HLD - SQL vs NoSQL

Contents

[Key Concepts of Sharded, Schema-less Databases 2](#_Toc184127508)

[ACID Properties and NoSQL Databases 4](#_Toc184127509)

[Types of NoSQL Databases 6](#_Toc184127510)

[NoSQL Database Schema and Storage Patterns 8](#_Toc184127511)

[NoSQL Database Schema 10](#_Toc184127512)

[NoSQL Database Selection for Use Cases 13](#_Toc184127513)

[Sharding in NoSQL Databases 17](#_Toc184127514)

[Uber Sharding and API Design 19](#_Toc184127515)

[Banking System Sharding 22](#_Toc184127516)

[Sharding in Slack-like Systems 24](#_Toc184127517)

## Key Concepts of Sharded, Schema-less Databases

**1. Problem with Traditional Relational Databases**

* Relational databases like **MySQL** and **PostgreSQL** do not natively support sharding.
  + **Sharding**: Splitting data across multiple machines to handle large datasets that can't fit on a single machine.
  + Sharding in relational databases is manual:
    - Developers implement a custom **sharding layer**.
    - Example: Partitioning data such that different user groups are stored in separate database instances.
  + Limitations:
    - **ACID properties** (Atomicity, Consistency, Isolation, Durability) can't be guaranteed across shards.
    - Joins across shards are complex or impossible.

**2. Goals for NoSQL Databases**

NoSQL databases address limitations of relational databases with the following principles:

**(a) Sharding Out-of-the-Box**

* Native support for **sharding**:
  + Database manages sharding internally.
  + Developers only need to specify a **sharding key**.
  + System handles data distribution and retrieval across machines.
  + Adding machines automatically increases capacity.

**(b) Schema-Less Design**

* Flexibility to store **variable-sized data** without predefined structures:
  + Fields can have different sizes:
    - Example: Storing resumes (small vs. large content).
  + No need to define a rigid schema with fixed column types.
  + Supports objects of different sizes dynamically.

**(c) Eliminating Traditional Relational Constraints**

* Focus on **document storage** or **key-value stores**:
  + Example: Storing JSON documents, where each document can have a different structure or number of fields.
* Avoid **tables and joins**:
  + Simplifies the system by focusing on direct data access.
  + Reduces the complexity of cross-table or cross-machine queries.

**3. Examples of Usage Scenarios**

* **Variable Number of Fields**:
  + Dynamic addition of new fields without prior schema definition.
  + Example: JSON-based document databases.
* **Large Binary Data**:
  + Relational databases struggle with large binary objects (BLOBs), while NoSQL databases handle them efficiently.
  + Example: Storing blog content or multimedia files.

**4. Key Benefits of NoSQL Databases**

* Built for **distributed systems**:
  + Designed with horizontal scalability in mind.
  + Efficient handling of large-scale datasets.
* Enhanced **developer productivity**:
  + Schema flexibility removes the overhead of rigid schema management.
  + Sharding support simplifies scaling efforts.

**5. Trade-offs of NoSQL Databases**

* Limited or no **ACID properties** across shards.
* No support for relational features like complex joins.

**6. Examples of NoSQL Database Models**

* **Document-oriented databases**: JSON-based storage (e.g., MongoDB).
* **Key-value stores**: Simple key-value pairs (e.g., Redis, DynamoDB).

## ACID Properties and NoSQL Databases

**1. ACID Properties Overview**

* **ACID**: A set of properties ensuring reliable database transactions.
  + **A - Atomicity**: Transactions are "all or nothing."
    - Either all operations in a transaction are completed, or none are.
  + **C - Consistency**: Database remains in a valid state after a transaction.
    - Ensures data integrity.
  + **I - Isolation**: Concurrent transactions do not interfere with each other.
    - Results are as if transactions were executed sequentially.
  + **D - Durability**: Once a transaction is committed, it remains permanent, even in case of system crashes.

**2. Relational Databases and ACID**

* Databases like **MySQL** and **PostgreSQL** guarantee all four ACID properties.
* ACID compliance ensures robust, reliable transactions, especially in environments where data integrity is critical (e.g., financial systems).

**3. NoSQL Databases and ACID**

* NoSQL databases do **not guarantee all ACID properties**.
  + They relax one or more properties to prioritize other factors like performance, scalability, or flexibility.
  + Each NoSQL database defines its guarantees explicitly.

**Redis Example:**

* **Redis** is an in-memory, key-value store often used for caching.
  + **Durability**: Not guaranteed by default.
    - Data is stored in RAM and is lost upon system restart.
    - Durability can be partially achieved using the **persistence flag**:
      * Periodic dumps (e.g., every 10 minutes) store data to disk.
      * However, data generated between the last dump and a crash is lost.
    - Hence, Redis is not fully durable.
  + **Other ACID properties**:
    - Atomicity, Consistency, and Isolation are generally handled since Redis is single-threaded.

**4. Key Differences Between Relational and NoSQL ACID Handling**

* **Relational Databases**:
  + All ACID properties are mandatory.
  + Ensures strict data consistency, which may impact performance.
* **NoSQL Databases**:
  + Developers can choose which properties to prioritize or relax.
  + Trade-offs:
    - **Durability** may be sacrificed for speed (e.g., Redis).
    - **Consistency** might be relaxed in favour of availability (CAP theorem considerations).

**5. Summary**

* ACID compliance is a hallmark of traditional relational databases, ensuring reliability and data integrity.
* NoSQL databases relax ACID guarantees depending on their use case:
  + Example: Redis focuses on speed and in-memory operations, sacrificing durability.
* Developers must understand the specific guarantees and trade-offs of each NoSQL database to use them effectively.

## Types of NoSQL Databases

A diagram of a diagram of a family and document

Description automatically generatedNoSQL databases can be broadly classified based on their storage patterns:

1. **Key-Value Databases**
   * **Structure**: Simple key-value pairs.
     + Example: key: "userID\_123", value: "{name: 'John', age: 25}"
   * **Use Case**: High-speed, lightweight data storage and retrieval.
   * **Examples**: Redis, DynamoDB, Riak.
   * **Advantages**:
     + Very fast due to simple structure.
     + Ideal for caching, session storage, or scenarios requiring constant lookups.
   * **Limitations**:
     + Not suitable for complex queries.
     + Lacks relationships between data.
2. **Column-Family Databases**
   * **Structure**: Data is stored in a **column-oriented** format.
     + Organized by rows and columns, but columns can vary per row.
     + Example: Similar to a sparse table with flexible column definitions.
   * **Use Case**: Analytical workloads, time-series data, and high write/read scalability.
   * **Examples**: Cassandra, HBase, ScyllaDB.
   * **Advantages**:
     + Optimized for write-heavy workloads.
     + Flexible schema with high scalability.
   * **Limitations**:
     + Complex for applications needing joins or multi-row transactions.
3. **Document Databases**
   * **Structure**: Data is stored in **document format** (e.g., JSON, BSON).
     + Each document can have a unique structure.
     + Example: { "id": 1, "name": "Alice", "attributes": { "age": 30, "city": "NYC" } }
   * **Use Case**: Applications needing schema flexibility and hierarchical data storage.
   * **Examples**: MongoDB, CouchDB.
   * **Advantages**:
     + Schema-less design allows flexibility for dynamic data.
     + Supports nested structures and rich queries.
   * **Limitations**:
     + May lack relational features like multi-document ACID transactions.

**Other Types of NoSQL Databases**

* **Graph Databases** *(Skipped in Detail for Practicality)*:
  + Specialized for graph-based data (nodes, edges, relationships).
  + Examples: Neo4j, ArangoDB.
  + Limited use cases in production due to complexity.
* **Time-Series Databases** *(Not Covered in Detail Yet)*:
  + Specialized for sequential, time-stamped data.
  + Examples: InfluxDB, TimescaleDB.

## NoSQL Database Schema and Storage Patterns

**Types of NoSQL Databases**

1. **Key-Value Databases**
   * **Example**: Redis, DynamoDB
   * **Structure**: A simple map where a unique key points to a value.
   * **Advantages**:
     + Easy to understand and implement (similar to hash maps).
     + High simplicity makes it ideal for certain use cases.
   * **Limitations**:
     + No relationships between keys and values.
     + Inefficient for large or complex data structures; fetching subsets of data (e.g., top 10 messages) can be slow because the entire value must be read.
2. **Column-Family Databases**
   * **Examples**: Cassandra, HBase
   * **Structure**:
     + Somewhere between key-value and table structures.
     + Each key corresponds to a "row," and each row contains multiple "column families."
     + Column families are sorted by timestamps in descending order.
   * **Advantages**:
     + Optimized for specific use cases like messaging systems where you only fetch recent data.
     + Reduces data read operations by allowing partial access to rows (e.g., fetching the most recent 10 messages).
   * **Use Case Example**:
     + Messaging systems: Key = user ID, Column Families = attributes (conversations, messages), enabling efficient retrieval of recent messages or conversations.
   * **Storage Mechanisms**:
     + Uses **LSM Trees** or **sorted sets** for fast reads/writes (to be discussed in later lectures).
3. **Document Databases**
   * **Examples**: MongoDB
   * Structure and applications will be discussed later.

**Key-Value vs. Column-Family Databases**

* **Key-Value**:
  + Best for simple and flat data.
  + Struggles with large datasets due to the need to fetch and parse the entire value.
* **Column-Family**:
  + Ideal for more complex data with sub-keys or grouped columns.
  + Faster access to subsets of data due to sorted storage.

**Key Concepts in Column-Family Databases**

1. **Row and Column Family**:
   * A row is defined by a key (e.g., user ID).
   * A value consists of one or more column families, each storing related data (e.g., user attributes, conversations, messages).
2. **Column Families Structure**:
   * Entries in each column family are sorted by timestamp in descending order.
   * Enables efficient querying of recent entries.
3. **Sharding**:
   * A key-value pair (entire row) resides on a single shard (machine).
   * Rows can be distributed across multiple shards for scalability.
4. **Example Schema for Messaging System**:
   * **Key**: user\_id
   * **Column Families**:
     + **User Attributes**: Metadata like location, gender, updated timestamps.
     + **Conversations**: Stores recent conversations, sorted by timestamp.
     + **Messages**: Messages grouped by conversation ID, also sorted by timestamp.

## NoSQL Database Schema

1. **Key-Value Store Schema**

* **Structure:**
  + Data is stored as simple key-value pairs.
  + Example:

|  |
| --- |
| Key: "user123"  Value: {"name": "Alice", "age": 25} |

* **Schema Design Principles:**
  + **Flat structure:** Avoid nesting or complex data relationships.
  + **Use meaningful keys:** Keys should be designed for efficient lookups.
    - Example: "userID123" for user-related data.
  + **Denormalization:** Store data in a way that optimizes reads since there are no relations between keys.
* **Limitations:**
  + Cannot handle complex queries efficiently.
  + Fetching partial data requires retrieving the entire value.

1. **Column-Family Store Schema**

* **Structure:**
  + Combines aspects of key-value stores and tabular databases.
  + Data is organized by **rows** and **column families.**
    - Each row is identified by a unique key.
    - Each key points to multiple **column families** containing related data.
  + Example (Messaging System):

|  |
| --- |
| Row Key: "user123"  Column Families:    - "Attributes": {"location": "Bangalore", "gender": "Female"}    - "Conversations": {"convID1": "Last Msg", "convID2": "Last Msg"}    - "Messages": {"msg1": "Hello", "msg2": "Hi"} |

* **Schema Design Principles:**
  + **Group related data:** Use column families to organize data into logical groups.
    - Example: Store user metadata, conversations, and messages in separate column families.
  + **Optimize for queries:** Design schema based on access patterns (e.g., fetching recent messages).
  + **Use timestamps for sorting:** Store entries in descending timestamp order for efficient retrieval of recent data.
* **Storage Mechanism:**
  + **Sharding:** Distribute rows across shards for scalability.
  + **Limit row size:** Ensure each row fits on a single machine.
* **Use Cases:**
  + Messaging systems:
    - Key = user ID.
    - Column Families = attributes (e.g., conversations, messages).
    - Efficient retrieval of recent messages using timestamp ordering.

1. **Document Database Schema**

* **Structure:**
  + Documents are typically stored in JSON format.
  + Each document is self-contained and can have a unique structure.
  + Example:

|  |
| --- |
| {      "user\_id": "123",      "name": "Alice",      "address": {        "city": "Bangalore",        "pincode": 56100      },      "hobbies": ["reading", "traveling"]  } |

* **Schema Design Principles:**
  + **Flexible structure:** No enforced schema; each document can have different fields.
    - Example:
      * Document 1: {"name": "Alice", "age": 25}
      * Document 2: {"name": "Bob", "city": "Mumbai", "hobbies": ["reading"]}
  + **Embed related data:** Store nested data within a single document when relationships are simple and frequently accessed together.
    - Example: Embed address details within a user document.
  + **Use unique identifiers:** Every document should have a unique key (e.g., user\_id).
  + **Denormalization for reads:** Store data redundantly to optimize query performance.
* **Query Capabilities:**
  + Query based on specific key-value conditions.
  + Example: Fetch documents where "city": "Bangalore".
* **Challenges:**
  + Optimizing performance for large datasets.
  + Designing indexes for fast lookups.
  + Sharding for distributing documents across multiple nodes.

Comparison of NoSQL Schema Types

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Key-Value Store** | **Column-Family Store** | **Document Store** |
| **Schema Flexibility** | Fixed (Key-Value format) | Grouped by column families | Fully flexible JSON |
| **Use Case** | Simple, flat data | Structured data with timestamps | Nested/complex data |
| **Query Capability** | Key-based lookups | Partial row reads (specific cols) | Key-value or condition |
| **Optimization Focus** | Key design | Column family organization | Indexing, sharding |
| **Example Use Case** | Caching | Messaging systems | User profiles, catalogues |
| **Popular Databases** | Redis, DynamoDB | Cassandra, HBase | MongoDB, Elasticsearch |

## NoSQL Database Selection for Use Cases

**Key Concepts in Database Selection**

1. **Database Types and Simplification Rule**:
   * **Key-Value**: Simplest and fastest for flat, single-purpose data storage.
   * **Column-Family**: Efficient for grouped or hierarchical data with multiple related attributes.
   * **DocumentDB**: Best for flexible, schema-less storage where JSON-like structures are required.
   * **Rule of Thumb**:
     + If multiple databases fit, choose the simplest one (Key-Value).
     + Only increase complexity when additional features or relationships are required.
2. **DocumentDB and JSON Storage**:
   * Stores JSON objects as individual entries (akin to files).
   * **Query Mechanism**:
     + Scans all JSON objects for a specific key-value pair (e.g., type = "taxi").
     + Can handle varied structures, but query optimization depends on developer-defined fields.
   * **Sharding**:
     + If JSON objects include a common field like ID, sharding becomes efficient.
     + Without a common field, sharding is complex and relies on the database's internal mechanisms.
3. **Key Differences**:
   * **Key-Value**:
     + Extremely fast lookups by key.
     + Limited capability for relationships or advanced querying.
   * **Column-Family**:
     + Structured grouping of data into column families.
     + Optimized for queries on subsets of grouped data.
   * **DocumentDB**:
     + Schema-less flexibility for storing and querying hierarchical data (e.g., JSON).
     + Requires developer-defined conventions for indexing and optimization.

**Use Case Analysis and Database Suggestions**

1. **Live Match Scores**:
   * **Requirement**: Only store and retrieve the latest scores for multiple matches.
   * **Suggested Database**: **Key-Value**.
     + Key = Match ID, Value = Score (e.g., 267/9 or 36/2 in 8 overs).
     + Justification: Simple flat data with a unique key for each match.
   * **Additional Requirement** (e.g., ball-by-ball details):
     + Shift to **Column-Family** for structured data like player scores, fall of wickets, or ball-by-ball commentary.
2. **Ball-by-Ball Match Details**:
   * **Requirement**: Store detailed information for every ball, including special events (e.g., boundaries, wickets).
   * **Suggested Database**: **Column-Family**.
     + Key = Match ID, Column Families = {Ball Details, Player Scores, Fall of Wickets}.
     + Justification: Enables efficient querying and retrieval of grouped details (e.g., all balls in a specific over).
3. **Times of India Live Score Widget**:
   * **Requirement**: Only show live match scores (no detailed stats or commentary).
   * **Suggested Database**: **Key-Value**.
     + Key = Match ID, Value = Current Score.
     + Justification: Simple display without additional queries or relationships.
4. **Hashtag Storage System for Twitter**

**Use Case**

* **Scenario**: Storing and retrieving hashtags and their associated tweet IDs.
* **Requirements**:
  + A **hashtag** can map to **multiple tweet IDs**.
  + The system should handle high write volumes for popular hashtags (many tweets added in real-time).
  + Efficiently retrieve the **most recent tweets** for a given hashtag.

**Database Options Considered**

1. **Key-Value Database**:
   * **How it works**:
     + Key = Hashtag, Value = List of associated Tweet IDs.
   * **Issues**:
     + Inefficient for write-heavy operations:
       - Every update involves reading the entire list, appending new tweet IDs, and rewriting the updated list.
     + Poor performance for highly popular hashtags with a large number of tweet IDs.
   * **Conclusion**: Not suitable for this use case.
2. **Column-Family Database**:
   * **How it works**:
     + Key = Hashtag, Column Family = Tweet IDs (individual tweet IDs as separate entries within the column family).
     + Supports appending new entries without rewriting the entire dataset.
   * **Advantages**:
     + Optimized for high write throughput.
     + Efficient retrieval of subsets (e.g., most recent 10 tweet IDs).
   * **Conclusion**: Best fit for this use case due to efficient write and read operations.
3. **DocumentDB**:
   * **How it works**:
     + Stores hashtags and their associated tweet IDs as documents (e.g., JSON objects).
   * **Issues**:
     + Similar to Key-Value, updates involve reading and rewriting the entire document, making it less efficient for high write volumes.
   * **Conclusion**: Less suitable than Column-Family for this use case.
4. **Relational Database**:
   * **How it works**:
     + Uses tables to map hashtags to tweet IDs (e.g., Hashtag table and TweetHashtagMapping table).
   * **Issues**:
     + Poor scalability for write-heavy workloads (due to joins and table locking during updates).
     + High latency for retrieving subsets of tweet IDs.
   * **Conclusion**: Not suitable for real-time systems like Twitter.

**Best Choice: Column-Family Database**

* **Reasoning**:
  + Efficient write operations for appending tweet IDs to hashtags.
  + Scalable for high-volume hashtags.
  + Supports querying subsets of tweet IDs quickly (e.g., most recent tweets).
  + Suitable for real-time systems like Twitter where speed and scalability are critical.

**How the System Works in Column-Family DB:**

1. **Schema**:
   * **Key**: Hashtag (e.g., #covid-19).
   * **Column Family**: Tweet IDs associated with the hashtag, stored as individual columns.
2. **Operations**:
   * **Insert**: New tweet IDs are added to the column family for the given hashtag.
   * **Query**: Retrieve a subset of tweet IDs (e.g., the 10 most recent ones).
   * **Scalability**: Handles high write volumes efficiently.

## Sharding in NoSQL Databases

**Introduction to Sharding**

* **Sharding**: A method of partitioning data across multiple machines to enable scalability and improve performance.
* **Sharding Key**: The primary attribute used to determine which shard (machine) will store a particular piece of data.

**Key Principles of Sharding**

1. **Sharding Key Role**:
   * All data related to a particular sharding key must reside on a single machine.
   * Enables efficient retrieval and updates without the need to query multiple machines.
2. **Challenges with Joins**:
   * Joins across shards are expensive and typically avoided.
   * Instead, application servers may fetch data from multiple shards and process joins locally.
   * For large-scale operations across shards, frameworks like **MapReduce** are used for asynchronous processing.

**Process for Choosing a Sharding Key**

1. **List Frequent Use Cases**:
   * Identify the most common operations (queries, writes, reads) in the system.
   * Example:
     + For a **bookmarking application** like Delicious:
       - **Add a bookmark** (User ID + URL).
       - **Retrieve all bookmarks for a user**.
2. **Analyse Entities**:
   * Identify the entities in your system.
   * Determine how these entities relate to the use cases.
3. **Choose the Sharding Key**:
   * Select an entity that ensures the majority of use cases can be served by accessing a single shard.
   * If no perfect entity exists, choose the one that minimizes cross-shard queries.

**Example: Delicious Bookmark Application**

* **Entities**:
  1. **User ID**: Represents a user in the system.
  2. **Site URL**: Represents the bookmarked website.
* **Possible Sharding Key Options**:
  1. **Site URL as Sharding Key**:
     + All bookmarks for a particular site (e.g., google.com) are stored on one machine.
     + **Issues**:
       - When retrieving all bookmarks for a user, the query spans multiple shards (inefficient).
     + **Conclusion**: Not suitable for the use case.
  2. **User ID as Sharding Key**:
     + All bookmarks for a user are stored on one machine.
     + **Advantages**:
       - Adding a bookmark involves a single shard (efficient).
       - Retrieving all bookmarks for a user involves a single shard (efficient).
     + **Conclusion**: Optimal choice.

## Uber Sharding and API Design

**1. Main Uber Use Case:**

* **Requesting a Taxi:**
  + User gives their **user ID** and **location**.
  + The system finds the **nearest available taxis** (usually 5) and sends requests to each driver.
  + Drivers can **accept** or **reject** the request.
* **Booking a Taxi:**
  + Once a driver accepts, the user can book the taxi.
  + User sends their **user ID** and **driver ID** to **book the taxi**.
* **Tracking a Taxi:**
  + After booking, the user can track their taxi with **user ID**.
* **Driver-Side Operations:**
  + Drivers register their **taxi** as **available**.
  + Drivers can **accept** or **reject** bookings.
  + Drivers send **driver ID** and **current location**.

**2. Sharding Considerations:**

**Key Shard ID Choices:**

* **City ID**:
  + Splitting by **city ID** works well for location-based services like Uber because users in one city care only about local drivers.
  + For instance, **Mumbai**'s data would be handled by one machine, **Delhi**'s by another.
  + **Pros**:
    - **Efficient locality**: All data (user, cab) for a city stays together.
    - Avoids cross-shard queries for most requests.
  + **Cons**:
    - Limited flexibility if users need to access data across cities (but this is rare in Uber’s case).
* **Driver ID**:
  + If sharding by **driver ID**, there would be a mix of drivers from various cities in the same shard.
  + **Problem**: It’s hard to get nearest cabs because a query would have to span multiple machines.
* **User ID**:
  + If sharding by **user ID**, the nearest cabs might be stored in different shards, which would increase the complexity of the query.
  + **Problem**: Like driver ID, this doesn’t ensure locality and makes it harder to find nearby taxis quickly.

**Final Choice for Uber**: **City ID** as the sharding key.

**3. Handling Data:**

* **User Data**:
  + User details (name, account info) are stored separately, ideally in a **user DB**, which could be sharded by **user ID**.
  + This can be **cached** for faster access and can be updated as needed (e.g., after a trip).
* **Cab Data**:
  + For each **city**, you store active **taxi** information (driver ID, cab model, location, active status).
  + This data can be stored **locally** in the machine responsible for that city shard.
* **Caching**:
  + **Global caching** solutions (e.g., Redis) can be used to store user details or any frequently accessed data to minimize database calls.

**4. API Flow (User and Driver Interactions):**

* **User Request Flow:**
  1. **Nearest Taxi Request**:
     + User provides **location**.
     + Query the relevant **city ID shard** to fetch the nearest taxis.
     + Return the available taxis to the user.
  2. **Book Taxi**:
     + Once a driver accepts, the booking is recorded with **user ID** and **driver ID**.
     + If necessary, additional details (like user profile) can be fetched from the **user DB**.
  3. **Track Taxi**:
     + After booking, the user can request **live tracking**.
     + The location and status of the taxi are fetched from the **city DB**.
  4. **Trip History**:
     + When the user requests their **trip history**, the system will query the **user DB** for trips related to the user.
* **Driver Request Flow:**
  1. **Register Availability**:
     + Driver provides **location** and sets status as available.
     + This updates the **city DB** for that city.
  2. **Accept or Reject Booking**:
     + Driver either accepts or rejects the user’s booking request.
     + Data about the trip (if accepted) is updated in the system.

**5. Sharding Impact on Latency:**

* **Sharding by City ID** ensures locality and reduces cross-shard queries.
* Queries that span across cities (e.g., for intercity travel) are avoided, reducing complexity and latency.
* Storing data **relevant to each city** locally makes it easier to scale horizontally (i.e., adding new machines for new cities).

## Banking System Sharding

**Core Concepts:**

* **User and Accounts**: A user can have multiple accounts (e.g., Savings, Current), and each account has a unique account ID.
* **Data Needs**: Users typically need to access:
  + Balance of a specific account
  + Transaction history (e.g., last 100 transactions)
  + Perform actions like withdrawing or depositing money
* **Sharding Challenge**: The goal is to shard the database efficiently while ensuring consistency, availability, and scalability.

**Sharding Strategies:**

1. **Account ID as Sharding Key**:
   * Initially considered but flawed because a user can have multiple accounts. If an account's data is split across multiple shards, consistency issues can arise.
   * If the user logs in and views all their accounts, inconsistencies could appear if the account data is not available on the same shard.
2. **User ID as Sharding Key**:
   * A better solution because it ensures all data for a specific user (including all accounts and transactions) is located on a single shard.
   * This prevents issues of inconsistent data between shards and allows atomic operations (e.g., adding an account or querying transactions for a user).
   * **Sharding by User ID** means a shard will contain multiple users’ data, but a user’s data will remain on the same shard, maintaining consistency.
   * **Consistency**: If a user is linked to multiple accounts, all account details (balances, transactions) will be stored together in one shard, reducing the risk of inconsistency.
   * **Scalability**: Multiple users can be distributed across several shards. For example, if there are 3 shards, each will hold data for a third of users.
3. **City ID as Sharding Key**:
   * If sharded by city, all accounts from one city (e.g., Mumbai) would go to one shard, and accounts from another city (e.g., Delhi) would go to a different shard.
   * **Drawback**: A user could have accounts in different cities, which would force data from multiple shards to be accessed. This increases latency and complexity.
4. **Timestamp as Sharding Key**:
   * Timestamp-based sharding was suggested, where recent transactions (e.g., last 12 hours) are stored on one shard, and older transactions are stored on others.
   * **Drawbacks**:
     + **Skewed load**: Most traffic will be directed to the shard containing recent transactions, leading to load imbalance.
     + **High Latency for Inactive Users**: If a user hasn’t made transactions recently, their data may be scattered across multiple shards, leading to increased query time and latency.
     + **Consistency and Aggregation Issues**: When a user’s transactions span multiple time periods, aggregating data to compute things like the balance will involve accessing several shards, leading to inefficiency and latency.

**Key Considerations for Sharding:**

* **Consistency**: Ensuring that data for a user’s accounts and transactions are all available in one place (ideally on the same shard).
* **Scalability**: Sharding must allow the system to scale horizontally without sacrificing performance.
* **Atomicity**: Database operations should be atomic within a single shard to maintain consistency when performing operations like withdrawals and deposits.

## Sharding in Slack-like Systems

**Primary Feature of Slack:**

* **Group communication**: Users send messages to channels, which can have many members (e.g., 10,000 users).
* **Operations**:
  + **Send message**: Given a channel ID and a message, the message is sent to all members of the channel.
  + **Get channels**: Retrieve all channels a user is part of.
  + **Get messages**: Retrieve messages for a particular channel.
  + **Get unread messages**: Get unread message count for a channel.
  + **Direct Messages**: Users can also send messages to other individual users in direct conversations.

**Sharding in Slack-like Systems**

When considering sharding for Slack, we need to ensure scalability and fast access to frequently accessed data while maintaining consistency and minimizing latency. Let's look at various sharding strategies discussed:

**Sharding Key Considerations:**

1. **Organization ID**:
   * **Pros**:
     + For each organization, messages, channels, and users will typically fit on a single machine. Sharding by organization ID can work efficiently if the data fits within the capacity of one machine (e.g., thousands of users, thousands of messages).
     + Each organization is a distinct unit, so sharding by organization can also logically separate data.
   * **Cons**:
     + The user-to-organization mapping needs to be managed separately, possibly in a different database.
     + If a user is part of multiple organizations, we need to ensure proper handling of their different organization memberships.
2. **Channel ID**:
   * **Pros**:
     + Each channel could have a separate shard, making it easier to distribute the load.
   * **Cons**:
     + A user could be part of multiple channels, meaning that user data would be spread across multiple shards.
     + For a single message, updates need to be made across multiple shards, complicating the system.
3. **User ID**:
   * **Cons**:
     + Sharding by user ID would cause issues when sending a message to a channel with thousands of users. The message would need to be written to the shards of every user in that channel, leading to slow operations.
     + Channels with large numbers of users (e.g., 50,000) would result in significant delays in message propagation.

**Additional Considerations:**

1. **User Profile and Organization Info**:
   * User login and profile information may be stored in a separate database (sharded by user ID or organization ID), while channel and message data could be sharded by organization ID or channel ID.
   * Upon login, users could select the organization they're currently working with, which would determine the relevant shard for further queries.
2. **Handling Media**:
   * Media (e.g., images) should not be stored in the main database. Instead, they are stored in object storage and a CDN, with URLs stored in the database.
3. **Shard Size**:
   * A shard typically fits on a single machine. For example, if the shard is 1 terabyte, it would likely be acceptable. However, if the shard becomes too large (e.g., 10 terabytes), it might require special handling or division into smaller shards.
4. **Avoid Sharding by Timestamp**:
   * Sharding by timestamp (e.g., storing data for the last 12 hours in one shard) is not ideal:
     + **Load imbalance**: Shards containing the most recent data will receive a disproportionate amount of traffic, while older shards will experience less load.
     + **Increased latency**: For inactive users or older data, retrieving information across multiple shards leads to higher latency.
     + **Consistency challenges**: Aggregating data across multiple time-based shards can slow down calculations (e.g., bank balance or aggregate transaction history).

**Conclusion:**

* **Best Sharding Strategy**: Sharding by **organization ID** seems to be the best approach for Slack-like systems because:
  + It logically groups users and channels under an organization, allowing efficient access to the data of a specific organization.
  + It minimizes the complexity of message delivery across channels and reduces the possibility of inefficient cross-shard communication.
* **Challenges with User ID or Channel ID**:
  + Sharding by user ID is not recommended due to the complexity of updating multiple shards when messages are sent to large channels.
  + Sharding by channel ID can be problematic for users who belong to many channels, requiring data spread across multiple shards.