HLD - CAP Theorem & Master-Slave

Contents

[Replication in Distributed Systems 2](#_Toc184071507)

[Real-World Analogy: 3](#_Toc184071508)

[Consistency Issue 4](#_Toc184071509)

[Availability Issue 7](#_Toc184071510)

[Partition in the System 10](#_Toc184071511)

[CAP Theorem 12](#_Toc184071512)

[PACELC Theorem 13](#_Toc184071513)

[Consistency vs. Availability (CAP Tradeoff) 15](#_Toc184071514)

[Case Studies: Application Scenarios 16](#_Toc184071515)

[Decision Framework for Consistency vs. Availability 18](#_Toc184071516)

[Replication in Distributed Systems 19](#_Toc184071517)

[Database Replication Using Master-Slave Architecture 20](#_Toc184071518)

[System Types Based on Trade-offs (CAP Theorem) 23](#_Toc184071519)

[Banking and Financial Systems 25](#_Toc184071520)

[Introduction to NoSQL Databases 28](#_Toc184071521)

## Replication in Distributed Systems

**Introduction**

* **Problem of Single Machine Storage:**
  + Storing information on a single machine creates risks:
    - **Hardware Failure:** Data loss occurs if the hard disk fails.
    - **Single Point of Failure (SPOF):** Entire system dependent on one machine.
* **Replication:**
  + Solution involves creating **replicas** (multiple copies) of the same data across multiple machines.

**Replication: Concept and Importance**

* **Definition:**  
  Storing multiple copies of the same data on different machines to ensure reliability and availability.
* **Benefits of Replication:**
  1. **Fault Tolerance:** Data remains accessible even if one machine fails.
  2. **Availability:** Improves system uptime by preventing SPOFs.
  3. **Load Balancing:** Distributes read/write operations across multiple replicas, reducing load on any single machine.

**Challenges in Distributed Systems with Replication**

* **Complications in Distributed Data:**
  + Unlike single-machine systems where all processes are local, multiple data storage points introduce complexity.
  + Examples:
    1. **Consistency Issues:** Ensuring that all copies of the data are identical.
    2. **Concurrency Management:** Handling simultaneous read/write operations across replicas.
    3. **Latency:** Data synchronization across machines takes time and may affect performance.
    4. **Network Dependency:** Increased reliance on network stability for communication between machines.

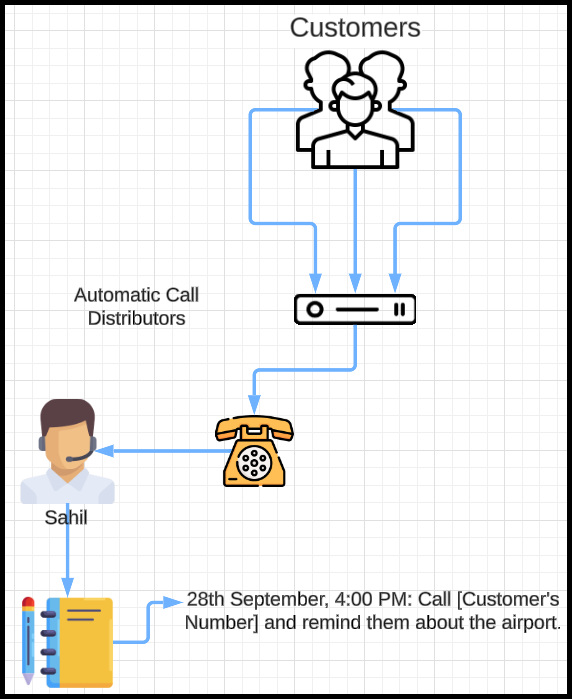
## Real-World Analogy:

* A hypothetical company setup will be used here to illustrate these complications.

**Story Framework**

* **Protagonist:** Sahil, an entrepreneur starting a company named **ABCD**.
* **Business Idea:**
  + A **reminder service** where customers can schedule reminders.
    - Example:
    - A customer has a flight at 7:00 PM tomorrow and requests a call at 4:00 PM to remind them to leave for the airport.
    - Sahil’s company makes the reminder call at the specified time.

**How the System Works**

1. **Input:**
   * Customers call Sahil to schedule reminders with details such as:
     + Date and time of the reminder.
     + A phone number to call back.
     + Reminder content (e.g., "Have you left for the airport?").
2. **Data Storage:**
   * Sahil manually records these details in a diary.
     + Example entry: **28th September, 4:00 PM: Call [Customer's Number] and remind them about the airport.**
3. **Output:**
   * At the scheduled time, Sahil checks his diary and makes the reminder call.

**Story Progression: Scaling the Reminder Service**

* **Problem of Growth:**
  + Sahil's company, **ABCD**, becomes highly popular, but Sahil alone cannot handle the increasing traffic.
  + A single person can:
    - Take only **one call at a time.**
    - Handle a limited number of requests (e.g., one call every 30 seconds).
* **Solution: Adding a Second Operator (Wife):**
  + Sahil involves his wife to share the workload, effectively **doubling the throughput.**
  + **Setup:**
    - Sahil and his wife each maintain their own **diary** for reminders.
    - A **load balancer/interface** redirects incoming calls to whoever is free.

## Consistency Issue

* **Incident with Sachin:**
  + Sachin calls Sahil, requesting a reminder for a flight at 6:00 PM the next day.
  + The call initially goes to Sahil's wife, who records the reminder in her diary.
  + Later, Sachin calls back, but the call is redirected to Sahil.
  + Sahil checks **his diary** but does not find the reminder because it was recorded in his wife’s diary.

A diagram of a call center

Description automatically generated

* + **Result:**
    - Sachin receives incorrect information, becomes dissatisfied, and leaves the service.
* **Key Observation:**
  + Entries in Sahil's and his wife's diaries are **inconsistent**, leading to a **bad user experience** and loss of a customer.

### Proposed Solution to Consistency Problem

1. **Two Diaries Approach:**
   * Sahil and his wife maintain **separate diaries** (replicated storage).
   * A **synchronization mechanism** is introduced to ensure both diaries have identical information.
2. **Process for Handling Write Requests:**
   * When a reminder request (e.g., **Sachin, 6:00 PM**) is received:
     1. **Primary Entry:**
        + The recipient of the call (e.g., Sahil’s wife) records the entry in her diary.
     2. **Replication:**
        + She ensures the same entry is written in Sahil's diary.
     3. **Confirmation:**
        + Only after both diaries contain the entry does she confirm to the customer (Sachin) that the reminder is set.
   * This process ensures **both diaries remain synchronized.**

A diagram of a call center

Description automatically generated

1. **Example for Clarity:**
   * If Sahil receives a write request (e.g., **Prakash, 8:00 PM**):
     1. He writes it in his diary first.
     2. Sahil then asks his wife to write it in her diary.
     3. Only after both diaries are updated does he confirm the reminder to Prakash.

**Why Separate Diaries Instead of a Shared Diary?**

1. **Challenges with a Shared Diary:**
   * **Read/Write Conflicts:**
     + If Sahil and his wife both try to write or read from the shared diary simultaneously, conflicts occur.
     + Example: Sahil reading page #25 while his wife reads page #50 creates contention.
   * **Limited Parallelism:**
     + Only one person can use the diary at a time, significantly reducing efficiency.
2. **Advantages of Separate Diaries:**
   * **Parallel Reads:**
     + Both can handle **read operations** (e.g., reminders or inquiries) simultaneously.
   * **Improved Efficiency for Reads:**
     + Faster response times for reading reminders.
   * **Consistency Maintained for Writes:**
     + Synchronizing writes ensures no inconsistency across the diaries.

## Availability Issue

1. **Problem Setup:**
   * Sahil’s wife is sick and stays at home, taking her diary with her.
   * Sahil is left at the office with his own diary, while his wife’s diary is **inaccessible**.
2. **Incoming Write Request:**
   * A customer (e.g., **Aditya**) calls to set a reminder (**flight tomorrow at 2:00 PM**).
   * Sahil follows the protocol:
     1. **Write in his diary.**
     2. **Ensure the same entry is written in his wife’s diary** before confirming to Aditya.
   * However, because his wife and her diary are unavailable:
     1. Sahil cannot synchronize the data.
     2. As a result, he **cannot confirm the reminder** to Aditya.
3. **Key Observation:**
   * All **write operations fail** because one diary is unavailable.
   * **Read operations** still work:
     1. Sahil can check his own diary and provide reminders for existing entries.

A diagram of a telephone system

Description automatically generated

**Understanding the Availability Problem**

1. **Definition of Availability:**
   * A system is considered **available** if it can handle both **read and write operations** despite certain failures or unavailability of components.
2. **What Happens Here:**
   * Sahil ensures **full consistency** by requiring updates to both diaries before confirming a write.
   * When one diary (or person) is unavailable, the system prioritizes consistency but sacrifices availability for write operations.
3. **Real-World Analogy:**
   * A messaging app like WhatsApp:
     + If you can read past messages but cannot send new ones due to server issues, the service is deemed **unavailable**.

### Proposed Solution to Availability Problem

1. **Problem Setup:**
   * Sahil’s wife is unavailable (e.g., sick at home) and cannot participate in the system.
   * Sahil must still handle **write requests** to maintain the service’s availability for users.
2. **Proposed Solution:**
   * Sahil **takes notes in his diary alone** and responds to the user (e.g., **Aditya**) that their reminder is successfully recorded.
   * When Sahil’s wife returns to the office:
     1. She must **synchronize her diary** with all the entries Sahil made in her absence.
     2. **Only after synchronization** can she resume taking calls and processing requests.

A diagram of a telephone connection

Description automatically generated

1. **Synchronization Details:**
   * Sahil maintains all entries in **sequential order** in his diary, ensuring easy identification of recent changes.
   * When his wife returns, she reads Sahil’s diary, copying over all the entries made during her absence.
   * This process ensures that both diaries have the **same information** before resuming normal operations.
2. **Key Outcome:**
   * This approach resolves the **availability issue** by allowing Sahil to continue handling write operations even when his wife is unavailable.
   * Synchronization ensures **eventual consistency** across the two diaries.

**Real-World Analogy:**

* **Database Recovery Process:**
  + If a database node goes down temporarily, it stops handling traffic.
  + When the node comes back online, it first **synchronizes with the latest data** from active nodes.
  + Only after catching up does, it resumes normal operations (both reads and writes).

## Partition in the System

**Scenario: Communication Failure (Partitioned System)**

1. **Problem Setup:**
   * Sahil and his wife have had a fight, and **they are not communicating**.
   * Both are still operational and taking calls, but **no synchronization** occurs between them.
2. **Challenges:**
   * A call from a user (e.g., Prabal) may go to either Sahil or his wife.
   * The system must decide how to handle **write requests** when there is no communication.

A diagram of a call center

Description automatically generated

1. **Two Choices for Handling Writes:**
   * **Choice 1: Write and Respond Success**
     + The node (Sahil or his wife) receiving the call **writes the entry** in their diary and informs the user of success.
     + **Result:**
       - **Inconsistent system:** The other node does not have the entry, leading to data discrepancies.
       - If the user later queries the other node, the reminder won't be found.
   * **Choice 2: Reject the Write Request**
     + The node refuses to take the write request, informing the user of failure.
     + **Result:**
       - The system remains **consistent** since no unsynchronized writes are made.
       - **Availability issue:** Users experience downtime as their requests are rejected.

## CAP Theorem

**CAP Theorem Overview**

1. **Definition:**
   * The **CAP Theorem** states that in a distributed system, when a **network partition** (P) occurs, you cannot have both **consistency (C)** and **availability (A)** simultaneously.
   * You must choose between:
     + **Consistency (C):** All nodes see the same data at any time.
     + **Availability (A):** Every request gets a response (success or failure).
     + **Partition Tolerance (P):** The system continues to function despite network partitioning.
2. **Key Points:**
   * If there is **no partition**, the system can have both consistency and availability.
   * During a **partition**, you can choose either:
     + **Consistency:** Ensure all nodes have synchronized data, but sacrifice availability (reject writes).
     + **Availability:** Allow the system to remain operational, but risk inconsistency.
3. **Real-World Analogy:**
   * **Partition:** Sahil and his wife are not communicating (network issue).
   * **Consistency Priority:** Reject write requests to maintain synchronization.
   * **Availability Priority:** Accept write requests but risk having different entries in their diaries.

**Handling Network Partition**

1. **Consistency-Centric Approach:**
   * Reject all writes during partition.
   * Ensure diaries remain synchronized.
   * **Trade-off:** Reduced availability (users experience downtime).
2. **Availability-Centric Approach:**
   * Allow writes to proceed on individual nodes.
   * Accept potential inconsistency during partition.
   * **Trade-off:** System becomes inconsistent.
3. **Eventual Consistency:**
   * After the partition resolves, nodes reconcile differences and synchronize their data.
   * During the partition:
     + Nodes operate independently, leading to temporary inconsistencies.
   * After synchronization:
     + System achieves **eventual consistency**, ensuring all nodes converge to the same state.

## PACELC Theorem

1. **Definition:**
   * The **PACELC theorem** expands on the CAP theorem by addressing system behaviour both during and outside of network partitions.
   * It states:
     + **P (Partition):** During a network partition, you must choose between **availability (A)** and **consistency (C)** (same as CAP).
     + **ELC (Else):** When there is no partition, you must choose between **latency (L)** and **consistency (C)**.

**PACELC Breakdown**

1. **During Network Partition (P):**
   * The system must compromise between:
     + **Availability:** Ensure that the system responds to all requests, even if some data is inconsistent.
     + **Consistency:** Ensure all nodes have synchronized data but reject requests when this cannot be guaranteed.
2. **When No Partition Exists (ELC):**
   * The system operates normally, and you face a trade-off between:
     + **Low Latency (L):** Fast response times, but the system may become **eventually consistent** (temporary inconsistency until all nodes synchronize).
     + **Strong Consistency (C):** Immediate synchronization across all nodes, but response times increase due to the synchronization overhead.

**Real-World Analogy**

1. **Sahil's Example:**
   * **Network Partition (P):** Sahil and his wife cannot communicate.
     + Sahil must decide whether to:
       - **Be Available (A):** Accept requests locally but risk inconsistency between his and his wife's diaries.
       - **Be Consistent (C):** Reject requests to ensure synchronization.
   * **No Network Partition (ELC):** Sahil and his wife can communicate.
     + Sahil must decide whether to:
       - **Optimize for Latency (L):** Write in his diary only and asynchronously update his wife’s diary later, leading to temporary inconsistency.
       - **Ensure Consistency (C):** Write in both diaries synchronously, increasing response time.
2. **Impact of Choices:**
   * **Latency-Focused System:**
     + Sahil writes quickly, promising immediate responses, but temporary mismatches between diaries may occur.
   * **Consistency-Focused System:**
     + Sahil delays responses until both diaries are updated, ensuring consistency but with increased wait time.

**Design Considerations**

1. **Consistency Trade-Offs:**
   * **Immediate Consistency:** Ensures all replicas have the same data instantly.
     + Example: Banking systems where data accuracy is critical.
   * **Eventual Consistency:** Ensures data consistency over time.
     + Example: Social media updates where slight delays are acceptable.
2. **Latency Optimization:**
   * Use **asynchronous synchronization** to handle writes in the background.
   * Risk: Temporary inconsistency until synchronization is complete.
3. **Failures in Async:**
   * Retry mechanisms can ensure eventual synchronization if the initial attempt fails.
   * Persistent failures may compromise consistency.

## Consistency vs. Availability (CAP Tradeoff)

1. **Definitions**:
   * **Consistency**: Ensures that all nodes display identical, up-to-date data. Every read reflects the most recent write.
   * **Availability**: Guarantees that the system always responds to requests, though the data may not be consistent.
   * **Partition Tolerance**: Ability to continue functioning despite network partitions. The CAP theorem states that in a partitioned network, you can only choose **Consistency** or **Availability**, not both.
2. **Modern Interpretation**:
   * **Context-Dependent Choice**: The choice between consistency and availability depends on the product’s needs.
   * Examples:
     + Banking systems prioritize **Consistency**.
     + Social media platforms like Facebook prioritize **Availability**.

## Case Studies: Application Scenarios

**Banking System (e.g., Axis Bank)**

* **Importance**: Consistency is critical.
* **Reasoning**:
  + Incorrect bank balances or transactions can result in severe financial and trust issues.
  + Availability is secondary because a temporary outage is less damaging than inconsistent data.
* **Example**:
  + Displaying ₹900 when the actual balance is ₹1200 or vice versa could cause serious issues.

**Social Media Platforms (e.g., Facebook News Feed)**

* **Importance**: Availability is prioritized.
* **Reasoning**:
  + Users expect the site to load quickly.
  + Temporary inconsistencies in the number of likes or missing posts are acceptable.
* **Key Tradeoff**:
  + Data like "number of likes" can be **eventually consistent**.

**Content Platforms (e.g., Quora)**

* **Primary Choice**: Availability.
* **Reasoning**:
  + Users are okay with minor inconsistencies in comments or likes but expect the platform to load and function reliably.
* **Additional Insight**:
  + Static content (e.g., a question post) is inherently consistent because it's stored in a single database entry.

**Stock Market Systems**

* **Primary Choice**: Consistency.
* **Reasoning**:
  + Accurate and up-to-date prices are essential for trading decisions.
  + Stale or incorrect data could result in significant financial losses.
* **Tradeoff**:
  + Availability might be compromised during network partitions to ensure consistency.

**Messaging Systems (e.g., Facebook Messenger)**

* **Primary Choice**: Consistency.
* **Reasoning**:
  + Out-of-order or missing messages can cause confusion and relationship issues.
* **Tradeoff**:
  + Availability may be compromised temporarily to ensure messages are consistent and in the correct order.

**E-Commerce Systems (e.g., Amazon)**

* **Balanced Approach**:
  + **Consistency** for critical operations like purchases and inventory updates.
  + **Availability** for non-critical features like product recommendations or reviews.

## Decision Framework for Consistency vs. Availability

1. **What is Being Written or Updated?**
   * The nature of the data being written or updated is crucial.
   * Example:
     + **Critical Data:** Bank transactions, stock prices — require high consistency.
     + **Non-Critical Data:** Social media likes, traffic patterns — can tolerate eventual consistency.
2. **How Critical is the Accuracy of That Data?**
   * If users expect **immediate correctness**, prioritize **consistency**.
     + Example: Bank account balances, stock market prices.
   * If slight delays or temporary inaccuracies are tolerable, prioritize **availability**.
     + Example: Traffic data, social media notifications.
3. **Consistency-First Approach:**
   * If the app **must display correct information all the time**, even at the cost of being unavailable occasionally, prioritize **consistency**.
     + Example: If the app shows incorrect account balances, it could lead to significant consequences.
4. **Availability-First Approach:**
   * If users are **OK with seeing delayed or eventually correct information**, prioritize **availability**.
     + Example: For non-critical updates like likes or traffic congestion, availability ensures a seamless user experience despite minor inconsistencies.
5. **The Core Question:**
   * Is the system **better off being consistently correct** or **always accessible but potentially inconsistent**?

## Replication in Distributed Systems

The discussion focuses on **replication** to avoid single points of failure and introduces the concepts of **master-slave architecture** and its synchronization mechanisms.

**1. Motivation for Replication**

* **Why Replication is Needed:**
  + A single machine storing all information is a **single point of failure**.
  + Replication ensures **data availability** and **fault tolerance** by storing copies of data on multiple machines.
* **Challenges of Replication:**
  + Ensuring that replicas have the same data leads to **consistency challenges**.
  + Once replication is introduced, decisions about **consistency vs. availability** (as per CAP theorem) must be made.

**2. Master-Slave Architecture**

* **Definition:**
  + **Master Node:** A single machine responsible for handling **all write operations**.
  + **Slave Nodes:** One or more machines that maintain copies of the data from the master.
* **Workflow:**
  + **Writes:** All writes (e.g., updates to scores, transactions) are directed to the **master**.
  + **Replication:** The master ensures that changes are replicated to the slaves.
  + **Reads:** Slaves can handle read operations, reducing load on the master.
* **Advantages:**
  + Reduces the load on the master by distributing read operations.
  + Ensures that data is not lost in case the master fails (provided slaves are synced).
* **Challenges:**
  + Keeping slaves **in sync** with the master can be complex, especially in highly available or distributed environments.
  + Synchronization mechanisms must be carefully designed.

A diagram of a computer system

Description automatically generated

**Questions Discussed**

1. **How Does Synchronization Happen in Distributed Systems?**
2. **What Would Master-Slave Look Like in Different Systems?**

## Database Replication Using Master-Slave Architecture

The discussion elaborates on **write-ahead logs (WAL)** and their role in maintaining consistency across replicas in a **master-slave setup**.

**1. Basics of Relational Databases**

* **Data Storage in Relational Databases:**
  + Information is stored in **tables** (rows and columns).
  + Databases maintain **write-ahead logs (WAL)** to record changes.
* **Write-Ahead Logs (WAL):**
  + A file that records all **write operations** (insert, update, delete) performed on the database.
  + Entries in the WAL are **timestamped** and ordered sequentially.
  + WAL contains operations like:
    - **Insert** (e.g., "Insert into Table X1 at timestamp T0").
    - **Update** (e.g., "Update Table X2 at timestamp T1").
    - **Delete** (e.g., "Delete from Table X1 at timestamp T3").
  + **Purpose of WAL:**
    - Serves as a backup to recover the database state if tables are lost.
    - By replaying the WAL operations in order, the database can rebuild tables and restore the exact state.
* **Relation Between WAL and Tables:**
  + Updates are **first written to WAL** and then applied to tables.
  + WAL is the **primary mechanism** for ensuring data persistence and recovery.

**2. Master-Slave Architecture**

* **Overview:**
  + The **master** handles all **write operations** (insert, update, delete).
  + The **slaves** maintain copies of the master’s data and are used for **read operations**.
* **Advantages:**
  + Distributes read loads across slaves, improving performance.
  + Provides fault tolerance, as slaves can replace the master if it fails (with appropriate failover mechanisms).
* **Replication Mechanism:**
  + **New Writes:**
    - When the master performs a write operation, it logs the change in its WAL.
    - The WAL entry is sent to slaves, which replay the entry to update their own data.
  + **Out-of-Sync Slaves:**
    - A slave can request the master for WAL entries created after a specific timestamp (e.g., "send all entries after September 27, 10 PM").
    - This allows the slave to update itself incrementally without fetching all data.

**3. Key Components in Replication**

* **Write-Ahead Logs:**
  + Stores every write operation in chronological order.
  + Called **binary logs** in MySQL and similar terms in other databases.
  + Optimized for machine processing, not human-readable.
* **B+ Trees:**
  + Databases use **B+ trees** to store table data and indexes for efficient search operations.
  + WAL and B+ trees work together to ensure data consistency.

**4. Replication Flow**

1. **Read Operations:**
   * Reads do not change data; hence, they do not affect replication.
   * Reads can happen on any slave or the master.
2. **Write Operations:**
   * Only performed on the master.
   * Updates are recorded in the master’s WAL and applied to the master’s tables.
   * The master sends WAL entries to the slaves to replicate the changes.
3. **Slave Synchronization:**
   * Slaves keep themselves synchronized by:
     + Listening for new WAL entries from the master.
     + Requesting missing entries after a timestamp if they go out of sync.
   * Synchronization involves transferring **incremental changes**, not the entire dataset, optimizing resource usage.

**5. Advantages of Using WAL**

* **Fault Recovery:**
  + If tables are lost, WAL can be replayed to recover the database state.
* **Efficient Synchronization:**
  + WAL allows slaves to sync incrementally, reducing the load on the master and network.
* **Durability:**
  + Ensures that all writes are durably logged before being applied to tables.

## System Types Based on Trade-offs (CAP Theorem)

1. **Inconsistent but Highly Available and Fast**:
   * Prioritizes speed and availability over consistency.
   * Writes are **asynchronous** to slaves.
   * **Risk**: If the master fails before syncing with slaves, data loss can occur.
   * Use cases: Social media metrics (e.g., live viewer counts).
2. **Eventually Consistent**:
   * Prioritizes availability with consistency achieved **over time**.
   * Write to the master and **at least one slave**, then return success.
   * Sync the rest of the slaves **in the background**.
   * **Risk**: Brief periods of inconsistency before synchronization.
   * Use cases: Social media posts, user profile updates.
3. **Highly Consistent**:
   * Ensures all nodes are fully synchronized before returning success.
   * Write to the master and **all slaves**, wait for acknowledgments.
   * **Drawbacks**: Slower performance, reduced availability during network partitions or failures.
   * Use cases: Banking systems, financial transactions.

**How Master and Slaves Are Kept in Sync**

1. **Write Ahead Logs (WAL)**:
   * Each write (insert, update, delete) creates a log entry in the **Write Ahead Log (WAL)** or **binary log**.
   * WAL ensures:
     + Durability: Data can be recovered by replaying logs.
     + Synchronization: Logs are shared with slaves to replicate changes.
   * Each machine (master/slave) has its **local WAL file**.
2. **Replication Methods**:
   * **Asynchronous Replication**:
     + Master writes to WAL and returns success immediately.
     + Changes are pushed to slaves **later in the background**.
     + **Risk**: Data loss if the master fails before syncing.
   * **Synchronous Replication**:
     + Master writes to WAL and syncs with all slaves **before** returning success.
     + Guarantees consistency but increases latency.
3. **Incremental Sync**:
   * If a slave falls out of sync (e.g., due to disconnection):
     + The slave requests updates starting from the last **timestamp** it processed.
     + Only the **delta logs** are sent to minimize overhead.
4. **Handling Failures**:
   * If the master fails:
     + A slave is promoted to a new master.
     + Remaining slaves sync with the new master.
   * For high availability, it’s critical to periodically back up WAL files.

**Examples of Use Cases**

* **Fast but Inconsistent**:
  + Counting YouTube subscribers or live viewers.
  + Reads return immediately, even if slaves are out of sync.
* **Eventually Consistent**:
  + Updating user posts or profiles on social media.
  + Ensures data consistency after some delay.
* **Highly Consistent**:
  + Financial transactions.
  + Requires strong guarantees, such as atomicity across all nodes.

## Banking and Financial Systems

**Why Can't We Use "Eventually Consistent" Systems for Banking?**

1. **Scenario**:
   * A user (e.g., Sahil) withdraws ₹1000 from his bank account:
     + The master updates the balance (-₹1000) and replicates it **asynchronously** to slaves.
   * Simultaneously, another withdrawal request reads from a slave that is **not yet updated**:
     + The slave reports the original balance (₹1000), leading to a potential overdraft if the withdrawal is allowed.
   * Result: Data inconsistency and financial loss for the bank.
2. **Requirement for Banking Systems**:
   * Banking systems must ensure **strong consistency** to prevent such scenarios.
   * All nodes must have the same updated balance **before allowing further transactions**.

**Key Properties of Highly Consistent Systems**

1. **Write Operation Workflow**:
   * Write requests (e.g., withdrawal) always go to the **master**.
   * The master updates its local state and replicates the changes to all slaves **synchronously**.
   * The system returns success **only when all nodes confirm the update**.
   * If any slave fails to update, the transaction is **rolled back** to maintain consistency.
2. **Read Operations**:
   * Reads can be directed to any slave, as long as all nodes are in sync.
   * This allows load balancing for read-heavy systems.
3. **Single Point of Failure**:
   * If the master node fails:
     + Write operations stop until a new master is elected.
     + Read operations can continue using the slaves.

**Master Failure and Slave Promotion**

1. **Handling Master Failures**:
   * When the master fails:
     + A new master is selected from the slaves.
     + The slave with the **most recent updates** (highest timestamp) is typically chosen.
   * Until the new master is elected, **write operations remain unavailable**.
2. **Master Selection Process**:
   * The election process usually takes **1–2 seconds**.
   * During this period:
     + Reads from slaves continue.
     + Writes are temporarily paused.
3. **Role of Zookeeper**:
   * A **coordination service** like Zookeeper is used for:
     + Monitoring the master node's health.
     + Electing a new master from available slaves.
     + Managing metadata and ensuring a consistent state across the system.

**Alternatives for Write Operations in Highly Consistent Systems**

1. **Write to Any Slave**:
   * Hypothetically, writes could go to any slave.
   * However, the slave would then need to propagate the update to the master and other slaves.
   * **Challenges**:
     + Slower as the slave must communicate with all nodes.
     + Adds complexity to the synchronization process.
2. **Broadcast Writes Directly to All Nodes**:
   * In highly consistent systems, the master broadcasts writes directly to all nodes.
   * This approach minimizes latency by avoiding intermediary steps.

**Why Synchronization Matters**

1. **Rollback for Consistency**:
   * If any slave fails to update, the system rolls back the transaction and reports an error (e.g., "bank unavailable").
   * Ensures that no partial updates occur, maintaining data integrity.
2. **Concurrency Issues**:
   * Multiple concurrent transactions can lead to conflicts if nodes are not synchronized.
   * A consistent system ensures serializability (transactions appear to occur one after the other).

## Introduction to NoSQL Databases

**Relational Databases (RDBMS)**

1. **Key Concepts**:
   * Data is stored in **tables** with well-defined **schemas**.
   * Relationships between tables are modelled through **foreign keys**.
   * Data normalization minimizes redundancy.
2. **Features**:
   * **ACID Transactions**: Ensure data integrity and consistency.
   * **Joins**: Enable complex queries by linking tables.
   * **Fixed Data Types**: Columns have predefined data types and sizes.
3. **Challenges in Some Use Cases**:
   * **Handling Diverse Categories**:
     + Example: An e-commerce platform like Amazon has thousands of categories, each with unique features.
       - T-Shirts: Attributes like size, material, sleeve length, collar type, etc.
       - Laptops: Attributes like RAM, CPU frequency, OS, refresh rate, etc.
     + In an RDBMS:
       - Separate tables are required for each category.
       - Adding new categories or attributes can lead to schema rigidity and inefficiency.
   * **Using JSON for Flexible Attributes**:
     + Storing attributes as a JSON string in a single table column can simplify schema design.
     + **Drawbacks**:
       - Violates **First Normal Form (1NF)** as data is non-atomic.
       - Queries (e.g., "Find T-shirts with collar type 'Polo'") require full-text search, which is **slow** and lacks indexing.
       - Complex string matching introduces errors and inefficiencies.
   * **Sharding Limitations**:
     + Relational databases assume all data resides on a **single machine**.
     + Sharding (splitting data across machines) breaks this assumption, making operations like **joins** across shards infeasible.
       - Example: In a sharded system, generating a **news feed** (joining posts, users, and friends) requires querying multiple machines.
   * **Variable-Sized Data**:
     + Relational databases prefer fixed-size columns (e.g., integers, VARCHAR with limits).
     + Real-world data often involves **variable sizes** (e.g., short vs. long text), which RDBMS handles inefficiently.

**Introduction to NoSQL Databases**

1. **Why NoSQL?**
   * Designed to handle the limitations of relational databases in certain scenarios:
     + **Unstructured or Semi-Structured Data**: NoSQL databases are schema-less, making them flexible for diverse data types.
     + **Scalability**: Built to scale horizontally across multiple machines.
     + **Denormalization**: Encourages storing data in fewer collections/tables to optimize retrieval.
2. **Key Concepts in NoSQL**:
   * Schema-less: Data does not require a fixed schema.
   * Data is distributed across multiple nodes, ensuring scalability.
3. **When to Use NoSQL?**
   * Applications with highly variable schemas (e.g., e-commerce with diverse product categories).
   * High-performance applications requiring horizontal scalability (e.g., social media, IoT systems).
   * Use cases where normalization is less critical, and data can be denormalized for faster reads.

RDBMS vs. NoSQL: A Comparison

|  |  |  |
| --- | --- | --- |
| **Feature** | **RDBMS** | **NoSQL** |
| **Schema** | Fixed, predefined | Schema-less |
| **Scalability** | Vertical (adding resources to a single machine) | Horizontal (adding more machines) |
| **Joins** | Supported within the same database | Generally, not supported across collections |
| **Normalization** | Encouraged | Often denormalized for performance |
| **Data Storage** | Structured (tables, rows, columns) | Flexible (key-value, document, columnar, graph) |
| **ACID Transactions** | Strongly supported | Varies depending on implementation |
| **Use Cases** | Financial systems, ERP, CRM | E-commerce, social media, IoT |

**NoSQL Database Formats:**

* **Key-Value Stores:**
  + Simplest form of NoSQL, storing data as a key-value pair.
  + Example: **Redis**:
    - Key is a string.
    - Value can be strings, sorted sets, lists, hashes, or other structures.
    - Flexible and schema-less: New keys and values can be added dynamically.
    - Variable value sizes: Values can range from a few bytes to megabytes.
    - Persistent storage: Optional feature to persist data to disk.
* **Document Stores:**
  + Stores data as structured documents (e.g., JSON or BSON).
  + Example: **MongoDB** (discussed in the next lecture).
* **Column-Family Stores:**
  + Data is organized in columns rather than rows, allowing for efficient read/writes of specific columns.
  + Example: **Cassandra**.
* **Graph Databases:**
  + Focus on relationships between entities using nodes and edges.
  + Example: **Neo4j**.

**Redis as an Example of a NoSQL Database:**

* **Characteristics:**
  + Schema-less: Keys and values are defined on the fly.
  + No fixed format: Values can be strings, hashes, or other types.
  + Variable size: Handles small and large values efficiently.
* **Use Cases:**
  + Simple key-value storage.
  + Caching due to fast access speeds.
  + Real-time data tracking (e.g., leaderboards, session data).