System Design

Scale from one to million users

Course Structure & Grading

- Topics to be covered:
  - System Design
  - Design Patterns
- Grading:
  - 15%+10%+5% - Presentation + Activity (quality of questions is considered) + Questions from other students (at least two)
  - 30% - Project (demo of scaling techniques).
  - 40% - Final exam (written/code + oral exam).
- Literature:
  - Alex Hu - System Design Interview
  - Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides - Design Patterns: Elements of Reusable Object-Oriented Software

Project outline

- Simple web app in Node.js, Python (Flask/Django), or even static website on Nginx.
- Load Balancer – Nginx or HAProxy
- Cache – Memcached or Redis for caching data and API responses.
- Database sharding - Vitess, ProxySQL or MySQL Router.
- Message queuing - RabbitMQ.
- Run the application on at least two instances (it can be a Docker container on different ports).

Introduction

- System design is an incremental process.
- It requires continuous refinement and endless improvement.
- It is an open-ended problem.
- There are many differences and variations in the system.
- The desired outcome is to come up with an architecture to achieve system design goals.

Single server setup

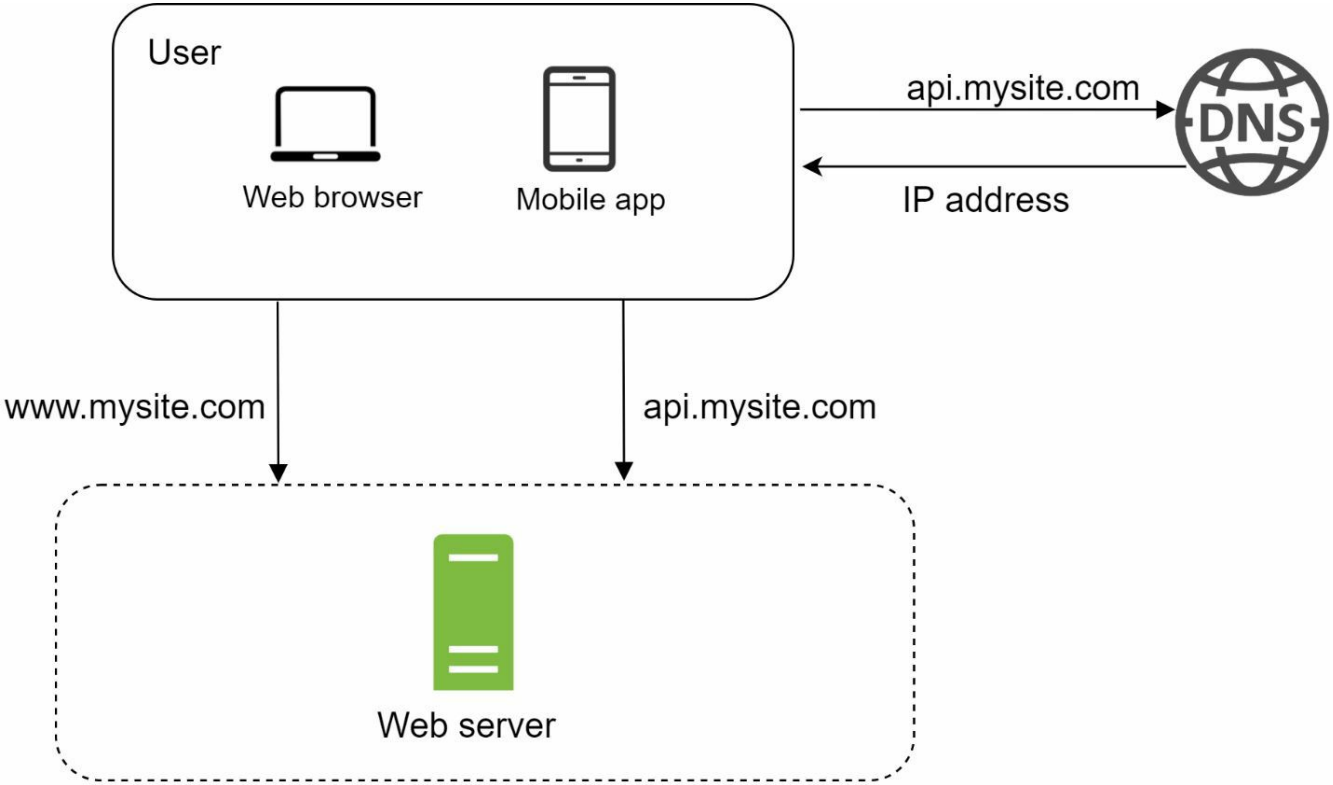

Figure 1-1

Request flow

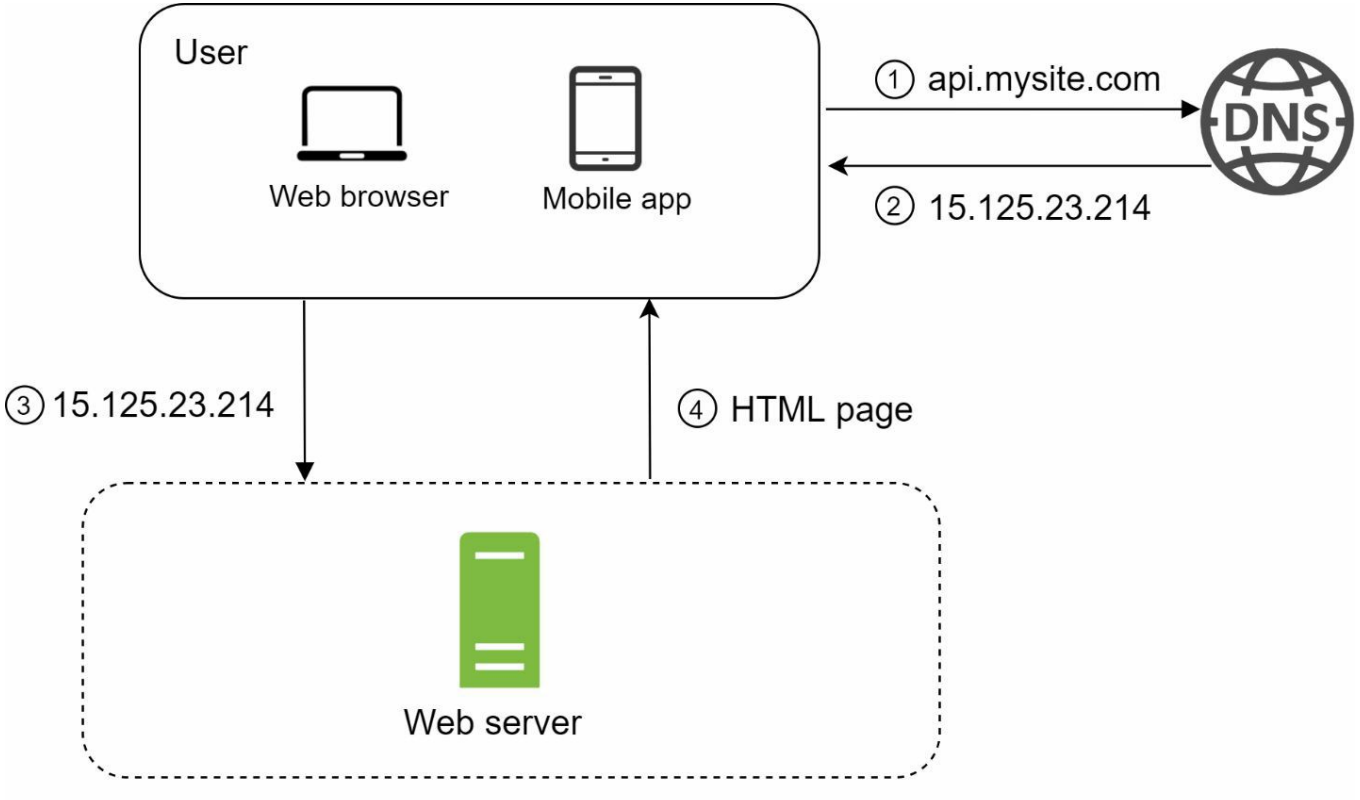

Figure 1-2

Request flow

1. Users access websites through domain names.
2. IP address is returned.
3. Once the IP address is obtained, Hypertext Transfer Protocol (HTTP) requests are sent directly to your web server.
4. The web server returns HTML pages or JSON response for rendering.

JavaScript Object Notation (JSON)

- Commonly used format for APIs and data exchange.
- Lightweight, human-readable, and easy to parse.
- Supports key-value pairs, arrays, and nested structures.
- Widely used in web development, mobile apps, and databases.


```
{
    "id": 12,
    "firstName": "John",
    "lastName": "Smith",
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        "212 555-1234",
        "646 555-4567"
    ]
}
```

Database

- With the growth of the user base, one server is not enough.
- We need multiple servers:
  - At least one for web/mobile traffic.
  - At least one for database.
- Separating web/mobile traffic (web tier) and database (data tier) servers allows them to be scaled independently.

Separate web and database tier

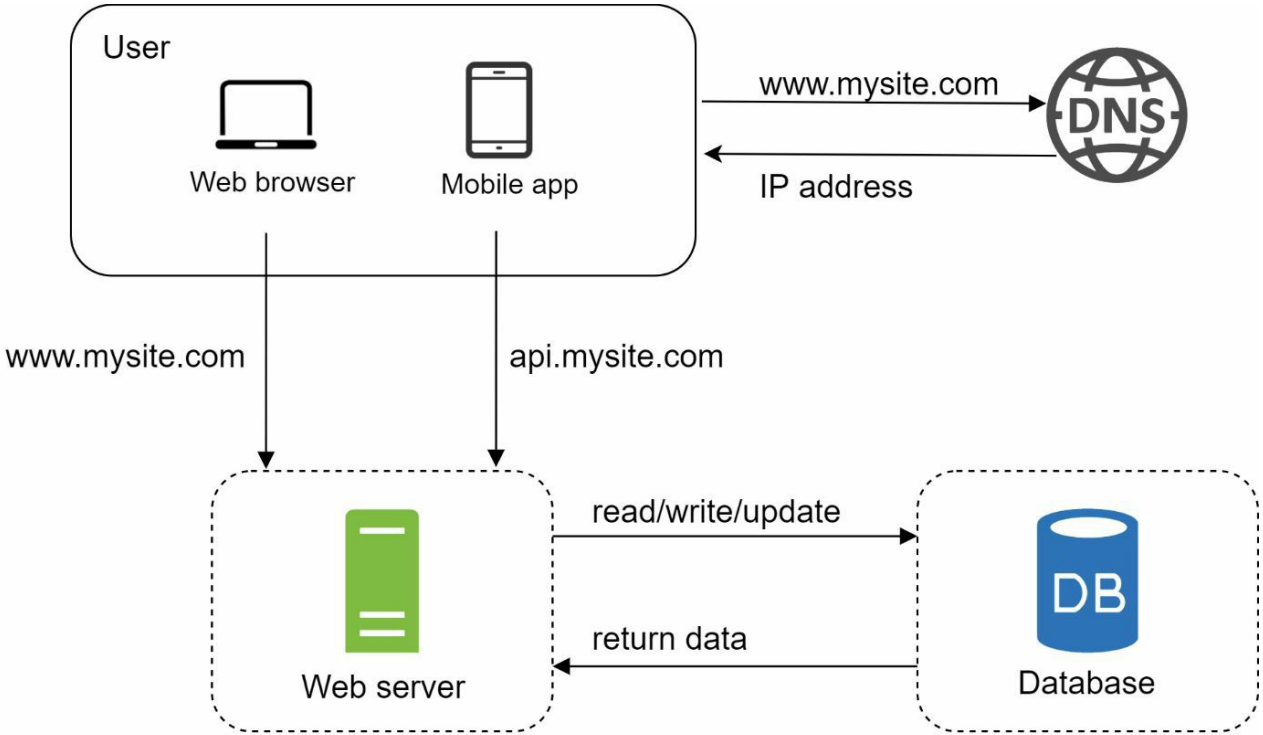

Figure 1-3

Types of databases

- There are generally two types of databases:
  - Traditional relational database i.e. SQL databases (e.g. MySQL, Oracle, PostgreSQL, etc.)
  - Non-relational database i.e. NoSQL databases (Cassandra, MongoDB, CouchDB, HBase, etc.)
- Relational databases represent and store data in tables and rows.
- You can perform join operations using SQL across different database tables.
- NoSQL databases are grouped into 4 categories:
  - Key-value stores (Redis)
  - Graph stores (Neo4j)
  - Column stores (Cassandra, HBase)
  - Document stores (MongoDB, CouchDB)
- Join operations are generally not supported in non-relational databases.

Which database to use?

- For most developers, relational databases are the best option.
- Non-relational databases might be the right choice if:
  - Your application requires super-low latency.
  - Your data are unstructured, or you do not have any relational data.
  - You only need to serialize and deserialize data (JSON, XML, YAML, etc.).
  - You need to store a massive amount of data.

Vertical vs horizontal scaling

- Vertical scaling (scale up) is the process of adding more power (CPU, RAM, etc.) to your servers.
- Horizontal scaling (scale out) allows you to scale by adding more servers into your pool of resources.
- Advantages and disadvantages?
- Simplicity of vertical scaling is its main advantage.
- Vertical scaling has a hard limit. It is impossible to add unlimited CPU and memory to a single server.
- Vertical scaling does not have failover and redundancy.
- Horizontal scaling is more desirable for large scale applications.

Load balancer

- In the previous design, users are connected to the web server directly.
- Users will be unable to access the website if the web server is offline.
- In another scenario, if many users access the web server simultaneously and it reaches the web server’s load limit, users generally experience slower response or fail to connect to the server.
- A **load balancer** evenly distributes incoming traffic among web servers that are defined in a load-balanced set.
- Web servers are unreachable directly by clients anymore.

System design with load balancer

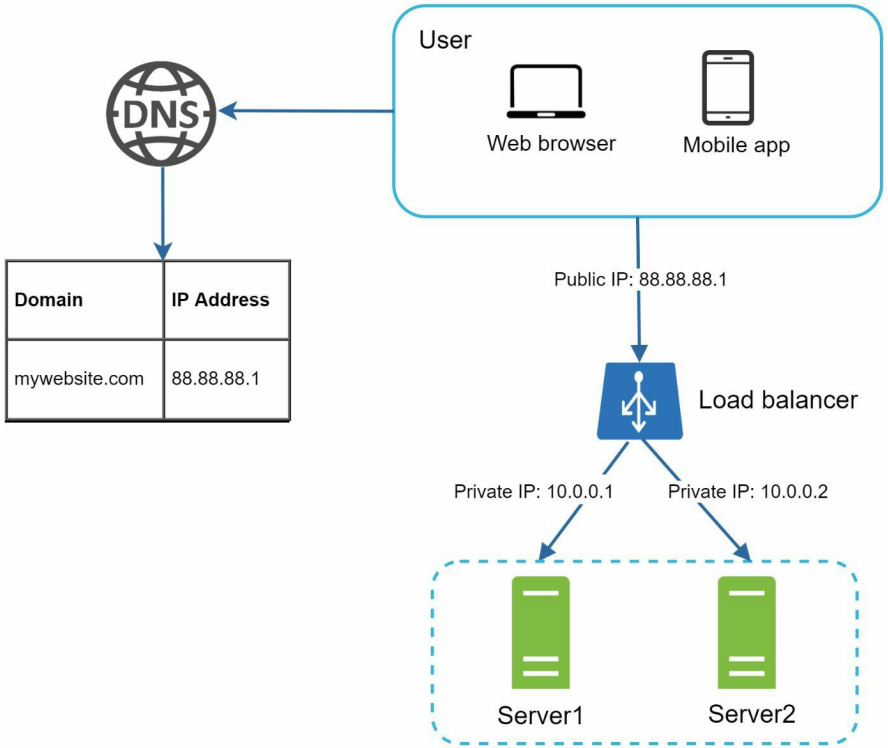

Figure 1-4

Database replication

- Database replication is usually done with master/slave principle.
- A master (original) database generally only supports write operations.
- A slave database gets copies of the data from the master database and only supports read operations.
- Most applications require a much higher ratio of reads to writes.
- Thus #slaves > #masters.

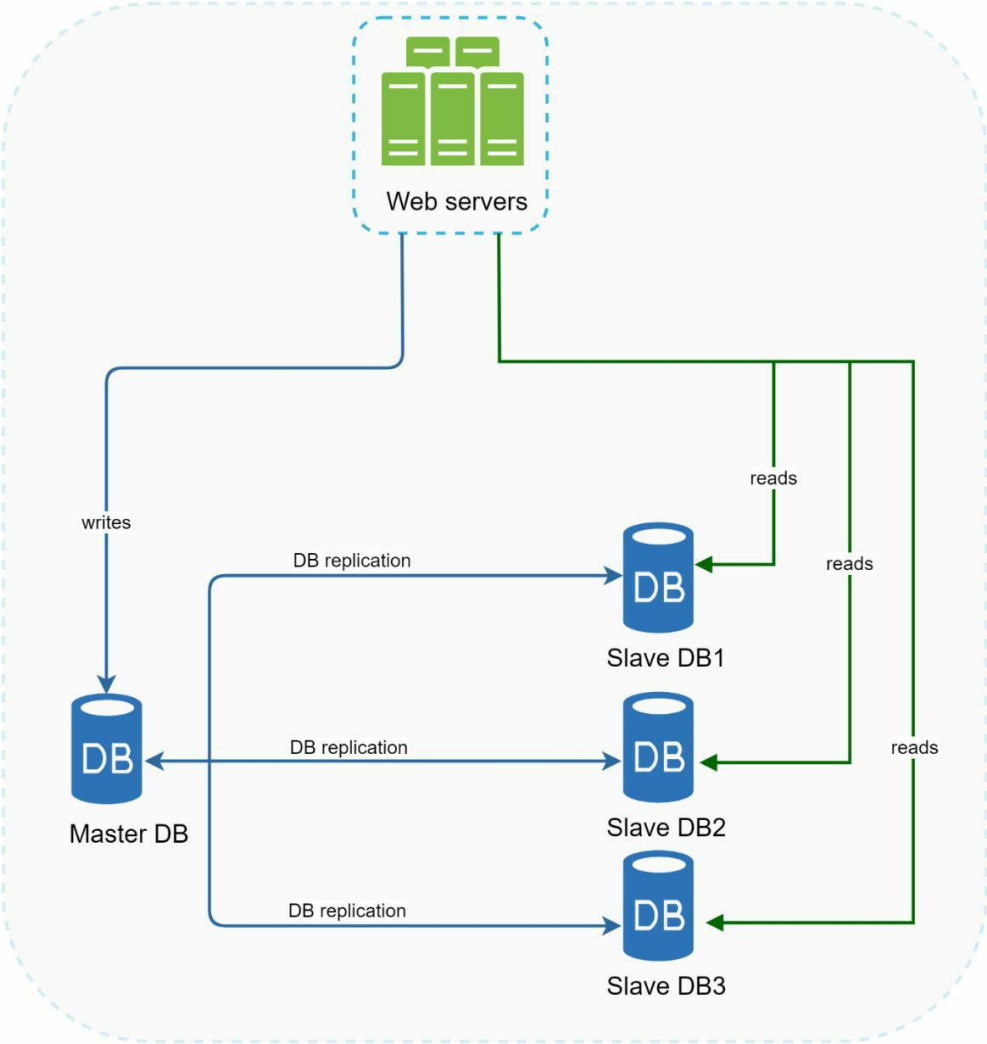

Figure 1-5

Advantages of database replication

- Better performance - this model improves performance because it allows more queries to be processed in parallel.
- Reliability - fail safe, because data is replicated across multiple locations.
- High availability - users can still access data is one database is offline
  - If only one slave database goes offline, read operations will be directed to another slave database or to the master database temporarily.
  - If the master database goes offline, a slave database will be promoted to be the new master.
  - A new slave database will replace the old one or the promoted one.
  - In production systems, promoting a new master is more complicated as the data in a slave database might not be up to date.
  - The missing data needs to be updated by running data recovery scripts.
  - Some other replication methods like multi-masters and circular replication could help, those setups are more complicated.

New design

1. A user gets the IP address of the load balancer from DNS.
2. A user connects the load balancer with this IP address.
3. The HTTP request is routed to either Server 1 or Server 2.
4. A web server reads user data from a slave database.
5. A web server routes any data-modifying operations to the master database. This includes write, update, and delete operations.

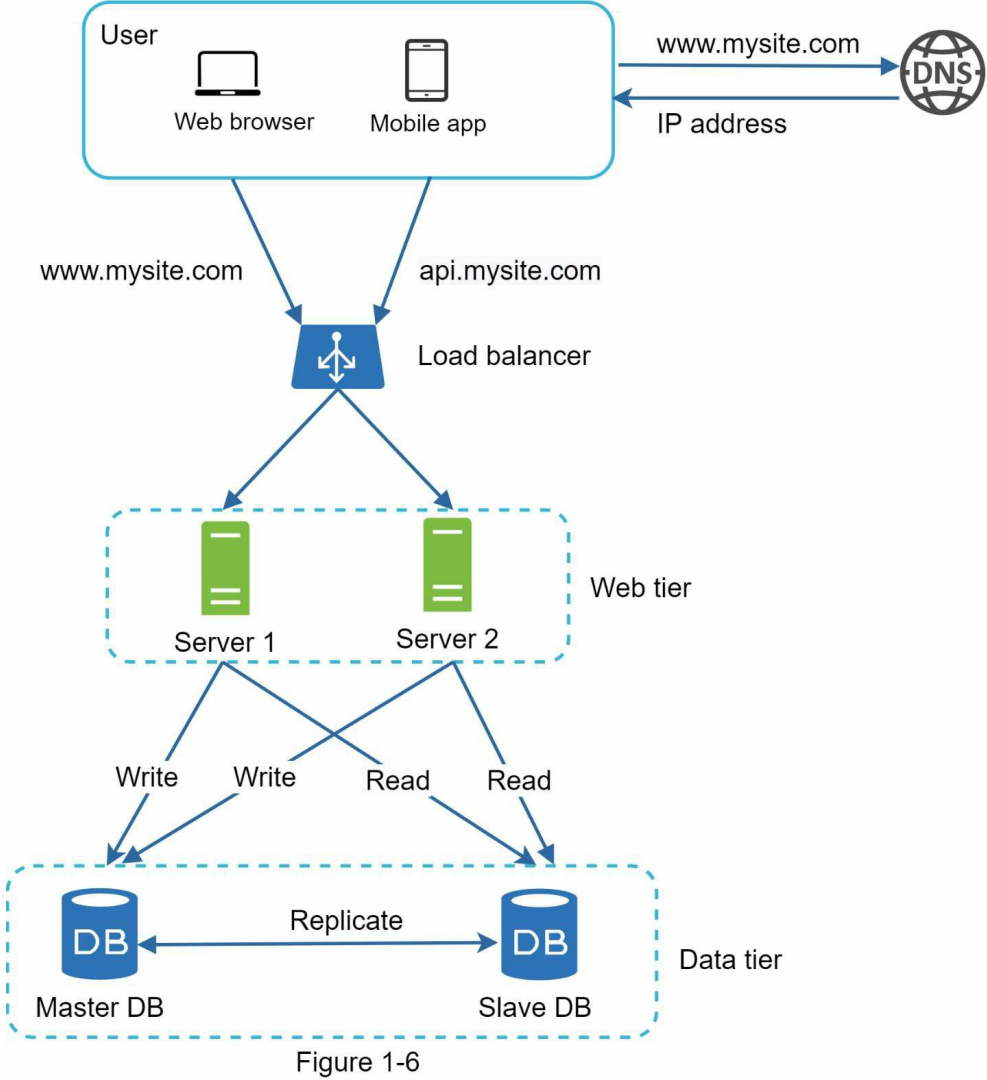

Figure 1-6

Cache

- A cache is a temporary storage area that stores the result of expensive responses or frequently accessed data in memory so that subsequent requests are served more quickly.
- The benefits of having a separate cache tier include better system performance, ability to reduce database workloads, and the ability to scale the cache tier independently.

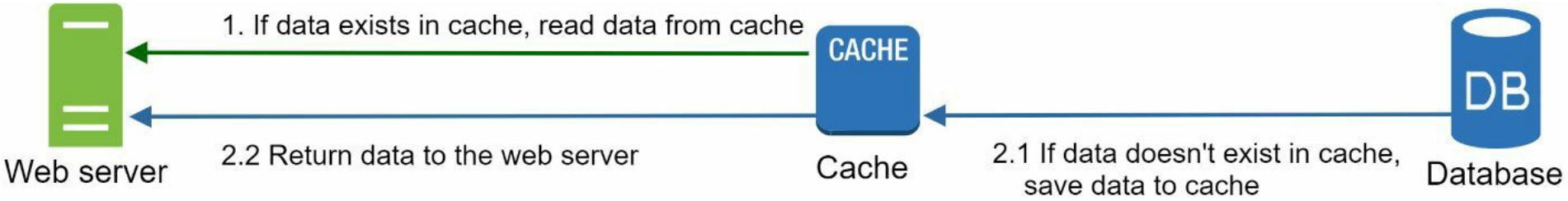

Figure 1-7

Interacting with cache

- Interacting with cache servers is simple because most cache servers provide APIs for common programming languages.
- The following code snippet shows typical Memcached APIs:

```
SECONDS = 1

cache.set('myKey', 'hi there', 3600 * SECONDS)

cache.get('myKey')
```

When and how to use cache

- Consider using cache when data is read frequently but modified infrequently
- Cached data is stored in volatile memory, so it is not ideal for persisting data.
- If a cache server restarts, all the data in memory is lost.
- Data store and the cache must be kept in sync.
- Multiple cache servers across different data centers are recommended to avoid SPOF (single point of failure).

Cache policies

- It is a good practice to implement an **expiration policy.**
- Once cached data is expired, it is removed from the cache.
- It is advisable not to make the expiration date too short as this will cause the system to reload data from the database too frequently.
- It is also advisable not to make the expiration date too long as the data can become stale.
- **Eviction policy:** Once the cache is full, any requests to add items to the cache might cause existing items to be removed.
- Least-recently-used (LRU) is the most popular cache eviction policy.
- Other eviction policies, such as the Least Frequently Used (LFU) or First in First Out (FIFO), can be adopted too.

Content delivery network (CDN)

- A CDN is a network of geographically dispersed servers used to deliver static content.
- CDN servers cache content like images, videos, CSS, JavaScript files, etc.
- When a user visits a website, a CDN server closest to the user will deliver static content.

Considerations of using a CDN

- Cost: CDNs are run by third-party providers and you are charged for data transfers in and out of the CDN.
- Caching infrequently used assets provides no significant benefits so you should consider moving them out of the CDN.
- Setting an appropriate cache expiry.
- The cache expiry time should neither be too long nor too short.
- CDN fallback: You should consider how your website/application copes with CDN failure.
- In case of the CDN outage, clients should be able to request resources from the origin.
- Invalidating files: You can remove a file from the CDN before it expires:
    - Invalidate the CDN object using APIs provided by CDN vendors.
    - Use object versioning to serve a different version of the object. To version an object, you can add a parameter to the URL, such as a version number. For example, version number 2 is added to the query string: image.png?v=2.

Design after adding cache

- Static assets (JS, CSS, images, etc.,) are no longer served by web servers. They are fetched from the CDN for better performance.
- The database load is lightened by caching data.

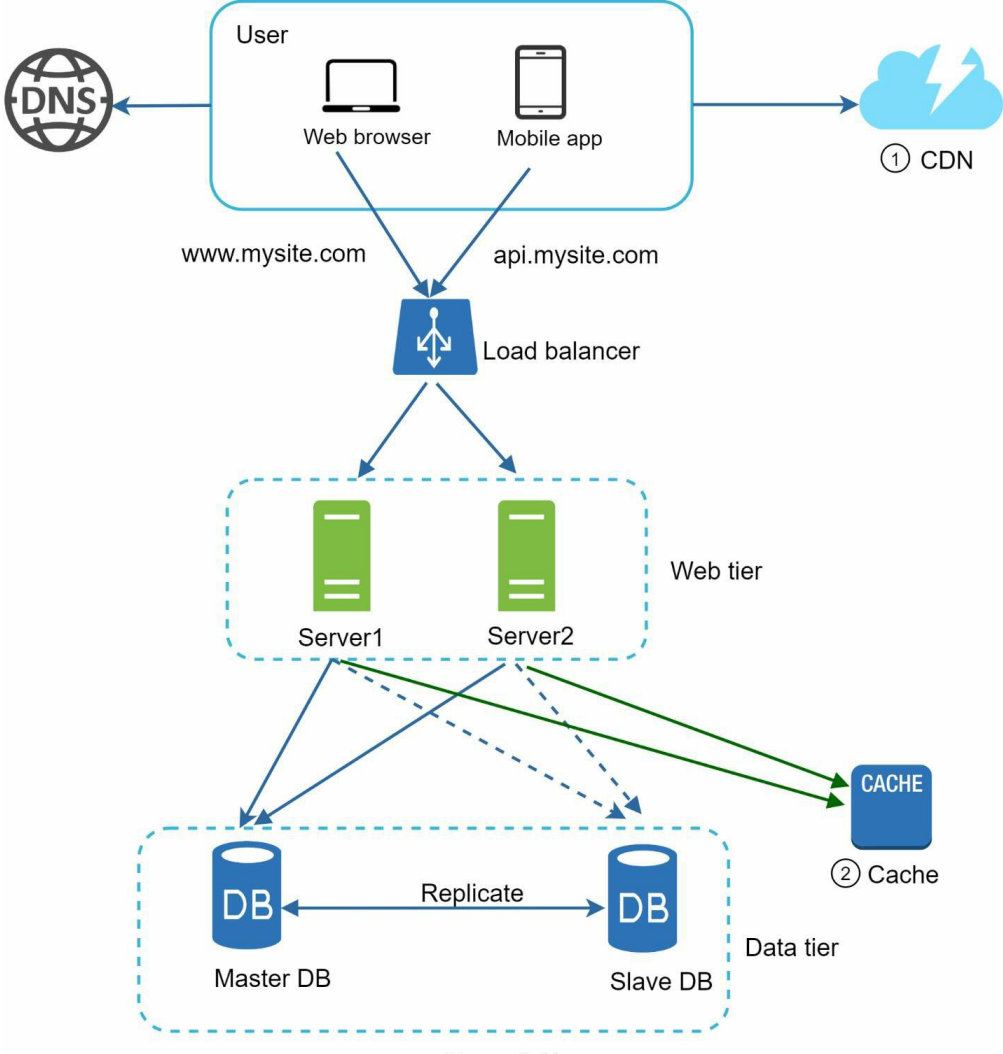

Figure 1-11

Stateful architecture

- A stateful server remembers client data (state) from one request to the next.
- User is authenticated only with one server.
- The issue is that every request from the same client must be routed to the same server.
- This can be done with sticky sessions in most load balancers
- However, this adds the overhead.
- Adding or removing servers is much more difficult with this approach.
- It is challenging to handle server failures.

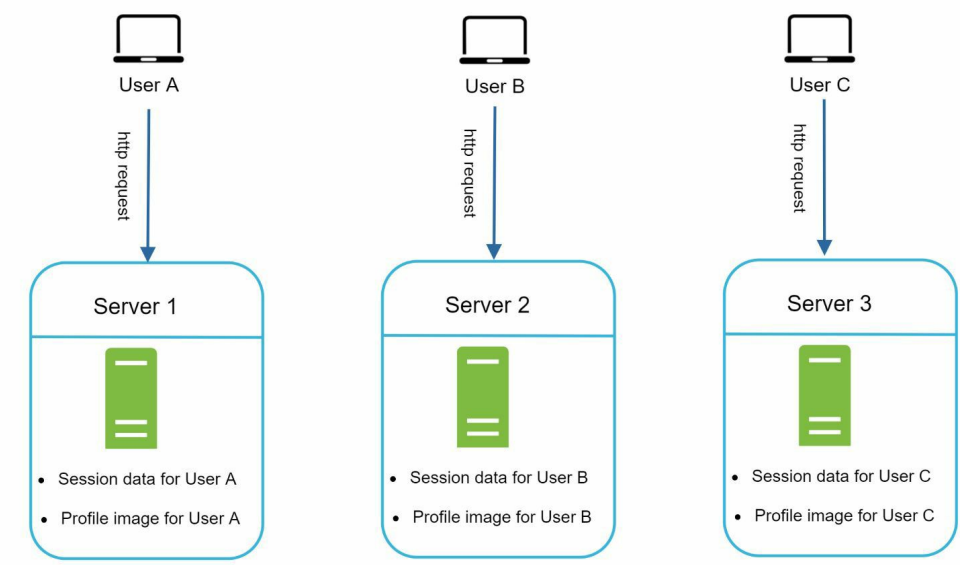

Figure 1-12

Stateless architecture

- In this stateless architecture, HTTP requests from users can be sent to any web servers.
- Servers fetch state data from a shared data store.
- State data is stored in a shared data store and kept out of web servers.
- A good practice is to store session data in the persistent storage such as relational database or NoSQL.

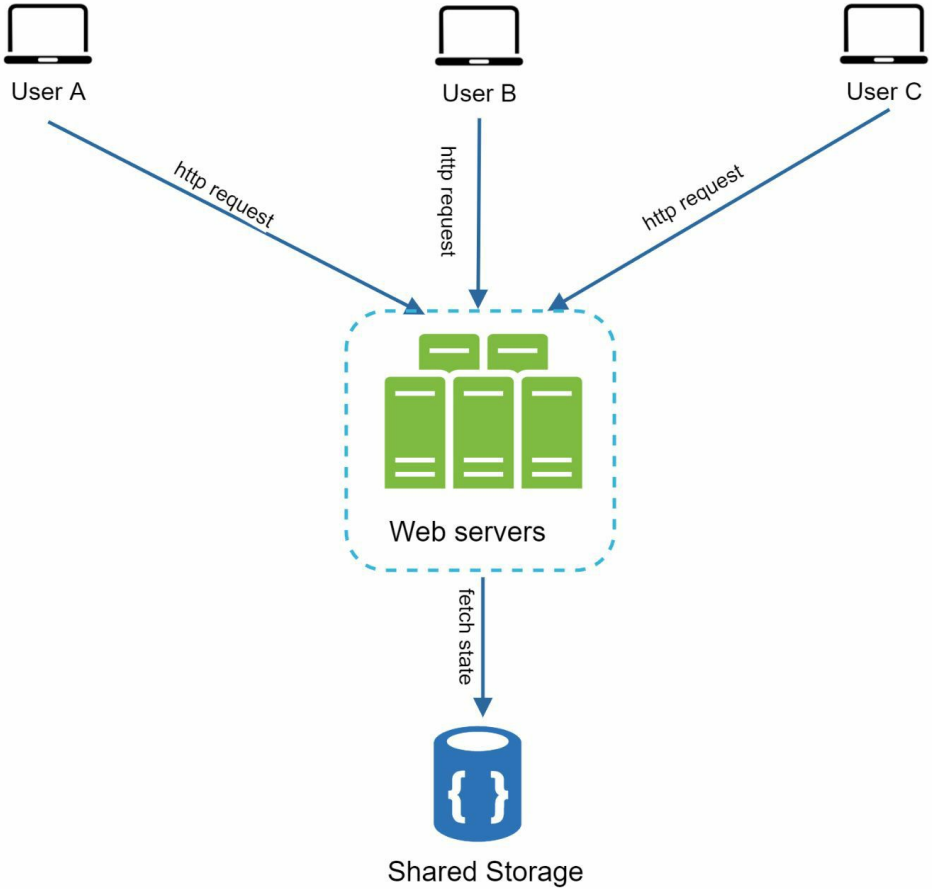

Figure 1-13

Design with stateless web tier

- Autoscaling means adding or removing web servers automatically based on the traffic load.
- After the state data is removed out of web servers, auto-scaling of the web tier is easily achieved.

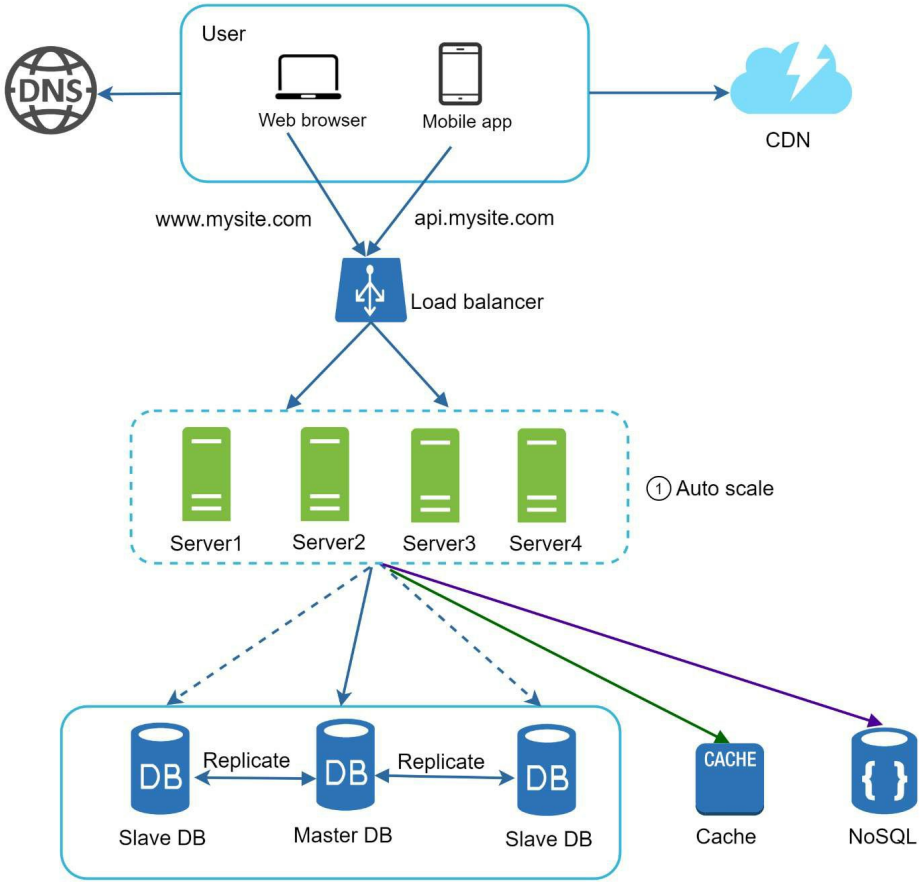

Figure 1-14

Message queues

- To further scale our system, we need to decouple different components of the system so they can be scaled independently.
- Messaging queue is a key strategy employed by many real-world distributed systems to solve this problem.
- A message queue is a durable component, stored in memory, that supports asynchronous communication.
- It serves as a buffer and distributes asynchronous requests.

Message broker architecture

- Message broker is a service that implements message queuing.
- The basic architecture of a message broker is simple.
- Input services, called producers/publishers, create messages, and publish them to a message queue.
- Other services or servers, called consumers/subscribers, connect to the queue, and perform actions defined by the messages.

Figure 1-17

Why use queuing mechanism?

- Decoupling makes the message queue a preferred architecture for building a scalable and reliable application.
- System is more failure resistant.
- With the message queue, the producer can post a message to the queue when the consumer is unavailable to process it.
- The consumer can read messages from the queue even when the producer is unavailable.
- The producer and the consumer can be scaled independently.
- When the size of the queue becomes large, more workers are added to reduce the processing time.
- However, if the queue is empty most of the time, the number of workers can be reduced.

Logging and metrics

- Logging: Monitoring error logs is important because it helps to identify errors and problems in the system.
- You can monitor error logs at per server level or use tools to aggregate them to a centralized service for easy search and viewing.
- Metrics: Collecting different types of metrics help us to gain business insights and understand the health status of the system.
- Some of the following metrics are useful:
    - Host level metrics: CPU, Memory, disk I/O, etc.
    - Key business metrics: daily active users, retention, revenue, etc.

Development automation

- When a system gets big and complex, we need to build or leverage automation tools to improve productivity.
- Continuous integration is a good practice, in which each code check-in is verified through automation, allowing teams to detect problems early.
- Besides, automating your build, test, deploy process, etc. could improve developer productivity significantly.

Current design

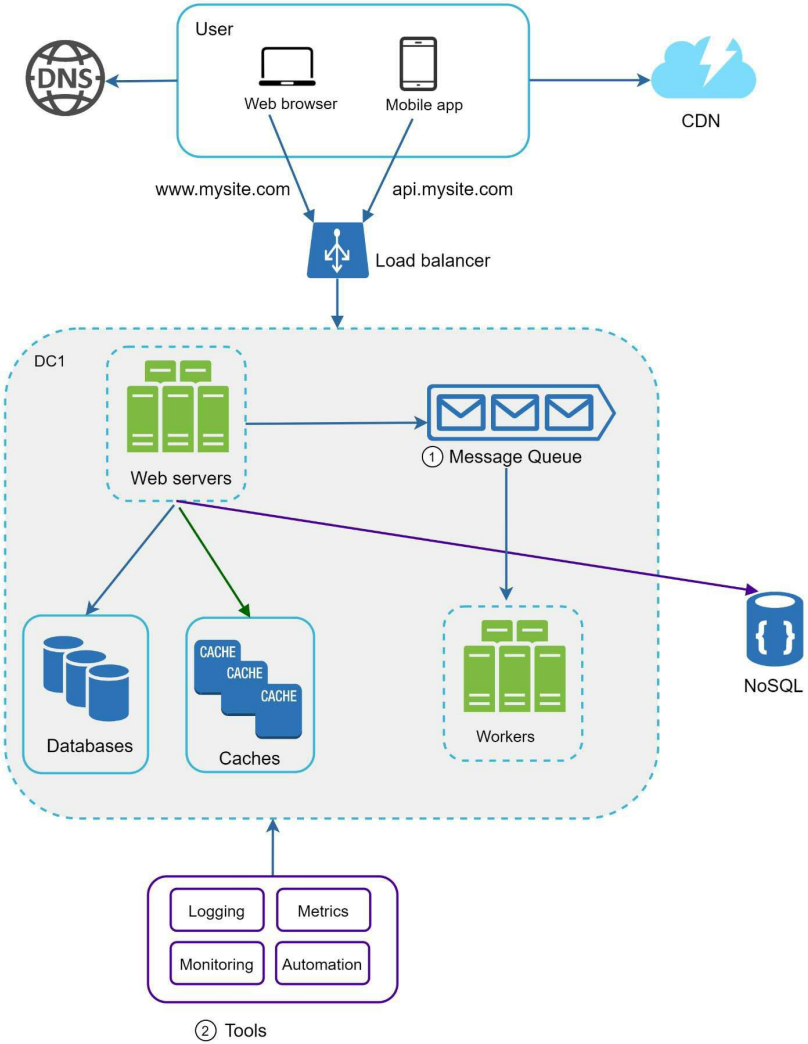

Figure 1-19

Database scaling

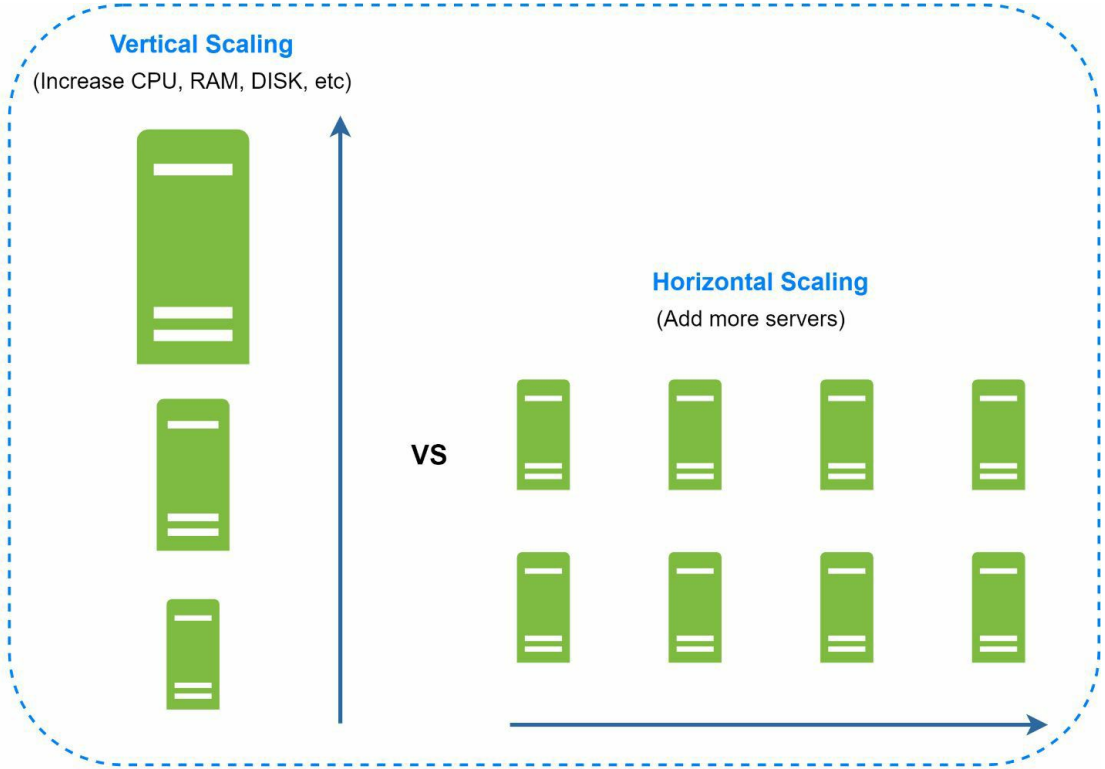

Figure 1-20

Vertical database scaling

- Vertical scaling, also known as scaling up, is the scaling by adding more power (CPU, RAM, DISK, etc.) to an existing machine.
- Vertical scaling comes with some serious drawbacks:
  - You can add more CPU, RAM, etc. to your database server, but there are hardware limits.
  - Greater risk of single point of failures.
  - The overall cost of vertical scaling is high. Powerful servers are much more expensive.

Horizontal database scaling

- Horizontal scaling, also known as **sharding**, is the practice of adding more servers.
- Sharding separates large databases into smaller parts called shards.
- Each shard shares the same schema, though the actual data on each shard is unique to the shard.
- Anytime you access data, a hash function is used to find the corresponding shard.

Key sharding - example

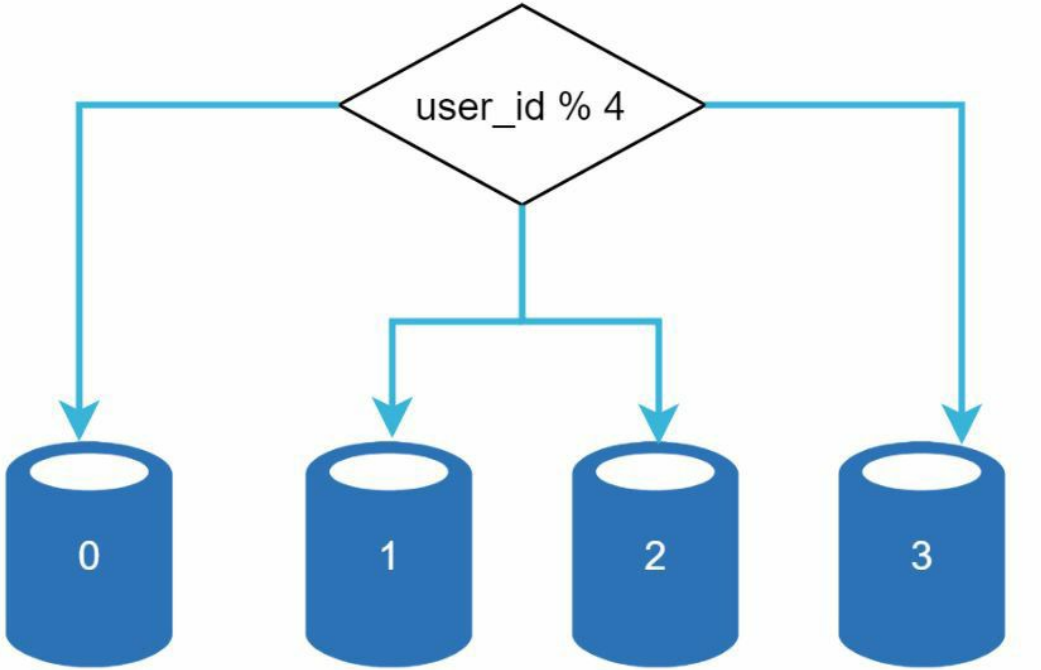

Figure 1-21

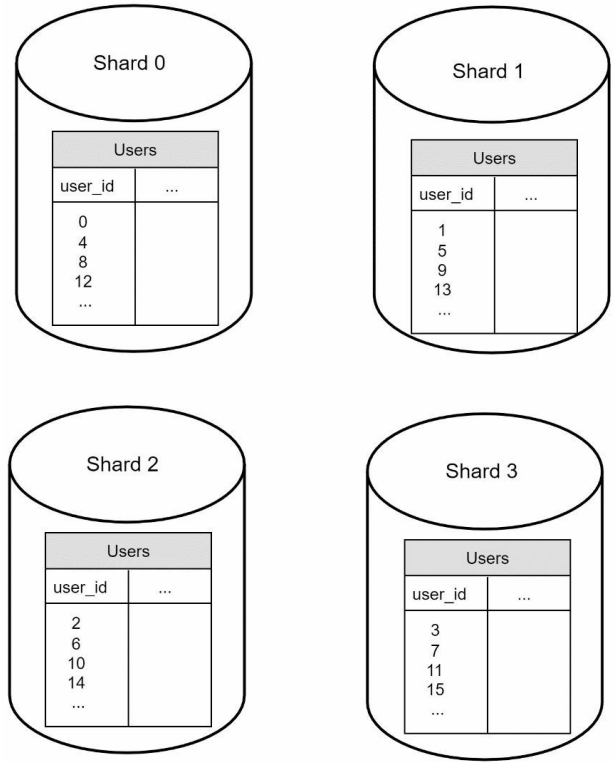

Figure 1-22

Sharding key

- A sharding key allows you to retrieve and modify data efficiently by routing database queries to the correct database.
- When choosing a sharding key, one of the most important criteria is to choose a key that can evenly distributed data.
- The most important factor to consider when implementing a sharding strategy is the choice of the **sharding key**.
- Sharding key (known as a partition key) consists of one or more columns that determine how data is distributed.

Database partitioning

- Partitioning is the process of dividing a large database table into smaller, more manageable pieces to improve performance and scalability.
- Vertical vs horizontal partitioning:
  - Horizontal Partitioning – Splitting rows into multiple tables based on a range or hash (e.g., users by region).
  - Vertical Partitioning – Splitting columns into different tables (e.g., frequently accessed vs. rarely accessed columns).
- Horizontal partitioning types
  - Range Partitioning - Data is divided into partitions based on a continuous range of values (e.g., sales data partitioned by year: 2020, 2021, 2022).
  - List Partitioning – Data is divided based on predefined categories (e.g., orders by country).
  - Hash Partitioning – Data is distributed evenly using a hash function to prevent hotspots.

Partitioning vs Sharding

- Partitioning keeps data within the same database server.
- Sharding splits data across multiple servers.
- Partitioning changes tables names.
- With sharding everything is the same, but the server.

Problems with partitioning and sharding

- Table updates that move rows from one partition/shard to another - slow and complex.
- Queries/transactions across shards/partitions are a problem - could slow down the performance.
- Schema updates are hard and error prone - especially for sharding.
- Developer (client) must be aware of sharding which adds complexity.
- Joins with sharding are complex.
- Resharding data is needed when:
  - A single shard could no longer hold more data due to rapid growth.
  - Certain shards might experience shard exhaustion faster than others due to uneven data distribution.
- When shard exhaustion happens, it requires updating the sharding function and moving data around.
- Celebrity problem - Excessive access to a specific shard could cause server overload.

Summary

- Keep web tier stateless
- Build redundancy at every tier
- Cache data as much as you can
- Host static assets in CDN
- Improve performance with database partitioning
- Scale your data tier by sharding
- Support multiple data centers
- Split tiers into individual services
- Monitor your system and use automation tools

Back-of-the-envelope estimation

Definition

- A back-of-the-envelope estimation is a quick and rough calculation used to approximate a value or solve a problem without detailed data or complex computations.
- It helps assess feasibility and make informed decisions efficiently.
- The term comes from the idea that such estimates can be done on the back of an envelope using just a pen and basic math.
- They are used in system design to get a good feel for which designs will meet your requirements.

Operation name	Time
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns = 10 µs
Send 2K bytes over 1 Gbps network	20,000 ns = 20 µs
Read 1MB sequentially from memory	250,000 ns = 250 µs
Disk seek	10,000,000 ns = 10 ms
Read 1MB sequentially from the network	10,000,000 ns = 10 ms
Read 1MB sequentially from disk	30,000,000 ns = 30 ms

Conclusions

- Memory is fast but the disk is slow.
- Avoid disk seeks if possible.
- Simple compression algorithms are fast.
- Compress data before sending it over the internet if possible.

Availability

- High availability is the ability of a system to be continuously operational for a desirably long period of time.
- High availability is measured as a percentage, with 100% means a service that has 0 downtime.
- Most services fall between 99% and 100%.
- A service level agreement (SLA) is a commonly used term for service providers.
- This is an agreement between the service provider and customer, and this agreement formally defines the level of uptime your service will deliver.

Availability

Availability %	Downtime per day	Downtime per year
99%	14.40 minutes	3.65 days
99.9%	1.44 minutes	8.77 hours
99.99%	8.64 seconds	52.60 minutes
99.999%	864.00 milliseconds	5.26 minutes
99.9999%	86.40 milliseconds	31.56 seconds

Example

- Estimate Twitter QPS (queries per second) and storage requirements.
- Assumptions:
  - 300 million monthly active users.
  - 50% of users use Twitter daily.
  - Users post 2 tweets per day on average.
  - 10% of tweets contain media.
  - Data is stored for 5 years.
  - Average tweet size:
    - Tweet_id - 64 bytes
    - Text - 140 bytes
    - Media - 1MB

Solution

- Daily active users (DAU) = 300 million * 50% = 150 million
- QPS = 150 million * 2 / 24 hour / 3600 seconds ~= 3500
- Peek QPS = 2 * QPS ~= 7000
- Media storage: 150 million * 2 * 10% * 1MB = 30 TB per day.
- 5-year media storage: 30 TB * 365 * 5 ~= 55 PB
- Text storage: 150 million * 2 * 200 bytes ~= 55 GB per day
- 5-year text storage: 55 GB * 365 * 5 ~= 100 TB