SCALING: VERTICAL & HORIZONTAL

- Let your server is hosted on a cloud, then the business requirement could be:
  - There are a lot of customers are using your server. But Can the server handle this much load?
  - So, the following solutions can be possible
    - Buy Bigger Machine (Vertical Scaling)
    - Buy More Machine (Horizontal Scaling)
  - These things are used to increase the Scalability of your system.

	Horizontal Scaling	Vertical Scaling
1	◦ Load Balancing required	◦ N/A
2	◦ ***Resilient*** (Able to recover readily)	◦ ***Single Point of Failure*** (if the machine goes down, then its gone. As only 1 machine is there)
3	◦ Network Calls (*RPC: Remote Procedure Calls*) ◦ The systems will be communicating with each other through network calls. So, comparatively ***Slower***.	◦ Inter-process communication. So, ***Faster***.
4	◦ Data is ***Inconsistent***. ◦ Data is complicated to maintain. ◦ If a transaction has to be made, the operation has to be atomic. ◦ So, we have to lock all the server which is impractical. So, ***loose transactional guarentee***.	◦ One system is there where the system resides. ◦ So, here, Data is ***Consistent***.
5	◦ It is ***Scalable***. ◦ We can add more system to scale up the system according to the number of customers.	◦ ***Hardware Limitation***. ◦ We can't make the computer bigger and bigger and bigger.

- In real-world, both Horizontal & Vertical Scaling are used.
  - We can take points 2, 5 from Horizontal Scaling & 3, 4 from Vertical scaling (Positive points).

- Initially we'll *Vertical Scale*, and at the point where we can't scale it vertically more, we'll use *Horizontal Scale.*

LOAD BALANCING & CONSISTENT HASHING

- Let you have ***N*** no. Of servers and A user sends one request, "which server that request will go to?"
- All the servers are carrying ***Load*** on it to process the incoming requests.
- Concept of taking ***N*** no. Of servers and trying to balance the load among them ***evenly*** is called **Load Balancing** .
- You'll get a request ID whenever the user sends the request to the server.
  - That request ID is **uniformly random** i.e. ***0*** *..to..* ***M-1*** (uniformly random: probability of getting any thing out of a list is same I.e. suppose an array of numbers is there and you are told to pick one among those numbers)
  - **H(r)** ⟶ **M** (H: Hash, r: request ID)
  - The server to which the request will go is: **M % N** (N: No. Of Servers)
- Let you have 4 servers, S0, S1, S2, S3.
  - H(10) = 3 (let) ; 3 % 4 = 3 ; It'll go to S3
  - H(20) = 15 (let) ; 15 % 4 = 3 ; It'll go to S3
  - H(35) = 12 (let) ; 12 % 4 = 0 ; It'll go to S0
- So, as the request IDs are uniformly random, also the hash function is uniformly random;
  - So, each of the servers will have ***X/N*** **loads if there are** ***X*** **requests in total.**
  - So, the *Load Factor* is ***1/N.***
- **But, what happens if we need to add more servers?**
  - Let 5 servers are there now. S0, S1, S2, S3, S4.
    - H(10) = 3 (let) ; 3 % 5 = 3 ; It'll go to S3
    - H(20) = 15 (let) ; 15 % 5 = 0 ; It'll go to S0
    - H(35) = 12 (let) ; 12 % 5 = 2 ; It'll go to S2
  - So, all the things now changed.
  - We can find the changes via a ***pi*** chart.
    - Initially we had 4 servers. So, the ***pi*** was having 25% keys per server. (Key: Request ID)
    - Servers were having keys S0(0-25), S1(25-50), S2(50-75), S3(75-100).
    - One more server added in the system. So the following would happen now.
      - **S0** : It will be having 0-20 now. Loosed 5%. Change = 5; total=5
      - **S1** :It will get 5 of S0 now. Means 20-50. Gained 5%. Change = 5; total = 10

- Again, S1 will be having 20-40. So loosed 10% i.e. loosed 40-50.
  Change = 10; total = 20
- **S2**: It'll get 10 of S1 now. Gained 10%. Change = 10; total = 30
  - Again, S2 will be having 40-75. So loosed 15% i.e. loosed 60-75.
    Change = 15; total = 45
- **S3**: It'll get 15 of S2 now. Gained 15%. Change = 15; total = 60
  - Again, S3 will be having 60-100. So loosed 15% i.e. loosed 80-100.
    Change = 20; total = 80
- **S4**: (new server): It'll gain 20%. Change = 20; total = 100

- From the above experiment we can see that the change of **Key-to-Server** mapping went to **100%** which is not good.
- So, total is **100%** changes.

- It is a ***Simple Hashing***. **Not *Consistent Hashing***.

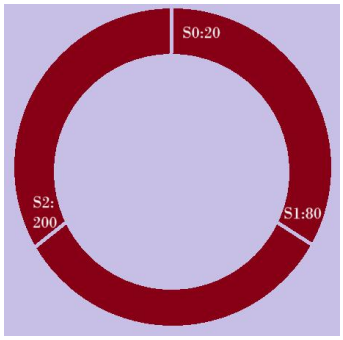
Simple Hashing

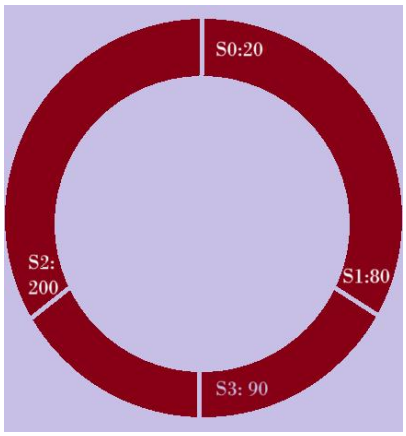
- *Hash(Key) = Val*; *Val % N = Particular server* (Key: Request Id)
- Here, the assignment of the hash to the server is not consistent. Means if the number of servers increases, the key assignments also changes.
  So, this is not **Consistent Hashing**. Here the change factor can be go up-to **100%**.

NOTE

- In practice, the Request ID (Key) is constant.
- So, if a particular key is mapped to a particular server, then it would be better if we store some frequently accessed data of that particular user as **Cache** rather than accessing the database repeated time.
- But, what in case of Simple Hashing, the Key-to-Server mapping depends upon the number of Servers present in the System.
- So, if we store the frequently accessed data of a particular user in a server, and the number of servers changes, then that cache will be dumped. And will be of no use.

- **We can take 5% of each server out of those 4, and assign those to the new 5th server S4. Then the change will be only 20%.**
  - **Only 20% of the keys will get affected.**

- **Consistent Hashing:**
  - Technique:
    - Hash the Servers and keep those in a circular ring.
    - Hash each Keys and keep those in that circular ring.
    - Assign each Key to its next Server in *clockwise* direction.
    - **Server's hash >= Key's hash** in **clockwise** direction, will be assigned to the key.
    - Let we have 2 servers; S1:30, S2:40; and the req:30
      - So, it'll go to *S1*, as 30(S1)>=30(req) True.
  - Example:
    - Let we have 3 servers having hash as follows
      - S0 = 20
      - S1 = 80
      - S2 = 200
    - 

      - 21 to 80 will go to S1:80
      - 81 to 200 will go to S2: 200
      - Remaining will go to S0: 20
    - Now, one server was added
      - S3 = 90
    - 

      - 21 to 80 will go to S1:80

- 81 to 90 will go to S3: 90
- 91 to 200 will go to S2: 200
- Remaining will go to S0: 20

- Here, only assignment of server for only 10 keys (81 to 90) got changed.

NOTE

- Let the ring is having capacity 0 to M.
- The user counts are N. It means the hash counts will be N (unique hash)
- If ***N < M***; there is no issue as the user will be anyhow mapped to the server that comes just next to it in clockwise direction.
- What if ***N > M***;
  - Here, ***hash(user)%M*** will be calculated.
  - Consider the following scenario:
    - Let M=20, N=30
    - Let S0:1, S1:10 is there inside the ring.
    - So, users having hash 2 to 10 will be using S1.
    - If a user comes whose hash will be more than 20; let 25
    - Now the value = 25 % 20 = 5
    - So, it'll use S1 now.
    - Its not a fault because, the users (having hash from 2 to 10) were already sharing S1 for their requests.
    - ***Even the modulo of server's hash will be stored.***

In case of *consistent hashing*, the drawback is **server load distribution**.

- S1:10, S2:11, S3:12, S4:90
- Here we can clearly see, S4 is overloaded; handling almost 78% of the key space.
- This is the main drawback of **Basic Consistent Hashing**.
- If we will be able to place the same server at multiple place of the rings, then it'll solve this problem.
- Like S1, S2, S3, S1, S4, S1, S3, S2, S4, S2, S1.. like this (of course in a ring only)
- If we take multiple hash functions.
  - Let we have **k** hash functions
  - So for each server, we'll be having **k** hashes.
  - So, one server can be placed at **k** points in the rings.

- Same for all the servers.
- So, if we have **n** servers, then we'll be having **k*n** server points inside the ring.
- So, here the load distribution will be possible evenly.
- **Now, the *pi* of 4 servers, from which we were taking *5% keys* of each of those 4 servers and assigning those *keys* to the 5^{th} server, is successful**.

➢ Load Balancing is extensively used in Distributed System.

MESSAGE QUEUE

- Pizza shop analogy:
  - Client-1 orders a pizza. All he wants is a confirmation that the order is placed.
  - Now, he got the confirmation that the order is placed.
  - Now, client-2 comes to order a pizza. He doesn't have to wait till pizza of client-1 is ready and his process is completed, rather client-2 also get a confirmation that his order is placed.
  - So, a list has to be maintained to keep track of the orders.
  - Whenever a client comes and place an order, the list order has to be written in the list according to the priority (the user might have an urgency) or complexity (the order might be just a coke).
  - It'll give flexibility to do something or use some other resources of the pizza shop rather than focusing on to the pizza only.
  - Also, the shop manager can also receive the orders of other customers.
  - When the pizza is ready, the pizza shop owner will ask the client to pay. In response of which the client pays the bill and take the pizza out.
  - After this, the process is completed, and this user's order will be removed from that list.
  - It all happened asynchronously.
- Lets take a complex scenario, you have 4 pizza shops. PS0, PS1, PS2, PS3
  - Let the shops got some orders. All are maintaining their lists of orders.
  - Now, if a shop, let PS3, goes down, the list will be gone and you'll not be able to process those orders.
  - Solution is using a database and keeping all those orders data in that database.
  - So, let some orders like
    - O1 > PS1
    - O2 > PS3
    - O3 > PS0
    - O4 > PS1
    - O5 > PS2
    - O6 > PS3
  - Now, PS3 goes down. But the orders data are stored in Database.
  - There will be a **Notifier** available in between the shops and the database.

- The Notifier will be checking in some particular interval of time (let each 15 seconds). As the PS3 is down, Notifier will not get any response from PS3 and will assume it as down.
- Now, the Notifier will query the database and check which orders are in N (Not done) state.
- Then it re-distribute all those orders to the Pizza Shops.
- But now, there might be chance of duplication here. For example
  - Let PS0 is preparing the order O3 already.
  - After the re-distribution, let PS1 got O3.
  - So, now the customer will receive 2 pizzas from PS0 and PS1.
  - So, it'll lead to too many duplication and complication.
  - So, here we can use Load Balancer with the Notifier.
  - One important feature of Load Balancer is to remove the duplication along with balancing the load (Consistence Hashing method).
  - So now, from that ring, all the instances of PS3 will be gone now. So, those orders which had to be processed by PS3 will be assigned to some other shop.
  - And, remaining shop's orders will not be removed. Just they'll get some extra orders which were assigned to the shop PS3.

- **Assignment/Notification + Load Balancing + Heart Beat + Persistence in one thing = Message/Task Queue**
- Working of Message Queue:
  - Producer sends a message to the message queue. (producer is not the user, it is the backend/server that puts the request to the message queue)
  - The *message queue* (RabbitMQ, Kafka, Amazon SQS ..etc) holds that message in memory/disc.
  - One or more consumers are waiting (or polling) for new messages.(consumer: background service or worker process. e.g., Node.js script who listen to the queue)
  - A consumer picks up the message, process it, and optionally sends an acknowledgement.
  - The queue removes the message after successful processing (or retires if failed).