From the vault of my engineering newsletter
"Systems That Scale"

Saurav Prateek's

Foundational concepts in System Design Part 1

Explaining the foundational concepts involved in **System Design**

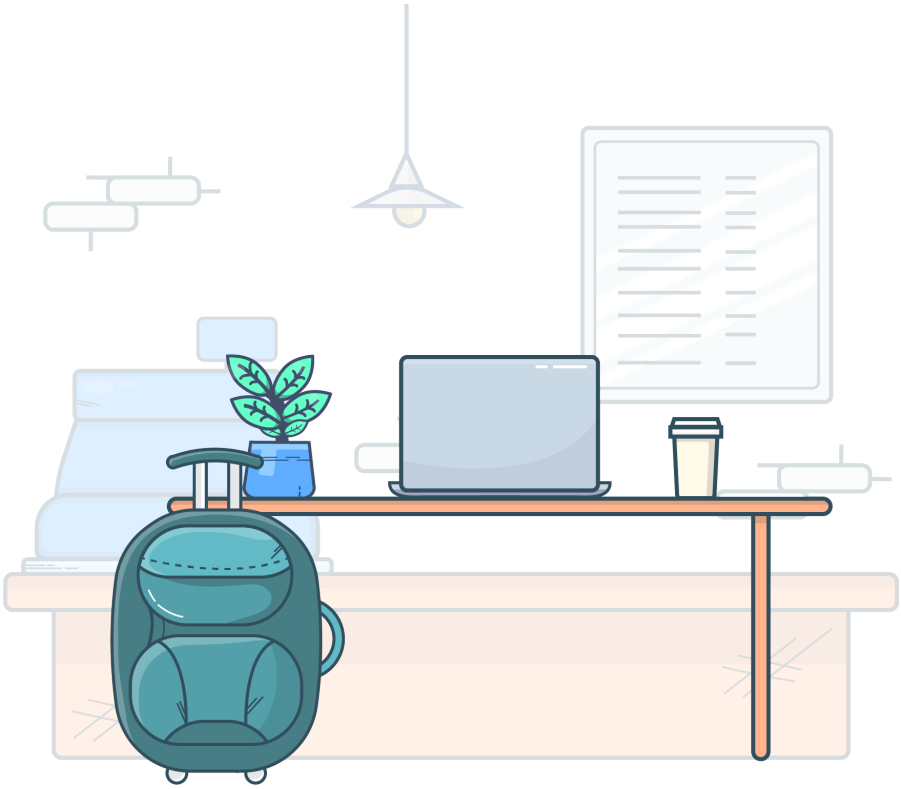


Table of Contents

Chapter 1

Getting Started with System Design

A basic introduction on how to approach the Design problems and understanding the concepts of **Reliability**, **Availability** and **Scalability** in System Design.

Introduction

System Design is the process of designing the architecture, components and interfaces for a system so that it meets the end user requirements.

It's a wide field of study in Engineering, and includes various concepts and principles that will help you in designing scalable systems. These concepts are extensively asked in the Interview Rounds for **SDE 2** and **SDE 3** Positions at various tech companies. These senior roles demand a better understanding of how you solve a particular design problem, how you respond when there is more than expected traffic on your system, how you design the database of your system and many more. All these decisions are required to be taken carefully keeping in mind about **Scalability**, **Reliability** and **Availability**. We will be covering all of these terminologies in this article.

Before we start discussing the terminologies, there are few things we should need to clarify. When you are given a System Design Problem, you should approach it in a planned manner. Initially the Problem may look huge, and one can easily get confused on how to start solving it. And moreover there is no fixed solution while you are designing a system. There is more than one way to reach the Solution.
So let's discuss how one should start with solving a Design Problem given in an Interview.

Approaching a System Design problem

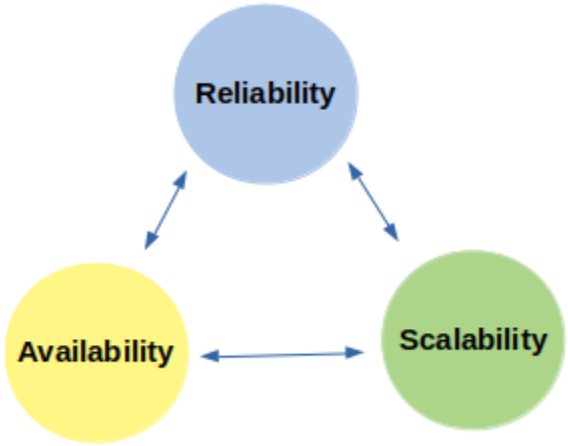
Few pointers to take care of when you approach a System Design problem:

- **Breaking Down the Problem**: When you are given a Design Problem start breaking it down to small components. These components can be Services or Features which you need to implement in the System. Initially a System given to be designed can have a large number of features and you are not expected to design everything if it's an Interview. Ask your interviewer about the Features you are planning to put in your system. Is there anything else you should be putting there ? Any Feature ? Any Service ? ... Ask !

- **Communicating your Ideas**: Communicate well with the Interviewer. While designing the system keep him in the loop. Discuss your process out loud. Try to demonstrate your design clearly on the whiteboard with flowcharts and diagrams. Describe your ideas to your interviewer, how you have planned to tackle the problem of scalability, how you will be designing your database and many others.

- **Assumptions that make sense**: Make some reasonable assumptions while you are designing the System. Suppose you have to assume the number of requests the system will be processing per day, the number of database calls made in a month or the efficiency rate of your Caching System. These are some of the numbers you need to assume while designing. Try to keep these numbers as reasonable as possible. Back up your assumption with some solid facts and figures.

Now we know how to approach a design problem. But in order to succeed in the Interview or to successfully build a scalable system, we need to ensure that our system is reliable, available, scalable and maintainable. So we should have knowledge of what these terms are and how they impact our system in the long run.



Reliability

A system is Reliable when it can meet the end user requirement. When you are designing a system you should have planned to implement a set of features and services in your system. If your system can serve all those features without wearing out then your System can be considered to be **Reliable**.

A **Fault Tolerant** system can be one which can continue to be functioning reliably even in the presence of faults. **Faults** are the errors which arise in a particular component of the system. An occurrence of fault doesn't guarantee Failure of the System.

Failure is the state when the system is not able to perform as expected. It is no longer able to provide certain services to the end users.

Availability

Availability is a characteristic of a System which aims to ensure an agreed level of Operational Performance, also known as **uptime**. It is essential for a system to ensure high availability in order to serve the user's requests.

The extent of Availability varies from system to system. Suppose you are designing a Social Media Application then high availability is not much of a need. A delay of a few seconds can be tolerated. Getting to view the post of your favourite celebrity on Instagram with a delay of 5 to 10 seconds will not be much of an issue. But if you are designing a system for Hospital, Data Centers or Banking, then you should ensure that your system is highly available. Because a delay in the service can lead to a huge loss.

There are various principles you should follow in order to ensure the Availability of your system :

- Your System should not have a Single Point of Failure. Basically your system should not be dependent on a single service in order to process all of its requests. Because when that service fails then your entire system can be jeopardised and end up becoming unavailable.

- Detecting the Failure and resolving it at that point.

Scalability

Scalability refers to the ability of the System to cope up with the increasing load. While designing the system you should keep in mind the load experienced by it. It's said that if you have to design a system for load **X** then you should plan to design it for **10X** and Test it for **100X**. There can be a situation where your system can experience an increasing load. Suppose you are designing an E-commerce application then you can expect a spike in the load during a Flash Sale or when a new Product is Launched for sale. In that case your system should be smart enough to handle the increasing Load efficiently and that makes it **Scalable**.

In order to ensure scalability you should be able to compute the load that your system will experience. There are various factors that describe the Load on the System:

- Number of requests coming to your system for getting processed per day.
- Number of Database calls made from your system.
- Amount of Cache Hit or Miss requests to your system.
- Users currently active on your system.

Chapter 2

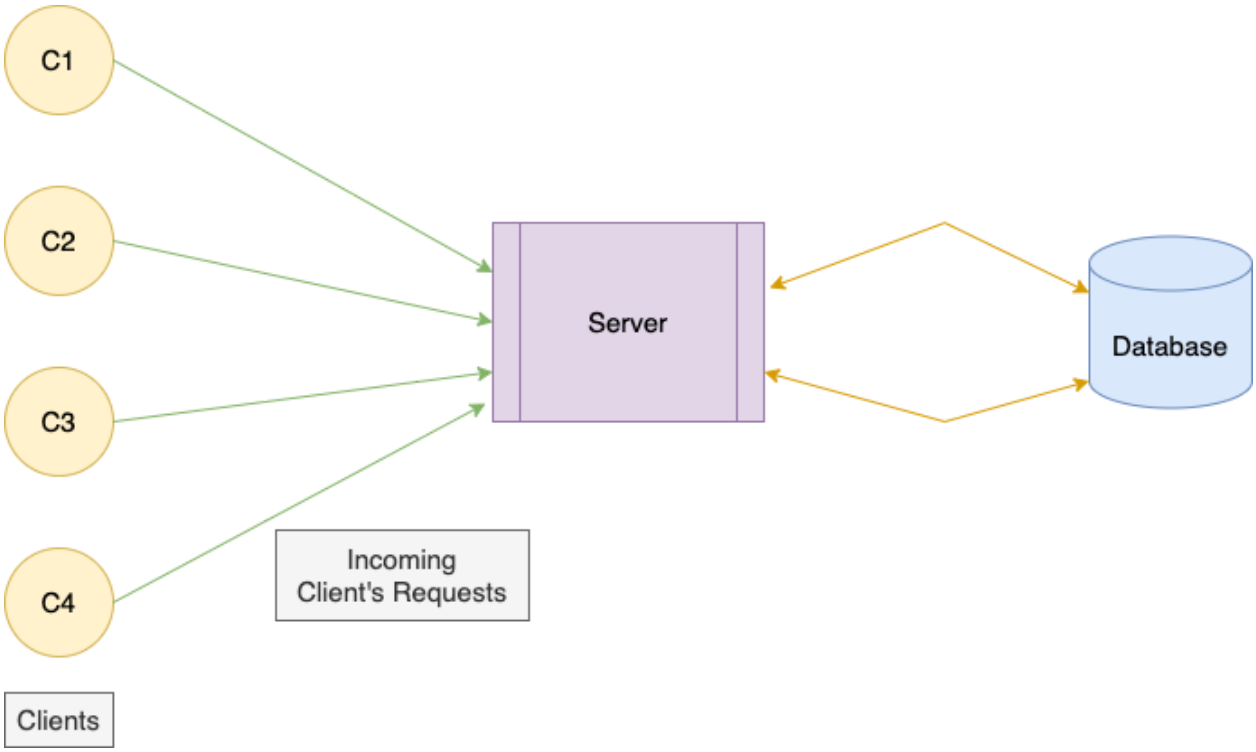
Load Balancing

Let's talk about Load Balancing, Algorithms used in Load Balancing, Statefulness and the emerging need for Consistent Hashing.

Introduction

Load Balancing is one of the most widely used topics when we design systems. Almost every system which deals with a considerable number of clients/requests will need Load balancers at some point of time.

Suppose you built a service which you want others to use. You have also set up a payment model which will allow your clients to pay you as they use your service. Since you have just launched your service and there are not many people aware about it, you might be having a handful of clients willing to use your service. So, initially with a small number of clients you can directly set up your service with a single server without thinking about Load Balancing. The initial architecture may look like this.

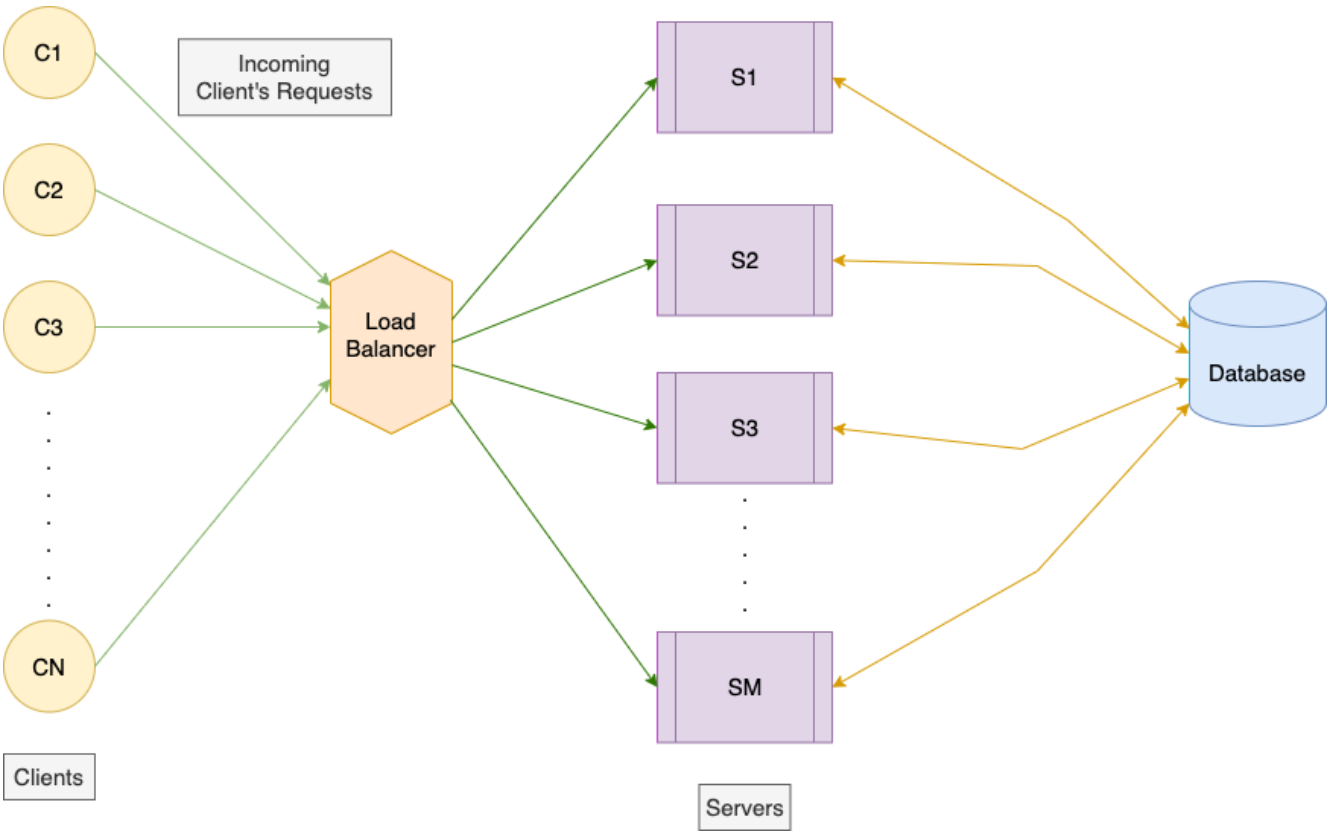


But, with time your service will start getting popular and people will come to know about it. You have made a flawless service and your clients liked it so much that

they are going to tell others as well. Soon you will be having a lot of Clients willing to use your service and pay you for that.

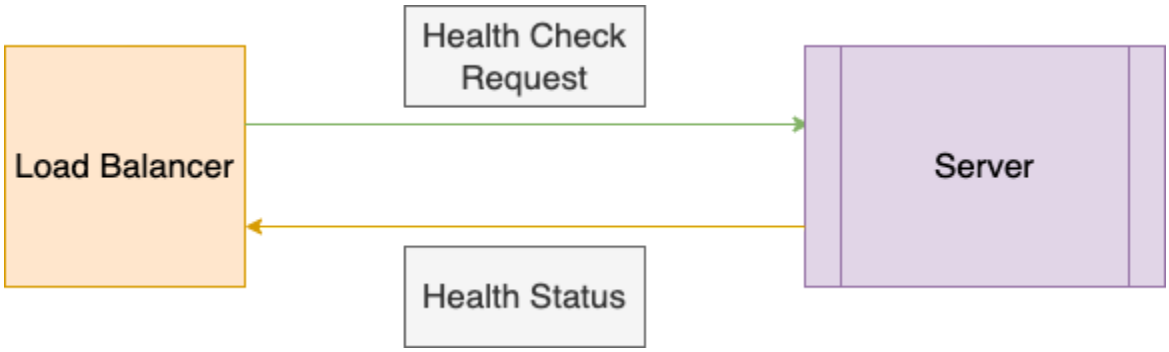
Now you have a lot of requests coming in and you realise that one system won't be able to handle this amount of load. With the money you received from the clients , you decide to buy multiple servers in order to handle the increasing load. With multiple servers with yourself now you need something to distribute the incoming client's requests to these servers evenly. This process of distributing the incoming requests evenly to the multiple machines is known as **Load Balancing**. The component which performs this is known as **Load Balancer**.

With Load Balancers in the picture, now your architecture will look like this.



Health Checks by Load Balancers

Load Balancers are also widely used to perform **Health Checks** of the backend servers in order to know their availability. In this scenario the load balancers ping the backend servers and the servers reply to them with their States, whether they are available to take requests or not.



Types of Load Balancers

We will discuss two types of **Load Balancers** which have been around for quite some time and are popular as well.

1. **L4 Load Balancers** or **Network Load Balancers**: L4 Load Balancer routes the request on the basis of the address information of the incoming requests. It does not inspect the content of the request. The Layer 4 (L4) Load Balancer makes the routing decisions based on address information extracted from the first few packets in the TCP stream.

2. **L7 Load Balancers** or **Application Load Balancers**: L7 Load Balancer routes the request on the basis of the packet content. There can be dedicated servers which could serve the requests based on their content, like URLs, Images, Graphics and Video contents. You can set up the system in such a way that the static contents are served by one dedicated server and requests

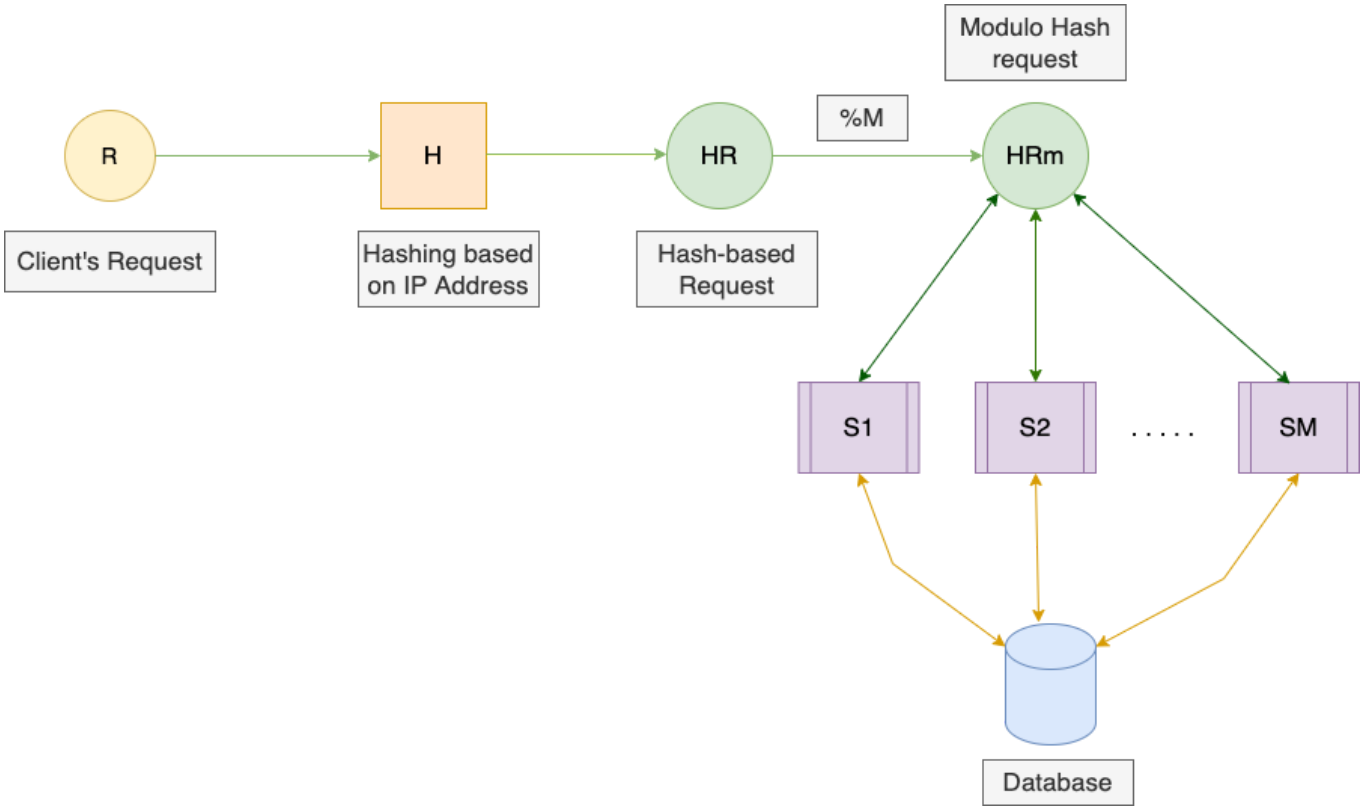
demanding certain information which needs a db call can be served by another dedicated server.

Maintaining States in Load Balancing

Suppose your system wants to maintain states between multiple requests. Load Balancers can ensure this Statefulness. They can make sure that a particular incoming request is always routed to the same server. And the backend server can then use a **Local Cache** to store the metadata regarding the requests to be used at a later point of time.

One possible way a Load Balancer can achieve this is by memorising the IP address of the incoming Client's request. The Load balancer can use a constant **Hash Function** that can build a **Hash** out of the incoming **Client's IP** address and the **total number of available backend servers** at that point of time. We can assume that the chosen Hash Function will evenly distribute the incoming requests to the backend servers.

Suppose we have **M** backend servers currently available and a hash function **H** that hashes the incoming requests on the basis of IP Address and number of backend servers (**M**). The hash function hashes the incoming client's request on the basis of the IP address and later modulo the generated hash by the total number of available backend servers (**M**). It does the modulo in order to direct the incoming hashed request to one of the available backend servers.



Hashing the Client's request on the basis of its IP address allows the request coming from the same IP address to be handled by the same backend server most of the time. This allows the backend server to make use of its Local Cache and reduce the service response time. Hence increasing the system performance.

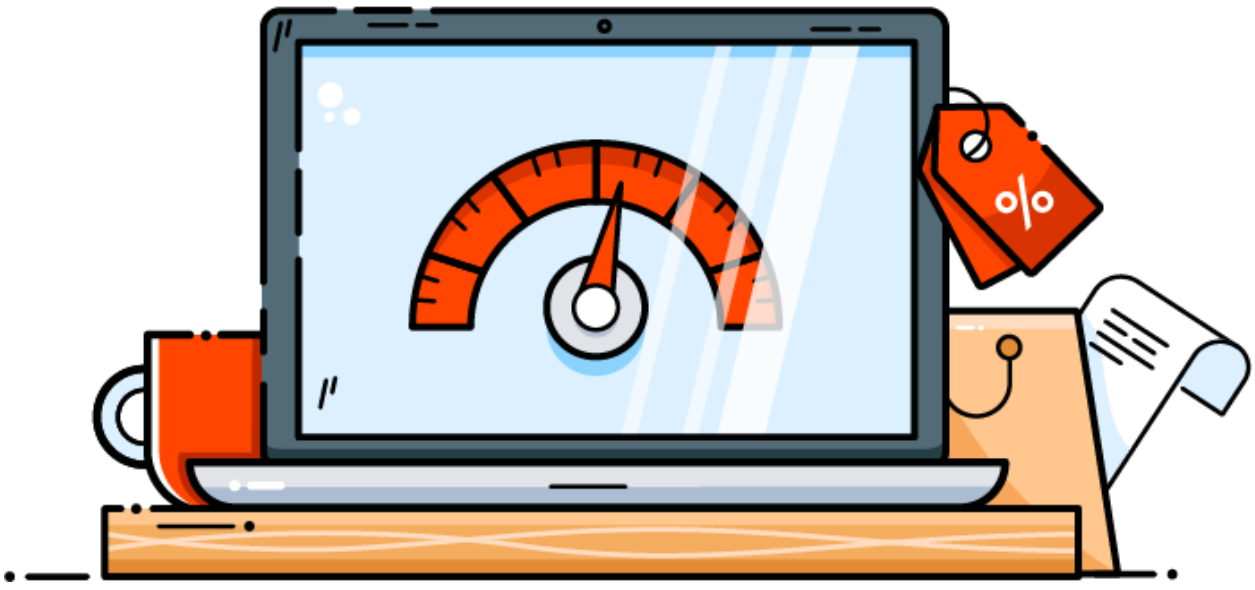
But what if the number of Servers changes frequently? There can be scenarios where some servers may get worn out or multiple new servers are introduced into the system to handle the increasing load. In those situations since the number of servers have changed, our Hash Function will route the same requests to completely different sets of backend servers. This can make the Local Cache of the servers completely useless and can also increase the response time of the service and ultimately degrade its performance.

We solve this problem with the help of **Consistent Hashing**. It allows a minimal change in the hash function of all the earlier Requests whenever any new server is added or removed.

Chapter 3

Caching

In this chapter, we will discuss **Caching** in general, its characteristics and will also improve an existing naive architecture step by step.



Introduction

Caching is one of the most common concepts which has been around us for a long time and we have been using it at multiple places in many different ways. Basically **Caching** is the process of storing copies of data in a temporary storage from where they can be accessed more quickly.

Initially when your system doesn't handle much load then we can come up with an initial architecture where a client sends the request to the backend server and our backend server further hits the database server to fetch the data needed in order to fulfil the client's request. This is a naive solution and can get our job done when our system is handling a tiny amount of load. But what will happen when the load on our system starts increasing? Will following the initial architecture be a good idea?

Remember, hitting a database server to fetch some data is a costly operation. It is both time taking as well as costly. Hence when the load on our system grows, then if we stick to the initial architecture, every request will further hit the database server and the load over the server will increase. This can be costly and will also make your system slow. Means the response time of your system will increase, clients will need to wait for some time to get their requests served.

Here we can involve Caching. The concept suggests to store the information which is frequently demanded in a temporary location so that we won't be needed to hit the database servers again and again in order to fetch the same data. Those information which are in demand and don't change frequently are ideal to cache. Fetching the information from a cache is much faster as compared to fetching it from the database servers. This can be cost effective as well and can safeguard your database servers from wearing out and eventually shutting down. The temporary location where we store the information is known as **Cache** and this process is known as **Caching**.

Since we are discussing Cache, let's get familiar with two important terms which will be used widely under this concept.

- **Cache Hit**: When we look for information in the Cache and it has that information stored then we call it a Cache Hit. In this case the request can be served solely by the Cache.

- **Cache Miss**: When we look for information in the Cache and it's not found in it, we call it a Cache Miss. In this case the system needs to hit the Database servers and fetch the information required. Some systems may store that information back in the Cache while processing the request.

Characteristics

Let's discuss the characteristics of Caching:

1. **Fast** : Caching can reduce the response time of your system to a great extent. You should be wondering why caching is fast. Basically a cache uses an **SSD** to store your response and they can be accessed very quickly by the system as compared to a database call.

2. **Less Database Calls**: Caching when used efficiently could reduce the DB calls to a large extent. Suppose there is a piece of data which is in a really high demand by the end user and approx. **70%** of the response your system receives demands this piece of data only. Then if you cache that data then you could possibly reduce your DB calls by approx. **70%** as most of the requests will be processed through the cache.

3. **Reduces Computation**: Caching can also reduce the computation performed by the system. Suppose you have a request which demands the maximum percentage obtained in a CS coursework, then you will have to make a db call which may go through every percentage and return the maximum through comparison. You could store this maximum percentage value in your cache so that when next time any request demands this data then you could avoid making all the computations and directly return the end result.

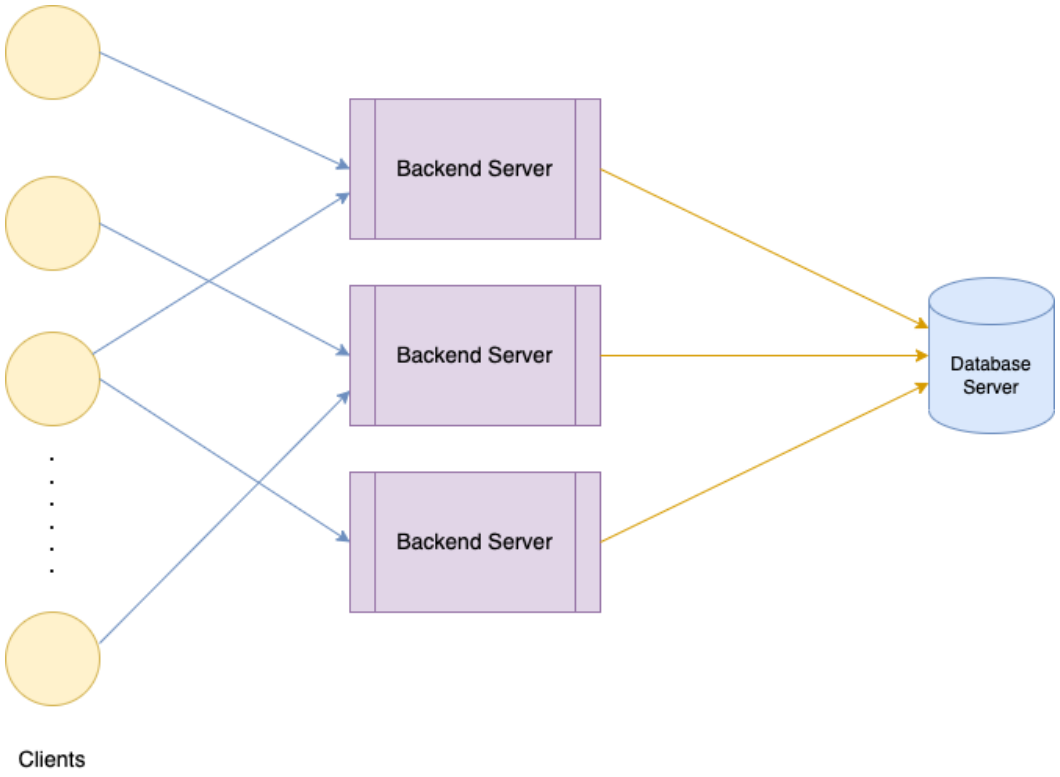
4. **Data Inconsistency**: There is a chance of data inconsistency in case of caching. Because during caching you have the same piece of data located in the cache as well as in your database. So if there are any updates in the database then it may cause the data to become inconsistent. Various cache writing methods like write back or write through are used to ensure the data consistency.

Evolving your Architecture

Since we spent some time discussing Caching, let's look at multiple architectures. We will be starting with a very minimal architecture and evolving it according to our use case and in order to meet the end user requirements. Basically we will be scaling our naive architecture to handle the increasing incoming load.

Architecture 1 : A very naive architecture

Initially when we don't have much load on our system the below architecture can get the job done for sometime.



17 - Saurav's handbook on System Design foundational concepts Part 1

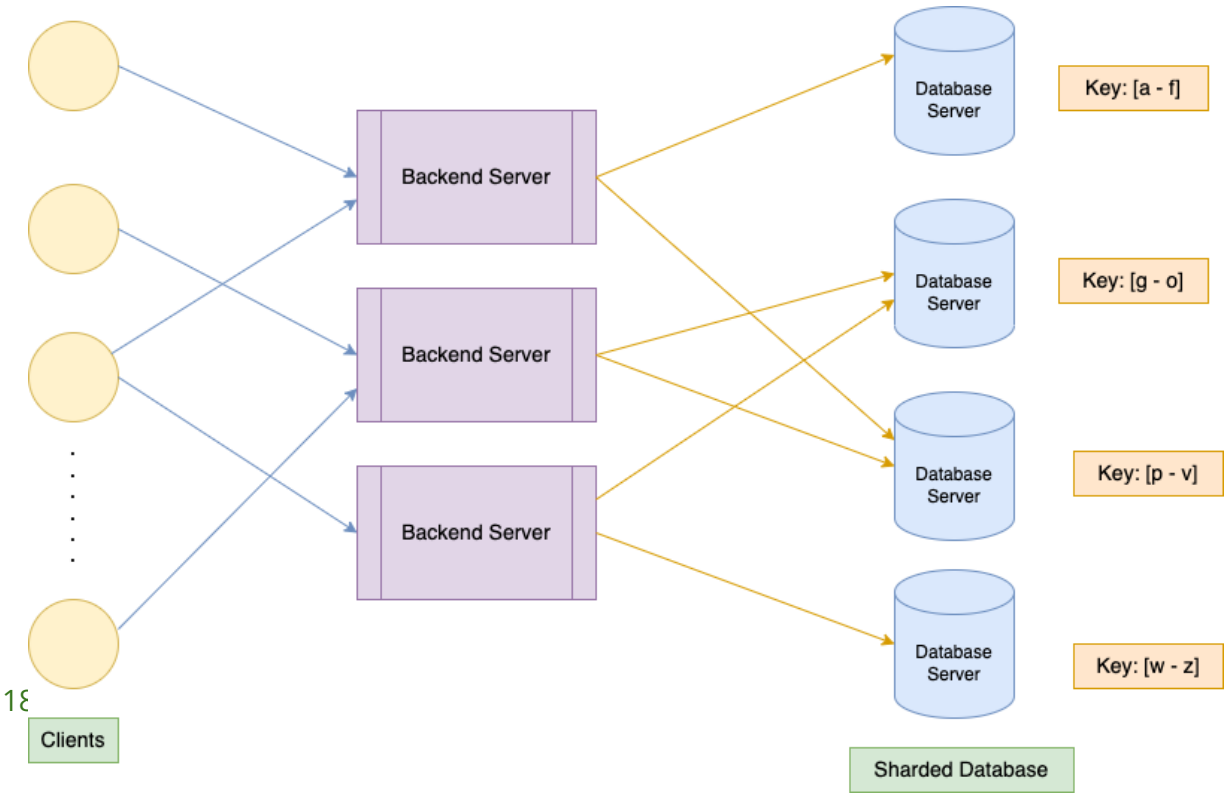
Here, we have multiple backend servers to process the incoming Clients' requests and a single Database server which holds our system's data. Now every incoming client's request will hit one of our backend servers and every backend server will further need to hit the database server for processing the request. The single Database Server will receive each and every **Read** and **Write** requests from the multiple servers.

But as soon as your service starts growing, more and more clients will start using it and eventually there will be a lot of requests coming to your system. Their requests will be distributed to the multiple backend servers but eventually all of them will fall on to a single database server present. This can let the database server wear out and eventually shut it down leaving your entire system incapable of handling any further requests.

Since the load has increased, we need to evolve our architecture as well. Let's move on to the second architecture.

Architecture 2 : With Sharded Database

In this architecture we will shard our Database and move it to multiple database servers to partition the incoming load.



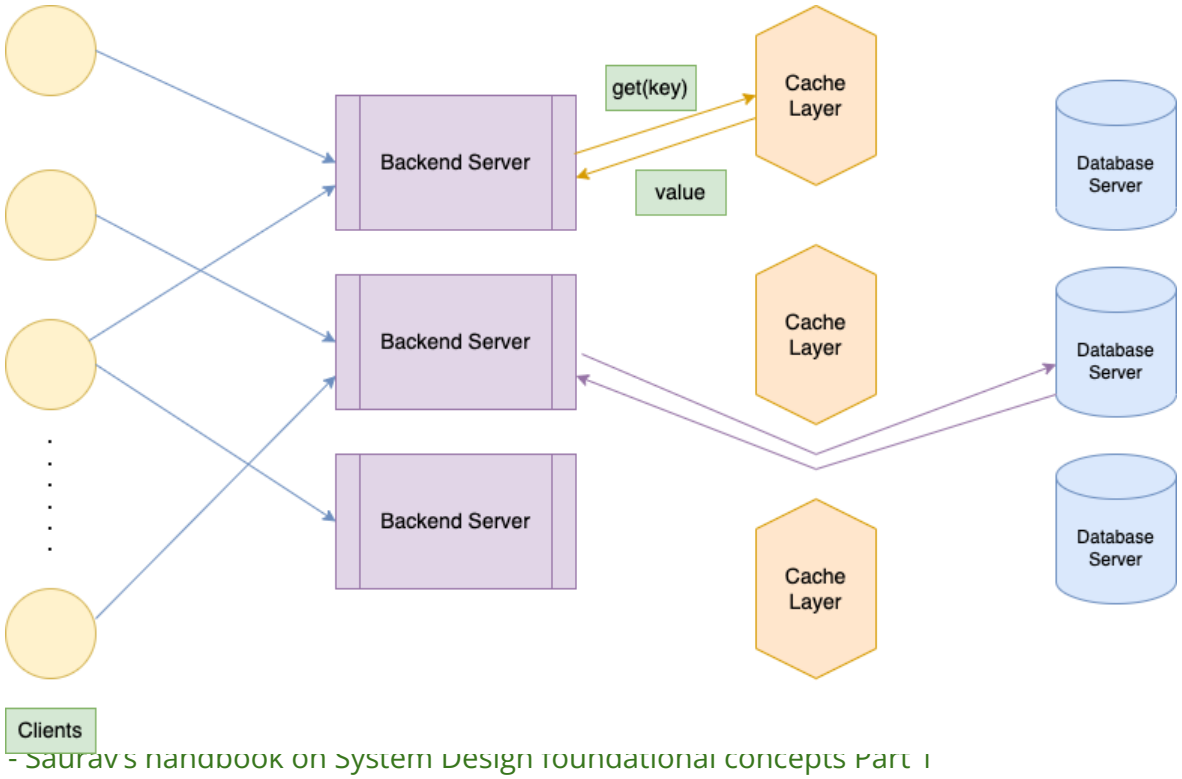
Now we have multiple **Database servers**, each server storing data around a certain key range. We took the entire data of the system, divided it into multiple buckets on the basis of a key, each bucket storing data around a certain range of keys and then kept each bucket in one Database server. Technically we shard our data.

This can work well and can reduce the load to some extent when our system scales. But this architecture has some drawbacks of its own. Suppose we have a scenario where data with a particular key is in huge demand and in that case the database server holding that key will get enormous load and may wear out. I have also discussed **Database Sharding** in my previous article along with its drawbacks, you can head over to it for more details on sharding.

Since we saw multiple drawbacks for this architecture, let's move on to another architecture in which we will be introducing a caching layer in our system.

Architecture 3 : Introducing a Caching Layer

In this architecture we will introduce a caching layer in between the Backend servers and the Database servers.



Since we have introduced a caching layer in between, now the backend servers will look for the information in the cache and if it's absent the request will be further sent to the database server in order to fetch the information. The Cache works much faster than the normal database servers, hence we can perform faster reads from the cache as compared to the DB servers.

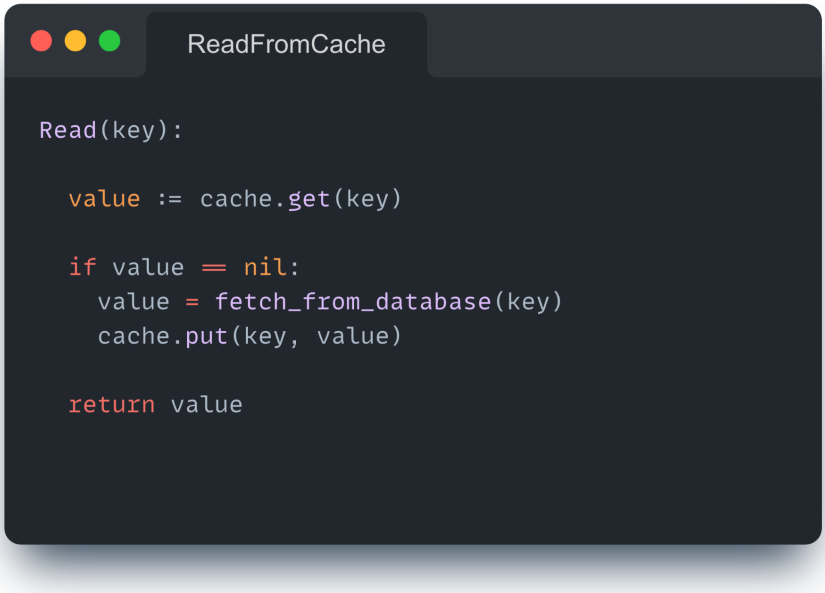
This resolves our previous issue of hot keys. If a particular information with a certain key is in huge demand then we can simply cache that data along with its key. This will avoid the requests to further hit the database servers since they will be handled by the Cache layer.

Note: We will still be required to send the Write requests to the Database servers. Since we need to keep the data consistent.

Reading and Writing from a Cache

Cache is indeed a very simple concept to work upon. A general cache provides two major operations i.e. **read** and **write**. Along with these basic operations a cache library can also involve multiple other operations like removing from the cache and much more.

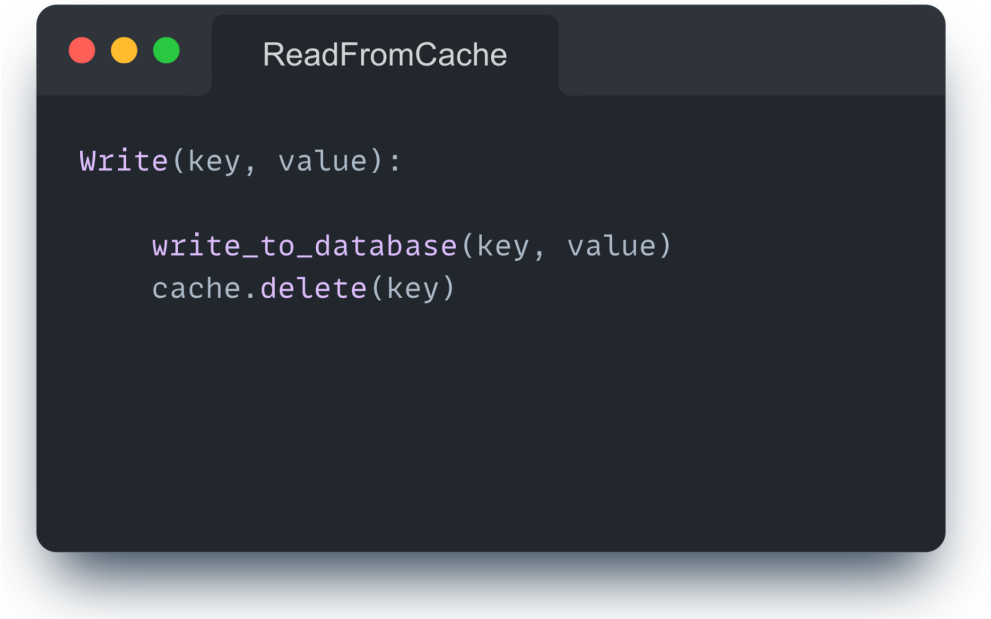
Let's look into the **Read** operation provided by a caching library.



20 - Saurav's handbook on System Design foundational concepts Part 1

The above code block explains the **Read** operation of a **Cache**. When performing a lookup in a cache with a particular key it checks for the presence of the key in its storage. If the value for that key is not found, then another request is issued to the database server from where we fetch the value for that key. Then the key-value pair is stored in the cache and the value is returned.

Now let's have a look into the **Write** operation of the **Cache**.



We earlier discussed that all the writes will be sent to the database. Here we have done the same. The key-value pair is written/updated to the database and then deleted from the cache. This avoids the clients from reading stale data from the cache in future. Once the key is removed from the cache, an incoming future request will fetch the updated value from the database first and then store that back to the cache. This solves the problem of data inconsistency to some extent.

Chapter 4

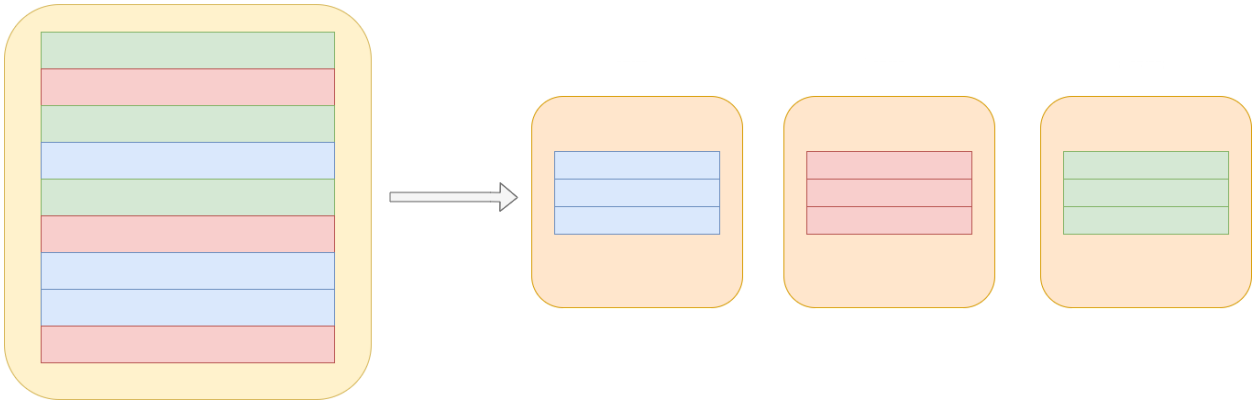
Database Sharding

Database Sharding has always been an essential concept under System Design and has always been a major part of multiple case studies. Let's discuss the concept in this chapter, its types and what can be the drawbacks of sharding a database.

Introduction

Initially when you have a small user base then you could have your entire data in a single database placed in one local machine. And your system will still perform accordingly. But what if the user base starts growing. The size of the database will increase and then keeping your entire db in a single place could make your system slow. The queries may take a lot of time to get executed. So, there are multiple solutions which you could adopt in order to optimise your queries. You can go with Indexing, use **NoSQL Databases** or can perform Sharding.

When your database grows, then you could divide your database into multiple smaller parts and store each part separately in different machines. The smaller parts are called Shards and the entire process is known as **Database Sharding**. It can be a possible solution when your database becomes too large to be stored in a single machine. There are various ways in which you could possibly shard your database. These are discussed below.

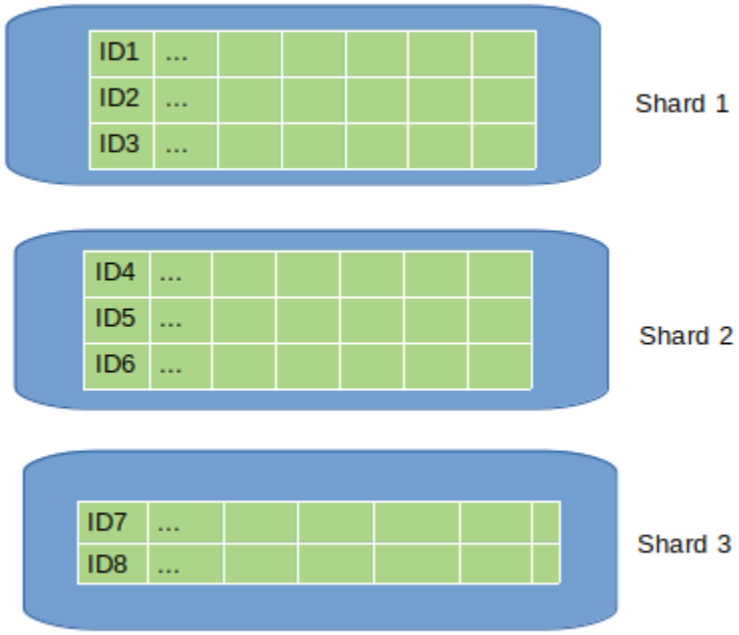


Horizontal Sharding

When a single table in your database becomes large then you could possibly divide that one table by **rows**. Now that one large table gets divided into multiple smaller

tables with the same schema but having different dataset. This process is known as **Horizontal Sharding** or **Horizontal Partitioning**.

Suppose you have a table having **1 Million** dataset and you are planning to store them in **5** different local machines. Then you could divide the entire dataset into 5 smaller tables each having a size of **200**,**000** datasets. You could possibly use some algorithm to divide your dataset into multiple machines in a well distributed manner.

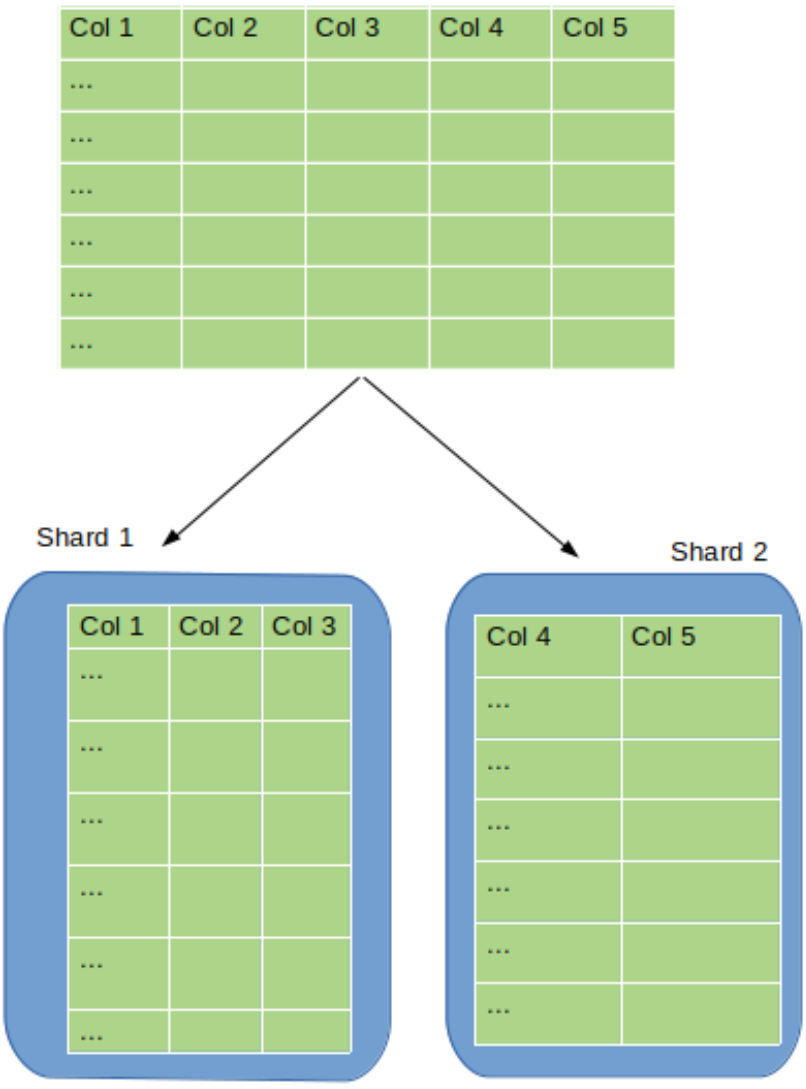


Vertical Sharding

When you have multiple tables in your database then you could put multiple tables in multiple different machines. Or you could also divide a single large table column wise and store them in multiple machines. This process of partitioning is known as **Vertical Sharding** or **Vertical Partitioning**.

Suppose you have 3 multiple tables in your database each storing different types of datasets. One storing user information, second one storing all the media files and

the last one storing organisation details. Now you could store these three tables in 3 different machines. When you are required to query the database then you would be required to know which type of data you are querying for and then look for that data in the machine which is responsible for storing that table.



Sharding can be a bad idea?

Sharding is not always a good idea. Before going with Sharding you should also look into other possible alternatives which could be less complex and still your system can be reliable.

- **Programming and Operational Complexity**: When you partition your database and store them in different machines then it could increase the complexity of your system. When you query your database then there could be a situation where your query will be sent to multiple machines and then datasets will be fetched according to the query and will be needed to join back when these results are returned to the system. This entire process can be expensive and complex as well. Maintaining a Sharded Database is also difficult. As you have multiple machines with yourself you need to be responsible for maintaining all of them and if any one of them fails then it could lead to data unavailability.

- **Hotspots in Sharding**: As you have multiple machines which are responsible for storing your data, you need to ensure that the incoming data to be stored gets equally distributed among the available machines. If this doesn't happen and one of the machines gets most of the data then in turn it will be responsible for answering most of the queries. Hence it will be having a maximum portion of the entire load and may wear out. This phenomenon of concentration of the entire load at a single machine is known as the existence of **Hotspot** in **Sharding** and that single machine is known as **Hotspot**.

- **Less Flexible**: If you have a certain number of machines among which your data sets are distributed then it's really complex to increase or decrease that number. It's complex to add or remove machines from the existing sharded database design. In order to avoid this problem Consistent Hashing is used which can provide flexibility to your system implementing sharded databases.

Algorithmic Sharding

There are multiple ways to partition the data into several machines. One of the methods is Algorithmic Sharding where a certain algorithm is used to determine the destination machine where the dataset is needed to be stored. The process is fast as you don't need to be dependent on any external device or service to determine the location of your data storage. The algorithm can be in the form of a **Partition Function** or a **Sharding Function** which can take the Partition ID or Partition Key as an input and return the Address or ID of the machine where that corresponding dataset is stored. This sharding function distributes the dataset into multiple machines. You should ensure that your sharding function distributes the data uniformly among all the available machines and avoid the existence of hotspots.

Suppose you have **7** machines which you will be using to shard your Database. You can build your algorithm which can distribute the datasets into these **7** machines.

One possible algorithm for dividing the datasets can be:

ShardFunction

```
function shard_function (partition_key) {

    var number_of_machines = 7;
    var destination_machine = partition_key % number_of_machines;

    return destination_machine;
}
```

According to the above algorithm, when multiple datasets having following IDs enter the system then it gets directed to the following machine IDs.

S.No.	Dataset ID	Shard Function	Machine ID
1.	121	shard_function(121)	2
2.	45	shard_function(45)	3
3.	195	shard_function(195)	6
4.	21	shard_function(21)	0
5.	67	shard_function(67)	4

Dynamic Sharding

In **Dynamic Sharding** an external service is used in order to determine the location of the machine in which the dataset will be present. It doesn't use any algorithm to determine the address of the destination machine. Every request coming to the system needs to go through the external services in order to fetch the address of the machine. Hence this process is comparatively slower than **Algorithmic Sharding**.

Moreover every request coming to the system is somewhere dependent upon the external locator service and if the service fails then the entire system can get jeopardised. This introduces a **Single point of Failure** in the System. On the other hand Dynamic Sharding is more resilient to non uniform distribution of data.

Thank You! ❤️

Hope this handbook helped you in clearing out the foundational concepts of System Design.

I will be soon coming up with **Part-2** of this handbook where we will discuss a few more foundational System Design topics and interview tips. Stay tuned!

Follow me on Linkedin
Do subscribe to my engineering newsletter "Systems That Scale"