A13

Type	Homework

CAP Theorem

Relational database lesson – provide an ACID consistency example:

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.

Key-Value database "Teachers"
key: teacher_name
value: information about the teacher

Provide a 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.

Rollback Behavior in MongoDB

In what scenario can a rollback happen in MongoDB?

- A primary node accepts writes.
- Then, due to a network partition or failure, this primary steps down (e.g., because it cannot reach the majority of nodes).
- A new primary is elected, and replication resumes from it.

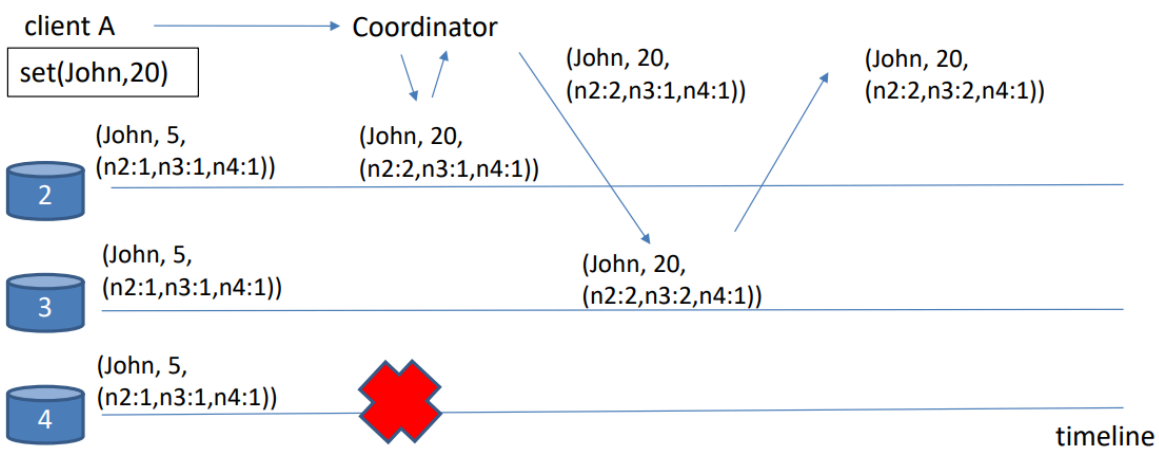
- If the old primary comes back online later, any writes that were not replicated to the new primary will be rolled back to maintain consistency.

Example: Node A is the primary and receives some writes. A network issue causes it to become isolated. Node B becomes the new primary. When A rejoins, MongoDB detects divergence and rolls back unreplicated operations on A.

What is the difference between a rollback in MongoDB and in a RDBMS like Postgres?

Aspect	MongoDB Rollback	Postgres Rollback
Trigger	Happens during replica set failover	Happens during a transaction failure
Cause	Replication divergence between nodes	User-defined transaction encounters an error
Scope	Rolls back unreplicated writes	Rolls back all operations in the transaction
Automatic?	Yes, triggered by the replica set mechanism	Yes, controlled by transaction logic (`ROLLBACK`)
Data loss risk	Yes, unless data was replicated	No, changes are not committed until `COMMIT`

Write Operation P2P - Version Vector (VV)



Write set(John,20) is executed successfully. Node 4 comes back on again and the next write → see next slide is executed.

All replicas store:

- Value: `John = 5`
- Version vector (VV) → `VV = {n2: 1, n3: 1, n4: 1}` meaning,
- All three replicas had performed or received one update to `"John"` up to this point.

client A performs `set(John, 20)`:

- the coordinator sends the write to `n2`
- `n2` increments its vv → `VV = {n2: 2, n3: 1, n4: 1}`, this node now stores `John = 20`.
- sent back to the coordinator
- now the write operation is sent to the node `n3`, it increments the vv received from n2 → `VV = {n2: 2, n3: 2, n4: 1}`
- since `n4` is down we check if the two vectors from `n2` and `n3` are comparable, they are $T \leq T'$ so we sync these vectors and get `VV = {n2: 2, n3: 2, n4: 1}`.
- node 4 comes back:

Write Quorum is majority.
What values do the nodes hold at the end of a successful write? What values do the VV hold?

client B performs `set(John, 100)` :

CASE 1:

- the coordinator sends the write to `n2`
- `n2` increments its vv → `VV = {n2: 3, n3: 2, n4: 1}`, this node now stores `John = 100`.
- sent back to the coordinator
- now the write operation is sent to the node `n3`, it increments the vv received from n2 → `VV = {n2: 3, n3: 3, n4: 1}`
- now the write operation is sent to the node `n4`, it increments the vv received from n3 → `VV = {n2: 3, n3: 3, n4: 2}`
- since write quorum is majority and coordinator has already received acknowledgment from 2 nodes out of three, it means that the majority has acknowledged the write and the message is sent to replica 4 telling to update

CASE 2:

- the coordinator sends the write to `n4`
- `n4` increments its vv → `VV = {n2: 1, n3: 1, n4: 2}`, this node now stores `John = 100`.
- sent back to the coordinator
- now the write operation is sent to the node `n3`, it increments the vv received from n2 → `VV = {n2: 1, n3: 2, n4: 2}`
- the vectors are not comparable so we can not sync them, the write will be successful, both VVs are stored as siblings with their respective values. reads will return both versions to the client. this conflict will later be solved by anti-entropy process.

State-Based CRDT Counter

Given is a replica set with 3 nodes that uses a CRDT incrementing and decrementing counter (normal counter where you can add to and subtract)
Counter is initialized 0. Apply the following updates. Write down the CRDT vectors for all nodes.

```
node1: incr counter by 10
node3: incr counter 50
Sync
node1: decr counter by 50
node2: incr counter 30
sync
node3: incr counter by 20
node3: decr counter by 40
sync
get(counter)
```

We're given a CRDT PN-Counter (Positive-Negative Counter), which works by keeping two vectors per node:

- P (increments): counts only increments
- N (decrements): counts only decrements
- When nodes synchronize, they merge the vectors using:

  ```
  merge(a, b) = element-wise max(a, b)
  ```

- The total value is:

$$\text{Value} = \sum P - \sum N$$

---

n1: (0, 0, 0), n2: (0, 0, 0), n3: (0, 0, 0)

1. node1: incr counter by 10 → `incr: (10, 0, 0)`, `decr: (0, 0, 0)`
2. node3: incr counter 50 → `incr: (0, 0, 50)`, `decr: (0, 0, 0)`
3. Sync → `incr: (10, 0, 50)`, `decr: (0, 0, 0)`
4. node1: decr counter by 50 → `incr: (10, 0, 50)`, `decr: (50, 0, 0)`
5. node2: incr counter 30 → `incr: (10, 30, 50)`, `decr: (0, 0, 0)`
6. Sync → `incr: (10, 30, 50)`, `decr: (50, 0, 0`)
7. node3: incr counter by 20 → `incr: (10, 30, 70)`, `decr: (50, 0, 0)`
8. node3: decr counter by 40 → `incr: (10, 30, 70)`, `decr: (50, 0, 40)`
9. Sync → `incr: (10, 30, 70)`, `decr: (50, 0, 40)`

What does the get(counter) return?

get(counter) → $(10 + 30 + 70) - (50 + 0 + 40) = 20$

What is the value of the state vectors of all nodes after the last sync?

- **P Vector:** `[10, 30, 70]`
- **N Vector:** `[50, 0, 40]`
- **Counter Value:** `20`

You want to reset the counter. How would a reset work?

The correct CRDT way to implement reset is to use epochs, e.g.:

```
counter = {
  epoch: 2,
  P: [0, 0, 0],
  N: [0, 0, 0]
}
```

On merge, choose the counter state with the highest epoch, and ignore others.

This preserves:

- Commutativity
- Associativity

- Idempotence

and ensures resets work correctly even with delayed updates.

State-Based CRDT Set

Given is a replica set with 3 nodes that uses a CRDT set datatype with the following sync rules (REDIS)

> "CRDT Sets & Redis Enterprise
> In a Redis Enterprise CRDT-enabled database, sets operations work under a few additional rules, the two most important of which get applied when merging operations coming from different nodes:
>
> 1. Adding wins over deleting.
> 2. Deleting works only on elements that the replica executing the command has already seen. The second rule is sometimes referred to as the “observed remove” rule, meaning that you can delete only items that you were able to observe when the command was issued.

Give an example for each of the rules to illustrate how they are executed.

1. **Adding wins over deleting.**

   "Adding wins over deleting" means that if one node adds an element and another deletes it concurrently (before syncing), the **add operation takes precedence**.

   - Initial state:
     - All nodes have an empty set: `{}`
   - Concurrent operations:
     - `N1`: `SADD myset "apple"`
     - `N2`: `SREM myset "apple"`

     These are not yet synced with each other.
   - Sync occurs between `N1` and `N2`:
     - Redis detects that `apple` was added and deleted concurrently.
   - According to the rule:
     - Add wins → `"apple"` remains in the set.

   myset = {"apple"}

2. **Deleting works only on elements that the replica executing the command has already seen.**

The second rule is sometimes referred to as the “observed remove” rule, meaning that you can delete only items that you were able to observe when the command was issued.

You can only delete an item if your node has already observed (i.e., knows about) that item.

a. Initial state:
   - All nodes have an empty set: `{}`

b. Node N1 adds "banana":
   - `N1`: `SADD myset "banana"`

c. Before N2 sees "banana", it tries to delete it:
   - `N2`: `SREM myset "banana"` ← Hasn’t seen it yet!

d. Now sync N1 and N2:
   - `N1` thinks "banana" was added.
   - `N2` tried to delete "banana", but had never seen it.

e. According to the rule:

- Delete is ignored because `N2` had no knowledge of "banana".

myset = {"banana"}

MongoDB Sharding

Let us assume you want to store the pictures taken from the cameras of the Duckey town cars into a MongoDB collection in order to be able to do analysis later. As there are a lot of pictures, you shard the documents. What shard key would you choose? Explain shard key and why you choose it.

---

image collection:

```
{
  "_id": ObjectId("..."),
  "bot_name": "gedi",
  "timestamp": Date("2025-06-09"),
  "image_data": BinData(...),
  ...
}
```

i would choose the composite shard key: { "bot_name": 1, "timestamp": 1 }

1. **Even data distribution**
   - `bot_name` helps spread data across different cars.
   - `timestamp` ensures that data from each car is also spread over time.
   - This prevents hotspots, especially if many pictures come from one car or one time period.
2. **Efficient query patterns**
   - Common queries for analysis will likely filter by:
     - specific `bot_name`
     - time ranges (`timestamp`)
   - This shard key allows for range-based queries on time, and equality filters on car, making queries efficient.
3. **Supports chunk splitting and balancing**
   - MongoDB can split chunks based on a compound key with a high cardinality like `(bot_name, timestamp)` and distribute them across shards more evenly.

Token Ring Sharding

1. **In what scenario can a hotspot happen in a token ring hash key distributed P2P database?**

   A hotspot occurs when too many keys hash to the same token range, causing one node to handle a disproportionate amount of data or traffic (too many requests (reads/writes) are directed to a small portion of the ring, causing Uneven load across nodes). This can happen due to:

   1. Poor key distribution
      - If keys are not uniformly random, e.g., keys like `user_0001`, `user_0002`, ... hash to nearby tokens.
      - This causes one or a few nodes in the ring to get overloaded.
   2. Skewed access patterns
      - Even if keys are uniformly distributed, accesses might concentrate on a few “hot” keys (e.g., a popular product or trending tag).
      - The nodes responsible for those keys become hotspots.
   3. Uneven token assignment
      - If token ranges are not evenly partitioned among nodes (e.g., some nodes have larger token responsibility), then those nodes may become overloaded.

2. **What mechanism can be used to prevent this?**
    - **Split hotspot keys into sub-keys**
        - If a single key is accessed heavily (e.g., a popular user or product), shard that key internally into multiple sub-keys (e.g., `"video_123#1"`, `"video_123#2"`).
        - This distributes reads/writes across different parts of the ring.
    - **Virtual nodes (vnodes)**
        - Each physical node is assigned multiple smaller token ranges instead of one.
        - This improves load distribution, simplifies rebalancing, and reduces the impact of adding/removing nodes.
    - **Adaptive load balancing**
        - Continuously monitor token ring segments for traffic or storage imbalances.
        - Dynamically move token ownership or redistribute virtual nodes to less-loaded nodes.
    - **Rate limiting**
        - Under high load, temporarily throttle traffic to hotspots to protect the cluster from being overwhelmed.
        - Often combined with backpressure and retry logic at the application level.