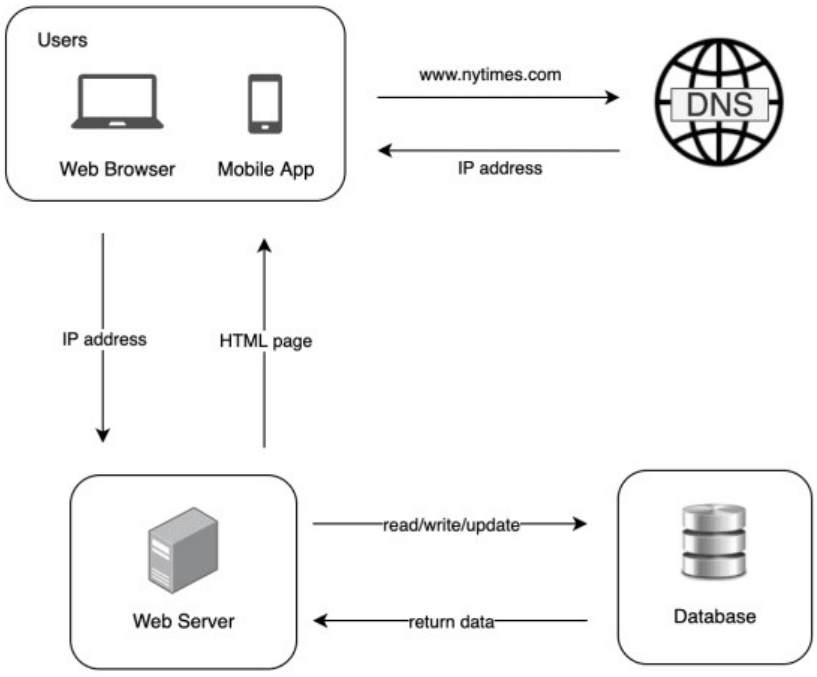
...Scale from Zero to Millions of Users

#system #design #scale #web

Lets start off with the simplest possible website setup: 1 web server and 1 database server. (yeah this could have been simpler by running both web service AND database on the same server).



Your choice for database type will probably be a traditional **relational** database like MySQL, PostgreSQL, or Oracle. These are popular and handle most use cases. These represent data in tables and rows, and supports join operations across multiple tables.

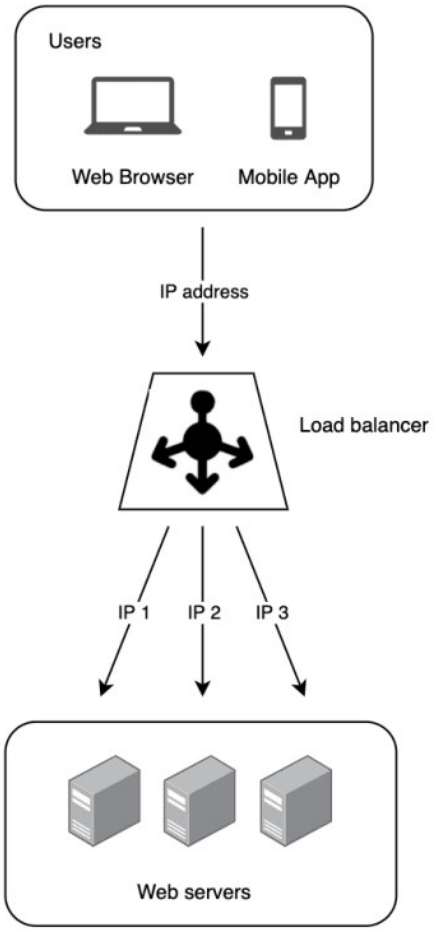
Non-relational databases might be the right choice for you if have the following requirements:

- Very low latency

- Unstructured data
- Serialization and de-serialization of data like json or yaml.
- Massive amounts of data

Scaling Web Layer: Part 1

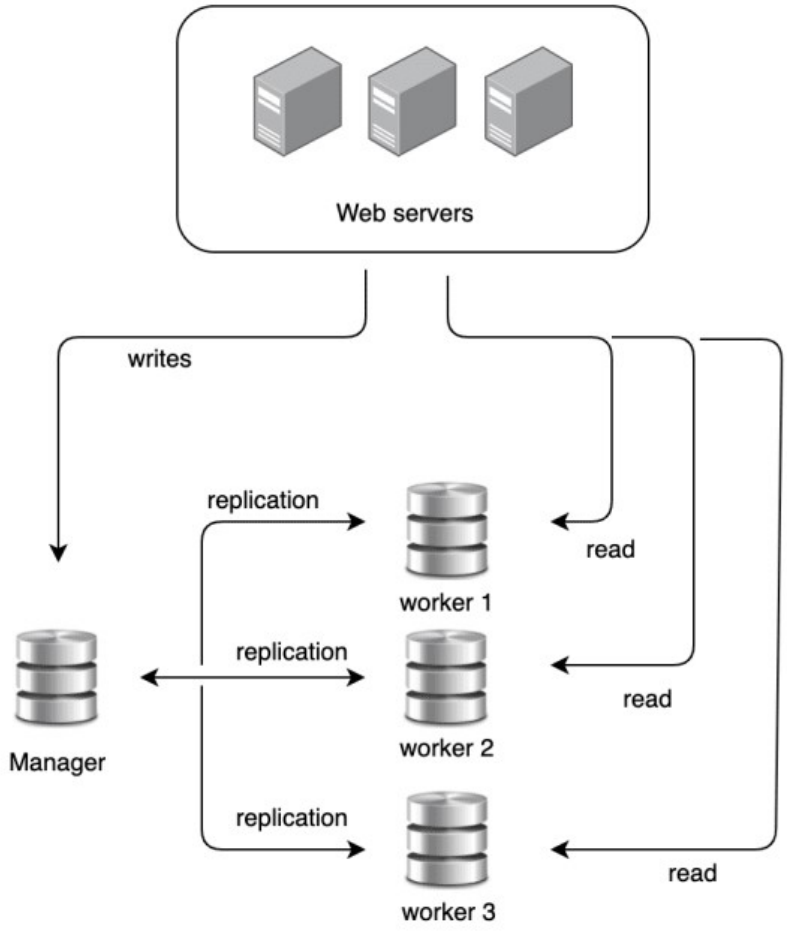
Vertical scaling is the process of adding more power like CPU and RAM. Horizontal scaling is the process of adding more servers. Horizontal scaling is more desirable for large scale applications. And to accomplish this, a **load balancer** is required.



Scaling Database Layer: Part 1

We can use the same principle of horizontal scaling for the database layer. Except

not all database servers will have the same responsibility. There will be 'manager' and 'worker' types. The manager server only supports write operations. The worker server gets copies of the data from the manager server and only supports read operations. All operations that make changes (INSERT, DELETE, etc) will be sent to the manager. All read-only operations will be sent to the workers.

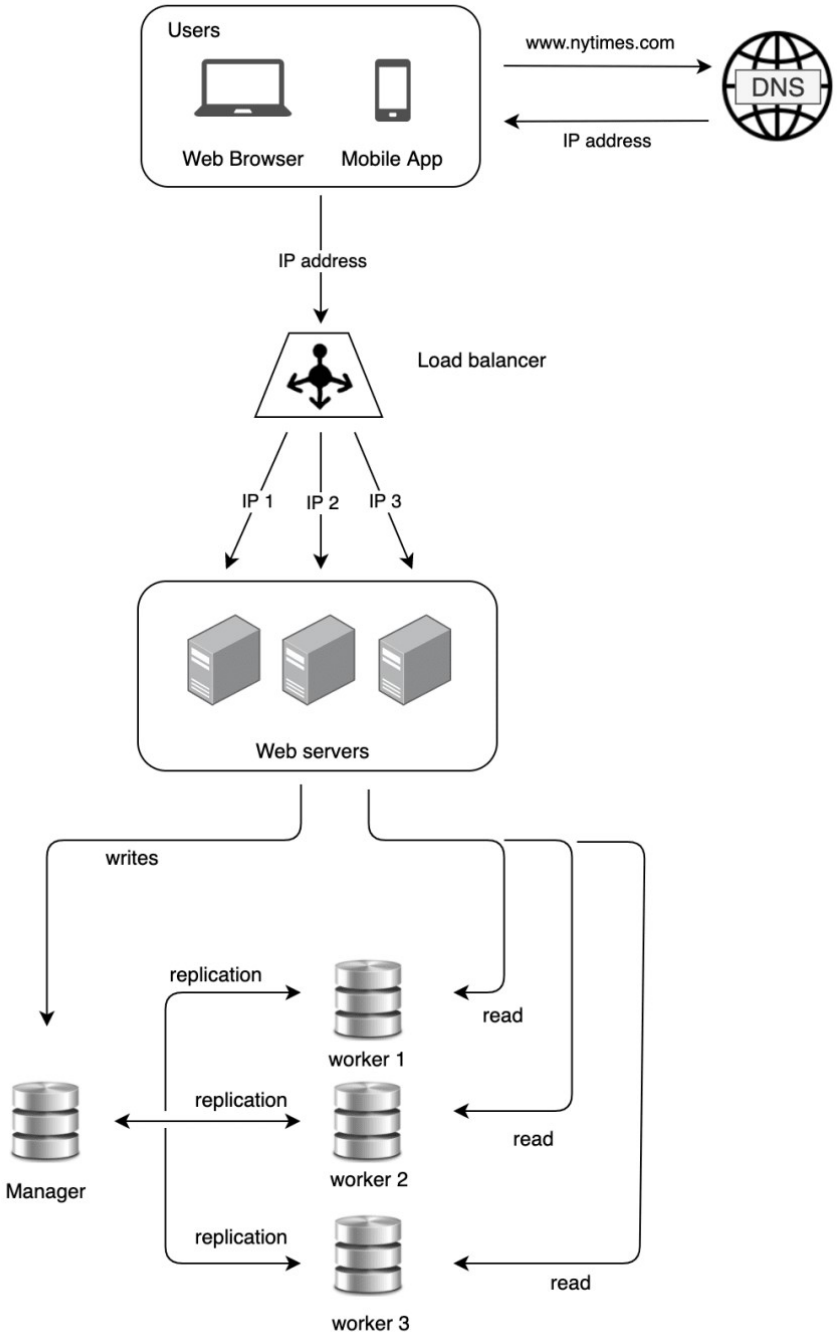


With database replication you gain the following advantages:

- Better Performance: Most applications require a much higher ratio of reads to writes. So it is best to distribute reads among several servers.
- Reliability & High Availability: If one of your databases crashes, you do not need to worry about data loss because data is replicated.

Putting It All Together (so far)

Ok, after successfully scaling both web and database layers, lets take a look at what we got so far.



Cache

A cache is a temporary storage space that stores database results of frequently accessed data in memory. This way subsequent database requests will fetch the cached results, instead of making a round trip from client to database and back. This caching strategy is called a read-through cache. Below are some considerations when using a cache system.

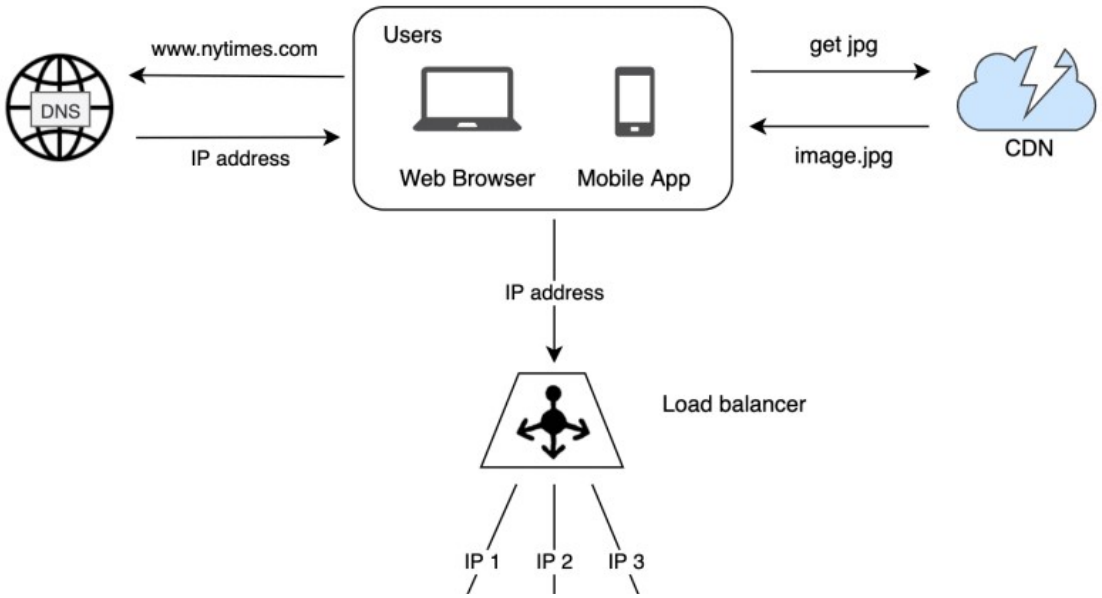
- Consider using a cache when data is read frequently but modified infrequently.
- It's good practice to implement an expiration policy. Once cached data is expired, it is removed from cache.
- Inconsistency between the cache and database can happen because data-modifying operations are not a single transaction.
- Mitigating failures by using multiple cache servers and the over-provision of memory.

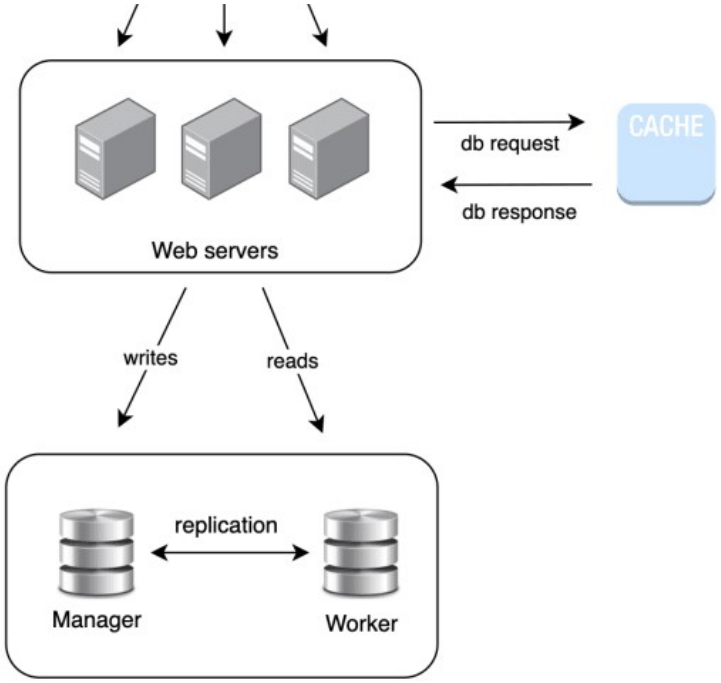
Content Delivery Network (CDN)

A CDN is a specific type of cache where static content like images and videos are stored on a network of geographically dispersed servers. A client will make a request for something like an image from the CDN server first, and will only make a request to the web server if the content isn't available on the CDN. Below are some considerations when using a CDN.

- Cost: CDNs are run by third party providers who charge you for data transfers in and out. Infrequently accessed assets should not be stored here.
- Cache Expiration: If the expiration policy is too long, you run the risk of the delivering stale content. If the expiration policy is too short, you run the risk of repeated requests to the origin server.
- CDN failures: Your application should be able to gracefully recover from outages and re-route requests to the origin web server.

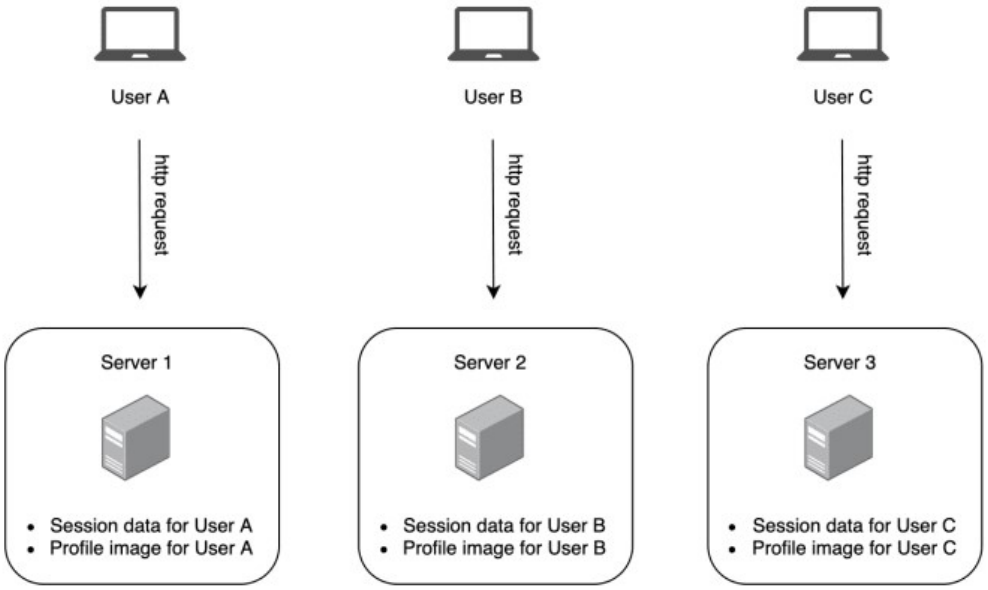
The architecture of our website is getting complex. Lets take another look at what our design looks like with a database cache and CDN added.





Scaling Web Layer Part 2: Stateless Web Servers

In order to scale our web layer further we need to allow a client to make requests to any web server at any time. Currently if user A logs into their social media by making requests to server 1, user A will need to continue to make requests to server 1 if they want to remain logged in. See below for an illustration of a "sticky session" scenario.

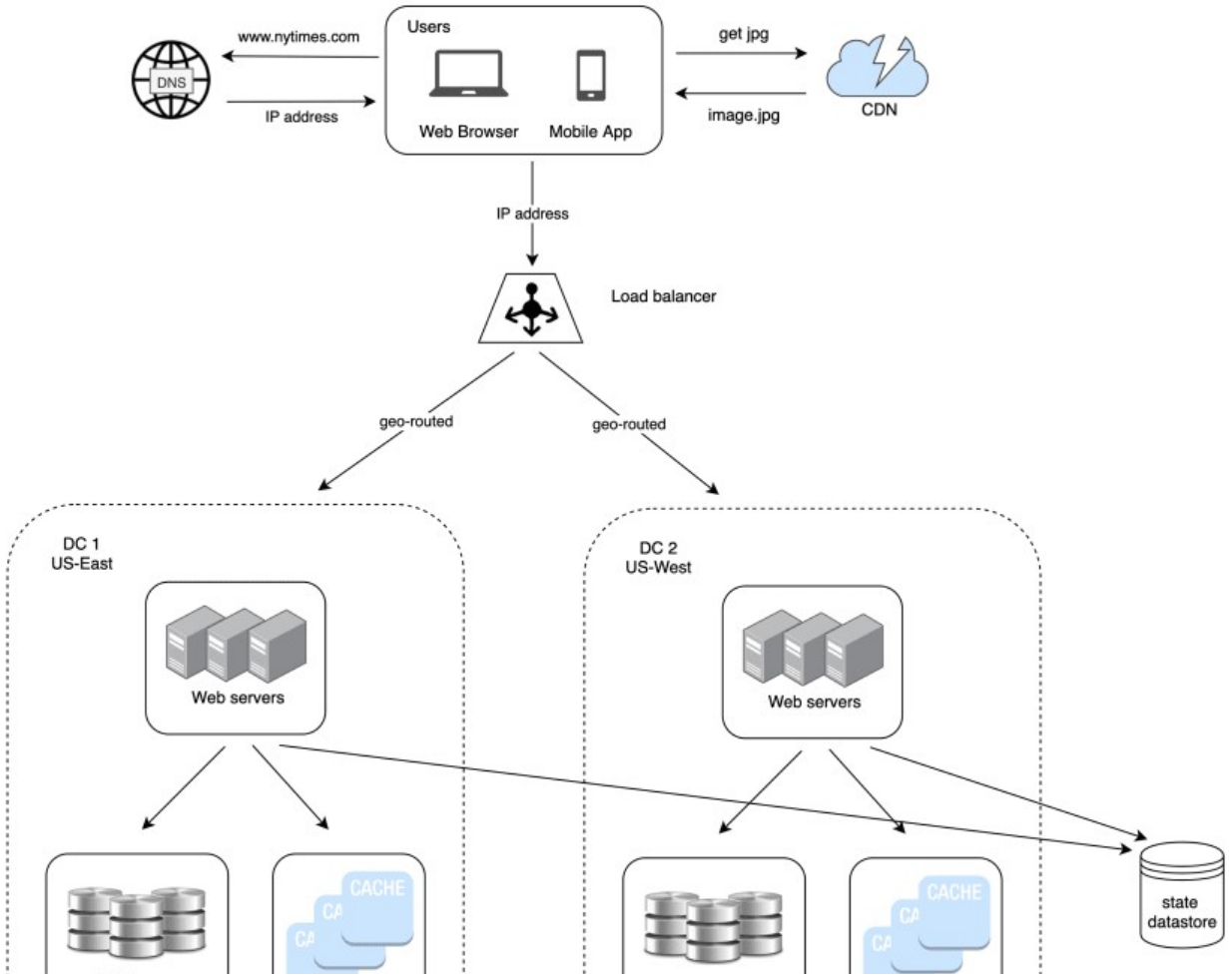


The opposite of the above scenario is a stateless architecture. Where a user's session data is not stored on the web server itself, but a separate persistent storage location like a relational database or s3.

Data Centers

If our website grows at a fast rate and attracts international users, setting up a clone of our core architecture (web + database) in other parts of the world is common practice. A user is then geo-routed to the closest data center. GeoDNS is a DNS service that allows domain names to be resolved to IP addresses based on the geographical location of a user. The relational database that helps us achieve a stateless web layer is not duplicated.

Lets take a look at our architecture now with stateless web servers and multiple data centers.



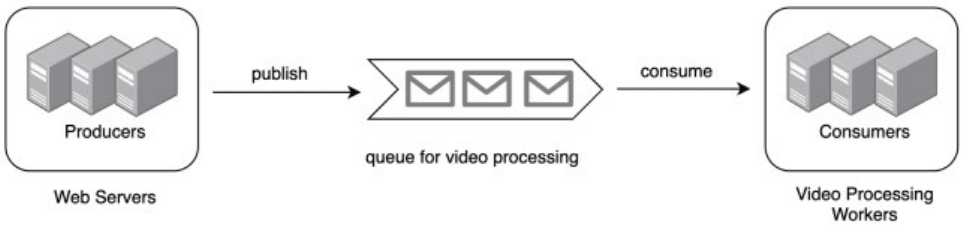
In the above scenario, in the event of a significant data center outage, we can direct all traffic to a healthy datacenter. There are some challenges that we need to address to achieve this multi-datacenter architecture.

- Traffic redirection: GeoDNS can be used to direct traffic to the nearest data center depending on where a user is located.
- Data synchronization: Users from two different regions could use two different databases or caches. In the event of a failover, traffic must be re-routed to a datacenter where data is unavailable. A strategy to replicate data across datacenters is required.
- Test and deployment: The use of automated deployment tools to quickly and consistently deploy across datacenters is crucial.

Message Queues

A message queue is a software component that decouples two steps of workflow into 'producers' and 'consumers'. Decoupling of components is a preferred architecture for building scalable and reliable applications. The message queue will serve as a buffer and distribute asynchronous requests.

Consider the use case of an application that supports video uploads. Video publishing is not instantaneous. The application will need to preform several steps like cropping, blurring, audio enhancements, and format conversions before it is available to the public. All of these tasks take time to complete. One group of servers act as the producers of video processing tasks, and another group of servers as the consumers of these tasks and will complete the actual work. See illustration below.



Logging, Metrics, and Automation

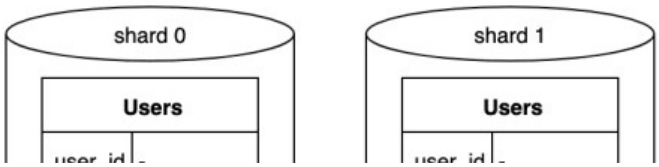
As a website's popularity grows adding the necessary infrastructure to support logging, metrics, and automation is essential.

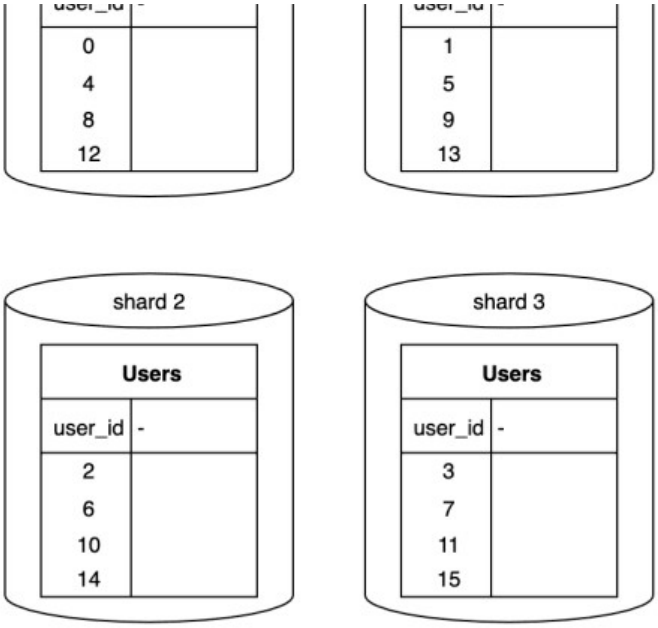
- Logging: Allows the dev team to investigate and debug errors.
- Metrics: Collecting and analyzing metrics that are relevant to the business helps the team gain insights. Metrics such as:
    - Host level metrics: CPU, Memory, disk I/O, etc.
    - Aggregated level metrics: performance of all layers of the application: Web, Database, Cache.
    - Key business metrics: daily active users(DAU), retention, revenue, etc.
- Automation: With a system as complex as this, we need to build or leverage automation tools to improve productivity.

Scaling Database Layer Part 2: Sharding

When the amount of data in application no longer fits on one server alone, a common approach is to distribute this data among several servers. Previously we used more than one database server, but one relational table still fit on one server at a time. In this next iteration, we'll essentially split up one table over several servers. Sharding separates large databases into smaller, more easily managed parts called shards. Each shard shares the same schema, though the actual data on each shard is unique to the shard. A hash function is used when accessing data to find the corresponding shard. For this next illustation, we'll use this simple hash function: user_id % 4.

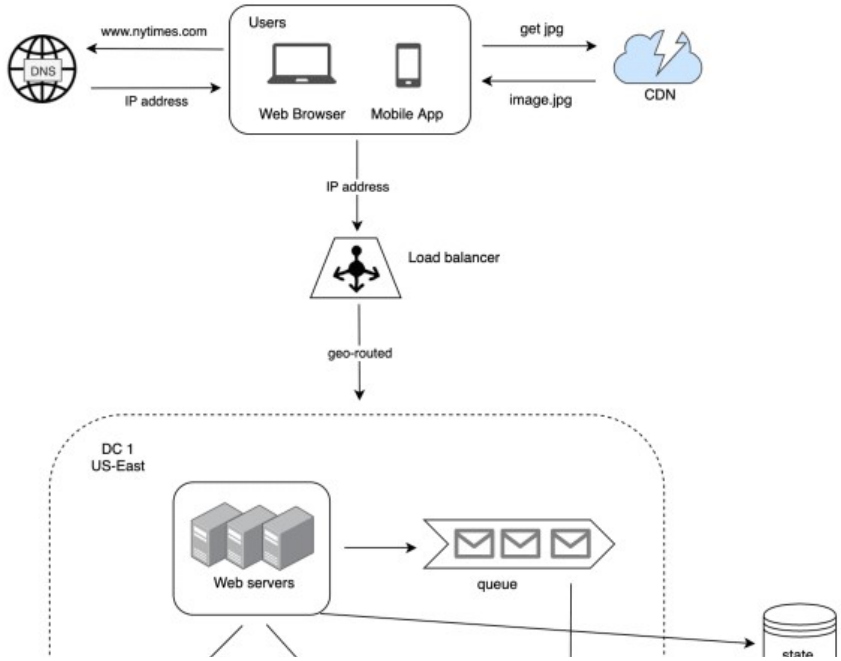
The most important factor to consider when implementing a sharding strategy is the choice of the sharding/parition key. The partition key is one or more table columns that make a unique key that determine how data is distributed. In the illustration below, "user_id" is the partition key.

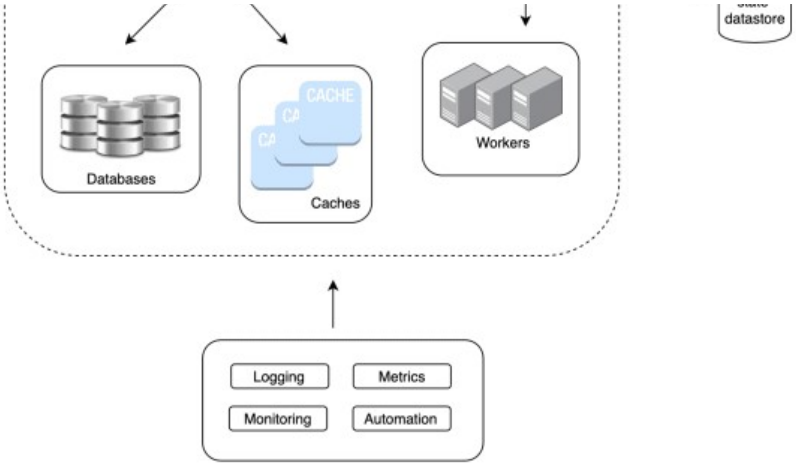




Sharding is a common approach to scaling a database layer. It does come with its own set of challenges:

- Resharding Data: redistributing data on the cluster is needed when a single shard is full.
- Celebrity problem: is when a single shard experiences excessive queries compared to the rest of the cluster.
- Join Operations: running a join operation on tables that are spread among several servers is not easy. De-normalizing so that these queries can be run on a single table is a possible solution.





Finally the illustration above shows what the architecture looks like with everything put together. Some parts are condensed.