System Design

Design Performance Metrics

1. ***Reliability –*** *Probability a system will fail during a time-period. Less the prob, more reliable it is****.***
2. ***Scalability –*** *ability to grow and manage increased traffic, increase data and request.*
    a. ***Web servers – use Load Balancers.***
    b. ***Database Servers – use Partitioning, Caching, Indexing and Replication***
    c. ***Asset Server(Image/Video etc) - CDNs***
3. ***Availability –*** *Amount of time, a system is operations during a period. It can be increased by replication. But replication increases redundancy which may not be reliable.*
4. ***Manageability –*** *how easy to maintain the system updates, bug fixes etc*

System architecture

Monolith architecture*: - functionalities coupled tightly i.e., everything is touching everything.*

- *Can scale horizontally in load.*
- *Good for small team, less interaction needed to understand things.*
- *Less parts because it is not broken into small parts.*
- *Common Test helper methods can be used.*
- *It is faster, because no Remote procedure calls needed to connect with other system for some functionality.*

Disadvantages

- *New team members need to understand whole system to make even small changes.*
- *Deployment is heavy and testing too.*
- *If once service crashes, whole system may fail too.*

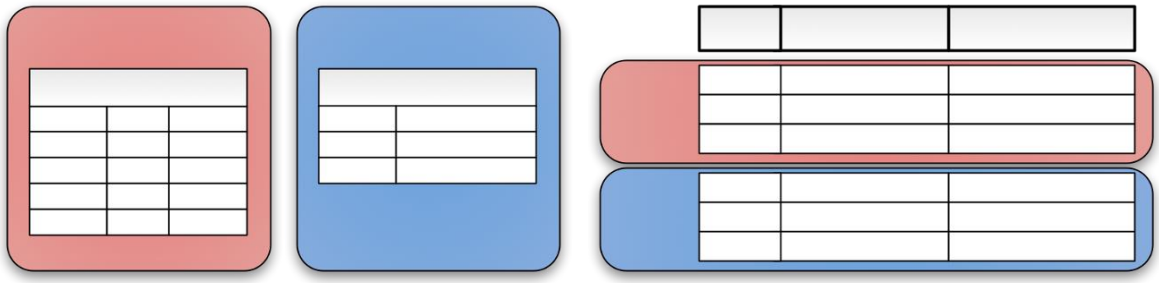
Microservice architecture *- functionalities coupled loosely, independent services.*

- *Easier to scale, also, partially scaling is possible, i.e., the service needs scaling will be scaled only.*
- *Good to big teams, as new members need to understand only task related service.*
- *Parallel development is easy.*
- *Deployment is easy.*

Disadvantages

- *Not easy to design. Need skills to identify which service needs to be clubbed as one service and which not to be.*
- *Less fast than Monolith, because of network calls.*

Database Scaling/partitioning.



Vertical partitioning *involves creating tables with fewer columns and using additional tables to store the remaining columns*. Normalization also involves this splitting of columns across tables, but vertical partitioning goes beyond that and partitions columns even when already normalized.

Horizontal partitioning *involves putting different rows into different tables*. Perhaps customers with ZIP codes less than 50000 are stored in CustomersEast, while customers with ZIP codes greater than or equal to 50000 are stored in CustomersWest. The two partition tables are then CustomersEast and CustomersWest, while a view with a union might be created over both to provide a complete view of all customers.

Sharding: 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.

- speed up query response times, because less rows needs to be scanned.
- If one shard goes down, you whole service using that table won't go down.

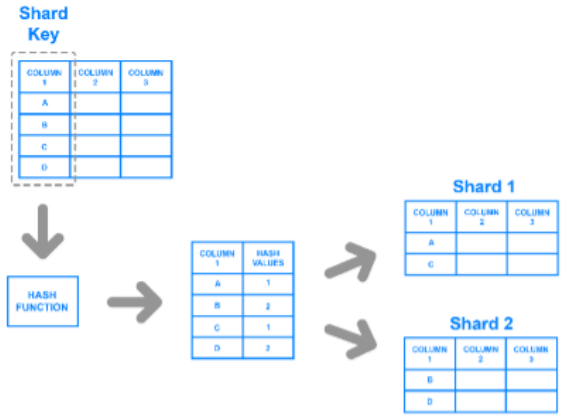
Disadvantages:

1. Hard to Decide the **key** on which shard needs to happen.
2. Joins across shards, to get the required data, will be slow
3. difficult to return it to its unsharded architecture

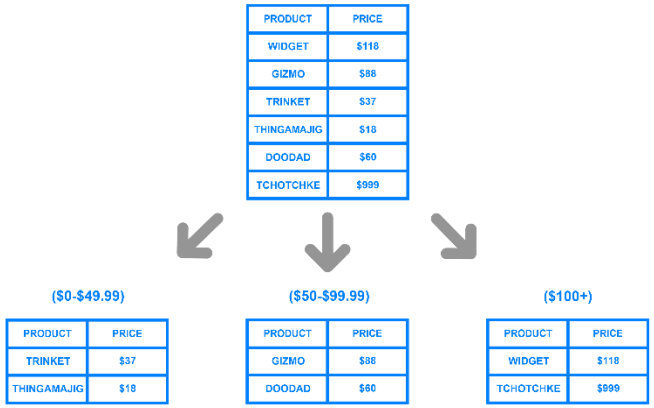
Horizontal partitioning *splits one or more tables by row, usually within a* ***single instance*** *of a schema and a database server,* ***Sharding*** *splits the tables across potentially* ***multiple instances*** *of the schema.*

Sharding Architectures

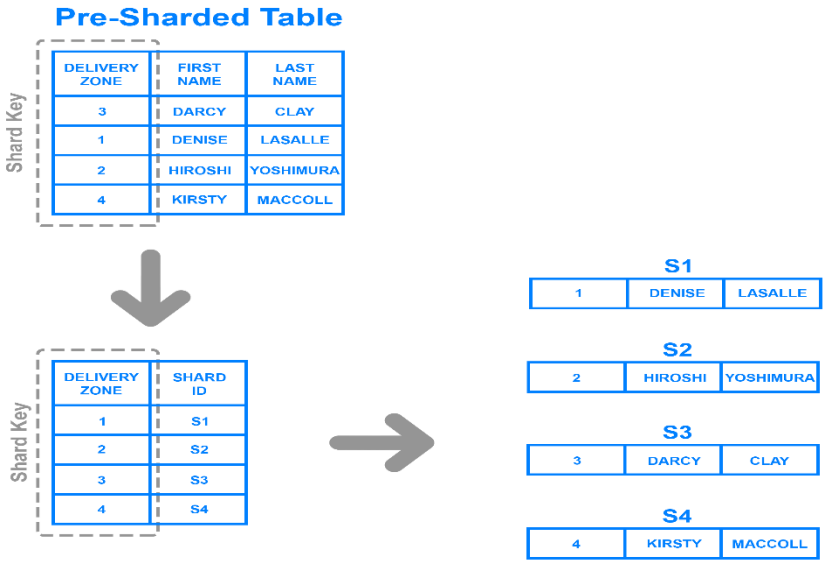
1. ***Key based*** – *Consistent Hashing.*



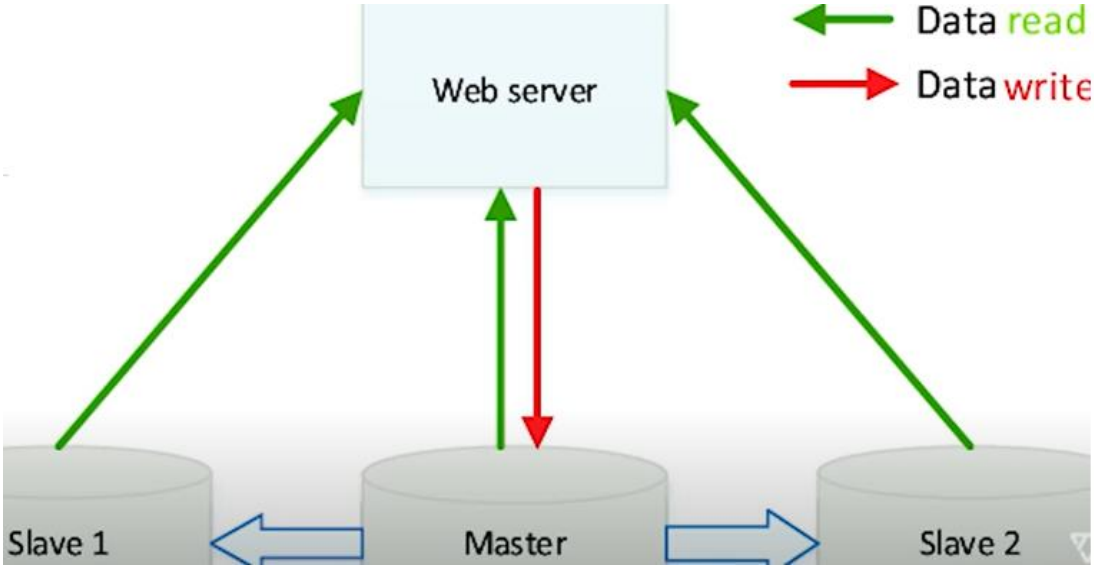
2. ***Range Based***



3. ***directory based-*** one must create and maintain a *lookup table* that uses a shard key to keep track of which shard holds which data.

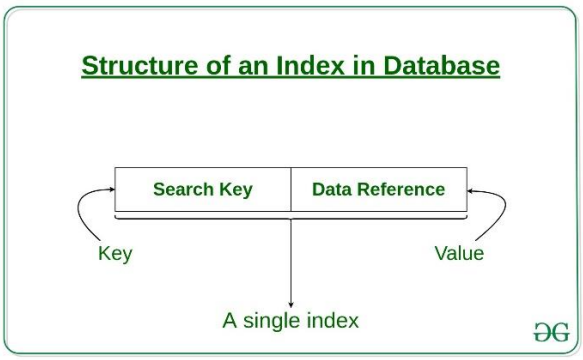


Replication – Using Master Slave



Indexing

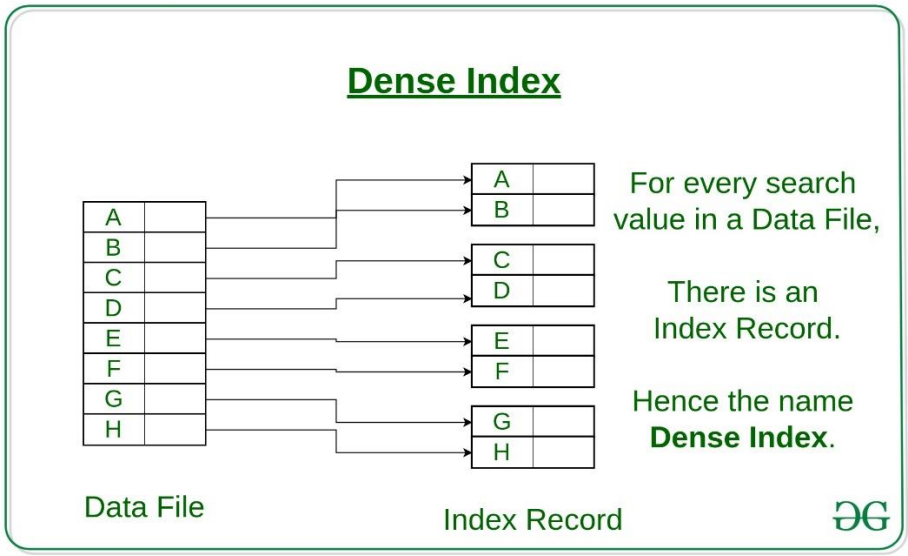
An index is just a data structure that makes the searching faster for a specific column in a database. This structure is usually a **b-tree** or a **hash table** but it can be any other logic structure.



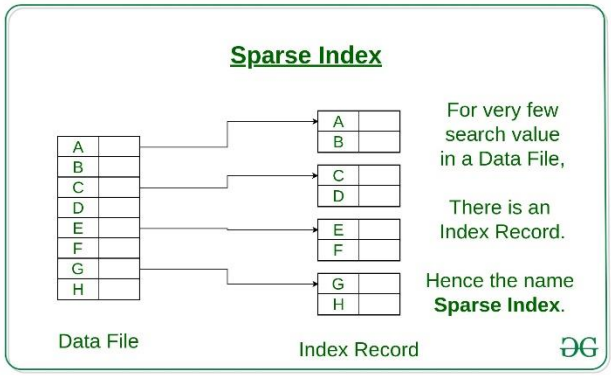
Index Table population techniques

Ordered Indexing (B-Tree): In this, the indices are based on a sorted ordering of the values.

1. Dense Indexing – every record has an index.



2. Sparse Index :- Only few records have an index to block of records. To find an index, we scan the index table till the (index key > required Key) value possible. For example, To Find F, we scan till G as G > F, so E will be taken and it's block will be scanned sequentially.

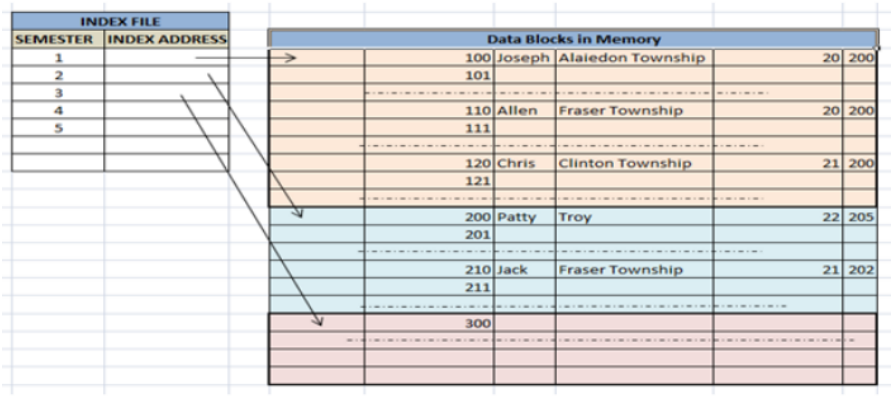


Un-Ordered Indexing (Hashing): the buckets to which a value is assigned is determined by a function called a hash function.

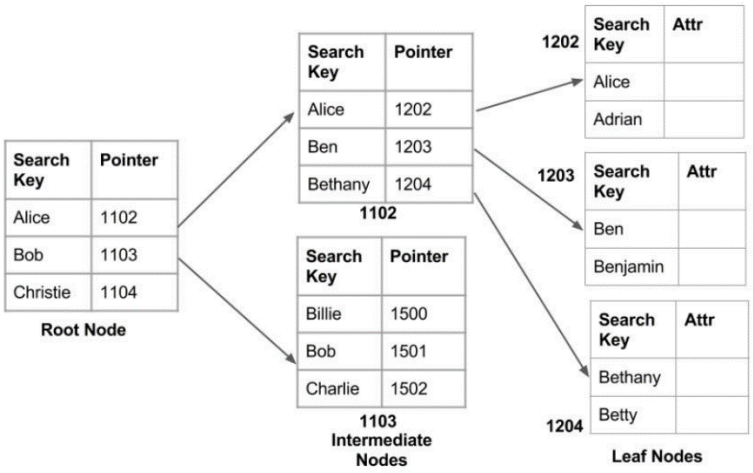
The disadvantages of a hash index is ***it will not be able to find key for range queries****, because a* ***hash table is only good for looking up key value pairs – which means queries that check for equality.***

There are primarily three methods of indexing:

1. Clustered Indexing *- records with similar characteristics/ groping attributes are grouped together and indexes are created for these groups. Index table is sorted by Search key, or groping attributes.*

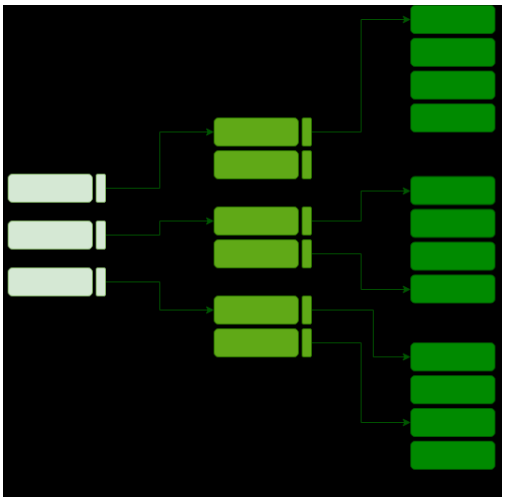


2. Non-Clustered or Secondary Indexing – *provide pointers to another index file, and from there, to DB.*

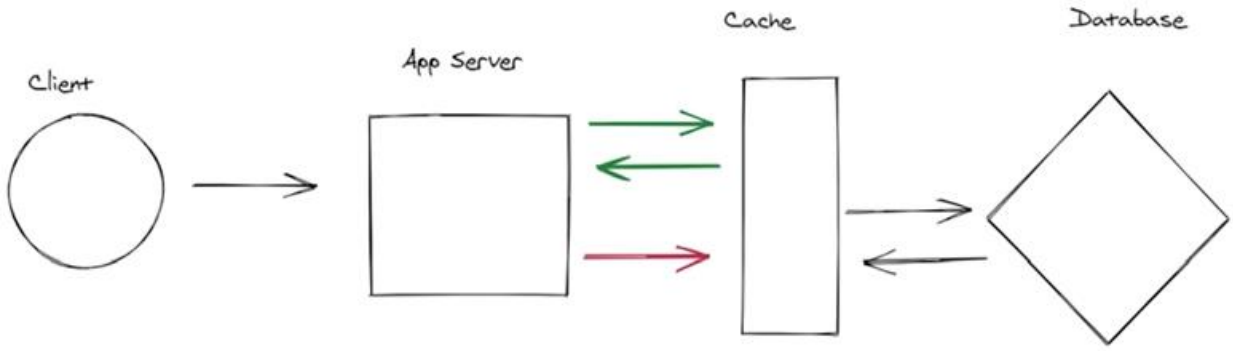


Non clustered index

3. Multilevel Indexing *- The multilevel indexing segregates the main block into various smaller blocks so that the same can stored in a single block. The outer blocks are divided into inner blocks which in turn are pointed to the data blocks****.***

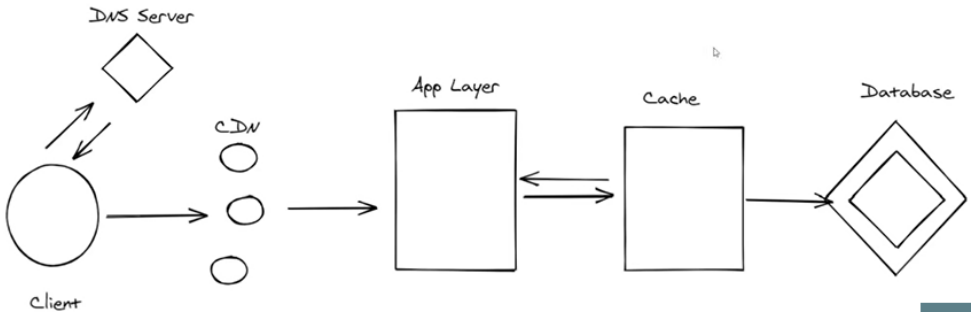


Caching



1. Improve performance of application, Can serve same amount of traffic
2. Systems have more read request than write, should use it.

Caching Layers



Cache Entry Deletion

Time to Live – Set a period before a cache entry is deleted. It is used to prevent data which is not being used for a long time.

LRU/LFU – caching techniques to remove cache entries.

Cache Insertion (Hit/Miss)

1. (Mostly used) **Cache Aside** – check in cache, if not present, take from DB
2. **Read Through** - When there is a cache miss, it loads missing data from database, populates the cache and returns it to the application.
3. **Write Through** – write in Cache then in DB, for heavy systems.
4. **Write Back** – Write In cache, and cache updates data in DB.

Some common cache systems - Memcached, Redis, Cassandra

We can use CDN to cache static files, JS files, images.

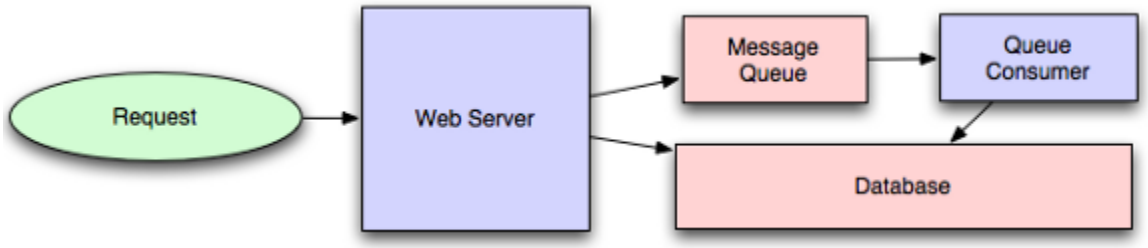
A CDN (Content Delivery Network) is a highly distributed platform of servers that helps minimize delays in loading web page content by reducing the physical distance between the server and the user. This helps users around the world view the same high-quality content without slow loading times.

CDN setup

1. Pull technique – CDN fetches file from server and cache it, takes time to load first, then faster for subsequent request.
2. Push technique – push files on CDN directly, more costly.

Message queues

1. It provides temporary storage between the sender and the receiver so that the sender can keep operating without interruption when the destination program is busy or not connected.
2. It is an **Asynchronous processing** allows a task to call a service and move on to the next task while the service processes the request at its own pace.
3. Messages placed onto the queue are stored until the consumer retrieves them.
4. the queue can provide protection from service outages and failures.
5. Examples of queues: Kafka, Heron, real-time streaming, Amazon SQS, and RabbitMQ.



Reference - https://medium.com/must-know-computer-science/system-design-message-queues-245612428a22

Load Balancing - Why do we need Consistent Hashing?

Let us assume the hash function is **H(req) = req.id % N,** where N is number of servers.

Now, since **req.Id** in real world generally doesn't change, for example, in social media, req.Id could be username of person. So, it is unique across the system. And when we find the hash value of it, we will get the same value every time. And it will be redirected to same server every time. Since the server gets same set of requests, it will cache data for each request for faster service.

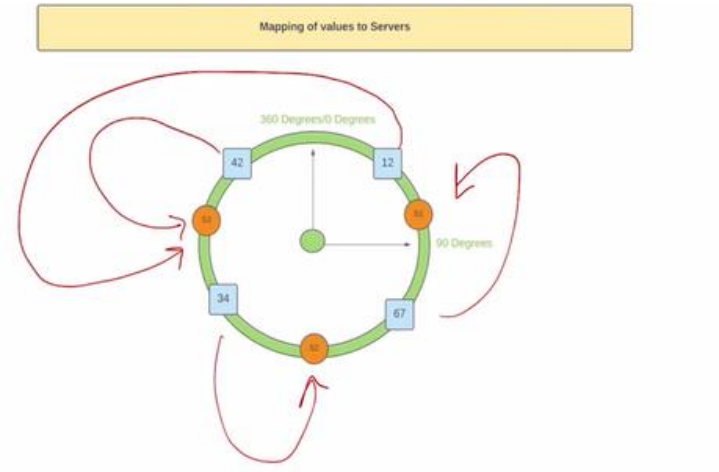
Now, if we want to add or remove servers from the system, the N changes, and because of which every request hash value change, due to which cache maintained by any server may be of no use. They need to recreate it from the start.

Also, if any server fails, it must be removed, in the same way, N changes, so as the hash value. The new hash value might cause uneven distribution of requests, which may cause some servers busy all the time and some remain idle. The System will suffer latency and is less fault tolerant. We need to change the policy of adding/removing the servers in system so that, it will not affect the latency, server's local data and will remain fault tolerant.

Consistent Hashing

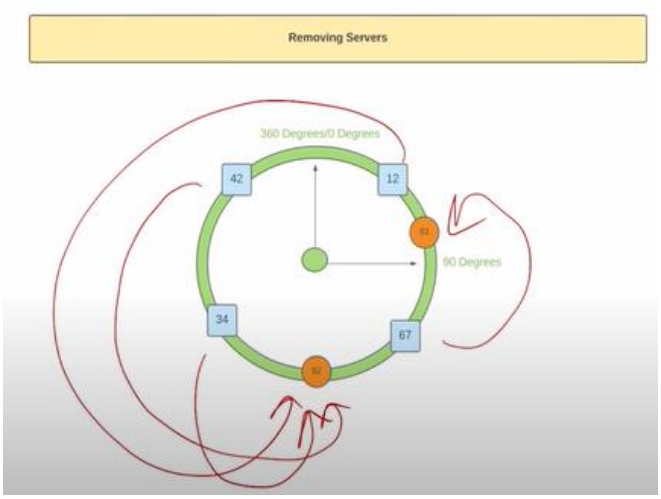
Basic concept of this policy is, get hash value of request and hash value of server. Assigning them a position on an abstract circle, or hash ring. As soon a request gets assigned to hash ring, it will be served by next server ahead of it, if taking clockwise, or behind of it, if taking anticlockwise.

For example, Let us say N in hash function is 100. i.e. the request range, And output value if Hash function is 360. i.e. 0-360. In the below image



Now, let us see how it has solved out problem of adding and removing nodes.

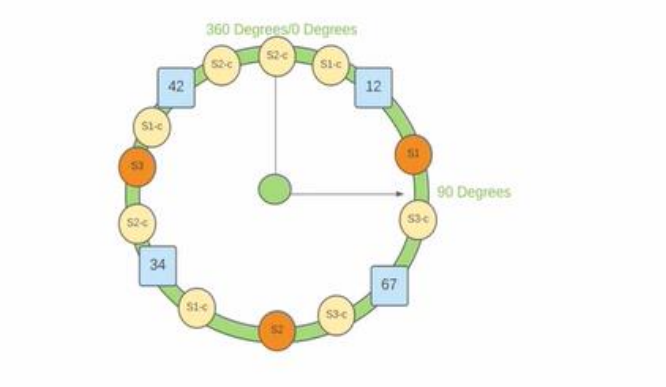
If any of the server goes down, lets sat S3 in above example, all the request which were supposed to be delivered by S3, will find the next active server S2, without affecting any other servers in system.



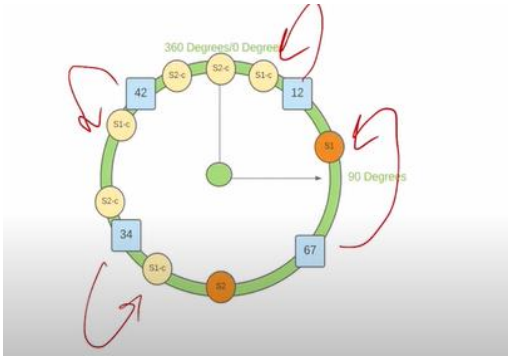
So, in this case, no other servers will be affected in terms of Latency and local storage.

The problem with the approach is, Once the S3 gets removed, load on S2 increases and it might get overloaded which can cause its failure.

Since requests need to be evenly distributed. Among all the servers. For this, we can replicate the Server instances on the hash ring, by using different hash functions.



Now, if S3 goes down, all the requests that were supposed to serve by S3, will distributed among all other server instances.



Airline reservations system – MakeMyTrip

Requirements

1. Search flights based on Demographics.
2. Flight booking based on Search results.
3. Payment process and notifications. **In all booking systems, these 2 modules(Payment and notifications) will be common**.

Imp Notes.

1. User does not Search a flight, he searches the schedule, because a flight can go A->B on some day and reverse on someday.
2. Every schedule can have different time, fare, seats, source and destination.
3. Airport has flights of different airlines.
4. An Airline has set of flights.

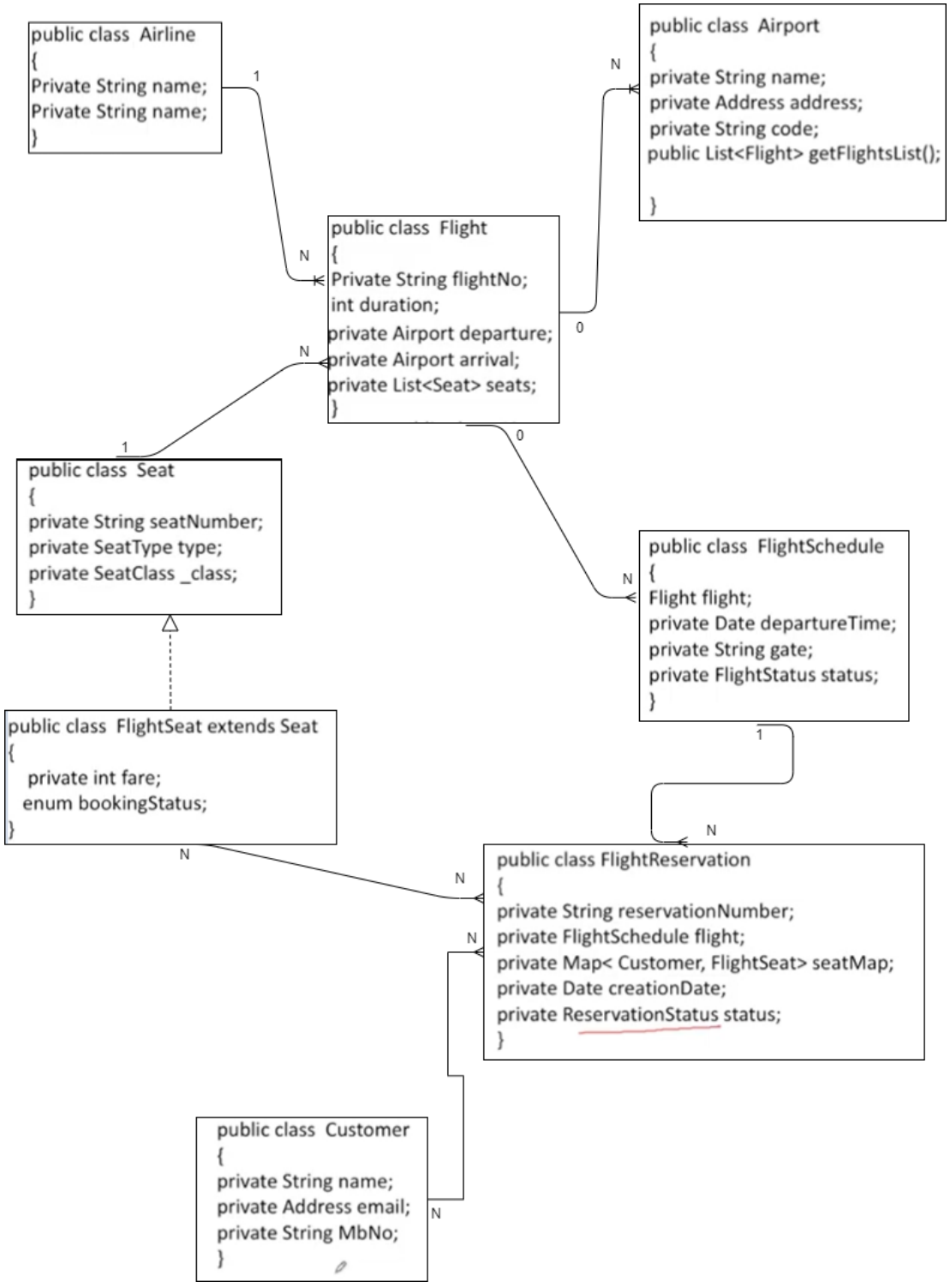
Entities:

1. Airline – can have many flights
2. Airport – Contains flights of different airlines
3. Flight – have seats
4. Seat
5. Flight Seat – instance of seat at any moment with current fare.
6. Flight Schedule – Schedule of a flight with source and destination, time, seats available and fare.
7. Flight Reservation – Schedule chosen by customer, with number of seats belongs to that flight.
8. Payment – using different Payment gateways.
9. Notification – Email, sms etc

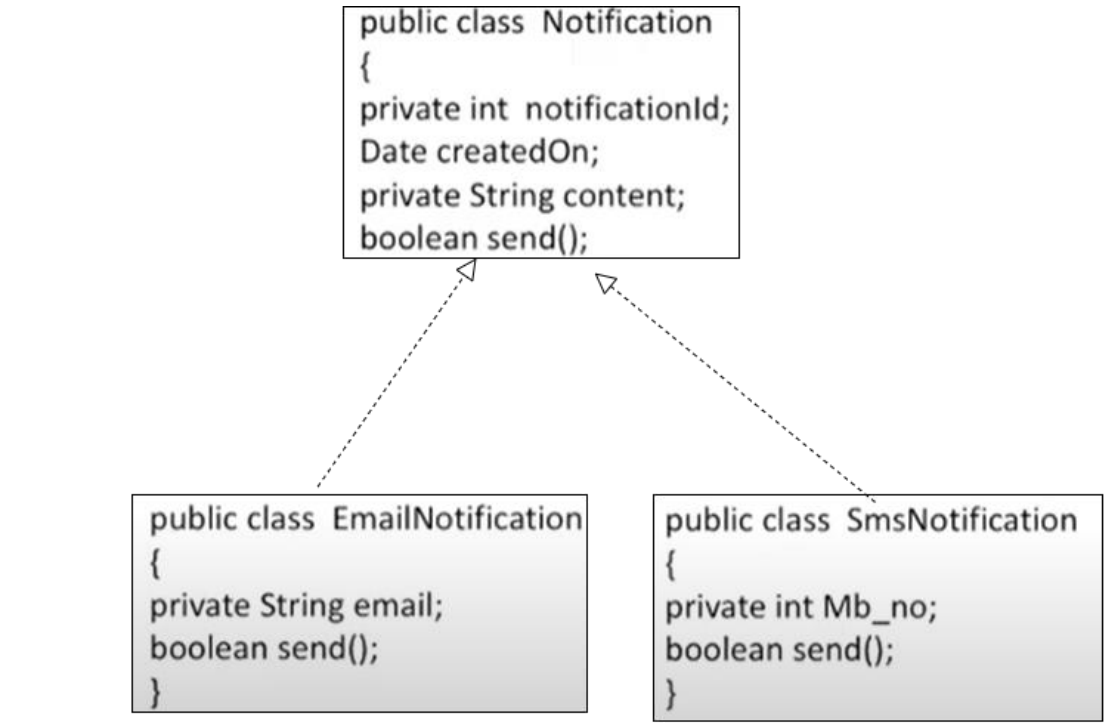
Airline Reservation System..drawio

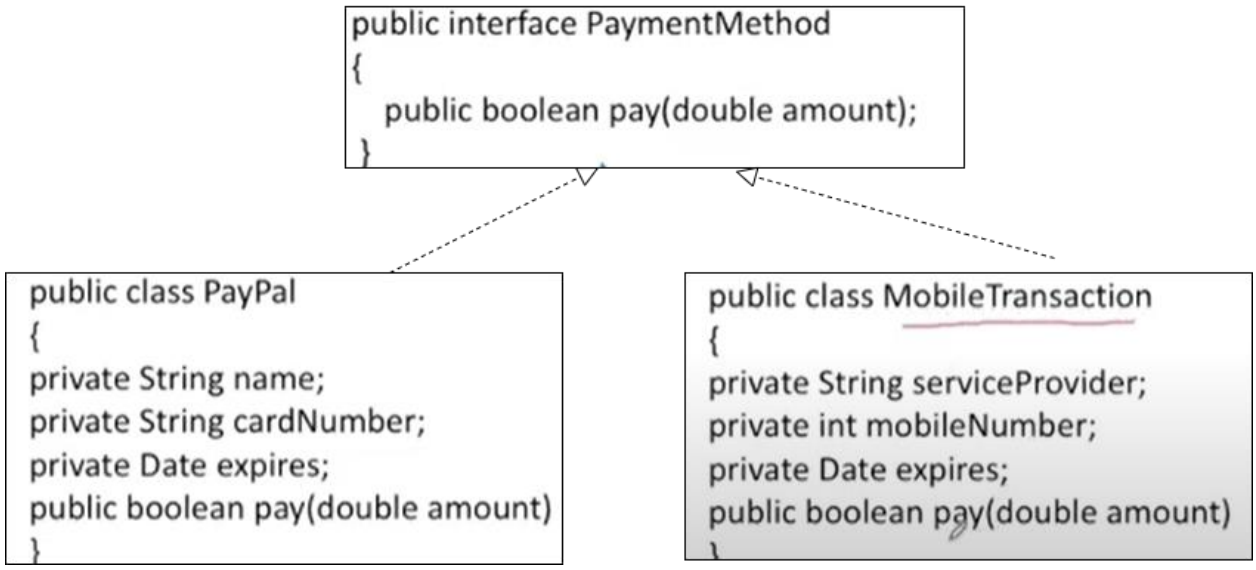
Payment and notifications.drawio

ER Model/Class Diagram for Search and reservation



Payment And notification

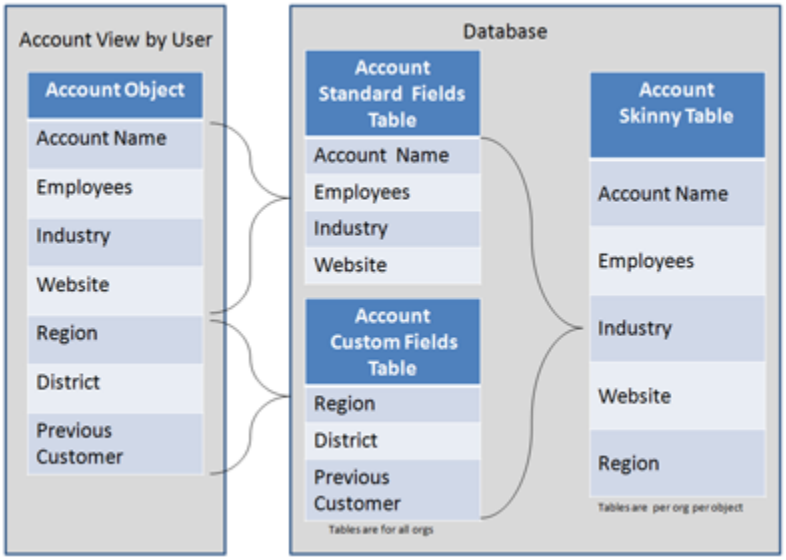




Skinny Tables in Salesforce – View in SQL

Salesforce can create skinny tables to contain frequently used fields and to avoid joins. This can improve the performance of certain read-only operations. Skinny tables are kept in sync with their source tables when the source tables are modified.

If you want to use skinny tables, contact Salesforce Customer Support. **When enabled, skinny tables are created and used automatically where appropriate. You can't create, access, or modify skinny tables yourself**. If the report, list view, or query you're optimizing changes—for example, to add new fields—**you'll need to contact Salesforce to update your skinny table definition.**



- Skinny tables can be created on custom objects, and on Account, Contact, Opportunity, Lead, and Case objects.
- Skinny tables can contain the following types of fields.
    - Checkbox
    - Date
    - Date and time
    - Email
    - Number
    - Percent
    - Phone
    - Picklist (multi-select)
    - Text
    - Text area
    - Text area (long)
    - URL

- Skinny tables and skinny indexes can also contain encrypted data.
- Skinny tables can contain a maximum of 100 columns.
- Skinny tables cannot contain fields from other objects.
- For Full sandboxes: Skinny tables are copied to your Full sandbox orgs.
- For other types of sandboxes: Skinny tables aren’t copied to your sandbox organizations. To have production skinny tables activated for sandbox types other than Full sandboxes, contact Salesforce Customer Support.