System Design - SQL vs NoSQL

Agenda

SQL Database

SQL databases are relational databases which consist of tables related to each other and every table has a fixed set of columns. You can query across tables to retrieve related information.

Features of SQL Databases:

Normalization

One of the requirements of SQL databases is to store the data in normalized form to avoid data redundancy and achieve consistency across tables. For example, let's assume two tables are storing a particular score and one of the scores gets changed due to an update operation. Now, there will be two different scores at two different tables leading to confusion as to which score should be used.

Hence, the data should be normalized to avoid this data redundancy and trust issue.

ACID Transactions

ACID stands for Atomicity, Consistency, Isolation and Durability.

- **Atomicity** means that either a transaction must be all or none. There should not be any partial states of execution of a transaction. Either all statements of a transaction are completed successfully or all of them are rolled back.
- **Consistency** refers to the property of a database where data is consistent before and after a transaction is executed. It may not be consistent while a transaction is being executed, but it should achieve consistency eventually.
- **Isolation** means that any two transactions must be independent of each other to avoid problems such as dirty reads.
- **Durability** means that the database should be durable, i.e. the changes committed by a transaction must persist even after a system reboot or crash.

Let's understand this with the help of an example.

Let's say Rohit wants to withdraw Rs 1000 from his bank account. This operation depends on the condition that Rohit's bank balance is greater than or equal to 1000 INR. Hence the withdrawal essentially consists of two operations:

- Check if the bank balance is greater than or equal to 1000 INR.
- If yes, perform a set operation: Balance = Balance - 1000 and disperse cash.

Now, imagine these operations are done separately in an app server. Let's assume that Rohit's bank balance is 1500 INR. And the first operation was completed successfully with a yes. Now,

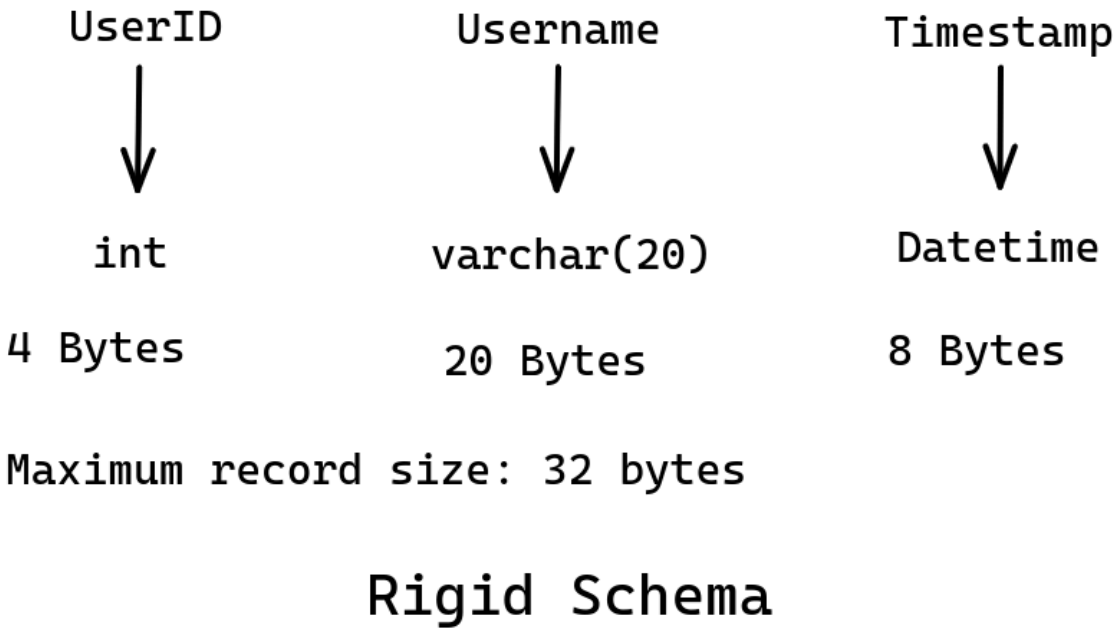
when the second operation is performed, there are chances that some other withdrawal request of 1000 INR has already changed Rohit's bank balance to 500 INR.

Now, if the second operation is performed, it would set Rohit's bank balance to -500 which does not make sense. Hence, if the database does not guarantee atomicity and isolation, these kinds of problems can happen when multiple requests attempt to access (and modify) the same node.

Now, when Rohit makes a request to withdraw 1000 INR from his account, both these operations represent a single transaction. The transaction either succeeds completely or fails. There won't be any race conditions between two transactions. This is guaranteed by a SQL database.

Defined Schema

Each table has a fixed set of columns and the size and type of each column is well-known.



Rigid Schema

However, there are a few features that are **not** supported by a SQL database.

Shortcomings of SQL Databases

Fixed Schema might not fit every use case

Let's design the schema for an ecommerce website and just focus on the Product table. There are a couple of pointers here:

- Every product has a different set of attributes. For example, a t-shirt has a collar type, size, color, neck-type, etc.. However, a MacBook Air has RAM size, HDD size, HDD type, screen size, etc.
- These products have starkly different properties and hence couldn't be stored in a single table. If you store attributes in the form of a string, filtering/searching becomes inefficient.
- However, there are almost 100,000 types of products, hence maintaining a separate table for each type of product is a nightmare to handle.
  SQL is designed to handle millions of records within a single table and not millions of tables itself.

Hence, there is a requirement of a flexible schema to include details of various products in an efficient manner.

Database Sharding nullifies SQL Advantages

- If there is a need of sharding due to large data size, performing a SQL query becomes very difficult and costly.
- Doing a **JOIN** operation across machines nullifies the advantages offered by SQL.
- SQL has almost zero power post sharding. You don't want to visit multiple machines to perform a SQL query. You rather want to get all data in a single machine.

As a result, most SQL databases such as postgresql and sqlite do not support sharding at all.

Given these two problems, you might need to think of some other ways of data storage. And that's where NoSQL databases come into the scenario.

NoSQL Databases

Let's pick the second problem where data needs to be sharded. First step to Sharding is choosing the **Sharding Key**. Second step is to perform **Denormalization**.

Let's understand this with the help of an example.

Imagine a community which exchanges messages through a messenger application. Now, the data needs to be sharded and let's choose **UserID** as the sharding key. So, there will be a single machine holding all information about the conversations of a particular user. For example, M1 stores data of U1 and M2 for U2 and so on.

Now, let's say U1 sends a message to U2. This message needs to be stored at both M1 (sent box) and M2 (received box). This is actually denormalization and it leads to data redundancy. To avoid such problems and inefficiencies, we need to choose the sharding key carefully.

Examples of Choosing a Good Sharding Key

Banking System

Situation:

- Users can have active bank accounts across cities.
- Most Frequent operations:
    - Balance Query
    - Fetch Transaction History
    - Fetch list of accounts of a user
    - Create new transactions

Why is CityID not a good sharding key?

- Since users can move across cities, all transactions of a user from the account in city C1 needs to be copied to another account in city C2 and vice versa.
- Some cities have a larger number of users than others. Load balancing poses a problem.

An efficient sharding key is the UserID:

- All information of a user at one place. Operations can be performed most efficiently.
- For Balance Query, Transaction History, List of accounts of a user, you only need to talk to one machine (Machine which stores info of that user).
- Load balancing can be achieved as you can distribute active and inactive users uniformly across machines.

Note: Hierarchical Sharding might not be a great design as if one machine crashes, all machines under its tree also become inaccessible.

Uber-like System

Situations:

- Most frequent use case is to search for nearby drivers.

CityID seems to be a good sharding key.

- You need to search only those cabs which are in your city. Most frequent use cases are handled smoothly.

DriverID is not a good choice as:

- The nearby drivers could be on any machines. So, for every search operation, there will be a need to query multiple machines which is very costly.

Also sharding by PIN CODE is not good as a cab frequently travels across regions of different pin codes.

Note: Even for inter-city rides, it makes sense to search drivers which are in your city.

Slack Sharding Key (Groups-heavy system)

Situation:

- A group in Slack may even consist of 100,000 users.

UserID is not a good choice due to the following reasons:

- A single message in a group or channel needs to perform multiple write operations in different machines.

For Slack, GroupID is the best sharding key:

- Single write corresponding to a message and events like that.
- All the channels of a user can be stored in a machine. When the user opens Slack for the first time, show the list of channels.
- Lazy Fetch is possible. Asynchronous retrieval of unread messages and channel updates. You need to fetch messages only when the user clicks on that particular channel.

Hence, according to the use case of Slack, GroupID makes more sense.

IRCTC Sharding Key

Main purpose is ticket booking which involves TrainID, date, class, UserID, etc.

Situation:

- Primary problem of IRCTC is to avoid double-booked tickets.
- Load Balancing, major problem in case of tatkal system.

Date of Booking is not a good Sharding Key:

- The machine that has all trains for tomorrow will be bombarded with requests.
- It will create problems with the tatkal system. The machine with the next date will always be heavily bombarded with requests when tatkal booking starts.

UserID is not a valid Sharding Key:

- As it is difficult to assure that the same ticket does not get assigned to multiple users.
- At peak time, it is not possible to perform a costly check every time about the status of a berth before booking. And if we don't perform this check, there will be issues with consistency.

TrainID is a good sharding key:

- Loads get split among trains. For example, tomorrow there will be a lot of trains running and hence load gets distributed among all machines.
- Within a train, it knows which user has been allocated a particular berth.
- Hence, it solves the shortcomings of Date and UserID as sharding keys.

Note: Composite sharding keys can also be a good choice.

Few points to keep in mind while choosing Sharding Keys:

- Load should be distributed uniformly across all machines as much as possible.
- Most frequent operations should be performed efficiently.
- Minimum machines should be updated when the most-frequent operation is performed. This helps in maintaining the consistency of the database.
- Minimize redundancy as much as possible.

Types of NoSQL databases

Key-Value NoSQL DBs

- Data is stored simply in the form of key-value pairs, exactly like a hashmap.
- The value does not have a type. You can assume Key-Value NoSQL databases to be like a hashmap from string to string.
- Examples include: DynamoDB, Redis, etc.
- Redis
    - https://try.redis.io/
    - https://redis.io/docs/
    - https://university.redis.com/

Document DBs

- Document DB structures data in the form of JSON format.
- Every record is like a JSON object and all records can have different attributes.
- You can search the records by any attribute.
- Examples include: MongoDB and AWS ElasticSearch, etc.
- Document DBs give some kind of tabular structure and it is mostly used for ecommerce applications where we have too many product categories.

Link to play with one of the popular Document DBs, **MongoDB**: MongoDB Shell. It has clear directions regarding how to:

- Insert data
- Use find() function, filter data
- Aggregate data, etc.

You can try this as well: MongoDB Playground. It shows the query results and allows you to add data in the form of dictionaries or JSON format.

Column-Family Storage

- The sharding key constitutes the RowID. Within the RowID, there are a bunch of column families similar to tables in SQL databases.
- In every column family, you can store multiple strings like a record of that column family. These records have a timestamp at the beginning and they are sorted by timestamp in descending order.

- Every column family in a CF DB is like a table which consists of only two columns: timestamp and a string.
- It allows prefix searching and fetching top or latest X entries efficiently. For example, the last 20 entries you have checked-in, latest tweets corresponding to a hashtag, posts that you have made, etc.
- It can be used at any such application where there are countable (practical) schemas instead of completely schema less.
- The Column-Family Storage system is very helpful when you want to implement Pagination. That too if you desire to implement pagination on multiple attributes, CF DBs is the NoSQL database to implement.
- Examples include Cassandra, HBase, etc.
- Cassandra
    - https://cassandra.apache.org/_/cassandra-basics.html
    - https://cassandra.apache.org/_/case-studies.html
    - https://cassandra.apache.org/doc/stable/cassandra/architecture/overview.html
    - https://cassandra.apache.org/doc/stable/cassandra/architecture/guarantees.html
    - https://cassandra.apache.org/doc/stable/cassandra/data_modeling/intro.html

Choose a Proper NoSQL DB

Twitter-HashTag data storage

Situation:

- With a hashtag, you store the most popular or latest tweets.
- Also, there is a need to fetch the tweets in incremental order, for example, first 10 tweets, then 20 tweets and so on.
- As you scroll through the application, fetch requests are submitted to the database.

Key-Value DB is not a good choice.

- The problem with Key-Value DB is that corresponding to a particular tweet(key), all the tweets associated with that tweet will be fetched.
- Even though the need is only 10 tweets, the entire 10000 tweets are fetched. This will lead to delay in loading tweets and eventually bad user experience.

Column-Family is a better choice

- Let's make the tweet a sharding key. Now, there can be column families such as Tweets, Popular Tweets, etc.
- When the posts related to a tweet are required, you only need to query the first X entries of the tweets column family.
- Similarly, if more tweets are required, you can provide an offset, and fetch records from a particular point.

Live scores of Sports/Matches

Situation:

- Given a recent event or match, you have to show only the ongoing score information.

Key-Value DB is the best choice

- In this situation, Key-Value DB is the best as we simply have to access and update the value corresponding to a particular match/key.
- It is very light weight as well.

Current Location of Cab in Uber-like Application

Situation:

- Uber needs to show the live location of cabs. How to store the live location of cabs?

If location history is needed: Column-Family DB is the best choice

- We can keep the cab as a sharding key and a column family: Location.
- Now, we have to simply fetch the first few records of the Location column family corresponding to a particular cab.
- Also, new records need to be inserted into the Location column family.

If location history is not needed: Key-Value DB is the best choice:

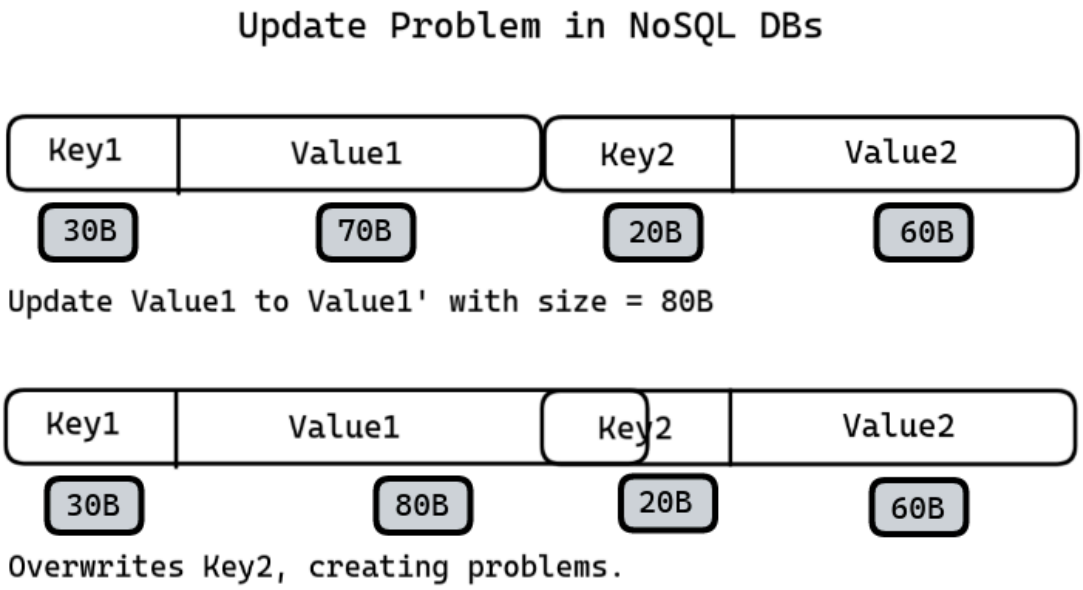
- If only the current location is needed, Key-Value makes a lot more sense.
- Simply fetch and update the value corresponding to the cab (key).

Questions for next class

Problem Statement 1

- Storing data in SQL DBs is easy as we know the maximum size of a record.
- Problem with NoSQL DBs is that the size of value can become exceedingly large. There is no upper limit practically. It can grow as big as you want.
    - Value in Key-Value DB can grow as big as you want.
    - Attributes in Document DB can be as many in number as you want.
    - In Column Family, any single entry can be as large as you want.
- This poses a problem in how to store such data structure in the memory like in HDD, etc.

Update Problem

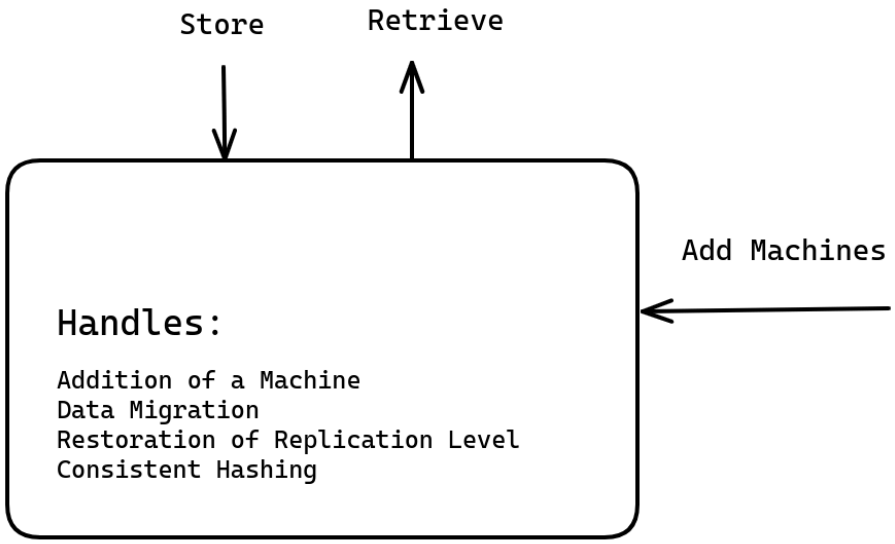


Problem Statement 2

Design a Manual Sharding system which supports:

- Adding new shard + data migration
- When a machine dies inside a shard, necessary actions are performed to restore the replication level.
- The system will keep scaling as new machines will be added.

Design the black box given below.



Manual Sharding
(Black Box)

Your task is to determine how to store data in the memory so that you are able to:

- find entries quickly &
- support updates

Points during questionnaire

- Facebook manually manages sharding in its UserDB which is a MySQL database instance.
- Manual Sharding involves adding a machine, data migration when a new machine is added, consistent hashing while adding a new shard, and if a machine goes down, it has to be handled manually.
- JavaScript developers generally deal with Document DBs pretty easily. JS developers work with JSON objects quite often. Since records in Document DBs are pretty similar to JSON objects, JS Developers find it easy to start with.
- SQL databases are enough to handle small to medium sized companies. If you start a company, design the database using SQL databases. Scaler still runs on (free) MySQL DB and it works perfectly fine. NoSQL is only needed beyond a certain scale.
- ACID is not built-in in NoSQL DBs. It is possible to build ACID on top of NoSQL, however consistency should not be confused with ACID.
- ACID features can be implemented on top of NoSQL by maintaining a central shard machine and channeling all write operations through it. Write operations can be performed only after acquiring the write lock from the central shard machine. Hence, atomicity and isolation can be ensured through this system.
- **Atomicity and Isolation** are different from **Consistency**.

How a transaction is carried out between two bank accounts and how actually it is rolled back in case of failure?
Ans: Such a transaction between two bank accounts has states. For example, let A transfers 1000 INR to B. When money has been deducted from A's account, the transaction goes to **send_initiated** state (just a term). In case of successful transfer, the state of the A's transaction is changed to **send_completed**.
However, let's say due to some internal problem, money did not get deposited into B's account. In such a case, the transaction on A's side is rolled back and 1000 INR is again added to A's bank balance. This is how it is rolled back. You may have seen that money comes back after 1-2 days. This is because the bank re-attempts the money transfer. However, if there is a permanent failure, the transaction on A's side is rolled back.

Resources

- https://try.redis.io/
- https://redis.io/docs/
- https://university.redis.com/
- https://mongoplayground.net/
- https://www.mongodb.com/docs/manual/tutorial/getting-started/
- https://www.mongodb.com/docs/manual/
  https://cassandra.apache.org/_/cassandra-basics.html
- https://cassandra.apache.org/_/case-studies.html
- https://cassandra.apache.org/doc/stable/cassandra/architecture/overview.html
- https://cassandra.apache.org/doc/stable/cassandra/architecture/guarantees.html
- https://cassandra.apache.org/doc/stable/cassandra/data_modeling/intro.html
- https://jbcodeforce.github.io/db-play/