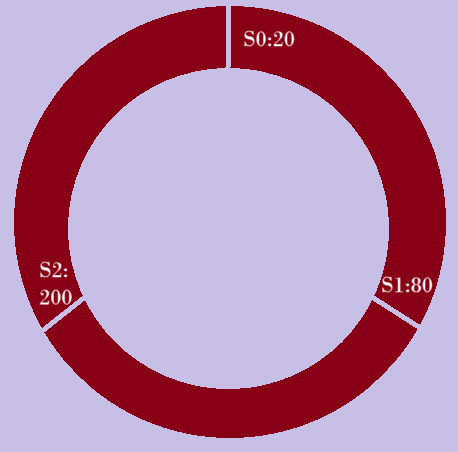
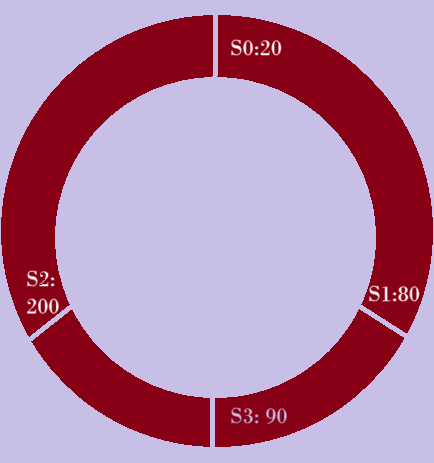
**SCALING: VERTICAL & HORIZONTAL**

* 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.