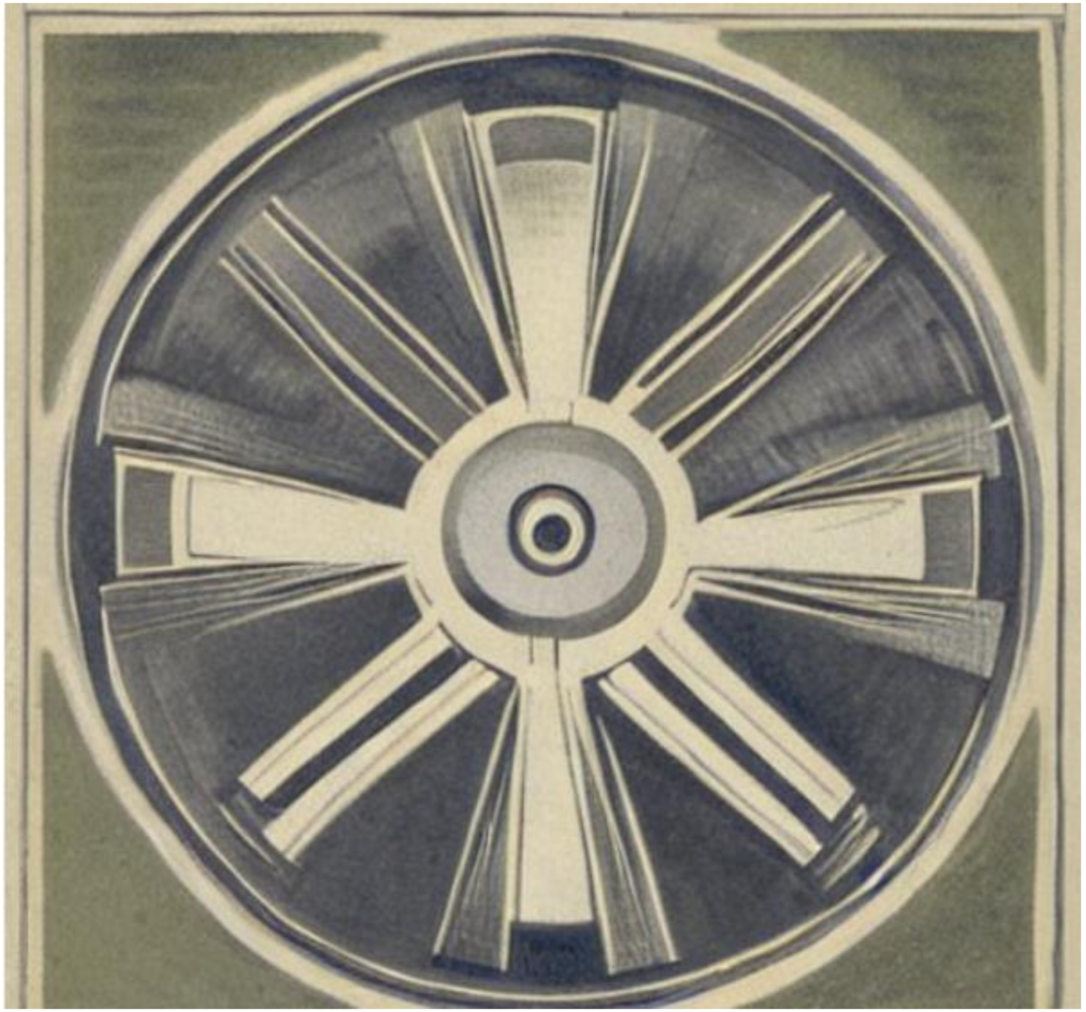
Eventual Consistency & Amazon Dynamo

“Dynamo Machine”, Natalia Sergeevna Goncharova, 1913



“electric dynamo in the style of early 20th century art”,
Stable Diffusion, November 14, 2022

Administrative

Quiz Review session:

Friday Nov 15th 4-5:30 pm at Star Room (32-D463)

Mid-term project review this week

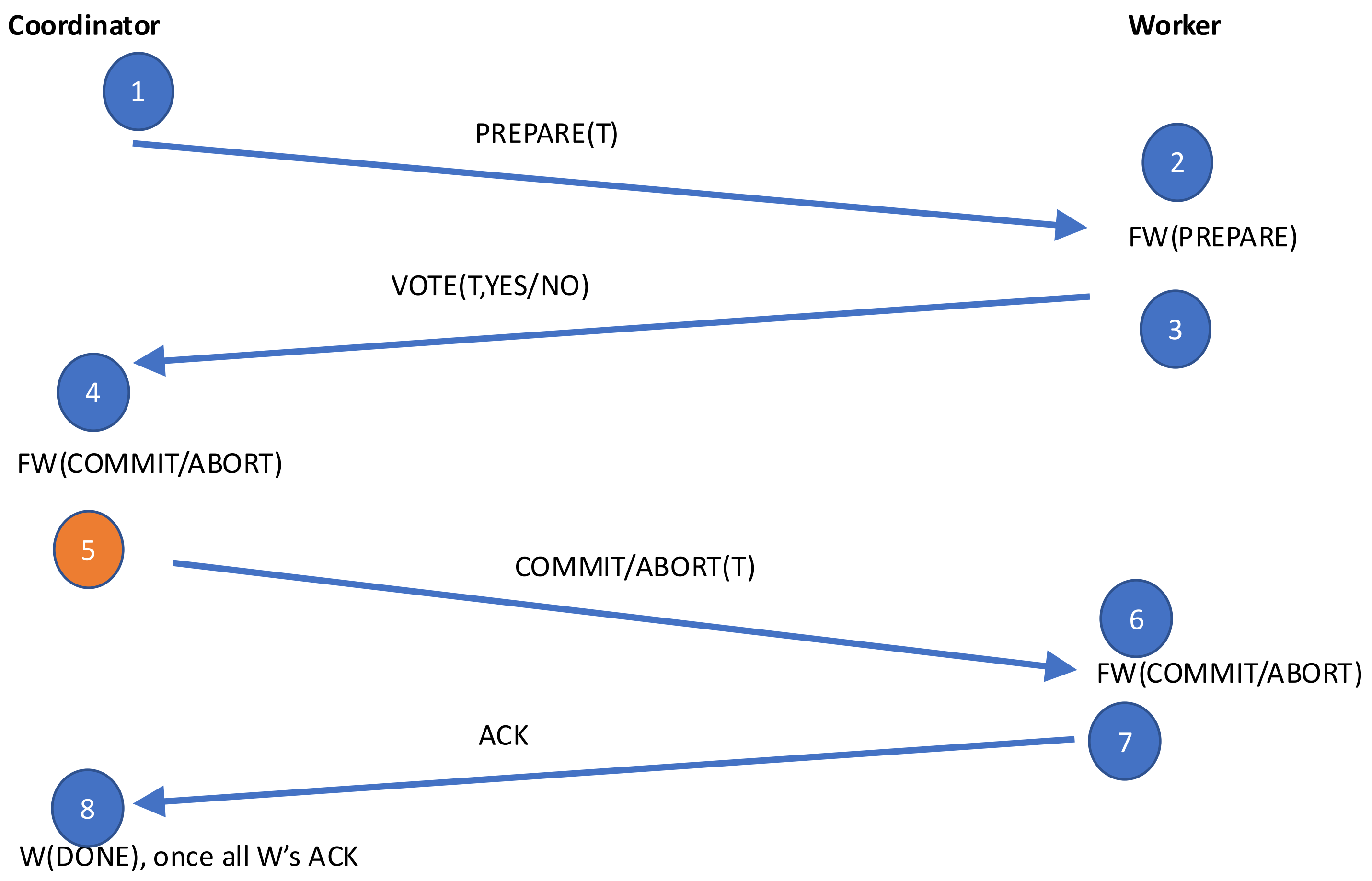
If they are planning to submit lab 3 after this Saturday, please let is know

2PC Recap

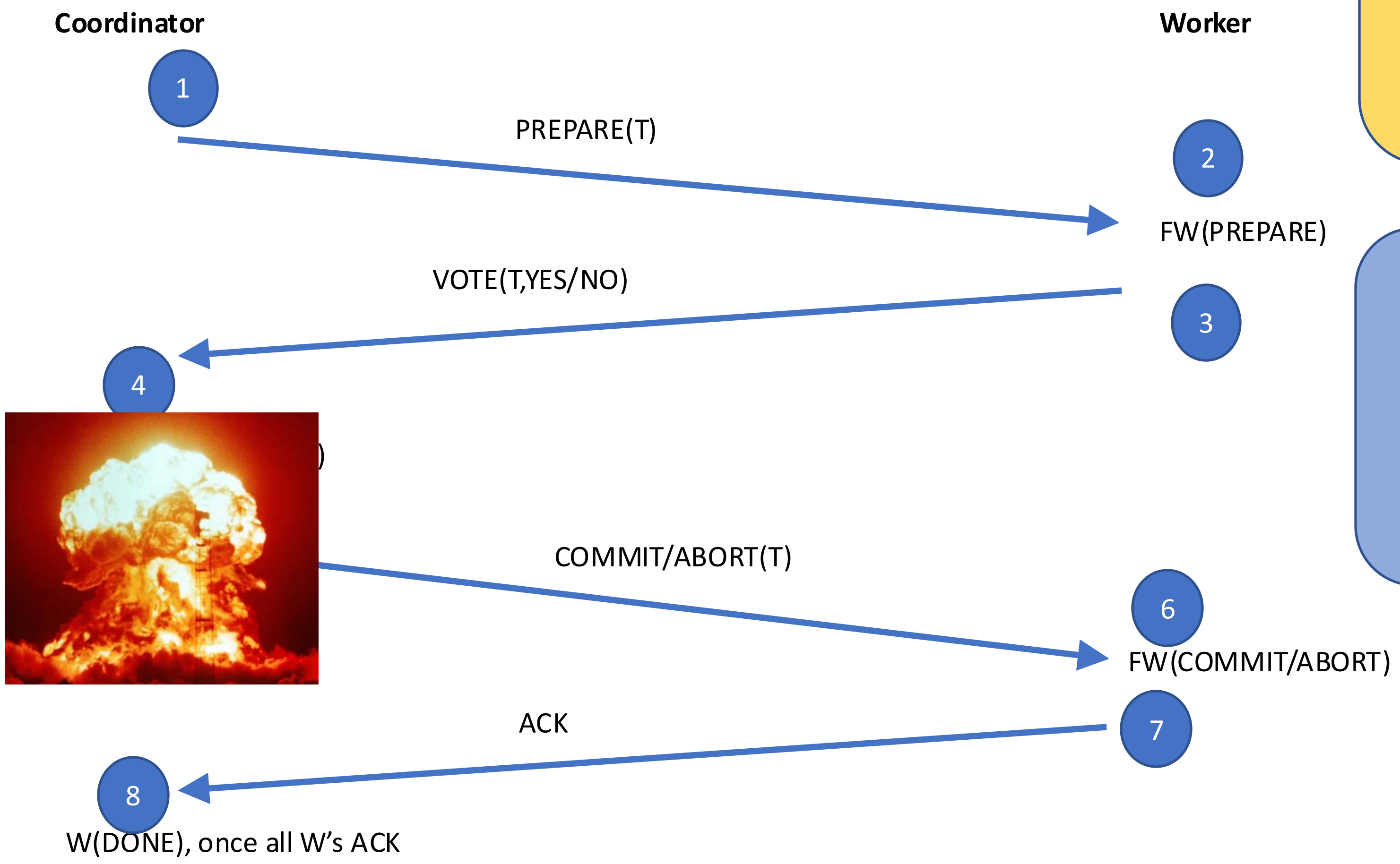
- Remember this?

- If Coord + 1 Worker fail, no way to recover
  - Coord may have told failed Worker about outcome, it may have exposed results

Failure Cases



Failure Cases



What happens if Coordinator transmits COMMIT to some workers, then dies forever?

Some workers think COMMIT took place, others can never obtain an outcome

Amazon Operational DB Desiderata

- ”Always Available” shopping cart
  - Should not go down even if a datacenter fails
  - No centralized point of failure
- Very low latency
  - Lots of orders being processed
  - Many lookups required to render a page
- No need for complex analytics
- Incrementally scalable

Enter Dynamo

- "Always Available" shopping cart
    - Data replicated across multiple nodes
    - Favor availability over consistency
- Very low latency
- No need for complex analytics
- Incrementally scalable
    - Key value store
    - CRUD semantics
    - Keys partitioned across workers using consistent hashing

Versus RDBMS

- "Always Available" shopping cart
  - Data replicated across multiple nodes
  - Favor availability over consistency

Favor consistency above all else

- Very low latency
- No need for complex analytics

Complex SQL queries can be slow

- Incrementally scalable
  - Key value store
  - CRUD semantics
  - Keys partitioned across workers using consistent hashing

Can add new nodes in shared nothing but shuffle joins may not scale incrementally

Replication Primer

- Replicating data helps with fault tolerance and performance
- Reads:
  - On a fault, reads can be directed to replica
  - Also, reads can be handled by local replica

- What about writes?
  - Slower? (More nodes to write)
  - Less available? (Have to write all nodes, what if some nodes crash?)

Availability

- Availability: can the system process requests?
- In large systems, even w/ very reliable nodes, **failures happen**!
- Replication clearly provides *read availability*
- What about writes?

Write Availability Tradeoff

- If we write to all replicas, availability is worse!
- If we only write some replicas, availability is better, but replicas can be stale
- Availability and consistency are a spectrum:

Replicas will eventually converge if updates stop

Eventual Consistency
HIGHLY AVAILABLE

Read your writes

Monotonicity (always see a view of data moving forward in time)

Strong Consistency
NOT HIGHLY AVAILABLE

1 copy serializability ("ACID")

- Many models of consistency

Write Availability Tradeoff

- If we write to all replicas, availability is worse!
- If we only write some replicas, availability is better, but replicas can be stale
- Availability and consistency are a spectrum:

- Many models of consistency

https://clicker.mit.edu/6.8530/

Consider the following 3 properties:

- **Consistency**: The data is always consistent (think serializability)
- **Availability**: The system is available as long as one replica is up and running
- **Partition tolerance**: The system can sustain network partitions

Is it possible to design a system that is (select true statements):

A) Consistent and Available?

B) Available and Partition tolerant?

C) Consistent and Partition tolerant?

D) Consistent, Available, and Partition tolerant?

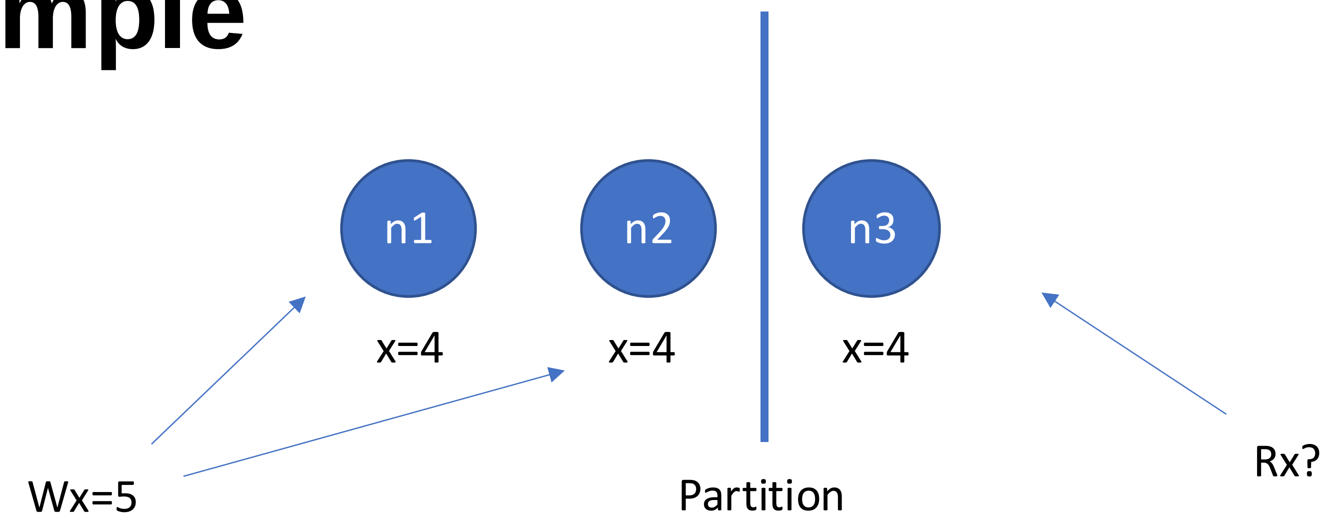
No Free Lunch

- Pick one of availability or consistency
- CAP Theorem
  - Eric Brewer at PODC 02; system can have 2 of 3 properties

    Consistency
    Availability
    Partition Tolerance

- CAP proof on systems with async communication

CAP Example



Options:

1. Wait for partition to heal (Consistent)
2. Forge ahead: n1 and n2 process write, somehow make n3 aware later? (Available)

If data is partitioned must choose either consistent or available!

NoSQL

- Class of systems like Dynamo that generally offer:
  - Key/value storage (not SQL!)
  - Partitioned and replicated by key
  - Favoring availability over consistency

Key/Value Store	Columnar *or* Extensible record	Document Store	Graph DB
Memcached	Google BigTable	CouchDB	Neo4j
Redis	HBase	MongoDB	FlockDB
Tokyo Cabinet	Cassandra	SimpleDB	InfiniteGraph
Dynamo	HyperTable	Lotus Domino	
Dynomite		Mnesia	
Riak			
Project Voldemort			

Early 2010's saw MANY such systems, with slightly different data models and semantics

Source: https://www.slideshare.net/danglbl/schemaless-databases/7

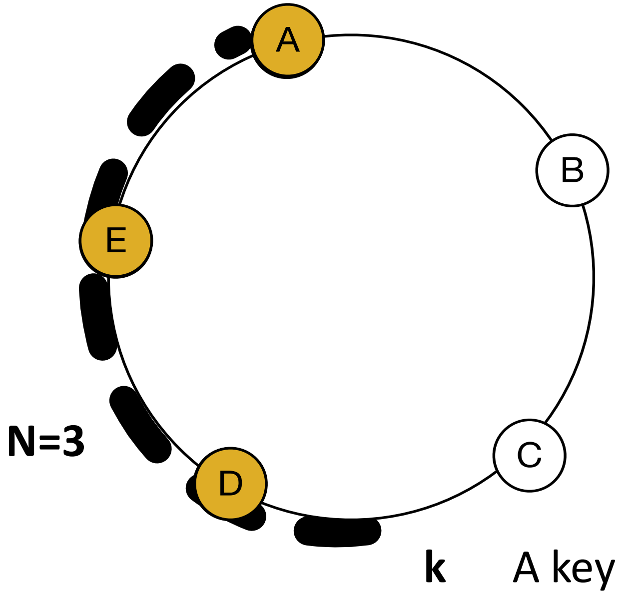
Dynamo Query Interface

- Key / Value store
- All keys and values are arbitrary byte arrays
  - md5 on key to generate ID
- get(key)
- put(key, context, value)
  - Context is a sequence number done by coordinator of write
  - More later
- single-key atomicity
  - I.e., each read/write is atomic, but only with respect to key

Dynamo Data Partitioning and Replication

- All data replicated on N nodes
- Each node has an address on a “ring” representing space of hash values from say, $0 \rightarrow 2^{128}$
- Data stored on ring as well

“Overlay network”: Nodes are not actually in a physical ring, but are just machines on the Internet

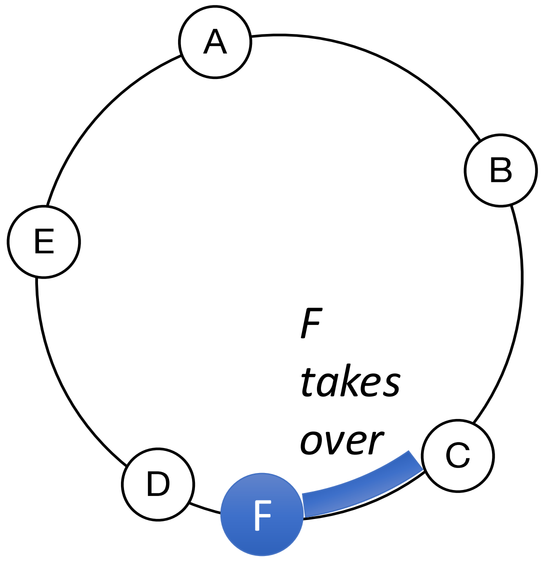


Each node occupies one (or multiple) random locations on ring

A key hashed to location k is stored on N *successors* in ring

Consistent Hashing

- Data and nodes mapped to ring
- Data assigned to nearest successor(s)
- When a node joins, it takes over only keys in range it joins
- No need to rehash all values!



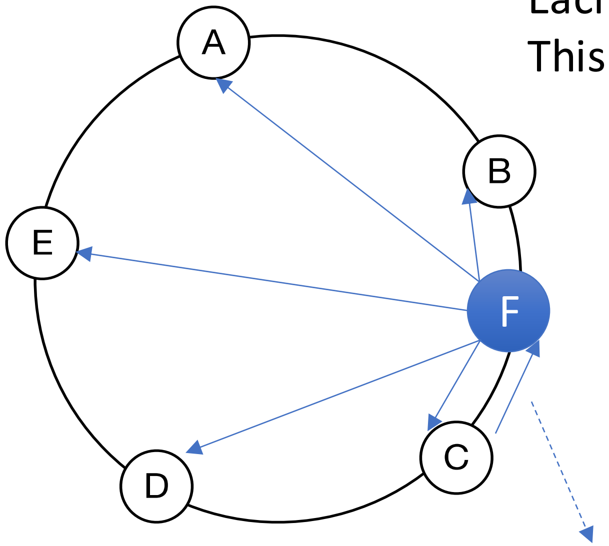
Joining the Ring

- Administrators explicitly add / remove nodes
- When a node joins, it contacts a list of “seed nodes”
  - Other nodes periodically “gossip” to learn about ring structure
- When a node i learns about new node j, i sends j any keys j is responsible for

External discovery service

Seed node list

Seed nodes are nodes clients and other nodes can ask for current mapping



Keys in range [X,W]

Each node has mapping of all other nodes. This is small, even for thousands of nodes

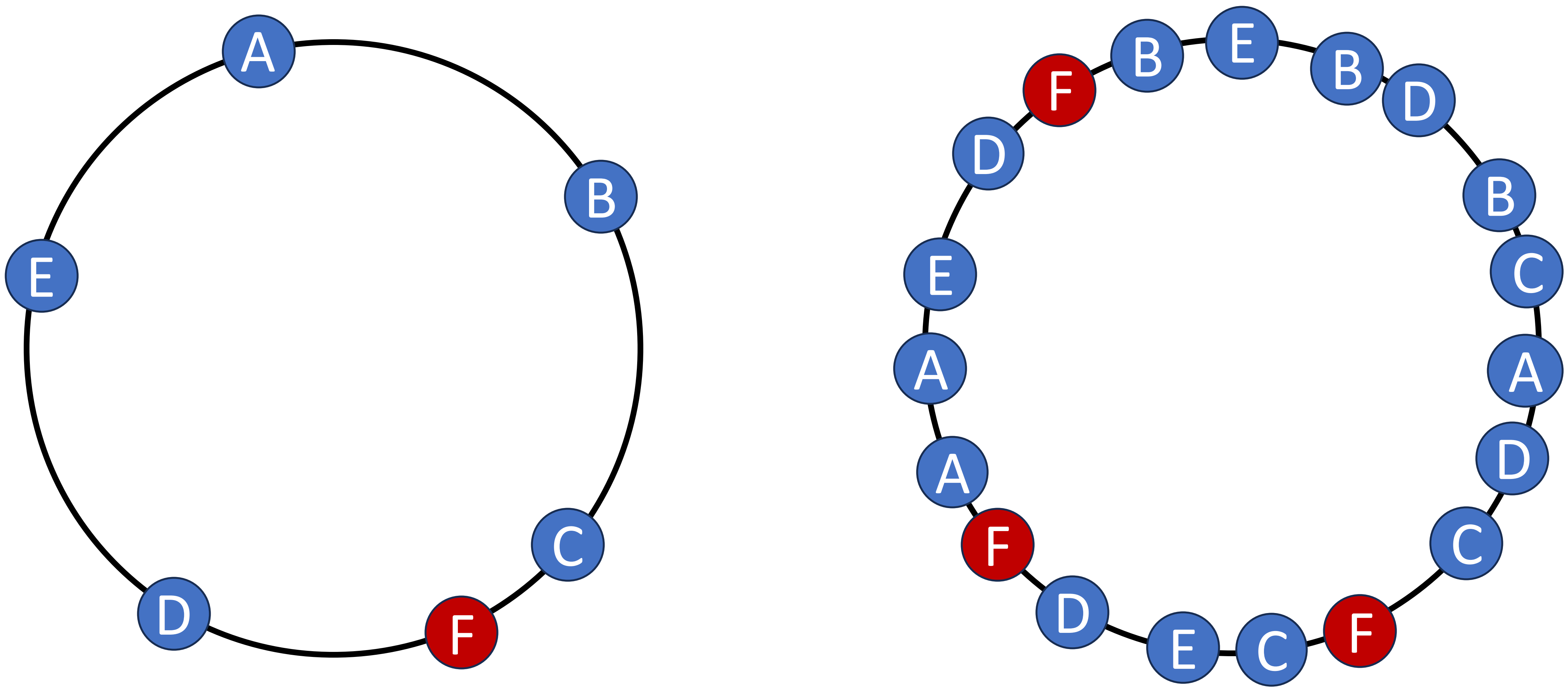
Node	Loc
A	0
B	X
C	Y
F	W

B Table

Node	Loc
A	0
B	X
C	Y
F	W

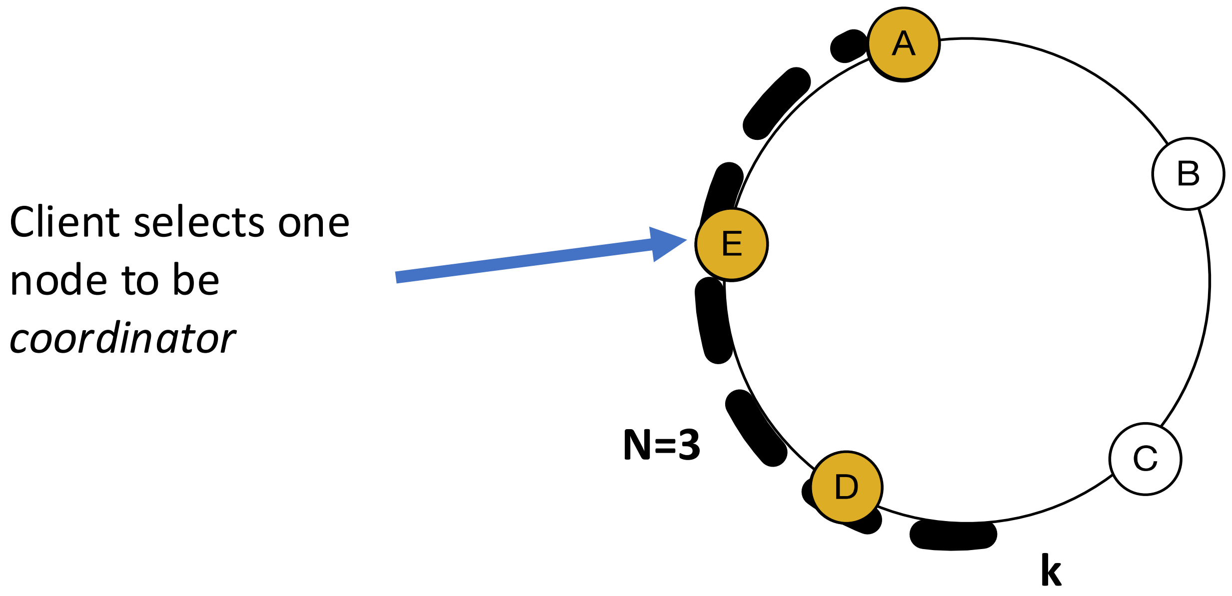
C Table

What about data skew?



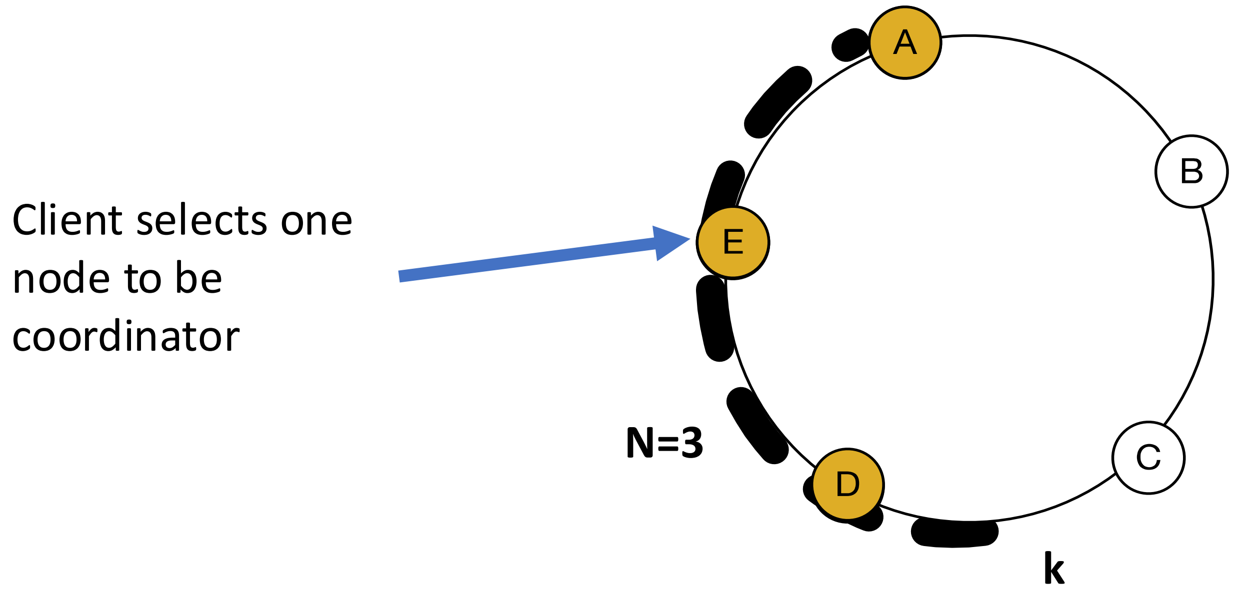
Handling Reads

- Each item is replicated on N nodes
- To read: hash key, send request to one replica
  - Client either uses Amazon front end or reads mapping table from seeds



Handling Writes

- Route as in reads
- Back to our availability conundrum
  - Do we write all replicas? What if one has failed / isn't available?
  - Do we write just one replica? How do we ensure that our read will be visible to other nodes?



Dynamo Consistency

- "Quorum Writes"
- R+W > N
  - N = number of replicas of each data item
  - R = number of replicas each read must be heard from
  - W = number of replicas each write must be sent to
- E.g., R = 2, W = 2, N = 3

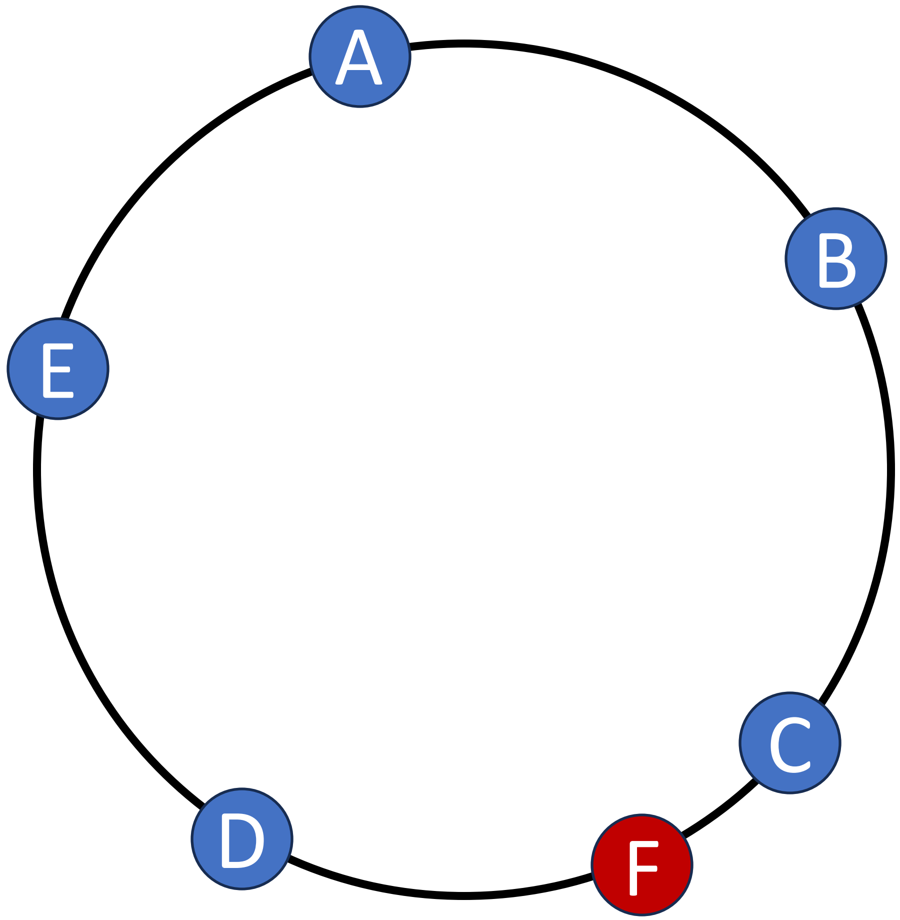
R1	R2	R3	
v1	v1	v1	
v2	v1	v2	← write to 2 out of 3

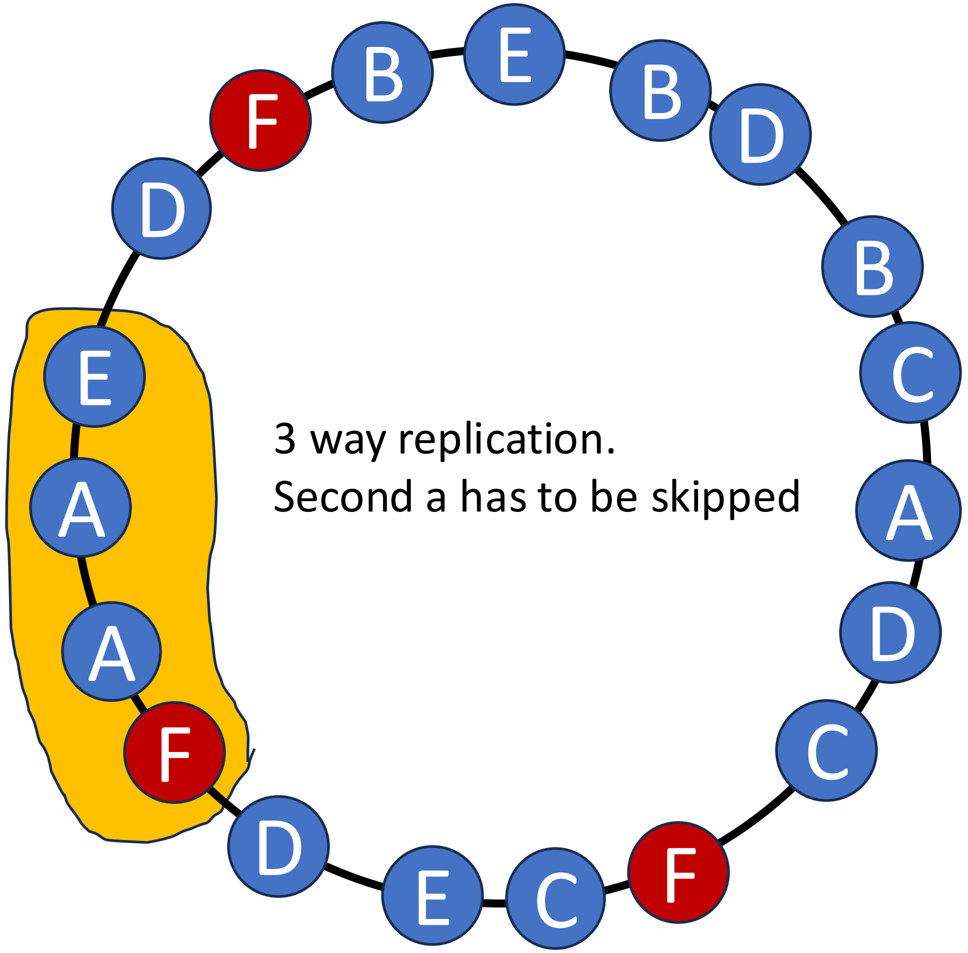
Any read of 2 will see v2!

Dynamo Consistency

- “Quorum Writes”
- R+W > N
  - N = number of replicas of each data item
  - R = number of replicas each read must be heard from
  - W = number of replicas each write must be sent to
- Need some way to ensure that if fewer than N nodes written to, write eventually propagates
  - If a reader sees that a replica has a stale version, it writes back

What about data skew?





https://clicker.mit.edu/6.8530/

Assume that we use Dynamo to store shopping cart items (e.g., (key:Tim, value:<milk, chocolate, bread>)

What statements are true with R+W > N (e.g., N=3, R=2, W=2)

1) The system is always available
2) The system can tolerate network partitions
3) The system is consistent

https://clicker.mit.edu/6.8530/

Assume that we use Dynamo to store shopping cart items (e.g., (key:Tim, value:<milk, chocolate, bread>)

What statements are true with N=3 R=1 W=2

1) The system is always available
2) The system can tolerate network partitions
3) The system is consistent

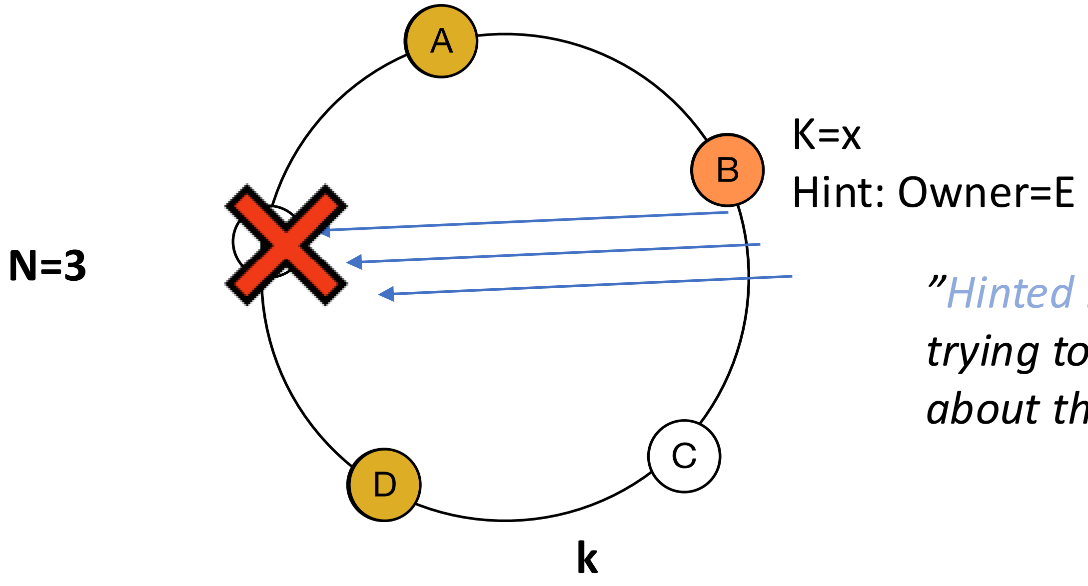
Sloppy Quorum

- Quorums still favor consistency too heavily, because:
  - Decreased durability (want to write all data at least N times)
  - Decreased availability in the case of partitioning.

- Solution: Sloppy Quorum

Sloppy Quorum & Hinted Handoff

- If fewer than N writes succeed, continue around ring, past successors



"Hinted Handoff": B will keep trying to read E to let it know about the value of K

2 our of 3 writes succeed
Continue around ring, write to B

https://clicker.mit.edu/6.8530/

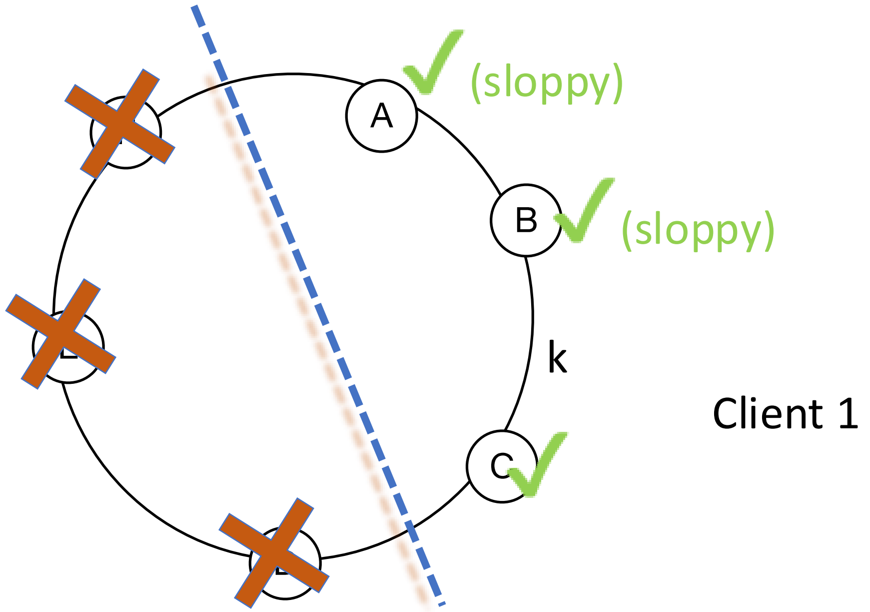
Assume that we use Dynamo to store shopping cart items (e.g., (key:Tim, value:<milk, chocolate, bread>)

With sloppy quorums is the system

1) Always available
2) Tolerant against network partitions
3) Consistent

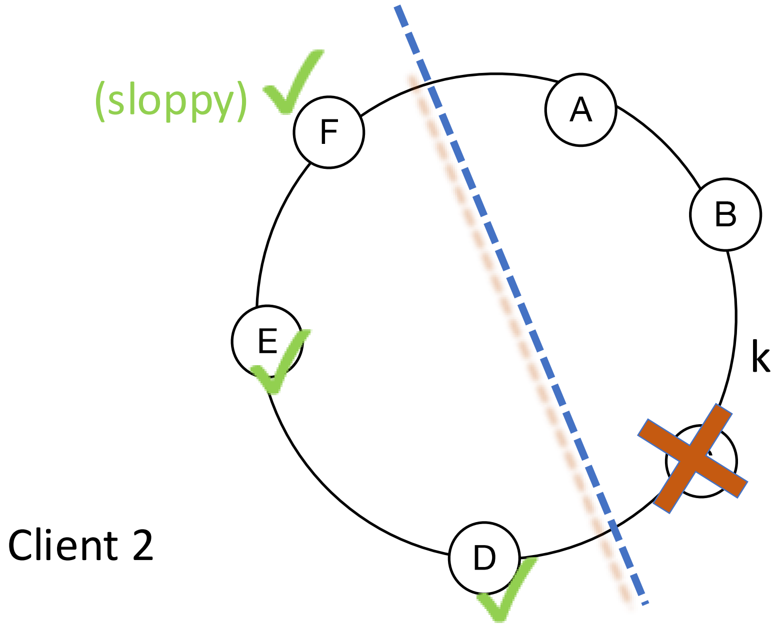
Sloppy Quorum ➔ Divergence

- If network is partitioned, hinted handoff can lead to divergent replicas
- E.g., suppose N=3, W=2, R=2, Partitioned



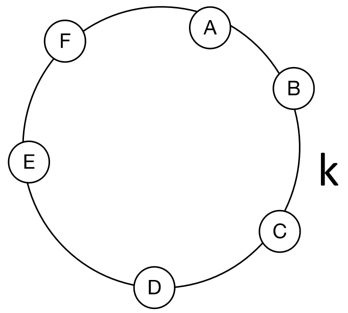
Sloppy Quorum ➔ Divergence

- If network is partitioned, hinted handoff can lead to divergent replicas
- E.g., suppose N=3, W=2, R=2, Partitioned



Two different versions of key k, k1 and k2 now exist

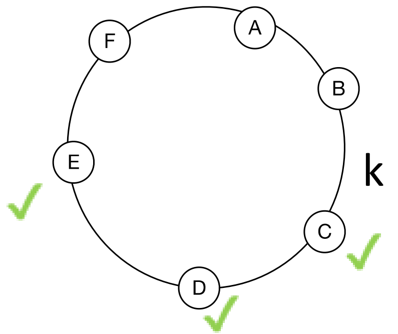
Vector Clocks



- Each node keeps a monotonic version counter that increments for every write it *coordinates*
- Each data item has a *clock*, consisting of a list of the most recent version it includes from each coordinator

A	B	C	D	E	F

Vector Clocks



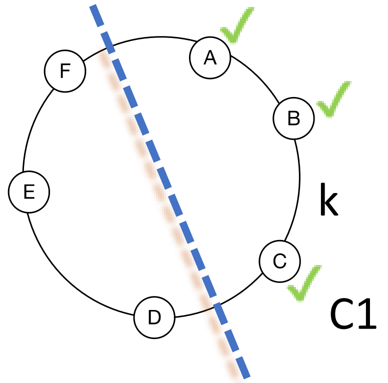
- Each node keeps a monotonic version counter that increments for every write it *coordinates*
- Each data item has a *clock*, consisting of a list of the most recent version it includes from each coordinator

<u>Client 1</u>
Create k → C
C writes [C,1] to C, D, E

A	B	C	D	E	F
		1 [C,1]	1 [C,1]	1 [C,1]	

[C,1]: Contains first version from C as coordinater

Vector Clocks



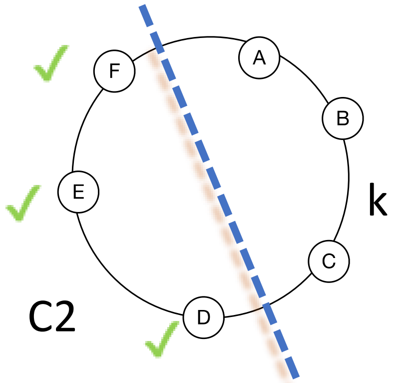
- Each node keeps a monotonic version counter that increments for every write it *coordinates*
- Each data item has a *clock*, consisting of a list of the most recent version it includes from each coordinator

<u>Client 1</u>
Create k → C
C writes [C,1] to C, D, E

<u>Client 1</u>
Read k → C
C reads C, D, E
C returns [C,1]
Write k [C, 1] → C
C writes [C,2] → C, A, B

A	B	C	D	E	F
		1 [C,1]	1 [C,1]	1 [C,1]	
2 [C,2]	2 [C,2]	2 [C,2]			

Vector Clocks



- Each node keeps a monotonic version counter that increments for every write it *coordinates*
- Each data item has a *clock*, consisting of a list of the most recent version it includes from each coordinator

<u>Client 2</u>
Read k → D
D reads D,E,F
D returns [C,1]
Write k [C, 1] → D
D writes [C,1][D,1]
to D, E, F

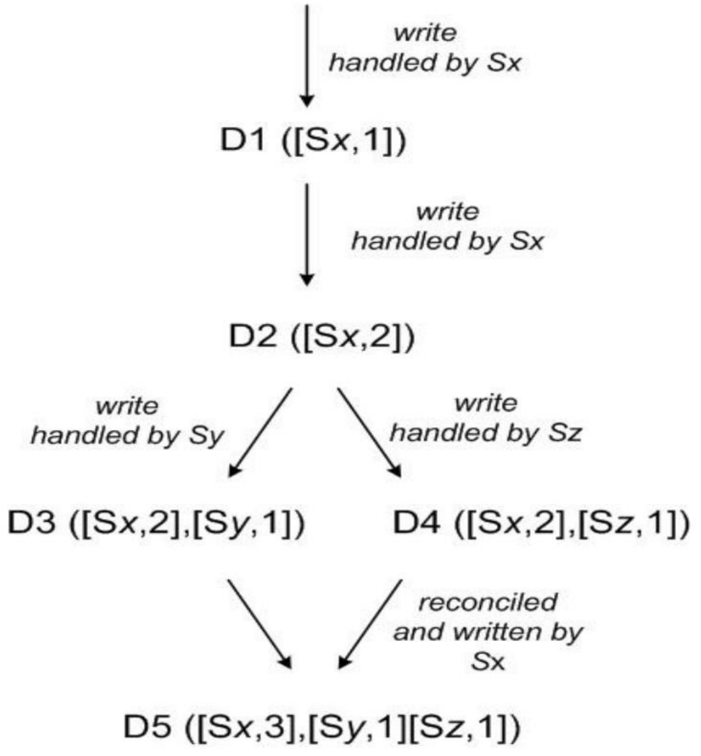
A	B	C	D	E	F
		1 [C,1]	1 [C,1]	1 [C,1]	
2 [C,2]	2 [C,2]	2 [C,2]			
			3 [C,1][D,1]	3 [C,1][D,1]	3 [C,1][D,1]

Incomparable
(can’t totally order)

Vector Clocks

Each data item associated with a list of (server, timestamp) pairs indicating its version history.

- A client writes D1 at server SX: D1 ([SX,1])
- Another client reads D1, writes back D2; also handled by SX: D2 ([SX,2]) (D1 garbage collected)
- Another client reads D2, writes back D3; handled by server SY: D3 ([SX,2], [SY,1])
- Another client reads D2, writes back D4; handled by server SZ: D4 ([SX,2], [SZ,1])
- Another client reads D3, D4: CONFLICT !



https://clicker.mit.edu/6.8530/

	Replica 1	Replica 2
A	([SX,3],[SY,6])	([SX,3],[SZ,2])
B	([SX,3])	([SX,5])
C	([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])
D	([SX,3],[SY,10])	([SX,3],[SY,6],[SZ,2])
E	([SX,3],[SY,10])	([SX,3],[SY,20],[SZ,2])

Select all versions, which are in conflict

Read Repair

- Possible for a client to read 2 incomparable versions
- Need *reconciliation*; options:
  - Latest writer wins
  - Application specific reconciliation (e.g., shopping cart union)
- After reconciliation, perform *write back*, so replicas know about new state

https://clicker.mit.edu/6.8530/

```
V1 =<R1:0,R2:3,R3:2>
V2 =<R1:1,R2:3,R3:2>
V3 =<R1:0,R2:0,R3:3>
```

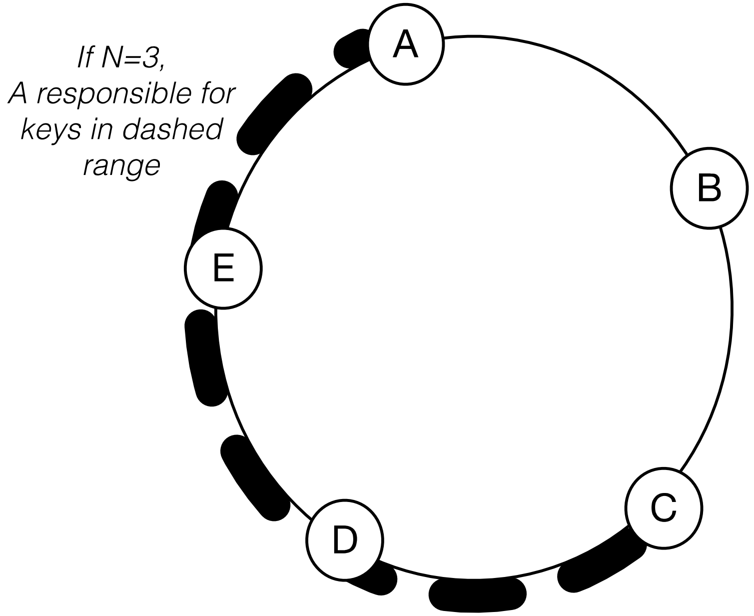
V2 was coordinated by R1, saw same versions as V1

V3 was coordinated by R3, did not see R2 1, 2, or 3, and happened concurrently with V2

1. The writer that produced V_1 observed V_2. ✗
2. The writer that produced V_2 observed V_1. ✓
3. The writer that produced V_3 observed V_1. ✗

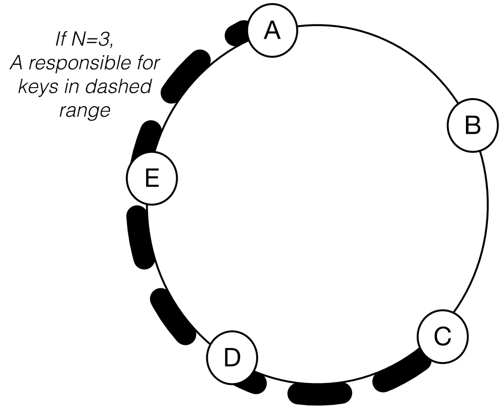
Anti-entropy

- Once a partition heals, or a node recovers, need a way to patch up
- Could rely on gossip & hinted handoff
- Dynamo also compares nodes responsible for each key range
  - Comparison done via hashing, using a technique called *Merkle trees*



Here, for EA range, B and C are also responsible

Merkle Trees



Here, for EA range, B and C are also responsible

Suppose EA range has keys u,v,w,x,y,z, A and B are comparing

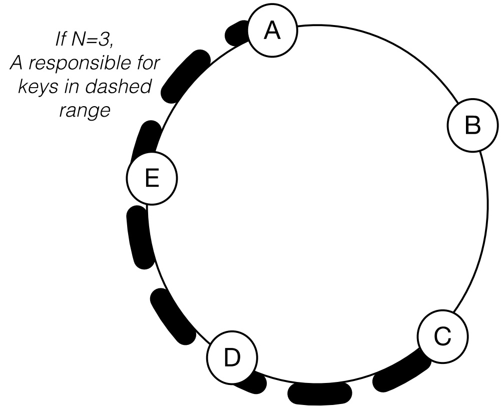
A

H(u,v,w,x,y,z) = h1

B

H(u,v,w,x,y,z) = h2

Merkle Trees



Here, for EA range, B and C are also responsible

Suppose EA range has keys u,v,w,x,y,z, A and B are comparing

A

H(u,v,w,x,y,z) = h1

H(x,y,z) = h3

H(u,v,w) = h4

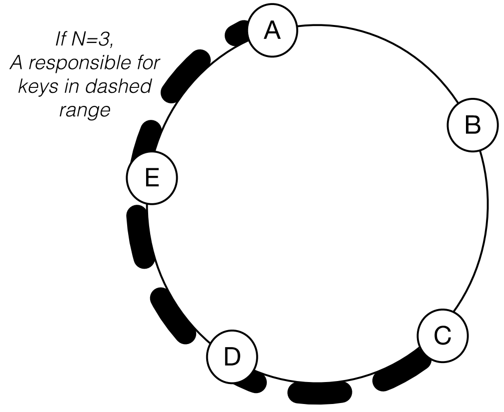
B

H(u,v,w,x,y,z) = h2

H(x,y,z) = h3

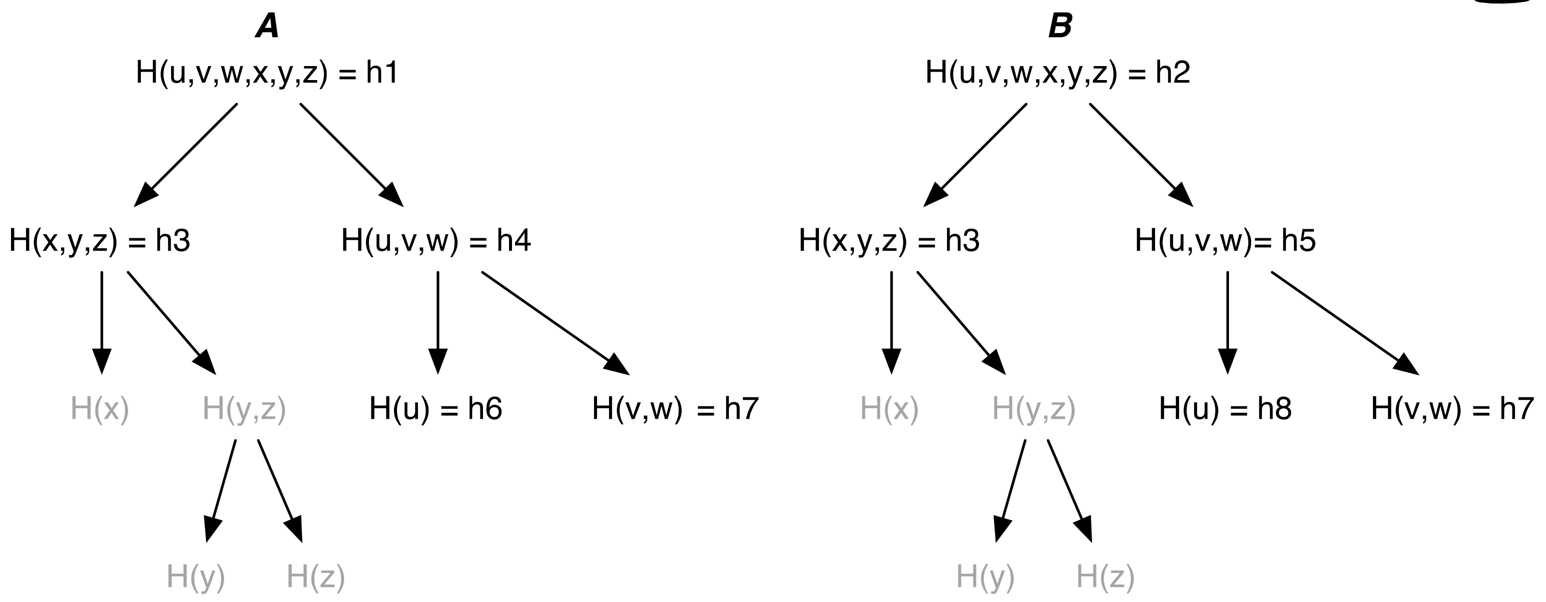
H(u,v,w)= h5

Merkle Trees

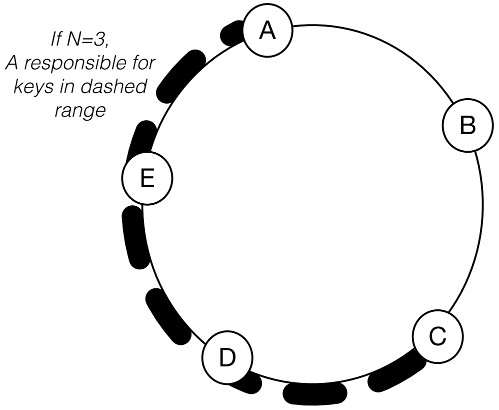


Here, for EA range, B and C are also responsible

Suppose EA range has keys u,v,w,x,y,z, A and B are comparing

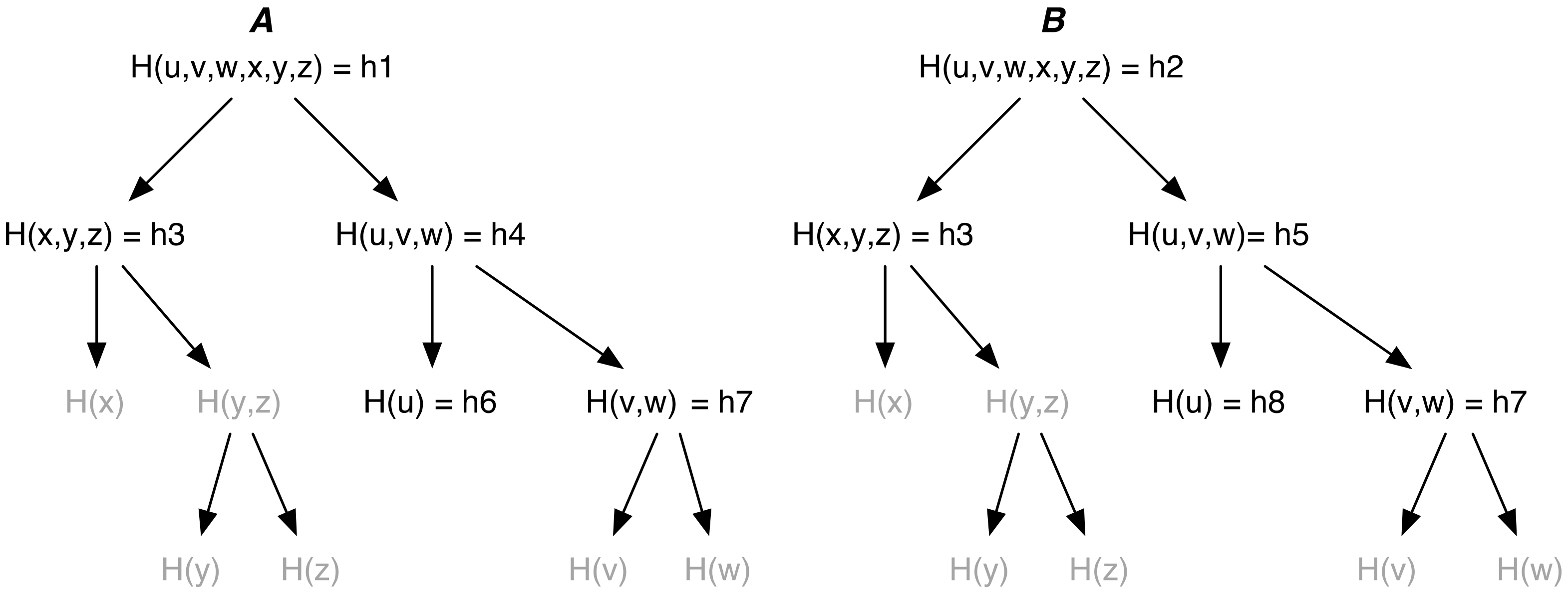


Merkle Trees

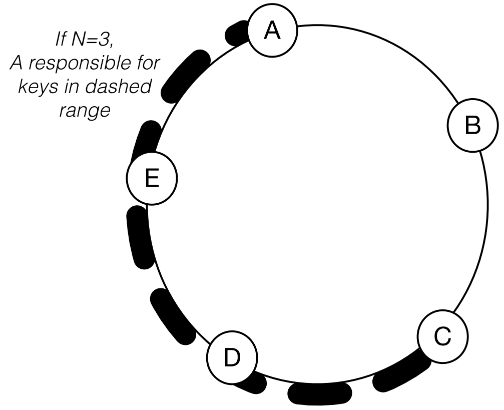


Here, for EA range, B and C are also responsible

Suppose EA range has keys u,v,w,x,y,z, A and B are comparing

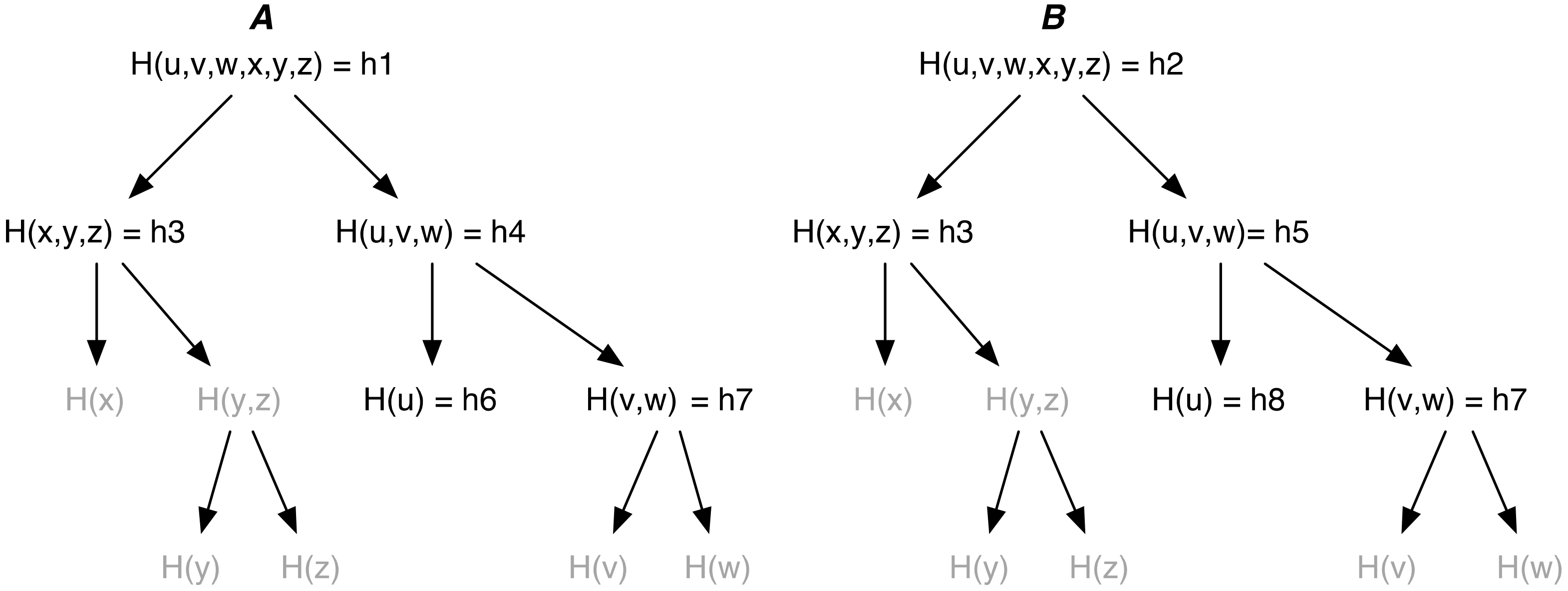


Merkle Trees



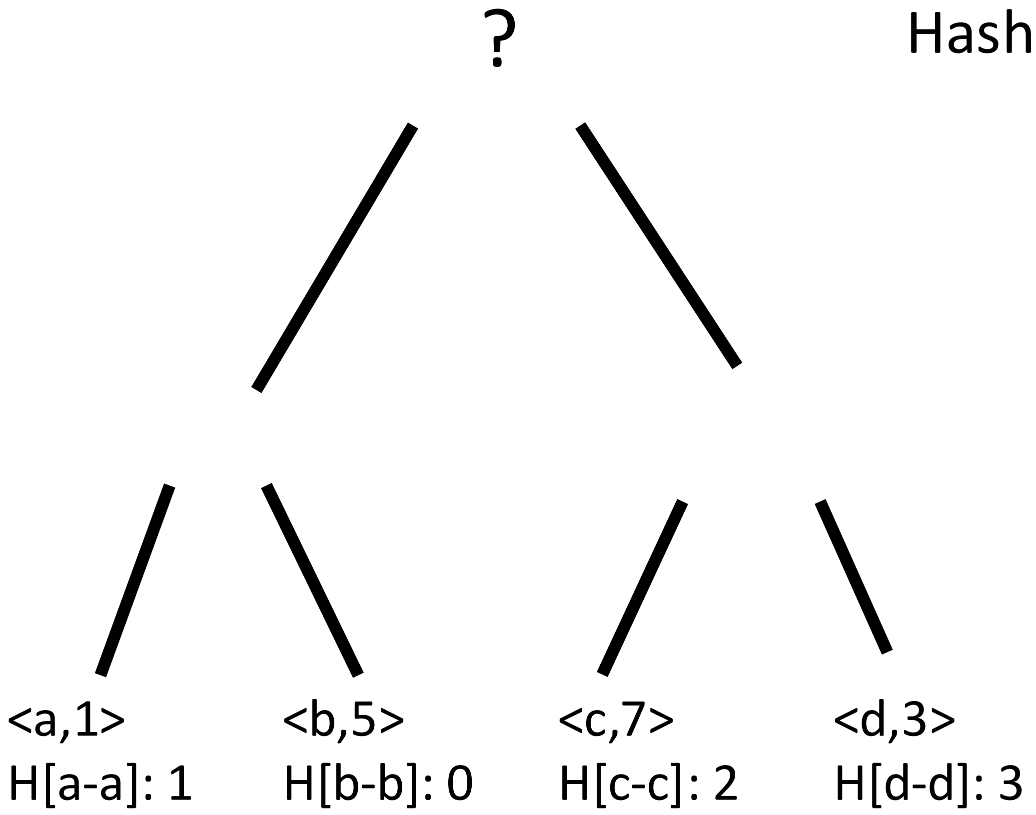
Here, for EA range, B and C are also responsible

Suppose EA range has keys u,v,w,x,y,z, A and B are comparing



This whole tree is as big as data, but only need to exchange parts of it that are different, i.e., no need to send light gray nodes in diagram, since parent hashes are all equal

https://clicker.mit.edu/6.8530/



Hash-function: x mod 5

What is the top-level hash value?

A) 0

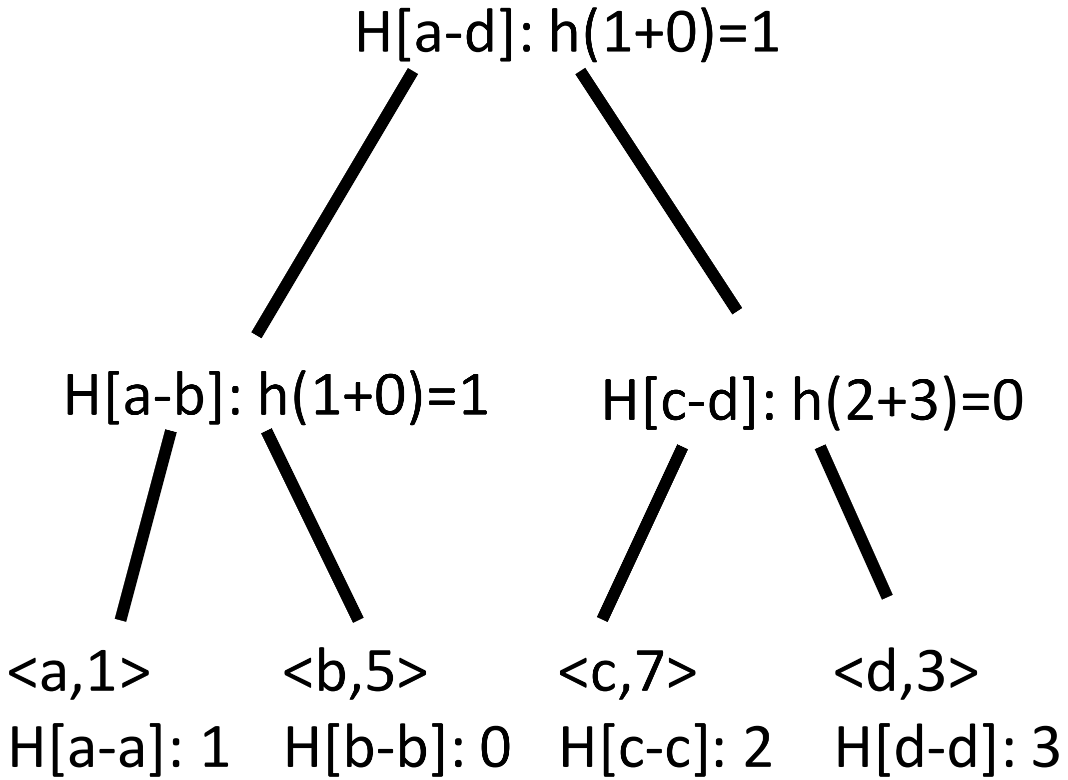
B) 1

C) 2

D) 3

E) 4

https://clicker.mit.edu/6.8530/



Hash-function: x mod 5

https://clicker.mit.edu/6.8530/

With R+W>N (read and write quorum overlap) and no sloppy quorums

What statements are true?

A) We do not need 2 phase commit anymore.
B) Single value reads are always consistent (i.e., monotonically increasing)
C) No updates can be lost

Summary

Problem	Technique	Purpose
Partitioning	Consistent hashing	Incremental scalability

Summary

Problem	Technique	Purpose
Partitioning	Consistent hashing	Incremental scalability
Highly available for writes	Vector clocks with read repair	Version size decoupled from update rate

Summary

Problem	Technique	Purpose
Partitioning	Consistent hashing	Incremental scalability
Highly available for writes	Vector clocks with read repair	Version size decoupled from update rate
Handle temporary failures	Sloppy quorum and hinted handoff	HA with some durability

Summary

Problem	Technique	Purpose
Partitioning	Consistent hashing	Incremental scalability
Highly available for writes	Vector clocks with read repair	Version size decoupled from update rate
Handle temporary failures	Sloppy quorum and hinted handoff	HA with some durability
Recovery from permanent failures	Anti-entropy	Sync replicas w/ Merkle Trees

Summary

Problem	Technique	Purpose
Partitioning	Consistent hashing	Incremental scalability
Highly available for writes	Vector clocks with read repair	Version size decoupled from update rate
Handle temporary failures	Sloppy quorum and hinted handoff	HA with some durability
Recovery from permanent failures	Anti-entropy	Sync replicas w/ Merkle Trees
Membership / failure detection	Gossip based membership	Symmetry and no centralized coordination