Back of the Envelope(BOE) Estimation

8. Back-Of-The-Envelope Estimation for System Design Interview | Capacity Plan…

Problem Statement:

You are tasked with estimating the bandwidth requirements for a new video streaming platform that will deliver high-definition (HD) content to users. The platform aims to serve 10,000 concurrent users during peak hours.
Below are the steps for Back of the Envelope Estimation

Step 1: Define Parameters

- Number of concurrent users: 10,000
- Quality of video: High-definition (HD)
- Average bitrate of HD video: 5 Mbps (megabits per second) for this example.

Step 2: Calculate Total Bandwidth

- To estimate the total bandwidth required, multiply the number of concurrent users by the average bitrate of the video stream.
- Total Bandwidth = Number of Users × Bitrate per User
- Total Bandwidth = 10,000 users × 5 Mbps
- Total Bandwidth ≈ 50,000 Mbps

Step 3: Convert Bandwidth

- Convert the total bandwidth from Mbps to Gbps for easier understanding.
- Total Bandwidth ≈ 50 Gbps (1 Gbps = 1000 Mbps)

Step 4: Consider Additional Factors

Consider additional factors such as overhead for protocol headers, network congestion, and peak traffic spikes. For simplicity, let's add a 20% overhead factor.

- Total Bandwidth with Overhead = Total Bandwidth × (1 + Overhead Factor)
- Total Bandwidth with Overhead ≈ 50 Gbps × (1 + 0.20)
- Total Bandwidth with Overhead ≈ 60 Gbps

Proxy server

Proxy In 5 Minutes | What Is A Proxy? | What Is A Proxy Server? | Proxy Explaine…
Proxy vs Reverse Proxy vs Load Balancer | Simply Explained

A proxy server is a system or router that provides a gateway between users and the internet. Therefore, it helps prevent cyber attackers from entering a private network. It is a server, referred to as an “intermediary” because it goes between end-users and the web pages they visit online.
When a computer connects to the internet, it uses an IP address. This is similar to your home’s street address, telling incoming data where to go and marking outgoing data with a return address for other devices to authenticate. A proxy server is essentially a computer on the internet that has an IP address of its own.

How to use a proxy? Some people use proxies for personal purposes, such as hiding their location while watching movies online, for example. For a company, however, they can be used to accomplish several key tasks such as:

1. Improve security
2. Secure employees’ internet activity from people trying to snoop on them
3. Balance internet traffic to prevent crashes
4. Control the websites employees and staff access in the office
5. Save bandwidth by caching files or compressing incoming traffic

Purpose of Proxy Servers

https://www.tutorialspoint.com/internet_technologies/proxy_servers.htm

Types

Proxy server types and uses for HTTP Server

Transparent proxy

A transparent proxy can give users an experience identical to what they would have if they were using their home computer. In that way, it is “transparent.” They can also be “forced” on users, meaning they are connected without knowing it.

Transparent proxies are well-suited for companies that want to make use of a proxy without making employees aware they are using one. It carries the advantage of providing a seamless user experience. On the other hand, transparent proxies are more susceptible to certain security threats, such as SYN-flood denial-of-service attacks.

Anonymous proxy

An anonymous proxy focuses on making internet activity untraceable. It works by accessing the internet on behalf of the user while hiding their identity and computer information.

An anonymous proxy is best suited for users who want to have full anonymity while accessing the internet. While anonymous proxies provide some of the best identity protection possible, they are not without drawbacks. Many view the use of anonymous proxies as underhanded, and users sometimes face pushback or discrimination as a result.

High anonymity proxy

A high anonymity proxy is an anonymous proxy that takes anonymity one step further. It works by erasing your information before the proxy attempts to connect to the target site.

The server is best suited for users for whom anonymity is an absolute necessity, such as employees who do not want their activity traced back to the organization. On the downside, some of them, particularly the free ones, are decoys set up to trap users in order to access their personal information or data.

Database sharding

If we want to scale up a system's performance, the last resort should be **Database Sharding**. Before reaching that point, we can try several optimizations:

- If read requests take too long, we can **improve queries** or **remove unnecessary joins**.
- **Database De-normalization** to reduce complex joins and improve read performance.
- **Duplicate Databases** to handle load balancing and distribute traffic.

If all these optimizations fail to improve performance, then **Database Sharding** should be considered as the final approach.

What is database sharding?

Database sharding is the process of storing a large database across multiple machines. A single machine, or database server, can store and process only a limited amount of data. Database sharding overcomes this limitation by splitting data into smaller chunks, called shards, and storing them across several database servers. All database servers usually have the same underlying technologies, and they work together to store and process large volumes of data.

Sharding involves breaking up one’s data into two or more smaller chunks, called **logical shards**. The logical shards are then distributed across separate database nodes, referred to as **physical shards**, which can hold multiple logical shards. Despite this, the data held within all the shards collectively represent an entire logical dataset.

Database shards exemplify a **shared-nothing architecture**. This means that the shards are autonomous; they don’t share any of the same data or computing resources. In some cases, though, it may make sense to replicate certain tables into each shard to serve as reference tables. For example, let’s say there’s a database for an application that depends on fixed conversion rates for weight measurements. By replicating a table containing the necessary conversion rate data into each shard, it would help to ensure that all of the data required for queries is held in every shard.

> Database sharding operates on a shared-nothing architecture. Each physical shard operates independently and is unaware of other shards. Only the physical shards that contain the data that you request will process the data in parallel for you.

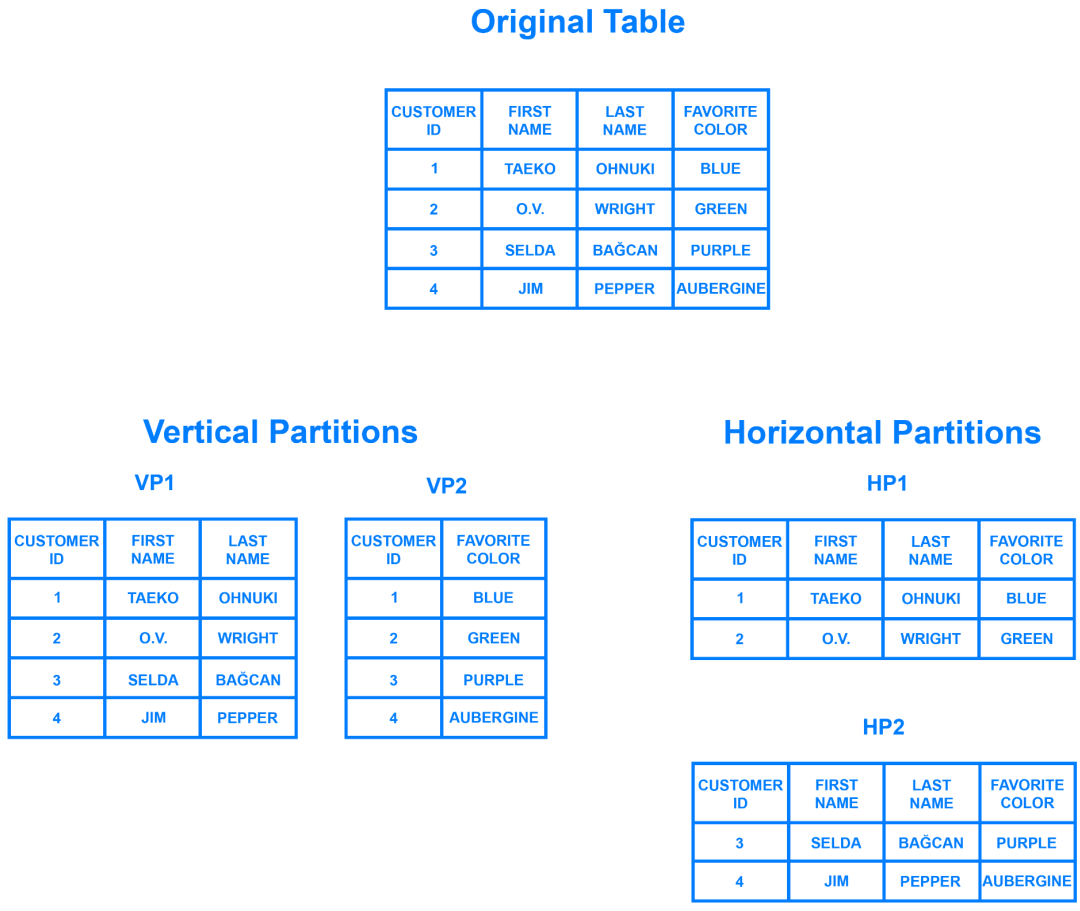
Horizontal Partition

Sharding is a database architecture pattern related to horizontal partitioning — the practice of separating one table's rows into multiple different tables, known as partitions. Each partition has the same schema and columns, but also entirely different rows. Likewise, the data held in each is unique and independent of the data held in other partitions.

> Sharding and partitioning are both techniques to divide a database for better performance and scalability, but they have different approaches. **Sharding** splits data across multiple databases or servers, while **partitioning** divides data into smaller, manageable pieces within a single database.

Vertical Partition

Entire columns are separated out and put into new, distinct tables. The data held within one vertical partition is independent from the data in all the others, and each holds both distinct rows and columns.



Algorithmic Sharding (Static Sharding)

In algorithmic sharding, data is distributed across shards based on a fixed algorithm, typically using a hash function or range-based rules. The shard where a piece of data is stored is determined by computing a key.

Example:
Imagine you're managing a user database. You decide to shard based on the user ID:

$$\text{Shard} = \text{User ID} \% 3$$

- **User ID 101 → Shard 2**
- **User ID 202 → Shard 1**

This way, each user is consistently assigned to a shard.

✅ **Pros:**

- Fast and predictable lookups (you always know where the data is).
- Simpler to implement and manage.

❌ **Cons:**

- Hard to scale: adding shards may require reshuffling all data.
- Uneven load distribution if the hash function isn't well balanced.

Dynamic Sharding (Shard Mapping)

Dynamic sharding assigns data to shards based on a mapping that can change over time. It often uses a lookup table or metadata service to track which shard holds which data.

Example: An e-commerce platform maps each product category to a shard:

- **Shard 1 → Electronics**
- **Shard 2 → Clothing**
- **Shard 3 → Books**

If the "Clothing" category grows too large, the mapping can be updated without rehashing all products.

✅ **Pros:**

- Easier to scale: new shards can be added without major disruptions.
- More flexible: can handle uneven data distribution.

❌ **Cons:**

- Extra lookup step: finding the shard requires querying the mapping.
- More complex to implement and maintain.

When to Use Which?

- **Algorithmic Sharding:** When you have stable, evenly distributed data and don't expect frequent scaling.
- **Dynamic Sharding:** When data grows unpredictably, and flexibility is more important than simplicity.

Sharding Criteria

[Key Based Sharding]

[Range Based Sharding]

[Directory Based Sharding]

List-Based Sharding

List-Based Sharding involves distributing data based on a predefined list of values that are mapped to specific shards. This is similar to range-based sharding, but instead of a continuous range, data is divided based on discrete values in a list. For example, customers could be assigned to shards based on predefined lists of values like geographic regions (e.g., North America, Europe, Asia) or product categories (e.g., Electronics, Clothing, Furniture).

Pros:

- **Simple to Implement:** List-based sharding is relatively easy to understand and implement, as it relies on discrete categories for partitioning.
- **Efficient for Certain Queries:** If most queries are related to specific categories (e.g., all users from a particular region), list-based sharding provides a direct mapping to the relevant shard, making these types of queries efficient.
- **Good for Segmented Workloads:** It works well when workloads can be naturally divided into non-overlapping categories, such as geographical regions or distinct product lines.

Cons:

- **Uneven Distribution:** If the list is not designed carefully, it could lead to uneven data distribution. Some categories may grow much larger than others, leading to hotspots in certain shards.
- **Difficult to Scale Dynamically:** Unlike key-based sharding, adding new categories to the list (or changing existing ones) can be complex and require a redistribution of data across the shards.
- **Limited Flexibility:** List-based sharding might not be suitable for data that doesn't fit into predefined categories, and it may require frequent updates to the list as new categories emerge.

When to Use:

- Best suited for systems where data can be logically divided into well-defined, discrete groups or categories that rarely change.
- Ideal for applications where specific queries consistently target distinct categories, such as retrieving all data from a particular region or product line.
- Works well for systems where the number of categories is relatively fixed and does not change frequently.

Round-Robin Based Sharding

Round-Robin Sharding involves distributing data across multiple shards in a circular manner. In this method, each incoming data item is assigned to the next shard in the sequence, ensuring that data is evenly distributed across all available shards. For example, if there are 3 shards, the first piece of data goes to shard 1, the second to shard 2, the third to shard 3, and the fourth goes back to shard 1, and so on.

Pros:

- **Simple to Implement:** Round-robin sharding is easy to implement because it does not require complex logic for assigning data to shards. The algorithm simply rotates through the shards in order.
- **Even Data Distribution:** Since data is distributed in a balanced manner, this method ensures that each shard holds roughly the same amount of data, which helps prevent hotspots and load imbalances.
- **Scalability:** Adding new shards is straightforward — new shards are simply added to the rotation, and data continues to be assigned evenly across the available shards.

Cons:

- **No Control Over Access Patterns:** Since data is distributed randomly, round-robin sharding does not take into account the frequency or importance of specific data, which may lead to inefficient queries. For example, if a query often

needs to access data from a particular shard, it may require accessing many shards unnecessarily.
- **Inefficient for Certain Queries:** Some queries that need to access a specific subset of data across multiple shards may require more work to locate the relevant data, as the data is not grouped by any specific characteristic (e.g., time, region, or user).
- **Rebalancing Complexity:** When adding or removing shards, the data must be redistributed across the new set of shards. This can lead to data migration, which may be resource-intensive.

When to Use:

- Best for situations where the workload is evenly distributed and queries do not frequently target specific subsets of data.
- Suitable for use cases where the primary goal is to achieve a balanced distribution of data across multiple shards with minimal complexity.
- Ideal for applications with uniform access patterns, where each shard is expected to handle similar loads and there is no specific need for categorizing the data.

Example:

User Session Data: In an online platform that handles user sessions, round-robin sharding can be used to distribute session data across multiple servers. Each new session is assigned to the next available shard in the sequence, ensuring a balanced load across all shards. Since session data is typically short-lived and doesn’t need to be queried in complex ways, this sharding method is efficient.

Log Data: For applications that generate log files, round-robin sharding can be used to distribute logs across different shards. The logs are often large and continuous, making an even distribution across multiple shards advantageous to prevent any one shard from becoming overloaded.

Geo Sharding

Geo Sharding splits and stores database information based on geographical locations. For instance, a dating service website may store customer information by cities. The software developers use cities as shard keys, so each customer's data is stored in physical shards that are located in the same geographical area as the customer. If the customer is from New York, their data would be stored in a shard located near New York.

Pros:

- **Faster Data Access:** Geo sharding can lead to faster data retrieval, as the requestor is accessing data stored in a shard physically closer to them. This reduces latency and improves the performance of geographically localized queries.
- **Reduced Latency:** By serving data from a shard that is closer to the geographical region of the user, the overall latency of data retrieval is reduced, which improves response times for users.
- **Efficient for Geo-Localized Workloads:** If the data access patterns are predominantly geographic (e.g., accessing local customer information), geo sharding can be highly efficient in addressing the use case.
- **Improved User Experience:** By reducing the time it takes to access data, geo sharding can provide a better and more responsive user experience, especially in applications with a global user base.

Cons:

- **Uneven Data Distribution:** Geo sharding may lead to uneven distribution of data across shards, especially if certain regions (cities, countries, etc.) have more users or more activity than others. This could result in hotspots where some shards are overburdened, while others remain underutilized.
- **Difficulty in Scaling:** As the data grows or the user base expands into new geographical regions, it can become challenging to scale the system. Redistributing data across new shards may be required, which can be complex and time-consuming.
- **Regional Imbalances:** In regions with higher user densities, the shards may become disproportionately large, leading to resource strain, while less populated areas may have much smaller shards.
- **Data Coherence and Migration Issues:** When users from one region travel or move, their data may need to be migrated to a different shard to reflect their new location. This can lead to additional complexity in data consistency and migration strategies.

When to Use:

- Best suited for applications where data access is region-specific, such as global applications where most of the traffic or queries come from specific geographic locations.
- Ideal for services that involve location-based data, like dating apps, e-commerce platforms targeting regional markets, or services with geographical content like news, weather, or mapping applications.
- Works well for applications with a high volume of localized traffic that needs fast, low-latency access to region-specific data.

Example:

Dating Service Website: A dating app may use geo sharding to store customer profiles in physical servers located in different cities or countries. When a user in New York queries the database for potential matches, the query is routed to the shard located in or near New York, ensuring faster response times and improved user experience.

E-commerce Site: An e-commerce website with a global customer base could use geo sharding to store product and customer data in shards based on user location. This would reduce the time it takes to retrieve information about shipping, pricing, and product availability for users in different regions.

Composite Sharding

Composite Sharding is a method that combines multiple shard keys to determine how data is distributed across shards. This method uses more than one attribute of the data to partition it, typically combining two or more shard keys to achieve a more precise and efficient distribution. For example, in an e-commerce system, the data could be distributed across shards using both **region** and **product category** as shard keys. This would mean the data is partitioned first by region (e.g., North America, Europe) and then by product category (e.g., Electronics, Clothing).

Pros:

- **Fine-Grained Data Distribution:** By combining multiple shard keys, composite sharding allows for a more fine-grained and tailored distribution of data. This can be beneficial in applications where data has multiple access patterns, such as querying both by geographic region and by product type.
- **Improved Query Efficiency:** Composite sharding can make certain queries more efficient, especially when the queries frequently involve filtering by multiple attributes (e.g., region and product category). With the right composite shard key, these queries can be routed to the exact shard that contains the relevant data, avoiding the need to scan multiple shards.
- **Reduces Hotspotting:** By using multiple keys, composite sharding can help prevent hotspots that may occur if a single key is used for partitioning, leading to better load balancing. If some regions or product categories grow faster than others, the data is spread across different shards in a more balanced way.

Cons:

- **Complexity in Shard Key Selection:** Choosing the right combination of shard keys can be challenging. The chosen keys must align well with the most common access patterns to achieve the desired benefits. Poor shard key choices can lead

to imbalances in the data distribution, as well as inefficiencies in query performance.

- **Increased Complexity in Data Management:** Managing composite sharding can be more complex than single-key sharding, especially when it comes to handling rebalancing, scaling, and data migrations. When new shards are added, the system may need to reassess the distribution of data to maintain the desired balance.
- **Limited Flexibility:** The composite shard key limits flexibility. If a user wants to query data based on only one of the shard key attributes (e.g., by region alone), it may be less efficient because the query might need to scan all shards, even if the data could be distributed differently.

When to Use:

- Best suited for applications with complex query patterns that involve multiple attributes or dimensions. For example, in a multi-tenant SaaS application where data queries often involve combinations of tenant IDs, regions, and product categories.
- Ideal for situations where data access can benefit from partitioning by two or more attributes, ensuring that common query patterns are optimized.
- Useful for systems where a single shard key would result in uneven distribution or where load balancing based on a single attribute is not sufficient.

Extra Ref:
https://www.geeksforgeeks.org/database-sharding-a-system-design-concept/#2-sharding-architectures

Ways optimize database sharding for even data distribution

https://www.geeksforgeeks.org/database-sharding-a-system-design-concept/#ways-optimize-database-sharding-for-even-data-distribution

Advantages of Sharding in System Design

1. Enhances Performance: By distributing the load among several servers, each server can handle less work, which leads to quicker response times and better performance all around.
2. Scalability: Sharding makes it easier to scale as your data grows. You can add more servers to manage the increased data load without affecting the system's performance.

3. Improved Resource Utilization: When data is dispersed, fewer servers are used, reducing the possibility of overloading one server.
4. Fault Isolation: If one shard (or server) fails, it doesn't take down the entire system, which helps in better fault isolation.
5. Cost Efficiency: You can use smaller, cheaper servers instead of investing in a large, expensive one. As the system grows, sharding helps keep costs in control.

Disadvantages of Sharding in System Design

1. Increased Complexity: Managing and maintaining multiple shards is more complex than working with a single database. It requires careful planning and management.
2. Rebalancing Challenges: If data distribution becomes uneven, rebalancing shards (moving data between servers) can be difficult and time-consuming.
3. Cross-Shard Queries: Queries that need data from multiple shards can be slower and more complicated to handle, affecting performance.
4. Operational Overhead: With sharding, you'll need more monitoring, backups, and maintenance, which increases operational overhead.
5. Potential Data Loss: If a shard fails and isn't properly backed up, there's a higher risk of losing the data stored on that shard.