**Distributed Data Stores**

A picture containing text, whiteboard

Description automatically generated

Consider a very simple scenario where you have a small application which has a single app server and a single instance of a database server. You have very less traffic and it used to work fine for you. Until, one day, a lot of people started using your application and everything becomes very slow. The app server becomes slow, the database server becomes slow. What should we do then? How do we scale the data base?

1. **Master-Slave:** We have multiple instances of the databases now. Any write that we do will be done on the master DB server. All the reads will be done from the slave DB servers. The data will be replicated from master to the slaves asynchronously. Since one may access data from slaves before this sync, the data in slaves for this delta time is “inconsistent”. So, we will not have the “C” from the CAP theorem.

The disadvantage of this is the fact that: For larger DBs, the sync delay between master-slave is there.

Diagram

Description automatically generated

1. **Sharding:** Sharding is a type of database partitioning that separates very large databases into smaller, faster, and more easily managed parts called data shards. We will partition the data into multiple databases based on a sharding key. For ex: We say data where usernames start from a-I will remain in one shard, j-s will remain in another shard and the remaining t-z will remain in another shard. All these databases will perform both the read and write operations.

A picture containing text, whiteboard

Description automatically generated

The disadvantage of this is:

1. One database may become a hotspot which gets more requests than other few shards of the database. We can solve this by creating more shards of the hotspot so that the load is distributed appropriately across shards.
2. SQL Joins & Aggregation queries like MAX, MIN, AVG, SUM is an issue.

**CAP Theorem: Consistency - Availability – Partitioning**

**1. Consistency:** Database is consistent if at all point of time, it returns the correct value. In the master slave strategy, there is a moment when the master should sync with the slaves and for that time, the data in slaves is inconsistent. So, if a DB adopt this, then the data is inconsistent and cannot be trusted all the time.

**2. Availability:** Even if some machine in my cluster goes down, the system should be available up and running.

**3. Partitioning:** If my connection between two machines goes down in a cluster, we should still be able to read and write data. However, the read-write may not be correct (if Availability is chosen over consistency) but still the system should respond if P is chosen. Even in case of internal network failure, the guarantees (either C or A) promised by system will continue to hold

One can't have all these three properties in the database. You can either have any of these two combinations. The database give us the configuration in our hands so that we can choose either of the two from these.

Traditional Relational databases like MySQL choose Consistency over Availability.

NoSQL systems believe in A and P by sacrificing C. But remember that we won't have strict consistency, but we will be eventual consistency here. Most of the applications can use this where the exact data is not important at all point of time. We have BASE property here meaning Basically Available and Soft State Eventual Consistency.

IRCTC, Banking systems, etc. where the data must be correct at all point of time should use 'RDBMS' which ensures consistency over everything.

**Consistent Hashing:** Read more from Alex Xu book.

1. Problem: If we have n cache servers, hashing is used as a common method to distribute the load across multiple servers. However, the issue comes when a server is added or removed, the number of keys which are to be rehashed to other servers is almost the total number of keys distributed across all servers.
2. Consistent Hashing solves this problem and minimizes the keys which has to be redistributed when servers are added or removed.

**Gossip Protocol:** Read more from Alex Xu book (Design a Key-Value data store)

1. In a Consistent Ring, all the servers/nodes are gossiping with each other to know what their ranges of the keys are. This talk between all of them is called Gossiping.
2. When a new server is added, they join the gossiping as well and then the relevant data is transferred to the newly added servers automatically.
3. When a server goes down or is removed, then each of them would try to talk with it and would get no response. Hence, they would know that it is down and then it will redistribute the keys in the ring to a new server.