# The basics

**Scalability:**

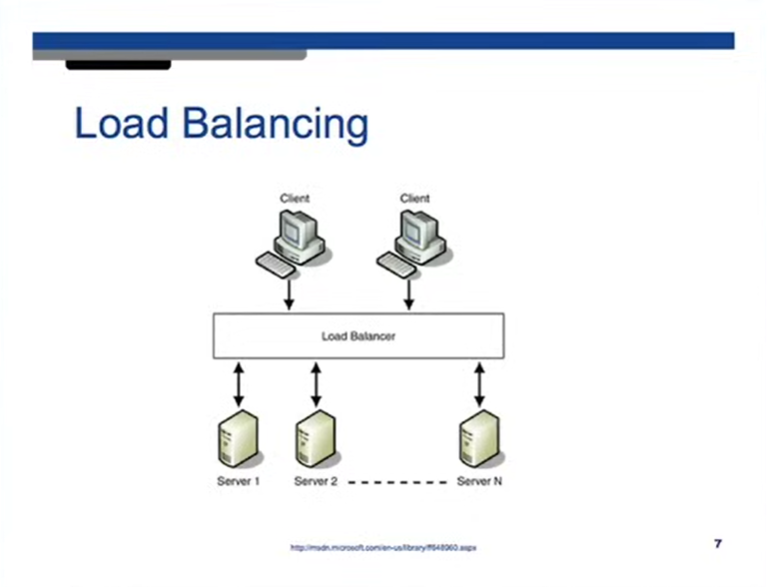
**A screen shot of a computer

Description automatically generated**

**Main Issues:** Hardware limit and single point of failure

**Horizontal Scaling:** Adding more machines.

**Main Issues:** Load balancing



In **vertical scaling**, when the client sends the request. **DNS** provides the IP address of the server to the client. In **horizontal scaling** we don’t have just one server, so, instead of the server, the **DNS** will provide the IP address of the **load balancer** which will then route the req to the server.

Another advantage of **horizontal scaling** is that we only need to make the IP address of the **load balancer** as public. The servers can have private IP addresses which is good for privacy and security.

We can also have a solution where the **DNS server** returns the IP address of one of the servers when user sends a request instead of sending the IP address of the load balancer.

How the **load balancer** distributes the load?

**Robin round scheduling:** The problem is if user sends a request to a webpage, he is served. If he sends the same request again, we should ideally route it to the same server because of cache. Another issue with this is that we lose sessions due to being routed to a different server. If you are logged in to a website, if we lose the session user will also lose its login and have to login again. So, using round robin is a bad idea.

We can store session details of the users in MySQL or other databases as well.

Or we can store the session details like which server you visited for a particular content in cookies. Cookie size is finite, so, you can store server id in the cookie and use that next time you want to visit the same site.

Load balancer can insert a cookie which has the server where we got the user data from in the browser of the user. This can be later used to go to the same server again and make use of the cache if possible.

**RAID: Redundant Array of Independent Disks:** RAID0, RAID1, RAID5, RAID10 etc.

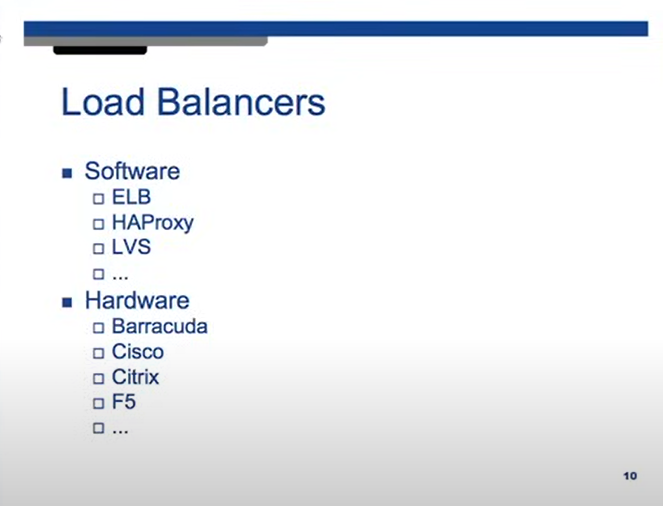
**RAID0:** If we have two hard disks, and the system wants to write a file, especially a large file, then the system can write some part of the file on disk1 and some part on disk2 and some part of disk1 and so on. This is because disk takes some time to write the data, in that time we write next data on the second machine and then back onto the first machine. This makes the speed of writing a lot faster. This is called **striping.**

**RAID1:** Whenever we write the file, we write it in both disks. There is performance overhead due to this. If any of the disk dies, data is not lost. This is **redundancy.**

**RAID10:** It is a combination of both RAID1 and RAID0. We use four drives. It has both **striping, striping.**

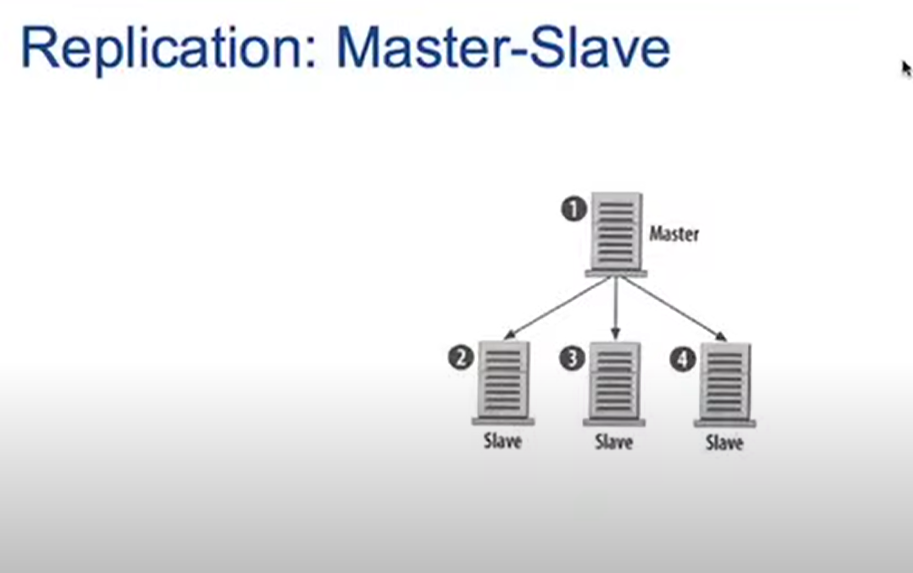
**RAID5, RAID6:** Variants of RAID1. If you have many disks, only one is used to keep redundant data.

All these techniques are used to reduce downtime.



**Cache: LRU (Least Recently Used):** Remove the data from the cache which was used in longest time. Cache will be used in the server.

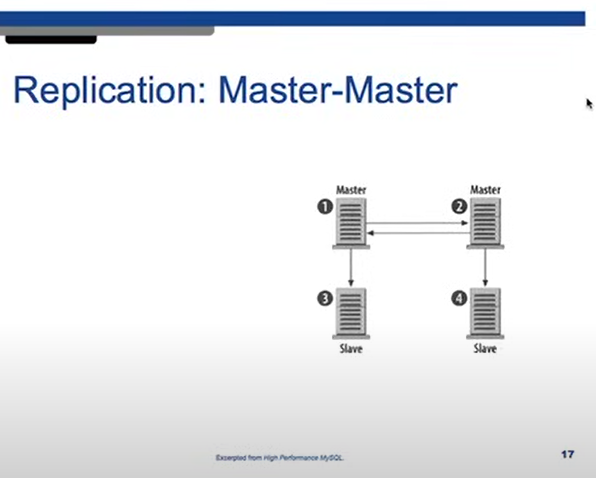
**Replication:**

****

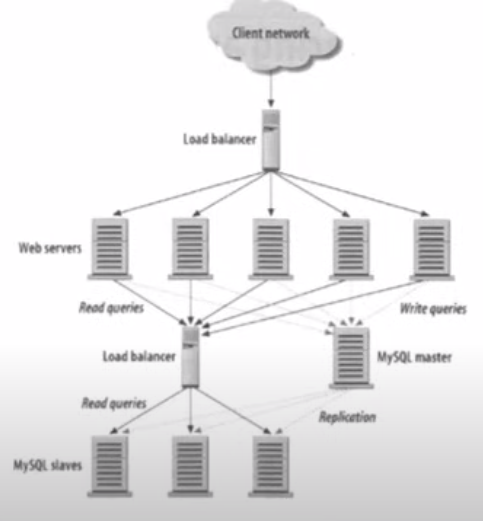
We read and write to **master database**. We copy this data to **slave databases**. Master and slaves are identical ideally. If one of the DB dies, we have backup. If master dies, we can make any of the slaves, master DB using some confuguration.

Above technique is much better for read heavy websites rather than write heavy websites. We can do reads from any of these servers. The reading becomes fast. If we want to write, we write to the master and then replicate that to other servers. This change has to propogate to the other DBs before we can do any reads. So, clearly, it is a better solution for read heavy systems.

The problem occures when our master dies. Since we can only write to the master. If the master dies, we can not do any reads until a slave is promoted to master. So, system becomes unusable until a master is chosen.



All servers are master. You can write to any and read from any.

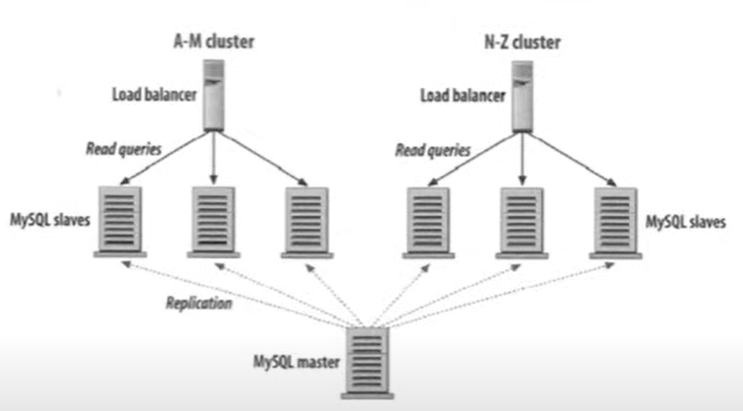


You can find the single point of failures by just looking at this picture and seeing any components which has a lot of incoming and outgoing and is bottlenecking.

Here, we see that load balancers are the bottleneck.

To avoid this, we usually also have a backup load balancer, the main load balancer sends a packet called **heartbeat,** to the backup load balancer. If the backup load balancer does not hear the heartbeat of the main load balancer, it is assumed to have died. This backup load balancer now becomes the main load balancer. Also, in load balancing terminology, we don’t call them master and slave, we call them **active and passive load balancers.**

How the passive balancer becomes active balancer. The backup one will just take the IP address of the active one and become active.



**A-M cluster** has the data for all the users with last name from a to m.

**N-Z cluster** has the data for all the users with last name from n to z.

This is called **partitioning.**

**High availability** is maintained using the heartbeat. Diagram of a server

Description automatically generated

Masters keep track of each other using the heartbeat.

Servers are usually in data centers ie some building in a geographical location. We also have replicas of the same servers in a different data center in some other grographical location. If one data center goes down, we need to send reuqest to the other data center. Moreover, requests that are closer to a certain geographic location must be routed to the closed server. This routing can be done through DNS. Different data centers with the same DB have different IP addresses. We can use DNS to give IP addresse of the closed data center to the user. This is called **global load balancing.**