

Lecture 11-12

▼ Type

Lecture

ACID Principle

Databases that support the ACID characteristics are considered transactional databases.

Atomicity	Every operation in a transaction must complete, or none at all. E.g., a bank transfer must withdraw from one account and deposit into another — you can't do just one.
Consistency	The database must move from one valid state to another. E.g., if a user ID must be unique, the DBMS ensures no duplicates.
Isolation	Concurrent transactions should not affect each other's outcomes. This is achieved through locking or MVCC.
Durability	Once a transaction is committed, it must survive crashes (e.g., via WAL or disk writes).

CAP Theorem

CAP stands for:

- **Consistency (C):** Every read gets the latest write, clients have the same view of the data.
- **Availability (A):** Every request gets a response, even during failure in a reasonable amount of time.
- **Tolerance to network failure (Partition Tolerance) (P):** The system continues working even if network partitions occur.

CAP says a distributed system can achieve at most two of these properties at the same time.

Relational database lesson – ACID consistency example:

In ACID, consistency means that any transaction will bring the database from one valid state to another, maintaining constraints, integrity, and business rules.

Let's say we have a relational database with a `Teachers` table that enforces a unique constraint on the `teacher_name`.

```
--T1
INSERT INTO Teachers (teacher_name, email)
VALUES ('Dr. Smith', 'smith@school.edu');

--T2
INSERT INTO Teachers (teacher_name, email)
VALUES ('Dr. Smith', 'smith2@school.edu');
```

It will fail because the unique constraint ensures consistency. The DBMS won't allow a violation of the schema rules. Thus, this is ACID-style consistency.

CAP Consistency example:

Now we're dealing with a distributed **key-value store** like Dynamo, Riak, or Cassandra. let's say:

- Key: `teacher_name` (e.g., `'Dr. Smith'`)
- Value: all other info, like email, courses, etc.

Let's say we store:

```
"Dr. Smith": { "email": "smith@school.edu" }
```

If two clients interact with different replicas:

- Client A updates "Dr. Smith" 's email to "smith@uni.edu" on Replica 1.
- Client B reads from Replica 2 and sees the old value "smith@school.edu".

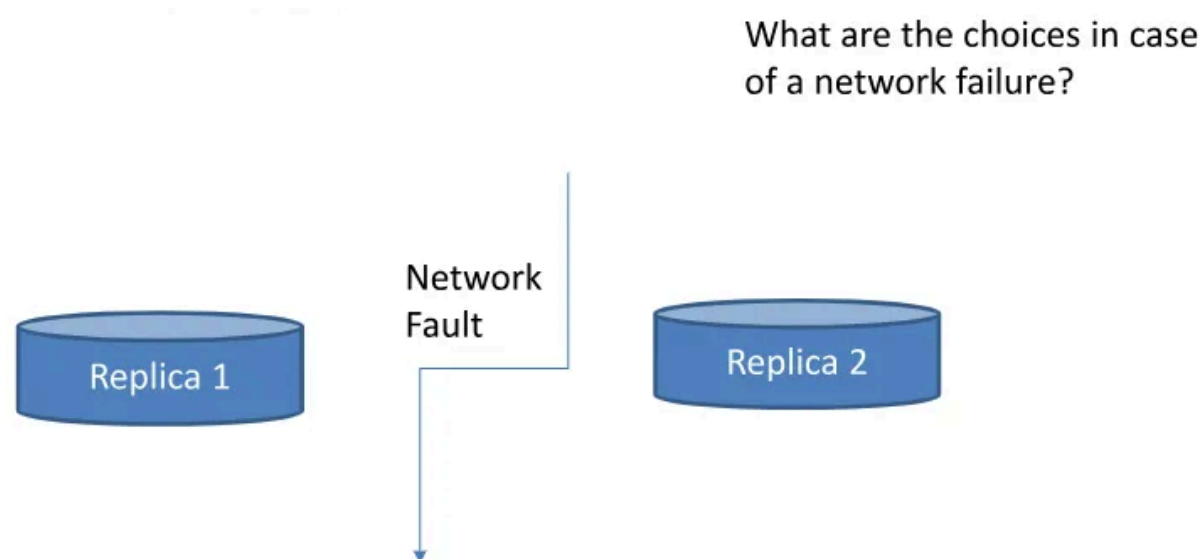
If the system prioritizes consistency (as per CAP), it will:

- Reject Client B's request until Replica 2 is updated, or
- Forward the read to a consistent replica, ensuring B gets the most recent data.

That's CAP-style consistency: all clients always see the latest data, but potentially at the cost of availability or performance.

The CAP theorem says that a distributed database system can achieve at most only two of the three characteristics. This does not mean "Pick two out of three"

- The CAP theorem says that during a network partition, a distributed system must sacrifice either availability or consistency.
- So, it's not about design-time picking two out of three.
- Rather, at runtime, when there's a partition (P), your system can be:



- **Consistency is prioritized C/P** : Consistent and Partition-tolerant, but some requests may be denied (Reject requests to maintain consistent state).
Databases: RDBMS, MongoDB, Google Spanner, HBase.
- **Availability is prioritized A/P**: Available and Partition-tolerant, but some data may be stale (eventually consistent, Accept requests but risk divergence.).
Databases: Cassandra, RIAK, DynamoDB, Voldemort.

CAP and Eventual Consistency

Linearizability --> Serializability --> Causal Consistency --> eventual consistency
strong > weak

Eventual consistency is a weaker form of consistency used in distributed databases. It means:

If no new updates are made to a given piece of data, eventually all accesses to that data will return the last updated value, even if some replicas were temporarily out of sync.

It is used for applications that:

- **Demand High Availability** (even if there are network partitions or node outages):
 - Applications must continue working even if some nodes are unreachable.
 - E.g. Amazon DynamoDB: shopping carts must accept additions even if some replicas are unreachable.

- **Require Low Latency:**
 - Prioritize speed — respond quickly instead of waiting for every replica to confirm.
 - Especially important for real-time user experiences (e.g., social media likes).
- **Can tolerate stale reads and write conflict solution:**
 - Reads may be outdated.
 - Conflicts (e.g., two users updating the same object) must be resolved later using:
 - Last-write-wins
 - Vector clocks
 - Merging strategies

If you prioritize

A and P, C (CAP) is eventually restored — that's **eventual consistency**.

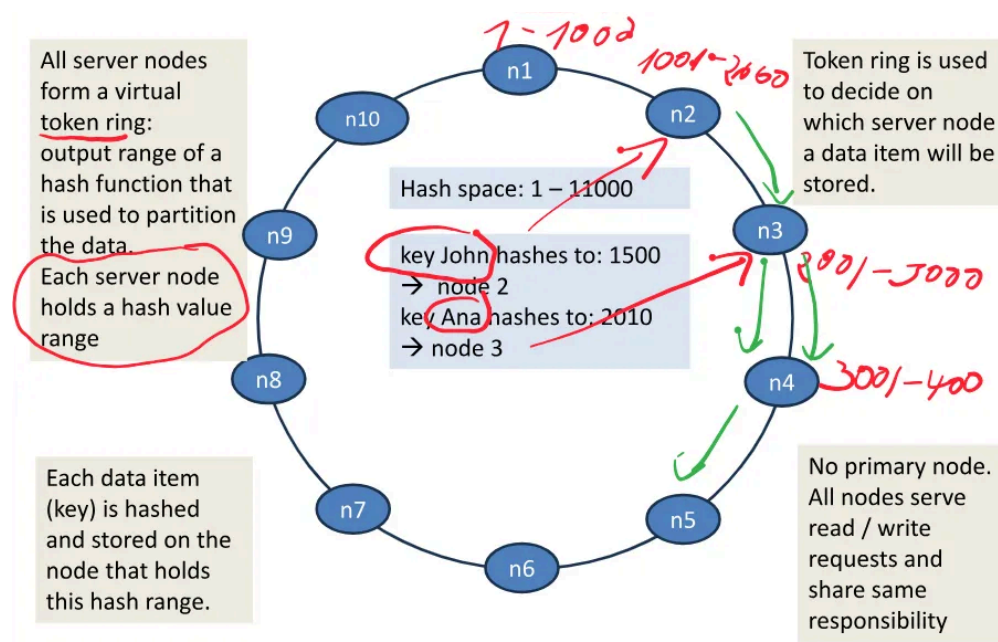
Dynamo Database as Research Project

Dynamo was a research project, it prioritized **Availability (A/P)** over **Consistency**:

- It relaxed ACID guarantees — no multi-key transactions, no isolation.
- Used vector clocks, quorums, and gossip protocols.
- DynamoDB implements many Dynamo concepts.
- Inspired systems like Cassandra and Riak.

Key Dynamo concept: Single key updates only — no joins or foreign keys, optimized for write availability.

Peer-to-Peer Distributed Databases Partitioning



- Nodes form a virtual token ring (modulo space of a hash function).
- Each node holds a portion of the hash space.
- Each key (e.g., "John") is hashed, and the key-value pair is stored on the node responsible for that hash range.

Example:

- Hash space: 1-10000
- Hash("John") = 1500 → goes to node 2

There's no master node — each node is equal.

Cassandra Peer-to-Peer Distributed Databases Partitioning

Cassandra is a peer-to-peer distributed database:

- All nodes are equal — no master.

- Each node is responsible for a portion of the hash space.
- It uses consistent hashing to assign keys to nodes.
- It uses LSM (Log-Structured Merge) Trees for storing data efficiently on disk.

Key "John" hashes to 1500, which falls into node 2's token range.

Step 1: Key hashing and partitioning:

- A partition key like "John" is hashed using a consistent hash function.
- The result (1500) determines where the data lives.
- Based on the token ring, this maps to server node 2.

Step 2: Data Routing:

- A client can contact any node in the system.
- That node becomes the coordinator for the request.
- The coordinator routes the request to the correct nodes (based on replication factor).
- In this case, it sends the data for "John" to node 2 (and in our case 2 others for replication).

Step 3: Storage in LSM Tree:

- New writes go into an in-memory structure (memtable).
- Eventually, data in the memtable is flushed to disk as SSTables.
- Periodically, SSTables are merged (compacted) for efficiency.

So, basically Token Ring decides WHO stores the data. LSM Tree decides HOW the data is stored once it gets there.

Replication Single Leader Replicated Database I

MongoDB uses single-leader (primary-secondary) replication:

- Writes go to the primary.
- Replicas pull changes from the primary's oplog.

number of replicas are defined in rs.initiate()

```
rs.initiate({
  _id: "rs1",
  members: [
    { _id: 0, host: "localhost:27018" },
    { _id: 1, host: "localhost:27019" },
    { _id: 2, host: "localhost:27020" }
  ]
});
```

Replication in Peer-to-Peer Distributed Databases

In replication, multiple copies of the same data are stored on different nodes. The purpose is:

- **High availability:** If one node fails, another can serve the data.
- **Fault tolerance:** System continues operating even if some replicas are down.
- **Read scalability:** Reads can be load-balanced across replicas.

Using a token ring architecture, replicas (copies) are NOT stored on all nodes. When creating a database key space (i.e., a namespace for your tables or buckets), one has to determine the number of replicas (copies).

RIAK:

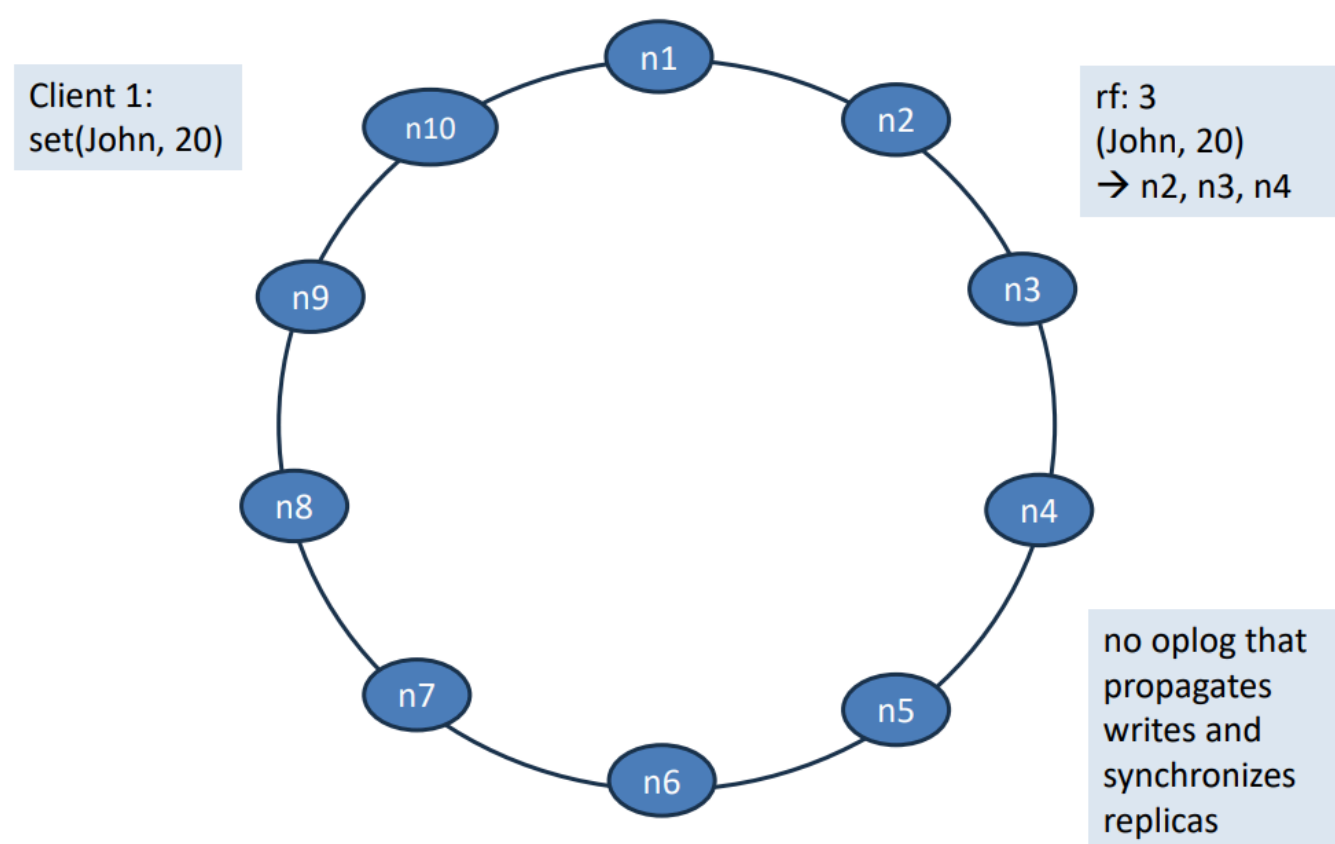
```
Bucket bucket = connection.createBucket(bucketName)
    .allow_mult(true / false)    // Allow multiple conflicting versions
```

```
.n_Val(3)           // Number of replication copies, default 3 (3 copies will be stored)
.last_write_wins(false) // default false ( Use version vectors to resolve conflicts instead of simply overwriting.)
.w(2)              // Number of nodes that must acknowledge a write (in this case 2/3)
.r(2)              // Number of nodes that must respond to a read (in this case 2/3)
.execute();
```

Cassandra:

```
CREATE KEYSPACE lesson
WITH replication = {
  'class': 'SimpleStrategy', // A basic strategy used when all nodes are in one datacenter.
  'replication_factor': 3 // The data will be stored on 3 nodes.
};
```

Peer-to-Peer Distributed Databases



- Key "John" is hashed to a numeric value (e.g., 1500).
- That value determines the responsible node, n2, via the token ring.

Since the replication factor is 3, the write must go to:

- The primary node for the key: n2
- And the next two successors on the token ring: n3 and n4

"John" → n2, n3, n4

- A coordinator node (n10) receives the client's set(John, 20) request.
- It routes the write to n2, n3, and n4.

In MongoDB:

- Writes go to the primary, then are copied to secondaries via an oplog.

In Cassandra/Riak:

- There is no primary.
- Writes are sent directly to the selected replica nodes.
- This gives higher write availability, but makes conflict resolution more complex (handled via timestamps or vector clocks).

Peer-to-Peer Replication Mechanisms

- Coordinator nodes
- Write and read quorums
- Timestamps, Vector Clocks (logical clock), Version Vectors
- Read Repair
- Merkle Trees

1. How are the writes onto the replicas coordinated?

- Since there is no primary node, coordination happens via a coordinator node.
- The node that receives the client request becomes the coordinator.
- It's responsible for:
 - Determining which replicas must receive the data (based on replication factor and hash).
 - Sending the write request to those nodes.
 - Waiting for acknowledgments from them (based on the write quorum).
 - Returning success/failure to the client.

2. When is a write/read successful?

- Write is successful: When enough replicas ($N \geq W$) confirm the write.
- Read is successful: When enough replicas ($N \geq R$) respond with the value.

3. What happens if one or more nodes are down?

- The system can still serve reads/writes if enough nodes respond to satisfy quorum.
- If not enough nodes respond:
 - The operation fails, or
 - In systems like Cassandra, the coordinator stores a hinted handoff: a temporary note to replay the write when the node comes back online.

4. What happens if nodes carry different values for the same object?

- This is called divergence.
- Example: Replica A has `John: 20`, Replica B has `John: 10`.
- This can happen due to:
 - Network partitions
 - Delayed writes
 - Concurrent updates

5. How does the system detect divergent values?

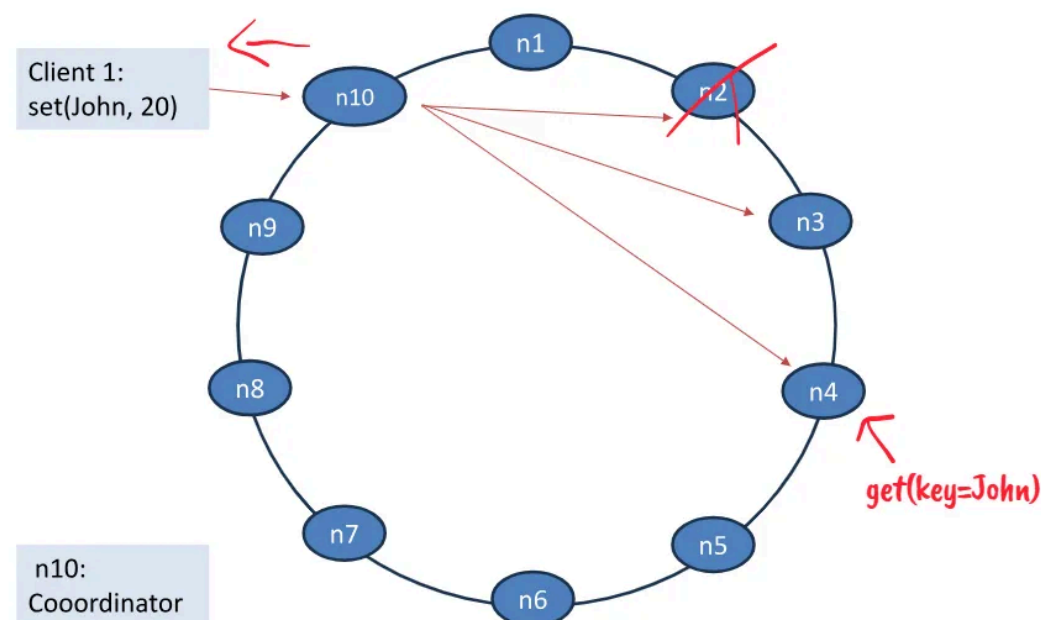
- During reads or background sync, it compares:
 - Timestamps
 - Vector clocks / version vectors
- If conflicting values exist (incomparable), the client may:
 - Get all versions (for app-side reconciliation)
 - Or rely on a conflict resolution policy (e.g., last-write-wins)

6. How are the values eventually synchronized?

- Through read repair and anti-entropy mechanisms:
 - Read repair happens during a read, if outdated values are detected.
 - Anti-entropy runs in the background to detect and resolve differences between replicas.

Distributed p2p databases typically use these concepts. Not all databases implement all concepts and implementations vary considerably

Peer-to-Peer Distributed Databases



- `n10` becomes the coordinator for this operation.
- This is a temporary role — it only applies to this request.
- The coordinator is responsible for:
 - Hashing the key `"John"` (e.g., to 1500)
 - Determining the replica nodes for this key
 - Forwarding the write to those replicas
 - Waiting for write acknowledgments from them ($W=2$).
 - Responding to the client with success/failure based on quorum logic (in this case, returns success to client 1).
- hash of `"John"` = 1500
- token ring is structured to assign 1500 to node `n2`, then with a replication factor of 3, the data will be sent to:
 - `n2` (primary replica)
 - `n3` and `n4` (next two replicas on the ring)

So `n10` sends `set(John, 20)` to `n2`, `n3`, and `n4`.

- Client 2 sends `get(John)` to node `n4`.
- Node `n4` acts as coordinator for the read, It:

- Knows that **John**'s key is stored on nodes **n2** , **n3** , **n4**
- Sends read requests to those replicas
- Waits for a read quorum (e.g., R=2)
- Compares values returned (may differ!)
- Picks the most recent version based on:
 - Timestamps, or
 - Vector clocks
- Returns the value (e.g., **"John" → 20**) to the client

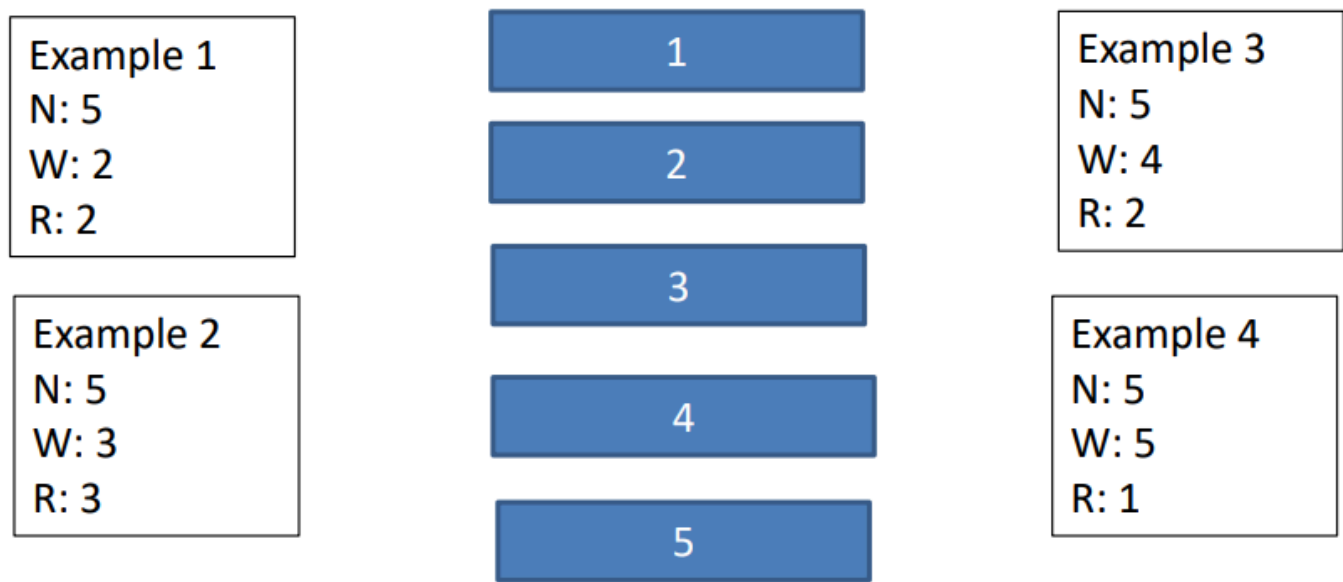
Peer-to-Peer Replication Concepts

Write and Read quorums:

Term	Description
N	Number of replicas
W	Number of replicas that must confirm a write
R	Number of replicas that must respond to a read
W + R > N	Guarantees overlap , allowing reads to see the most recent write
W + R ≤ N	Risk of stale reads
W < N ; R < N	allows the system to continue processing when nodes are down.

- write and read quorums are always given as number of nodes.
- If a write or read does not get the quorum, it is considered to be failed and the write or read operation returns an error.
- Coordinator node is responsible for acknowledgement / failure return to client.

Write and Read Quorum Replication



If $W + R > N \rightarrow$ strong consistency (at least one node overlaps between read & write sets)

If $W + R \leq N \rightarrow$ stale reads possible (no overlap guaranteed)

How many nodes can be down without interrupting availability?

Example 1: 3 nodes

You only need 2 replicas online for reads and 2 for writes.

$$5 - 2 = 3$$

Example 3: 1 node

You only need 2 replicas online for reads and 4 for writes.

$$5 - 4 = 1$$

Example 2: 2 nodes

You only need 3 replicas online for reads and 3 for writes.

$$5 - 3 = 2$$

Example 4: 0 nodes

You only need 1 replica online for reads and 5 for writes.

$$5 - 5 = 0$$

$$5 - 2 = 3$$

$$1 < 3$$

$$5 - 1 = 4$$

$$0 < 4$$

Can you get stale reads only?

Example 1: Yes

$$W + R = 2 + 2 = 4$$

Since `4 > 5` is false, there's no guarantee of overlap between the read and write sets.

Example 3: No

$$W + R = 4 + 2 = 6$$

Since `6 > 5` is true, Stale reads are avoided.

Example 2: No

$$W + R = 3 + 3 = 6$$

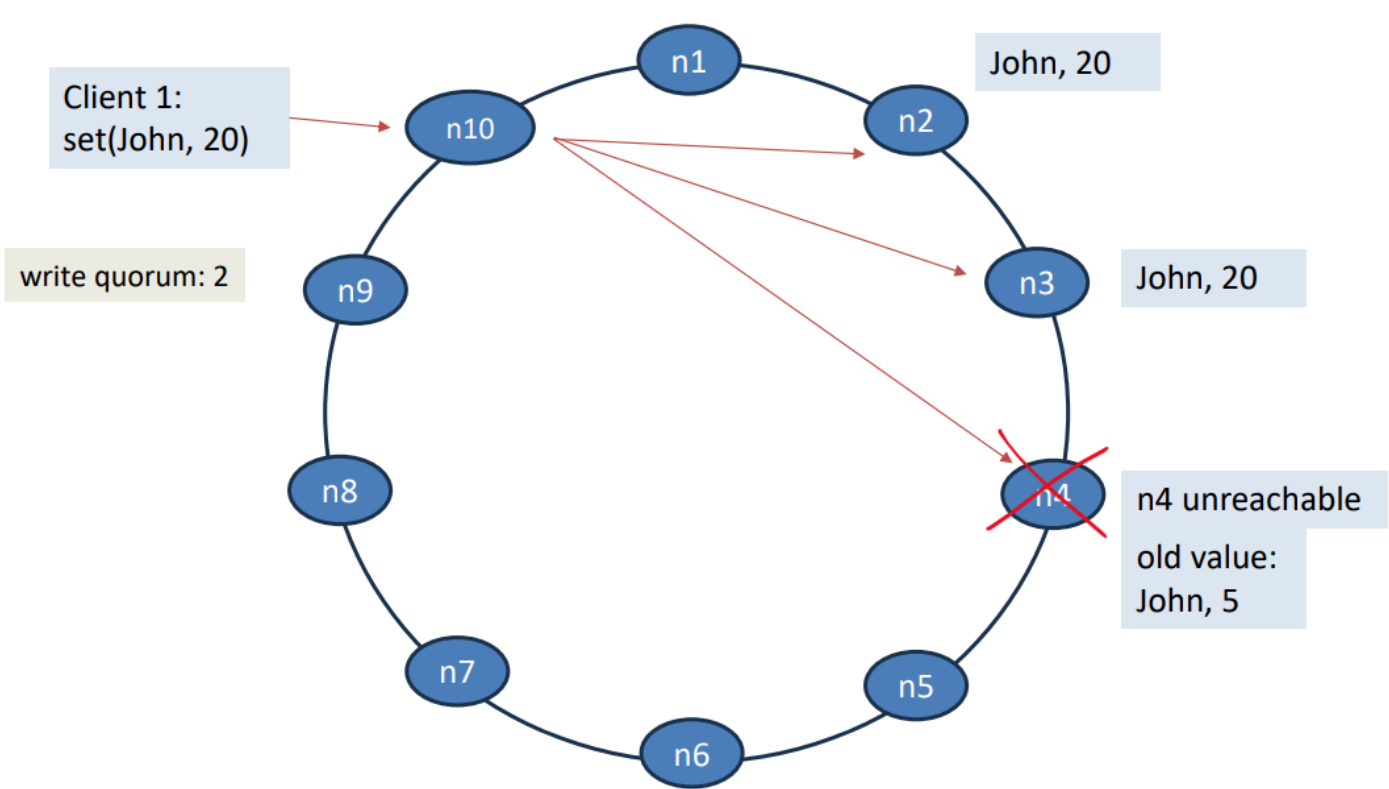
Since `6 > 5` is true, Stale reads are avoided.

Example 4: No

$$W + R = 5 + 1 = 6$$

Since `6 > 5` is true, Stale reads are avoided.

Peer-to-Peer Distributed Databases

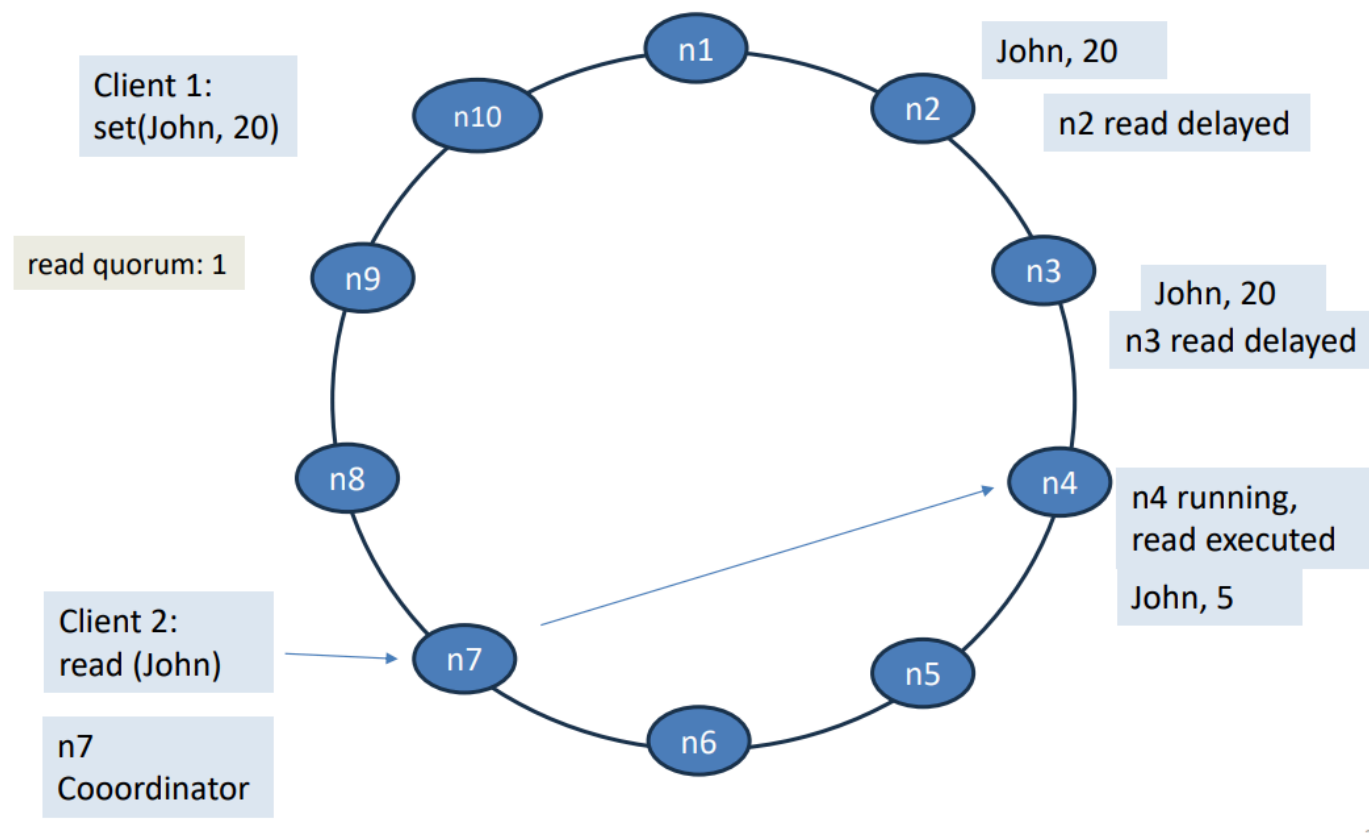


- A client sends a write: `set(John, 20)`
- Replication factor: $N = 3$, responsible replicas: `n2` , `n3` , `n4`
- Write quorum: $W = 2$, replica `n4` is unreachable
- Previous value on `n4` : `John, 5`

! Write is successful since two replicas `n2` and `n3` respond

After write:

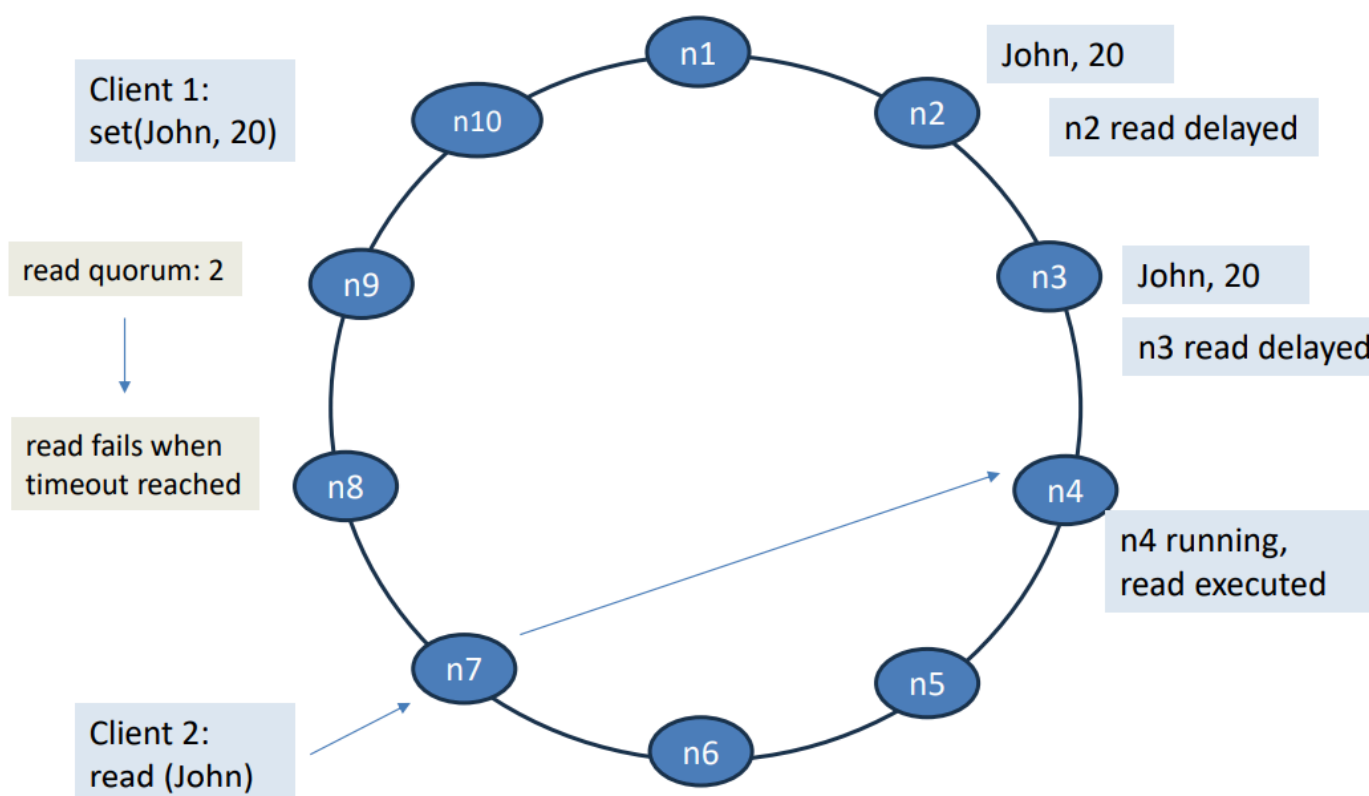
Node	Value for "John"
<code>n2</code>	<code>20</code>
<code>n3</code>	<code>20</code>
<code>n4</code>	<code>5 (stale)</code>



- Write was performed earlier: John → 20
 - n2 → has 20
 - n3 → has 20
 - n4 → still has 5 (stale, not updated)
- A new **client** sends a read request: get(John)
- Coordinator: n7
- Read quorum R = 1, meaning the read will succeed as soon as **any 1 replica** responds.

Since R = 1, the coordinator:

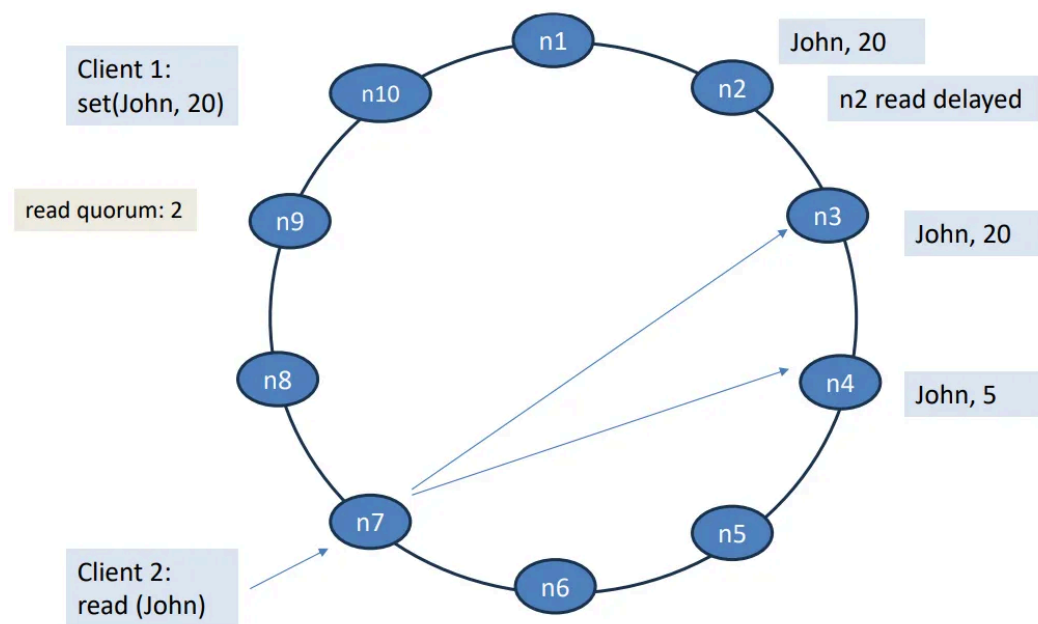
- Immediately accepts the first reply (n4 → John: 5)
- Returns John: 5 to the client



The key "John" was previously written with value 20

- Replicas: n2, n3, n4

- Client 2 (`n7`) performs: `get(John)`
- Coordinator `n7` expected 2 replies ($R = 2$)
- Received only 1 reply (`n4`)
- After waiting for a predefined timeout, it determines:
 - Read failed
 - Responds to the client with an error (e.g., "read timeout" or "unavailable")



since $R = 2$, coordinator `n7` must receive at least 2 replies to consider the read successful. This gives us the chance to detect and correct inconsistencies if replicas have diverging data.

The coordinator queries:

- `n2` → does not respond in time
- `n3` → responds with `John: 20`
- `n4` → responds quickly with `John: 5` (stale)

Now the coordinator has two replies:

- `n3` → `20`
- `n4` → `5`

Coordinator checks the timestamps or version vectors associated with each value.

- `20` is newer
- `5` is older

The correct value `John: 20` is selected and returned to the client. The client gets a fresh, consistent read despite one stale replica.

Vector Clocks / Version Vectors


In distributed databases, operations happen independently on different nodes, and network delays or partitions may reorder messages. So, we must track the order of operations logically to ensure correct behavior.

Example:

- Node1 inserts object **a**.
- Node2 reads object **a**.

If the read on Node2 happens after the insert on Node1, then it must see object **a**. But in distributed systems, "after" isn't always obvious — especially with network delays.

Some systems use physical timestamps (like Cassandra):

- When Node1 writes object , it assigns a timestamp.
- If Node2 reads later, it checks timestamps to see if it has the latest version.

But:

Issue	Explanation
Clock skew	System clocks across nodes may not be perfectly in sync
NTP drift	Network Time Protocol sync might fail or lag
Incorrect ordering	Writes can appear "in the future" or "in the past" on different nodes

Some systems work with logical clocks (vector clocks) or version vectors (RIAK, CouchDB).

Peer-to-Peer Replication: Version Vectors vs. Vector Clocks

In peer-to-peer replication, there's no single timeline. Different replicas may receive updates in different orders or experience delays. So, we need a way to:

- Know which update happened before, after, or concurrently with another.
- Decide whether a replica is stale, up-to-date, or conflicting.
- Synchronize replicas in the presence of network partitions or latency.

Version Vectors:

- Used for synchronizing replicas in distributed databases (e.g., Riak, CouchDB).
- For each object, a vector keeps a counter per replica node.
- When an update happens on a node:
 - That node increments its counter in the vector.
- When nodes synchronize, they compare version vectors to determine:
 - If one version is newer (supersedes the other)
 - Or if two versions are concurrent (conflict)

Used for replica reconciliation: keep track of updates and merge changes intelligently.

Vector Clocks:

- Designed for causal ordering of events, especially in messaging systems (e.g., distributed chat, logging, etc.).
- Helps answer: Did A happen before B?
- Tracks causality to preserve event order — so you never “see the answer before the question.”

Used more for event ordering than data synchronization.


Difference:

- Vector clocks → focus on ordering events in systems.
- Version vectors → focus on synchronizing object versions in replicated storage.

Version Vectors / Version Clocks

1. A version counter is used per each replica and per each object

Each replica maintains a counter for every object it stores. That means:

- If there are 3 replicas and 1 object , then each replica maintains its own integer counter specific to that object.

2. Each replica increments its version counter when processing a write

- When a replica updates an object:
 - It **increments its local counter** (for that object only).

- Other elements in the vector remain unchanged.

This makes the update **traceable** to the node that made it.

3. A version vector (VV) is the collection of all version counters for an object

For any object (like key "John"), the system maintains a vector like:

```
John:20 = { n2: 1, n3: 1, n4: 1 }
```

This means:

- Node `n2`, `n3` and `n4` have performed 1 update on `John`.

4. A VV has the size of the replication factor

- If `N = 3` (replication factor), then each object's version vector will always have 3 entries — one per replica.
- Even if a replica hasn't seen the object yet, its counter will be `0`.

Wikipedia Quote:

Each time a replica experiences a local update event, it increments its own version counter in the vector

- This ensures that every write leaves behind a unique trace:
 - Who did it
 - When (in logical terms)
- So you can reason about causality and version ancestry

Each time two replicas a and b synchronize, they both set the counters in their copy of the vector to the maximum of the counter:

This means:

- When two nodes share their vectors during replication, they compare:

```
V1[x] = V2[x] = max(V1[x], V2[x])
```

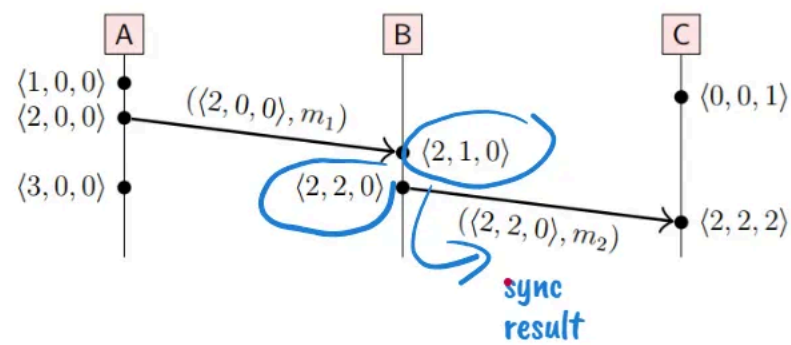
This ensures:

- Both replicas **merge** their update history.
- After sync, both have the same version vector (you can not just sync the vectors, they should be comparable.).

Vector Clocks

Vector clocks example

Assuming the vector of nodes is $N = \langle A, B, C \rangle$:



Notice the difference between vector clock and version vector: vector clock increments at each event (e.g. send, receive); version vector increments with each update.

Each event across the system is annotated with a **vector clock**: a triple like $\langle 2, 0, 0 \rangle$ representing the logical time from the perspective of each node.

Notation:

Each vector $\langle a, b, c \rangle$ means:

- $a \rightarrow$ events seen by process **A**
- $b \rightarrow$ events seen by process **B**
- $c \rightarrow$ events seen by process **C**

Vector Clocks Increment at every event:

- Local actions (e.g., computation, writing to memory)
- Sending messages
- Receiving messages (after merging clocks)

Example:

Process B sees $\langle 2, 1, 0 \rangle$ and $\langle 2, 2, 0 \rangle$ it then syncs these vectors, since they are comparable and sends the result $\langle 2, 2, 0 \rangle$ of this sync to the process C

After receiving m_2 (which carries $\langle 2, 2, 0 \rangle$):

- sync \rightarrow take max of local $\langle 0, 0, 1 \rangle$ and incoming $\langle 2, 2, 0 \rangle$
- Result: $\langle 2, 2, 1 \rangle$
- Increment own counter $\rightarrow \langle 2, 2, 2 \rangle$

Vector clocks ordering

$$T = T' \text{ iff } T[i] = T'[i] \text{ for all } i \in \{1, \dots, n\}$$

The vectors are identical in all components — they represent the same state/event (vectors after sync).

$$T \leq T' \text{ iff } T[i] \leq T'[i] \text{ for all } i \in \{1, \dots, n\}$$

T is less than or equal to T' if every element in T is less than or equal to the corresponding element in T'.

$$T < T' \text{ iff } T \leq T' \text{ and } T \neq T'$$

T happened-before T' if it is less than or equal and different — i.e., at least one element is strictly smaller.

$$T || T' \text{ iff } T \not\leq T' \text{ and } T' \not\leq T$$

T and T' are concurrent — they cannot be ordered because each vector has a greater value in different components.

vector comparison using causal sets:

$$V(a) \leq V(b) \text{ iff } (\{a\} \cup \{e|e \rightarrow a\}) \subseteq (\{b\} \cup \{e|e \rightarrow b\})$$

This means:

- Event **a** is causally before **b** if everything **a** knows about is also known by **b**.
- Vector clock **V(a)** is less than or equal to **V(b)** in this case.

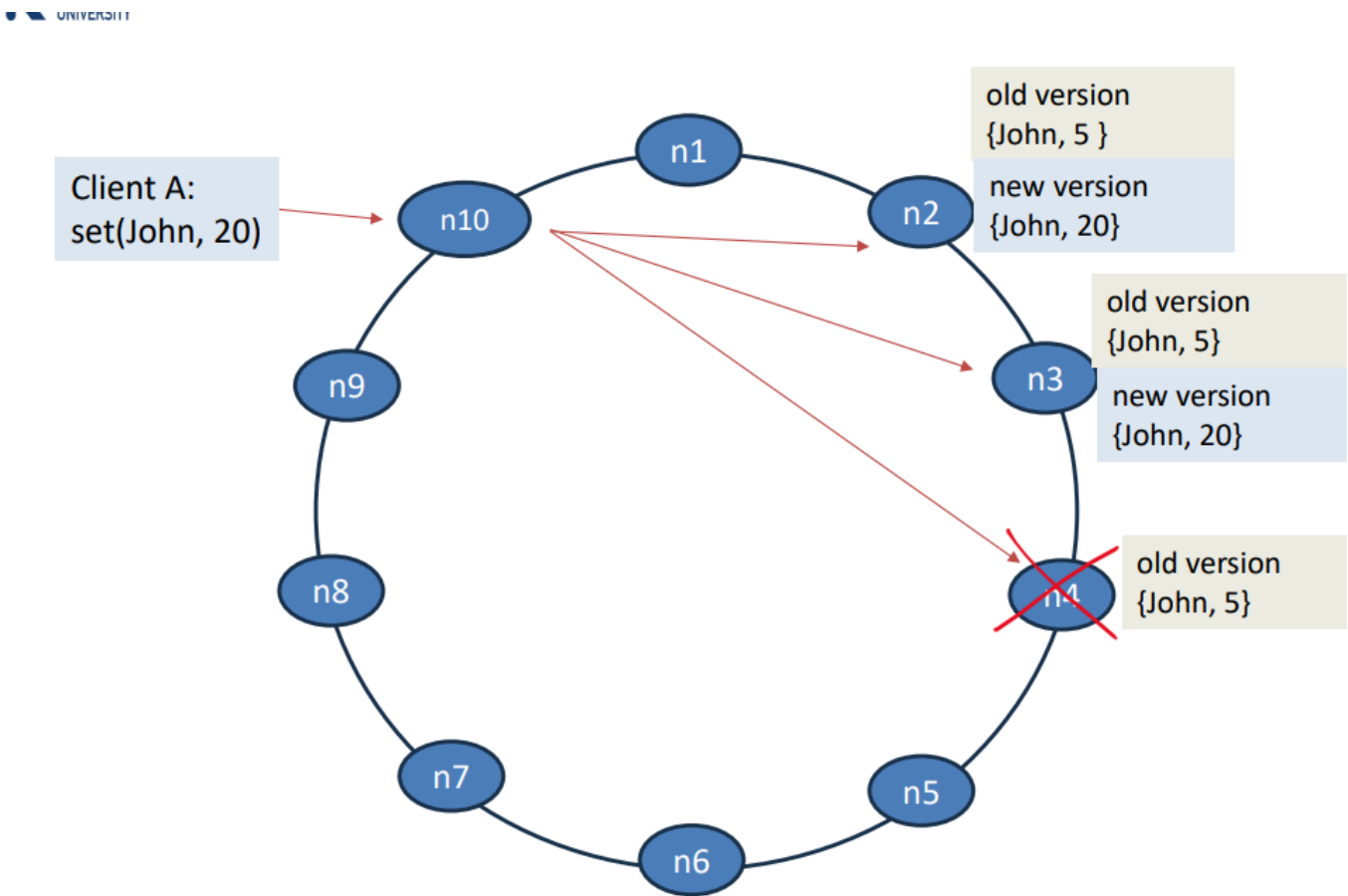
Properties of this order:

Expression	Meaning
$(V(a) < V(b)) \Leftrightarrow (a \rightarrow b)$	a happened-before b
$(V(a) = V(b)) \Leftrightarrow (a = b)$	both events are the same
$(V(a) \parallel V(b)) \Leftrightarrow (a \parallel b)$	a and b are concurrent

We say that one vector is less than or equal to another vector if every element of the first vector is less than or equal to the corresponding element of the second vector. One vector is strictly less than another vector if they are less than or equal, and if they differ in at least one element. However, two vectors are incomparable if one vector has a greater value in one element, and the other has a greater value in a different element

Write Operation -Version Vector (VV)

Here we have our token ring, where client A, coordinator node **n10** sends a write operation **set(John, 20)** to **n2**, **n3** replicates, while **n4** is down

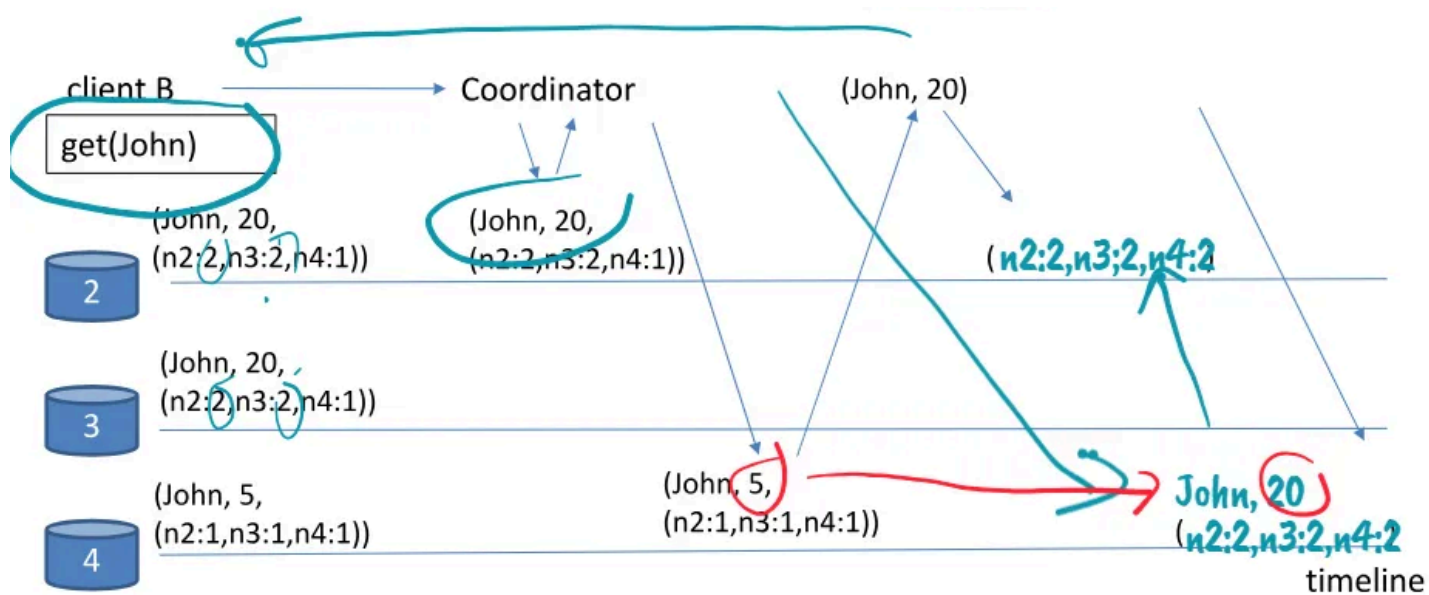


what happens to the write operation using **version vector (vv)**:

- VV_2 and VV_4 → $VV = \{n2: 2, n3: 2, n4: 1\}$
- VV_3 and VV_4 → $VV = \{n2: 2, n3: 2, n4: 1\}$

The coordinator selects the most recent version from n2/n3.

Version Vector (VV) – Read Repair



Client B requests: `get("John")`

- The coordinator queries replicas n2 and n4 and receives:
- From n2 : `John: 20` , $VV = \{n2:2, n3:2, n4:1\}$
- From n4 : `John: 5` , $VV = \{n2:1, n3:1, n4:1\}$
- coordinator compares these two vectors, $VV(n4) < VV(n2)$ so n4 is stale and n2 is newer
- Coordinator writes back the new version (`John: 20`), n4 receives this vector and updates it's own counter so vector written to is → $VV = \{n2:2, n3:2, n4:2\}$.
- the 2 nodes are synced → same VV on both nodes --> sync result is $VV = \{n2:2, n3:2, n4:2\}$.

Read Repair

Read repair is a mechanism by which a successful read also performs data repair by syncing outdated or missing replicas to the correct version.

It's a key component of eventual consistency, helping systems converge without requiring immediate write quorum across all replicas.

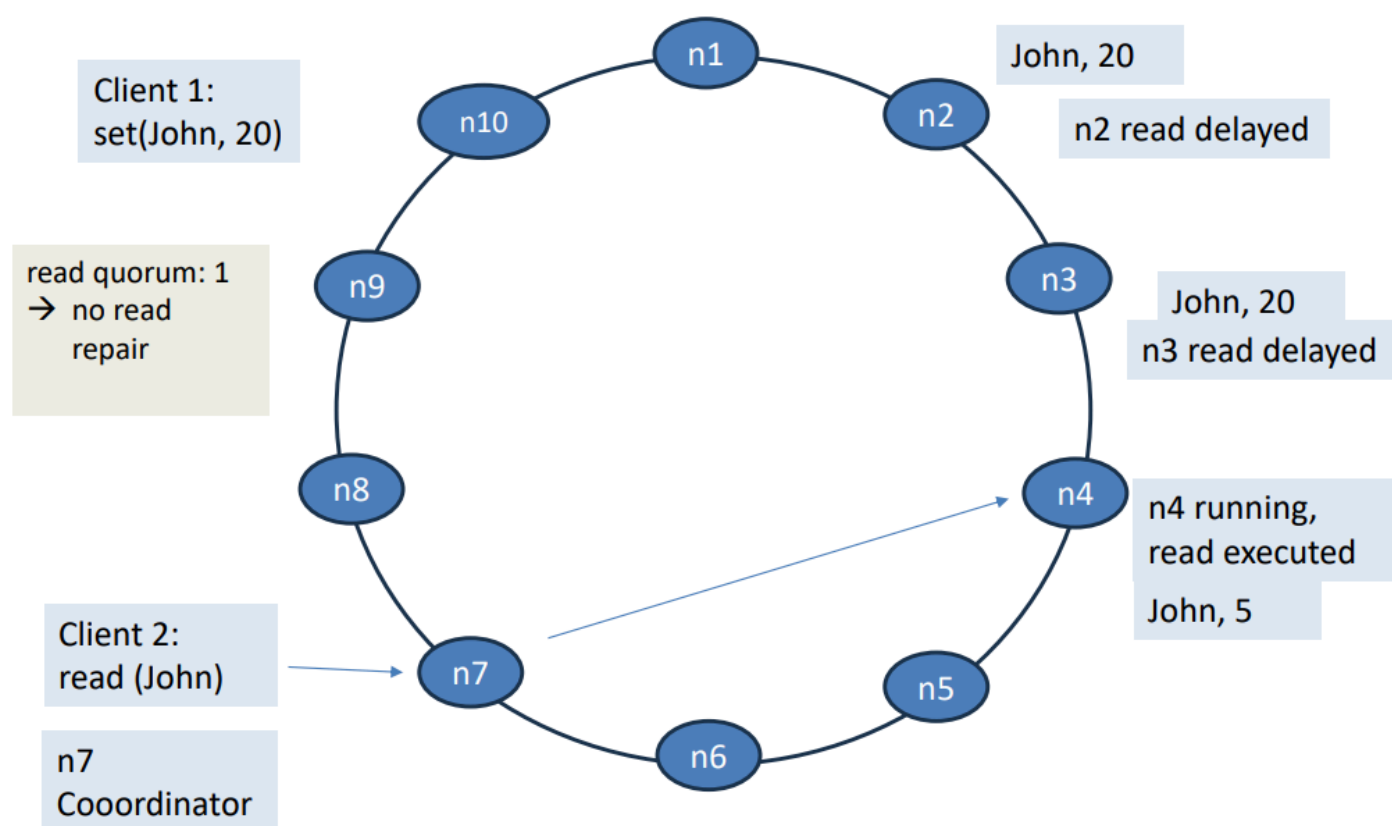
Read repair happens when:

- A read quorum is satisfied (e.g., 2 out of 3 replicas respond, if 2 nodes are down, coordinator will simply give us node 3's value, because there is nothing to compare to.)
- But some of the responding replicas:
 - Are stale (i.e., have an older version), or
 - Don't have the key at all

So to fix this The coordinator:

- Uses version vectors to detect staleness or missing values
- Returns the correct value to the client (from the freshest replica)
- Then writes that value to stale/missing replicas

So a read triggers a write to fix divergence.

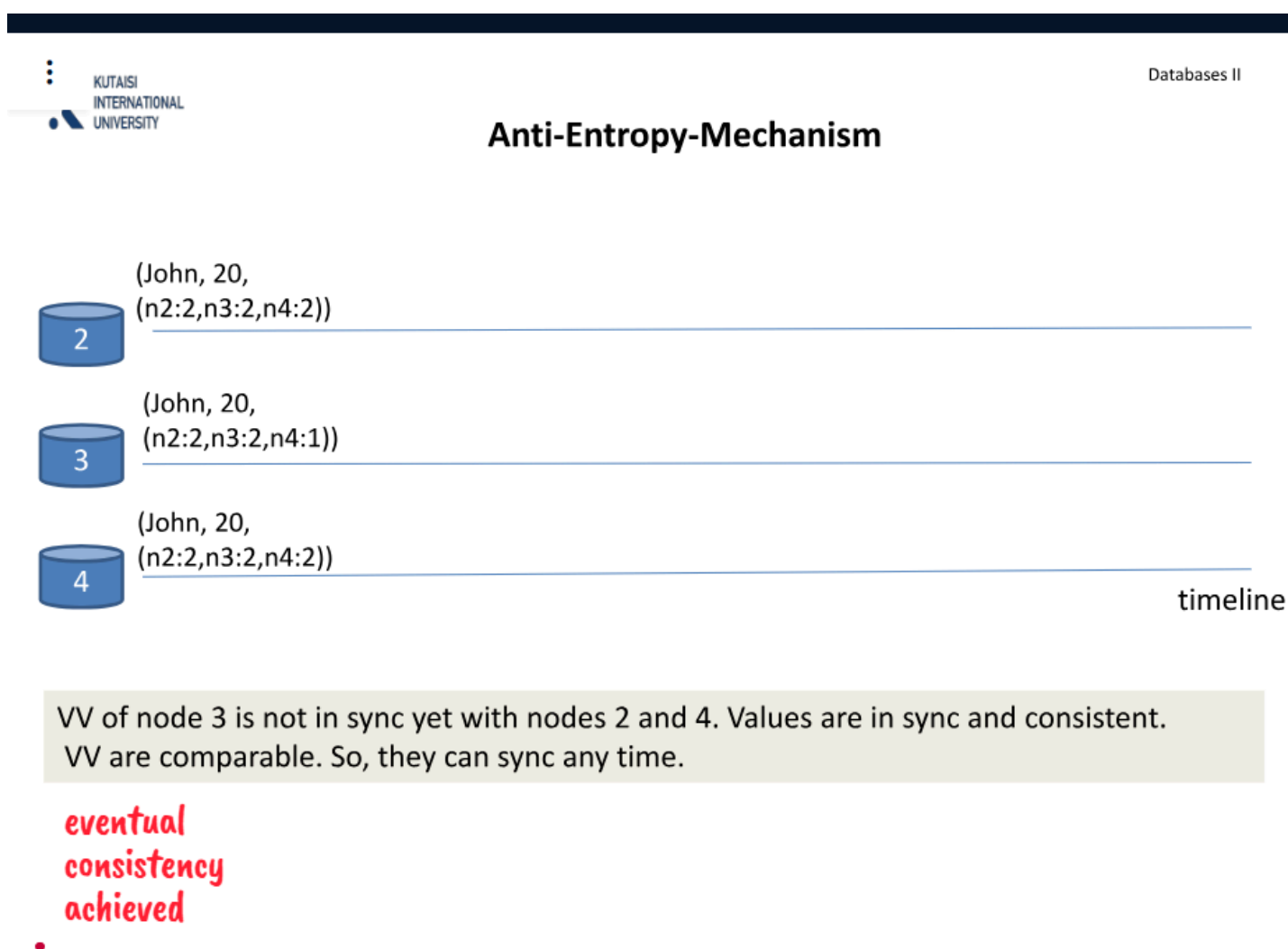


- n4's data is out of date
- But since n2 and n3 were delayed, they didn't respond in time
- The coordinator doesn't detect any divergence because it only hears from one node

Therefore:

- John: 5 is returned to the client
- No read repair is initiated, because there are no other vectors to compare to
- stale data is returned.

Anti-Entropy-Mechanism



In this case, Merkle tree-based anti-entropy will detect the mismatch:

1. Nodes periodically exchange summaries of their data trees (Merkle trees).
2. The mismatch in "John" 's hash will be noticed.
3. The newer version ({2,2,2}) will be pushed from n2/n4 → n3.
4. n3 updates its VV to match: {2,2,2}

Now all replicas are in sync, eventual consistency is achieved (in the absence of further updates, all replicas will eventually converge to the same value).

This happens in the background with no client action required.

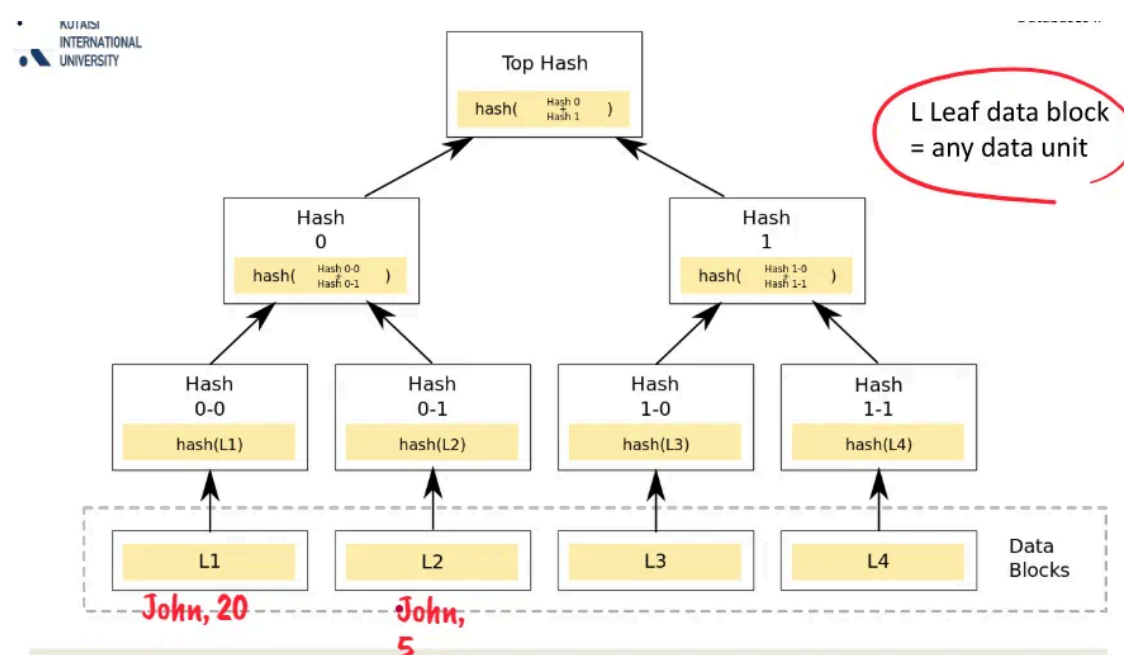
Anti-entropy is a background synchronization process in distributed systems that:

- Periodically compares replica data
- Detects divergence
- Automatically repairs inconsistent replicas

Even without reads or writes from clients.

How it works:

- Each node builds a Merkle tree (**binary tree of hashes**) over its keyspace.
 - Leaf nodes represent data blocks (key-value pairs like John → 20)
 - Internal nodes hash combinations of their children
 - The top hash summarizes the entire tree
- Nodes exchange tree roots with each other.
- If roots differ → they descend into the tree recursively to find which key ranges diverge.
- Once a mismatch is found (e.g., for "John"):
 - The nodes exchange full value + VV for that key.
 - The one with the older version vector accepts the newer value.
 - The older VV is replaced.



When replicas are Consistent:

- Two replicas (say n2 and n3) build identical Merkle trees
- Their top hashes are equal
 - So are all internal and leaf hashes
- No mismatch → No further action needed

- Just one top-level comparison
- No key-by-key scanning
- Fast and bandwidth-efficient

When replicas Diverge:

- `hash(L1) ≠ hash(L2)` , their parent hash (`Hash 0`) is also different
- This causes a top hash mismatch
- System descends into the tree:
 1. Finds which subtree is inconsistent (`Hash 0`)
 2. Drills down into children: `Hash 0-0` , `Hash 0-1`
 3. Eventually identifies specific block (like `L1 ≠ L2`)
- The node with the older value (as determined by version vector comparison) updates its data
- Read repair-like behavior, but triggered passively via anti-entropy

Without Merkle Trees	With Merkle Trees
Must compare every key-value	Compares only hashes
Expensive for millions of keys	Fast mismatch detection
Wastes bandwidth on identical data	Only syncs divergent blocks
Not scalable	Used in Riak, Cassandra, DynamoDB

Read Repair and Anti-Entropy Process

The read repair mechanism can only identify and update stale values (and sync VV) if the data is queried. Data, that is not queried, does not get updated.

That is why, many systems have additional Anti-Entropy processes running in the background. These processes constantly check whether replicas differ in data.

In principle, the anti-entropy processes have to compare data units (records, key-values pairs or collections, documents, sets, ..) of any two replicas pair-wise. As this is a cost-intensive process, Merkle trees are often used to facilitate the anti-entropy process.

Entropy: describes the state of divergence between the replicas. Is is thus not desirable.

One advantage of using read repair alone is that it doesn't require any kind of background process to take effect, which can cut down on CPU resource usage. The drawback is that the healing process can only ever reach those objects that are read by clients. Any conflicts in objects that are not read by clients will go undetected.

The active anti-entropy (AAE) subsystem runs as a continuous background process

In order to compare object values between replicas without using more resources than necessary, Riak relies on Merkle tree hash exchanges between nodes.