System Design

Summary

- About me.
- Experience in building high scale systems.
- Industry need of design
- Resume building and Job Application
- System Design Fundamentals
- Detailed walkthrough on building blocks
- Real world design problems and Case Studies
- Q&A

System Design Fundamentals

Glosssary

- Introduction
- HLD vs LLD
- Design Fundamentals
  - Client-Server Model, Storage basics, Latency & Throughput, Availability
  - Redundancy/SLA/SLO, Caching, Proxy, Load Balancer, Hashing
  - ACID Transaction, Consistency, Relational databases, Non-relational databases
  - Key-Value store, Specialized Storage Paradigms, Replication and Sharding
  - Leader Election, Peer to Peer Network, Polling and Streaming, Rate Limiting
  - Logging, Monitoring and Alerting, Publish/Subscribe Pattern, Distributed File System, Monolith vs Microservice Architecture, The CAP theorem, PACELC theorem

Introduction

- **System Design** is the process of designing the architecture, components, and interfaces for a system **that meet a specified set of functional and non-functional requirements.**
- **System design aims to build systems that are reliable, effective, and maintainable, among other characteristics.**
  - Reliable systems **handle faults, failures, and errors.**
  - Effective systems **meet all user needs and business requirements.**
  - Maintainable systems **are flexible and easy to scale up or down. The ability to add new features also comes under the umbrella of maintainability.**
- The high level design (**HLD**) deals on a macroscopic level of the system while the low level language deals on a microscopic level(**LLD**) of the system.

HLD vs LLD

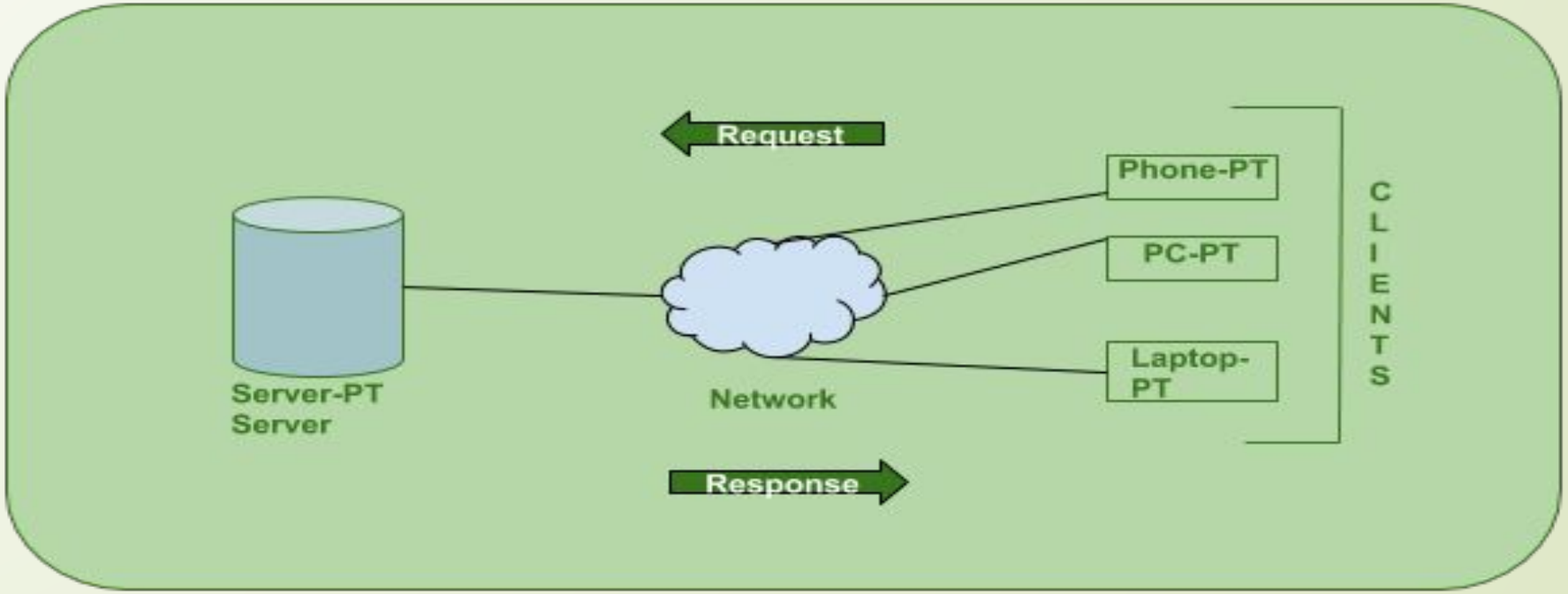
High-Level Design	Low-Level Design
The focus is more on designing the high-level architecture of the system, defining the high-level components with their interactions, and also the database design.	The focus is more on designing each component in detail such as what classes are needed, what abstractions to use, how object creation should happen, how data flows between different objects, etc.
HLD converts the business requirements to a high-level solution.	LLD converts the high-level design into detailed design (ready to code) components.

Design Fundamentals

- Building scalable, production-ready applications is both art and science.
- Science, in that it requires knowledge of many topics in computer engineering; art, in that it demands an eye for making smart design choices and piecing together the right technologies.

Client-Sever model

- The paradigm by which modern systems are designed, which consists of clients requesting data or service from servers and servers providing data or service to clients.



Storage Basics: Memory/Disk/Database

- Memory
  - Short for Random Access Memory (RAM).
  - Data stored in memory will be lost when the process containing that data dies.
- Disk
  - Usually refers to either HDD (hard-disk drive) or SSD (solid-state drive).
  - Data written to disk will persist through power failures and general machine crashes.
- Databases:
  - Databases are programs that either use disk or memory to do two core things: **record data and query data**.
  - In general, they are themselves servers that are long lived and interact with the rest of your application through network calls, with protocols on top of TCP or even HTTP.

Power of Two

Although data volume can become enormous when dealing with distributed systems, calculation all boils down to the basics. To obtain correct calculations, it is critical to know the data volume unit using the power of 2. A byte is a sequence of 8 bits. An ASCII character uses one byte of memory (8 bits). Below is a table explaining the data volume unit (Table 2-1).

Power	Approximate value	Full name	Short name
10	1 Thousand	1 Kilobyte	1 KB
20	1 Million	1 Megabyte	1 MB
30	1 Billion	1 Gigabyte	1 GB
40	1 Trillion	1 Terabyte	1 TB
50	1 Quadrillion	1 Petabyte	1 PB

Latency and Throughput

Latency

The time it takes for a certain operation to complete in a system. Most often this measure is a time duration, like milliseconds or seconds. You should know these orders of magnitude:

- **Reading 1 MB from RAM**: 250 µs (0.25 ms)
- **Reading 1 MB from SSD**: 1,000 µs (1 ms)
- **Transfer 1 MB over Network**: 10,000 µs (10 ms)
- **Reading 1MB from HDD**: 20,000 µs (20 ms)
- **Inter-Continental Round Trip**: 150,000 µs (150 ms)

Throughput

The number of operations that a system can handle properly per time unit. For instance the throughput of a server can often be measured in requests per second (RPS or QPS).

Availability

Availability

The odds of a particular server or service being up and running at any point in time, usually measured in percentages. A server that has 99% availability will be operational 99% of the time (this would be described as having two **nines** of availability).

High Availability

Used to describe systems that have particularly high levels of availability, typically 5 nines or more; sometimes abbreviated "HA".

Nines

Typically refers to percentages of uptime. For example, 5 nines of availability means an uptime of 99.999% of the time. Below are the downtimes expected per year depending on those 9s:

```
- 99% (two 9s): 87.7 hours
- 99.9% (three 9s): 8.8 hours
- 99.99%: 52.6 minutes
- 99.999%: 5.3 minutes
```

Redundancy/SLA/SLO

Redundancy

The process of replicating parts of a system in an effort to make it more reliable.

SLA

Short for "service-level agreement", an SLA is a collection of guarantees given to a customer by a service provider. SLAs typically make guarantees on a system's availability, amongst other things. SLAs are made up of one or multiple SLOs.

SLO

Short for "service-level objective", an SLO is a guarantee given to a customer by a service provider. SLOs typically make guarantees on a system's availability, amongst other things. SLOs constitute an SLA.

Caching

Cache

A piece of hardware or software that stores data, typically meant to retrieve that data faster than otherwise.

Caches are often used to store responses to network requests as well as results of computationally-long operations.

Note that data in a cache can become **stale** if the main source of truth for that data (i.e., the main database behind the cache) gets updated and the cache doesn't.

Cache Hit

When requested data is found in a cache.

Cache Miss

When requested data could have been found in a cache but isn't. This is typically used to refer to a negative consequence of a system failure or of a poor design choice. For example:

If a server goes down, our load balancer will have to forward requests to a new server, which will result in cache misses.

Cache Eviction Policy

The policy by which values get evicted or removed from a cache. Popular cache eviction policies include **LRU** (least-recently used), **FIFO** (first in first out), and **LFU** (least-frequently used).

Proxy

Forward Proxy

A server that sits between a client and servers and acts on behalf of the client, typically used to mask the client's identity (IP address). Note that forward proxies are often referred to as just proxies.

Reverse Proxy

A server that sits between clients and servers and acts on behalf of the servers, typically used for logging, load balancing, or caching.

Nginx

Pronounced "engine X"—not "N jinx", Nginx is a very popular webserver that's often used as a **reverse proxy** and **load balancer**.

Learn more: https://www.nginx.com/

Load Balancer

A type of **reverse proxy** that distributes traffic across servers. Load balancers can be found in many parts of a system, from the DNS layer all the way to the database layer.

Server-Selection Strategy

How a **load balancer** chooses servers when distributing traffic amongst multiple servers. Commonly used strategies include round-robin, random selection, performance-based selection (choosing the server with the best performance metrics, like the fastest response time or the least amount of traffic), and IP-based routing.

Hot Spot

When distributing a workload across a set of servers, that workload might be spread unevenly. This can happen if your **sharding key** or your **hashing function** are suboptimal, or if your workload is naturally skewed: some servers will receive a lot more traffic than others, thus creating a "hot spot".

Hashing

Consistent Hashing

A type of hashing that minimizes the number of keys that need to be remapped when a hash table gets resized. It's often used by load balancers to distribute traffic to servers; it minimizes the number of requests that get forwarded to different servers when new servers are added or when existing servers are brought down.

Rendezvous Hashing

A type of hashing also coined **highest random weight** hashing. Allows for minimal re-distribution of mappings when a server goes down.

SHA

Short for "Secure Hash Algorithms", the SHA is a collection of cryptographic hash functions used in the industry. These days, SHA-3 is a popular choice to use in a system.

ACID Transaction

ACID Transaction

A type of database transaction that has four important properties:

- **Atomicity**: The operations that constitute the transaction will either all succeed or all fail. There is no in-between state.
- **Consistency**: The transaction cannot bring the database to an invalid state. After the transaction is committed or rolled back, the rules for each record will still apply, and all future transactions will see the effect of the transaction. Also named **Strong Consistency**.
- **Isolation**: The execution of multiple transactions concurrently will have the same effect as if they had been executed sequentially.
- **Durability**: Any committed transaction is written to non-volatile storage. It will not be undone by a crash, power loss, or network partition.

Consistency

Strong Consistency

Strong Consistency usually refers to the consistency of ACID transactions, as opposed to **Eventual Consistency**.

Eventual Consistency

A consistency model which is unlike **Strong Consistency**. In this model, reads might return a view of the system that is stale. An eventually consistent datastore will give guarantees that the state of the database will eventually reflect writes within a time period (could be 10 seconds, or minutes).

Relational/SQL databases

- A type of structured database in which data is stored following a tabular format; often supports powerful querying using SQL.
- SQL databases are valuable in handling structured data, or data that has relationships between its variables and entities.
- Scalability
  - In general, SQL databases can scale vertically, meaning you can increase the load on a server by migrating to a larger server that adds more CPU, RAM or SSD capability.
  - While vertical scalability is used most frequently, SQL databases can also scale horizontally through sharding or partitioning logic, although that's not well-supported.

SQL

Structured Query Language. Relational databases can be used using a derivative of SQL such as PostgreSQL in the case of Postgres.

SQL Database

Any database that supports SQL. This term is often used synonymously with "Relational Database", though in practice, not *every* relational database supports SQL.

Non-Relational/NoSQL databases

NoSQL databases are not relational, so they don't solely store data in rows and tables. Instead, they generally fall into one of four types of structures:

- **Column-oriented,** where data is stored in cells grouped in a virtually unlimited number of columns rather than rows.
- **Key-value stores**, which use an associative array (also known as a dictionary or map) as their data model. This model represents data as a collection of key-value pairs.
- **Document stores**, which use documents to hold and encode data in standard formats, including XML, YAML, JSON (JavaScript Object Notation) and BSON. A benefit is that documents within a single database can have different data types.
- **Graph databases**, which represent data on a graph that shows how different sets of data relate to each other. Neo4j, RedisGraph (a graph module built into Redis) and OrientDB are examples of graph databases.

SQL vs NoSQL databases

	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune
Schemas	Rigid	Flexible
Scaling	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)
Multi-Record ACID Transactions	Supported	Most do not support multi-record ACID transactions. However, some — like MongoDB — do.

Key-Value Store

Key-Value Store

A Key-Value Store is a flexible NoSQL database that's often used for caching and dynamic configuration. Popular options include DynamoDB, Etcd, Redis, and ZooKeeper.

Etcd

Etcd is a strongly consistent and highly available key-value store that's often used to implement leader election in a system.

Learn more: https://etcd.io/

Redis

An in-memory key-value store. Does offer some persistent storage options but is typically used as a really fast, best-effort caching solution. Redis is also often used to implement **rate limiting**.

Learn more: https://redis.io/

ZooKeeper

ZooKeeper is a strongly consistent, highly available key-value store. It's often used to store important configuration or to perform leader election.

Learn more: https://zookeeper.apache.org/

Specialized Storage Paradigms

Blob Storage

Widely used kind of storage, in small and large scale systems. They don't really count as databases per se, partially because they only allow the user to store and retrieve data based on the name of the blob. This is sort of like a key-value store but usually blob stores have different guarantees. They might be slower than KV stores but values can be megabytes large (or sometimes gigabytes large). Usually people use this to store things like **large binaries, database snapshots, or images** and other static assets that a website might have.

Blob storage is rather complicated to have on premise, and only giant companies like Google and Amazon have infrastructure that supports it. So usually in the context of System Design interviews you can assume that you will be able to use **GCS** or **S3**. These are blob storage services hosted by Google and Amazon respectively, that cost money depending on how much storage you use and how often you store and retrieve blobs from that storage.

Time Series Database

A **TSDB** is a special kind of database optimized for storing and analyzing time-indexed data: data points that specifically occur at a given moment in time. Examples of TSDBs are InfluxDB, Prometheus, and Graphite.

Spatial Database

A type of database optimized for storing and querying spatial data like locations on a map. Spatial databases rely on spatial indexes like **quadtrees** to quickly perform spatial queries like finding all locations in the vicinity of a region.

Quadtree

A tree data structure most commonly used to index two-dimensional spatial data. Each node in a quadtree has either zero children nodes (and is therefore a leaf node) or exactly four children nodes.

Replication and Sharding

Replication

The act of duplicating the data from one database server to others. This is sometimes used to increase the redundancy of your system and tolerate regional failures for instance. Other times you can use replication to move data closer to your clients, thus decreasing the latency of accessing specific data.

Sharding

Sometimes called **data partitioning**, sharding is the act of splitting a database into two or more pieces called **shards** and is typically done to increase the throughput of your database. Popular sharding strategies include:

- Sharding based on a client's region
- Sharding based on the type of data being stored (e.g: user data gets stored in one shard, payments data gets stored in another shard)
- Sharding based on the hash of a column (only for structured data)

Leader Election

Leader Election

The process by which nodes in a cluster (for instance, servers in a set of servers) elect a so-called "leader" amongst them, responsible for the primary operations of the service that these nodes support. When correctly implemented, leader election guarantees that all nodes in the cluster know which one is the leader at any given time and can elect a new leader if the leader dies for whatever reason.

Consensus Algorithm

A type of complex algorithms used to have multiple entities agree on a single data value, like who the "leader" is amongst a group of machines. Two popular consensus algorithms are **Paxos** and **Raft**.

Paxos & Raft

Two consensus algorithms that, when implemented correctly, allow for the synchronization of certain operations, even in a distributed setting.

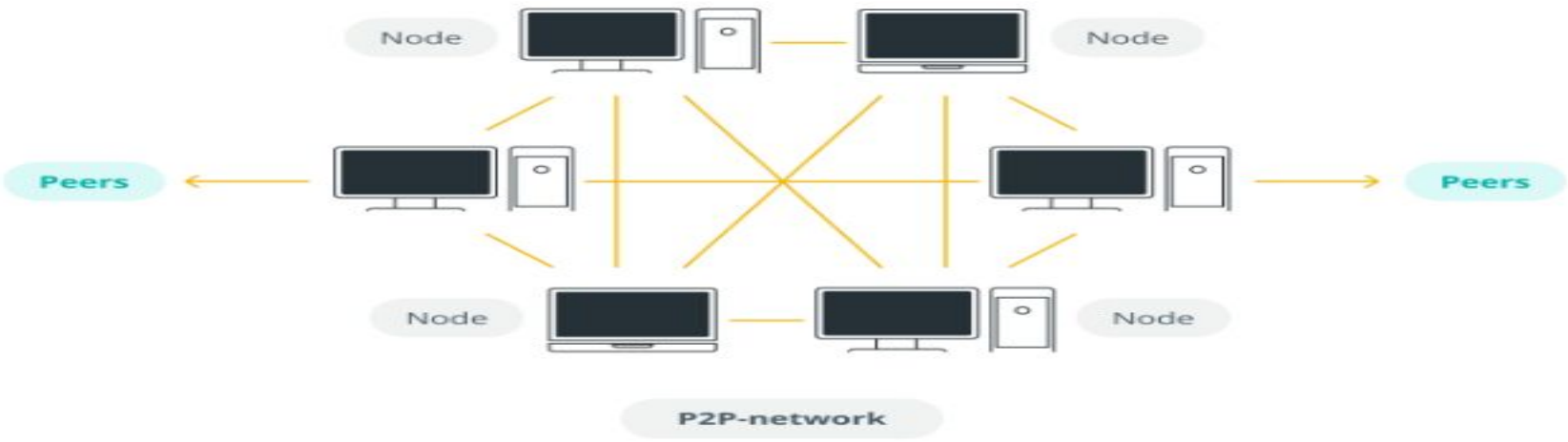
Peer-to-Peer Network

Peer-To-Peer Network

A collection of machines referred to as peers that divide a workload between themselves to presumably complete the workload faster than would otherwise be possible. Peer-to-peer networks are often used in file-distribution systems.

Gossip Protocol

When a set of machines talk to each other in a uncoordinated manner in a cluster to spread information through a system without requiring a central source of data.



Polling and Streaming

Socket

A kind of file that acts like a stream. Processes can read and write to sockets and communicate in this manner. Most of the time the sockets are fronts for TCP connection.

Polling

The act of fetching a resource or piece of data regularly at an interval to make sure your data is not too stale.

Streaming

In networking, it usually refers to the act of continuously getting a feed of information from a server by keeping an open connection between the two machines or processes.

Rate Limiting

Rate Limiting

The act of limiting the number of requests sent to or from a system. Rate limiting is most often used to limit the number of incoming requests in order to prevent **DoS attacks** and can be enforced at the IP-address level, at the user-account level, or at the region level, for example. Rate limiting can also be implemented in tiers; for instance, a type of network request could be limited to 1 per second, 5 per 10 seconds, and 10 per minute.

DoS Attack

Short for "denial-of-service attack", a DoS attack is an attack in which a malicious user tries to bring down or damage a system in order to render it unavailable to users. Much of the time, it consists of flooding it with traffic. Some DoS attacks are easily preventable with rate limiting, while others can be far trickier to defend against.

DDoS Attack

Short for "distributed denial-of-service attack", a DDoS attack is a DoS attack in which the traffic flooding the target system comes from many different sources (like thousands of machines), making it much harder to defend against.

Logging, Monitoring and Alerting

Logging

The act of collecting and storing logs--useful information about events in your system. Typically your programs will output log messages to its STDOUT or STDERR pipes, which will automatically get aggregated into a **centralized logging solution**.

Monitoring

The process of having visibility into a system's key metrics, monitoring is typically implemented by collecting important events in a system and aggregating them in human-readable charts.

Alerting

The process through which system administrators get notified when critical system issues occur. Alerting can be set up by defining specific thresholds on monitoring charts, past which alerts are sent to a communication channel like Slack.

Publish/Subscribe Pattern

Publish/Subscribe Pattern

Often shortened as **Pub/Sub**, the Publish/Subscribe pattern is a popular messaging model that consists of **publishers** and **subscribers**. Publishers publish messages to special **topics** (sometimes called **channels**) without caring about or even knowing who will read those messages, and subscribers subscribe to topics and read messages coming through those topics.

Pub/Sub systems often come with very powerful guarantees like **at-least-once delivery**, **persistent storage**, **ordering** of messages, and **replayability** of messages.

Idempotent Operation

An operation that has the same ultimate outcome regardless of how many times it's performed. If an operation can be performed multiple times without changing its overall effect, it's idempotent. Operations performed through a **Pub/Sub** messaging system typically have to be idempotent, since Pub/Sub systems tend to allow the same messages to be consumed multiple times.

For example, increasing an integer value in a database is *not* an idempotent operation, since repeating this operation will not have the same effect as if it had been performed only once. Conversely, setting a value to "COMPLETE" *is* an idempotent operation, since repeating this operation will always yield the same result: the value will be "COMPLETE".

Apache Kafka

A distributed messaging system created by LinkedIn. Very useful when using the **streaming** paradigm as opposed to **polling**.

Distributed File System

Distributed File System

A Distributed File System is an abstraction over a (usually large) cluster of machines that allows them to act like one large file system. The two most popular implementations of a DFS are the **Google File System** (GFS) and the **Hadoop Distributed File System** (HDFS).

Typically, DFSs take care of the classic **availability** and **replication** guarantees that can be tricky to obtain in a distributed-system setting. The overarching idea is that files are split into chunks of a certain size (4MB or 64MB, for instance), and those chunks are sharded across a large cluster of machines. A central control plane is in charge of deciding where each chunk resides, routing reads to the right nodes, and handling communication between machines.

Different DFS implementations have slightly different APIs and semantics, but they achieve the same common goal: extremely large-scale persistent storage.

Hadoop

A popular, open-source framework that supports MapReduce jobs and many other kinds of data-processing pipelines. Its central component is **HDFS** (Hadoop Distributed File System), on top of which other technologies have been developed.

Microservice vs Monolith Architecture

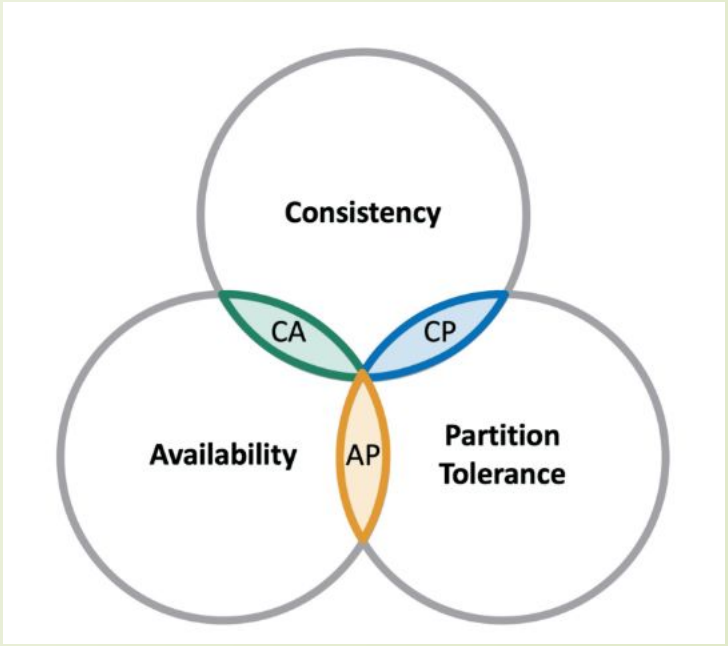
- **Microservice Architecture**
  When a system is made up of many small web services that can be compiled and deployed independently. This is usually thought of as a counterpart of **monoliths**.

- **Monolith Architecture**
  When a system is primarily made up of a single large web application that is compiled and rolled out as a unit. Typically a counterpart of **microservices**. Companies sometimes try to split up this monolith into microservices once it reaches a very large size in an attempt to increase **developer productivity**.

The CAP theorem

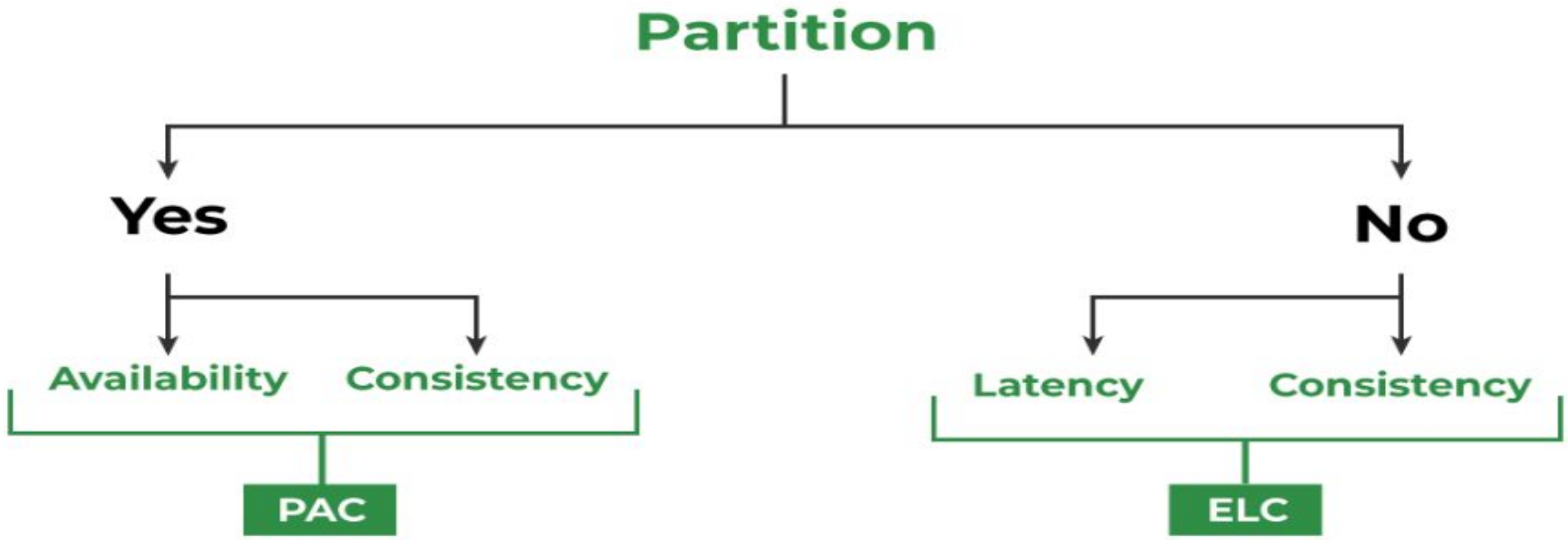
- The three letters in CAP refer to three desirable properties of distributed systems with replicated data: **consistency** (among replicated copies), **availability** (of the system for read and write operations) and **partition tolerance** (in the face of the nodes in the system being partitioned by a network fault).

- The CAP theorem states that it is possible to guarantee only two out of the three desirable properties – consistency, availability, and partition tolerance at the same time in a distributed system with data replication.



The PACELC theorem

PACELC theorem states that in the case of **Network Partition 'P'** a distributed system can have tradeoffs between **Availability 'A'** and **Consistency 'C' Else 'E'** if there is no Network Partition then a distributed system can have tradeoffs between **Latency 'L'** and **Consistency 'C'**.



Detailed Walkthrough on building blocks

Glosssary

- Caching and Distributed Cache
- Load Balancer
- Replication
- Sharding, Partitioning and Consistent Hashing
- Microservice Architecture
- Publisher Subscriber Pattern

Caching at Different Layers of a System

- **A system is designed in layers, and each layer should have its caching mechanism to ensure the decoupling of sensitive data from different layers.**
- **Caching at different locations helps reduce the serving latency at that layer.**
-

System Layer	Technology in Use	Usage
Web	HTTP cache headers, web accelerators, key-value store, CDNs, and so on	Accelerate retrieval of static web content, and manage sessions
Application	Local cache and key-value data store	Accelerate application-level computations and data retrieval
Database	Database cache, buffers, and key-value data store	Reduce data retrieval latency and I/O load from database

How does Caching work ?

- A cache is primarily used to store the most recently or frequently used data, in the hope that it will soon be fetched again.
- Caches are typically faster than databases and services when it comes to re-accessing this stored information. What makes them so fast is the fact that caches store data in SSDs and mostly RAMs which reduces the lookup time. However, this does not mean that we should cache everything.

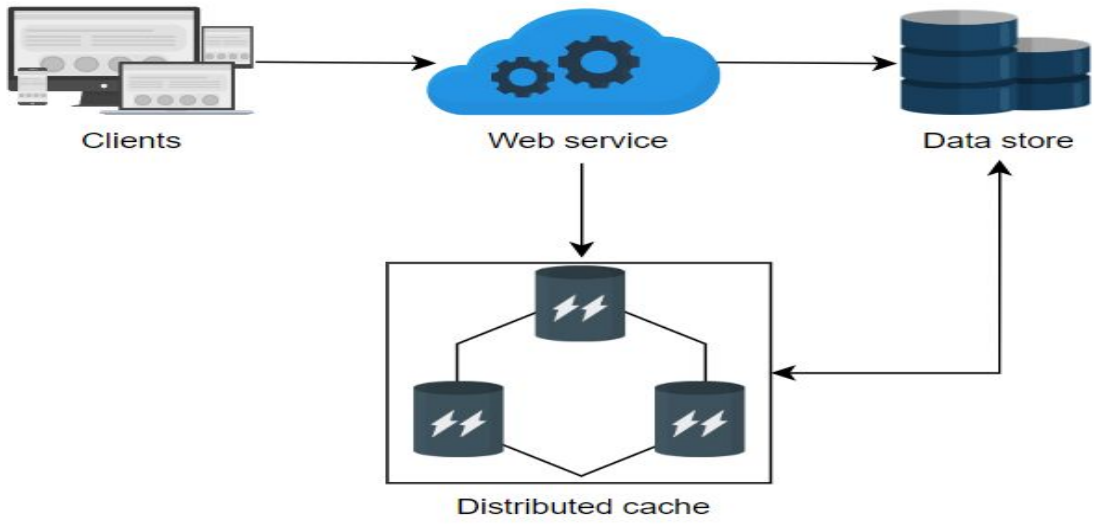
When Not to Cache?

There are some scenarios where the negative aspects of caching outweigh its benefits:

- **High Consistency requirements**: When we fetch the previously stored data from the cache, there is a possibility of stale data being displayed to the user. For example, for a social media app, then some stale data is probably fine. However, for a stock price display app, then the cache must be in sync with the primary data source.
- **Write heavy / Read Once**: When write operations (updates to data) are more frequent than read operations (data retrieval). For example, caching the data of an analytics system would only increase the hardware maintenance cost.
- **Low repetition**: When the action of retrieval of the same information is not frequently repeated by the user. For example, the cost calculated by the trip cost estimation module of a cab booking app between the exact two points need not be cached.

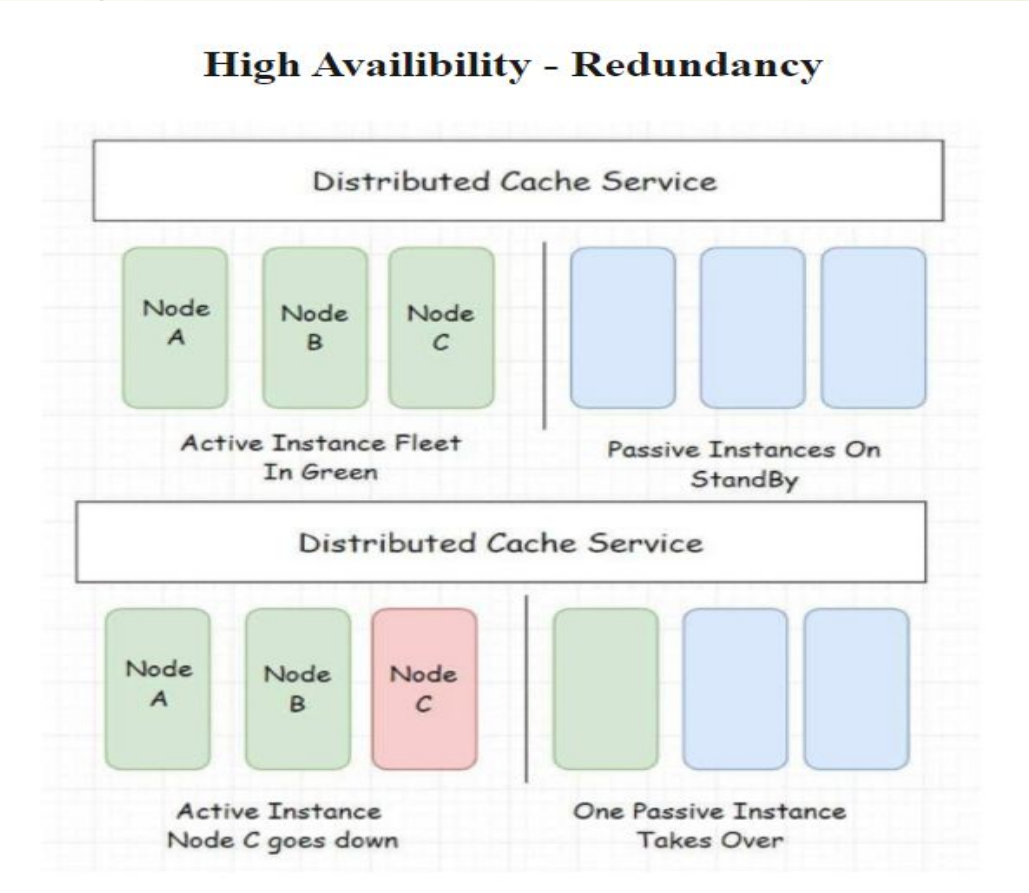
Distributed Cache

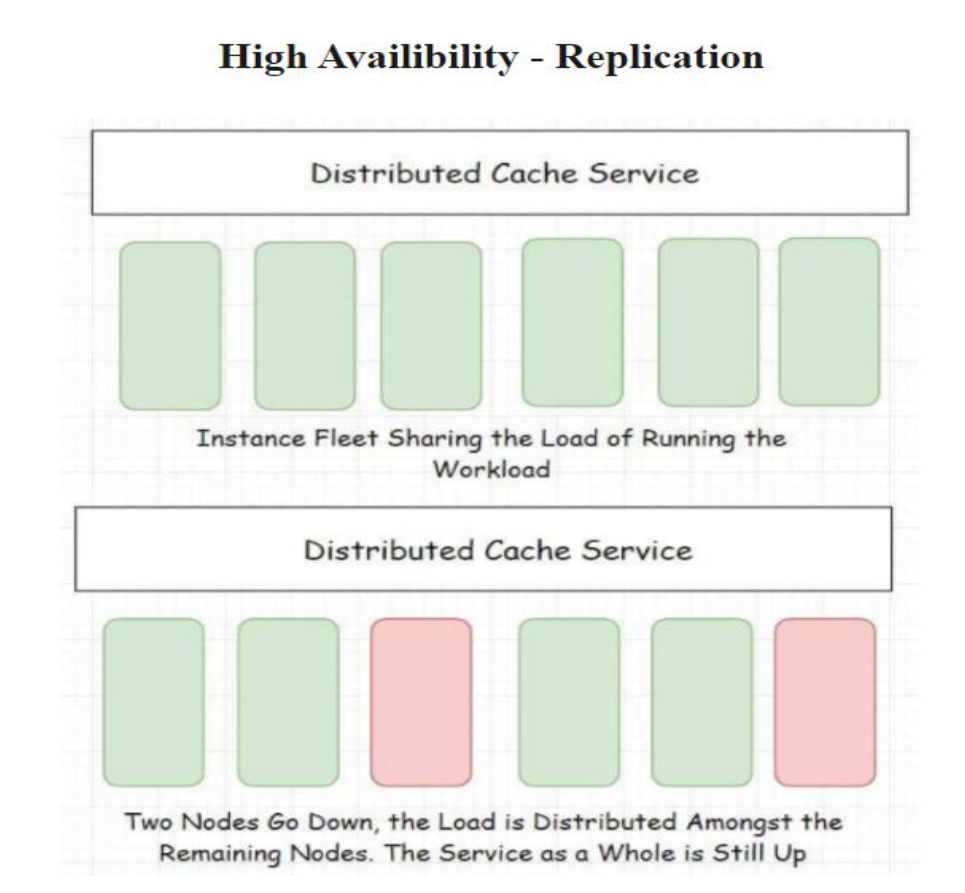
- **A distributed cache is a caching system where multiple cache servers coordinate to store frequently accessed data. Distributed caches are needed in environments where a single cache server isn't enough to store all**



Advantages of Distributed Caching

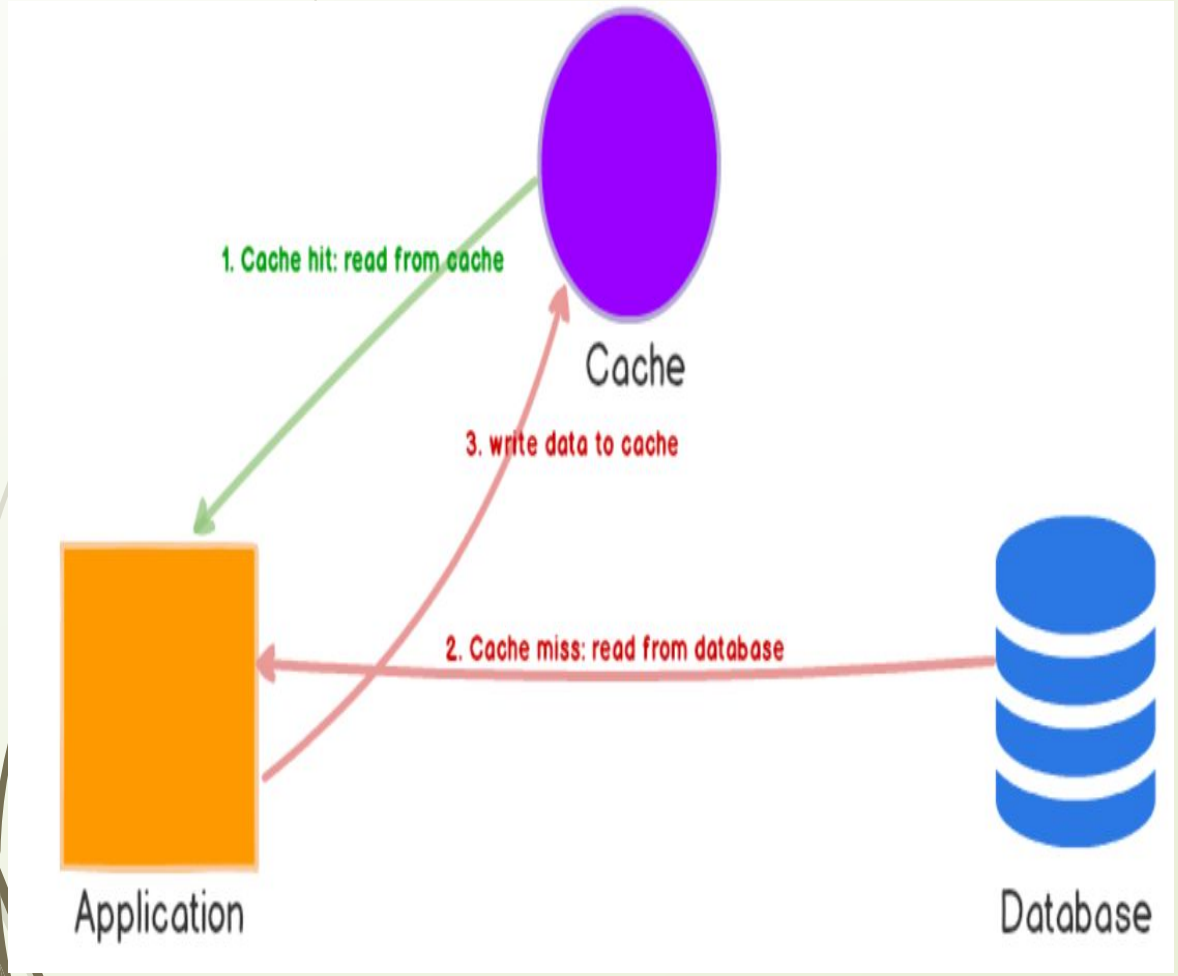
- Scalability, High Availability, Fault tolerance





Caching Strategies

- Cache-Aside (Lazy Loading)



When to use this pattern

Use this pattern when:

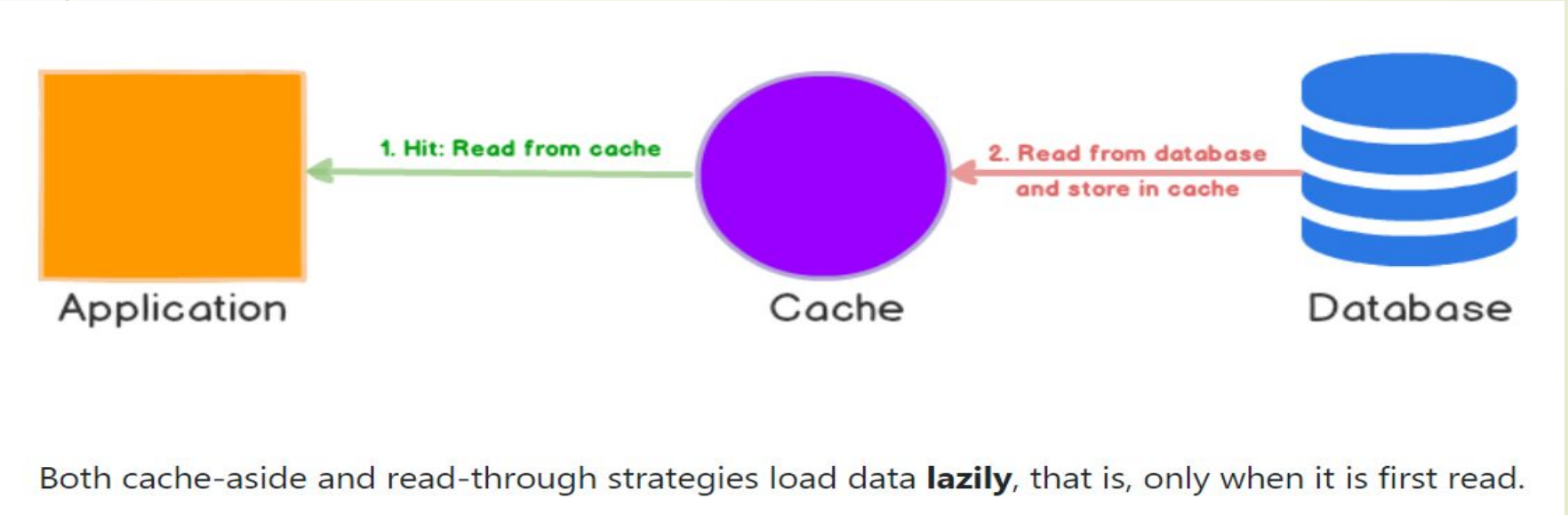
- A cache doesn't provide native read-through and write-through operations.
- Resource demand is unpredictable. This pattern enables applications to load data on demand. It makes no assumptions about which data an application will require in advance.

This pattern might not be suitable:

- When the cached data set is static. If the data will fit into the available cache space, prime the cache with the data on startup and apply a policy that prevents the data from expiring.
- For caching session state information in a web application hosted in a web farm. In this environment, you should avoid introducing dependencies based on client-server affinity.

Caching Strategies (Contd..)

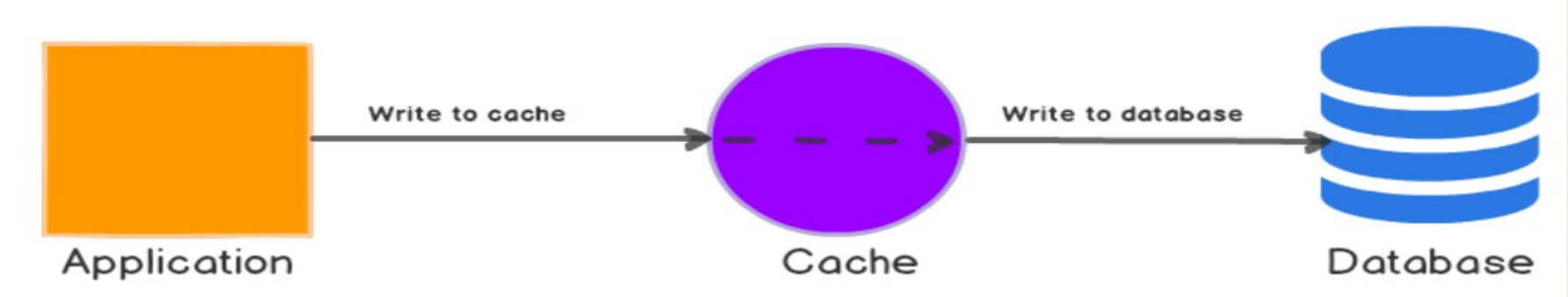
- Read-Through Cache
  - Read-through cache sits in-line with the database. When there is a cache miss, it loads missing data from database, populates the cache and returns it to the application.



  - Read-through caches work best for **read-heavy** workloads when the same data is requested many times. For example, a news story.
  - The disadvantage is that when the data is requested the first time, it always results in cache miss and incurs the extra penalty of loading data to the cache.

Caching Strategies (Contd..)

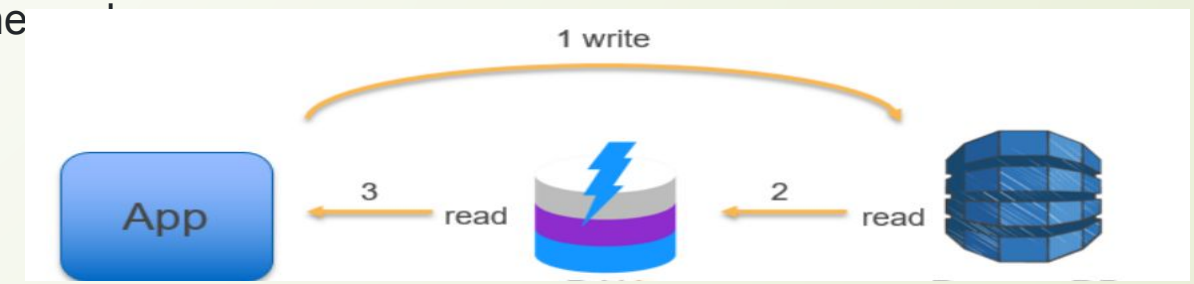
- **Write-Through Cache**
  - In this write strategy, data is first written to the cache and then to the database.
  - The cache sits in-line with the database and writes always go *through* the cache to the main database. This helps cache maintain consistency with the main database.



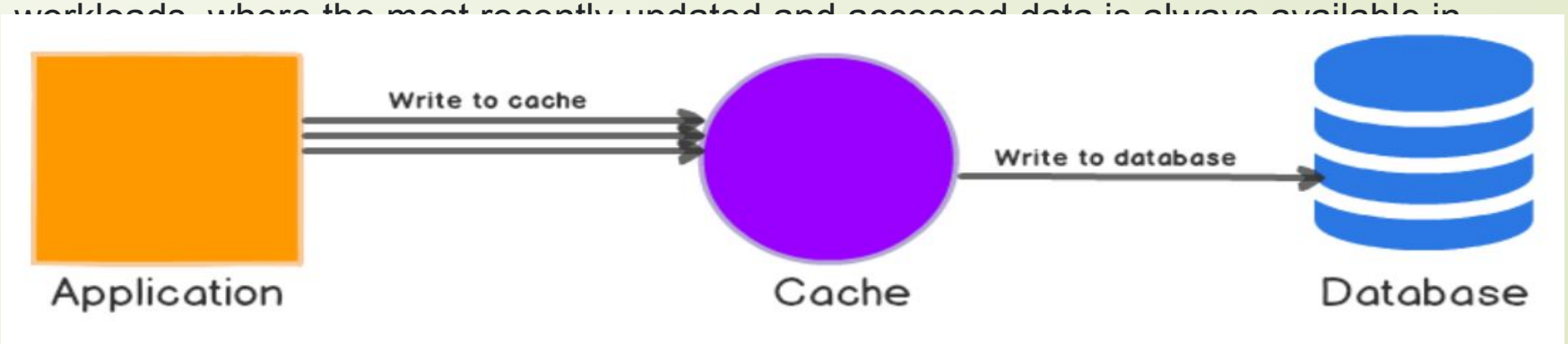
  - On its own, write-through caches don't seem to do much, in fact, they introduce extra write **latency** because data is written to the cache first and then to the main database (two write operations.)
  - But when paired with read-through caches, we get all the benefits of read-through and we also get data **consistency** guarantee, freeing us from using cache invalidation (assuming ALL writes to the database go through the cache.)
  - DynamoDB Accelerator (DAX) is a good example of read-through / write-through cache.

Caching Strategies (Contd..)

- **Write-Around Cache**
  - Here, data is written directly to the database and only the data that is read makes it way into the

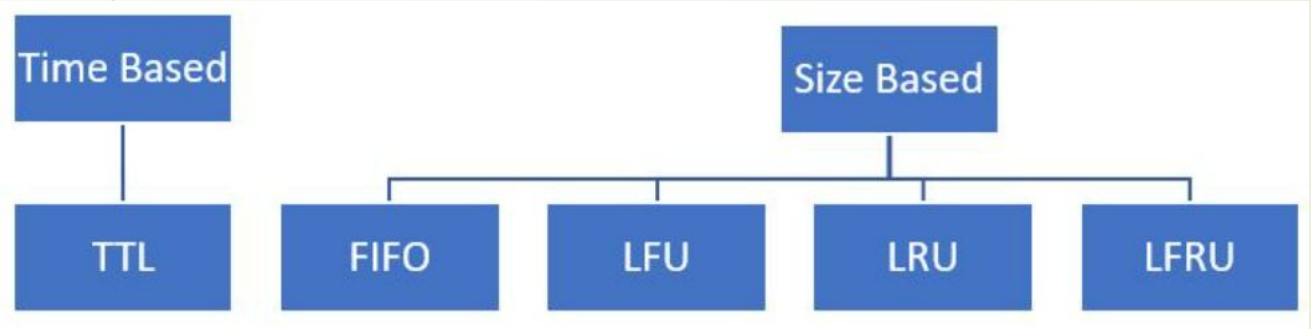


- **Write-Back Cache**
  - Here, the data is updated only in the cache and updated into the database at a later time.
  - Write back caches improve the write performance and are good for **write-heavy** workloads. When combined with read-through, it works good for mixed



Cache Eviction Strategies

- We need to regularly expel data from the cache to limit the size of the cache and to maintain its speed while ensuring that the entries are up to date.



- Time Based
  - We keep an entry in the cache for some amount of time. In this strategy, we set a TTL (Time To Live).
  - We will evict the entry from the cache after a certain pre-determined time has elapsed.
  - The time which an entry stays in the cache before being evicted is called TTL.

Cache Eviction Strategies(Contd..)

- Size Based : We keep at most some number of entries in the cache.
  - **FIFO (First In First Out)**: We harness the FIFO property of the queue data structure to evict old entries.
  - **LFU (Least Frequently Used)**: When we must evict an entry, we will evict the least frequently used entry.
  - **LRU (Least Recently Used)**: When we must evict an entry, we will evict the least recently used entry.
  - **LFRU (Least Frequently and Recently Used)**: We evict the least valuable entry, the one that's neither used frequently nor recently. This strategy gives the best results.

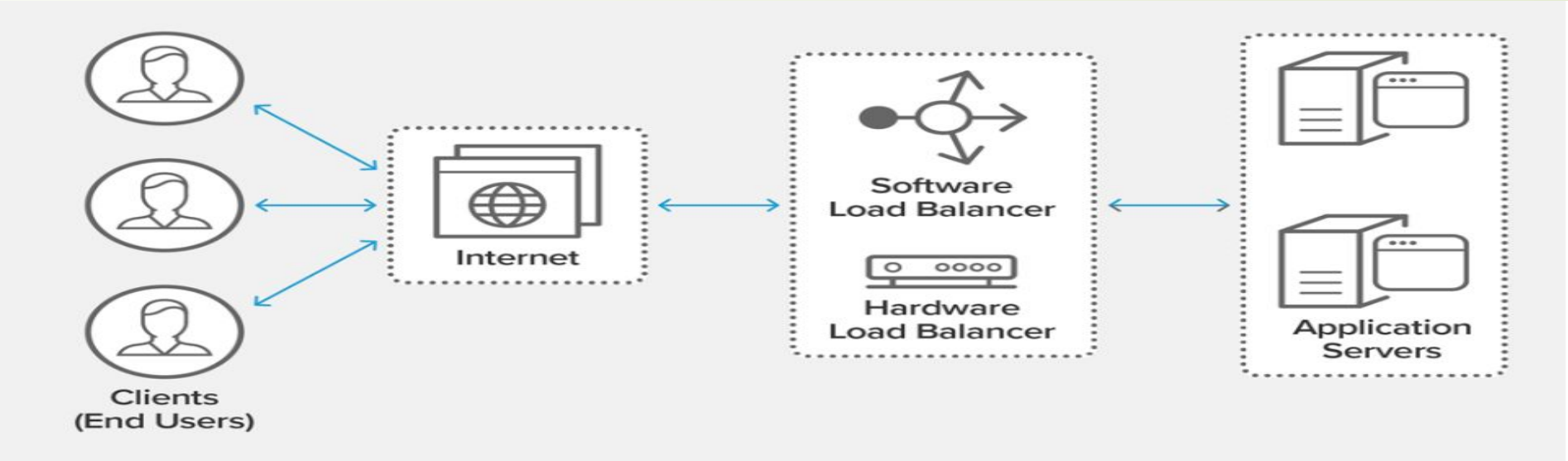
Out of the Box Solutions

It's usually not recommended to write your own cache implementation, because there are a lot of really good cache implementations out there that you can use. Some of the most popular caching products that are available in the market are:

- Ehcache
- Hazelcast
- Memcached
- Redis

Load Balancer

- A load balancer performs the following functions:
  - Distributes client requests or network load efficiently across multiple servers
  - Ensures high availability and reliability by sending requests only to servers that are online



The distinction between "hardware" and "software" load balancers is no longer meaningful.
A so-called "hardware" load balancer is a PC class CPU, network interfaces with packet processing capabilities, and some software to bind it all together. A "software" load balancer realized on a good server with modern NICs is ... the same.

Where do we add a load balancer ?

Between the client and frontend web servers: This is often the first point of contact between the client and the system. The load balancer receives incoming requests from clients and distributes them across the frontend web servers.

Between frontend web servers and backend application servers: In a system with multiple frontend web servers, a load balancer can be used to distribute incoming requests from the web servers to the backend application servers.

Between backend application servers and cache servers: Load balancers can be used to distribute requests from the application servers to cache servers, which store frequently accessed data in memory to reduce response times.

Between cache servers and database servers: In systems with multiple cache servers, a load balancer can be used to distribute requests from the cache servers to the database servers, which store the actual data. This helps to ensure that the database servers are not overwhelmed with requests.

Load Balancer and Session Stickiness

- Session stickiness, or session persistence, is a mechanism by which load balancers can couple the requests to the backend systems. This ensures that different requests for the same session can be processed by different servers without loss of information.

Load Balancing Algorithms

- Different load balancing algorithms provide different benefits; the choice of load balancing method depends on your needs:
  - **Round Robin** – Requests are distributed across the group of servers sequentially.
  - **Least Connections** – A new request is sent to the server with the fewest current connections to clients. The relative computing capacity of each server is factored into determining which one has the least connections.
  - **Least Time** – Sends requests to the server selected by a formula that combines the fastest response time and fewest active connections.
  - **Hash** – Distributes requests based on a key you define, such as the client IP address or the request URL. NGINX Plus can optionally apply a consistent hash to minimize redistribution
    of loads if the set of upstream servers changes.
  - **IP Hash** – The IP address of the client is used to determine which server receives the request.
  - **Random with Two Choices** – Picks two servers at random and sends the request to the one that is selected by then applying the Least Connections algorithm.

Various forms of Load Balancing

- **Cloud load balancing** refers to distributing client requests across multiple application servers that are running in a cloud environment.
- **DNS load balancing** is the practice of configuring a domain in the Domain Name System (DNS) such that client requests to the domain are distributed across a group of server machines.
- **Global server load balancing** (GSLB) refers to the intelligent distribution of traffic across server resources located in multiple geographies.
- **Hybrid load balancing** refers to distributing client requests across a set of server applications that are running in various environments: on premises, in a private cloud, and in the public cloud.
- **Layer 4 load balancing** uses information defined at the networking *transport* layer (Layer 4) as the basis for deciding how to distribute client requests across a group of servers.
- **Layer 7 load balancing** operates at the high-level *application* layer, which deals with the actual content of each message.

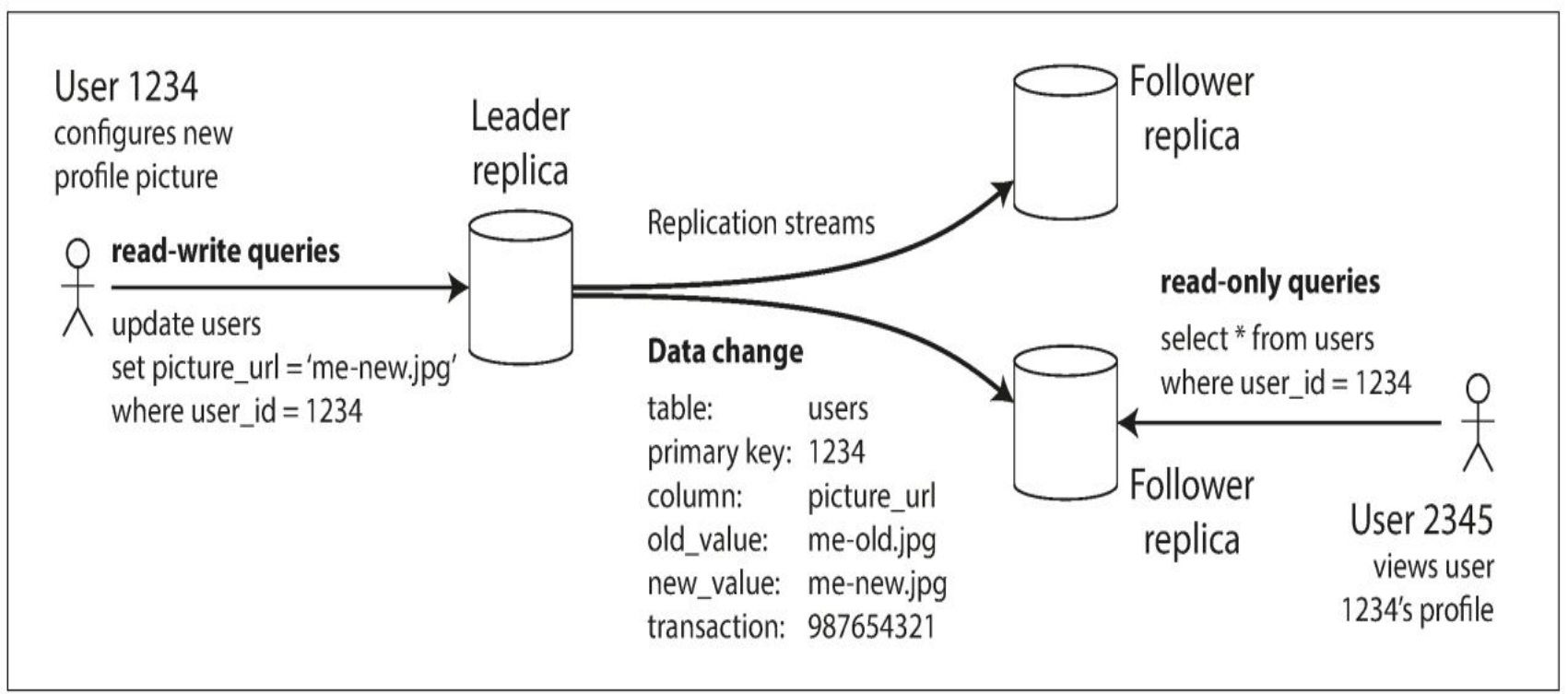
Replication

- **Replication** means keeping a copy of the same data on multiple machines that are connected via a network.
- There are several reasons why you might want to replicate data: •
  - To keep data geographically close to your users (and thus **reduce latency**) •
  - To allow the system to continue working even if some of its parts have failed (and thus **increase availability**) •
  - To scale out the number of machines that can serve read queries (and thus **increase read throughput**)
- If the data that you're replicating does not change over time, then replication is easy: you just need to copy the data to every node once, and you're done.
- All of the **difficulty in replication lies in handling changes to replicated data**, and that's what we'll cover in the next few slides.
- We will discuss **three popular algorithms** for replicating changes between nodes: **single-leader, multi-leader, and leaderless replication.**

Single Leader Replication

- **One of the replicas is designated the leader (also known as master or primary).** When clients want to write to the database, they must send their requests to the leader, **which first writes the new data to its local storage**.
- **The other replicas are known as followers (read replicas, slaves, secondaries, or hot standbys).**
  - **Whenever the leader writes new data to its local storage, it also sends the data change to all of its followers as part of a replication log or change stream. Each follower takes the log from the leader and updates its local copy of the database accordingly, by applying all writes in the same order as they were processed on the leader.**
- **When a client wants to read from the database**, it can query either the leader or any of the followers.
- **However, writes are only accepted on the leader** (the followers are read-only from the client's point of view).

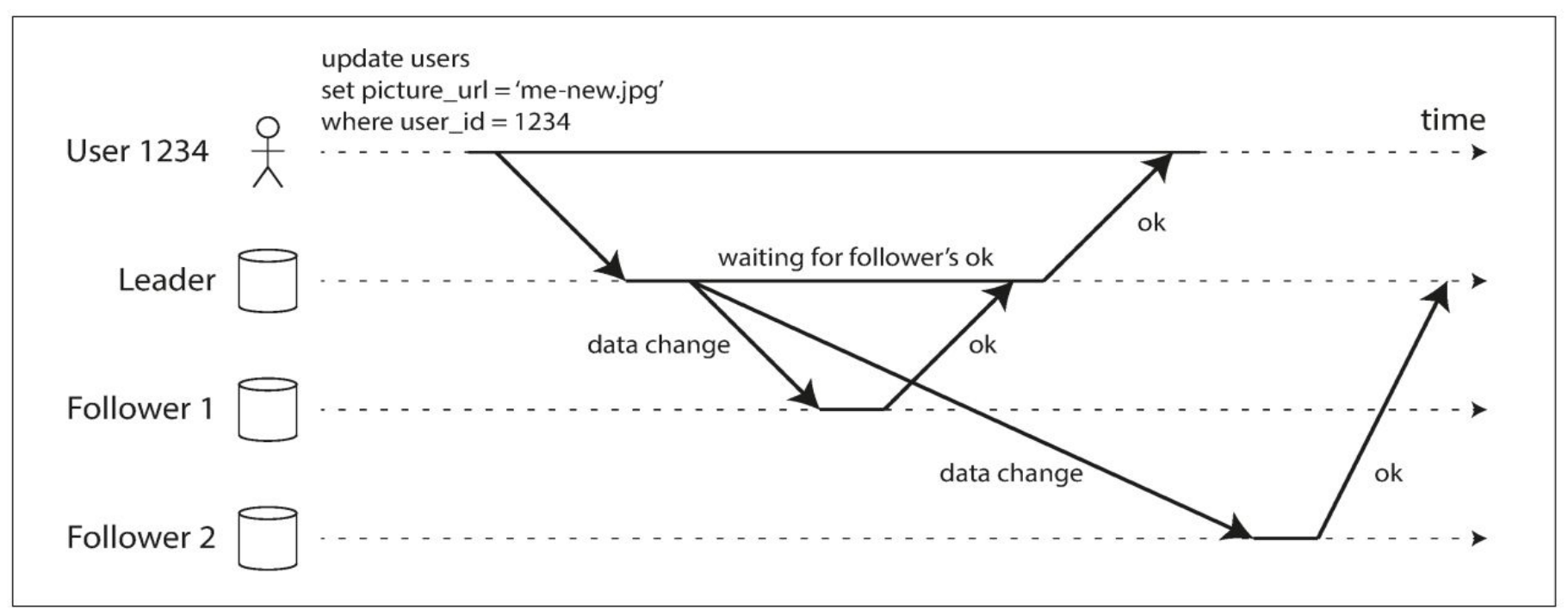
Single Leader Replication



- This mode of replication is a built-in feature of many **relational databases**(PostgreSQL, MySQL), **non-relational databases**(MongoDB) and **distributed message brokers**(Kafka, RabbitMQ).

Synchronous Vs Asynchronous Replication

- An important detail of a replicated system is whether the replication happens synchronously or asynchronously. In the below figure, **time flows from left to right. A request or response message is shown as a thick arrow**.



The **replication to follower 1 is synchronous** while **the replication to follower 2 is asynchronous**.

Sync Vs Async Replication(Contd..)

- The **advantage of synchronous replication** is that the **follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader**. If the leader suddenly fails, we can be sure that the data is still available on the follower.
- **The disadvantage is that if the synchronous follower doesn't respond (because it has crashed, or there is a network fault, or for any other reason), the write cannot be processed**. The leader must block all writes and wait until the synchronous replica is available again.
- **For that reason, it is impractical for all followers to be synchronous**: any one node outage would cause the whole system to grind to a halt. **In practice, if you enable synchronous replication on a database, it usually means that one of the followers is synchronous, and the others are asynchronous.**
- **If the synchronous follower becomes unavailable or slow, one of the asynchronous followers is made synchronous.** This guarantees that you have an up-to-date copy of the data on at least two nodes: the leader and one synchronous follower.

Setting Up New Followers

- **Simply copying data files from one node to another is typically not sufficient**: clients are constantly writing to the database, and the data is always in flux, so a standard file copy would see different parts of the database at different points in time. The result might not make any sense.
- **You could make the files on disk consistent by locking the database (making it unavailable for writes), but that would go against our goal of high availability**.
- Fortunately, setting up a follower can usually be done without downtime. Conceptually, the process looks like this:
  - Take a consistent snapshot of the leader's database at some point in time
  - Copy the snapshot to the new follower node
  - The follower connects to the leader and requests all the data changes that have happened since the snapshot was taken.
  - When the follower has processed the backlog of data changes since the snapshot, we say it has caught up. It can now continue to process data changes from the leader as they happen.

Follower failure: Catch-up recovery

- On its local disk, **each follower keeps a log of the data changes it has received from the leader**.
- **If a follower crashes and is restarted**, or if the network between the leader and the follower is temporarily interrupted, the follower can recover quite easily: from its log, **it knows the last transaction that was processed before the fault occurred.**
- **Thus, the follower can connect to the leader and request all the data changes that occurred during the time when the follower was disconnected.**
- **When it has applied these changes, it has caught up to the leader and can continue receiving a stream of data changes as before.**

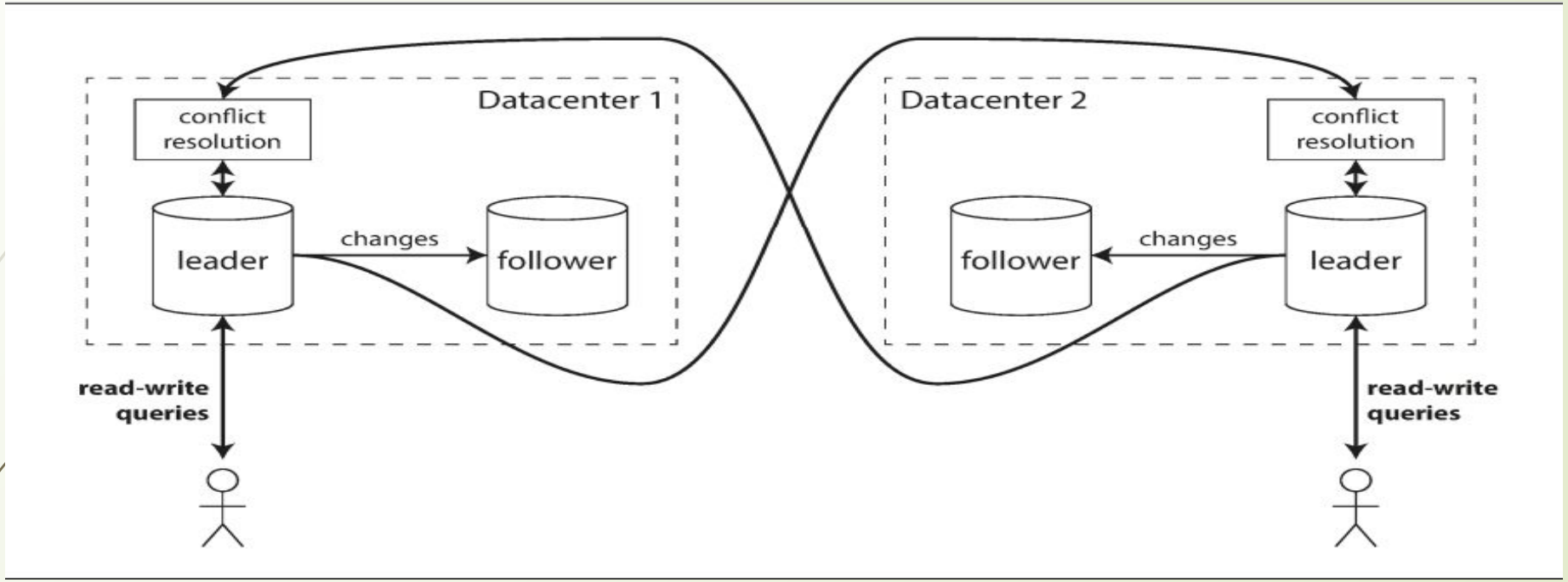
Leader failure: Failover

- Handling a failure of the leader is trickier**: one of the followers needs to be promoted to be the new leader**, **clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader**. **This process is called failover.**
- An automatic failover process usually consists of the following steps:
  - Determining that the leader has failed
  - Choosing a new leader
  - Reconfiguring the system to use the new leader
- Failover is fraught with things that can go wrong:
  - If asynchronous replication is used, the new leader may not have received all the writes from the old leader before it failed.
  - In certain fault scenarios, it could happen that two nodes both believe that they are the leader. This situation is called split brain, and it is dangerous: if both leaders accept writes, and there is no process for resolving conflicts, data is likely to be lost or corrupted.

Multi-Leader Replication

- Single leader-based replication has one **major downside: there is only one leader, and all writes must go through it**.
  - If you can't connect to the leader for any reason, for example due to a network interruption between you and the leader, you can't write to the database.
- **A natural extension of the leader-based replication model is to allow more than one node to accept writes.**
- Replication still happens in the same way**: each node that processes a write must forward that data change to all the other nodes.**
- In this setup**, each leader simultaneously acts as a follower to the other leaders.**
- Use Cases for Multi-Leader Replication
  - It rarely makes sense to use a multi-leader setup within a single datacenter, because the benefits rarely outweigh the added complexity.
  - However, there are some situations in which this configuration is reasonable.

Multi-leader replication across multiple datacenters



- Within each datacenter, regular leader– follower replication is used;
- Between datacenters, each datacenter's leader replicates its changes to the leaders in other datacenters.
- The biggest problem with multi-leader replication is that write conflicts can occur, which means that conflict resolution is required.

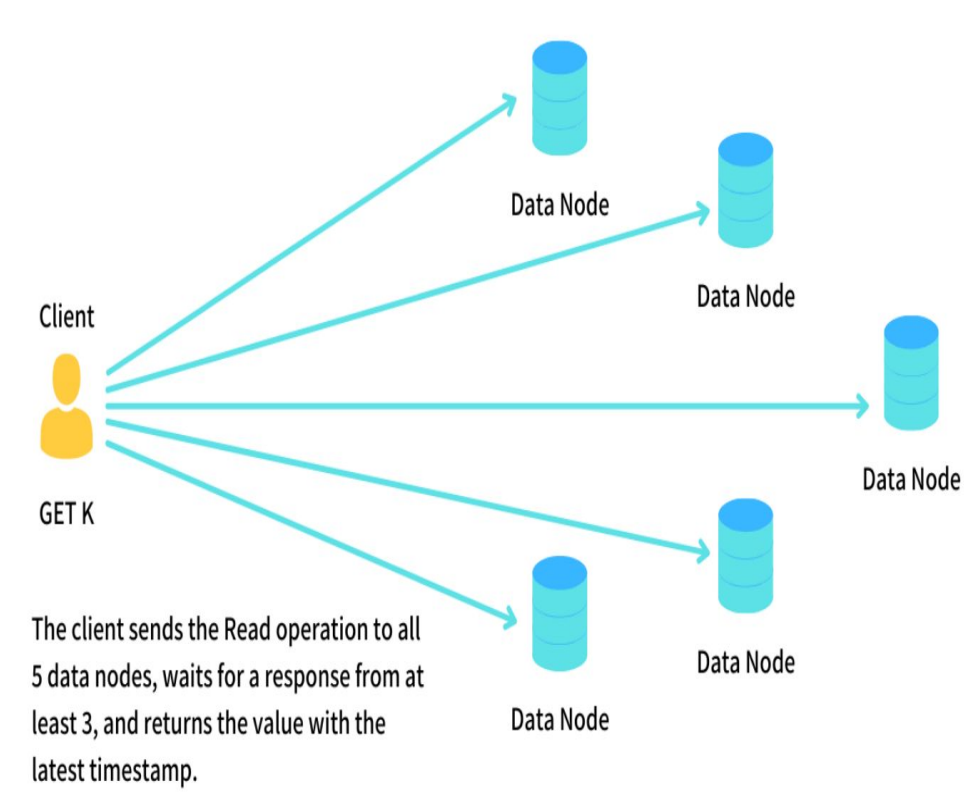
Leaderless Replication

- Some data storage systems take a different approach, abandoning the concept of a leader and allowing any replica to directly accept writes from clients.
- In some leaderless implementations, the client directly sends its writes to several replicas, while in others, a coordinator node does this on behalf of the client.
  - However, unlike a leader database, that coordinator does not enforce a particular ordering of writes.
- It leverages quorum to ensure strong consistency across multiple nodes and good tolerance to failures.

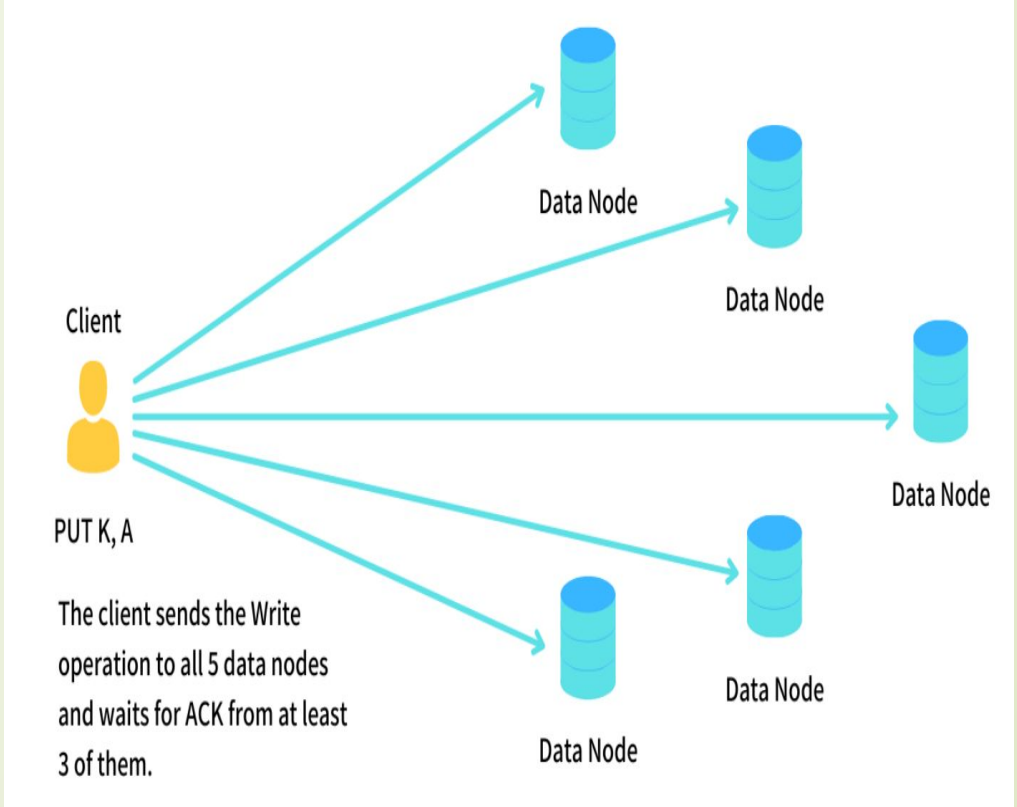
- `w` : number of nodes that confirm the writes with an `ACK`
- `r` : number of nodes we query for the read
- `n` : total number of nodes

to have a strong consistency i.e. we can expect to get the latest value of any record so long as `w + r > n`, because with this there will be at least one node that has the latest value that will return it. Such reads and writes are called Quorum Reads and Writes.

Leaderless Replication Read & Write Example



Read Operation



Write Operation

Sharding

- **Sharding** is the practice of optimizing database management systems by separating the rows or columns of a larger database table into multiple smaller tables.
- The new tables are called “shards” (or partitions), and each new table either has the **same schema but unique rows** (as is the case for “horizontal sharding”) or has a schema that is **a proper subset of the original table’s schema** (as is the case for “vertical sharding”).

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Vertical Shards

VS1

CUSTOMER ID	FIRST NAME	LAST NAME
1	Alice	Anderson
2	Bob	Best
3	Carrie	Conway
4	David	Doe

VS2

CUSTOMER ID	CITY
1	Austin
2	Boston
3	Chicago
4	Denver

Horizontal Shards

HS1

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston

HS2

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Why Is Sharding Used?

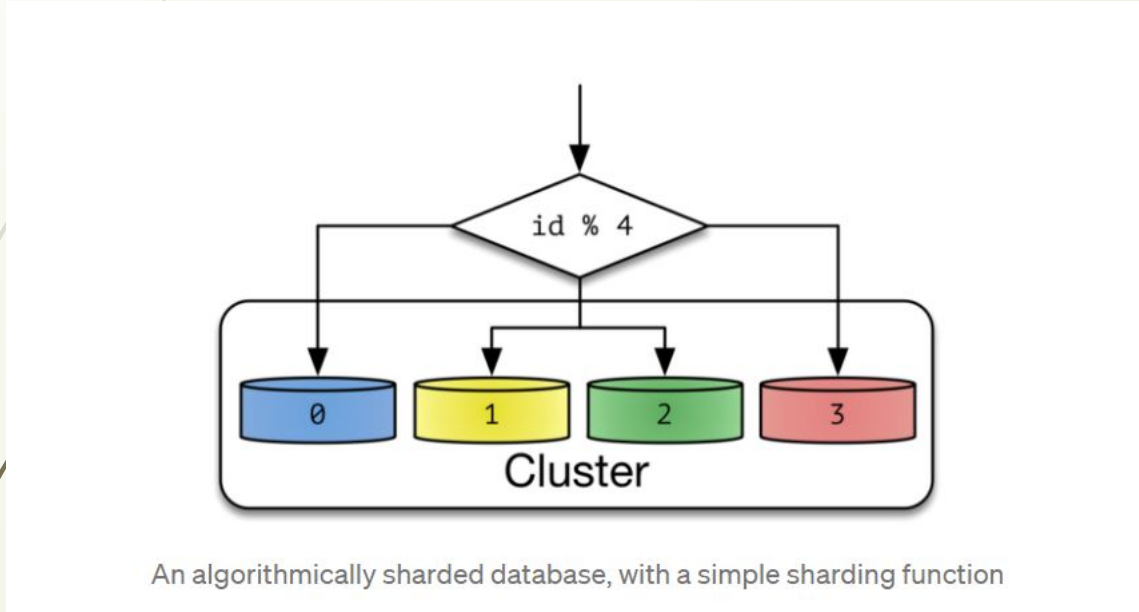
- By sharding a larger table, you can store the new chunks of data, called **logical shards**, across multiple nodes to achieve horizontal scalability and improved performance. Once the logical shard is stored on another node, it is referred to as a **physical shard**.
- When running a database on a single machine, you will eventually reach the limit of the amount of computing resources you can apply to any queries, and you will obviously reach a maximum amount of data with which you can efficiently work. By horizontally scaling out, you can **enable a flexible database design** that increases performance in two key ways:
  - With massively parallel processing, you can take advantage of all the compute resources across your cluster for every query.
  - Because the individual shards are smaller than the logical table as a whole, each machine has to scan fewer rows when responding to a query.
- Also, sharded databases can **offer higher levels of availability**. In the event of an outage on an unsharded database, the entire application is unusable. With a sharded database, only the portions of the application that relied on the missing chunks of data are unusable.

How sharding works ?

- **Partition Key**
  - is a portion of primary key which determines how data should be distributed.
  - allows you to retrieve and modify data efficiently by routing operations to the correct database.
  - Entries with the same partition key are stored in the same node.
- To compare the pros and cons of various sharding strategies, following principles can be used:
  - **How the data is read** — Databases are used to store and retrieve data. If we don't need to read data at all, we can simply write it to */dev/null*. If we only need to batch process the data once in a while, we can append to a single file and periodically scan through them. Data retrieval requirements (or lack thereof) heavily influence the sharding strategy.
  - **How the data is distributed** — Once you have a cluster of machines acting together, it is important to ensure that data and work is evenly distributed. Uneven load causes storage and performance hotspots. Some databases redistribute data dynamically, while others expect clients to evenly distribute and access data.
  - **How the data is redistributed -** ensuring both the data and locators are in sync.

Sharding Strategies

- Algorithmic Sharding



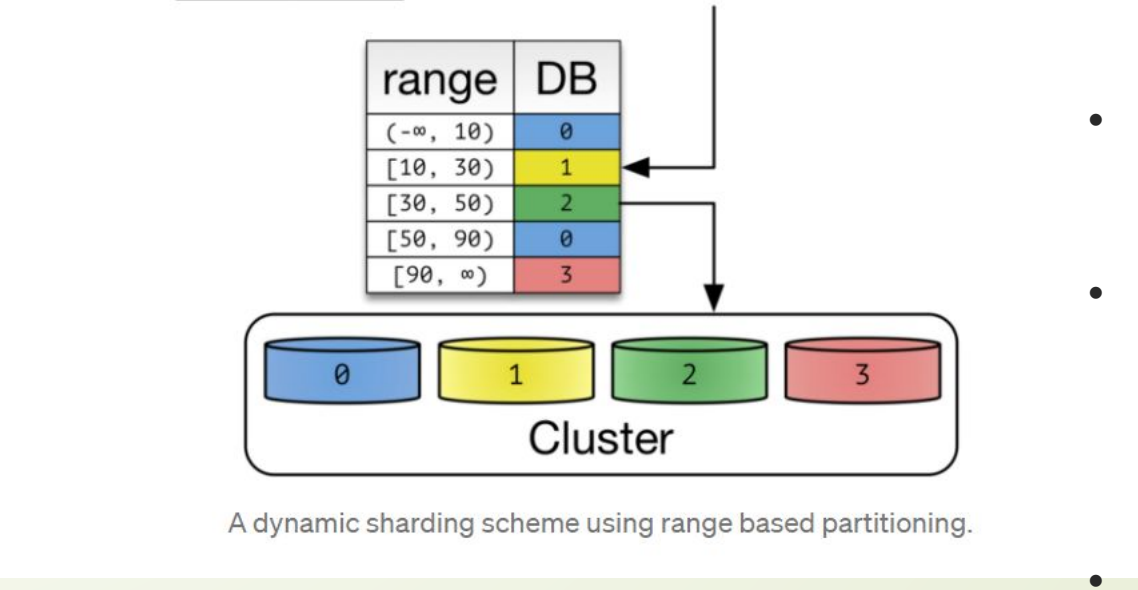
An algorithmically sharded database, with a simple sharding function

- suitable for key-value databases with homogeneous values so that there is uniform data distribution and all partitions should be of similar sizes.
- Resharding data can be challenging. It requires updating the sharding function and moving data around the cluster. Doing both at the same time while maintaining consistency and availability is hard. [**Solution : Consistent Hashing**]

Examples of such system include **Memcached**. Memcached is not sharded on its own, but expects client libraries to distribute data within a cluster. Such logic is fairly easy to implement at the application level.

Sharding Strategies (Contd..)

- Dynamic Sharding



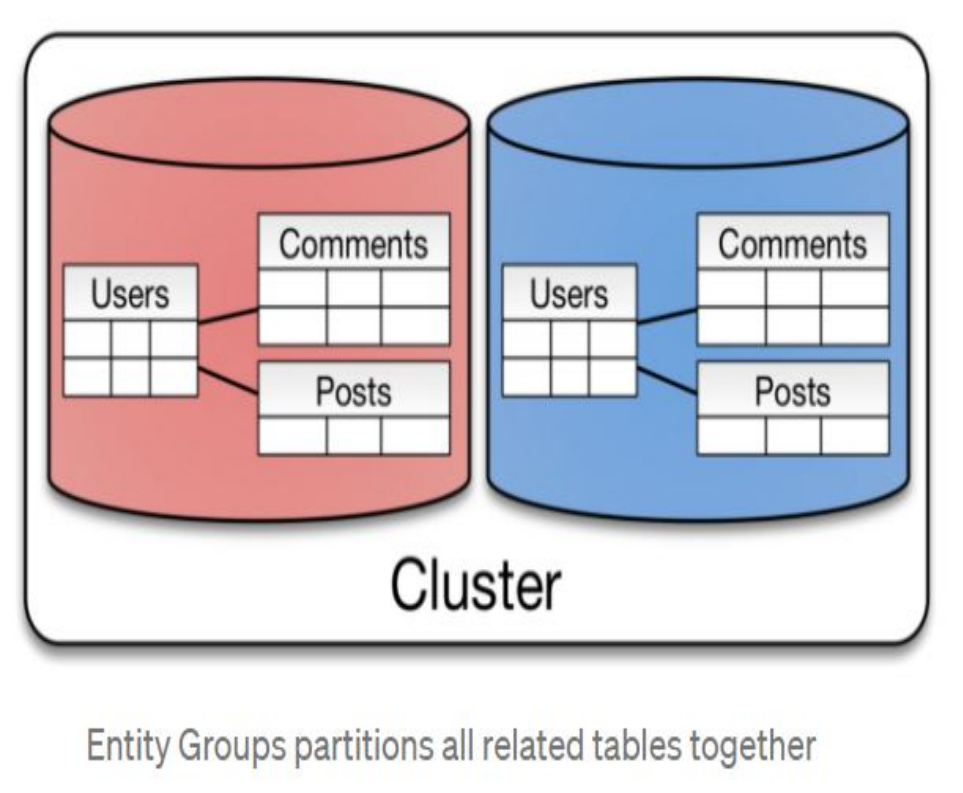
A dynamic sharding scheme using range based partitioning.

E.g. **HDFS**, **Apache Hbase**, **MongoDB**

- In dynamic sharding, an external **locator service** determines the location of entries.
- To read and write data, clients need to consult the locator service first.
- Dynamic sharding is more resilient to nonuniform distribution of data. Locators can be created, split, and reassigned to redistribute data.
- The locator service becomes a single point of contention and failure.
- Consensus algorithms and synchronous replications are used to store locator service data.

Sharding Strategies (Contd..)

- Entity Groups



- Store related entities in the same partition to provide additional capabilities within a single partition. Specifically:
  1. Queries within a single physical shard are efficient.
  2. Stronger consistency semantics can be achieved within a shard.
- Entity groups can be implemented either algorithmically or dynamically.
- They are usually implemented dynamically since the total size per group can vary greatly. The same caveats for updating locators and moving data around applies here. Instead of individual tables, an entire entity group needs to be moved together.

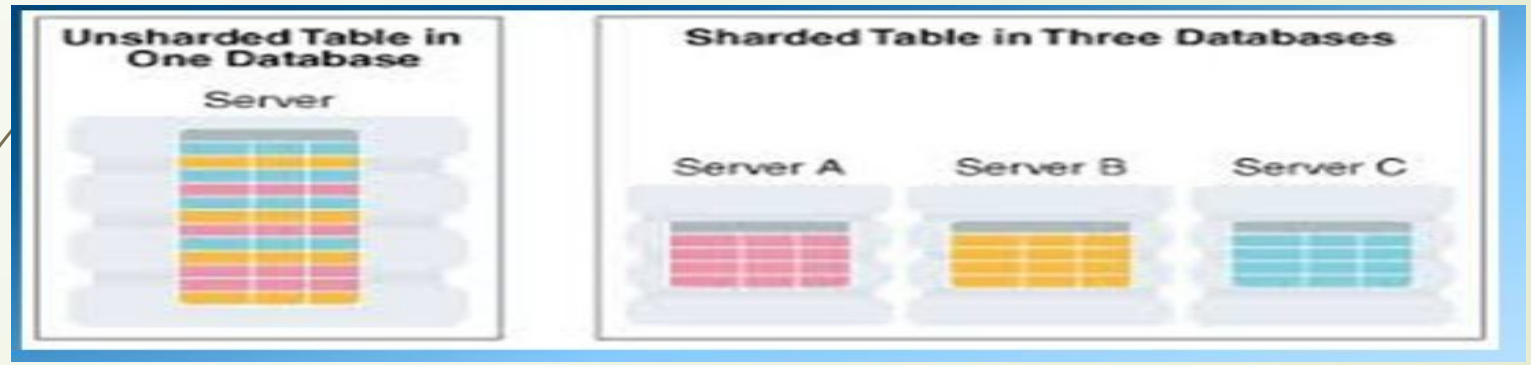
Other than sharded RDBMS solutions, **Google Megastore** is an example of such a system. Megastore is publicly exposed via Google App Engine's Datastore API.

Understanding the pitfalls of Sharding strategies

- A logical shard (data sharing the same partition key) must fit in a single node.
  - **This is the most important assumption, and is the hardest to change in future**.
  - A logical shard is an atomic unit of storage and cannot span across multiple nodes.
  - In such a situation, the database cluster is effectively out of space.
  - Having finer partitions mitigates this problem, but it adds complexity to both database and application. The cluster needs to manage additional partitions and the application may issue additional cross-partition operations.
- Even though dynamic sharding is more resilient to unbalanced data, an unexpected workload can reduce its effectiveness.
- Many web applications shard data by user. This may become problematic over time, as the application accumulates power users with a large amount of data.
- **Cross-shard queries** are inefficient, and many sharding schemes attempt to minimize the number of cross-partition operations. On the other hand, partitions need to be granular enough to evenly distribute the load amongst nodes. Finding the right balance can be tricky.

What Is the Difference between Sharding and Partitioning?

- Sharding and partitioning are both about breaking up a large data set into smaller subsets.
- **The difference is that sharding implies the data is spread across multiple computers while partitioning does not.**



- Partitioning is about grouping subsets of data within a single database instance.
- **In many cases, the terms sharding and partitioning are even used synonymously**, especially when preceded by the terms “horizontal” and “vertical.” Thus, “horizontal sharding” and “horizontal partitioning” can mean the same thing.

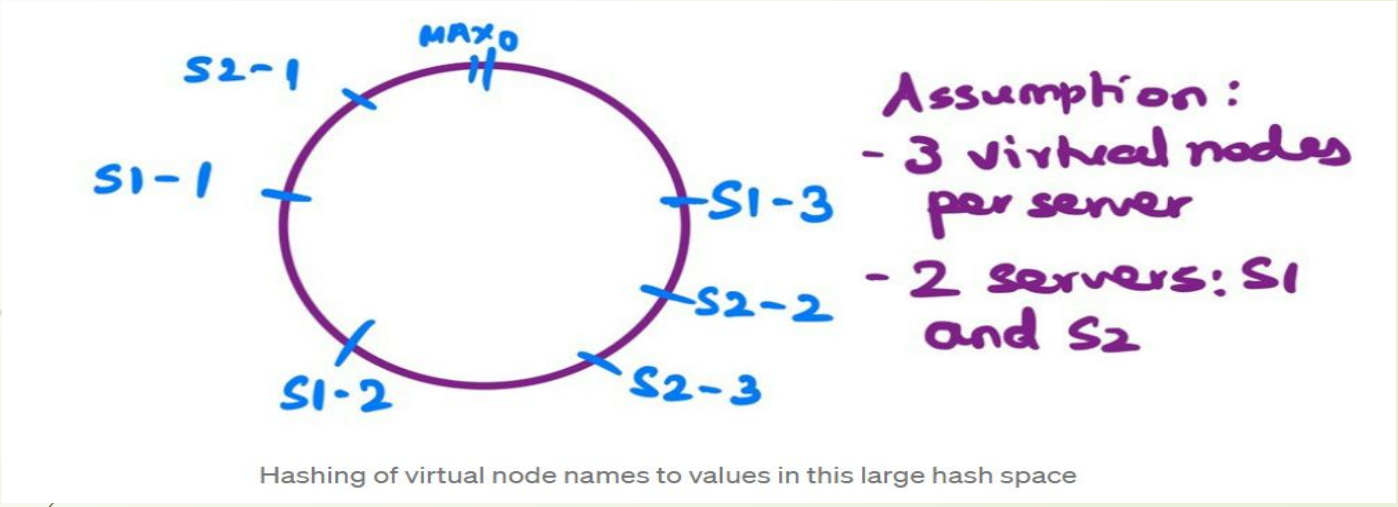
Partitioning Strategy

- Can't we just divide our data into n parts and assign each part to one node and then round robin through them while inserting data?
  - NO because this approach has following drawbacks:
    - We don't know which particular piece of data went to which node.
    - The partitioning might be skewed creating few highly loaded partitions(hotspots) which will act as bottleneck for the system and compromising the effectiveness of partitioning.
- We need a smart way to distribute our data which ensures equitable sharing of load and also provides quick lookup capability. (In most distributed systems, every piece of data is recognized by a unique identifier, usually an *id* or a *key.*)
- **Key Range partitioning:** In this form of partitioning, we divide the entire keyset into continuous ranges and assign each range to a partition.
  - For example, suppose, we are storing data related to the animal species. We can assign all species starting with letter A-D to partition 1, E-G to partition 2 and so on.
  - So when we query data regarding the species "Panthera tigris", we know it will be in the partition which contains data for the species starting with letter 'P'.

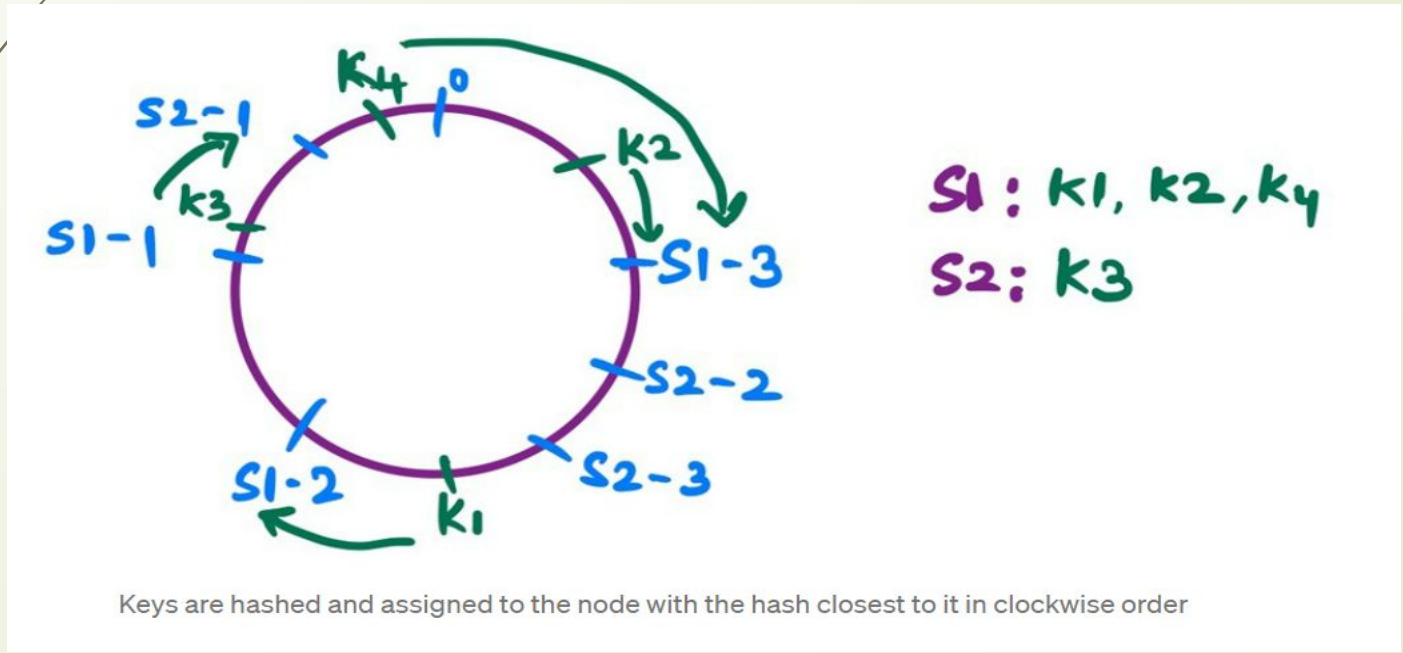
Partitioning Strategy(Contd..)

- **Key Hash Partitioning**
  - The general idea here is to apply a hash function on the key which results in a hash value, and then mod it with the number of partitions.
  - The same key will always return the same hash code, so once you've figured out how you spread out a range of keys across the nodes available, you can always find the right partition by looking at the hash code for a key.
  - This will give you the partition number with a fair amount of randomness. Thus, the problem of forming hotspots is solved.
  - The disadvantage is that you lose the order of keys.
  - However, the real problem with hash partitioning starts when you change the number of partitions.
  - Since we are doing a mod of *n(number of partitions)* , all our previous data will now get assigned to partitions different from there existing partitions.
  - Changing the number of partitions is fairly common in distributed systems, and moving data of such scale can prove to be a nightmare.
  - But, there is a solution for this: **consistent hashing**.

Consistent Hashing

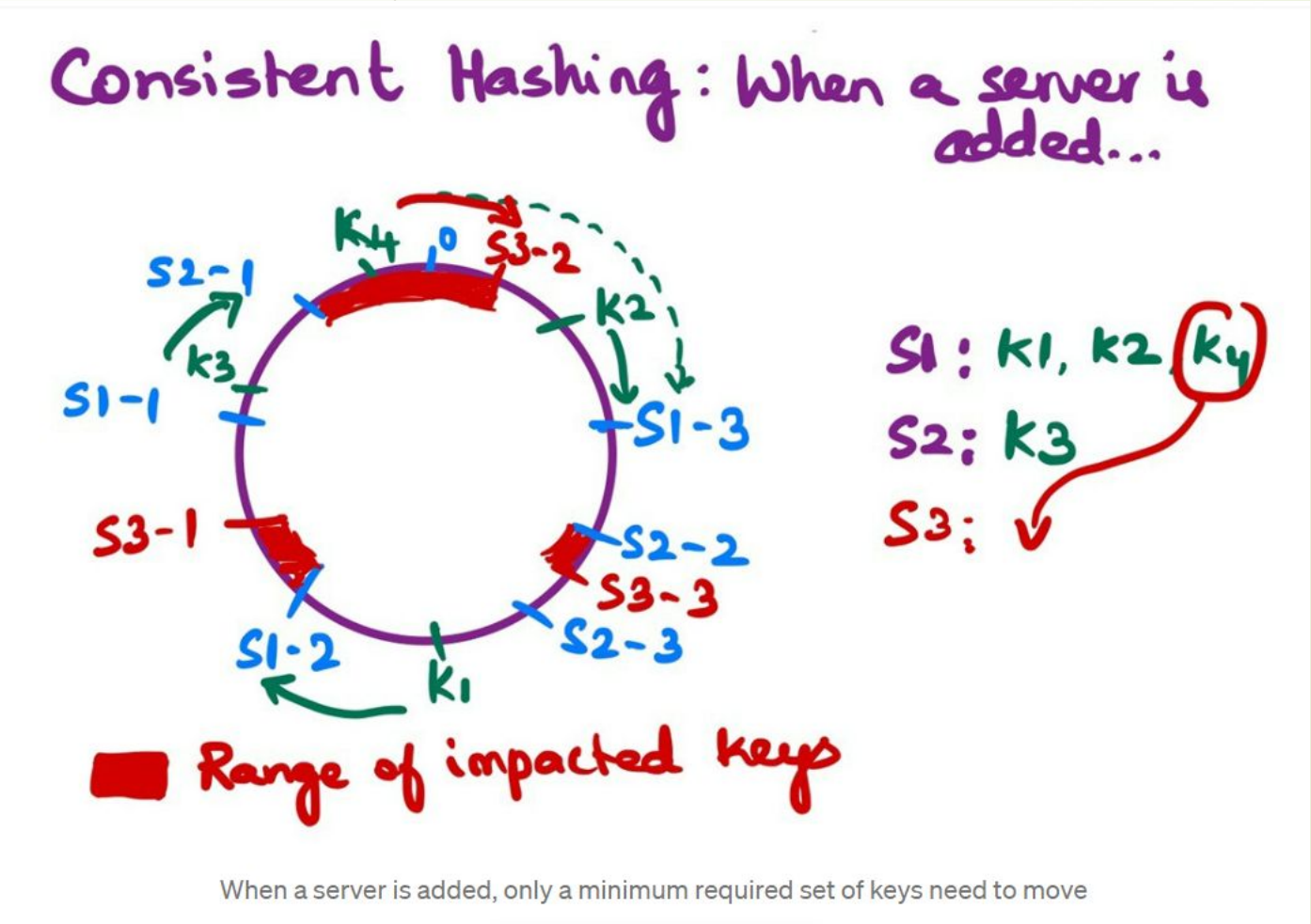


Hashing of virtual node names to values in this large hash space



Keys are hashed and assigned to the node with the hash closest to it in clockwise order

Consistent Hashing (Contd..)



When a server is added, only a minimum required set of keys need to move

- The main benefit of Consistent Hashing is that any server can be added or removed dynamically and only the minimal set of objects need to move.
- On average, it requires only k/n objects to move where k is the total number of keys and n is the total number of nodes.
- This property is known as the *monotonicity*: when a server is added, objects only move from old servers to the new server; there is no unnecessary movement *between* the old servers.

Microservices

Microservices are small, autonomous services that work together:

- Small, and Focused on Doing One Thing Well (Single Responsibility Principle)
  - We focus our service boundaries on business boundaries, making it obvious where code lives for a given piece of functionality.
  - And by keeping this service focused on an explicit boundary, we avoid the temptation for it to grow too large, with all the associated difficulties that this can introduce.
  - A strong factor in helping us answer how small? is how well the service aligns to team structures. If the codebase is too big to be managed by a small team, looking to break it down is very sensible.
  - As you get smaller, the benefits around interdependence increase. But so too does some of the complexity that emerges from having more and more moving parts.
- Autonomous
  - A microservice is a separate entity. It might be deployed as an isolated service.
  - All communication between the services themselves are via network calls, to enforce separation between the services and avoid the perils of tight coupling.
  - These services need to be able to change independently of each other, and be deployed by themselves without requiring consumers to change.
  - Our service exposes an application programming interface (API), and collaborating services communicate with us via those APIs.
  - The golden rule: can you make a change to a service and deploy it by itself without changing anything else ?

Key Benefits of Microservices

- Technology Heterogeneity
  - With a system composed of multiple, collaborating services, we can decide to use different technologies inside each one. This allows us to pick the right tool for each job.

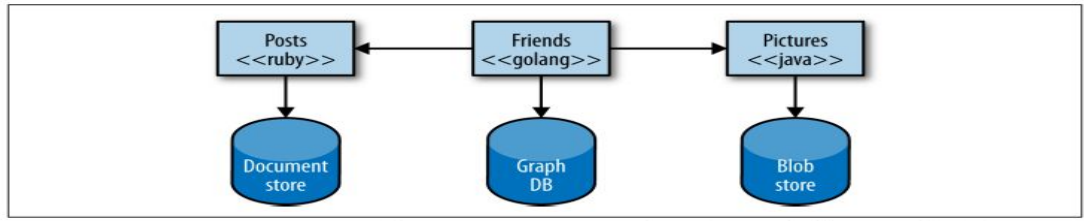


Figure 1-1. Microservices can allow you to more easily embrace different technologies

With a monolithic application, if I want to try a new programming language, database, or framework, any change will impact a large amount of my system.

  - For example, for a social network, we might store our users' interactions in a graph-oriented database to reflect the highly interconnected nature of a social graph, but perhaps the posts the users make could be stored in a document-oriented data store, giving rise to a heterogeneous architecture like above.
- Resilience
  - If one component of a system fails, but that failure doesn't cascade, you can isolate the problem and the rest of the system can carry on working.
  - With a monolithic system, we can run on multiple machines to reduce our chance of failure, but with microservices, we can build systems that handle the total failure of services and degrade functionality accordingly.
  - To ensure our microservice systems can properly embrace this improved resilience, we need to understand the new sources of failure that distributed systems have to deal with. Networks can and will fail, as will machines. We need to know how to handle this, and what impact (if any) it should have on the end user of our software.

Key Benefits of Microservices(Contd..)

- Scaling
  - With a large, monolithic service, we have to scale everything together. With smaller services, we can just scale those services that need scaling, allowing us to run other parts of the system on smaller, less powerful hardware.

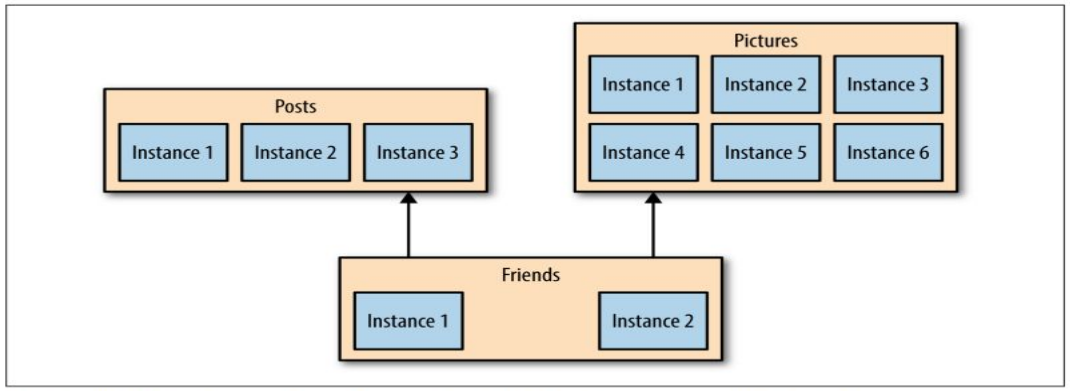


Figure 1-2. You can target scaling at just those microservices that need it

When embracing on-demand provisioning systems like those provided by Amazon Web Services, we can even apply this scaling on demand for those pieces that need it. This allows us to control our costs more effectively.

- Ease of deployment
  - A one-line change to a million-line-long monolithic application requires the whole application to be deployed in order to release the change. And the bigger the delta between releases, the higher the risk that we'll get something wrong !
  - With microservices, we can make a change to a single service and deploy it independ- ently of the rest of the system. This allows us to get our code deployed faster. If a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve.

Key Benefits of Microservices(Contd..)

- Organizational Alignment
  - Microservices allow us to better align our architecture to our organization, helping us minimize the number of people working on any one codebase to hit the sweet spot of team size and productivity
- Composability
  - With microservices, we allow for our functionality to be consumed in different ways for different purposes i.e. open up opportunities for reuse of functionality.
- Optimizing for Replaceability
  - With our individual services being small in size, the cost to replace them with a better implementation, or even delete them altogether, is much easier to manage as compared to some big, nasty legacy system sitting in the corner as it's too big and risky a job.

Microservices: No silver bullet

- Microservices are no free lunch or silver bullet, and make for a bad choice as a golden hammer. They have all the associated complexities of distributed systems.
- If one is coming from a monolithic system point of view, one'll have to get much better at handling deployment, testing, and monitoring to unlock the benefits.
- One would also need to think differently about how you scale your systems and ensure that they are resilient.
- Every company, organization, and system is different. A number of factors will play into whether or not microservices are right for you, and how aggressive you can be in adopting them.

Managing Trade-offs: Evolutionary Architect

- Making decisions in system design is all about trade-offs, and microservice architectures give us lots of trade-offs to make!
- When picking a datastore, do we pick a platform that we have less experience with, but that gives us better scaling?
- Is it OK for us to have two different technology stacks in our system? What about three?
- Some decisions can be made completely on the spot with information available to us, and these are the easiest to make. But what about those decisions that might have to be made on incomplete information.
- The evolutionary architect is one who understands that pulling off this feat is a constant balancing act. Forces are always pushing you one way or another, and understanding where to push back or where to go with the flow is often something that comes only with experience.

What Makes a Good Service?

- Do's
  - Loose Coupling
  - High Cohesion
  - The Bounded Context
  - Modular/Service boundaries aligning with the bounded context
  - Shared and Hidden Models

- What might go wrong ?
  - Premature Decomposition
    - Having an existing codebase you want to decompose into microservices is much easier than trying to go to microservices from the beginning.
- Also need to think in terms of Business Capabilities/Communication in Terms of Business Concepts.

In nutshell: Principles of microservices

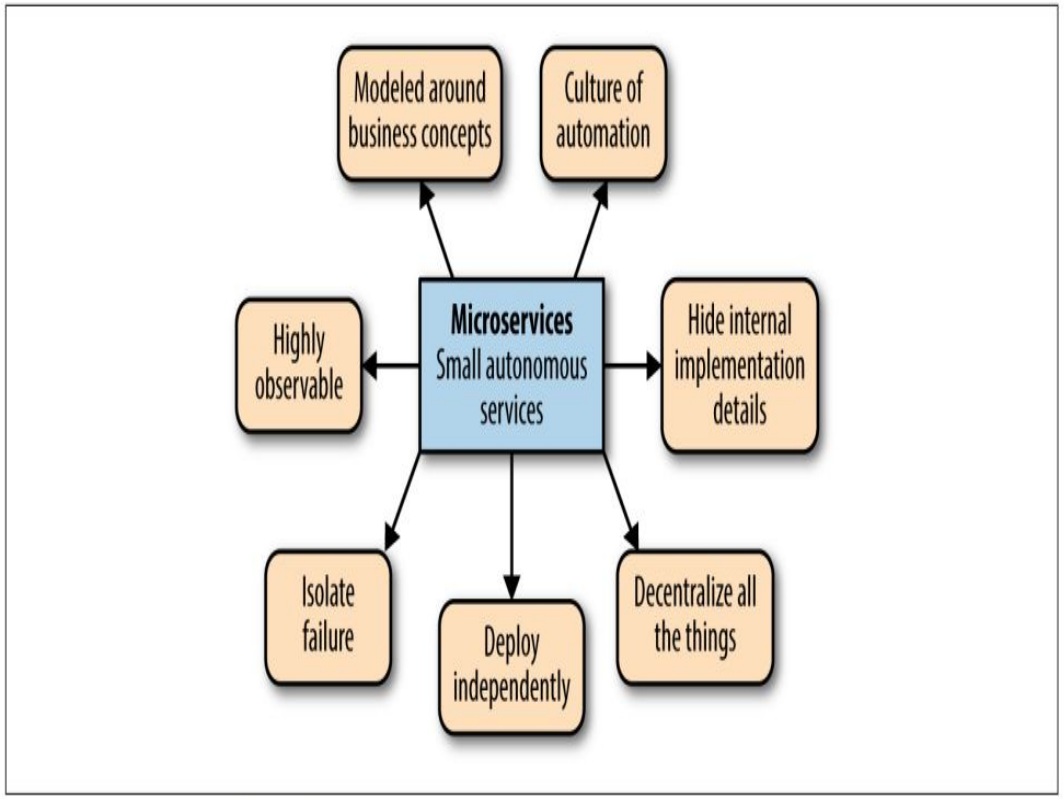


Figure 12-1. Principles of microservices

Microservice architectures give you more options, and more decisions to make. Making decisions in this world is a far more common activity than in simpler, monolithic systems.

We won't get all of these decisions right, it is guaranteed. So, knowing we are going to get some things wrong, what are our options? Well, suggestion is finding ways to make each decision small in scope; that way, if you get it wrong, you only impact a small part of your system.

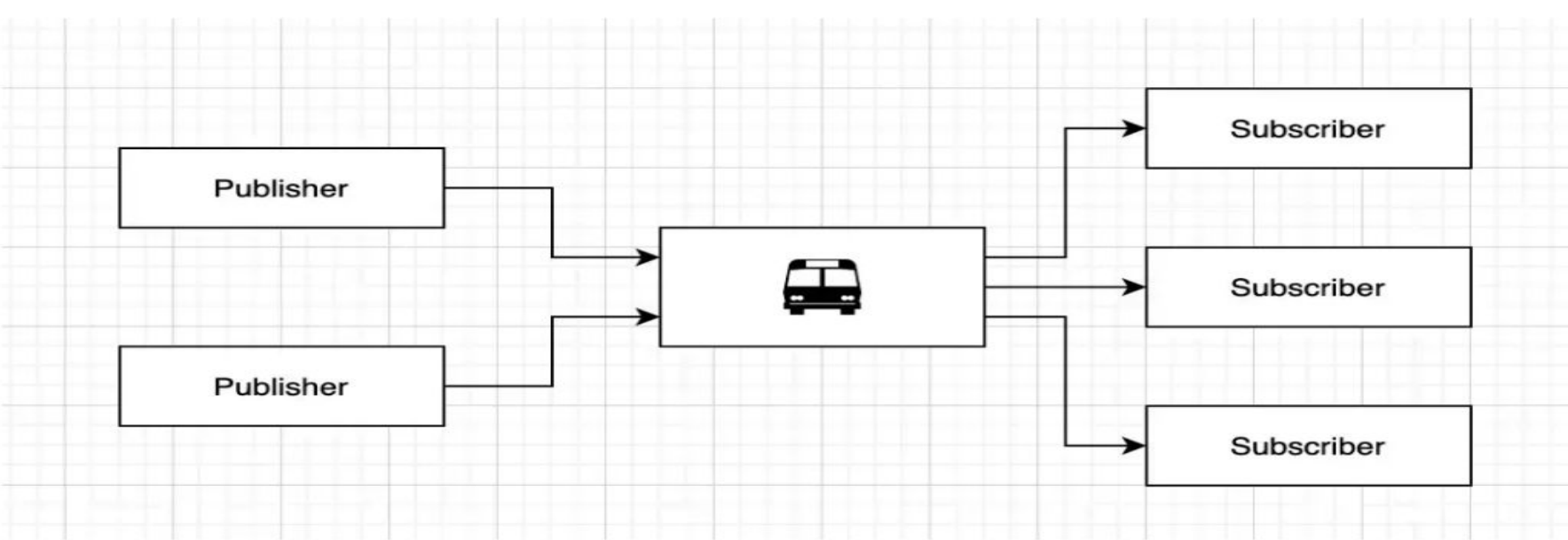
Learn to embrace the concept of evolutionary architecture, where your system bends and flexes and changes over time as you learn new things.

Introduction to Publish-Subscribe (pub/sub) model

- More and more of the **web is moving to microservice architecture**, which allows for loosely-coupled services to work together to provide functionality to users. While these services will often communicate with each other via **network requests**, the publish-subscribe model is **another common way for this collaboration** to occur.
- There are many different technologies that can be used to implement pub-sub, including but not limited to **Apache Kafka, Amazon Kinesis, Google Pub/Sub, and Microsoft Event Hubs.**
- Before we dive into pub-sub, let's take a quick detour to decoupling. If we're not convinced of the value of decoupling, pub-sub won't do us any good!
  - **Decoupling** means reducing the dependency between pieces of a system. This can mean two classes in the same codebase or two services in a system built on microservices, or anything in between.
  - Ok, but what do we mean by **"reducing the dependency"?** Obviously the different parts of the system rely on each other — they're part of the same system! Typically what people mean when they talk about the level of dependency between systems is whether or not one part of the system needs to be changed when the other does, or whether they can be updated independently of the other.

Publish-Subscribe Model(Contd..)

- Pub-sub is **decoupled** because using it, **we can change anything we want about a publisher** — we could even go so far as to rebuild it using an entirely different language or framework! — but **as long as it continues to produce messages in the same way**, we **don't need to change anything at all about the downstream subscriber**.
- **Same goes in reverse** — as long as a subscriber continues to be able to process messages that come to it in the same format, we can change anything we want about it and the publisher doesn't need to be aware of this change at all.



Two publishers producing data to a single message bus, with three subscribers consuming data from the message bus

Terminology commonly used in pub-sub offerings

- **Topics**
  - Each **message bus**, the servers where messages are sent to, stored, and pulled from, can have one or more **topics** on it.
  - Topics are typically used to separate different types of messages from each other, since not all subscribers will need to see all messages — they usually only need to operate on a pre-defined subset of messages.
- **Publishing**
  - This also sometimes referred to as “producing.” A publisher is anything that puts messages onto the message bus.
  - This can be a database trigger, a web request, an API call, running a script or chron job, whatever. Messages typically take a pre-defined format, and are in JSON. They are pushed to a specific **topic**.
- **Subscribing**
  - This also sometimes referred to as “consuming.”
  - Subscribers are typically responsible for a specific piece of behavior, and will subscribe to a specific **topic**.
  - The beauty of pub-sub is that you can have as many subscribers as you want, which allows you to scale them all separately, depending on how long each message takes a given subscriber to process, and how much traffic is on the given topic to which they subscribe.

Terminology used in pub-sub offerings(Contd..)

- **Partitioning**
  - Topics can be further broken down into partitions.
  - How you partition your data can be important, as ordering is not guaranteed across partitions.
  - If the order in which messages are processed is important, all messages that need to be processed in order, will need to be on the same partition.
  - For example, you may partition messages based on a user_id, if all messages for a given user need to be processed in order, but order doesn't matter across different users.
  - The number of partitions you choose can be an important early decision, since it's difficult to change later (it would cause downtime and the possibility of losing ordering). You also cannot ever have more workers on a given subscriber than you have partitions, since messages on partitions are never divided (that would result in losing ordering).

How is this different from background tasks?

- The primary difference between **background task queues and pub-sub systems** is the lifecycle of a given message or event.
- With background tasks, messages can be pushed to the queue from any number of places, just like with pub-sub. And a background task queue can have any number of workers processing messages from that queue, just like with pub-sub.
- **However, with background queues, once a message has been processed by *any one of the workers*, that message is removed from the queue and not processed any further.**
- With pub-sub, each subscriber **group** (cluster of workers) is responsible for a discrete piece of logic, so each group must process each message.
- **Pub-sub message buses also often retain data for some period of time, instead of discarding messages from the queue as soon as they're processed.**
- This allows events to be replayed, should something have gone wrong the first time a subscriber processed them, or even if a new subscriber needs to be added after the fact!

Real world Design Problems and Case Studies

Glossary

- Need of right system design
- Types of Design Requirements
- Approaching a design problem
- Design Trade-Offs
- Performance Optimization
- Resource Awareness and Compute Management
- Importance of Data Modelling
- Case Studies
  - URL SHORTENER
  - Messaging App
  - News Feed Service
  - Video Hosting Service

Need of right system design

- **Over the last two decades, there have been a lot of advancements in large-scale web applications. These advancements have redefined the way we think about software development.**
- **All of the apps and services that we use daily, like Facebook, Instagram, and Twitter, are scalable systems. Billions of people worldwide access these systems concurrently, so they need to be designed to handle large amounts of traffic and data. This is where system design comes in.**
- A good system design strategy is **key for enabling optimal product development** as it offers the necessary roadmap for the project execution.
- It allows **breaking down problems** into multiple well-defined subproblems and **enables problem-solving of the smaller elements** so that they **can fit into the bigger picture**.
- **In the early stages of your career, learning**

Types of Design Requirements

- **Functional/Feature Requirements:**
  - They are basically the requirements stated by the user which one can see directly in the final product.
  - Helps you verify the functionality of the software.

Example

1) Authentication of user whenever he/she logs into the system.

2) System shutdown in case of a cyber attack.

3) A Verification email is sent to user whenever he/she registers for the first time on some software system.

- **Non-functional/Behavioral requirements:**
  - These are basically the quality constraints that the system must satisfy.
  - They basically deal with issues like

Example

1) Emails should be sent with a latency of no greater than 12 hours from such an activity.

2) The processing of each request should be done within 10 seconds

3) The site should load in 3 seconds when the number of simultaneous users are > 10000

Approaching a design problem

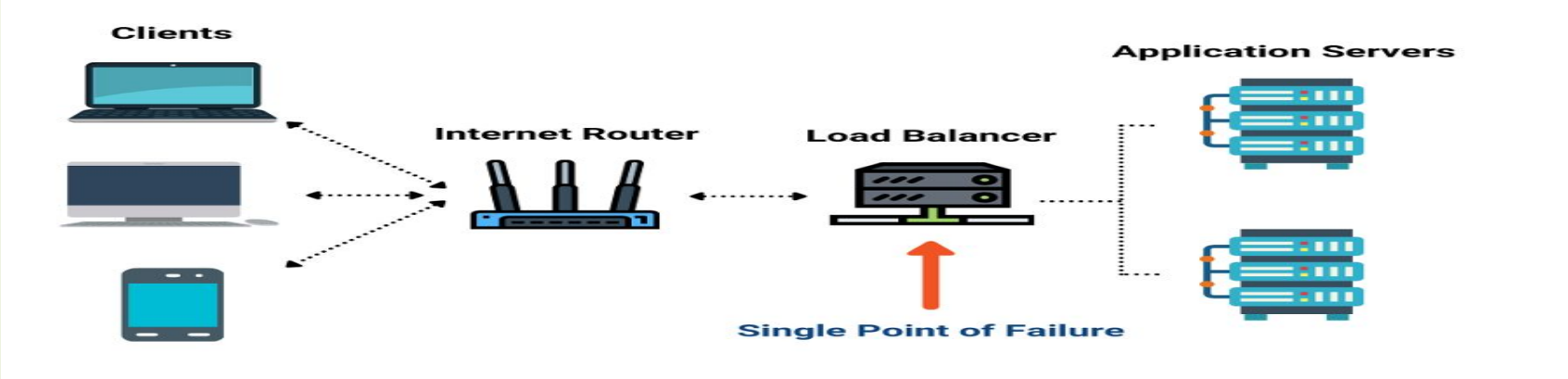
- **Step 1: Requirements clarification (Functional & Non-Functional requirements)**
  - In this step, we convert a vague business requirement into a concrete computer software problem since we really need to know what we are doing afterward.
  - Think deeply and ask questions to clarify requirements and assumptions.
- **Step 2: BACK-OF-THE-ENVELOPE estimation**
  - back-of-the-envelope calculations **are estimates you create using a combination of thought experiments and common performance numbers** to get a good feel for which designs will meet your requirements.
  - Power of two , latency numbers every programmer should know, and availability numbers are very useful in this estimation.
- **Step 3: Create a high-level design**
  - To begin the design process, it can be helpful to outline a high-level design with all of the critical components and try to draw **a diagram representing the system's core components**.
  - This **will help you to identify all of the components that are needed to solve the problem** from end to end.
  - **By outlining the interaction of key components, you can better explain the overall structure of the system and how the various components will work together to achieve the desired results**.

Approaching a design problem (Contd..)

- **Step 4: Database Design**
  - it is important to define how data will be processed.
  - This includes identifying the inputs and outputs of the system, how they will be stored, and how the data will flow through the system.
  - **Determining which database would be the best fit for the problem** can also be helpful at this stage.
- **Step 5: Design core components**
  - Once you have outlined the high-level structure of the system, it can be helpful to design the major components in more detail.
  - It is important to think through various approaches, benefits, and drawbacks of each option and justify why you have chosen a particular approach.
- **Step 6: Scale the design**
  - To ensure that your system is able to meet the scale requirements of the problem, it is important to identify key concepts relevant to scalability. **This may include considering approaches such as load balancing, horizontal scaling, caching, database sharding, replication, etc. to address scalability issues.**
  - **By considering these options and determining which are most appropriate for your system, you can ensure that your design is able to handle the expected workload and meet the needs of your users.**

Approaching a design problem (Contd..)

- **Step 7: Identifying and resolving bottlenecks**
  - To ensure that your system is robust and able to handle unexpected challenges, it is important to think about as many potential bottlenecks as possible and consider different approaches to mitigate them. Some questions to consider might include:
    - Is there any single point of failure in the system? What steps are being taken to mitigate this risk?
    - Are there enough replicas of the data to ensure that users can still be served if a few servers are lost?
    - Are there enough copies of different services running to ensure that a few failures will not cause a total system shutdown?
    - How is the performance of the service being monitored? Are there alerts in place to notify the



Design Trade-Offs

- Here are some of the popular design tradeoffs in distributed systems:
  - **Pull vs. Push Notifications**
    - **Pull notification** means **a client requesting a server** to check if there are any updates.
    - **Push notification** means **a server notifying a client** about the updates.
  - **Throughput vs Latency**
    - Throughput can be defined as the number of times a certain task can be performed in a given duration of time.
    - Latency is the time to perform some action or to produce some result.
  - **Memory vs Latency**
    - Large amount of memory usage(caching) can reduce latency of read operations
  - **Replication vs Consensus**
    - Higher replication leads to complexity in quorum based consensus among replicated nodes.
  - **Consistency vs Availability**
    - Standard CAP theorem

Performance Optimization example: CDN

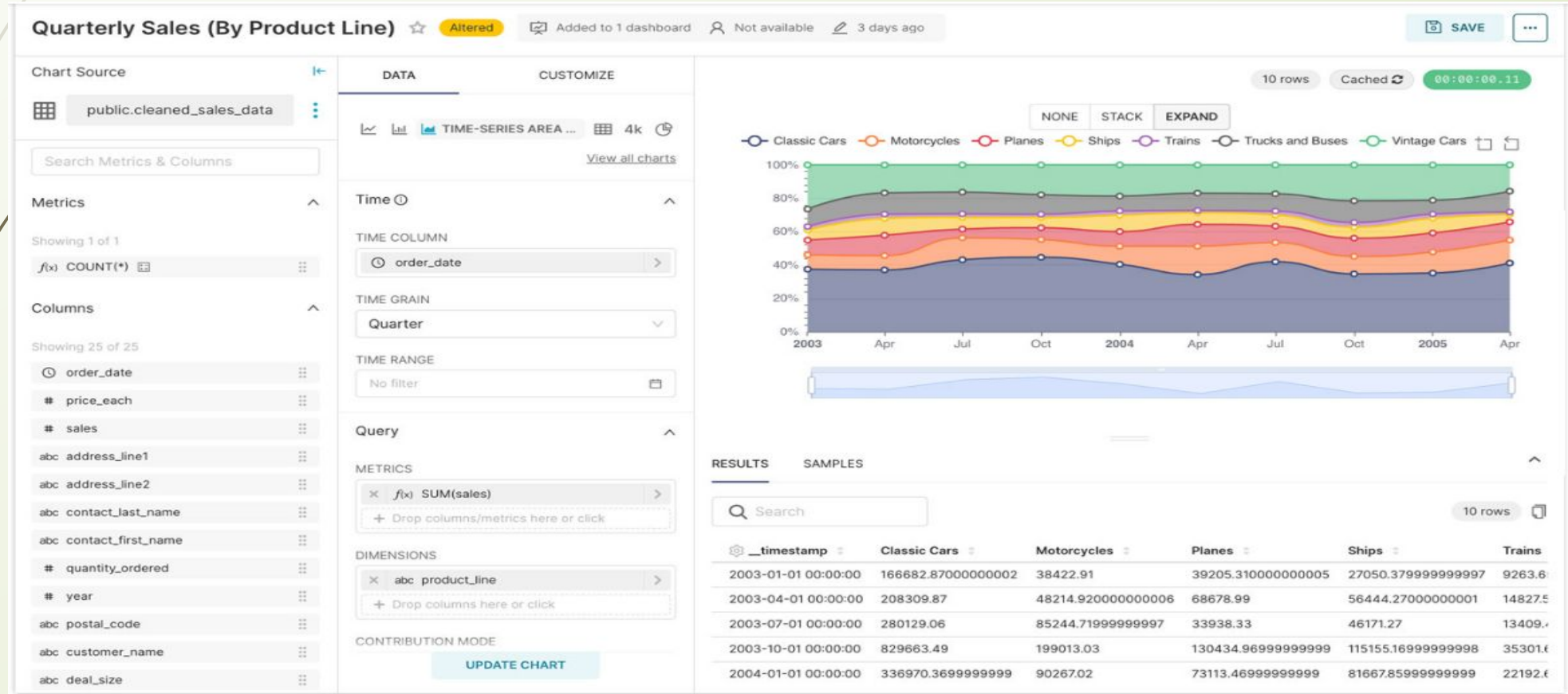
- A CDN is **a network of geographically dispersed servers** used to deliver static content. CDN servers **cache static content** like images, videos, CSS, JavaScript files, etc.
- Here is how CDN works at the high-level and **improves load time**:
  - When a **user visits a website**, **a CDN server closest to the user will deliver static content**.
  - Intuitively, the further users are from CDN servers, the slower the website loads.
  - **For example, if CDN servers are in San Francisco, users in Los Angeles will get content faster than users in Europe.**

Resource Awareness and Compute Management

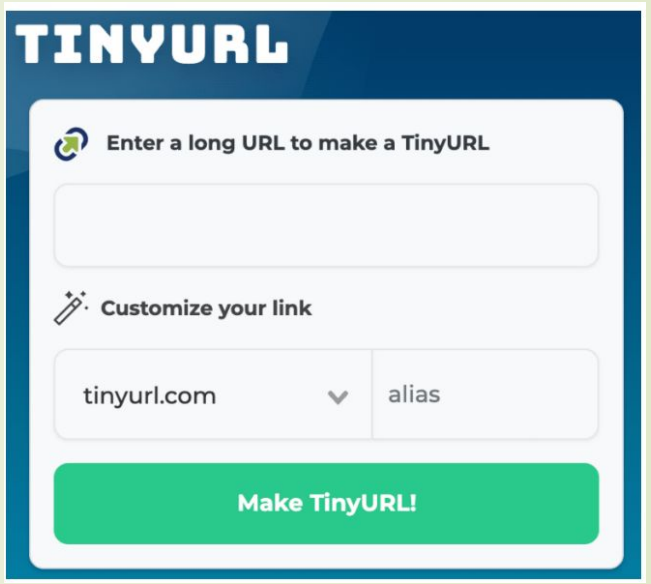
- The design should also be aware about the resource requirements:
  - Number of nodes where the final system would be deployed:
    - CPU requirements
    - Memory Requirements
    - Network Bandwidth requirements
- Compute management
  - Normally organizations have dedicated teams for compute management who are responsible for oversight and maintenance of an organization's compute resources.
  - This promotes quicker time to action when solving problems, reducing downtime and promotes the overall seamless function of your data infrastructure.
  - Cloud-based providers : AWS, Google Cloud

Importance of Data Modelling

- **A comprehensive and optimized data model helps create a simplified, logical database that eliminates redundancy, reduces storage requirements, and enables efficient retrieval.** Today's data models **transform raw data into useful information** that can be turned into **dynamic visualizations**.



Case Study : URL Shortener

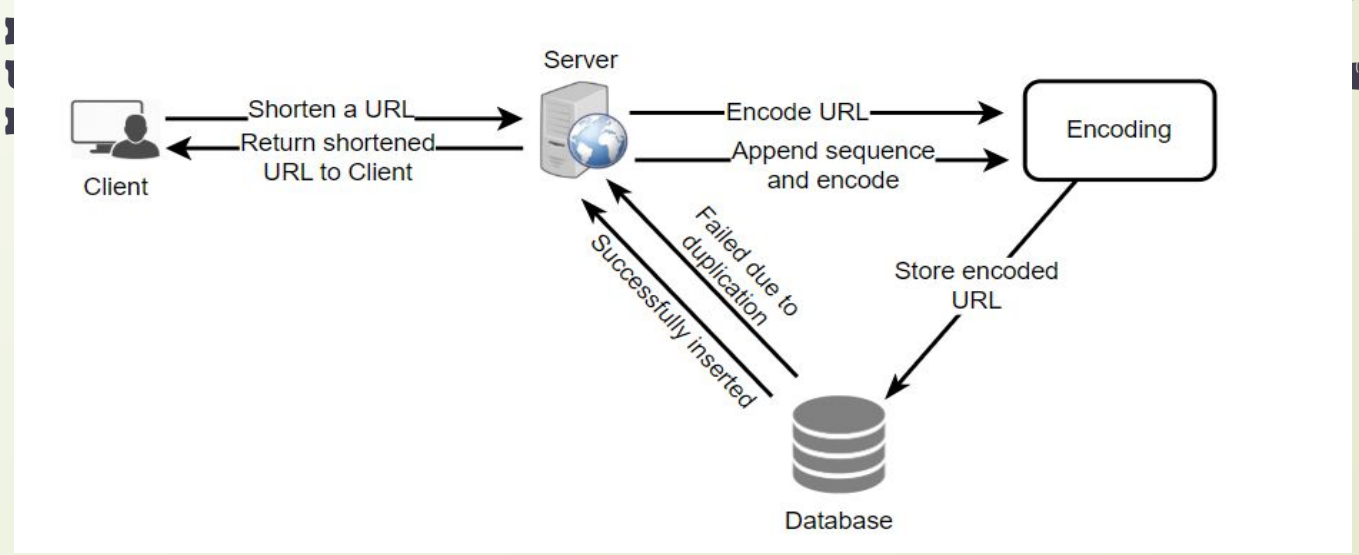


- Functional Requirements
  - Get a short URL from a long URL.
  - Redirect to the long URL when a user tries to access the short URL.
- Non Functional Requirements
  - Very low latency
  - Very High Availability
- Back of the envelope estimation

- Write operation: 100 million URLs are generated per day.
- Write operation per second: 100 million / 24 /3600 = 1160
- Read operation: Assuming ratio of read operation to write operation is 10:1, read operation per second: 1160 * 10 = 11,600
- Assuming the URL shortener service will run for 10 years, this means we must support 100 million * 365 * 10 = 365 billion records.
- Assume average URL length is 100.
- Storage requirement over 10 years: 365 billion * 100 bytes * 10 years = 365 TB

URL shortener (Contd..)

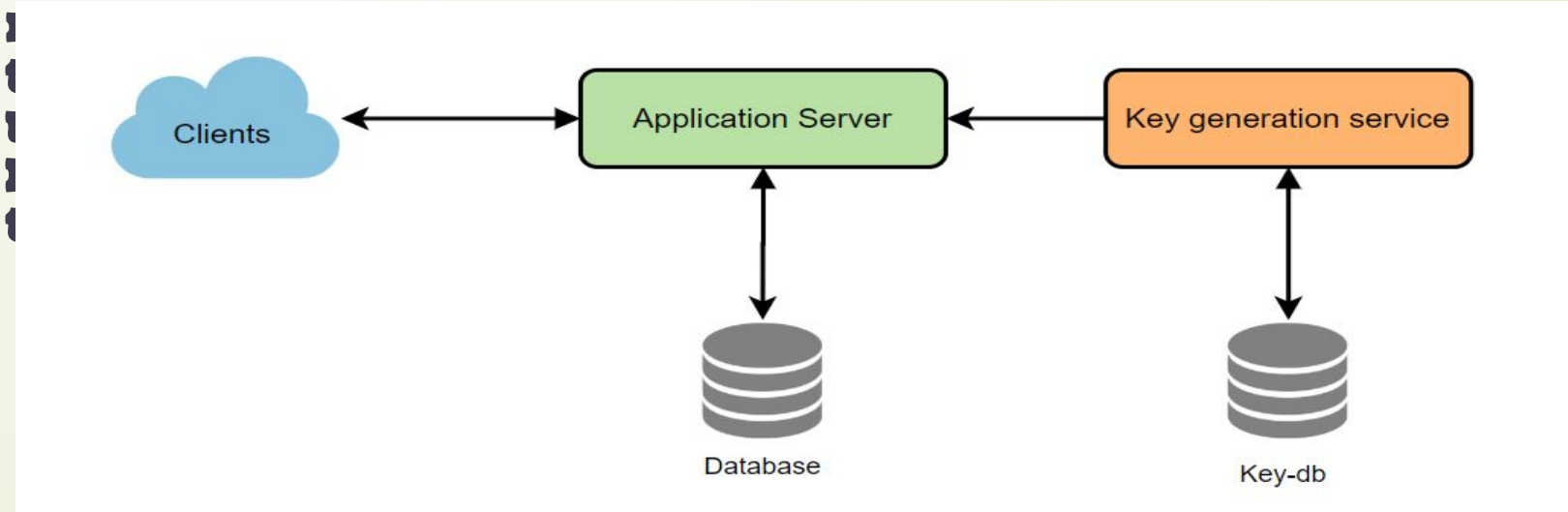
- **We'll explore two solutions here:**
  - a. Encoding actual URL
    - **We can compute a unique hash (e.g., MD5 or SHA256, etc.) of the given URL. The hash can then be encoded for display. This encoding could be base36 ([a-z ,0-9]) or base62 ([A-Z, a-z, 0-9]) and if we add '+' and '/' we can use Base64 encoding.**
    - **Using base64 encoding, a 6 letters long key would**



Request flow for shortening of a URL

URL shortener (Contd..)

- b. Generating keys offline
  - We can have a standalone Key Generation Service (KGS) that generates random six-letter strings beforehand and stores them in a database (let's call it key-DB).
  - Whenever we want to shorten a URL, we will take one of the already-generated keys and use it.
  - Servers can use Key Generation Service to



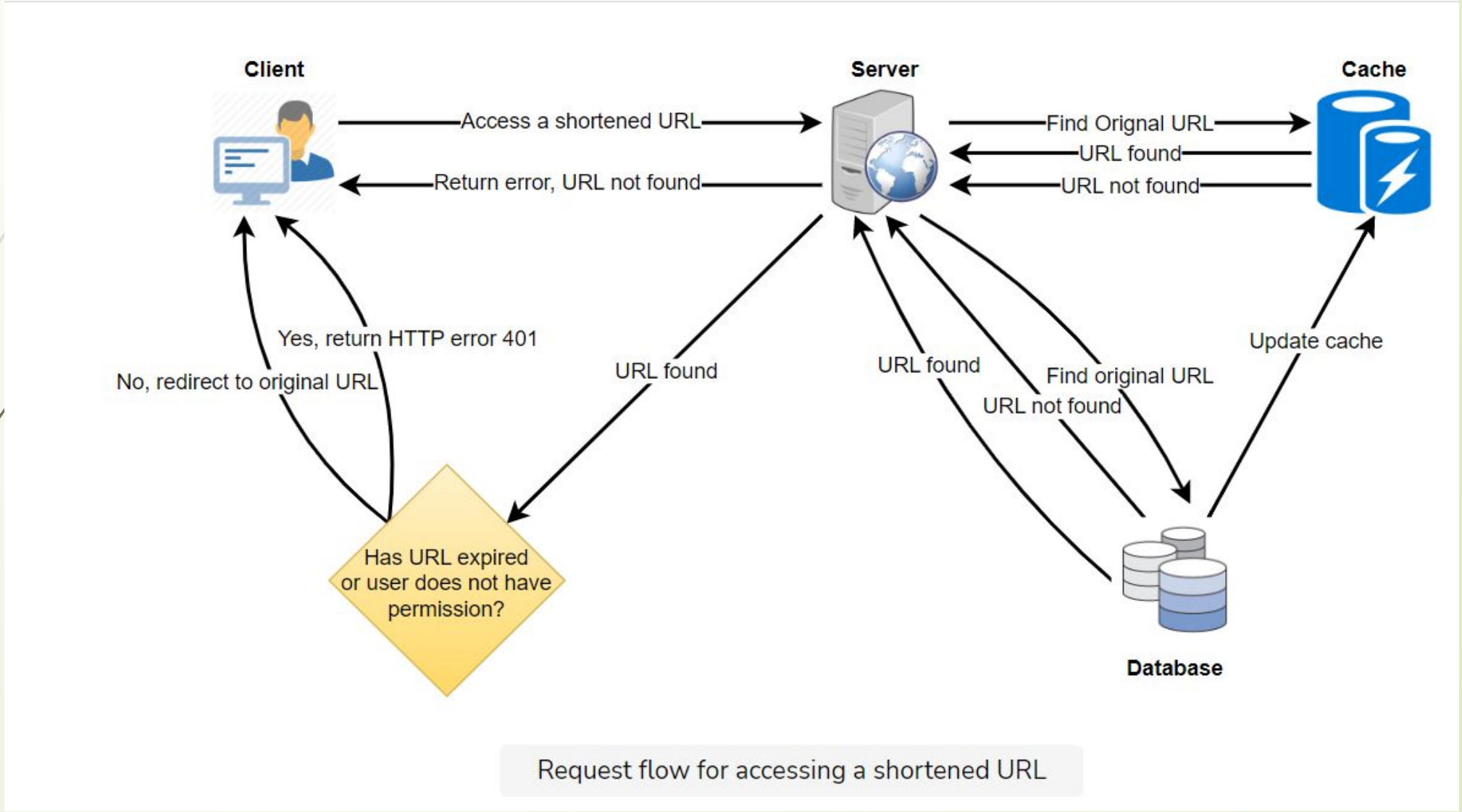
URL shortener (Contd..)

- To scale out our DB, we need to partition it so that it can store information about billions of URLs. Therefore, we need to develop a partitioning scheme that would divide and store our data into different DB servers.
  - a. Range Based Partitioning: We can store URLs in separate partitions based on the hash key's first letter. Hence we will save all the URL hash keys starting with the letter 'A' (and 'a') in one partition, save those that start with the letter 'B' in another partition, and so on.
    - The main problem with this approach is that it can lead to unbalanced DB servers.
    - For example, we decide to put all URLs starting with the letter 'E' into a DB partition, but later we realize that we have too many URLs that start with the letter 'E.'
  - b. Hash-Based Partitioning: In this scheme, we take a hash of the object we are storing. We then calculate which partition to use based upon the

URL shortener (Contd..)

- To improve latency of user requests for accessing shortened URL, we can cache URLs that are frequently accessed. We can use any off-the-shelf solution like Memcached, which can store full URLs with their respective hashes. Thus, the application servers, before hitting the backend storage, can quickly check if the cache has the desired URL.
  - How much cache memory should we have?
    - We can start with 20% of daily traffic and, based on clients' usage patterns, we can adjust how many cache servers we need.
    - As estimated above, we need 170GB of memory to cache 20% of daily traffic.
  - Which cache eviction policy would best fit our needs?
    - Least Recently Used (LRU) can be a reasonable policy for our system.
  - How can each cache replica be updated?

URL shortener (Contd..)



Request flow for accessing a shortened URL

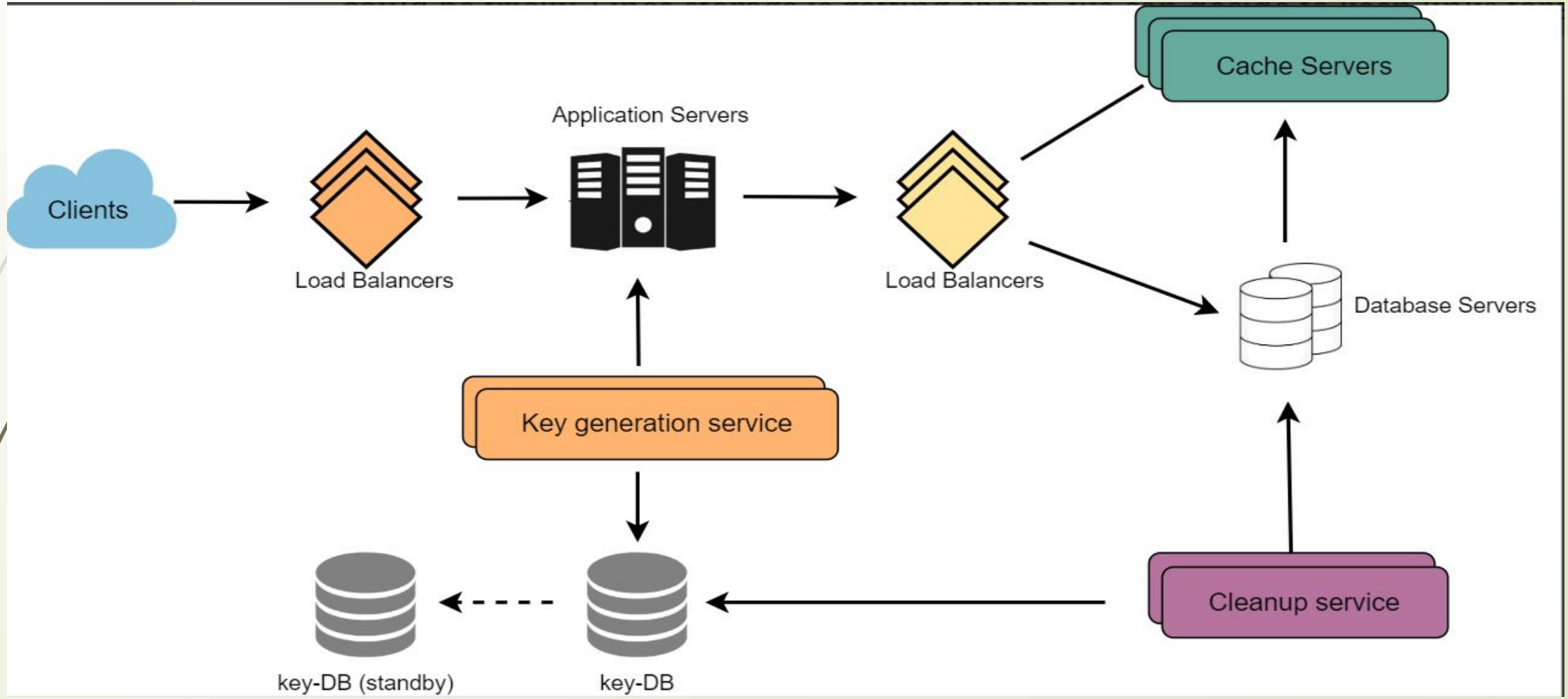
URL shortener (Contd..)

- **Load Balancer (LB)**
  - **We can add a Load balancing layer at three places in our system:**
    1. **Between Clients and Application servers**
    2. **Between Application Servers and database servers**
    3. **Between Application Servers and Cache servers**
- **Initially, we could use a simple Round Robin approach that distributes incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is that if a server is dead, LB will take it out of the rotation and stop sending any traffic to it.**
- **A problem with Round Robin LB is that we do not consider the server load. As a result, if a server is overloaded or slow, the LB will not stop sending new requests to that server. To**

URL shortener (Contd..)

- **Purging or DB cleanup**
  - Should entries stick around forever, or should they be purged? If a user-specified expiration time is reached, what should happen to the link?
  - If we chose to continuously search for expired links to remove them, it would put a lot of pressure on our database. Instead, we can slowly remove expired links and do a lazy cleanup.
  - Our service will ensure that only expired links will be deleted, although some expired links can live longer but will never be returned to users.
    - Whenever a user tries to access an expired link, we can delete the link and return an error to the user.
  - A separate Cleanup service can run periodically to remove expired links from our storage and cache. This service should be very lightweight and scheduled to run only when the user traffic is expected to be low.
  - We can have a default expiration time for each link (e.g., two years).

URL shortener (Contd..)



Case Study : Messaging Application

- Functional Requirements
  1. **Messenger should support one-on-one conversations between users.**
  2. **Messenger should keep track of the online/offline statuses of its users.**
  3. **Messenger should support the persistent storage of chat history.**

- Non Functional Requirements
  1. **Users should have a real-time chatting experience with minimum latency.**
  2. **Our system should be highly consistent; users should see the same chat history on all their devices.**
  3. **Messenger's high availability is desirable; we can tolerate lower availability in the interest of consistency.**
- Back of the envelope estimation
  - Let's assume that we have **500 million daily active users**, and on average, **each user sends 40 messages daily**; this gives us **20 billion messages per day**.
  - Storage Estimation: Let's assume that, on average, **a message is 100 bytes. So to**

Messaging Application(Contd..)

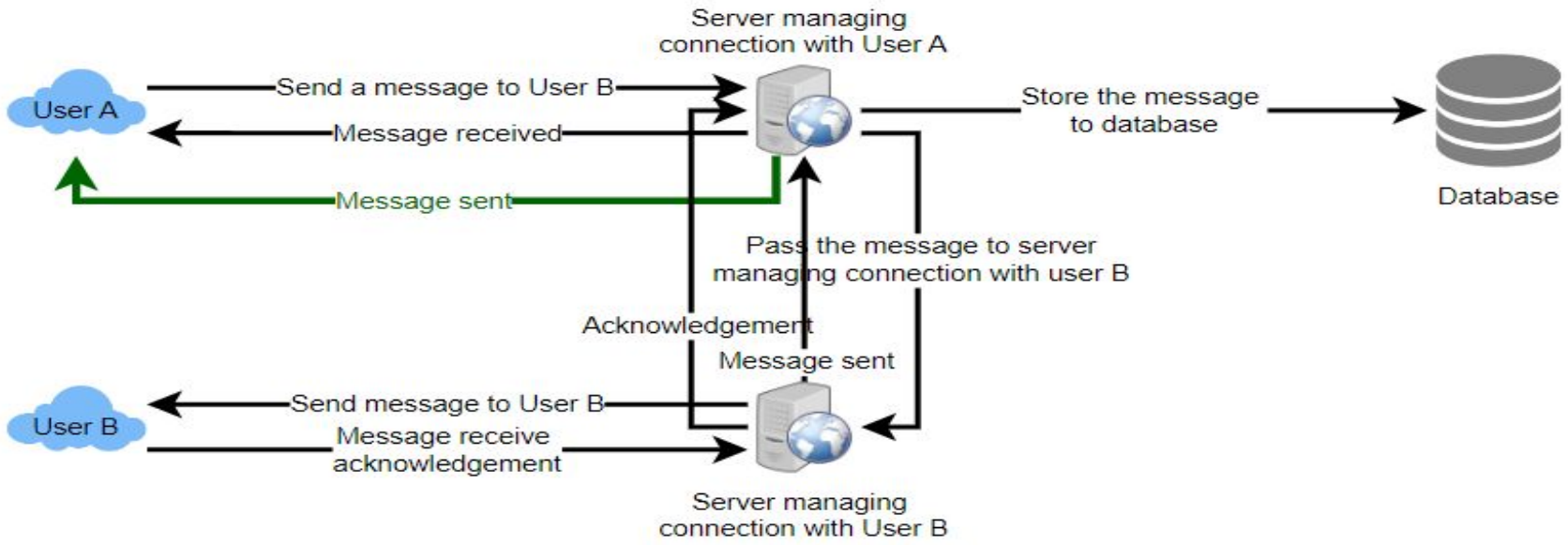
- Besides chat messages, we also need to store users' information, messages' metadata (ID, Timestamp, etc.). Not to mention, the above calculation doesn't take data compression and replication into consideration.
- **Bandwidth Estimation**: If our service is getting 2TB of data every day, this will give us 25MB of incoming data for each second.
  - 2 TB / 86400 sec ~= 25 MB/s
  - Since each incoming message needs to go out to another user, we will need the same amount of bandwidth 25MB/s for both upload and download.

High level estimates

Messages	Bandwith
Total messages	20 billion per day
Storage for each day	2TB
Storage for 5 years	3.6PB
Incoming data	25MB/s
Outgoing data	25MB/s

Messaging Application(Contd..)

- High level design : **At a high level, we will need a chat server that will be the central piece**



The detailed workflow would look like this:

1. User-A sends a message to User-B through the chat server.
2. The server receives the message and sends an acknowledgment to User-A.
3. The server stores the message in its database and sends the message to User-B.
4. User-B receives the message and sends the acknowledgment to the server.
5. The server notifies User-A that the message has been delivered successfully to User-B.

Messaging Application(Contd..)

- Detailed Component Design : **Let's try to build a simple solution first where everything runs on one server and talk about these scenarios one by one:**
  - Messages Handling : How would we efficiently send/receive messages?
    - **To send messages, a user needs to connect to the server and post messages for the other users. To get a message from the server, the user has two options:**

1. Pull model: **Users can periodically ask the server if there are any new messages for them. The server needs to keep track of messages that are still waiting to be delivered, and as soon as the receiving user connects to the server to ask for any new message, the server can return all the pending messages. To minimize latency for the user, they have to check the server quite frequently, and most of the time, they will be getting an empty response if there are no pending**

How will clients maintain an open connection with the server? We can use HTTP Long Polling or WebSockets.

Messaging Application (Contd..)

How can the server keep track of all the opened connections to efficiently redirect messages to the users? The server can maintain a hash table, where "key" would be the UserID and "value" would be the connection object. So whenever the server receives a message for a user, it looks up that user in the hash table to find the connection object and sends the message on the open request.

What will happen when the server receives a message for a user who has gone offline? If the receiver has disconnected, the server can notify the sender about the delivery failure. However, if it is a temporary disconnect, e.g., the receiver's long-poll request just timed out, then we should expect a reconnect from the user. In that case, we can ask the sender to retry sending the message. This retry could be embedded in the client's logic so that users don't have to retype the message. The server can also store the message for a while and retry sending it once the receiver reconnects.

How many chat servers do we need? Let's plan for 500 million connections at any time. Assuming a modern server can handle 50K concurrent connections at any time, we would need 10K such servers.

How do we know which server holds the connection to which user? We can introduce a software load balancer in front of our chat servers; that can map each UserID to a server to redirect the request.

Messaging Application (Contd..)

- Storing and retrieving the messages from the database : **Whenever the chat server receives a new message, it needs to store it in the database. To do so, we have two options:**
  1. **Start a separate thread, which will work with the database to store the message.**
  2. **Send an asynchronous request to the database to store the message.**
- **Which storage system should we use?**
  - **We need to have a database that can support a very high rate of small updates and also fetch a range of records quickly.**
  - **This is required because we have a huge number of small messages that need to be inserted in the database and, while querying, a user is mostly interested in sequentially accessing the messages.**

How should clients efficiently fetch data from the server? Clients should paginate while fetching data from the server. Page size could be different for different clients, e.g., cell phones have smaller screens, so we need fewer messages/conversations in the viewport.

columns.

Messaging Application (Contd..)

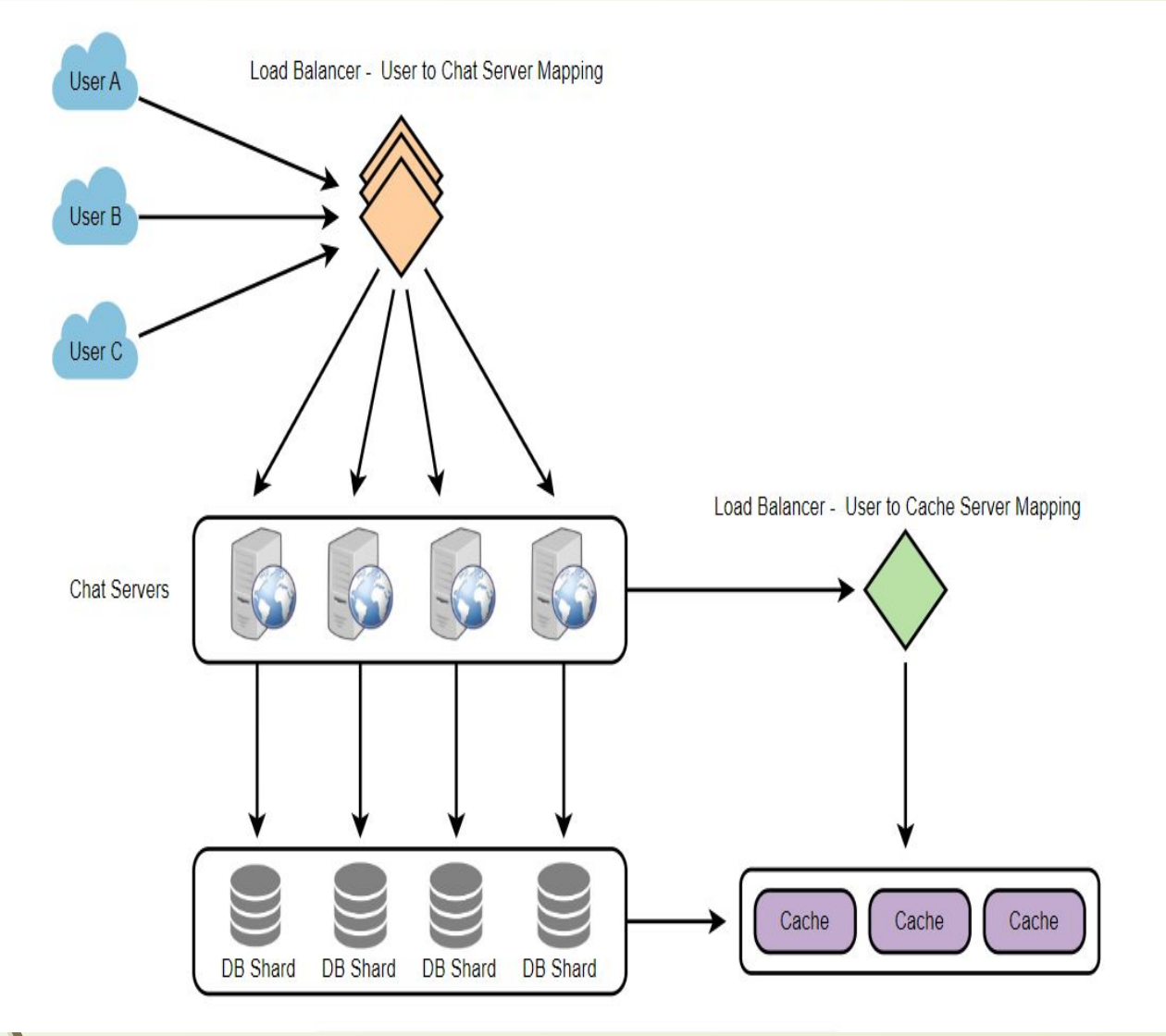
- **Managing user's status**
  - **We need to keep track of user's online/offline status and notify all the relevant users whenever a status change happens.**
  - **Since we are maintaining a connection object on the server for all active users, we can easily figure out the user's current status from this.**
  - **With [illegible] active users at any time, it will be**

1. Whenever a client starts the app, it can pull the current status of all users in their friends' list.
2. Whenever a user sends a message to another user that has gone offline, we can send a failure to the sender and update the status on the client.
3. Whenever a user comes online, the server can always broadcast that status with a delay of a few seconds to see if the user does not go offline immediately.
4. Clients can pull the status from the server about those users that are being shown on the user's viewport. This should not be a frequent operation, as the server is broadcasting the online status of users and we can live with the stale offline status of users for a while.
5. Whenever the client starts a new chat with another user, we can pull the status at that time.

Messaging Application (Contd..)

- Data partitioning
  - **Since we will be storing a lot of data (3.6PB for five years), we need to distribute it onto multiple database servers. So, what will be our partitioning scheme?**
  - Partitioning based on UserID: **Let's assume we partition based on the hash of the UserID so that we can keep all messages of a user on the same database. If one DB shard is 4TB, we will have "3.6PB/4TB ~= 900" shards for five years. For simplicity, let's assume we keep 1K shards. So we will find the shard number by "hash(UserID) % 1000" and then store/retrieve the data from there. This partitioning scheme will also be very quick to fetch chat history for any user.**
  - Partitioning based on MessageID: **If we store different messages of a user on separate database shards, fetching a range of messages of a chat would be very slow, so we should not adopt this scheme.**
- Cache
  - **We can cache a few recent messages (say last 15) in a few recent conversations that are visible in a user's viewport (say last 5). Since we decided to store all of the user's**

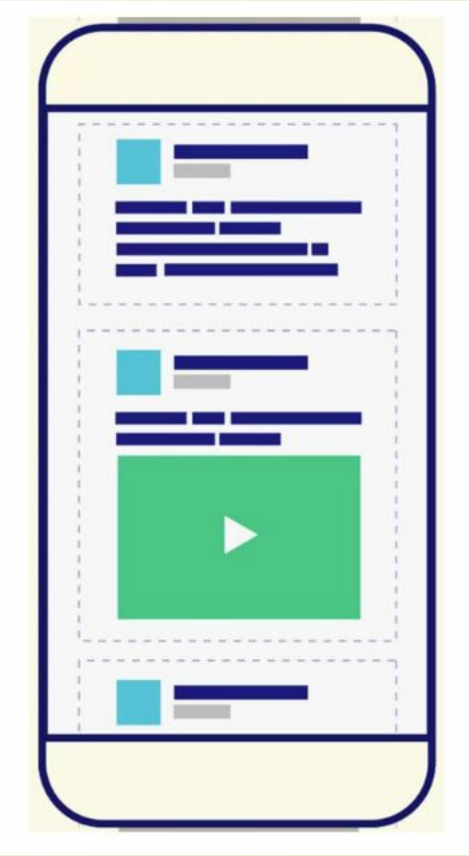
Messaging Application (Contd..)



- **Design Summary:**
  - Clients will open a connection to the chat server to send a message; the server will then pass it to the requested user.
  - All the active users will keep a connection open with the server to receive messages.
  - Whenever a new message arrives, the chat server will push it to the receiving user on the long poll request.
  - Messages can be stored in HBase, which supports quick small updates and

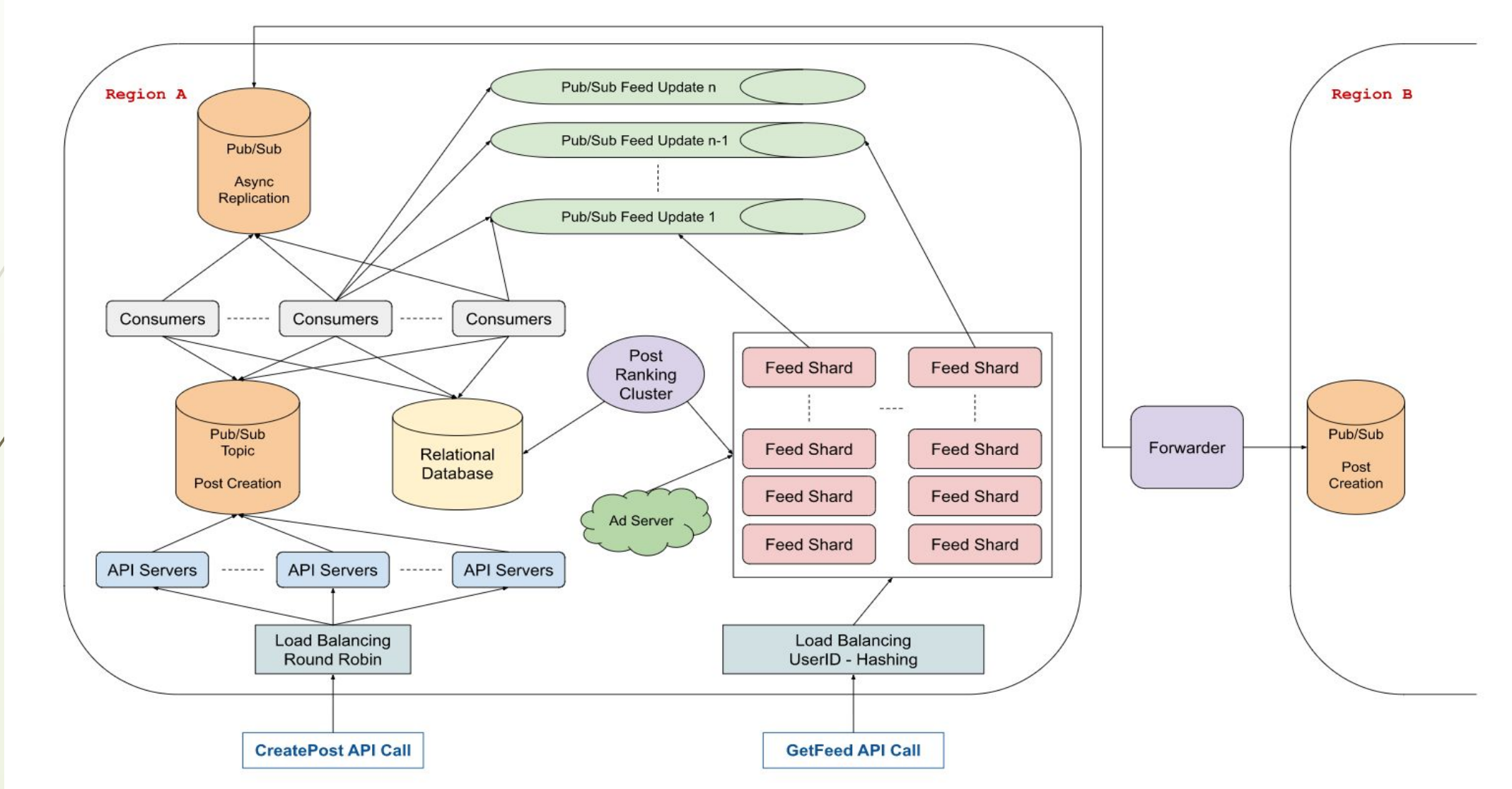
Case Study : News Feed Service

- Functional Requirements
  - Loading a user's news feed
  - Updating the news feed
  - Posting status updates.
  - Cross-region support
- Non-functional requirements
  - Low latency
  - Highly scalable (1 billion+ users)



- News feed is the constantly updating list of stories in the middle of your home page.
- News Feed includes status updates, photos, videos, links, app activity, and likes from people, pages, and groups that you follow.

News Feed Service(Contd..)



High Level Design

News Feed Service(Contd..)

- We'll start with the extremities of our system and go inward, first talking about the two API calls**, CreatePost and GetNewsFeed**, then, getting into the **feed creation** and **storage strategy**, our **cross-region design**, and **finally tying everything together** in a fast and scalable way.
- **CreatePost API**

```
CreatePost(
    user_id: string,
    post: data
)
```

  - When a user creates a post:
    - the API call goes through some load balancing before landing on one of many API servers (which are stateless).
    - Those API servers then create a message on a **Pub/Sub topic**, notifying its subscribers of the new post that was just created.
    - Those subscribers will do a few things, so let's call them S1 for future reference.
    - Each of the subscribers S1 reads from the topic and is responsible for creating the post inside a relational database.
- **Post Storage**
  - We can have one main relational database to store most of our system's data, including posts and users. This database will have *very large* tables.

News Feed Service(Contd..)

- **GetNewsFeed API**

The *GetNewsFeed* API call will most likely look like this:

```
GetNewsFeed(
    user_id: string,
    pageSize: integer,
    nextPageToken: integer,
) => (
    posts: []{
        user_id: string,
        post_id: string,
        post: data,
    },
    nextPageToken: string,
)
```

The *pageSize* and *nextPageToken* fields are used to **paginate** the newsfeed; pagination is necessary when dealing with large amounts of listed data, and since we'll likely want each news feed to have up to 1000 posts, pagination is very appropriate here.

News Feed Service(Contd..)

- **Feed creation and storage**
  - Since our databases tables are going to be so large, with billions of millions of users and tens of millions of posts every week, **fetching news feeds from our main database every time a *GetNewsFeed* call is made isn't going to be ideal**.
  - **We can't expect low latencies when building news feeds from scratch because querying our huge tables takes time**, and sharding the main database holding the posts wouldn't be particularly helpful since news feeds would likely need to aggregate posts across shards, which would require us to perform cross-shard joins when generating news feeds; we want to avoid this.
  - **Instead, we can store news feeds separately from our main database across an array of shards**.
  - We can have a **separate cluster of machines** that can act as a proxy to the relational database and **be in charge of aggregating posts, ranking them via the ranking algorithm that we're given, generating news feeds, and sending them to our shards every so often** (every 5, 10, 60 minutes, depending on how often we want news feeds to be updated).

News Feed Service(Contd..)

- If we average each post at 10kB, and a newsfeed comprises of the top 1000 posts that are relevant to a user, that's 10MB per user, or **10000TB** of data total.
- Assuming 1 billion news feeds (for 1 billion users) containing 1000 posts of up to 10 KB each, we can estimate that we'll need 10 PB (petabytes) of storage to store all of our users' news feeds.
- **We can use 1000 machines of 10 TB each as our news-feed shards**.

```
~10 KB per post
~1000 posts per news feed
~1 billion news feeds
~10 KB * 1000 * 1000^3 = 10 PB = 1000 * 10 TB
```

- **To distribute the newsfeeds roughly evenly, we can shard based on the user id**.
- **When a *GetNewsFeed* request comes in, it gets load balanced to the right news feed machine, which returns it by reading on local disk.** If the newsfeed doesn't exist locally, we then go to the source of truth (the main database, but going through the proxy ranking service) to gather the relevant posts. This will lead to increased latency but shouldn't happen frequently.

News Feed Service(Contd..)

- **Wiring Updates Into Feed Creation**
  - We now **need to have a notification mechanism that lets the feed shards know that a new relevant post was just created and that they should incorporate it** into the feeds of impacted users.
  - **We would use a Pub/Sub service for this. Each one of the shards will subscribe to its own topic--we'll call these topics the Feed Notification Topics (FNT)--and the original subscribers S1 will be the publishers for the FNT.**
    - When S1 gets a new message about a post creation, it searches the main database for all of the users for whom this post is relevant (i.e., it searches for all of the friends of the user who created the post), it filters out users from other regions who will be taken care of asynchronously, and it maps the remaining users to the FNT using the same hashing function that our *GetNewsFeed* load balancers rely on.
  - **For posts that impact too many people, we can cap the number of FNT topics** that get messaged to reduce the amount of internal traffic that gets generated from a single post.
    - For those big users we can rely on the **asynchronous feed creation** to eventually kick in and let the post appear in feeds of users whom we've skipped when the feeds get refreshed manually.

News Feed Service(Contd..)

- **Cross-Region Strategy**
  - When *CreatePost* gets called and reaches our Pub/Sub subscribers, they'll **send a message to another Pub/Sub topic that some forwarder service in between regions will subscribe to.**
  - The forwarder's job will be, as its name implies, to forward messages to other regions so as to replicate all of the *CreatePost* logic in other regions.
  - **Once the forwarder receives the message, it'll essentially mimic what would happen if that same *CreatePost* were called in another region, which will start the entire feed-update logic in those other regions.**
  - We can have some additional logic passed to the forwarder to prevent other regions being replicated to from notifying other regions about the *CreatePost* call in question, which would lead to an infinite chain of replications; in other words, **we can make it such that only the region where the post originated from is in charge of notifying other regions.**

Case Study : Video Hosting Service

- Functional Requirements:
  1. Users should be able to upload videos
  2. Users should be able to share and v
  3. Users should be able to perform se n video titles.
  4. Our services should be able to record stats of videos, e.g., likes/dislikes, total number of views, etc.
  5. Users should be able to add and view comments on videos.
- Non-Functional Requirements:
  1. The system should be highly reliable, any video uploaded should not be lost.
  2. The system should be highly available. Consistency can take a hit (in the interest of availability); if a user doesn't see a video for a while, it should be

Video Hosting Service (Contd..)

- Back of the envelop estimation

Let's assume we have 1.5 billion total users, 800 million of whom are daily active users. If, on average, a user views five videos per day then the total video-views per second would be:

800M * 5 / 86400 sec => 46K videos/sec

Let's assume our upload:view ratio is 1:200, i.e., for every video upload we have 200 videos viewed, giving us 230 videos uploaded per second.

46K / 200 => 230 videos/sec

Storage Estimates: Let's assume that every minute 500 hours worth of videos are uploaded to Youtube. If on average, one minute of video needs 50MB of storage (videos need to be stored in multiple formats), the total storage needed for videos uploaded in a minute would be:

500 hours * 60 min * 50MB => 1500 GB/min (25 GB/sec)

These are estimated numbers ignoring video compression and replication, which would change real numbers.

Bandwidth estimates: With 500 hours of video uploads per minute (which is 30000 mins of video uploads per minute), assuming uploading each minute of the video takes 10MB of the bandwidth, we would be getting 300GB of uploads every minute.

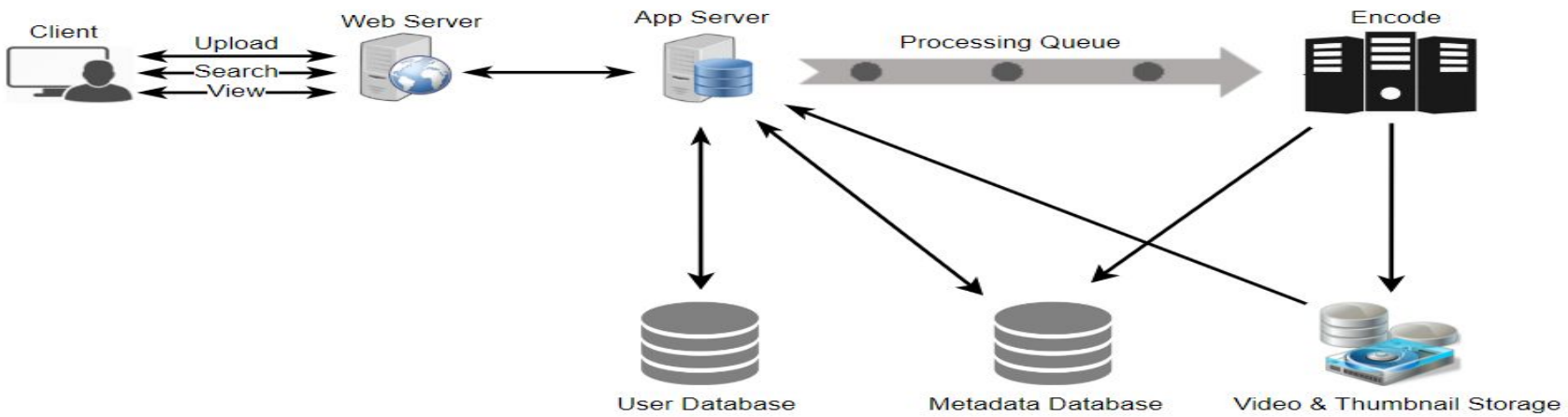
500 hours * 60 mins * 10MB => 300GB/min (5GB/sec)

Assuming an upload:view ratio of 1:200, we would need 1TB/s outgoing bandwidth.

Video Hosting Service (Contd..)

At a high-level we would need the following components:

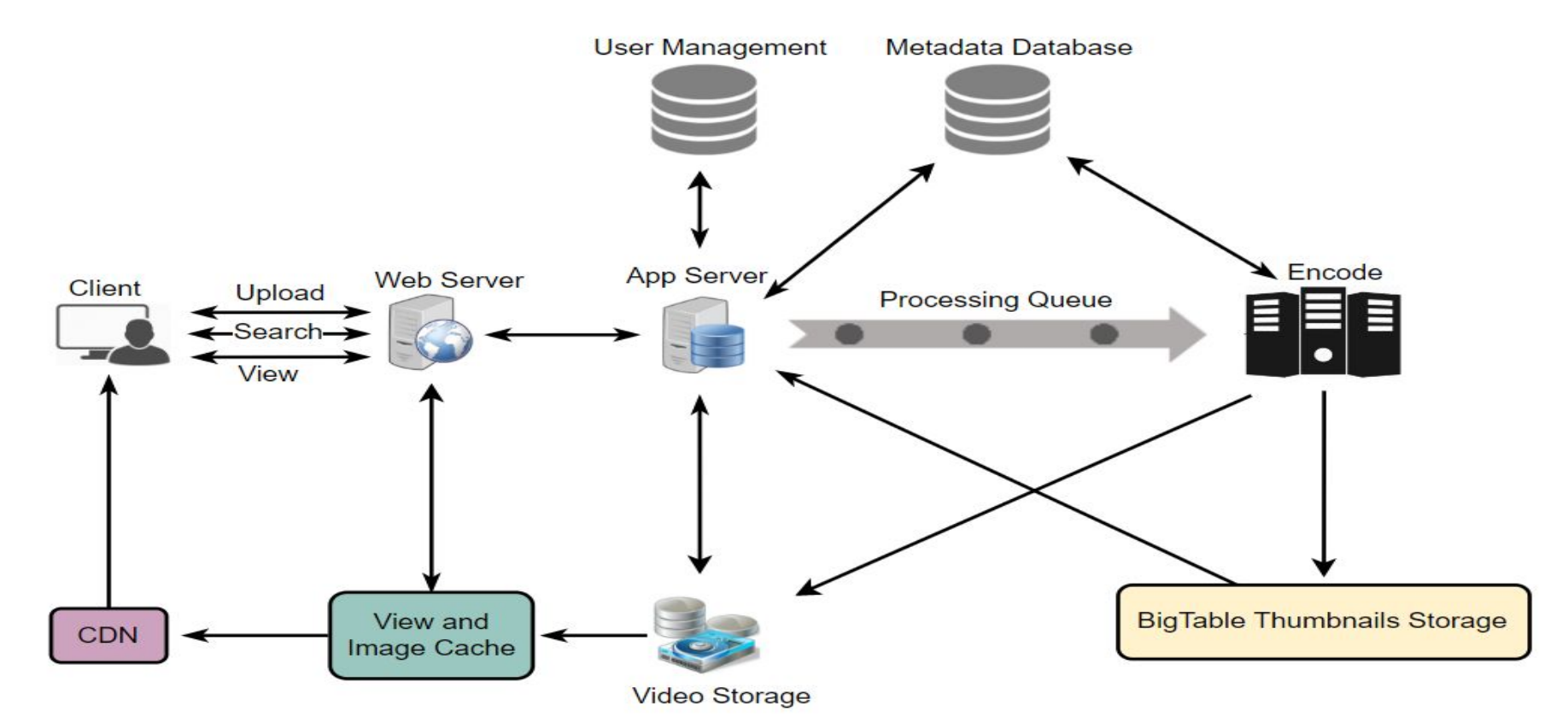
1. **Processing Queue:** Each uploaded video will be pushed to a processing queue to be de-queued later for encoding, thumbnail generation, and storage.
2. **Encoder:** To encode each uploaded video into multiple formats.
3. **Thumbnails generator:** To generate a few thumbnails for each video.
4. **Video and Thumbnail storage:** To store video and thumbnail files in some distributed file storage.
5. **User Database:** To store user's information, e.g., name, email, address, etc.
6. **Video metadata storage:** A metadata database to store all the information about videos like title, file path in the system, uploading user, total views, likes, dislikes, etc. It will also be used to store all the video comments.



Video Hosting Service (Contd..)

- **Detailed Component Design**
  - **The service would be read-heavy, so we will focus on building a system that can retrieve videos quickly. We can expect our read:write ratio to be 200:1, which means for every video upload, there are 200 video views.**
  - Where would videos be stored? **Videos can be stored in a distributed file storage system like HDFS**
  - **How should we efficiently manage read traffic?**
    - **We should segregate our read traffic from write traffic.**
    - **Since we will have multiple copies of each video, we can distribute our read traffic on different servers.**
    - **For metadata, we can have primary-secondary configurations where writes will go to primary first and then get applied at all the secondaries.**
  - **Where would thumbnails be stored?**
    - **Read traffic for thumbnails will be huge compared to videos. Users will be watching one video at a time, but they might be looking at a page with 20 thumbnails of other videos.**
    - **Let's evaluate storing all the thumbnails on a disk. Given that we have a huge number of files, we have to perform many seeks to different locations on the disk to read these files. This is quite inefficient and will result in higher latencies.**
    - **Bigtable can be a reasonable choice here as it combines**

Video Hosting Service (Contd..)



Detailed Component Design

Video Hosting Service (Contd..)

- **Metadata Sharding**
  - **Since we have a huge number of new videos every day and our read load is extremely high, therefore, we need to distribute our data onto multiple machines so that we can perform read/write operations efficiently.**
  - **We have many options to shard our data. Let's go through different strategies of sharding this data one by one:**
    - **Sharding based on UserID: We can try storing all the data for a particular user on one server. While storing, we can pass the UserID to our hash function, which will map the user to a database server where we will store all the metadata for that user's videos.**
      - **What if a user becomes popular? There could be a lot of queries on the server holding that user; this could create a performance bottleneck. This will also affect the overall performance of our service.**
      - **Over time, some users can end up storing a lot of videos compared to others. Maintaining a uniform distribution of growing user data is quite tricky.**
    - **Sharding based on VideoID: Our hash function will map each VideoID to a random server where we will store that Video's metadata. To find videos of a user, we will query all servers, and each server will return a set of videos.**
      - **A centralized server will aggregate and rank these results before returning them to the user.**
      - **This approach solves our problem of popular users but shifts it to popular videos.**

Video Hosting Service (Contd..)

- **Cache**
  - To serve globally distributed users, our service needs a massive-scale video delivery system.
  - Our service should push its content closer to the user using a large number of geographically distributed video cache servers.
  - We need to have a strategy that will maximize user performance and also evenly distributes the load on its cache servers.
  - We can introduce a cache for metadata servers to cache hot database rows.
  - Using Memcache to cache the data and Application servers before hitting the database can quickly check if the cache has the desired rows.
  - Least Recently Used (LRU) can be a

Video Hosting Service (Contd..)

- **Load Balancing**
  - We should use Consistent Hashing among our cache servers, which will also help in balancing the load between cache servers.
  - Since we will be using a static hash-based scheme to map videos to hostnames, it can lead to an uneven load on the logical replicas due to each video's different popularity.
  - For instance, if a video becomes popular, the logical replica corresponding to that video will experience more traffic than other servers.
  - These uneven loads for logical replicas can then translate into uneven load distribution on corresponding physical servers.
  - To resolve this issue, any busy server in one location can redirect a client to a less busy server in the same cache location. We can use dynamic HTTP redirections for this scenario.
  - However, the use of redirections also has its drawbacks. First, since our service tries to load balance locally, it leads to multiple redirections if the host that receives the redirection can't serve the video.
  - Also, each redirection requires a client to make

Video Hosting Service (Contd..)

- **Content Delivery Network (CDN)**
  - **Our service can move popular videos to CDNs.**
  - **CDNs replicate content in multiple places. There's a better chance of videos being closer to the user and, with fewer hops, videos will stream from a friendlier network.**
  - **CDN machines make heavy use of caching and can mostly serve videos out of memory.**
  - **Less popular videos (1-20 views per day) that are not cached by CDNs can be served by our servers in various data centers.**