HLD NoSQL - Orchestration & Shard Creation

Contents

[Summary of the last class 2](#_Toc184150646)

[Sharding, and Denormalization 3](#_Toc184150647)

[Distributed Data Systems and Fault Tolerance 6](#_Toc184150648)

[Under-Replication and Multi-Master Architectures 9](#_Toc184150649)

[Multi-Master Architecture 10](#_Toc184150650)

[Multi-Master Consistent Hashing Architecture 11](#_Toc184150651)

[Tuneable Consistency in Multi-Master 17](#_Toc184150652)

[Gossip Protocol 21](#_Toc184150653)

[Case Studies 22](#_Toc184150654)

## Summary of the last class

**1. Limitations of Relational Databases (SQL)**

* **Joins across tables**: In large-scale systems, joining tables in SQL can be inefficient, especially when data is sharded across multiple servers.
* **Schema rigidity**: Relational databases require a fixed schema, which can be restrictive in scenarios where the structure of data might change frequently. This lack of flexibility is problematic when the content varies from record to record (e.g., categories or metadata).
* **Fixed data types**: SQL databases expect consistent lengths for fields (e.g., VARCHAR columns), which is not always practical for variable-length content like large text blocks or binary data.
* **Sharding**: SQL databases don't natively support sharding. Implementing sharding requires building a manual layer on top of SQL systems, which complicates management and scalability.

**2. Introduction to NoSQL Databases**

* **NoSQL Advantages**:
  + **Schema flexibility**: NoSQL databases allow for more flexible data structures that don’t require predefined schema.
  + **Support for large or complex data**: Perfect for use cases where data varies in structure, length, or type from one record to another.
  + **Sharding support**: NoSQL databases (like MongoDB) come with built-in sharding capabilities to distribute data across multiple servers easily.
* **Types of NoSQL Databases**:
  + **Key-Value Stores**: Simple storage models where data is stored as key-value pairs. Example: Redis, DynamoDB.
  + **Column Family Stores**: Data is stored in columns rather than rows, allowing efficient reading and writing of large volumes of data. Example: Cassandra.
  + **Document Stores**: Data is stored in flexible, semi-structured documents (e.g., JSON). Example: MongoDB, CouchDB.

**3. Sharding in NoSQL Databases**

* **Why Shard?** Sharding is necessary to scale databases across multiple machines to handle large volumes of data and traffic.
* **Sharding Key Selection**:
  + **Impact on query efficiency**: The sharding key determines how data is distributed across servers. A poorly chosen sharding key can lead to data hotspots and inefficient queries.
  + **Case studies**: The choice of sharding key depends on the specific use case and the most common queries in the system. Different use cases may require different sharding strategies.

**4. Two Key Steps for Sharding**

1. **Identifying the most frequent queries (Primary APIs)**:
   * Determine which API calls are the most common, as these will guide the choice of the sharding key. For example, if most queries are based on a user’s ID or channel ID, that key might be a good candidate for sharding.
2. **Choosing the appropriate sharding key**:
   * The sharding key should be chosen to distribute load evenly and to ensure that queries can be efficiently handled. For instance, using user ID or organization ID might be a good fit if most queries are tied to these entities.

## Sharding, and Denormalization

**1. Basic Architecture in NoSQL Databases**

* **Request Flow**: The flow starts when a user sends a request via a browser:
  1. **DNS** resolves the address and sends it to the **load balancer**.
  2. The **load balancer** forwards the request to one of the **application servers**.
  3. The application server processes the request (e.g., fetching the last 5 posts made by a user).
  4. The app server needs to access the storage (NoSQL database) to fetch the data.

**2. Relational DB (Monolith Example)**

* **Single Machine (e.g., MySQL)**: In a relational DB setup:
  + The application server queries the MySQL database to fetch data (e.g., SELECT \* FROM posts WHERE user\_id = X ORDER BY date DESC LIMIT 5).
  + The app server connects directly to the database and executes SQL queries.
  + This approach works well for a monolithic system where everything is stored in one machine, but scaling becomes an issue as data grows.

**3. Sharding in NoSQL Databases**

* **Problem with Scaling**: If data doesn’t fit on one machine, you need to shard the data across multiple machines.
  + **Sharding Key**: A sharding key (e.g., user\_id) is selected to distribute data across machines.
  + Each shard holds a subset of data (e.g., all posts by a particular user).
  + Sharding helps distribute the load across multiple machines, enabling horizontal scalability.

**4. Denormalization**

* **Definition**: Denormalization is the process of intentionally duplicating data across multiple places to improve query performance and reduce the need for joins.
  + In a relational database, normalization ensures no redundancy, but in sharded NoSQL databases, redundancy (denormalization) is required for efficient data retrieval.
* **Why De-normalize**:
  + In a sharded system, you may have information spread across multiple machines, making joins inefficient.
  + For example, a user's friends list and their posts might be stored on different shards, so instead of performing a join across machines, you store redundant data to avoid inter-shard communication.
* **Example**:
  + If **Abhi** and **Yusuf** are friends, instead of joining data from the friends table and posts table across different shards, you duplicate this information. Both **Abhi** and **Yusuf** will have entries in their own data to represent their friendship.
* **Normalization vs. Denormalization**:
  + **Normalization**: Avoids redundant data (one entry for a piece of information).
  + **Denormalization**: Allows duplication of data to improve query efficiency by avoiding joins across machines.

**5. Why Avoid Joins in Sharded Databases?**

* **Cross-shard Joins are Expensive**:
  + In NoSQL, data is distributed across multiple machines (shards), so joining data across shards requires network calls, which can be slow.
  + For example, if user data is on **Shard A** and post data is on **Shard B**, performing a join between these two shards would be inefficient and slow.
* **Solution**: De-normalize data so that related information resides within the same shard.

**6. Replication vs. Denormalization**

* **Replication**: Ensures that the same data is available on multiple machines for fault tolerance.
* **Denormalization**: Involves duplicating data within a shard to make it more efficient for queries. This doesn't imply fault tolerance or backup of data, but instead, ensures that data needed for a query resides within the same shard, avoiding the need for cross-shard joins.

**7. Example of Denormalization in Messaging Systems**

* Imagine a messaging app where messages are stored in a table. In a relational DB:
  + A single entry represents a message, with the sender and recipient IDs in one row.
* In a sharded system, if **User A** sends a message to **User B**, two copies of the same message are stored:
  + One in **User A’s shard** (representing the sent message).
  + One in **User B’s shard** (representing the received message).
* This is denormalization because the same message appears in two places, but it avoids cross-shard joins.

**8. Denormalization in Group Messaging**

* If users are part of a group and messages are sent to that group:
  + In a relational model, a message is stored once for the entire group, and multiple users are linked to that message.
* In a denormalized NoSQL model, each member of the group receives a copy of the message in their own shard.
  + This ensures that each user can retrieve the message from their own shard without needing to access other shards.

## Distributed Data Systems and Fault Tolerance

**Step-by-Step Process for Distributed Data Systems**

1. **Sharding Key Selection:**
   * Identify a sharding key to split data when it cannot fit on a single machine.
   * The sharding key determines how data is partitioned across multiple shards.
2. **Denormalization:**
   * Store related data together within the same shard to minimize cross-shard operations.
   * Some data may have duplicate copies in different shards for better access patterns.
3. **Fault Tolerance and Application-Level Transparency:**
   * Ensure shards have fault-tolerant setups.
   * Users and applications should not worry about shard management; the app server must handle shard routing transparently.

**Shard Setup and Fault Tolerance**

1. **Shard Structure:**
   * Each shard has a **master** and multiple **slave replicas**.
   * **Writes** go to the master, while **reads** can happen from either the master or slaves.
   * This ensures high availability and fault tolerance.
2. **Replication:**
   * Master-slave architecture is used for replication.
   * Fault tolerance ensures:
     + If a master fails, a slave is promoted to become the new master.
     + If any machine fails (e.g., hardware or network issues), operations continue without data loss.

A diagram of a computer system

Description automatically generated

**Handling Master Failures**

1. **Master Election:**
   * If the master dies, a new master is elected from the slaves.
   * **Zookeeper** (a distributed coordination service) facilitates this process:
     + Monitors the health of the master.
     + Notifies slaves and app servers when the master fails.
     + Elects a new master by allowing one slave to acquire a lock (based on criteria like having the latest data).
   * App servers are informed of the new master.
2. **Notification System:**
   * Zookeeper acts as a "magic box":
     + Keeps a record of master and slave configurations.
     + Notifies subscribers (e.g., app servers) of changes, enabling them to refresh their configuration.

**Application Server and Shard Interaction**

1. **Client-Server Code:**
   * App servers run client code (e.g., HBase Client, MongoDB Client) to interact with shards.
   * Shards (master/slaves) run server code to process requests.
2. **Initialization:**
   * During initialization, the app server:
     + Sets up Zookeeper connections.
     + Defines the number of shards, consistent hashing settings, and cluster details.
   * App servers subscribe to Zookeeper to get notified of master changes.
3. **Request Routing:**
   * For a request:
     + The sharding key determines the target shard (using consistent hashing).
     + For **writes**: The client library routes the request to the master of the shard.
     + For **reads**: The request can be routed to any replica.
4. **Master Updates:**
   * When the master changes:
     + Zookeeper notifies the app server.
     + The app server refreshes the master information in its client library.

## Under-Replication and Multi-Master Architectures

**Problem with Under-Replication**

1. **Definition of Under-Replication:**
   * Occurs when a shard does not meet the desired replication factor (e.g., three copies of data per shard).
   * Example: If a shard has only a master and two slave (instead of three slaves) due to a failure, it is under-replicated.

A diagram of a computer system

Description automatically generated

1. **Risks of Under-Replication:**
   * If another machine (e.g., master or remaining slave) fails, data loss becomes a possibility.
   * Reduces fault tolerance and increases the likelihood of system failure.
2. **Manual Intervention Requirement:**
   * If a machine fails, additional machines must be added manually to restore the desired replication factor.
   * Operational challenges:
     + Identify under-replicated shards.
     + Allocate available machines to these shards.
     + Synchronize data to the new machine.
3. **Desired System Behaviour:**
   * The system should self-heal without human intervention.
   * The goal is to build an automated, fault-tolerant system that operates for extended periods without manual oversight.

**Solutions for Under-Replication**

1. **Orchestration Systems:**
   * Some NoSQL databases (e.g., Cassandra, MongoDB) include built-in orchestrators to handle replication automatically.
   * Features:
     + Monitor shard replication status.
     + Reallocate machines from over-replicated or underutilized shards.
   * Challenges:
     + Requires excess machines to reallocate dynamically.
     + Manual maintenance is still required at the data center level (e.g., hardware repairs).
2. **Cloud-Based Solutions:**
   * Modern cloud systems can provide auto-scaling and auto-repair features.
   * Automatically adds machines when needed and replaces failed nodes.

## Multi-Master Architecture

1. **Motivation for Multi-Master Systems:**
   * Address two key problems:
     1. Latency during **master election** when the master fails.
     2. Manual effort required to handle under-replicated shards.
2. **How Multi-Master Works:**
   * Unlike master-slave systems, multiple masters can handle **writes** simultaneously.
   * Data is replicated across all masters.
   * Eliminates the need for a single point of failure during writes.
3. **Advantages of Multi-Master:**
   * No downtime during master failures (no need for elections).
   * Improved write availability since any master can handle writes.
4. **Challenges of Multi-Master:**
   * **Higher Latency:**
     1. Write operations need synchronization between all masters.
     2. This can lead to increased latency for write operations.
   * **Conflict Resolution:**
     1. Concurrent writes to multiple masters may lead to data conflicts.
     2. Requires robust conflict resolution strategies (e.g., last write wins, custom application logic).
   * **Complexity:**
     1. More complex system design and higher operational overhead.

## Multi-Master Consistent Hashing Architecture

**Problem Statements**

1. **Master Failure:**
   * In traditional master-slave systems, master failure leads to downtime or degraded performance during master re-election.
   * Need a mechanism to ensure continuous data availability and performance.
2. **Under-Replication:**
   * Manual effort is often required to maintain replication levels when machines fail.
   * Efficient utilization of all available machines and automatic handling of failures is desired.

**Solution Using Consistent Hashing**

1. **Introduction to Consistent Hashing:**
   * Machines (or shards) are represented on a logical ring (circle).
   * Each machine's position is determined by its hash value.
   * User data is mapped to machines based on hash values.
2. **Replica Placement in Consistent Hashing:**
   * **Optimal Replica Location:**
     + The next machine (hash-wise) on the ring is the most logical place to store replicas.
     + Reason: In case of failure, the next machine becomes responsible for the data and already having a replica minimizes recovery effort.
   * Multiple replicas:
     + Data can be replicated to more than one subsequent machine for redundancy.
     + Example: First replica goes to the next machine, second replica to the one after that.
3. **Illustration of Replica Placement:**
   * Machine M1 stores its own data (e.g., A, B) and a replica of M6’s data (last machine on the circle).
   * Machine M2 stores:
     + Its own data (e.g., C, D).
     + Replica of M1’s data (A, B).

A diagram of a diagram of people

Description automatically generated

* + This pattern continues for all machines, ensuring that each machine has:
    - Its own data.
    - Copies of data from neighbouring machines based on the consistent hashing ring.

**Handling Machine Failures**

1. **Data Mapping on Failure:**
   * If machine M2 fails:
     + Data originally mapped to M2 (e.g., C, D) is now mapped to M3 (next machine on the ring).
     + Since M3 already has a replica of M2’s data, operations can continue without interruption.
2. **Dynamic Recovery:**
   * The system can start copying data from M3 to another machine (e.g., M4) to maintain replication levels dynamically.
   * This minimizes manual intervention.
3. **Replication Markers:**
   * Machines differentiate between:
     + **Own data:** Data assigned directly by the hash function.
     + **Replica data:** Data copied from neighbouring machines.
   * Markers or metadata can track this distinction to ensure correct replication and recovery operations.

**Multi-Master Consistent Hashing with Optimal Utilization**

**Key Problem Statements**

1. **Fault Tolerance:**
   * Ensure that the system remains functional even when machines fail.
   * Maintain data availability through replicas without downtime.
2. **Optimal Utilization:**
   * Avoid idle machines by ensuring all participate in serving traffic.
   * Maximize hardware utilization while maintaining fault tolerance.

**Concepts and Solutions**

1. **Replication Levels:**
   * **Definition:** Number of copies of each data item.
   * **Trade-offs:**
     + Higher replication levels increase fault tolerance but require more storage and may increase latency.
     + Lower replication levels save space but risk data loss in case of multiple failures.
   * Example: A typical replication level is 3 (primary + 2 replicas).
2. **Replica Placement Strategy:**
   * **Primary Data:** Stored on the machine determined by the consistent hashing algorithm.
   * **Replica 1:** Placed on the next machine in the clockwise order.
     + Rationale: If the primary machine fails, requests naturally route to this machine.
   * **Replica 2 (and beyond):** Placed on subsequent machines in clockwise order.
     + Rationale: Provides redundancy for cascading failures (e.g., primary and first replica failing).

**Illustration of Replica Placement**

* **Example Setup:**
  + Machines: M1, M2, M3, M4, M5.
  + Replication level: 3.
  + Users: User A and B.
* **Data Placement:**
  + Primary data for A and B: M3.
  + First replica: M5.
  + Second replica: M4.
* **Failover Sequence:**
  + If M3 fails: Requests for A and B go to M5.
  + If both M3 and M5 fail: Requests go to M4.

A diagram of a computer network

Description automatically generated

**Optimized Machine Utilization**

1. **Breaking Down Workload:**
   * In a multi-master setup, each machine acts as the master for a subset of data.
   * All machines serve traffic for both primary data and replicas.
2. **Benefits of Multi-Master Approach:**
   * Spreads workload evenly across machines.
   * Eliminates idle "slave" machines, ensuring all resources are utilized.
   * Reduces the load per machine by dividing user assignments.
3. **Storage Distribution:**
   * Each machine:
     + Stores its own data (e.g., 30% of storage).
     + Stores replicas of other machines’ data (e.g., 60-70% of storage).
   * Requires more machines but avoids single points of failure.

**Consistent Hashing Recap**

1. **Data Mapping:**
   * User requests are mapped to machines based on hash values.
   * If a machine fails, requests are routed to the next available machine in clockwise order.
2. **Order of Replica Placement:**
   * Primary -> Replica 1 -> Replica 2 -> Replica 3 (clockwise on the hash ring).
   * Example: M3 -> M5 -> M4.
3. **Adding or Removing Machines:**
   * When a new machine is added:
     + Data reassignment happens locally, affecting only nearby machines.
     + New machine takes a portion of the data and starts participating in serving traffic.

**Comparison to Traditional Master-Slave**

1. **Master-Slave Model:**
   * One machine serves as the master; others are idle slaves.
   * Slaves are underutilized unless the master fails.
2. **Multi-Master Model:**
   * All machines act as masters for specific subsets of data.
   * Balances load across machines and improves utilization.
3. **Fault Tolerance:**
   * Multi-master with consistent hashing provides redundancy and rapid failover without manual intervention.

**Trade-Offs and Considerations**

1. **Fault Tolerance vs. Space Usage:**
   * Higher replication increases safety but uses more storage.
   * Systems need to balance fault tolerance, space, and performance.
2. **Latency:**
   * Writing multiple replicas may slightly increase write latency.
   * Read latency is typically unaffected due to distributed data placement.
3. **Scalability:**
   * Multi-master systems scale well with consistent hashing.
   * Minimal redistribution of data when machines are added or removed.

## Tuneable Consistency in Multi-Master

**Key Concepts**

1. **Replication Level (X):**
   * Represents the number of replicas in the system.
   * Example: If X = 3, data for a key is replicated across 3 machines.
2. **Consistency Hashing Ring:**
   * Data is distributed across nodes using consistent hashing.
   * Each node is responsible for keys within a specific range on the hash ring.
3. **R and W Parameters:**
   * **R (Read Threshold):**
     + The minimum number of replicas to read from for a read request to be considered successful.
   * **W (Write Threshold):**
     + The minimum number of replicas to write to for a write request to be considered successful.
4. **Behaviour Control Using R and W:**
   * : Ensures high consistency.
   * Smaller **R** → Faster reads.
   * Smaller **W** → Faster writes.

**Case Scenarios**

1. **R = 1, W = 1:**
   * **Behaviour:**  
     Highly available, but less consistent.
     + Writes are fast, as only one machine needs to be updated.
     + Reads are fast, as they require only one replica.
   * **Risk:** Inconsistent data if writes haven't propagated to other replicas.
2. **R = 3, W = 1:**
   * **Behaviour:**  
     High consistency but lower availability for reads.
     + Read requests require all 3 replicas to respond.
     + Writes are fast, requiring only one replica.
   * **Risk:** Read failures if any replica is down.
3. **R = 1, W = 3:**
   * **Behaviour:**  
     High consistency but lower availability for writes.
     + Write requests require all 3 replicas to succeed.
     + Reads are fast as only one replica is needed.
   * **Risk:** Write failures if any replica is down.
4. **R = 3, W = 3:**
   * **Behaviour:**  
     Maximum consistency but low availability.
     + Both reads and writes are slower, as all replicas must be accessed.
5. **R + W < X:**
   * Behaviour: The system may fail to achieve consistency.
   * Risk of data loss if failures occur before replicas synchronize.

**Advantages of Configurability**

* **Read-Optimized Systems:**
* Keep R small to ensure fast reads. Suitable for applications with high read frequency (e.g., content delivery).
* **Write-Optimized Systems:**
* Keep W small for faster writes. Suitable for write-heavy applications (e.g., logging systems).

**Data Propagation and Synchronization**

* **Gossip Protocol:**
  + Mechanism by which replicas exchange information to achieve eventual consistency.
  + Nodes periodically communicate to propagate updates.

**Consistency Spectrum**

* **Strong Consistency:**
* . Ensures the latest data is always read but sacrifices availability and speed.
* **Eventual Consistency:**
* Smaller R or W. Allows for temporary inconsistencies but offers high availability and lower latency.

**Application-Level Handling**

* **Conflict Resolution:**
  + Use timestamps to identify the latest update.
  + Application logic determines how to merge conflicting updates (e.g., Amazon’s shopping cart merging).
* **Custom Merging Strategies:**
  + Take the latest timestamp.
  + Merge multiple responses (e.g., union of items in shopping carts).

**Usage in Modern Databases**

* Databases like **Cassandra** and **HBase** natively support this configuration.
* Relational Databases (RDBMS): Lack built-in mechanisms; require custom implementation for replication and conflict resolution.

**R + W Configuration for Tuning Consistency and Availability**

**Variables**

* X: Number of replicas (replication level).
* R: Number of nodes required for a read to succeed.
* W: Number of nodes required for a write to succeed.

**Key Rules:**

1. R + W > X: Ensures **strong consistency**.
2. Lower R: Improves **read speed**.
3. Lower W: Improves **write speed**.

**Scenarios**

1. **R = 1, W = 1**
   * **Behaviour**:
     + Read succeeds if any one replica is reachable.
     + Write succeeds if any one replica is updated.
   * **Outcome**: Highly **available** but less **consistent**.
2. **R = 3, W = 1**
   * **Behaviour**:
     + Write is quick (updates only one replica).
     + Read requires data from all replicas.
   * **Outcome**: Highly **consistent**, but not **available**.
3. **R = 1, W = 3**
   * **Behaviour**:
     + Read is quick (reads from any one replica).
     + Write requires updating all replicas.
   * **Outcome**: Highly **consistent**, but writes are slow, and the system is not highly **available**.
4. **R = 2, W = 2 (R + W > X)**
   * **Behaviour**:
     + Balances consistency and availability.
     + Requires at least two replicas to read/write.
   * **Outcome**: Achieves a balance between availability and consistency.

**Implications of Configuration Choices**

1. **Highly Available but Risky**
   * Example: W = 1
     + If the node where the write is performed (e.g., M4) fails before synchronization, data loss is possible.
     + Suitable for scenarios prioritizing availability over consistency (e.g., caching systems).
2. **Highly Consistent but Not Highly Available**
   * Example: R = 3, W = 3
     + Ensures strong consistency but makes the system vulnerable to node failures.
     + Suitable for financial systems or shopping cart applications where consistency is critical.

## Gossip Protocol

1. **Overview**
   * Nodes exchange recent transaction logs to synchronize data.
   * Each node maintains a **timestamp** of the last successful synchronization with its peers.
2. **Process**
   * **Node A asks Node B**: "What transactions have occurred since timestamp T?"
   * **Node B responds**: Shares updates after T.
   * Both nodes merge their updates to achieve consistency.
3. **Key Properties**
   * Data eventually becomes consistent across all nodes.
   * Efficient for systems requiring eventual consistency.

**General Observations**

* **Lower R/W Values**: Favor availability and speed but reduce consistency.
* **Higher R/W Values**: Favor consistency but reduce speed and availability.
* The gossip protocol enables **eventual consistency**, ensuring all replicas converge to the same state over time.

**Best Practices for Configuring Multi-Master Systems**

1. **Understand Application Needs**
   * If reads dominate, set a low R for faster responses.
   * If writes dominate, set a low W to reduce write latency.
2. **Critical Data**
   * Use R + W > X for critical data to ensure strong consistency.
3. **Avoid W = 1 in Critical Systems**
   * Risk of data loss if the single write node fails before replication.
4. **Monitor Node Health**
   * Ensure node reliability to maintain consistency and avoid data loss.

## Case Studies

**1. YouTube View Counter**

* **Priority**: High availability (minor inconsistencies acceptable).
* **Read-Write Profile**:
  + Reads: Very frequent (must be fast).
  + Writes: Frequent but losing a few counts is acceptable.
* **Configuration**:
  + R = 1: Reads from any replica to prioritize speed.
  + W = 1: Writes to one replica for low latency.
* **Reasoning**:
  + Minor inconsistencies (e.g., view count differing by 1-2) are imperceptible to users.
  + Gossip protocol ensures eventual consistency.

**2. Instagram Post Updates**

* **Priority**: High availability, eventual consistency (data must not be lost).
* **Read-Write Profile**:
  + Reads: Much more frequent than writes.
  + Writes: Losing data (e.g., posts) is unacceptable.
* **Configuration**:
  + R = 1: Fast reads for user experience.
  + W = 2: Writes to at least two replicas to ensure data safety.
* **Reasoning**:
  + Fast reads provide a seamless user experience.
  + Writing to two nodes minimizes the risk of data loss even if one node fails.

**3. Bank Transactions and Balance Updates**

* **Priority**: High consistency (financial data integrity critical).
* **Read-Write Profile**:
  + Reads: Frequent balance inquiries.
  + Writes: Equally frequent due to transactions.
* **Configuration Options**:
  + **Option 1**: R = 2, W = 2 (balanced consistency and availability).
  + **Option 2**: R = 1, W = 3 (faster reads, slower writes).
* **Reasoning**:
  + R = 2, W = 2: Balances consistency and availability; supports operations during a single-node failure.
  + R = 1, W = 3: Prioritizes strong consistency by ensuring all writes succeed before being acknowledged.

**Critical Observations**

1. **High Availability Systems**:
   * Suitable for use cases where minor inconsistencies do not impact user experience (e.g., YouTube counters, social media).
   * Typically use lower values for R and W.
2. **Strong Consistency Systems**:
   * Necessary for systems like financial services, inventory management, and order processing.
   * Require R + W > X for strict data integrity.
3. **Trade-offs in Write Failures**:
   * In W = X, any node failure prevents writes.
   * In W < X, systems can write but might delay or fail to propagate updates to all replicas.