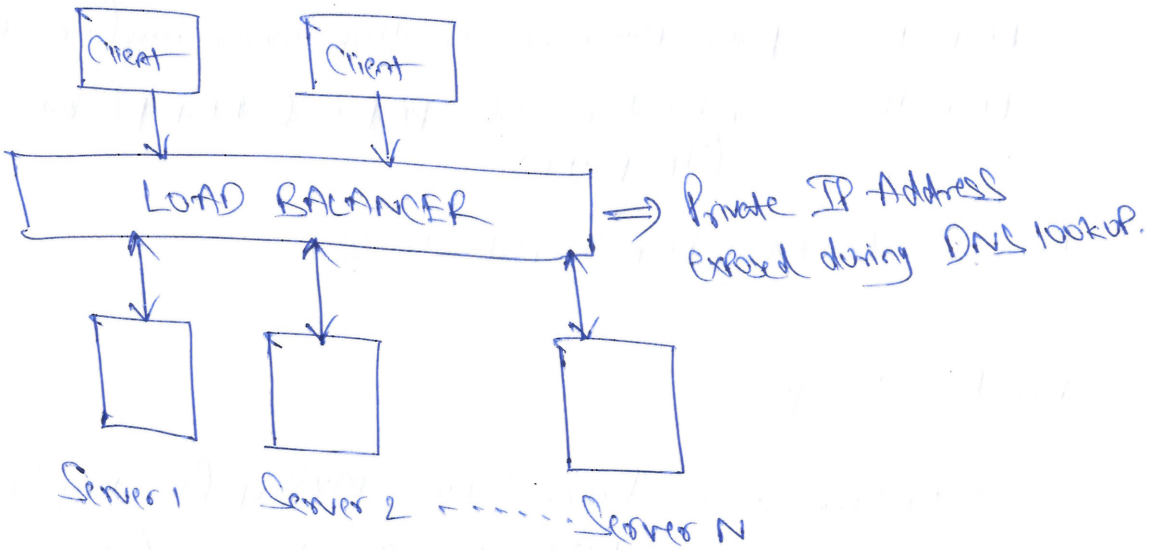
Scalability

* Horizontal Scaling - Using a bunch of Servers with a Load Balancer.



* Load Balancing using a Round Robin technique (Load Balancing with BIND)
* But then the above technique might not be so optimal as one server might always get computationally difficult tasks than other server.

* DNS lookup could be cached. Along with a TTL (Time to live) kind of a configuration.

⇒ Shared Sessions State :

* If session info stored at server level then same user might have to login multiple times (i.e session state stored as a serialized file on Disk).

* If shared session info stored on LB, then again its prone to issues like if LB server is down then whole login/session info is lost. [weakness in the network Topology] i.e Robustness is lost

⇒ RAID (Redundant Array Of Independent Disks)

* Instead of Storing Session info on Hard Disks Use a RAID.
  (Ex: RAID 0, RAID 5, RAID 10 ...)

  RAID 0 – Double Write Speed by Data Striping.

  RAID 1 – Data Mirroring i.e High availability (i.e Redundancy)

  RAID 10 – Combination of RAID 0 & RAID 1 (Data Striping + Redundancy)
  (4 Disks)

Above Techniques help in Robustness.

⇒ Shared Storage

Sticky Storage – Fiber Cable, MySQL (Writing Session Objects to DB)
NFS (N/w File System) } Both are expensive

* Still its a Single Point of Failure

Load Balancers : S/w – HA Proxy
H/w – Citrix, F5 etc

* We Could Use Cookies to have Sticky Session (i.e to end up on the Same Back end Server on every login)
  - Cons – Finite Size
  - Privacy
  - Could be disabled

⇒ Caching

• html → Caching the Page itself as Something.html So that Page regeneration not needed.
  - Redundant Tags
  - Space issues.
  - Difficult to do modifications, like Page Color & all.

→ MySQL Query Cache

→ MemCached – Software running on a Servers. For key value Store Caching.

Replication: Master – Slave via N/W Connection

- Copy every row in master to one or many Slaves
- No Single Point of failure / less downtime.
- Balance Read heavy request.

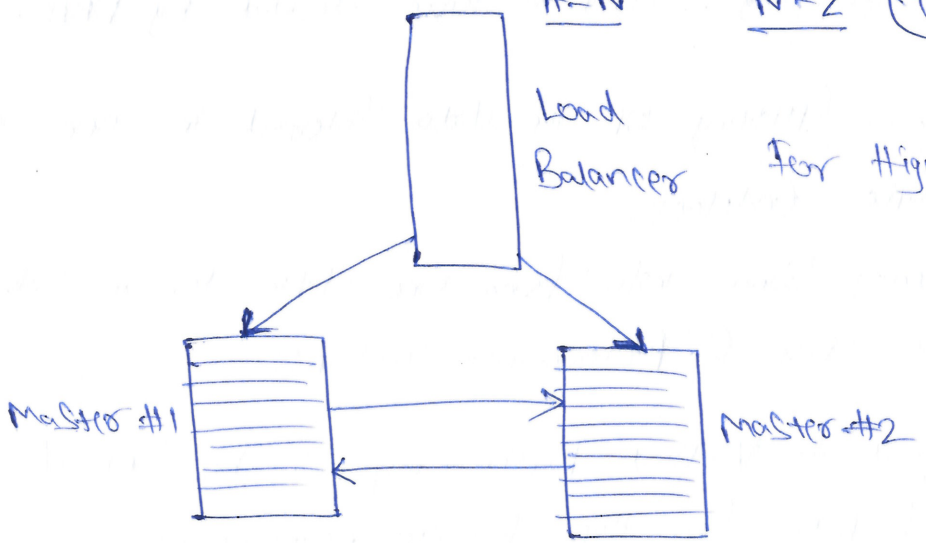
* Writes can have Some downtime
  - Have a Master – Master Replication to Solve this.

* Multi tiered Architecture to have a load Balancer for HTTP Requests and Data Replication at DB level.

⇒ load Balancer → Having a Active – Active or Active – Passive Setup to avoid Single Point of failure. i.e High Availability

Partitioning → To Have different Clusters.

A – N N – 2 (Two Clusters)



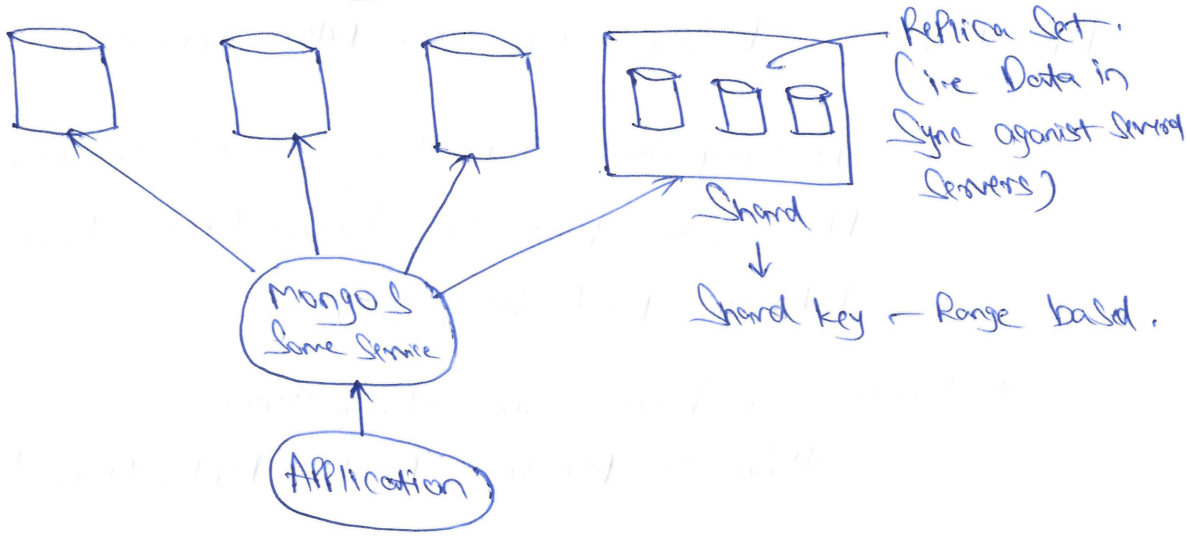
For High Availability.

⇒ DNS lookup @ LB level – For Data Centers Spread over Geography. Location Specific Load balancing.

i.e IP Adress Specific to Some building i.e a Data Center near them.

SHARDING (AKA Horizontal Partitioning)

→ Splitting a large collection across multiple servers. For the purpose of performance, availability & maintainence.



→ Shard database → Create replicas of the schema, and then divide what data is stored in each shard based on shard key.

Example: Shard a customer DB using customer Id as shard key i.e 0 – 1000 in one shard
1001 – 2000 in other shard etc

Choosing shard key – complex logic decided by DBA.

★ Vertical Partitioning. – Splitting up the data stored in one entity into multiple entities.

Example – moving some info from one table to an other for space & performance reasons.

Ps: Horizontal & Vertical Partitioning can be combined like in BIG Data for high traffic management.

* Vertical Scaling

More

* CPU — Cores, L2 caches
* RAM
* DISK — PATA, SATA, RAID (Hard Drives), SSD — No moving Parts.

⇒ limitations to upgradations.

Compare and Swap. ⟹

* Lock Free Algorithm → Based on non-blocking algorithms which Promises Global Progress instead of indiv Stall

* A Technique used when Designing Concurrent Algorithms. Basically CAS Compares an expected value to the Concrete value of a Variable, and if the Concrete value of the variable is equals to the expected value, Swaps the value of the variable for a new variable.

```
Class MyLock {
    Private boolean locked = false;
    Public boolean lock () {
        if (! locked) {
            locked = true;
            return true;
        }
        return false;
    }
}
```

CAS as Atomic Operation :

```
Public Static Class MyLock {
    Private AtomicBoolean locked = new AtomicBoolean(false);
    Public boolean lock () {
        return locked . CompareAndSet (false, true);
    }
}
```

* Here it compares the value of locked to false and if it is false it sets the new value of AtomicBoolean to true.

The compareAndSet () method returns true if the value is swapped, and false if not.

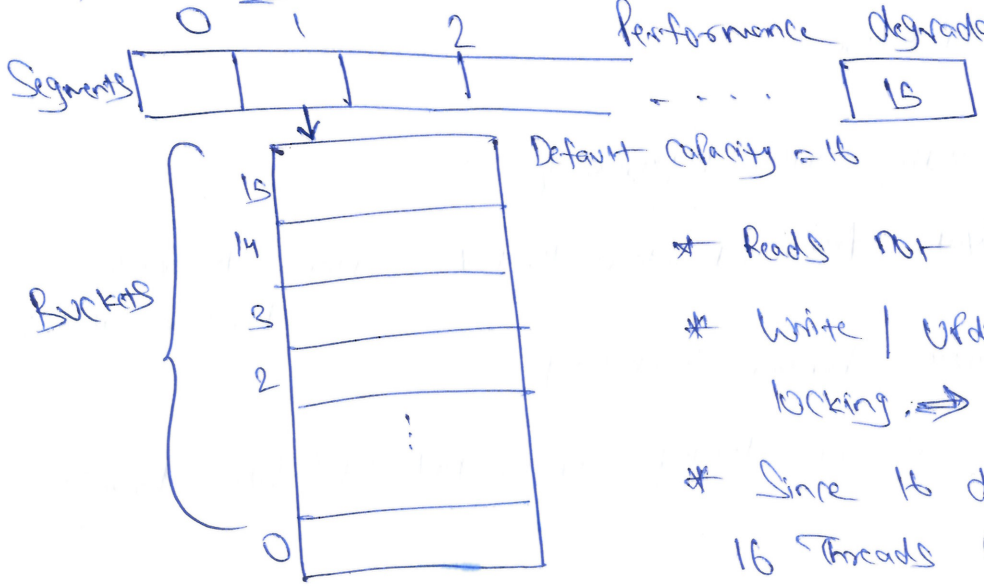
* CAS is generally much faster than locking, but it does depend on the degree of contention. Because CAS may force a retry if the value changes between reading and comparing, a thread can get stuck in a busy-wait if the variable in question is hit hard by many other threads.

Synchronized Collections vs Concurrent Collections.

* Collections.synchronizedMap() and Collections.synchronizedList() provides a basic conditionally thread-safe implementation of Map and List. (Fail-fast Iterator)

* Single collection-wide lock is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to prevent ConcurrentModificationException.

⇒ ConcurrentHashMap → Introduced a concept of segmentation so performance degradation wont happen.



* Reads not having lock
* Write / Update having segment level locking ⇒ lock striping.
* Since 16 default capacity by default 16 threads can write / update simultaneously.
* Concurrency-level is 16.

* HashTable all methods including get is synchronized.
* CHM - Put & remove use lock striping → final value always cached.
  - Get : value is volatile, next is final - Hence no stale reads.

	Re ordering	Caching	happens-before	Mutual-exclusion
Sync	✓	✓	✓	✓ (Pessimistic lock)
Volatile	✓	✓	✓	✗
Atomic (CAS)	✓	✓	✓	✓ (Optimistic lock)

* Optimistic lock → Read is allowed across thread. Write is not allowed.
* Pessimistic lock → no read, write till lock is released.

Issues with Synchronization:

1. One condition per lock
2. No lock polling. i.e Thread which does not get a lock can't do anything else. [tryLock() method]
3. No Timed lock waits. [tryLock(longtime, TimeUnit unit)]
4. No Interrupted lock waits.
5. Block Structured lock (JVM is doing an instrinsic lock. Ex: Calling an inner method lock)

Wait() / Notify():

1. In Java, the Object is the entity that is shared between threads which allows them to communicate with each other. The threads have no knowledge of each other and they run Asynchronously.
2. In Java, you wait() on a particular instance of an Object called the monitor. notify() will notify other threads waiting on same Object.

* If wait() and notify() was on a Thread, each thread would have to know the status of every other thread. How would thread1 know that thread 2 was waiting for access to particular resource?

* JVM has a hidden wait queue() which puts all threads in a queue waiting for same object and then notify() will dequeue & put to running state.

* notify() → Printer example

  notifyAll() → Application Cache Cleaning → Objects wake up all after this.

  Ex: Parallel Garbage Collector.

⇒ <u>Transfer Queue</u>

* Extends Blocking Queue.
* transfer() method is like a synchronous put() method.
* tryTransfer() – returns boolean

  ie weather consumer able to consume now or not?

Linked Transfer Queue : implements Transfer Queue.

⇒ Object & Thread itself in a Queue, hence called Dual Queue or Dual blocking Queue

<u>transfer</u>	<u>put</u>	<u>tryTransfer</u>
Blocking	Normal	Depends on consumer

* hasWaitingConsumerCount() → return blocking count
* hasWaitingConsumer() → true / false.

* Fastest blocking Queue:
  → linkedBlockingQueue (JDK 5 & 6)
  → linkedTransferQueue (JDK 7)