

What is a System Design Interview?

The purpose of a system design interview is to assess a candidate's ability to design and understand complex systems. It's a crucial part of the hiring process for roles that involve system architecture and engineering, such as software engineers, system architects, and DevOps engineers. Here's a breakdown of what these interviews aim to evaluate:

1. Evaluating Technical Proficiency

- **Design Skills:** Assessing the ability to design a scalable, efficient, and robust system architecture.
- **Problem-solving Skills:** Gauging how you approach and solve complex, open-ended problems.
- **Technical Knowledge:** Understanding your familiarity with various technologies, databases, frameworks, and protocols.

2. Understanding Approach and Methodology

- **Requirements Gathering:** Your ability to understand and outline the requirements of a system before diving into the solution.
- **Balancing Trade-offs:** How you balance various trade-offs in system design, such as between scalability and cost, or performance and reliability.
- **Decision-making Process:** Evaluating your reasoning behind choosing certain technologies or architectures over others.

3. Testing Soft Skills

- **Communication:** Your ability to clearly articulate your thought process and design choices.
- **Collaboration:** Assessing how you interact with interviewers, which can mimic real-life collaboration with team members.
- **Adaptability:** Your response to new information or feedback during the interview, reflecting your ability to adapt in real-world projects.

4. Real-world Application

- **Practicality:** Ensuring that your designs can be practically implemented and are not just theoretical.
- **Scalability and Performance:** Your understanding of how the system would perform under real-world constraints and loads.

5. Innovation and Creativity

- **Creative Thinking:** Looking for innovative solutions and ideas that showcase your ability to think outside the box.
- **Future-Proofing:** How you design systems with future growth, maintenance, and potential challenges in mind.

What System Design Interview is Not About

- It's not about getting the perfect answer. In fact, there often isn't one "correct" solution in system design.
- It's not a test of memorization. While familiarity with certain tools and technologies is helpful, understanding the concepts is more important.

Conclusion

The system design interview is a critical tool for employers to assess not just your technical abilities, but also your problem-solving approach, communication skills, and overall fit for roles that require designing and working with large-scale systems. It's a comprehensive evaluation of how you would handle real-life challenges in system architecture and engineering.

Functional vs. Non-functional Requirements

In the context of system design interviews, understanding functional and non-functional requirements is key to showcasing your ability to design a system that meets both the specific actions it should perform and how it should perform them.

Functional Requirements

- **Definition:** These are the requirements that define what a system is supposed to do. They describe the various functions that the system must perform.
- **Examples:**
 - A user authentication system must validate user credentials and provide access levels.
 - An e-commerce website should allow users to browse products, add them to a cart, and complete purchases.
 - A report generation system must collect data, process it, and generate timely reports.

Importance in Interviews

- **Demonstrates Understanding of Core Features:** Shows you know what the system needs to do to satisfy its primary objectives.
- **Basis for System Design:** Functional requirements often form the backbone of your system design.

Non-Functional Requirements

- **Definition:** These requirements describe how the system performs a task, rather than what tasks it performs. They are related to the quality attributes of the system.
- **Examples:**
 - Scalability: The system should handle growth in users or data.
 - Performance: The system should process transactions within a specified time.
 - Availability: The system should be up and running a defined percentage of time.
 - Security: The system must protect sensitive data and resist unauthorized access.

Importance in Interviews

- **Showcases Depth of Design Knowledge:** Demonstrates your understanding of the broader implications of system design.
- **Highlights System Robustness and Quality:** Reflects how well your system design can meet real-world constraints and user expectations.

Integrating Both in Interviews

- **Scenario-Based Discussions:** When presented with a scenario, identify both the functional (what the system should do) and non-functional (how the system should do it) requirements.
- **Balancing Act:** Exhibit your ability to balance both types of requirements, showing that you can design a system that not only meets its functional goals but also performs effectively, securely, and reliably.

In System Design Interviews

When you're in a system design interview, here's how you can handle these requirements:

- 1. Clarify Requirements:** Start by asking questions to understand both functional and non-functional requirements. Interviewers often leave these vague to see if you'll probe for more details.
- 2. Prioritize:** Not all requirements are equally important. Identify which ones are critical for the system's success.
- 3. Trade-offs:** Discuss trade-offs related to different architectural decisions, especially concerning non-functional requirements. For example, a system highly optimized for read operations might have slower write operations.
- 4. Use Real-World Examples:** If you can, relate your points to real-world systems or your past experiences. This shows practical understanding.
- 5. Balance:** Ensure you're not focusing too much on one type of requirement over the other. A well-rounded approach is often necessary.

Remember, in system design interviews, interviewers are often interested in seeing how you think and approach problems, not just your final solution. Demonstrating a clear understanding of both functional and non-functional requirements is key to showing your comprehensive knowledge in system design.

What are Back-of-the-Envelope Estimations?

Back of the envelope estimations in system design interviews are like quick, rough calculations you might do on a napkin during lunch - they're not detailed or exact, but give you a good ballpark figure. These rough calculations help you quickly assess the feasibility of a proposed solution, estimate its performance, and identify potential bottlenecks.

Purpose

Back-of-the-envelope estimation is a technique used to quickly approximate values and make rough calculations using simple arithmetic and basic assumptions. This method is particularly useful in system design interviews, where interviewers expect candidates to make informed decisions and trade-offs based on rough estimates.

Why is Estimation Important in System Design Interviews?

During a system design interview, you'll be asked to design a scalable and reliable system based on a set of requirements. Your ability to make quick estimations is essential for several reasons:

- 1. Indicates System Scalability:** Highlights your understanding of how the system can grow or adapt.
- 2. Validate proposed solutions:** Estimation helps you ensure that your proposed architecture meets the requirements and can handle the expected load.
- 3. Identify bottlenecks:** Quick calculations help you identify potential performance bottlenecks and make necessary adjustments to your design.

4. **Demonstrate your thought process:** Estimation showcases your ability to make informed decisions and trade-offs based on a set of assumptions and constraints.
5. **Communicate effectively:** Providing estimates helps you effectively communicate your design choices and their implications to the interviewer.
6. **Quick Decision Making:** Reflects your ability to make swift estimations to guide your design decisions.

Estimation Techniques

1. Rule of thumb

Rules of thumb are general guidelines or principles that can be applied to make quick and reasonably accurate estimations. They are based on experience and observation, and while not always precise, they can provide valuable insights in the absence of detailed information. For example, estimating that a user will generate 1 MB of data per day on a social media platform can serve as a starting point for capacity planning.

2. Approximation

Approximation involves simplifying complex calculations by rounding numbers or using easier-to-compute values. This technique can help derive rough estimates quickly and with minimal effort. For instance, assuming 1,000 users instead of 1,024 when estimating storage requirements can simplify calculations and still provide a reasonable approximation.

3. Breakdown and aggregation

Breaking down a problem into smaller components and estimating each separately can make it easier to derive an overall estimate. This technique involves identifying the key components of a system, estimating their individual requirements, and then aggregating these estimates to determine the total system requirements. For example, estimating the storage needs for user data, multimedia content, and metadata separately can help in determining the overall storage requirements of a social media platform.

4. Sanity check

A sanity check is a quick evaluation of an estimate to ensure its plausibility and reasonableness. This step helps identify potential errors or oversights in the estimation process and can lead to more accurate and reliable results. For example, comparing the estimated storage requirements for a messaging service with the actual storage used by a similar existing service can help validate the estimate.

Types of Estimations in System Design Interviews

In system design interviews, there are several types of estimations you may need to make:

1. **Load estimation:** Predict the expected number of requests per second, data volume, or user traffic for the system.
2. **Storage estimation:** Estimate the amount of storage required to handle the data generated by the system.
3. **Bandwidth estimation:** Determine the network bandwidth needed to support the expected traffic and data transfer.
4. **Latency estimation:** Predict the response time and latency of the system based on its architecture and components.

5. **Resource estimation:** Estimate the number of servers, CPUs, or memory required to handle the load and maintain desired performance levels.

Process

1. **Understand the Scope:** Clarify the scale of the problem - how many users, how much data, etc.
2. **Use Simple Math:** Utilize basic arithmetic to estimate the scale of data and resources.
3. **Round Numbers for Simplicity:** Use round numbers to make calculations easier and faster.
4. **Be Logical and Reasonable:** Ensure your estimations make sense given the context of the problem.

Practical Examples

1. Load Estimation

Suppose you're asked to design a social media platform with 100 million daily active users (DAU) and an average of 10 posts per user per day. To estimate the load, you'd calculate the total number of posts generated daily:

$$100 \text{ million DAU} * 10 \text{ posts/user} = 1 \text{ billion posts/day}$$

Then, you can estimate the request rate per second:

$$1 \text{ billion posts/day} / 86,400 \text{ seconds/day} \approx 11,574 \text{ requests/second}$$

2. Storage Estimation

Consider a photo-sharing app with 500 million users and an average of 2 photos uploaded per user per day. Each photo has an average size of 2 MB. To estimate the storage required for one day's worth of photos, you'd calculate:

$$500 \text{ million users} * 2 \text{ photos/user} * 2 \text{ MB/photo} = 2,000,000,000 \text{ MB/day}$$

3. Bandwidth Estimation

For a video streaming service with 10 million users streaming 1080p videos at 4 Mbps, you can estimate the required bandwidth:

$$10 \text{ million users} * 4 \text{ Mbps} = 40,000,000 \text{ Mbps}$$

4. Latency Estimation

Suppose you're designing an API that fetches data from multiple sources, and you know that the average latency for each source is 50 ms, 100 ms, and 200 ms, respectively. If the data fetching process is sequential, you can estimate the total latency as follows:

$$50 \text{ ms} + 100 \text{ ms} + 200 \text{ ms} = 350 \text{ ms}$$

If the data fetching process is parallel, the total latency would be the maximum latency among the sources:

$$\max(50 \text{ ms}, 100 \text{ ms}, 200 \text{ ms}) = 200 \text{ ms}$$

5. Resource Estimation

Imagine you're designing a web application that receives 10,000 requests per second, with each request requiring 10 ms of CPU time. To estimate the number of CPU cores needed, you can calculate the total CPU time per second:

$$10,000 \text{ requests/second} * 10 \text{ ms/request} = 100,000 \text{ ms/second}$$

Assuming each CPU core can handle 1,000 ms of processing per second, the number of cores required would be:

$$100,000 \text{ ms/second} / 1,000 \text{ ms/core} = 100 \text{ cores}$$

System Design Examples

1. Designing a messaging service

Imagine you are tasked with designing a messaging service similar to WhatsApp. To estimate the system's requirements, you can start by considering the following aspects:

- **Number of users:** Estimate the total number of users for the platform. This can be based on market research, competitor analysis, or historical data.
- **Messages per user per day:** Estimate the average number of messages sent by each user per day. This can be based on user behavior patterns or industry benchmarks.
- **Message size:** Estimate the average size of a message, considering text, images, videos, and other media content.
- **Storage requirements:** Calculate the total storage needed to store messages for a specified retention period, taking into account the number of users, messages

per user, message size, and data redundancy.

- **Bandwidth requirements:** Estimate the bandwidth needed to handle the message traffic between users, considering the number of users, messages per user, and message size.

By breaking down the problem into smaller components and applying estimation techniques, you can derive a rough idea of the messaging service's requirements, which can guide your design choices and resource allocation.

2. Designing a video streaming platform

Suppose you are designing a video streaming platform similar to Netflix. To estimate the system's requirements, consider the following aspects:

- **Number of users:** Estimate the total number of users for the platform based on market research, competitor analysis, or historical data.
- **Concurrent users:** Estimate the number of users who will be streaming videos simultaneously during peak hours.
- **Video size and bitrate:** Estimate the average size and bitrate of videos on the platform, considering various resolutions and encoding formats.
- **Storage requirements:** Calculate the total storage needed to store the video content, taking into account the number of videos, their sizes, and data redundancy.
- **Bandwidth requirements:** Estimate the bandwidth needed to handle the video streaming traffic, considering the number of concurrent users, video bitrates, and user locations.

By applying estimation techniques and aggregating the individual estimates, you can get a ballpark figure of the video streaming platform's requirements, which can inform your design decisions and resource allocation.

Tips for Successful Estimation in Interviews

Estimation plays a crucial role in system design interviews, as it helps you make informed decisions about your design and demonstrates your understanding of the various factors that impact the performance and scalability of a system. Here are some tips to help you ace the estimation part of your interviews:

1. Break down the problem

When faced with a complex system design problem, break it down into smaller, more manageable components. This will make it easier to estimate each component's requirements and help you understand how they interact with each other. By identifying the key components and estimating their requirements separately, you can then aggregate your estimates to get a comprehensive view of the system's needs.

2. Use reasonable assumptions

During an interview, you may not have all the necessary information to make precise estimations. In such cases, make reasonable assumptions based on your knowledge of similar systems, industry standards, or user behavior patterns. Clearly state your assumptions to the interviewer, as this demonstrates your thought process and enables them to provide feedback or correct your assumptions if necessary.

Operation name	Time
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns = 10 µs
Send 2K bytes over 1 Gbps network	20,000 ns = 20 µs
Read 1 MB sequentially from memory	250,000 ns = 250 µs
Round trip within the same datacenter	500,000 ns = 500 µs
Disk seek	10,000,000 ns = 10 ms
Read 1 MB sequentially from network	10,000,000 ns = 10 ms
Read 1 MB sequentially from disk	30,000,000 ns = 30 ms
Send packet CA->Netherlands->CA	150,000,000 ns = 150 ms

3. Leverage your experience

Drawing from your past experiences can be beneficial when estimating system requirements. If you have worked on similar systems or have experience with certain technologies, use that knowledge to inform your estimations. This will not only help you make more accurate estimations but also showcase your expertise to the interviewer.

4. Be prepared to adjust your estimations

As you progress through the interview, the interviewer may provide additional information or challenge your assumptions, requiring you to adjust your estimations. Be prepared to adapt and revise your estimations accordingly. This demonstrates your ability to think critically and shows that you can handle changing requirements in a real-world scenario.

5. Don't Forget to Ask Clarifying Questions

Don't hesitate to ask the interviewer clarifying questions if you're unsure about a requirement or assumption. This will help you avoid making incorrect estimations and showcase your problem-solving abilities.

6. Communicate your thought process

Throughout the estimation process, communicate your thought process clearly to the interviewer. Explain how you arrived at your estimations and the assumptions you made along the way. This allows the interviewer to understand your reasoning, provide feedback, and assess your problem-solving skills.

Conclusion

Back of the envelope estimations are crucial in system design interviews as they showcase your ability to grasp the scale of a system quickly and assess the feasibility and resource needs of your design. It's a skill that demonstrates both technical knowledge and practical problem-solving ability.

Things to Avoid During System Design Interview

In a system design interview, while it's important to showcase your skills and knowledge, it's equally crucial to be aware of common pitfalls. Avoiding these mistakes can greatly improve your chances of success. Here are some key "don'ts" for a system design interview:

1. Don't Ignore the Requirements

- **Neglecting to Clarify:** Failing to ask questions or clarify requirements can lead to a design that misses the mark.
- **Oversimplifying the Problem:** Don't oversimplify the problem or ignore the complexities involved.

2. Don't Dive into Details Too Soon

- **Rushing into Low-Level Details:** Starting with low-level details before establishing the high-level design can make your solution seem disjointed.
- **Losing Sight of the Big Picture:** Focus first on the overall architecture and how different components interact.

3. Don't Stick Rigidly to One Idea

- **Being Inflexible:** Being too rigid with your initial idea can prevent you from considering better alternatives.
- **Ignoring Interviewer's Hints:** The interviewer might provide hints or feedback; not adapting your design accordingly can be seen as a lack of collaboration or adaptability.

4. Don't Overlook Trade-offs

- **Ignoring Trade-offs:** Every design decision has trade-offs. Not discussing these can show a lack of depth in your understanding.
- **Failing to Justify Decisions:** Be prepared to explain why you chose one approach over another.

5. Don't Neglect Non-Functional Requirements

- **Overlooking Scalability, Reliability, etc.:** Focusing solely on functional aspects and neglecting non-functional requirements like scalability and reliability is a common mistake.
- **Not Considering System Constraints:** Real-world constraints such as cost, time, and existing technology stack should be considered.

6. Don't Under-Communicate

- **Poor Explanation:** Failing to clearly articulate your thoughts can leave interviewers unsure about your understanding and approach.
- **Not Engaging the Interviewer:** This is a dialogue, not a monologue. Engage with the interviewer, ask questions, and be receptive to feedback.

7. Don't Be Overconfident or Arrogant

- **Overconfidence:** Being overly confident can lead to dismissing valuable feedback or overlooking key aspects of the problem.
- **Not Acknowledging What You Don't Know:** It's okay not to know everything. Being open about areas you are unsure of is better than providing incorrect information.

Conclusion

A system design interview is not just about getting the right answer. It's about demonstrating your problem-solving approach, your ability to adapt, and how you communicate and collaborate. Avoiding these pitfalls can help you present yourself as a well-rounded candidate capable of handling the complexities of real-world system design.

System Design Basics

Whenever we are designing a large system, we need to consider a few things:

1. What are the different architectural pieces that can be used?
2. How do these pieces work with each other?
3. How can we best utilize these pieces: what are the right tradeoffs?

Investing in scaling before it is needed is generally not a smart business proposition; however, some forethought into the design can save valuable time and resources in the future. In the following chapters, we will try to define some of the core building blocks of scalable systems. Familiarizing these concepts would greatly benefit in understanding distributed system concepts. In the next section, we will go through Consistent Hashing, CAP Theorem, Load Balancing, Caching, Data Partitioning, Indexes, Proxies, Queues, Replication, and choosing between SQL vs. NoSQL.

Let's start with the Key Characteristics of Distributed Systems.

Key Characteristics of Distributed Systems

Key characteristics of a distributed system include Scalability, Reliability, Availability, Efficiency, and Manageability. Let's briefly review them:

Scalability

Scalability is the capability of a system, process, or a network to grow and manage increased demand. Any distributed system that can continuously evolve in order to support the growing amount of work is considered to be scalable.

A system may have to scale because of many reasons like increased data volume or increased amount of work, e.g., number of transactions. A scalable system would like to achieve this scaling without performance loss.

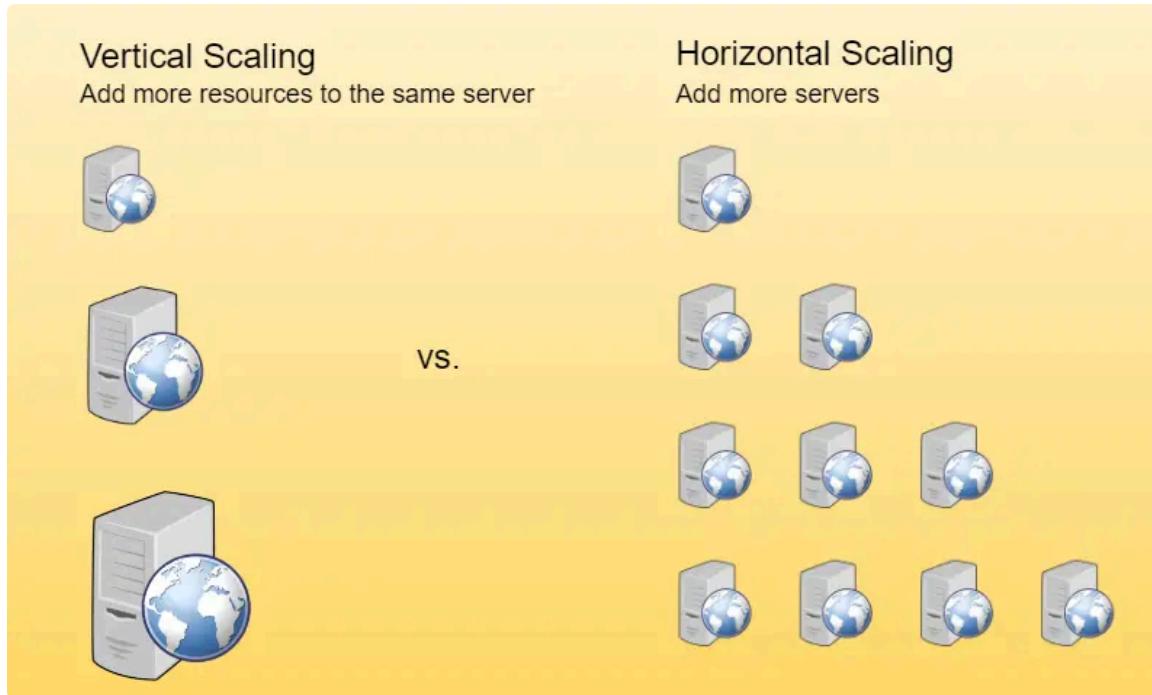
Generally, the performance of a system, although designed (or claimed) to be scalable, declines with the system size due to the management or environment cost. For instance, network speed may become slower because machines tend to be far apart from one another. More generally, some tasks may not be distributed, either because of their inherent atomic nature or because of some flaw in the system design. At some point, such tasks would limit the speed-up obtained by distribution. A scalable architecture avoids this situation and attempts to balance the load on all the participating nodes evenly.

Horizontal vs. Vertical Scaling: Horizontal scaling means that you scale by adding more servers into your pool of resources whereas Vertical scaling means that you scale by adding more power (CPU, RAM, Storage, etc.) to an existing server.

With horizontal-scaling it is often easier to scale dynamically by adding more machines into the existing pool; Vertical-scaling is usually limited to the capacity

of a single server and scaling beyond that capacity often involves downtime and comes with an upper limit.

Good examples of horizontal scaling are [Cassandra](#) and [MongoDB](#) as they both provide an easy way to scale horizontally by adding more machines to meet growing needs. Similarly, a good example of vertical scaling is MySQL as it allows for an easy way to scale vertically by switching from smaller to bigger machines. However, this process often involves downtime.



Vertical scaling vs. Horizontal scaling

Reliability

Reliability refers to the ability of a system to continue operating correctly and effectively in the presence of faults, errors, or failures. In simple terms, a distributed system is considered reliable if it keeps delivering its services even when one or several of its software or hardware components fail. Reliability represents one of the main characteristics of any distributed system, since in such

systems any failing machine can always be replaced by another healthy one, ensuring the completion of the requested task.

Take the example of a large electronic commerce store (like [Amazon](#)), where one of the primary requirement is that any user transaction should never be canceled due to a failure of the machine that is running that transaction. For instance, if a user has added an item to their shopping cart, the system is expected not to lose it. A reliable distributed system achieves this through redundancy of both the software components and data. If the server carrying the user's shopping cart fails, another server that has the exact replica of the shopping cart should replace it.

Obviously, redundancy has a cost and a reliable system has to pay that to achieve such resilience for services by eliminating every single point of failure.

Availability

By definition, availability is the time a system remains operational to perform its required function in a specific period. It is a simple measure of the percentage of time that a system, service, or a machine remains operational under normal conditions. An aircraft that can be flown for many hours a month without much downtime can be said to have a high availability. Availability takes into account maintainability, repair time, spares availability, and other logistics considerations. If an aircraft is down for maintenance, it is considered not available during that time.

Reliability is availability over time considering the full range of possible real-world conditions that can occur. An aircraft that can make it through any possible weather safely is more reliable than one that has vulnerabilities to possible conditions.

Reliability Vs. Availability

If a system is reliable, it is available. However, if it is available, it is not necessarily reliable. In other words, high reliability contributes to high availability, but it is possible to achieve a high availability even with an unreliable product by minimizing repair time and ensuring that spares are always available when they are needed. Let's take the example of an online retail store that has 99.99% availability for the first two years after its launch. However, the system was launched without any information security testing. The customers are happy with the system, but they don't realize that it isn't very reliable as it is vulnerable to likely risks. In the third year, the system experiences a series of information security incidents that suddenly result in extremely low availability for extended periods of time. This results in reputational and financial damage to the customers.

Efficiency

To understand how to measure the efficiency of a distributed system, let's assume we have an operation that runs in a distributed manner and delivers a set of items as result. Two standard measures of its efficiency are the response time (or latency) that denotes the delay to obtain the first item and the throughput (or bandwidth) which denotes the number of items delivered in a given time unit (e.g., a second). The two measures correspond to the following unit costs:

- Number of messages globally sent by the nodes of the system regardless of the message size.
- Size of messages representing the volume of data exchanges.

The complexity of operations supported by distributed data structures (e.g., searching for a specific key in a distributed index) can be characterized as a function of one of these cost units. Generally speaking, the analysis of a distributed

structure in terms of 'number of messages' is over-simplistic. It ignores the impact of many aspects, including the network topology, the network load, and its variation, the possible heterogeneity of the software and hardware components involved in data processing and routing, etc. However, it is quite difficult to develop a precise cost model that would accurately take into account all these performance factors; therefore, we have to live with rough but robust estimates of the system behavior.

Serviceability or Manageability

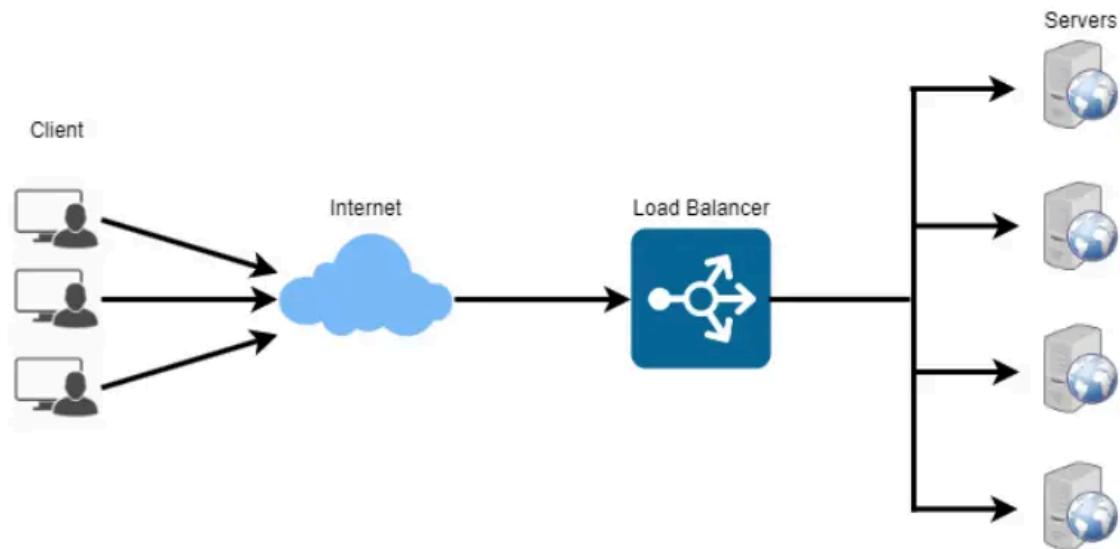
Another important consideration while designing a distributed system is how easy it is to operate and maintain. Serviceability or manageability is the simplicity and speed with which a system can be repaired or maintained; if the time to fix a failed system increases, then availability will decrease. Things to consider for manageability are the ease of diagnosing and understanding problems when they occur, ease of making updates or modifications, and how simple the system is to operate (i.e., does it routinely operate without failure or exceptions?).

Early detection of faults can decrease or avoid system downtime. For example, some enterprise systems can automatically call a service center (without human intervention) when the system experiences a system fault.

Load Balancing

Load Balancer (LB) is another critical component of any distributed system. It helps to spread the traffic across a cluster of servers to improve responsiveness and availability of applications, websites or databases. LB also keeps track of the status of all the resources while distributing requests. If a server is not available to take new requests or is not responding or has elevated error rate, LB will stop sending traffic to such a server.

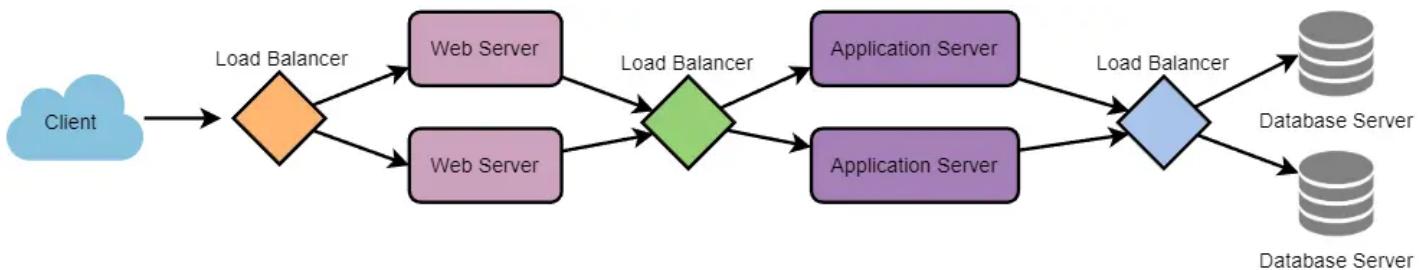
Typically a load balancer sits between the client and the server accepting incoming network and application traffic and distributing the traffic across multiple backend servers using various algorithms. By balancing application requests across multiple servers, a load balancer reduces individual server load and prevents any one application server from becoming a single point of failure, thus improving overall application availability and responsiveness.



To utilize full scalability and redundancy, we can try to balance the load at each layer of the system. We can add LBs at three places:

- Between the user and the web server

- Between web servers and an internal platform layer, like application servers or cache servers
- Between internal platform layer and database.



Benefits of Load Balancing

- Users experience faster, uninterrupted service. Users won't have to wait for a single struggling server to finish its previous tasks. Instead, their requests are immediately passed on to a more readily available resource.
- Service providers experience less downtime and higher throughput. Even a full server failure won't affect the end user experience as the load balancer will simply route around it to a healthy server.
- Load balancing makes it easier for system administrators to handle incoming requests while decreasing wait time for users.
- Smart load balancers provide benefits like predictive analytics that determine traffic bottlenecks before they happen. As a result, the smart load balancer gives an organization actionable insights. These are key to automation and can help drive business decisions.
- System administrators experience fewer failed or stressed components. Instead of a single device performing a lot of work, load balancing has several devices perform a little bit of work.

Load Balancing Algorithms

How does the load balancer choose the backend server?

Load balancers consider two factors before forwarding a request to a backend server. They will first ensure that the server they choose is actually responding appropriately to requests and then use a pre-configured algorithm to select one from the set of healthy servers. We will discuss these algorithms shortly.

Health Checks - Load balancers should only forward traffic to "healthy" backend servers. To monitor the health of a backend server, "health checks" regularly attempt to connect to backend servers to ensure that servers are listening. If a server fails a health check, it is automatically removed from the pool, and traffic will not be forwarded to it until it responds to the health checks again.

There is a variety of load balancing methods, which use different algorithms for different needs.

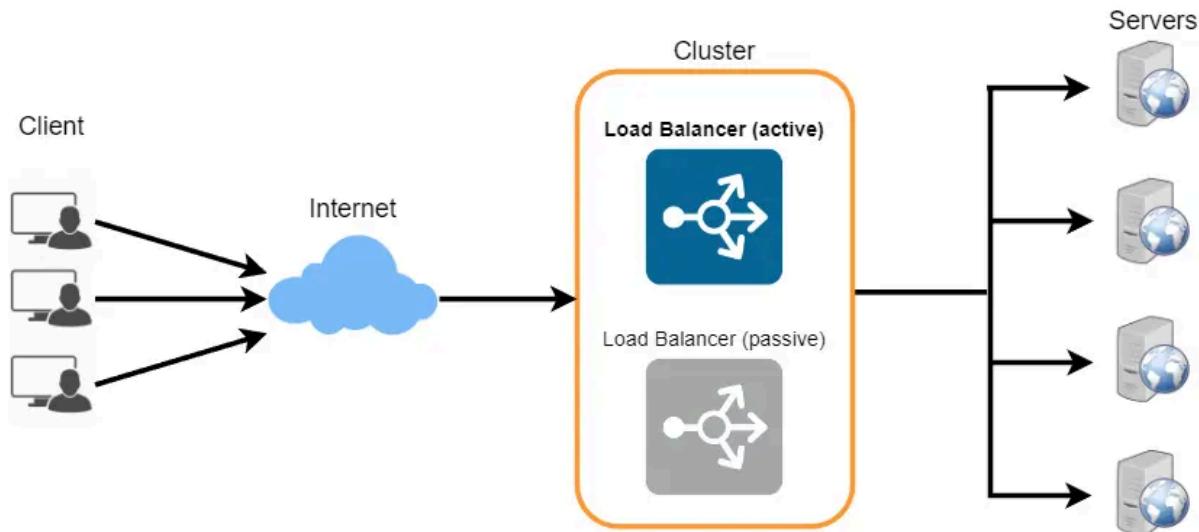
- **Least Connection Method** – This method directs traffic to the server with the fewest active connections. This approach is quite useful when there are a large number of persistent client connections which are unevenly distributed between the servers.
- **Least Response Time Method** – This algorithm directs traffic to the server with the fewest active connections and the lowest average response time.
- **Least Bandwidth Method** - This method selects the server that is currently serving the least amount of traffic measured in megabits per second (Mbps).
- **Round Robin Method** – This method cycles through a list of servers and sends each new request to the next server. When it reaches the end of the list, it starts over at the beginning. It is most useful when the servers are of equal specification and there are not many persistent connections.
- **Weighted Round Robin Method** – The weighted round-robin scheduling is designed to better handle servers with different processing capacities. Each server is assigned a weight (an integer value that indicates the processing capacity). Servers with higher weights receive new connections before those

with less weights and servers with higher weights get more connections than those with less weights.

- **IP Hash** – Under this method, a hash of the IP address of the client is calculated to redirect the request to a server.

Redundant Load Balancers

The load balancer can be a single point of failure; to overcome this, a second load balancer can be connected to the first to form a cluster. Each LB monitors the health of the other and, since both of them are equally capable of serving traffic and failure detection, in the event the main load balancer fails, the second load balancer takes over.



Following links have some good discussion about load balancers:

- [1] [What is load balancing](#)
- [2] [Introduction to architecting systems](#)
- [3] [Load balancing](#)

Caching

Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have as well as making otherwise unattainable product requirements feasible. Caches take advantage of the locality of reference principle: recently requested data is likely to be requested again. They are used in almost every computing layer: hardware, operating systems, web browsers, web applications, and more. A cache is like short-term memory: it has a limited amount of space, but is typically faster than the original data source and contains the most recently accessed items. Caches can exist at all levels in architecture, but are often found at the level nearest to the front end, where they are implemented to return data quickly without taxing downstream levels.

Application server cache

Placing a cache directly on a request layer node enables the local storage of response data. Each time a request is made to the service, the node will quickly return locally cached data if it exists. If it is not in the cache, the requesting node will fetch the data from the disk. The cache on one request layer node could also be located both in memory (which is very fast) and on the node's local disk (faster than going to network storage).

What happens when you expand this to many nodes? If the request layer is expanded to multiple nodes, it's still quite possible to have each node host its own cache. However, if your load balancer randomly distributes requests across the nodes, the same request will go to different nodes, thus increasing cache misses. Two choices for overcoming this hurdle are global caches and distributed caches.

Content Delivery (or Distribution) Network (CDN)

CDNs are a kind of cache that comes into play for sites serving large amounts of static media. In a typical CDN setup, a request will first ask the CDN for a piece of static media; the CDN will serve that content if it has it locally available. If it isn't available, the CDN will query the back-end servers for the file, cache it locally, and serve it to the requesting user.

If the system we are building is not large enough to have its own CDN, we can ease a future transition by serving the static media off a separate subdomain (e.g., static.yourservice.com) using a lightweight HTTP server like Nginx, and cut-over the DNS from your servers to a CDN later.

Cache Invalidation

While caching is fantastic, it requires some maintenance to keep the cache coherent with the source of truth (e.g., database). If the data is modified in the database, it should be invalidated in the cache; if not, this can cause inconsistent application behavior.

Solving this problem is known as cache invalidation; there are three main schemes that are used:

Write-through cache: Under this scheme, data is written into the cache and the corresponding database simultaneously. The cached data allows for fast retrieval and, since the same data gets written in the permanent storage, we will have complete data consistency between the cache and the storage. Also, this scheme ensures that nothing will get lost in case of a crash, power failure, or other system disruptions.

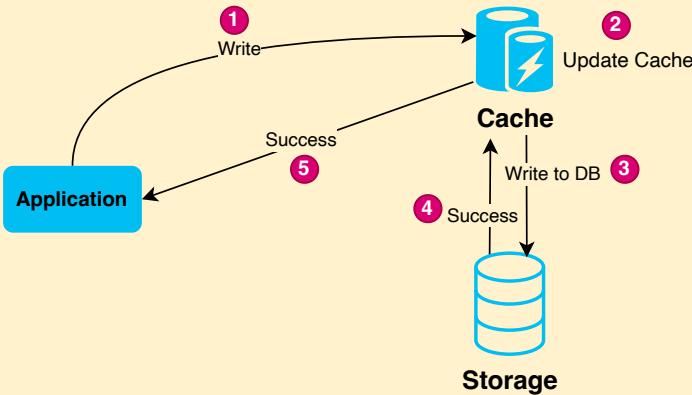
Although, write-through minimizes the risk of data loss, since every write operation must be done twice before returning success to the client, this scheme has the disadvantage of higher latency for write operations.

Write-around cache: This technique is similar to write-through cache, but data is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write operations that will not subsequently be re-read, but has the disadvantage that a read request for recently written data will create a “cache miss” and must be read from slower back-end storage and experience higher latency.

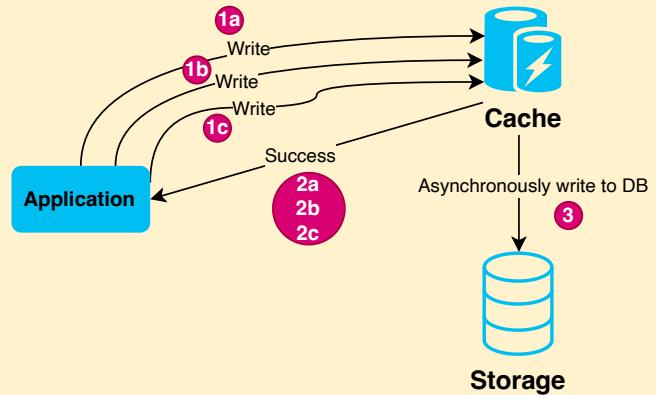
Write-back cache: Under this scheme, data is written to cache alone, and completion is immediately confirmed to the client. The write to the permanent storage is done after specified intervals or under certain conditions. This results in low-latency and high-throughput for write-intensive applications; however, this speed comes with the risk of data loss in case of a crash or other adverse event because the only copy of the written data is in the cache.

Cache Write Strategies

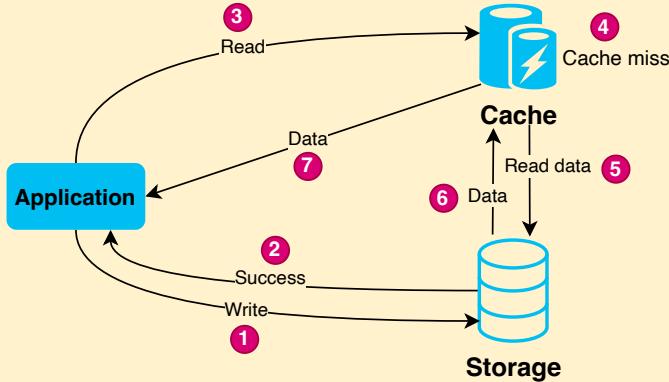
Write Through



Write Back



Write Around



Cache Write Strategies

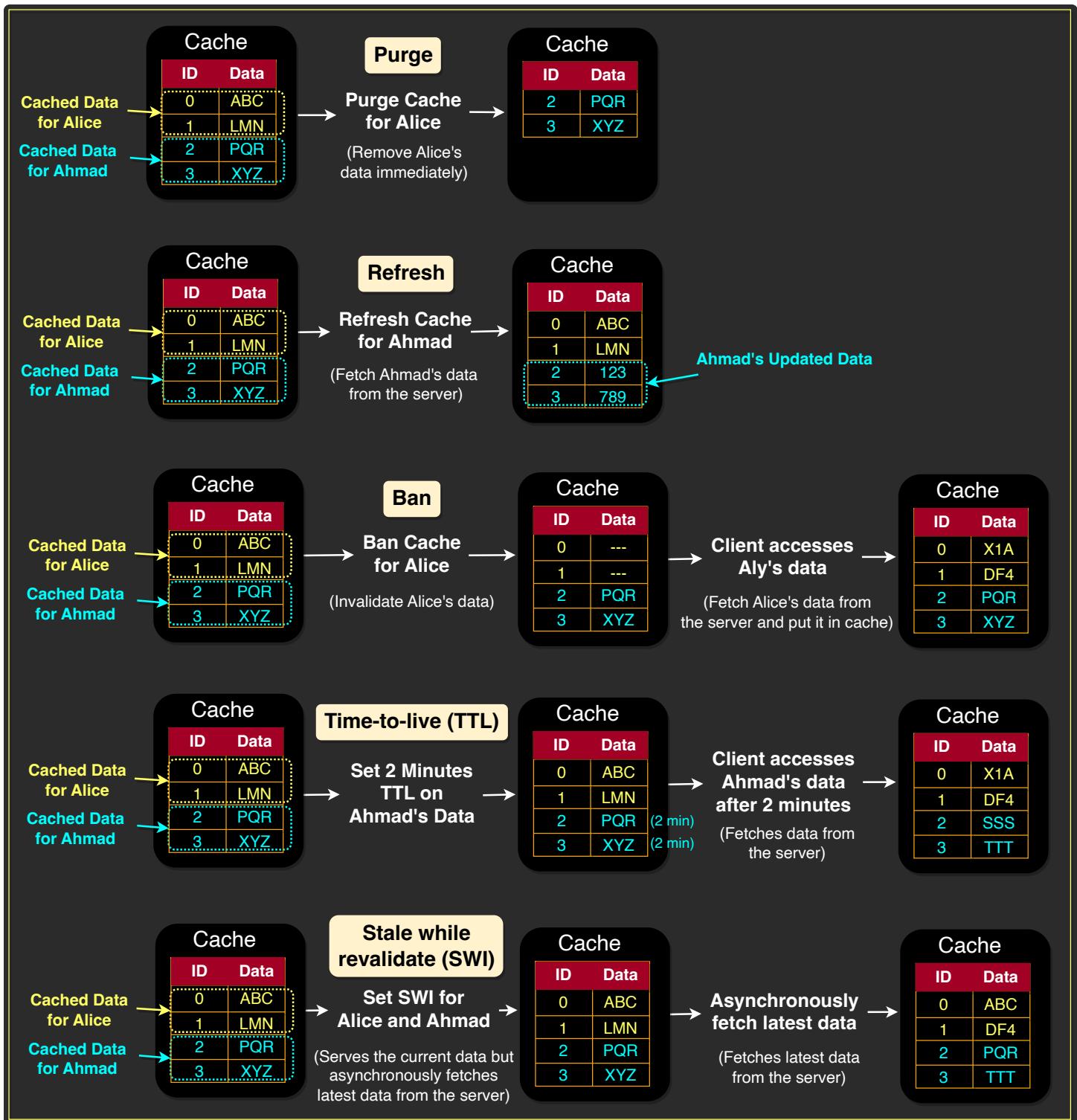
Cache Invalidation Methods

Here are the famous cache invalidation methods:

- **Purge:** The purge method removes cached content for a specific object, URL, or a set of URLs. It's typically used when there is an update or change to the

content and the cached version is no longer valid. When a purge request is received, the cached content is immediately removed, and the next request for the content will be served directly from the origin server.

- **Refresh:** Fetches requested content from the origin server, even if cached content is available. When a refresh request is received, the cached content is updated with the latest version from the origin server, ensuring that the content is up-to-date. Unlike a purge, a refresh request doesn't remove the existing cached content; instead, it updates it with the latest version.
- **Ban:** The ban method invalidates cached content based on specific criteria, such as a URL pattern or header. When a ban request is received, any cached content that matches the specified criteria is immediately removed, and subsequent requests for the content will be served directly from the origin server.
- **Time-to-live (TTL) expiration:** This method involves setting a time-to-live value for cached content, after which the content is considered stale and must be refreshed. When a request is received for the content, the cache checks the time-to-live value and serves the cached content only if the value hasn't expired. If the value has expired, the cache fetches the latest version of the content from the origin server and caches it.
- **Stale-while-revalidate:** This method is used in web browsers and CDNs to serve stale content from the cache while the content is being updated in the background. When a request is received for a piece of content, the cached version is immediately served to the user, and an asynchronous request is made to the origin server to fetch the latest version of the content. Once the latest version is available, the cached version is updated. This method ensures that the user is always served content quickly, even if the cached version is slightly outdated.



Cache Invalidation Methods

Cache read strategies

Here are the two famous cache read strategies:

Read through cache

A read-through cache strategy is a caching mechanism where the cache itself is responsible for retrieving the data from the underlying data store when a cache miss occurs. In this strategy, the application requests data from the cache instead of the data store directly. If the requested data is not found in the cache (cache miss), the cache retrieves the data from the data store, updates the cache with the retrieved data, and returns the data to the application.

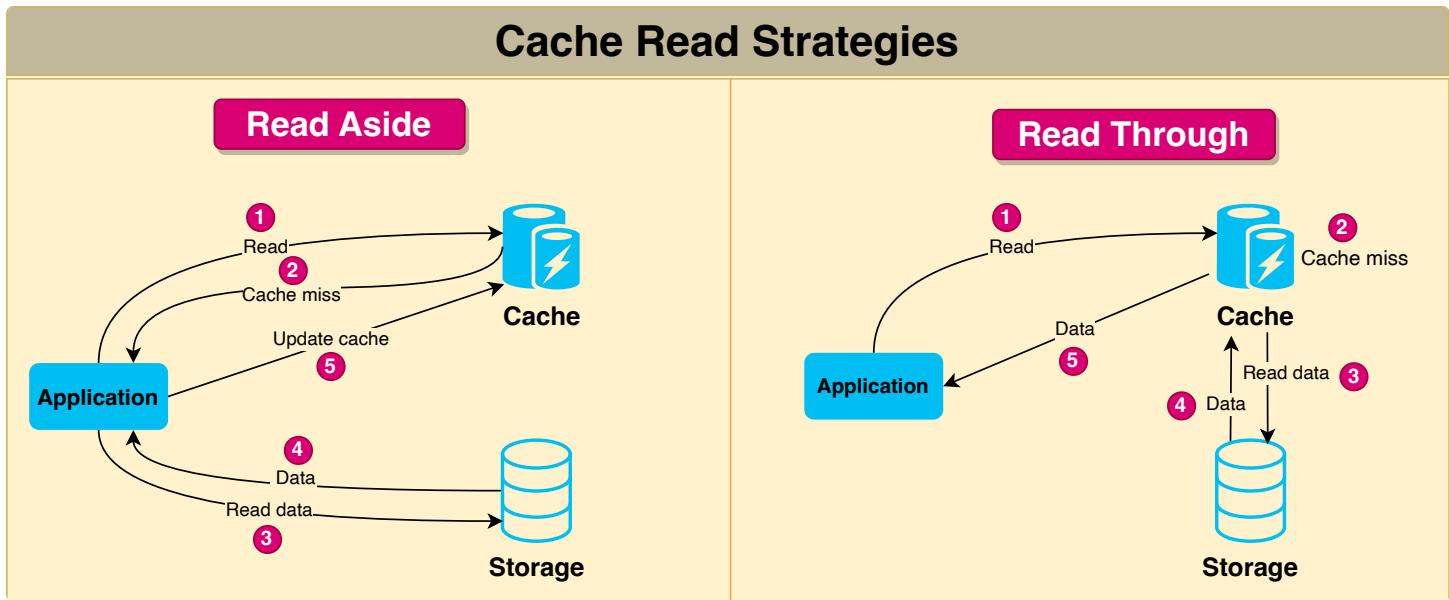
This approach helps to maintain consistency between the cache and the data store, as the cache is always responsible for retrieving and updating the data. It also simplifies the application code since the application doesn't need to handle cache misses and data retrieval logic. The read-through cache strategy can significantly improve performance in scenarios where data retrieval from the data store is expensive, and cache misses are relatively infrequent.

Read aside cache

A read-aside cache strategy, also known as **cache-aside** or **lazy-loading**, is a caching mechanism where the application is responsible for retrieving the data from the underlying data store when a cache miss occurs. In this strategy, the application first checks the cache for the requested data. If the data is found in the cache (cache hit), the application uses the cached data. However, if the data is not present in the cache (cache miss), the application retrieves the data from the data store, updates the cache with the retrieved data, and then uses the data.

The read-aside cache strategy provides better control over the caching process, as the application can decide when and how to update the cache. However, it also

adds complexity to the application code, as the application must handle cache misses and data retrieval logic. This approach can be beneficial in scenarios where cache misses are relatively infrequent, and the application wants to optimize cache usage based on specific data access patterns.



Cache eviction policies

Following are some of the most common cache eviction policies:

1. First In First Out (FIFO): The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
2. Last In First Out (LIFO): The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
3. Least Recently Used (LRU): Discards the least recently used items first.
4. Most Recently Used (MRU): Discards, in contrast to LRU, the most recently used items first.

5. Least Frequently Used (LFU): Counts how often an item is needed. Those that are used least often are discarded first.
6. Random Replacement (RR): Randomly selects a candidate item and discards it to make space when necessary.

Following links have some good discussion about caching:

- [1] [Cache](#)
- [2] [Introduction to architecting systems](#)

Data Partitioning

Data partitioning is process of dividing a large database (DB) into smaller, more manageable parts called partitions or shards. Each partition is independent and contains a subset of the overall data.

In data partitioning, the dataset is typically partitioned based on a certain criterion, such as data range, data size, or data type. Each partition is then assigned to a separate processing node, which can perform operations on its assigned data subset independently of the others.

Data partitioning can help improve performance and scalability of large-scale data processing applications, as it allows processing to be distributed across multiple nodes, minimizing data transfer and reducing processing time. Secondly, by distributing the data across multiple nodes or servers, the workload can be balanced, and the system can handle more requests and process data more efficiently.

Data partitioning can be done in several ways, including horizontal partitioning, vertical partitioning, and hybrid partitioning.

1. Partitioning Methods

Designing an effective partitioning scheme can be challenging and requires careful consideration of the application requirements and the characteristics of the data being processed. Below are three of the most popular schemes used by various large-scale applications.

a. Horizontal Partitioning: Also known as sharding, horizontal data partitioning involves dividing a database table into multiple partitions or shards, with each partition containing a subset of rows. Each shard is typically assigned to a

different database server, which allows for parallel processing and faster query execution times.

For example, consider a social media platform that stores user data in a database table. The platform might partition the user table horizontally based on the geographic location of the users, so that users in the United States are stored in one shard, users in Europe are stored in another shard, and so on. This way, when a user logs in and their data needs to be accessed, the query can be directed to the appropriate shard, minimizing the amount of data that needs to be scanned.

The key problem with this approach is that if the value whose range is used for partitioning isn't chosen carefully, then the partitioning scheme will lead to unbalanced servers. For instance, partitioning users based on their geographic location assumes an even distribution of users across different regions, which may not be valid due to the presence of densely or sparsely populated areas.

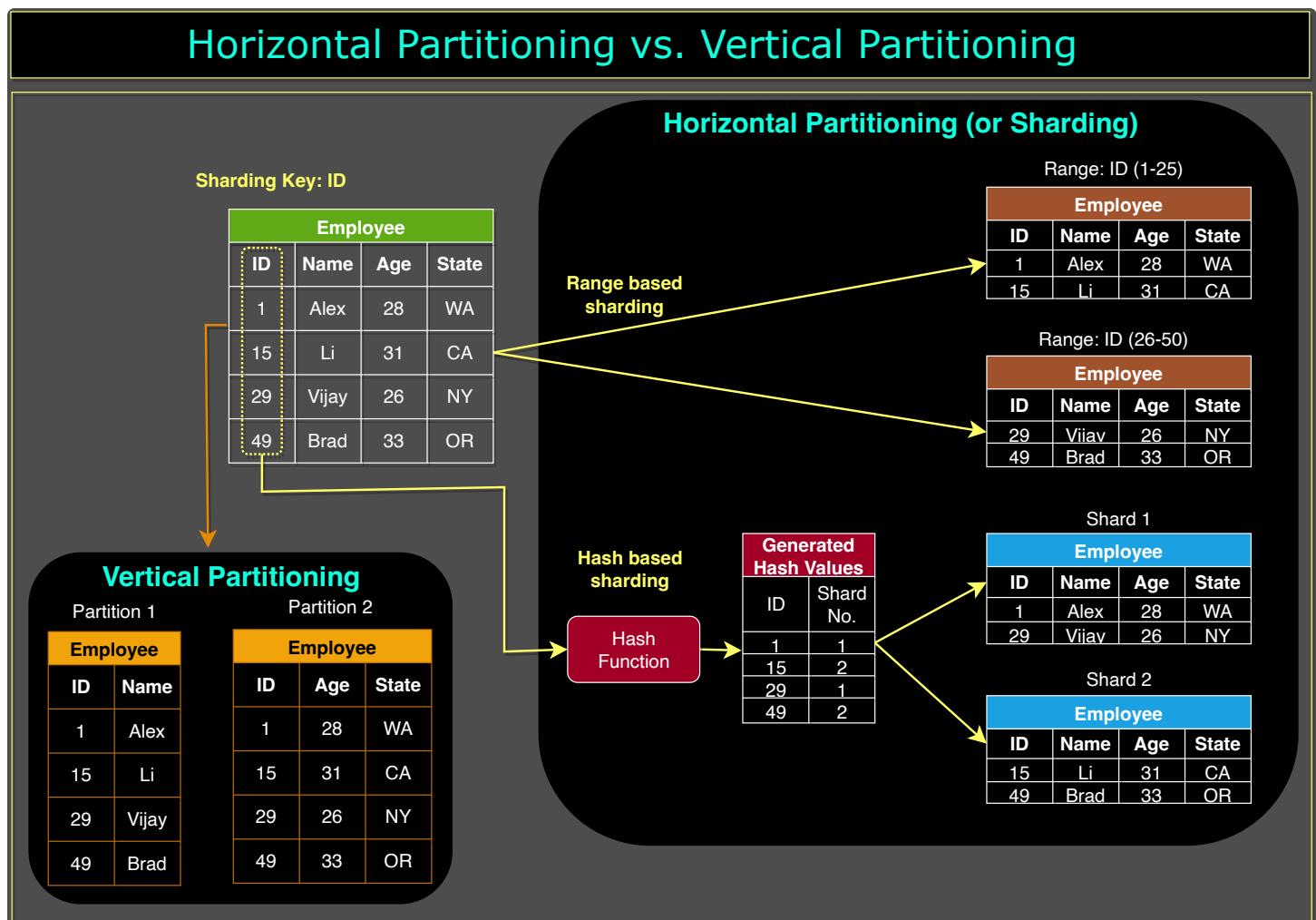
b. Vertical Partitioning: Vertical data partitioning involves splitting a database table into multiple partitions or shards, with each partition containing a subset of columns. This technique can help optimize performance by reducing the amount of data that needs to be scanned, especially when certain columns are accessed more frequently than others.

For example, consider an e-commerce website that stores customer data in a database table. The website might partition the customer table vertically based on the type of data, so that personal information such as name and address are stored in one shard, while order history and payment information are stored in another shard. This way, when a customer logs in and their order history needs to be accessed, the query can be directed to the appropriate shard, minimizing the amount of data that needs to be scanned.

c. Hybrid Partitioning: Hybrid data partitioning combines both horizontal and vertical partitioning techniques to partition data into multiple shards. This

technique can help optimize performance by distributing the data evenly across multiple servers, while also minimizing the amount of data that needs to be scanned.

For example, consider a large e-commerce website that stores customer data in a database table. The website might partition the customer table horizontally based on the geographic location of the customers, and then partition each shard vertically based on the type of data. This way, when a customer logs in and their data needs to be accessed, the query can be directed to the appropriate shard, minimizing the amount of data that needs to be scanned. Additionally, each shard can be stored on a different database server, allowing for parallel processing and faster query execution times.



Horizontal Partitioning vs. Vertical Partitioning

2. Partitioning Criteria

Data partitioning criteria are the factors or characteristics of data that can be used to divide a large dataset into smaller parts or partitions. Here are some of the most common criteria used for data partitioning:

- a. Key or Hash-based Partitioning:** Under this scheme, we apply a hash function to some key attributes of the entity we are storing; that yields the partition number. For example, if we have 100 DB servers and our ID is a numeric value that gets incremented by one each time a new record is inserted. In this example, the hash function could be 'ID % 100', which will give us the server number where we can store/read that record. This approach should ensure a uniform allocation of data among servers. The fundamental problem with this approach is that it effectively fixes the total number of DB servers, since adding new servers means changing the hash function which would require redistribution of data and downtime for the service. A workaround for this problem is to use 'Consistent Hashing'.
- b. List partitioning:** In this scheme, each partition is assigned a list of values, so whenever we want to insert a new record, we will see which partition contains our key and then store it there. For example, we can decide all users living in Iceland, Norway, Sweden, Finland, or Denmark will be stored in a partition for the Nordic countries.
- c. Round-robin partitioning:** This is a very simple strategy that ensures uniform data distribution. With 'n' partitions, the 'i' tuple is assigned to partition $(i \bmod n)$.
- d. Composite Partitioning:** Under this scheme, we combine any of the above partitioning schemes to devise a new scheme. For example, first applying a list partitioning scheme and then a hash-based partitioning. Consistent hashing could

be considered a composite of hash and list partitioning where the hash reduces the key-space to a size that can be listed.

3. Common Problems of Data Partitioning

On a partitioned database, there are certain extra constraints on the different operations that can be performed. Most of these constraints are due to the fact that operations across multiple tables or multiple rows in the same table will no longer run on the same server. Below are some of the constraints and additional complexities introduced by Partitioning:

a. Joins and Denormalization: Performing joins on a database that is running on one server is straightforward, but once a database is partitioned and spread across multiple machines it is often not feasible to perform joins that span database partitions. Such joins will not be performance efficient since data has to be compiled from multiple servers. A common workaround for this problem is to denormalize the database so that queries that previously required joins can be performed from a single table. Of course, the service now has to deal with denormalization's perils, such as data inconsistency.

b. Referential integrity: As we saw that performing a cross-partition query on a partitioned database is not feasible; similarly, trying to enforce data integrity constraints such as foreign keys in a partitioned database can be extremely difficult.

Most RDBMS do not support foreign keys constraints across databases on different database servers. This means, applications that require referential integrity on partitioned databases often have to enforce it in application code. Often in such cases, applications have to run regular SQL jobs to clean up dangling references.

c. Rebalancing: There could be many reasons we have to change our partitioning scheme:

1. The data distribution is not uniform, e.g., there are a lot of places for a particular ZIP code that cannot fit into one database partition.
2. There is a lot of load on a partition, e.g., there are too many requests being handled by the DB partition dedicated to user photos.

In such cases, either we have to create more DB partitions or have to rebalance existing partitions, which means the partitioning scheme changed and all existing data moved to new locations. Doing this without incurring downtime is extremely difficult. Using a scheme like directory-based Partitioning does make rebalancing a more palatable experience at the cost of increasing the complexity of the system and creating a new single point of failure (i.e. the lookup service/database).

Indexes

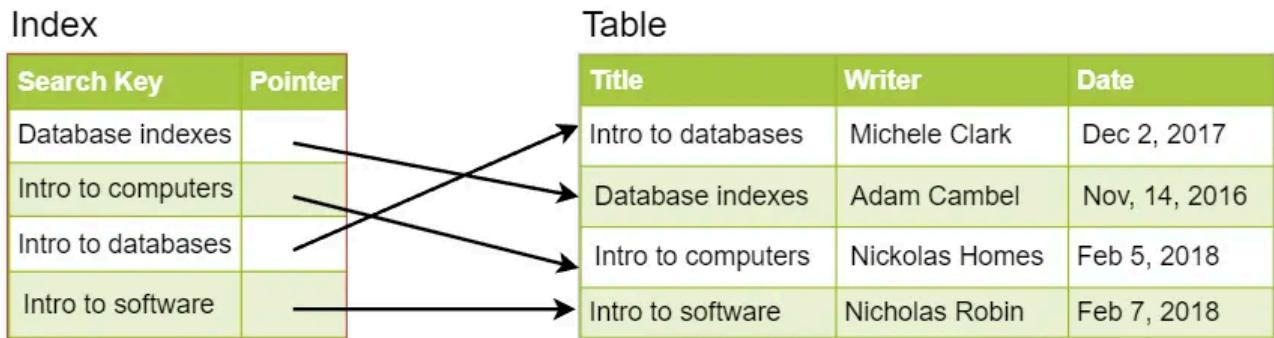
Indexes are well known when it comes to databases. Sooner or later there comes a time when database performance is no longer satisfactory. One of the very first things you should turn to when that happens is database indexing.

The goal of creating an index on a particular table in a database is to make it faster to search through the table and find the row or rows that we want. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

Example: A library catalog

A library catalog is a register that contains the list of books found in a library. The catalog is organized like a database table generally with four columns: book title, writer, subject, and date of publication. There are usually two such catalogs: one sorted by the book title and one sorted by the writer name. That way, you can either think of a writer you want to read and then look through their books or look up a specific book title you know you want to read in case you don't know the writer's name. These catalogs are like indexes for the database of books. They provide a sorted list of data that is easily searchable by relevant information.

Simply saying, an index is a data structure that can be perceived as a table of contents that points us to the location where actual data lives. So when we create an index on a column of a table, we store that column and a pointer to the whole row in the index. Let's assume a table containing a list of books, the following diagram shows how an index on the 'Title' column looks like:



Just like a traditional relational data store, we can also apply this concept to larger datasets. The trick with indexes is that we must carefully consider how users will access the data. In the case of data sets that are many terabytes in size, but have very small payloads (e.g., 1 KB), indexes are a necessity for optimizing data access. Finding a small payload in such a large dataset can be a real challenge, since we can't possibly iterate over that much data in any reasonable time. Furthermore, it is very likely that such a large data set is spread over several physical devices—this means we need some way to find the correct physical location of the desired data. Indexes are the best way to do this.

How do Indexes decrease write performance?

An index can dramatically speed up data retrieval but may itself be large due to the additional keys, which slow down data insertion & update.

When adding rows or making updates to existing rows for a table with an active index, we not only have to write the data but also have to update the index. This will decrease the write performance. This performance degradation applies to all insert, update, and delete operations for the table. For this reason, adding unnecessary indexes on tables should be avoided and indexes that are no longer used should be removed. To reiterate, adding indexes is about improving the performance of search queries. If the goal of the database is to provide a data store

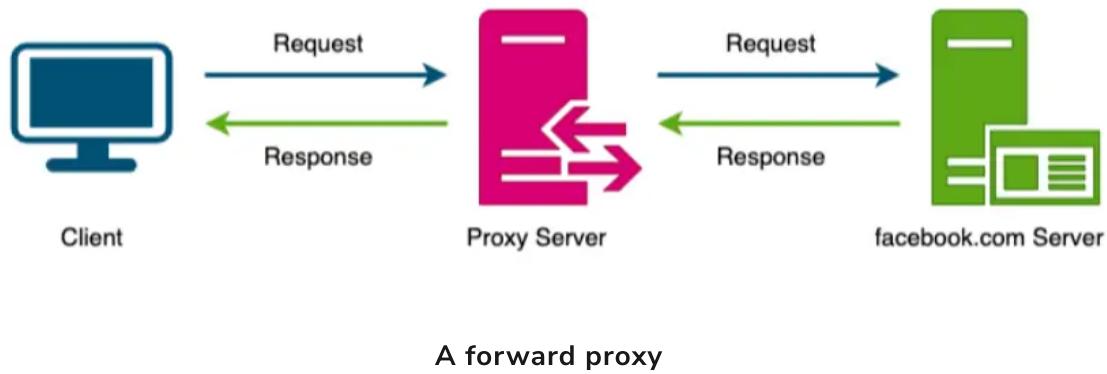
that is often written to and rarely read from, in that case, decreasing the performance of the more common operation, which is writing, is probably not worth the increase in performance we get from reading.

For more details, see [Database Indexes](#).

Proxies

What is a proxy server?

A proxy server is an intermediate piece of software or hardware that sits between the client and the server. Clients connect to a proxy to make a request for a service like a web page, file, or connection from the server. Essentially, a proxy server (aka the forward proxy) is a piece of software or hardware that facilitates the request for resources from other servers on behalf of clients, thus anonymizing the client from the server.



Typically, forward proxies are used to cache data, filter requests, log requests, or transform requests (by adding/removing headers, encrypting/decrypting, or compressing a resource).

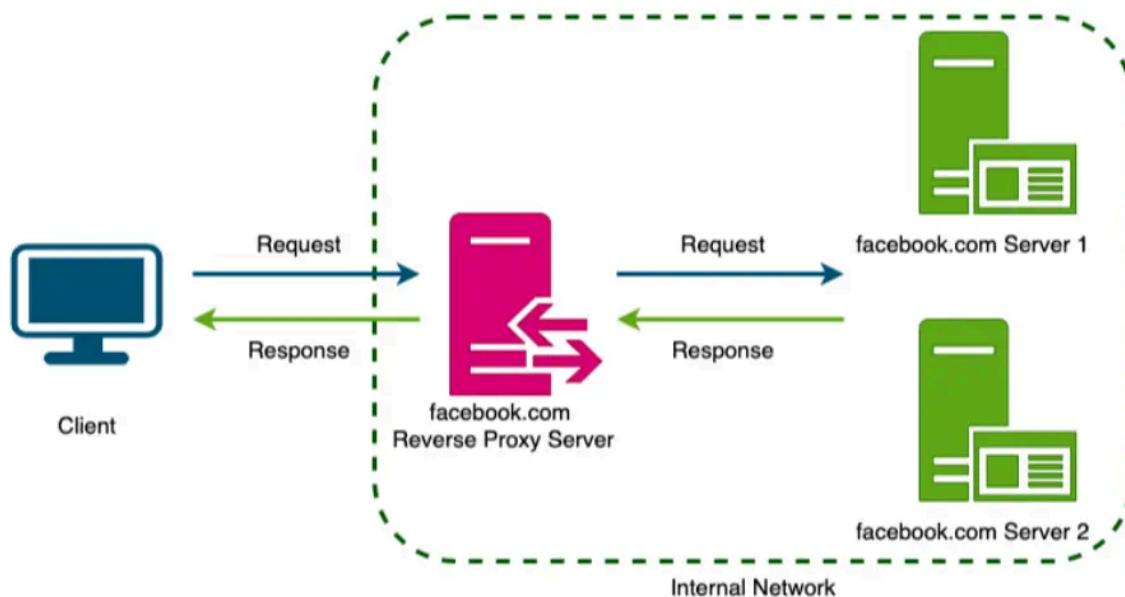
A forward proxy can hide the identity of the client from the server by sending requests on behalf of the client.

In addition to coordinating requests from multiple servers, proxies can also optimize request traffic from a system-wide perspective. Proxies can combine the

same data access requests into one request and then return the result to the user; this technique is called **collapsed forwarding**. Consider a request for the same data across several nodes, but the data is not in cache. By routing these requests through the proxy, they can be consolidated into one so that we will only read data from the disk once.

Reverse Proxy

A reverse proxy retrieves resources from one or more servers on behalf of a client. These resources are then returned to the client, appearing as if they originated from the proxy server itself, thus anonymizing the server. Contrary to the forward proxy, which hides the client's identity, a reverse proxy hides the server's identity.



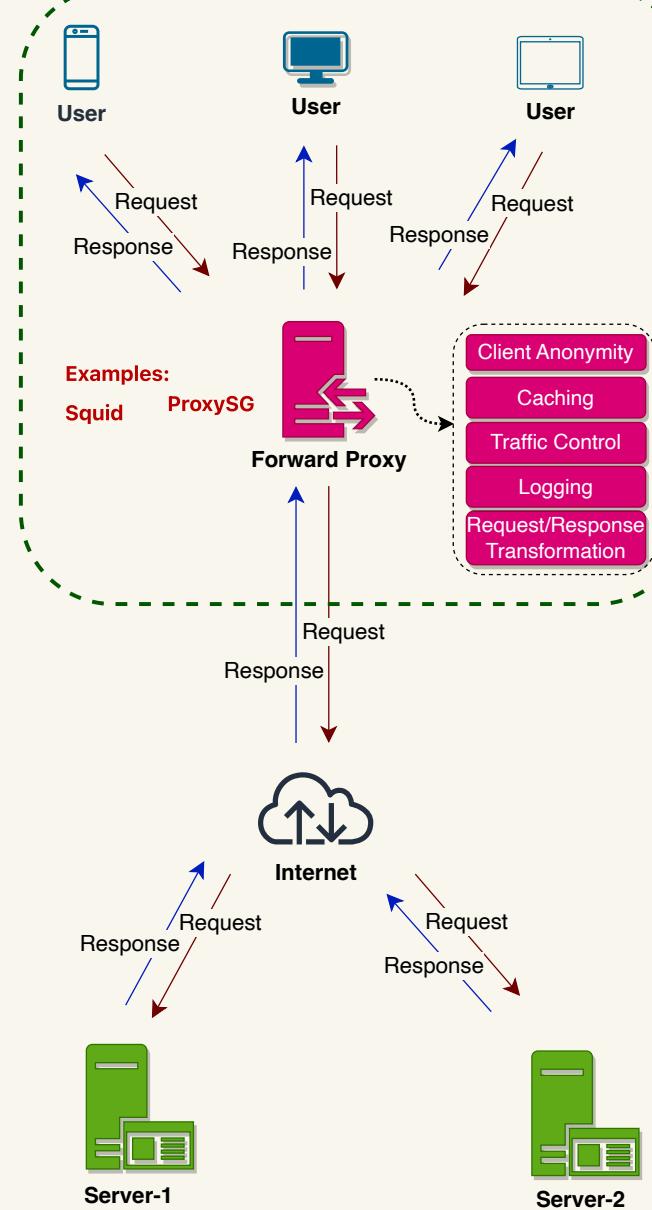
A reverse proxy

In the above diagram, the reverse proxy hides the final server that served the request from the client. The client makes a request for some content from facebook.com; this request is served by facebook's reverse proxy server, which gets the response from one of the backend servers and returns it to the client.

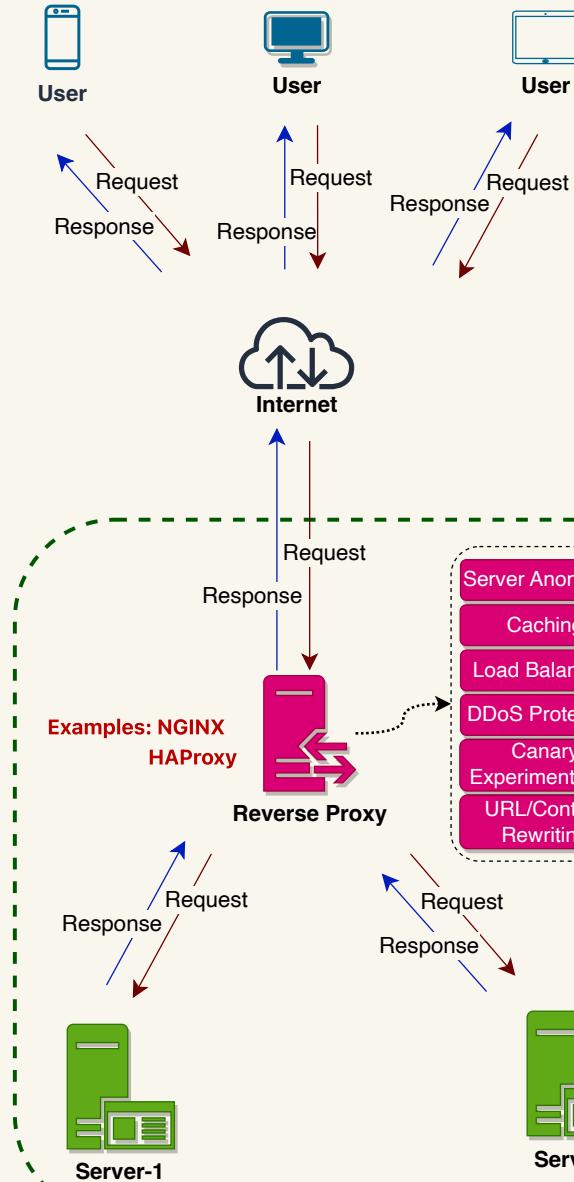
A reverse proxy, just like a forward proxy, can be used for caching, load balancing, or routing requests to the appropriate servers.

Forward Proxy vs. Reverse Proxy

Forward Proxy



Reverse Proxy



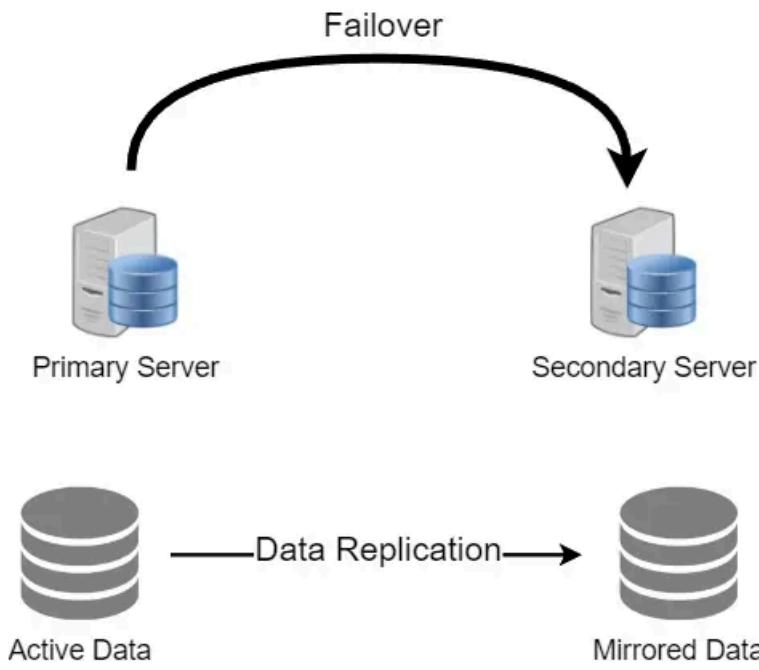
Summary

A proxy is a piece of software or hardware that sits between a client and a server to facilitate traffic. A forward proxy hides the identity of the client, whereas a reverse proxy conceals the identity of the server. So, when you want to protect your clients on your internal network, you should put them behind a forward proxy; on the other hand, when you want to protect your servers, you should put them behind a reverse proxy.

Redundancy and Replication

Redundancy is the duplication of critical components or functions of a system with the intention of increasing the reliability of the system, usually in the form of a backup or fail-safe, or to improve actual system performance. For example, if there is only one copy of a file stored on a single server, then losing that server means losing the file. Since losing data is seldom a good thing, we can create duplicate or redundant copies of the file to solve this problem.

Redundancy plays a key role in removing the single points of failure in the system and provides backups if needed in a crisis. For example, if we have two instances of a service running in production and one fails, the system can failover to the other one.



Database **replication** is the process of copying and synchronizing data from one database to one or more additional databases. This is commonly used in distributed systems where multiple copies of the same data are required to ensure data availability, fault tolerance, and scalability.

Replication is widely used in many database management systems (DBMS), usually with a primary-replica relationship between the original and the copies. The primary server gets all the updates, which then ripple through to the replica servers. Each replica outputs a message stating that it has received the update successfully, thus allowing the sending of subsequent updates.

Here are the top three typical database replication strategies:

Synchronous replication is a type of database replication where changes made to the primary database are immediately replicated to the replica databases before the write operation is considered complete. In other words, the primary database waits for the replica databases to confirm that they have received and processed the changes before the write operation is acknowledged.

In synchronous replication, there is a strong consistency between the primary and replica databases, as all changes made to the primary database are immediately reflected in the replica databases. This ensures that the data is consistent across all databases and reduces the risk of data loss or inconsistency.

Asynchronous replication is a type of database replication where changes made to the primary database are not immediately replicated to the replica databases. Instead, the changes are queued and replicated to the replicas at a later time.

In asynchronous replication, there is a delay between the write operation on the primary database and the update on the replica databases. This delay can result in temporary inconsistencies between the primary and replica databases, as the data on the replica databases may not immediately reflect the changes made to the primary database.

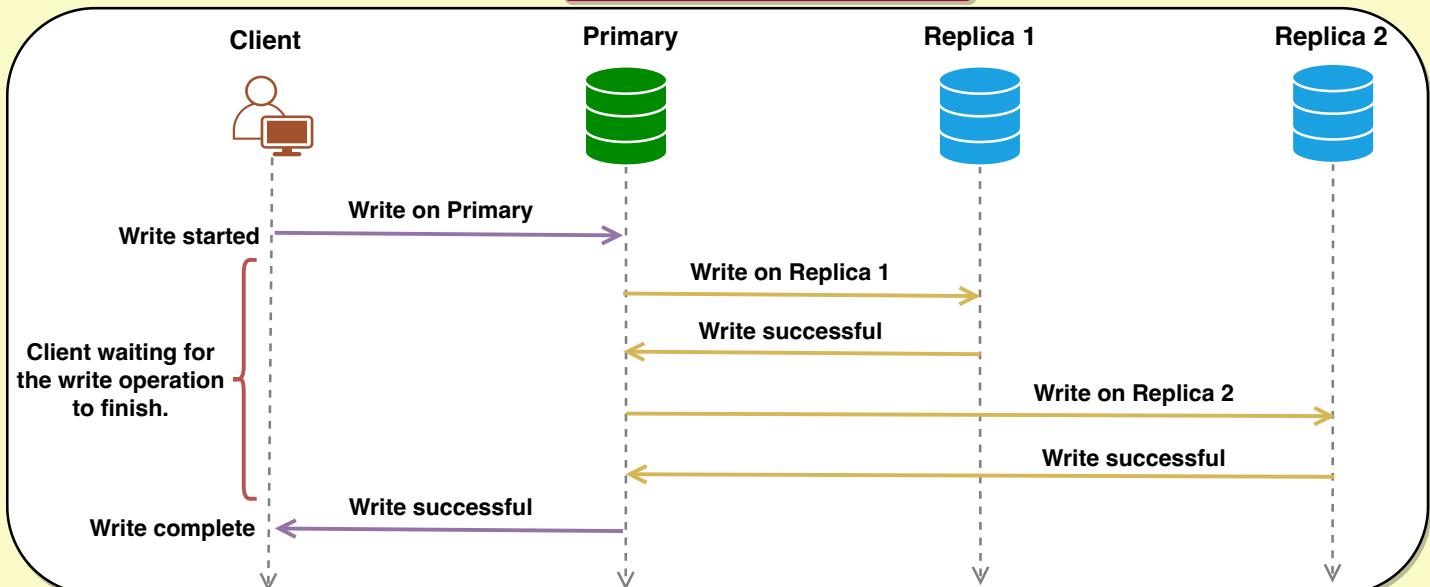
However, asynchronous replication can also have performance benefits, as write operations can be completed quickly without waiting for confirmation from the replica databases. In addition, if one or more replica databases are unavailable, the

write operation can still be completed on the primary database, ensuring that the system remains available.

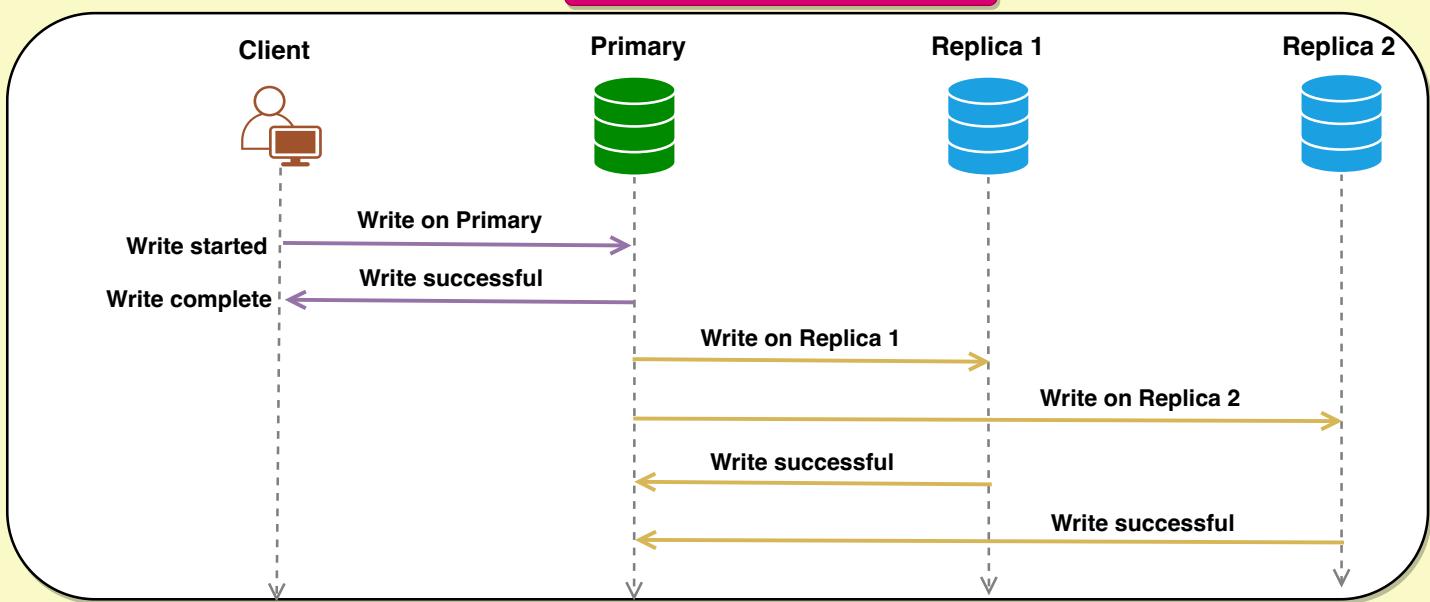
Semi-synchronous replication is a type of database replication that combines elements of both synchronous and asynchronous replication. In semi-synchronous replication, changes made to the primary database are immediately replicated to at least one replica database, while other replicas may be updated asynchronously.

In semi-synchronous replication, the write operation on the primary is not considered complete until at least one replica database has confirmed that it has received and processed the changes. This ensures that there is some level of strong consistency between the primary and replica databases, while also providing improved performance compared to fully synchronous replication.

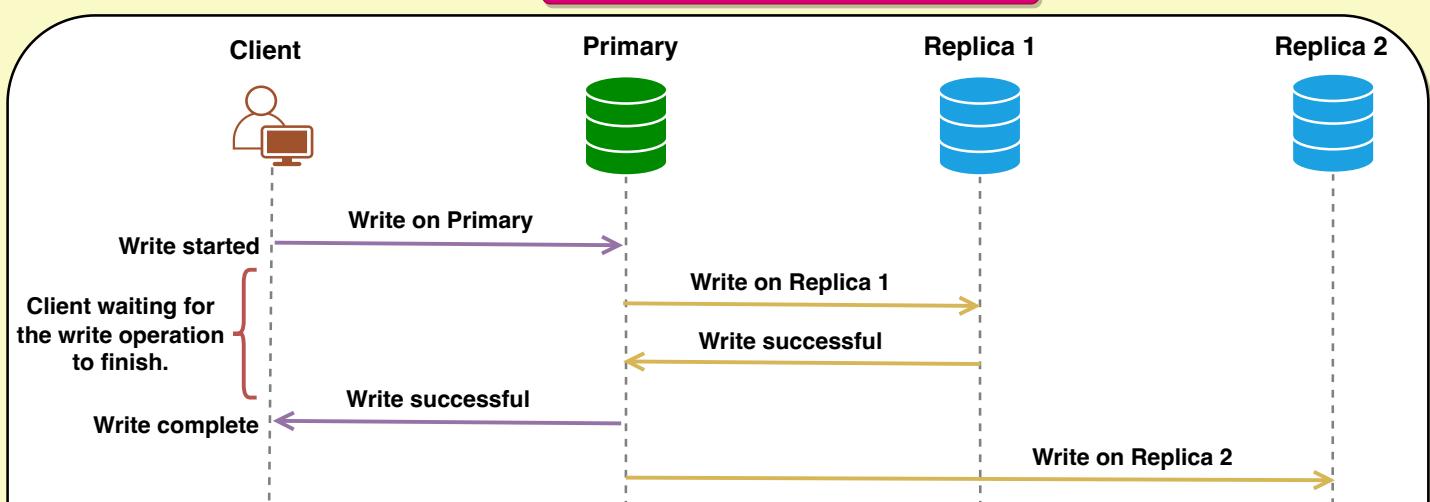
Synchronous Replication



Asynchronous Replication



Semi-synchronous Replication





Write successful

Replication Strategies

SQL vs. NoSQL

In the world of databases, there are two main types of solutions: SQL and NoSQL (or relational databases and non-relational databases). Both of them differ in the way they were built, the kind of information they store, and the storage method they use.

Relational databases are structured and have predefined schemas like phone books that store phone numbers and addresses. Non-relational databases are unstructured, distributed, and have a dynamic schema like file folders that hold everything from a person's address and phone number to their Facebook 'likes' and online shopping preferences.

SQL

Relational databases store data in rows and columns. Each row contains all the information about one entity and each column contains all the separate data points. Some of the most popular relational databases are MySQL, Oracle, MS SQL Server, SQLite, Postgres, and MariaDB.

NoSQL

Following are the most common types of NoSQL:

Key-Value Stores: Data is stored in an array of key-value pairs. The 'key' is an attribute name which is linked to a 'value'. Well-known key-value stores include Redis, Voldemort, and Dynamo.

Document Databases: In these databases, data is stored in documents (instead of rows and columns in a table) and these documents are grouped together in

collections. Each document can have an entirely different structure. Document databases include the CouchDB and MongoDB.

Wide-Column Databases: Instead of 'tables,' in columnar databases we have column families, which are containers for rows. Unlike relational databases, we don't need to know all the columns up front and each row doesn't have to have the same number of columns. Columnar databases are best suited for analyzing large datasets - big names include Cassandra and HBase.

Graph Databases: These databases are used to store data whose relations are best represented in a graph. Data is saved in graph structures with nodes (entities), properties (information about the entities), and lines (connections between the entities). Examples of graph database include Neo4J and InfiniteGraph.

High level differences between SQL and NoSQL

Storage: SQL stores data in tables where each row represents an entity and each column represents a data point about that entity; for example, if we are storing a car entity in a table, different columns could be 'Color', 'Make', 'Model', and so on.

NoSQL databases have different data storage models. The main ones are key-value, document, graph, and columnar. We will discuss differences between these databases below.

Schema: In SQL, each record conforms to a fixed schema, meaning the columns must be decided and chosen before data entry and each row must have data for each column. The schema can be altered later, but it involves modifying the whole database and going offline.

In NoSQL, schemas are dynamic. Columns can be added on the fly and each 'row' (or equivalent) doesn't have to contain data for each 'column.'

Querying: SQL databases use SQL (structured query language) for defining and manipulating the data, which is very powerful. In a NoSQL database, queries are focused on a collection of documents. Sometimes it is also called UnQL (Unstructured Query Language). Different databases have different syntax for using UnQL.

Scalability: In most common situations, SQL databases are vertically scalable, i.e., by increasing the horsepower (higher Memory, CPU, etc.) of the hardware, which can get very expensive. It is possible to scale a relational database across multiple servers, but this is a challenging and time-consuming process.

On the other hand, NoSQL databases are horizontally scalable, meaning we can add more servers easily in our NoSQL database infrastructure to handle a lot of traffic. Any cheap commodity hardware or cloud instances can host NoSQL databases, thus making it a lot more cost-effective than vertical scaling. A lot of NoSQL technologies also distribute data across servers automatically.

Reliability or ACID Compliancy (Atomicity, Consistency, Isolation, Durability): The vast majority of relational databases are ACID compliant. So, when it comes to data reliability and safe guarantee of performing transactions, SQL databases are still the better bet.

Most of the NoSQL solutions sacrifice ACID compliance for performance and scalability.

SQL VS. NoSQL - Which one to use?

When it comes to database technology, there's no one-size-fits-all solution. That's why many businesses rely on both relational and non-relational databases for different needs. Even as NoSQL databases are gaining popularity for their speed

and scalability, there are still situations where a highly structured SQL database may perform better; choosing the right technology hinges on the use case.

Reasons to use SQL database

Here are a few reasons to choose a SQL database:

1. We need to ensure ACID compliance. ACID compliance reduces anomalies and protects the integrity of your database by prescribing exactly how transactions interact with the database. Generally, NoSQL databases sacrifice ACID compliance for scalability and processing speed, but for many e-commerce and financial applications, an ACID-compliant database remains the preferred option.
2. Your data is structured and unchanging. If your business is not experiencing massive growth that would require more servers and if you're only working with data that is consistent, then there may be no reason to use a system designed to support a variety of data types and high traffic volume.

Reasons to use NoSQL database

When all the other components of our application are fast and seamless, NoSQL databases prevent data from being the bottleneck. Big data is contributing to a large success for NoSQL databases, mainly because it handles data differently than the traditional relational databases. A few popular examples of NoSQL databases are MongoDB, CouchDB, Cassandra, and HBase.

1. Storing large volumes of data that often have little to no structure. A NoSQL database sets no limits on the types of data we can store together and allows us to add new types as the need changes. With document-based databases, you can

store data in one place without having to define what “types” of data those are in advance.

2. Making the most of cloud computing and storage. Cloud-based storage is an excellent cost-saving solution but requires data to be easily spread across multiple servers to scale up. Using commodity (affordable, smaller) hardware on-site or in the cloud saves you the hassle of additional software and NoSQL databases like Cassandra are designed to be scaled across multiple data centers out of the box, without a lot of headaches.
3. Rapid development. NoSQL is extremely useful for rapid development as it doesn't need to be prepped ahead of time. If you're working on quick iterations of your system which require making frequent updates to the data structure without a lot of downtime between versions, a relational database will slow you down.

CAP Theorem

Let's learn about the CAP theorem and its usage in distributed systems.

Background

In distributed systems, different types of failures can occur, e.g., servers can crash or fail permanently, disks can go bad resulting in data losses, or network connection can be lost, making a part of the system inaccessible. How can a distributed system model itself to get the maximum benefits out of different resources available?

Solution

CAP theorem states that it is **impossible** for a distributed system to simultaneously provide all three of the following desirable properties:

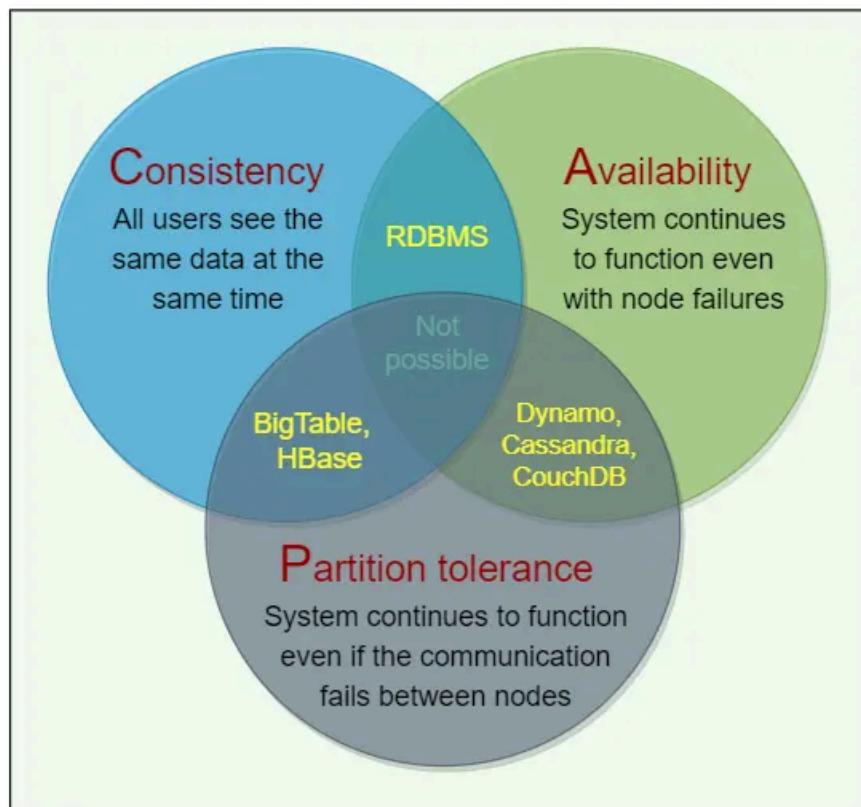
Consistency (C): All nodes see the same data at the same time. This means users can read or write from/to any node in the system and will receive the same data. It is equivalent to having a single up-to-date copy of the data.

Availability (A): Availability means every request received by a non-failing node in the system must result in a response. Even when severe network failures occur, every request must terminate. In simple terms, availability refers to a system's ability to remain accessible even if one or more nodes in the system go down.

Partition tolerance (P): A partition is a communication break (or a network failure) between any two nodes in the system, i.e., both nodes are up but cannot communicate with each other. A partition-tolerant system continues to operate

even if there are partitions in the system. Such a system can sustain any network failure that does not result in the failure of the entire network. Data is sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.

According to the CAP theorem, any distributed system needs to pick two out of the three properties. The three options are CA, CP, and AP. However, CA is not really a coherent option, as a system that is not partition-tolerant will be forced to give up either Consistency or Availability in the case of a network partition. Therefore, the theorem can really be stated as: **In the presence of a network partition, a distributed system must choose either Consistency or Availability.**



CAP theorem

We cannot build a general data store that is continually available, sequentially consistent, and tolerant to any partition failures. We can only build a system that has any two of these three properties. Because, to be consistent, all nodes should

see the same set of updates in the same order. But if the network loses a partition, updates in one partition might not make it to the other partitions before a client reads from the out-of-date partition after having read from the up-to-date one. The only thing that can be done to cope with this possibility is to stop serving requests from the out-of-date partition, but then the service is no longer 100% available.

PACELC Theorem (New)

Let's learn about the PACELC theorem and its usage.

Background

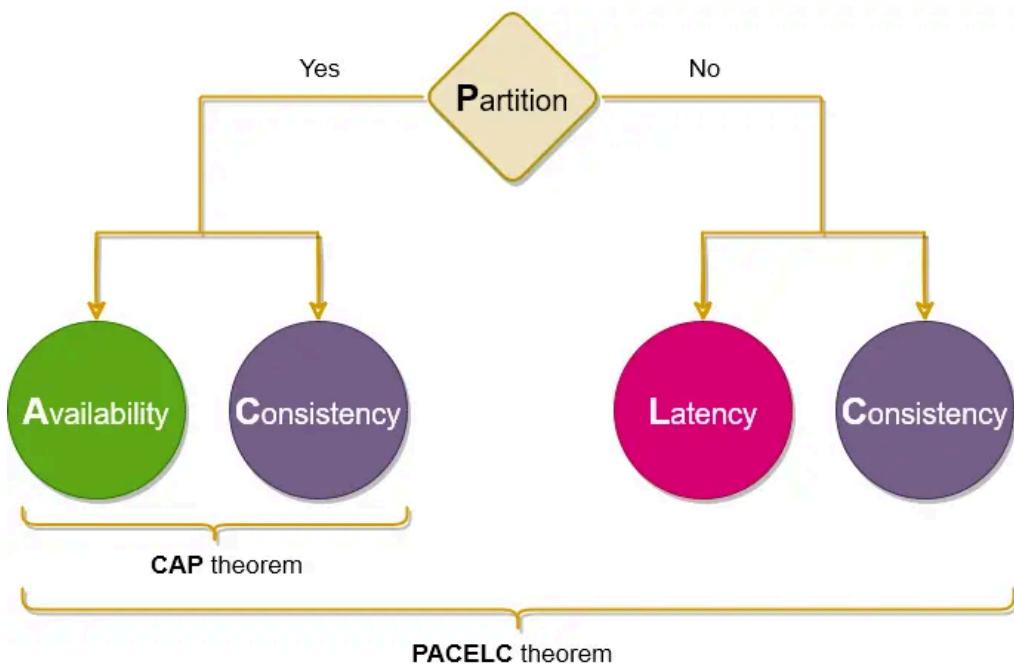
We cannot avoid partition in a distributed system, therefore, according to the CAP theorem, a distributed system should choose between consistency or availability. **ACID** (Atomicity, Consistency, Isolation, Durability) databases, such as RDBMSs like MySQL, Oracle, and Microsoft SQL Server, chose consistency (refuse response if it cannot check with peers), while **BASE** (Basically Available, Soft-state, Eventually consistent) databases, such as NoSQL databases like MongoDB, Cassandra, and Redis, chose availability (respond with local data without ensuring it is the latest with its peers).

One place where the CAP theorem is silent is what happens when there is no network partition? What choices does a distributed system have when there is no partition?

Solution

The PACELC theorem states that in a system that replicates data:

- if there is a partition ('P'), a distributed system can tradeoff between availability and consistency (i.e., 'A' and 'C');
- else ('E'), when the system is running normally in the absence of partitions, the system can tradeoff between latency ('L') and consistency ('C').



The first part of the theorem (**PAC**) is the same as the CAP theorem, and the **ELC** is the extension. The whole thesis is assuming we maintain high availability by replication. So, when there is a failure, CAP theorem prevails. But if not, we still have to consider the tradeoff between consistency and latency of a replicated system.

Examples

- **Dynamo and Cassandra** are **PA/EL** systems: They choose availability over consistency when a partition occurs; otherwise, they choose lower latency.
- **BigTable and HBase** are **PC/EC** systems: They will always choose consistency, giving up availability and lower latency.
- **MongoDB** can be considered **PA/EC** (default configuration): MongoDB works in a primary/secondaries configuration. In the default configuration, all writes and reads are performed on the primary. As all replication is done asynchronously (from primary to secondaries), when there is a network partition in which primary is lost or becomes isolated on the minority side, there is a chance of losing data that is unreplicated to secondaries, hence there

is a loss of consistency during partitions. Therefore it can be concluded that **in the case of a network partition, MongoDB chooses availability, but otherwise guarantees consistency.** Alternately, when MongoDB is configured to write on majority replicas and read from the primary, it could be categorized as PC/EC.

Consistent Hashing (New)

Let's learn about Consistent Hashing and its usage.

Background

While designing a scalable system, the most important aspect is defining how the data will be partitioned and replicated across servers. Let's first define these terms before moving on:

Data partitioning: It is the process of distributing data across a set of servers. It improves the scalability and performance of the system.

Data replication: It is the process of making multiple copies of data and storing them on different servers. It improves the availability and durability of the data across the system.

Data partition and replication strategies lie at the core of any distributed system. A carefully designed scheme for partitioning and replicating the data **enhances the performance, availability, and reliability of the system** and also defines how efficiently the system will be scaled and managed.

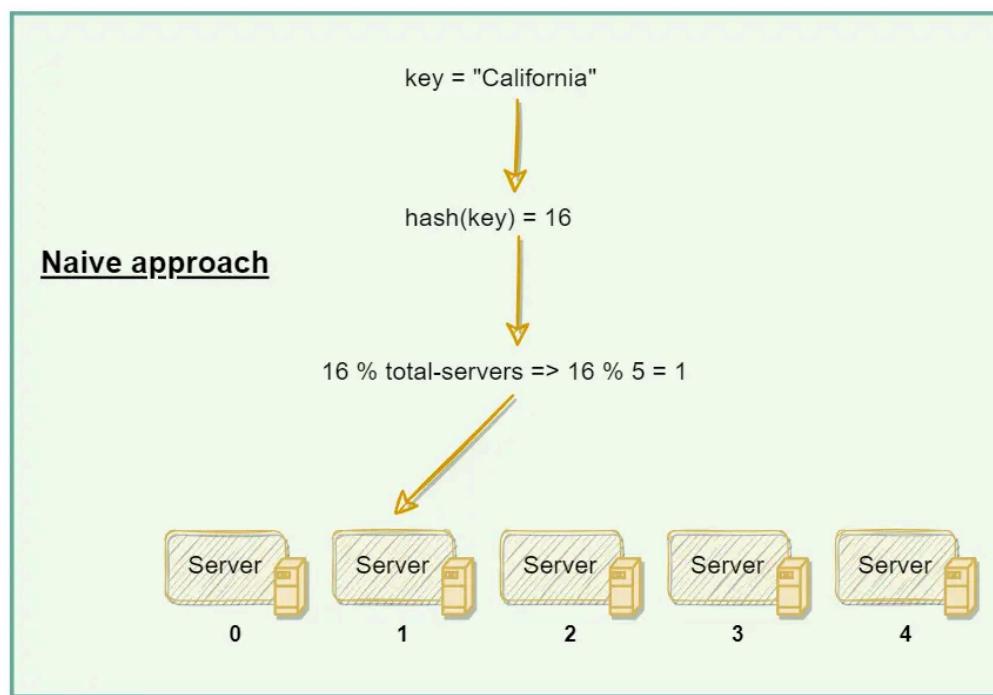
David Karger et al. first introduced Consistent Hashing in their [1997 paper](#) and suggested its use in distributed caching. Later, Consistent Hashing was adopted and enhanced to be used across many distributed systems. In this lesson we will see how Consistent Hashing efficiently solves the problem of data partitioning and replication.

What is data partitioning?

As stated above, the act of distributing data across a set of nodes is called data partitioning. There are two challenges when we try to distribute data:

1. How do we know on which node a particular piece of data will be stored?
2. When we add or remove nodes, how do we know what data will be moved from existing nodes to the new nodes? Additionally, how can we minimize data movement when nodes join or leave?

A naive approach will use a suitable hash function to map the data key to a number. Then, find the server by applying modulo on this number and the total number of servers. For example:



Data partitioning using simple hashing

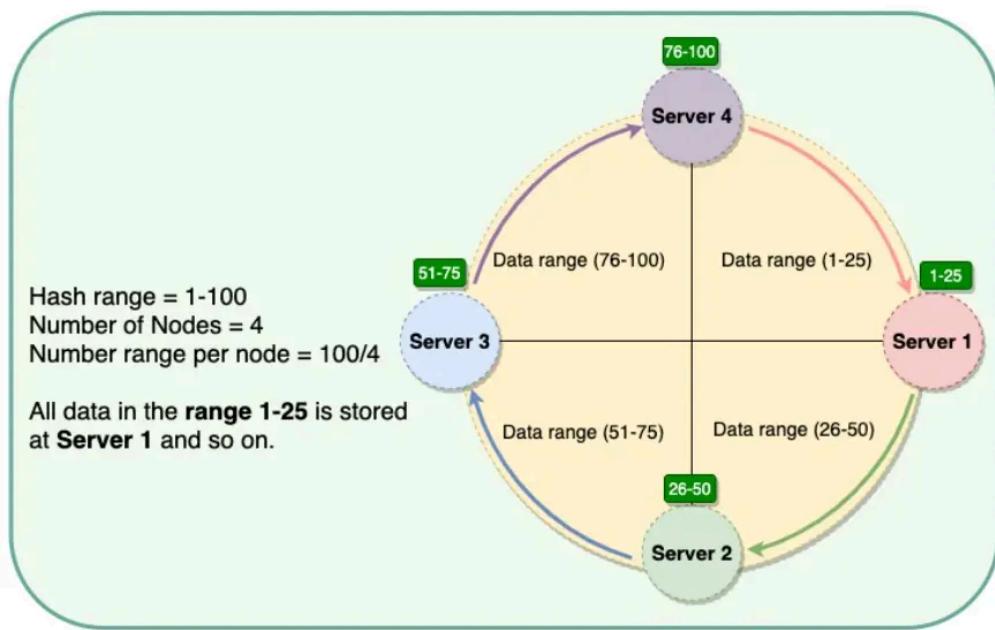
The scheme described in the above diagram solves the problem of finding a server for storing/retrieving the data. But when we add or remove a server, all our existing mappings will be broken. This is because the total number of servers will be changed, which was used to find the actual server storing the data. So to get

things working again, we have to **remap all the keys** and move our data based on the new server count, which will be a **complete mess!**

Consistent Hashing to the rescue

Distributed systems can use Consistent Hashing to distribute data across nodes. Consistent Hashing maps data to physical nodes and ensures that **only a small set of keys move when servers are added or removed.**

Consistent Hashing stores the data managed by a distributed system in a ring. Each node in the ring is assigned a range of data. Here is an example of the consistent hash ring:



Consistent Hashing ring

With consistent hashing, the ring is divided into smaller, predefined ranges. Each node is assigned one of these ranges. The start of the range is called a **token**. This means that each node will be assigned one token. The range assigned to each node is computed as follows:

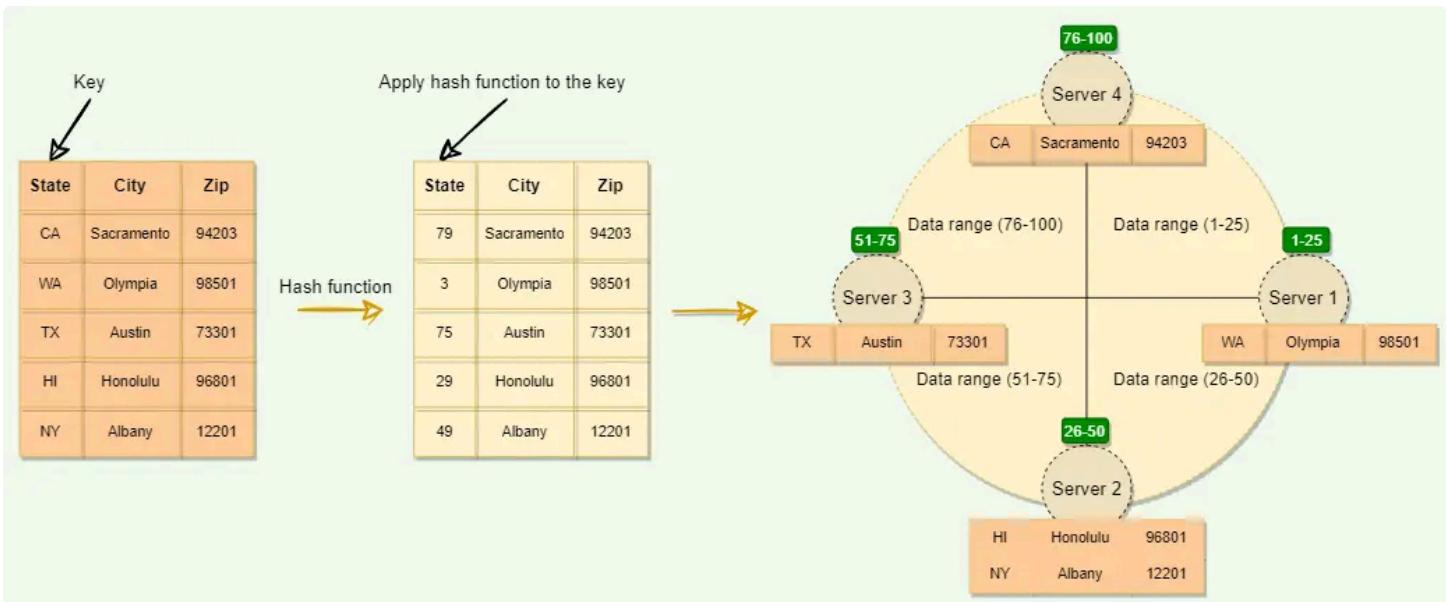
Range start: Token value

Range end: Next token value - 1

Here are the tokens and data ranges of the four nodes described in the above diagram:

Server	Token	Range Start	Range End
Server 1	1	1	25
Server 2	26	26	50
Server 3	51	51	75
Server 4	76	76	100

Whenever the system needs to read or write data, the first step it performs is to apply the [MD5 hashing algorithm](#) to the key. The output of this hashing algorithm determines within which range the data lies and hence, on which node the data will be stored. As we saw above, each node is supposed to store data for a fixed range. Thus, the hash generated from the key tells us the node where the data will be stored.



Distributing data on the Consistent Hashing ring

The Consistent Hashing scheme described above works great when a node is added or removed from the ring, as in these cases, since only the next node is affected. For example, when a node is removed, the next node becomes responsible for all of the keys stored on the outgoing node. However, this scheme can **result in non-uniform data and load distribution**. This problem can be solved with the help of Virtual nodes.

Virtual nodes

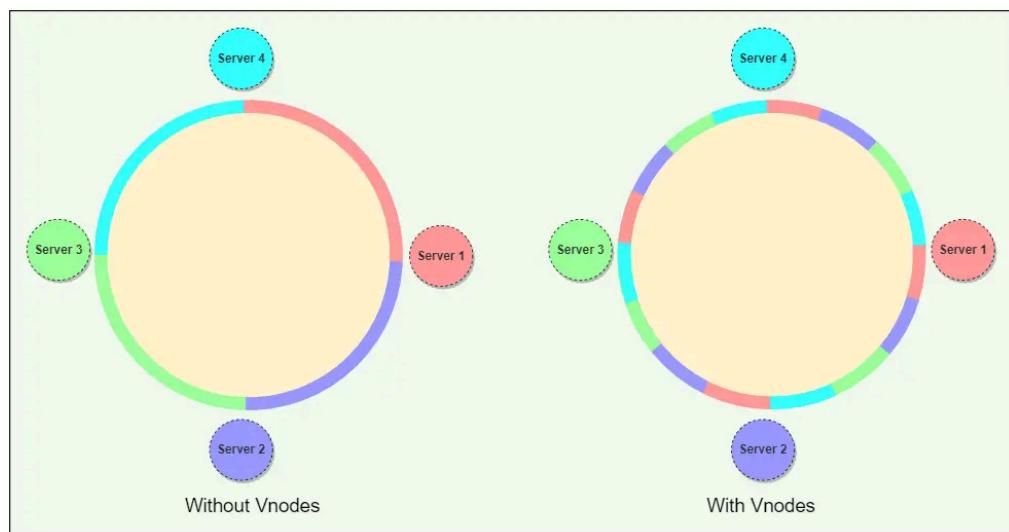
Adding and removing nodes in any distributed system is quite common. Existing nodes can die and may need to be decommissioned. Similarly, new nodes may be added to an existing cluster to meet growing demands. To efficiently handle these scenarios, Consistent Hashing makes use of virtual nodes (or Vnodes).

As we saw above, the basic Consistent Hashing algorithm assigns a single token (or a consecutive hash range) to each physical node. This was a static division of ranges that requires calculating tokens based on a given number of nodes. This

scheme made adding or replacing a node an expensive operation, as, in this case, we would like to rebalance and distribute the data to all other nodes, resulting in moving a lot of data. Here are a few potential issues associated with a manual and fixed division of the ranges:

- **Adding or removing nodes:** Adding or removing nodes will result in recomputing the tokens causing a significant administrative overhead for a large cluster.
- **Hotspots:** Since each node is assigned one large range, if the data is not evenly distributed, some nodes can become **hotspots**.
- **Node rebuilding:** Since each node's data might be replicated (for fault-tolerance) on a fixed number of other nodes, when we need to rebuild a node, only its replica nodes can provide the data. This puts a lot of pressure on the replica nodes and can lead to service degradation.

To handle these issues, Consistent Hashing introduces a new scheme of distributing the tokens to physical nodes. Instead of assigning a single token to a node, the hash range is divided into multiple smaller ranges, and each physical node is assigned several of these smaller ranges. Each of these subranges is considered a Vnode. With Vnodes, instead of a node being responsible for just one token, it is responsible for many tokens (or subranges).



Mapping Vnodes to physical nodes on a Consistent Hashing ring

Advantages of Vnodes

Vnones gives the following advantages:

1. As Vnones help spread the load more evenly across the physical nodes on the cluster by dividing the hash ranges into smaller subranges, this speeds up the rebalancing process after adding or removing nodes. When a new node is added, it receives many Vnones from the existing nodes to maintain a balanced cluster. Similarly, when a node needs to be rebuilt, instead of getting data from a fixed number of replicas, many nodes participate in the rebuild process.
2. Vnones make it easier to maintain a cluster containing heterogeneous machines. This means, with Vnones, we can assign a high number of subranges to a powerful server and a lower number of sub-ranges to a less powerful server.
3. In contrast to one big range, since Vnones help assign smaller ranges to each physical node, this decreases the probability of hotspots.

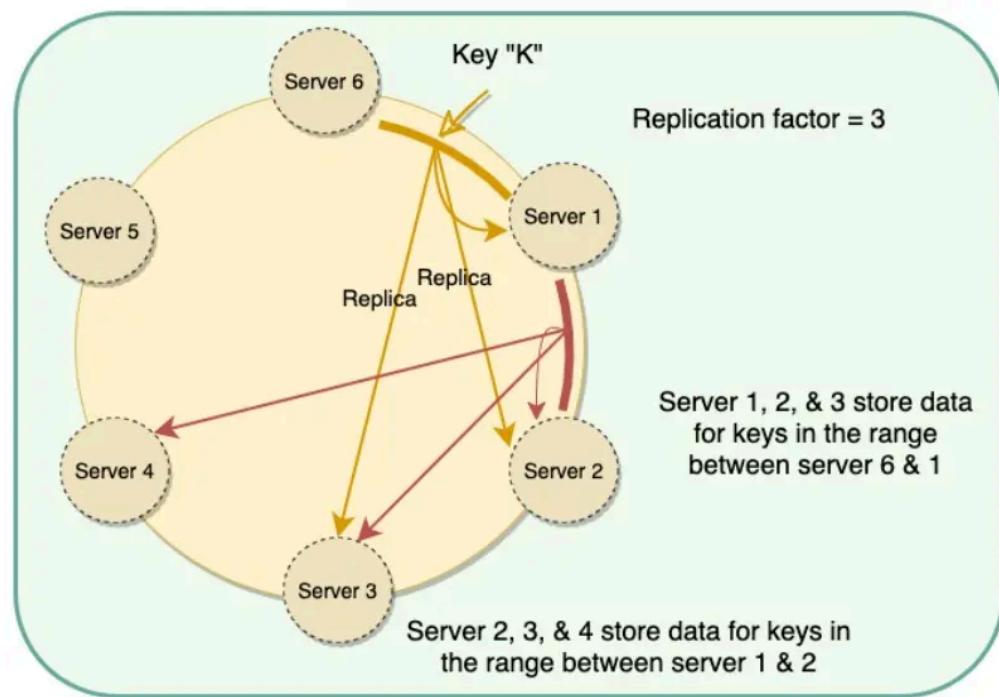
Data replication using Consistent Hashing

To ensure [highly availability](#) and [durability](#), Consistent Hashing replicates each data item on multiple N nodes in the system where the value N is equivalent to the replication factor.

The replication factor is the number of nodes that will receive the copy of the same data. For example, a replication factor of two means there are two copies of each data item, where each copy is stored on a different node.

Each key is assigned to a **coordinator node** (generally the first node that falls in the hash range), which first stores the data locally and then replicates it to $N - 1$ clockwise successor nodes on the ring. This results in each node owning the region on the ring between it and its N^{th} predecessor. In an **eventually consistent** system, this replication is done asynchronously (in the background).

In eventually consistent systems, copies of data don't always have to be identical as long as they are designed to eventually become consistent. In distributed systems, eventual consistency is used to achieve high availability.



Replication in Consistent Hashing

Consistent Hashing in System Design Interviews

As we saw above, Consistent Hashing helps with efficiently partitioning and replicating data; therefore, any distributed system that needs to scale up or down

or wants to achieve high availability through data replication can utilize Consistent Hashing. A few such examples could be:

- Any system working with a set of storage (or database) servers and needs to scale up or down based on the usage, e.g., the system could need more storage during Christmas because of high traffic.
- Any distributed system that needs dynamic adjustment of its cache usage by adding or removing cache servers based on the traffic load.
- Any system that wants to replicate its data shards to achieve high availability.

Consistent Hashing use cases

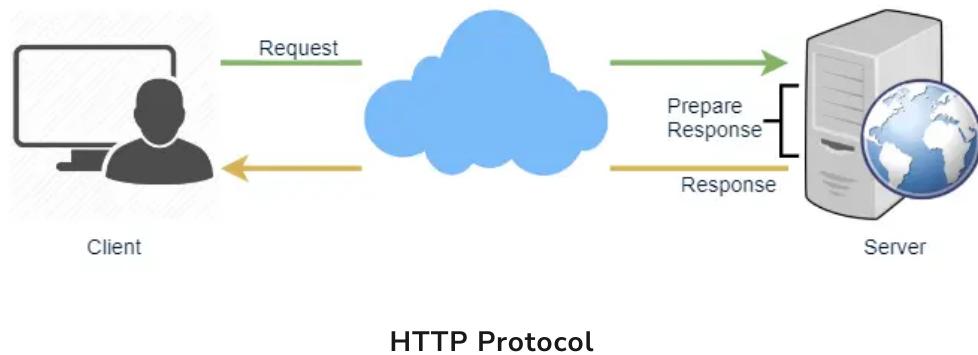
Amazon's [Dynamo](#) and Apache [Cassandra](#) use Consistent Hashing to distribute and replicate data across nodes.

Long-Polling vs WebSockets vs Server-Sent Events

What is the difference between Long-Polling, WebSockets, and Server-Sent Events?

Long-Polling, WebSockets, and Server-Sent Events are popular communication protocols between a client like a web browser and a web server. First, let's start with understanding what a standard HTTP web request looks like. Following are a sequence of events for regular HTTP request:

1. The client opens a connection and requests data from the server.
2. The server calculates the response.
3. The server sends the response back to the client on the opened request.

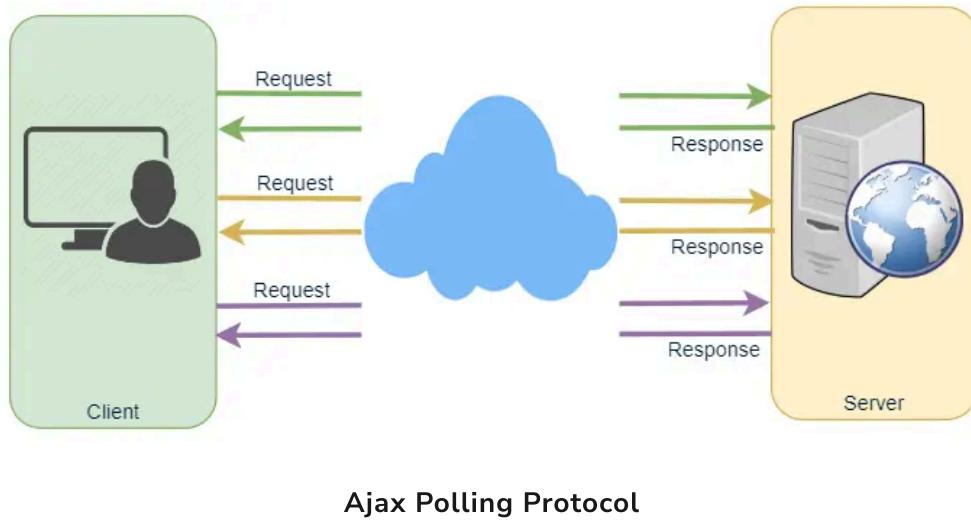


Ajax Polling

Polling is a standard technique used by the vast majority of AJAX applications. The basic idea is that the client repeatedly polls (or requests) a server for data. The client makes a request and waits for the server to respond with data. If no data is available, an empty response is returned.

1. The client opens a connection and requests data from the server using regular HTTP.
2. The requested webpage sends requests to the server at regular intervals (e.g., 0.5 seconds).
3. The server calculates the response and sends it back, just like regular HTTP traffic.
4. The client repeats the above three steps periodically to get updates from the server.

The problem with Polling is that the client has to keep asking the server for any new data. As a result, a lot of responses are empty, creating HTTP overhead.



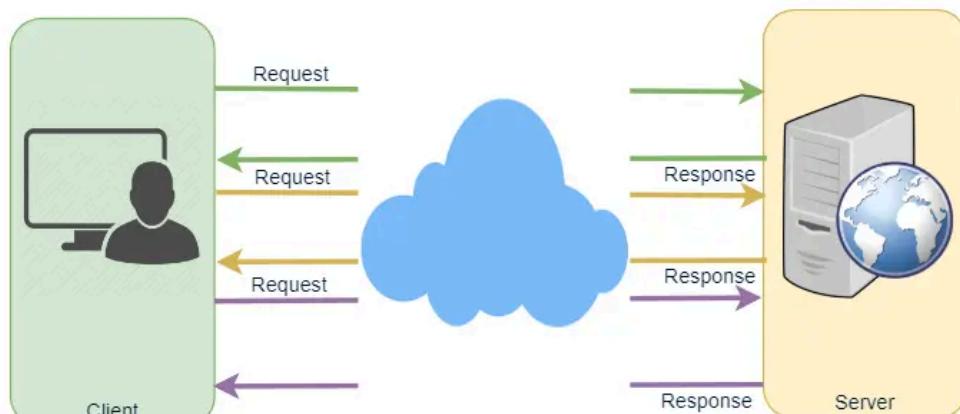
HTTP Long-Polling

This is a variation of the traditional polling technique that allows the server to push information to a client whenever the data is available. With Long-Polling, the client requests information from the server exactly as in normal polling, but with the expectation that the server may not respond immediately. That's why this technique is sometimes referred to as a "Hanging GET".

- If the server does not have any data available for the client, instead of sending an empty response, the server holds the request and waits until some data becomes available.
- Once the data becomes available, a full response is sent to the client. The client then immediately re-request information from the server so that the server will almost always have an available waiting request that it can use to deliver data in response to an event.

The basic life cycle of an application using HTTP Long-Polling is as follows:

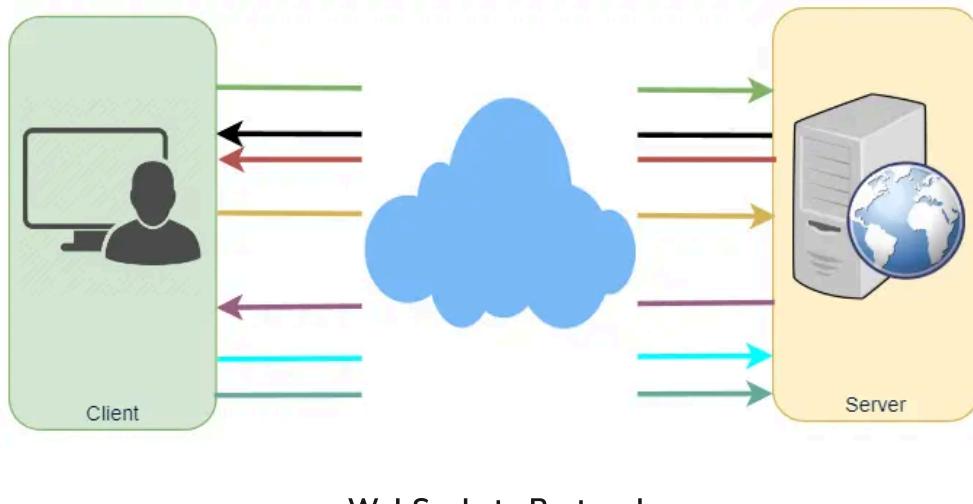
1. The client makes an initial request using regular HTTP and then waits for a response.
2. The server delays its response until an update is available or a timeout has occurred.
3. When an update is available, the server sends a full response to the client.
4. The client typically sends a new long-poll request, either immediately upon receiving a response or after a pause to allow an acceptable latency period.
5. Each Long-Poll request has a timeout. The client has to reconnect periodically after the connection is closed due to timeouts.



Long Polling Protocol

WebSockets

WebSocket provides **Full duplex** communication channels over a single TCP connection. It provides a persistent connection between a client and a server that both parties can use to start sending data at any time. The client establishes a WebSocket connection through a process known as the WebSocket handshake. If the process succeeds, then the server and client can exchange data in both directions at any time. The WebSocket protocol enables communication between a client and a server with lower overheads, facilitating real-time data transfer from and to the server. This is made possible by providing a standardized way for the server to send content to the browser without being asked by the client and allowing for messages to be passed back and forth while keeping the connection open. In this way, a two-way (bi-directional) ongoing conversation can take place between a client and a server.

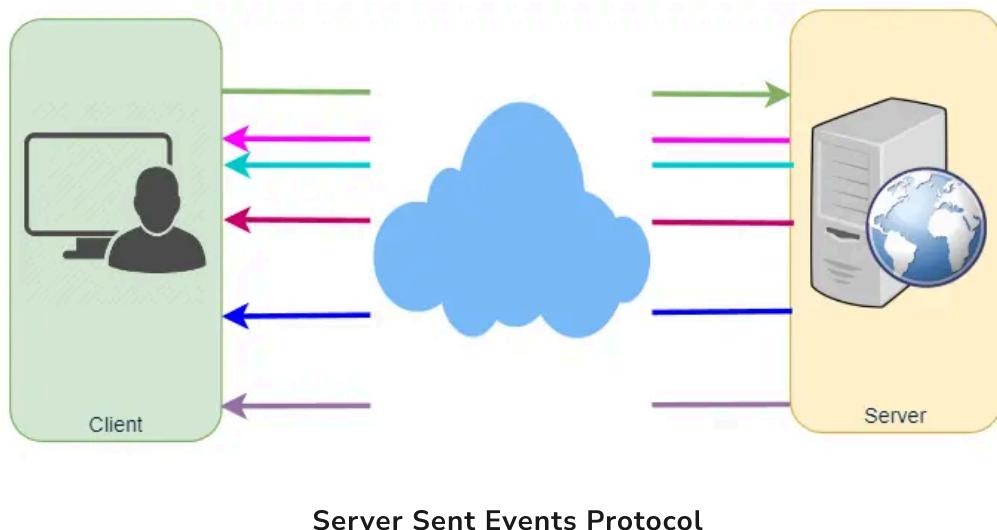


Server-Sent Events (SSEs)

Under SSEs the client establishes a persistent and long-term connection with the server. The server uses this connection to send data to a client. If the client wants to send data to the server, it would require the use of another technology/protocol to do so.

1. Client requests data from a server using regular HTTP.
2. The requested webpage opens a connection to the server.
3. The server sends the data to the client whenever there's new information available.

SSEs are best when we need real-time traffic from the server to the client or if the server is generating data in a loop and will be sending multiple events to the client.



Bloom Filters (New)

Let's learn about Bloom filters and how to use them.

Background

If we have a large set of structured data (identified by record IDs) stored in a set of data files, what is the most efficient way to know which file might contain our required data? We don't want to read each file, as that would be slow, and we have to read a lot of data from the disk. One solution can be to build an index on each data file and store it in a separate index file. This index can map each record ID to its offset in the data file. Each index file will be sorted on the record ID. Now, if we want to search an ID in this index, the best we can do is a Binary Search. Can we do better than that?

Solution

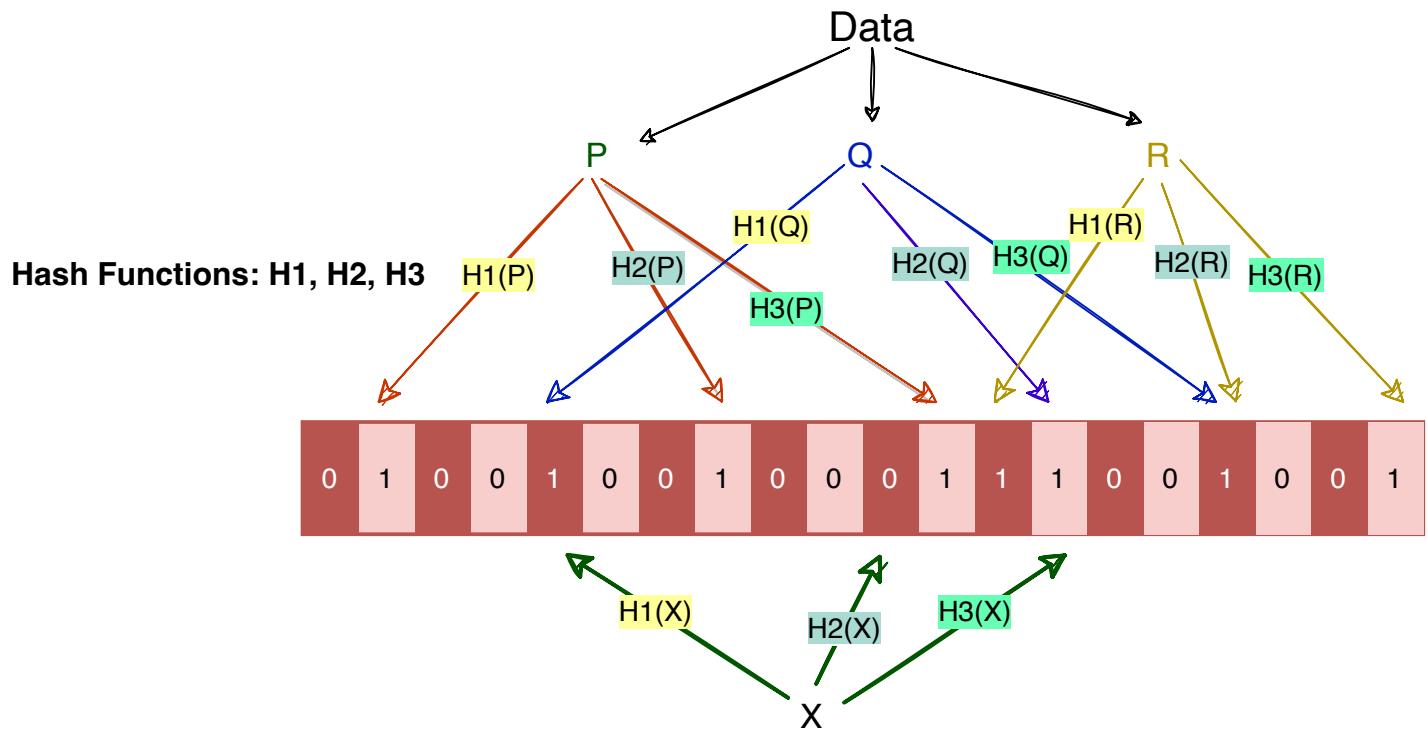
Use Bloom filters to quickly find if an element might be present in a set.

The Bloom filter data structure tells whether an element **may be in a set, or definitely is not**. The only possible errors are false positives, i.e., a search for a nonexistent element might give an incorrect answer. With more elements in the filter, the error rate increases. An empty Bloom filter is a bit-array of m bits, all set to 0. There are also k different hash functions, each of which maps a set element to one of the m bit positions.

- To add an element, feed it to the hash functions to get k bit positions, and set the bits at these positions to 1.

- To test if an element is in the set, feed it to the hash functions to get k bit positions.
- If any of the bits at these positions is 0, the element is **definitely not** in the set.
- If all are 1, then the element **may be** in the set.

Here is a Bloom filter with three elements P , Q , and R . It consists of 20 bits and uses three hash functions. The colored arrows point to the bits that the elements of the set are mapped to.



- The element X definitely is not in the set, since it hashes to a bit position containing 0.
- For a fixed error rate, adding a new element and testing for membership are both constant time operations, and a filter with room for ' n ' elements requires $O(n)$ space.

Quorum (New)

Let's learn about Quorum and its usage.

Background

In Distributed Systems, data is replicated across multiple servers for fault tolerance and high availability. Once a system decides to maintain multiple copies of data, another problem arises: how to make sure that all replicas are consistent, i.e., if they all have the latest copy of the data and that all clients see the same view of the data?

Solution

In a distributed environment, a quorum is the minimum number of servers on which a distributed operation needs to be performed successfully before declaring the operation's overall success.

Suppose a database is replicated on five machines. In that case, quorum refers to the minimum number of machines that perform the same action (commit or abort) for a given transaction in order to decide the final operation for that transaction. So, in a set of 5 machines, three machines form the majority quorum, and if they agree, we will commit that operation. Quorum enforces the consistency requirement needed for distributed operations.

In systems with multiple replicas, there is a possibility that the user reads inconsistent data. For example, when there are three replicas, R1 , R2 , and R3 in a cluster, and a user writes value v1 to replica R1 . Then another user reads from

replica R2 or R3 which are still behind R1 and thus will not have the value v1, so the second user will not get the consistent state of data.

What value should we choose for a quorum? More than half of the number of nodes in the cluster: $(N/2 + 1)$ where N is the total number of nodes in the cluster, for example:

- In a 5-node cluster, three nodes must be online to have a majority.
- In a 4-node cluster, three nodes must be online to have a majority.
- With 5-node, the system can afford two node failures, whereas, with 4-node, it can afford only one node failure. Because of this logic, it is recommended to always have an odd number of total nodes in the cluster.

Quorum is achieved when nodes follow the below protocol: $R + W > N$, where:

N = nodes in the quorum group

W = minimum write nodes

R = minimum read nodes

If a distributed system follows $R + W > N$ rule, then every read will see at least one copy of the latest value written. For example, a common configuration could be ($N=3$, $W=2$, $R=2$) to ensure strong consistency. Here are a couple of other examples:

- ($N=3$, $W=1$, $R=3$): fast write, slow read, not very durable
- ($N=3$, $W=3$, $R=1$): slow write, fast read, durable

The following two things should be kept in mind before deciding read/write quorum:

- $R=1$ and $W=N \Rightarrow$ full replication (write-all, read-one): undesirable when servers can be unavailable because writes are not guaranteed to complete.
- Best performance (throughput/availability) when $1 < r < w < n$, because reads are more frequent than writes in most applications

How It Works

- **Majority-Based Quorum:** The most common type of quorum where an operation requires a majority (more than half) of the nodes to agree or participate. For instance, in a system with 5 nodes, at least 3 must agree for a decision to be made.
- **Read and Write Quorums:** For read and write operations, different quorum sizes can be defined. For example, a system might require a write quorum of 3 nodes and a read quorum of 2 nodes in a 5-node cluster.

Use Cases

Distributed Databases

- Ensuring consistency in a database cluster, where multiple nodes might hold copies of the same data.

Cluster Management

- In server clusters, a quorum decides which nodes form the 'active' cluster, especially important for avoiding 'split-brain' scenarios where a cluster might be divided into two parts, each believing it is the active cluster.

Consensus Protocols

- In algorithms like Paxos or Raft, a quorum is crucial for achieving consensus among distributed nodes regarding the state of the system or the outcome of an

operation.

Advantages

1. **Fault Tolerance:** Allows the system to tolerate a certain number of failures while still operating correctly.
2. **Consistency:** Helps maintain data consistency across distributed nodes.
3. **Availability:** Increases the availability of the system by allowing operations to proceed as long as the quorum condition is met.

Challenges

1. **Network Partitions:** In cases of network failures, forming a quorum might be challenging, impacting system availability.
2. **Performance Overhead:** Achieving a quorum, especially in large clusters, can introduce latency in decision-making processes.
3. **Complexity:** Implementing and managing quorum-based systems can be complex, particularly in dynamic environments with frequent node or network changes.

Conclusion

Quorum is a fundamental concept in distributed systems, playing a crucial role in ensuring consistency, reliability, and availability in environments where multiple nodes work together. While it enhances fault tolerance, it also introduces additional complexity and requires careful design and management to balance consistency, availability, and performance.

Leader and Follower (New)

Let's learn about the leader and follower patterns and its usage in distributed systems.

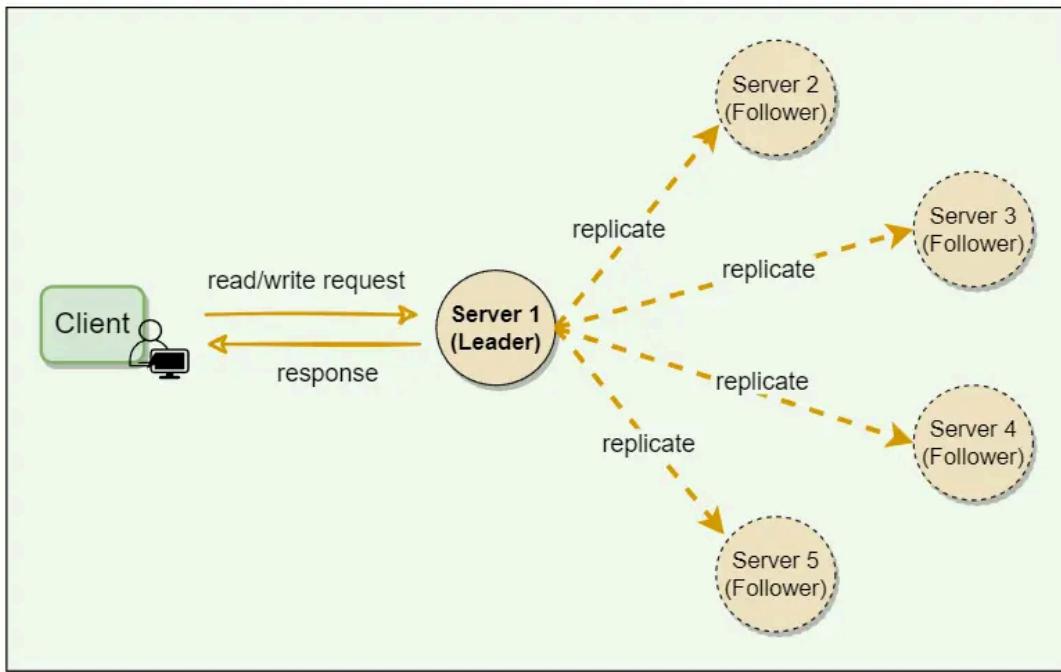
Background

Distributed systems keep multiple copies of data for fault tolerance and higher availability. A system can use quorum to ensure data consistency between replicas, i.e., all reads and writes are not considered successful until a majority of nodes participate in the operation. However, using quorum can lead to another problem, that is, lower availability; at any time, the system needs to ensure that at least a majority of replicas are up and available, otherwise the operation will fail. Quorum is also not sufficient, as in certain failure scenarios, the client can still see inconsistent data.

Solution

Allow only a single server (called leader) to be responsible for data replication and to coordinate work.

At any time, one server is elected as the leader. This leader becomes responsible for data replication and can act as the central point for all coordination. The followers only accept writes from the leader and serve as a backup. In case the leader fails, one of the followers can become the leader. In some cases, the follower can serve read requests for load balancing.



Leader entertains requests from the client and is responsible for replicating and coordinating with followers

Heartbeat (New)

Let's learn about the heartbeat and its usage.

Background

In a distributed environment, work/data is distributed among servers. To efficiently route requests in such a setup, servers need to know what other servers are part of the system. Furthermore, servers should know if other servers are alive and working. In a decentralized system, whenever a request arrives at a server, the server should have enough information to decide which server is responsible for entertaining that request. This makes the timely detection of server failure an important task, which also enables the system to take corrective actions and move the data/work to another healthy server and stop the environment from further deterioration.

Solution

Each server periodically sends a heartbeat message to a central monitoring server or other servers in the system to show that it is still alive and functioning.

Heartbeating is one of the mechanisms for detecting failures in a distributed system. If there is a central server, all servers periodically send a heartbeat message to it. If there is no central server, all servers randomly choose a set of servers and send them a heartbeat message every few seconds. This way, if no heartbeat message is received from a server for a while, the system can suspect that the server might have crashed. If there is no heartbeat within a configured

timeout period, the system can conclude that the server is not alive anymore and stop sending requests to it and start working on its replacement.

Checksum (New)

Let's learn about checksum and its usage.

Background

In a distributed system, while moving data between components, it is possible that the data fetched from a node may arrive corrupted. This corruption can occur because of faults in a storage device, network, software, etc. How can a distributed system ensure data integrity, so that the client receives an error instead of corrupt data?

Solution

Calculate a checksum and store it with data.

To calculate a checksum, a cryptographic hash function like MD5, SHA-1, SHA-256, or SHA-512 is used. The hash function takes the input data and produces a string (containing letters and numbers) of fixed length; this string is called the checksum.

When a system is storing some data, it computes a checksum of the data and stores the checksum with the data. When a client retrieves data, it verifies that the data it received from the server matches the checksum stored. If not, then the client can opt to retrieve that data from another replica.

System Design Interviews - A step by step guide

Generally, software engineers have difficulty with system design interviews (SDIs) for three primary reasons:

- SDIs are unstructured, where candidates are asked to take on an open-ended design problem that doesn't have a standard solution.
- Candidates lack experience in developing complex and large-scale systems.
- Candidates did not spend enough time preparing for SDIs.

SDIs are similar to coding interviews in that candidates who don't prepare well tend to do poorly, particularly at high-profile companies like Google, Facebook, Amazon, and Microsoft. In these companies, candidates who do not perform above average have a limited chance to get an offer. On the other hand, a good performance always results in a better offer (a higher position and salary) since it proves the candidate's ability to handle a complex system.

In this course, we'll follow a step-by-step approach to solve multiple design problems. First, let's go through these steps:

Step 1: Requirements clarifications

It is always a good idea to ask questions about the exact scope of the problem we are trying to solve. Design questions are mostly open-ended, and they don't have ONE correct answer. That's why clarifying ambiguities early in the interview becomes critical. Candidates who spend enough time to define the end goals of the system always have a better chance to be successful in the interview. Also, since we only have 35-40 minutes to design a (supposedly) large system, we should clarify what parts of the system we will be focusing on.

Let's expand this with an actual example of designing a Twitter-like service. Here are some questions for designing Twitter that should be answered before moving on to the next steps:

- Will users of our service be able to post tweets and follow other people?
- Should we also design to create and display the user's timeline?
- Will tweets contain photos and videos?
- Are we focusing on the backend only, or are we developing the front-end too?
- Will users be able to search tweets?
- Do we need to display hot trending topics?
- Will there be any push notification for new (or important) tweets?

All such questions will determine what our end design will look like.

Step 2: Back-of-the-envelope estimation

It is always a good idea to estimate the scale of the system we're going to design. This will also help later when we focus on scaling, partitioning, load balancing, and caching.

- What scale is expected from the system (e.g., number of new tweets, number of tweet views, number of timeline generations per sec., etc.)?
- How much storage will we need? We will have different storage requirements if users can have photos and videos in their tweets.
- What network bandwidth usage are we expecting? This will be crucial in deciding how we will manage traffic and balance load between servers.

Step 3: System interface definition

Define what APIs are expected from the system. This will establish the exact contract expected from the system and ensure if we haven't gotten any requirements wrong. Some examples of APIs for our Twitter-like service will be:

```
postTweet(user_id, tweet_data, tweet_location,  
user_location, timestamp, ...)
```

```
generateTimeline(user_id, current_time, user_location, ...)
```

```
markTweetFavorite(user_id, tweet_id, timestamp, ...)
```

Step 4: Defining data model

Defining the data model in the early part of the interview will clarify how data will flow between different system components. Later, it will guide for data partitioning and management. The candidate should identify various system entities, how they will interact with each other, and different aspects of data management like storage, transportation, encryption, etc. Here are some entities for our Twitter-like service:

User: UserID, Name, Email, DoB, CreationDate, LastLogin, etc.

Tweet: TweetID, Content, TweetLocation, NumberOfLikes, TimeStamp, etc.

UserFollow: UserID1, UserID2

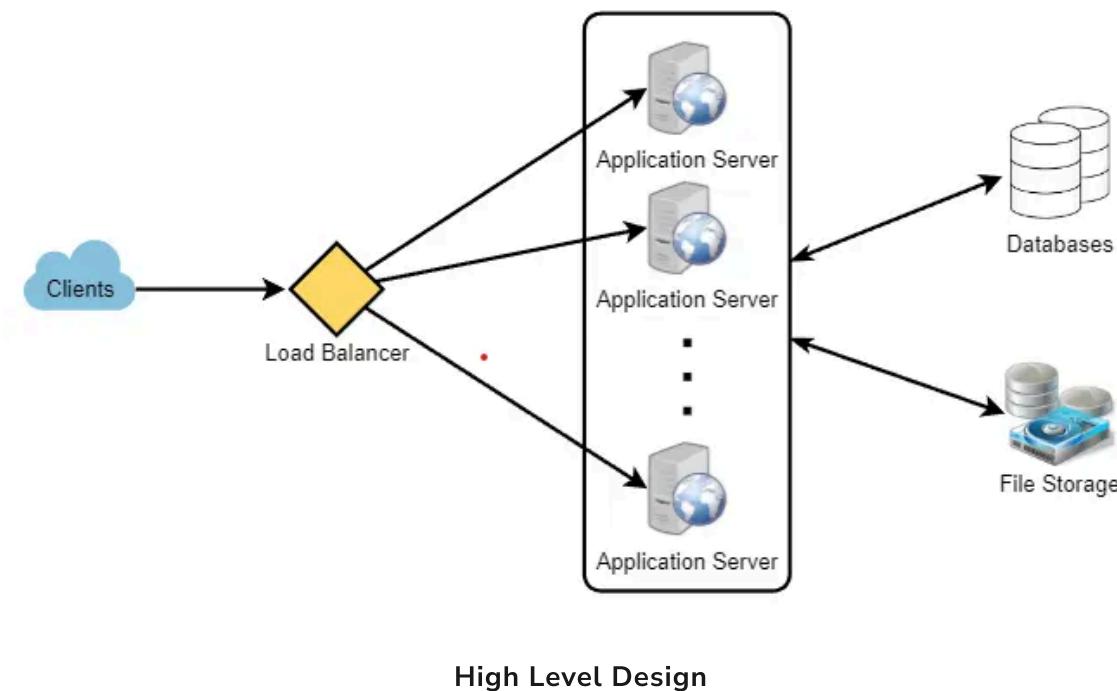
FavoriteTweets: UserID, TweetID, TimeStamp

Which database system should we use? Will NoSQL like [Cassandra](#) best fit our needs, or should we use a MySQL-like solution? What kind of block storage should we use to store photos and videos?

Step 5: High-level design

Draw a block diagram with 5-6 boxes representing the core components of our system. We should identify enough components that are needed to solve the actual problem from end to end.

For Twitter, at a high level, we will need multiple application servers to serve all the read/write requests with load balancers in front of them for traffic distributions. If we're assuming that we will have a lot more read traffic (compared to write), we can decide to have separate servers to handle these scenarios. On the back-end, we need an efficient database that can store all the tweets and support a large number of reads. We will also need a distributed file storage system for storing photos and videos.



Step 6: Detailed design

Dig deeper into two or three major components; the interviewer's feedback should always guide us to what parts of the system need further discussion. We should present different approaches, their pros and cons, and explain why we will prefer one approach over the other. Remember, there is no single answer; the only important thing is to consider tradeoffs between different options while keeping system constraints in mind.

- Since we will be storing a massive amount of data, how should we partition our data to distribute it to multiple databases? Should we try to store all the data of a user on the same database? What issue could it cause?
- How will we handle hot users who tweet a lot or follow lots of people?
- Since users' timeline will contain the most recent (and relevant) tweets, should we try to store our data so that it is optimized for scanning the latest tweets?
- How much and at which layer should we introduce cache to speed things up?
- What components need better load balancing?

Step 7: Identifying and resolving bottlenecks

Try to discuss as many bottlenecks as possible and different approaches to mitigate them.

- Is there any single point of failure in our system? What are we doing to mitigate it?
- Do we have enough replicas of the data so that we can still serve our users if we lose a few servers?
- Similarly, do we have enough copies of different services running such that a few failures will not cause a total system shutdown?

- How are we monitoring the performance of our service? Do we get alerts whenever critical components fail or their performance degrades?

Summary

In short, preparation and being organized during the interview are the keys to success in system design interviews. The steps mentioned above should guide you to remain on track and cover all the different aspects while designing a system.

Let's apply the above guidelines to design a few systems that are asked in SDIs.

Happy learning!

Design Guru's team

System Design Master Template

System design interviews are unstructured by design. In these interviews, you are asked to take on an open-ended design problem that doesn't have a standard solution.

The two biggest challenges of answering a system design interview question are:

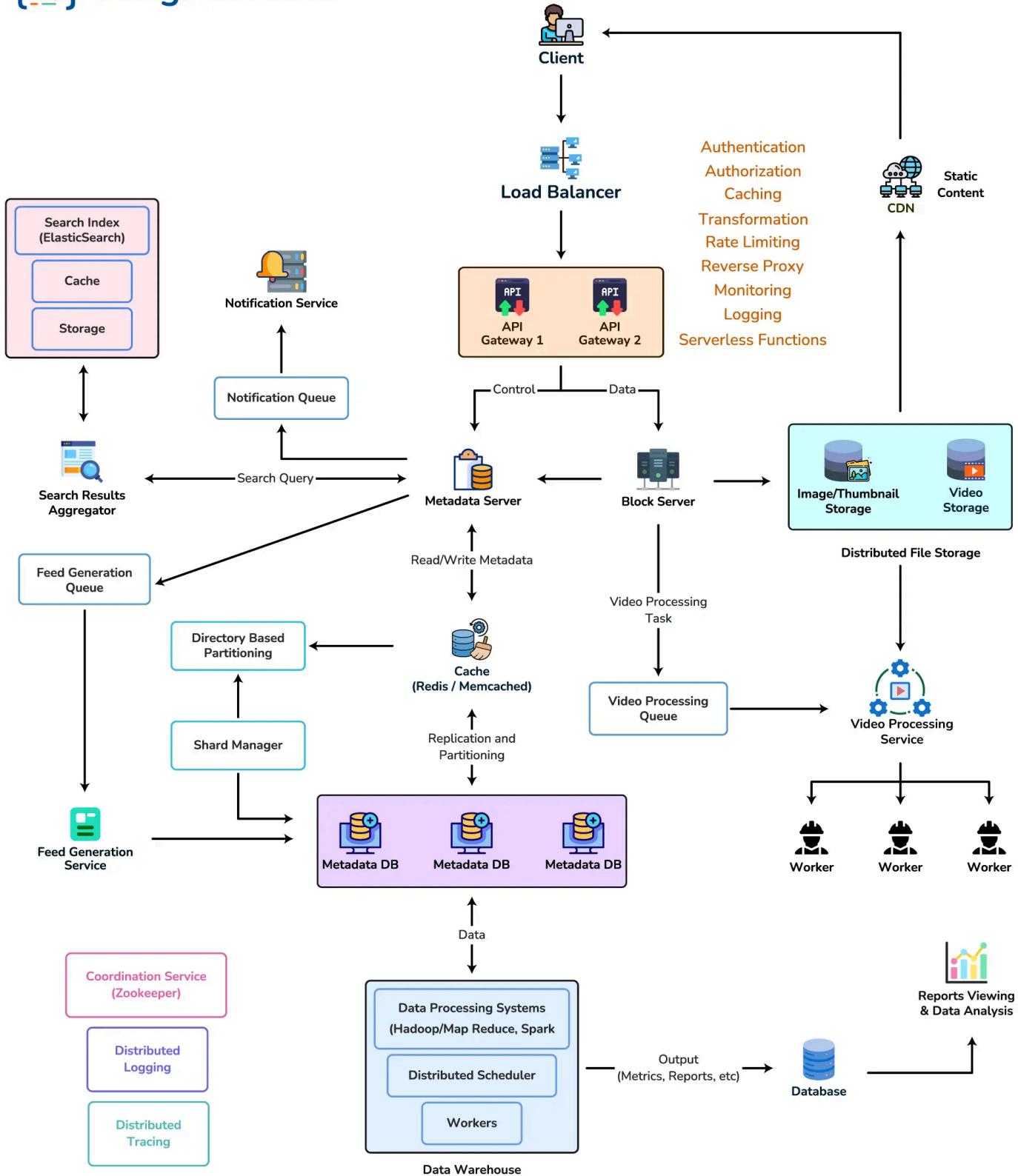
1. To know where to start.
2. To know if you have talked about all the important parts of the system.

To simplify this process, this course offers a comprehensive system design template that can effectively guide you in addressing any system design interview question.

Have a look at the following image to understand the major components that could be part of any system design and how these components interact with each other.

System Design Master Template

{ } DesignGurus.io



System Design Master Template

System Design Master Template (video)

Here is a video discussing the System Design Master Template:

1:37:59

System Design Master Template

With this master template in mind, we will discuss the 18 essential system design concepts. Here is a brief description of each:

1. Domain Name System (DNS)

The Domain Name System (DNS) serves as a fundamental component of the internet infrastructure, translating user-friendly domain names into their corresponding IP addresses. It acts as a phonebook for the internet, enabling users to access websites and services by entering easily memorable domain names, such as www.designgurus.io, rather than the numerical IP addresses like "192.0.2.1" that computers utilize to identify each other.

When you input a domain name into your web browser, the DNS is responsible for finding the associated IP address and directing your request to the appropriate server. This process commences with your computer sending a query to a recursive resolver, which then searches a series of DNS servers, beginning with the root server, followed by the Top-Level Domain (TLD) server, and ultimately the authoritative name server. Once the IP address is located, the recursive resolver returns it to your computer, allowing your browser to establish a connection with the target server and access the desired content.



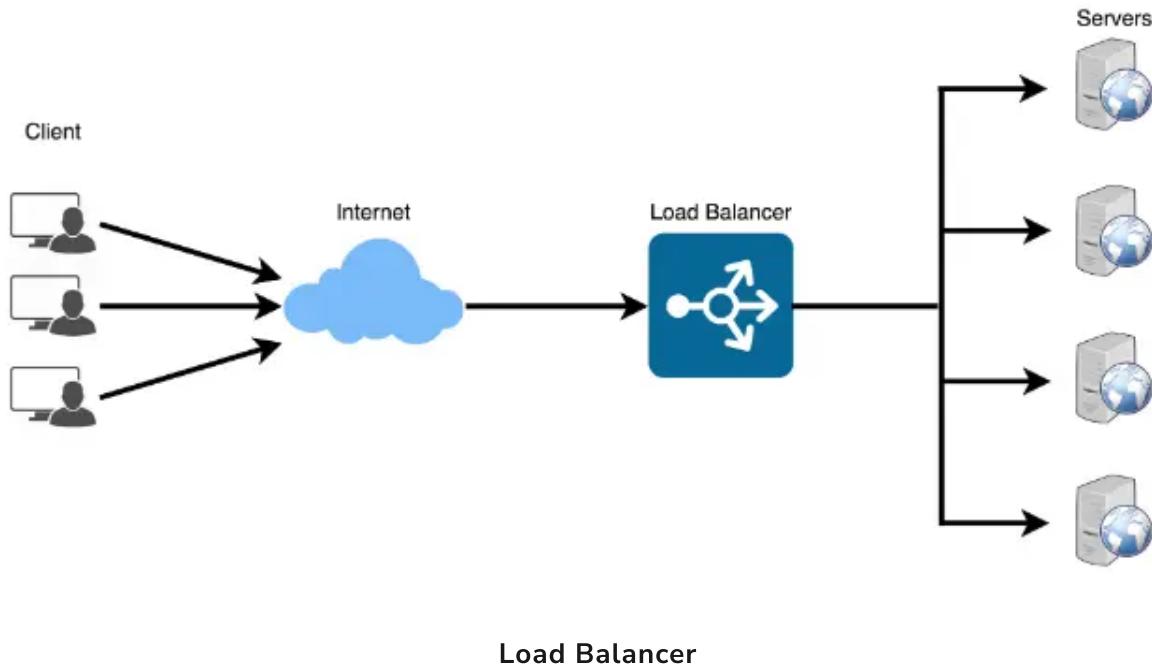
DNS

2. Load Balancer

A load balancer is a networking device or software designed to distribute incoming network traffic across multiple servers, ensuring optimal resource utilization, reduced latency, and maintained high availability. It plays a crucial role in scaling applications and efficiently managing server workloads, particularly in situations where there is a sudden surge in traffic or uneven distribution of requests among servers.

Load balancers employ various algorithms to determine the distribution of incoming traffic. Some common algorithms include:

- **Round Robin:** Requests are sequentially and evenly distributed across all available servers in a cyclical manner.
- **Least Connections:** The load balancer assigns requests to the server with the fewest active connections, giving priority to less-busy servers.
- **IP Hash:** The client's IP address is hashed, and the resulting value is used to determine which server the request should be directed to. This method ensures that a specific client's requests are consistently routed to the same server, helping maintain session persistence.



3. API Gateway

An API Gateway serves as a server or service that functions as an intermediary between external clients and the internal microservices or API-based backend services of an application. It is a vital component in contemporary architectures, particularly in microservices-based systems, where it streamlines the communication process and offers a single entry point for clients to access various services.

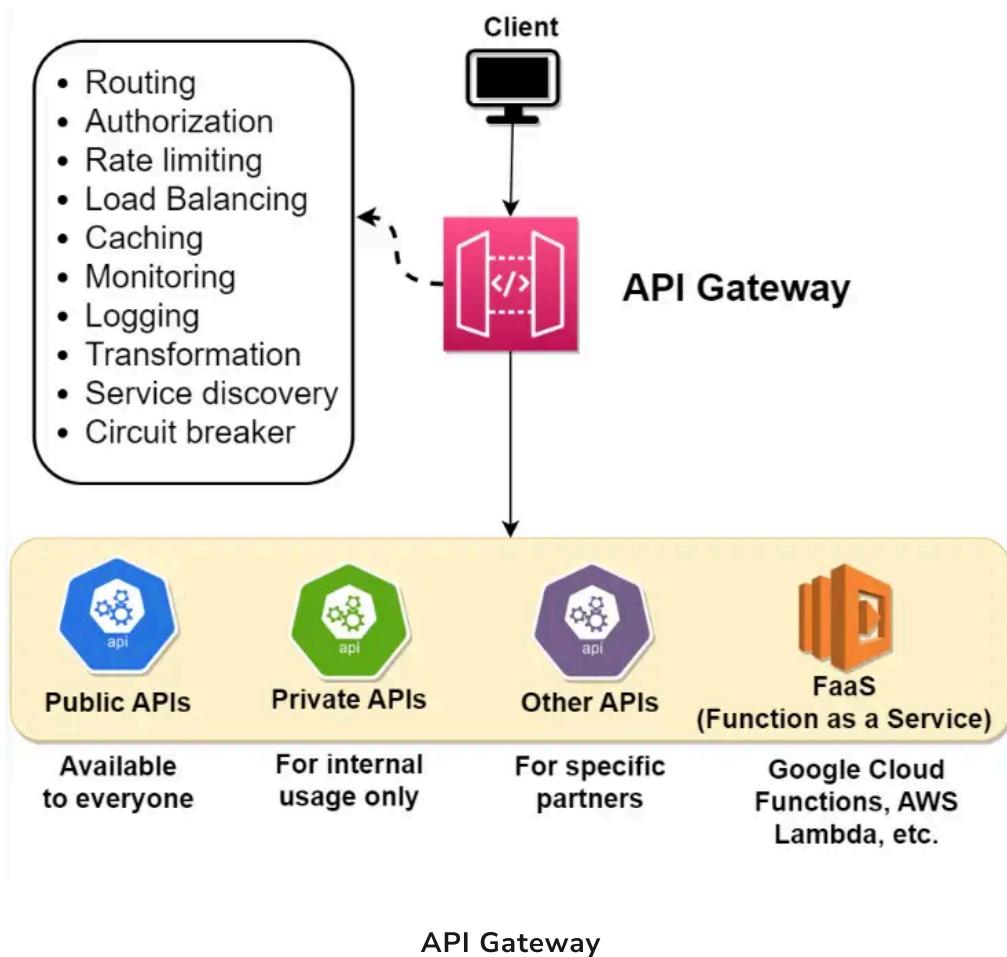
The primary functions of an API Gateway encompass:

1. Request Routing: The API Gateway directs incoming API requests from clients to the appropriate backend service or microservice, based on predefined rules and configurations.
2. Authentication and Authorization: The API Gateway manages user authentication and authorization, ensuring that only authorized clients can

access the services. It verifies API keys, tokens, or other credentials before routing requests to the backend services.

3. Rate Limiting and Throttling: To safeguard backend services from excessive load or abuse, the API Gateway enforces rate limits or throttles requests from clients according to predefined policies.
4. Caching: In order to minimize latency and backend load, the API Gateway caches frequently-used responses, serving them directly to clients without the need to query the backend services.
5. Request and Response Transformation: The API Gateway can modify requests and responses, such as converting data formats, adding or removing headers, or altering query parameters, to ensure compatibility between clients and services.

Is it Possible to Use a Load Balancer and an API Gateway Together? Yes, in many real-world architectures, both of these components work together, where the Load Balancer effectively manages traffic across multiple instances of API Gateways or directly to services, depending on the setup. See details [here](#).



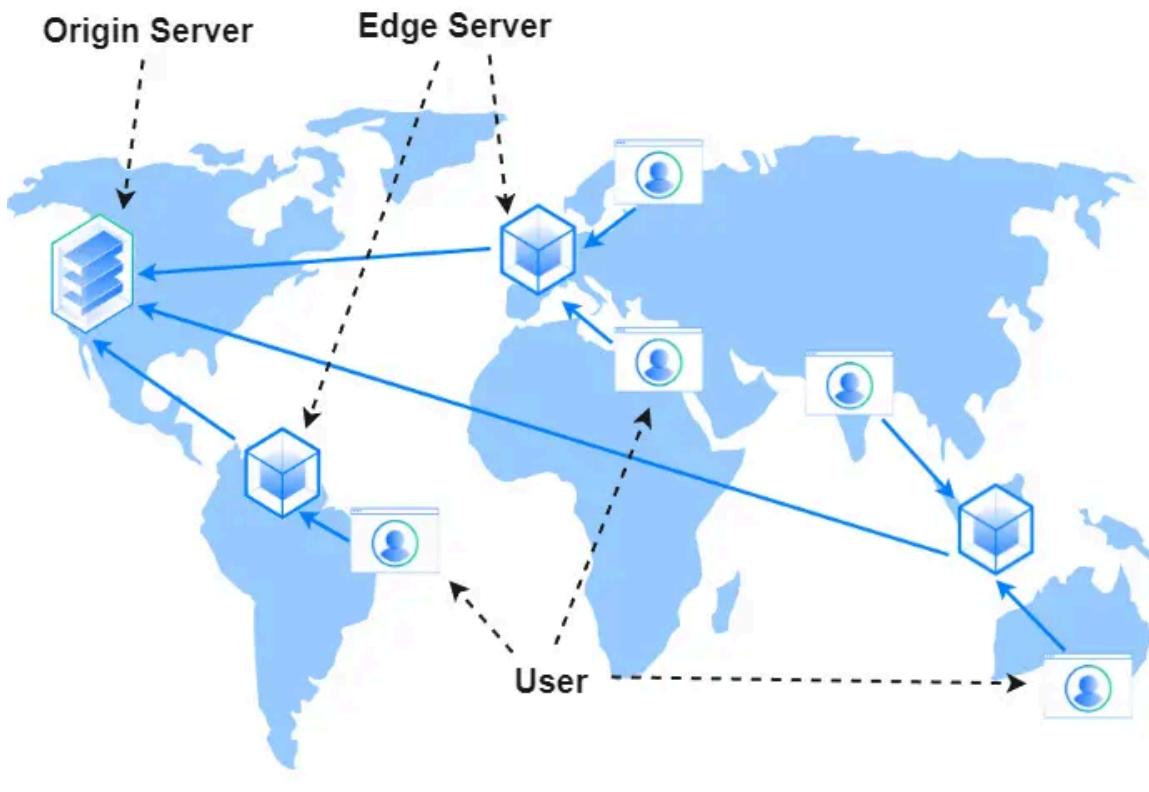
4. CDN

A Content Delivery Network (CDN) is a distributed network of servers that store and deliver content, such as images, videos, stylesheets, and scripts, to users from locations that are geographically closer to them. CDNs are designed to enhance the performance, speed, and reliability of content delivery to end-users, irrespective of their location relative to the origin server. Here's how a CDN operates:

1. When a user requests content from a website or application, the request is directed to the nearest CDN server, also known as an edge server.
2. If the edge server has the requested content cached, it directly serves the content to the user. This process reduces latency and improves the user

experience, as the content travels a shorter distance.

3. If the content is not cached on the edge server, the CDN retrieves it from the origin server or another nearby CDN server. Once the content is fetched, it is cached on the edge server and served to the user.
4. To ensure the content stays up-to-date, the CDN periodically checks the origin server for changes and updates its cache accordingly.



Content Delivery Network (CDN)

CDN

5. Forward Proxy vs. Reverse Proxy

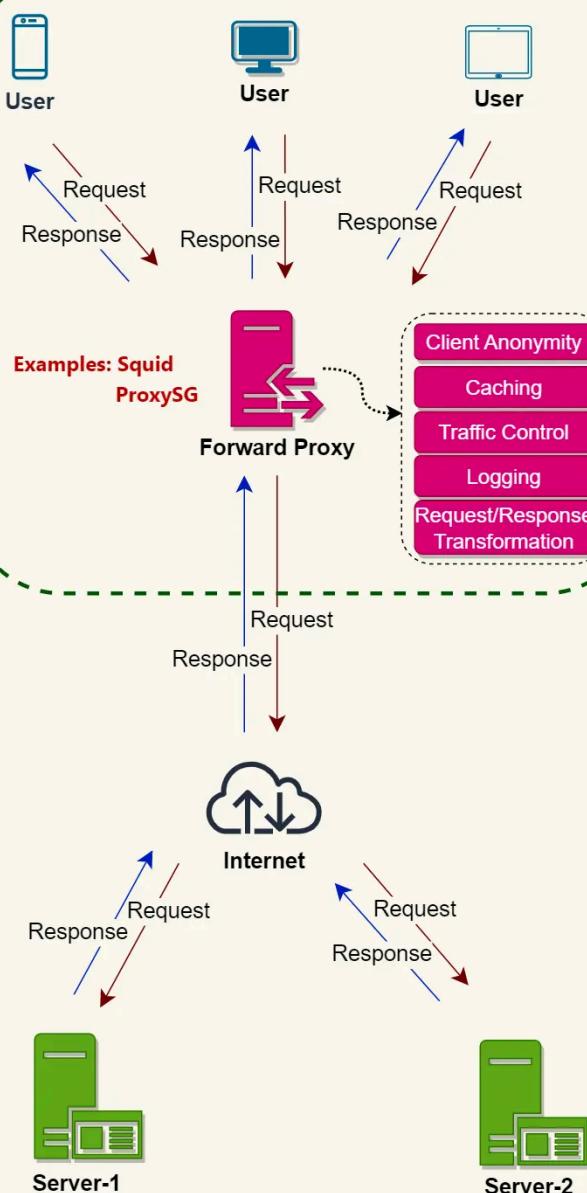
A forward proxy, also referred to as a "proxy server" or simply "proxy," is a server positioned in front of one or more client machines, acting as an intermediary between the clients and the internet. When a client machine requests a resource on

the internet, the request is initially sent to the forward proxy. The forward proxy then forwards the request to the internet on behalf of the client machine and returns the response to the client machine.

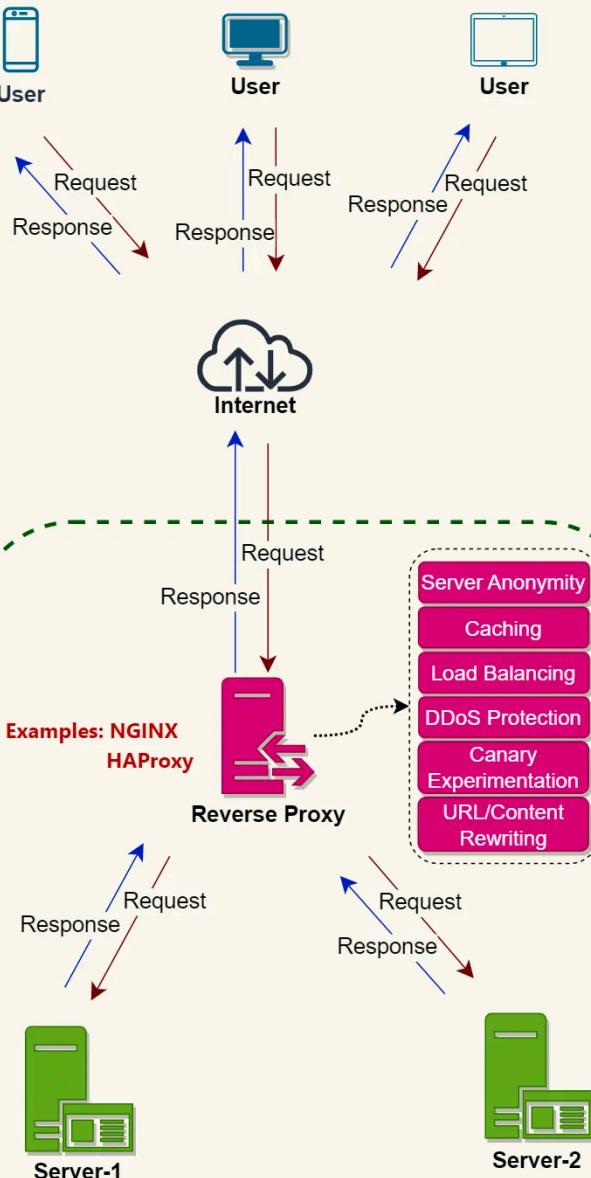
On the other hand, a reverse proxy is a server that sits in front of one or more web servers, serving as an intermediary between the web servers and the internet. When a client requests a resource on the internet, the request is first sent to the reverse proxy. The reverse proxy then forwards the request to one of the web servers, which returns the response to the reverse proxy. Finally, the reverse proxy returns the response to the client.

Forward Proxy vs. Reverse Proxy

Forward Proxy



Reverse Proxy



DesignGurus.org (one stop portal for coding and system design interviews)

Forward Proxy vs. Reverse Proxy

6. Caching

Cache is a high-speed storage layer positioned between the application and the original data source, such as a database, file system, or remote web service. When an application requests data, the cache is checked first. If the data is present in the cache, it is returned to the application. If the data is not found in the cache, it is retrieved from its original source, stored in the cache for future use, and then returned to the application. In a distributed system, caching can occur in multiple locations, including the client, DNS, CDN, load balancer, API gateway, server, database, and more.



Client Cache

Use Case: Faster retrieval of web content.

Solutions: Browser Cache



DNS Cache

Use Case: Faster domain to IP resolution.

Solutions: Amazon Route 53, Azure DNS, Google Cloud DNS



CDN Cache

Use Case: Faster retrieval of static content.

Solutions: Akamai, CloudFront, ElastiCache, Azure CDN



Web Server Cache

Use Case: Faster retrieval of web content.

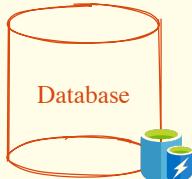
Solutions: CloudFront, ElastiCache



App Server Cache

Use Case: Accelerated application performance and data access.

Solutions: Local server cache, Remote cache on Redis, Memcached, ElastiCache



Database Cache

Use Case: Faster access to data stored.

Solutions: Local DB cache, Remote cache on Redis, Memcached, ElastiCache, etc.

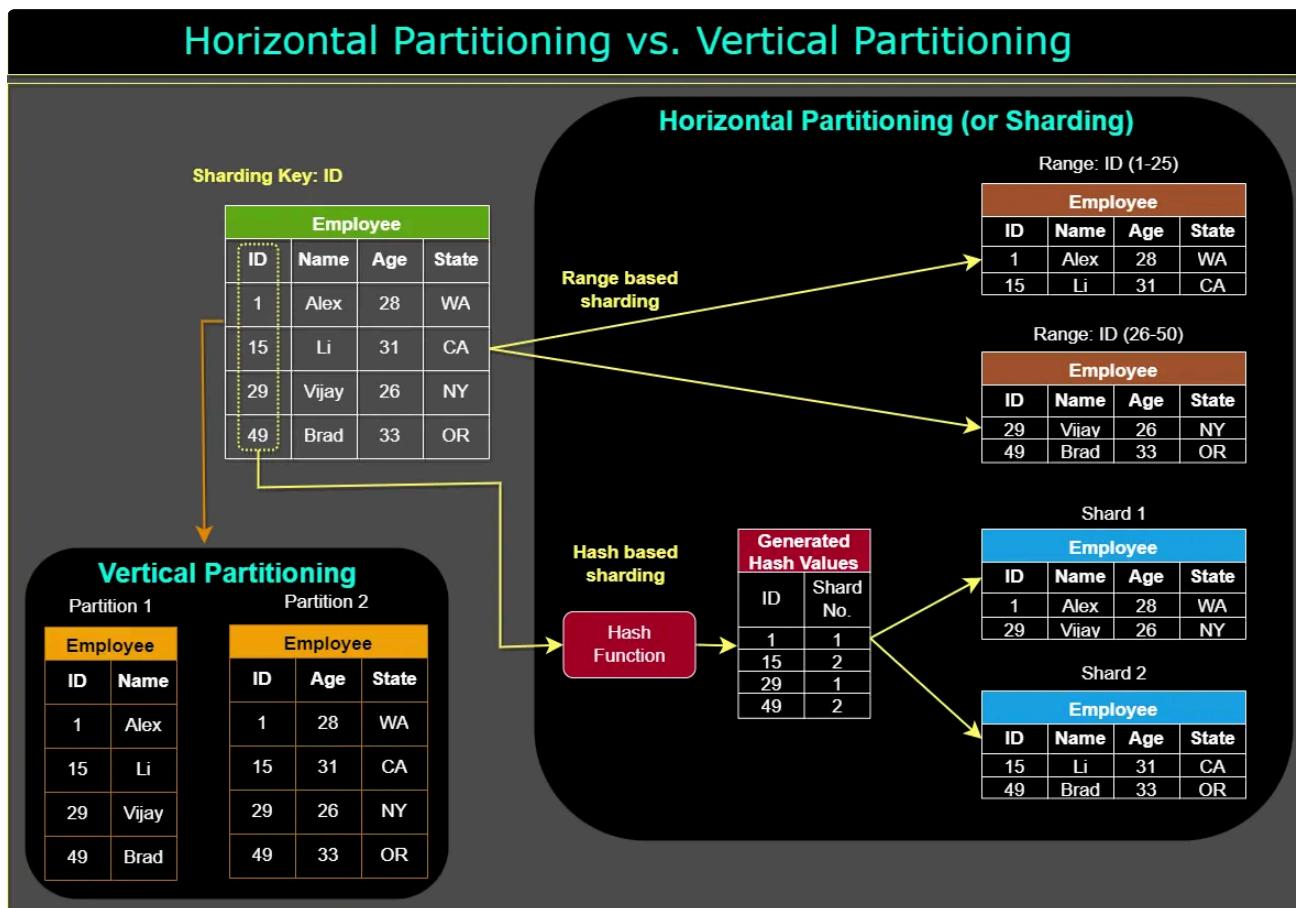
Cache

7. Data Partitioning

In a database, **horizontal partitioning**, often referred to as **sharding**, entails dividing the rows of a table into smaller tables and storing them on distinct servers or

database instances. This method is employed to distribute the database load across multiple servers, thereby enhancing performance.

Conversely, **vertical partitioning** involves splitting the columns of a table into separate tables. This technique aims to reduce the column count in a table and boost the performance of queries that only access a limited number of columns.



8. Database Replication

Database replication is a method employed to maintain multiple copies of the same database across various servers or locations. The main objective of database replication is to enhance data availability, redundancy, and fault tolerance,

ensuring the system remains operational even in the face of hardware failures or other issues.

In a replicated database configuration, one server serves as the primary (or master) database, while others act as replicas (or slaves). This process involves synchronizing data between the primary database and replicas, ensuring all possess the same up-to-date information. Database replication provides several advantages, including:

1. Improved Performance: By distributing read queries among multiple replicas, the load on the primary database can be reduced, leading to faster query response times.
2. High Availability: If the primary database experiences failure or downtime, replicas can continue to provide data, ensuring uninterrupted access to the application.
3. Enhanced Data Protection: Maintaining multiple copies of the database across different locations helps safeguard against data loss due to hardware failures or other disasters.
4. Load Balancing: Replicas can handle read queries, allowing for better load distribution and reducing overall stress on the primary database.

9. Distributed Messaging Systems

Distributed messaging systems provide a reliable, scalable, and fault-tolerant means for exchanging messages between numerous, possibly geographically-dispersed applications, services, or components. These systems facilitate communication by decoupling sender and receiver components, enabling them to develop and function independently. Distributed messaging systems are especially valuable in large-scale or intricate systems, like those seen in microservices architectures or distributed computing environments. Examples of these systems include Apache Kafka and RabbitMQ.

10. Microservices

Microservices represent an architectural style wherein an application is organized as an assembly of small, loosely-coupled, and autonomously deployable services. Each microservice is accountable for a distinct aspect of functionality or domain within the application and communicates with other microservices via well-defined APIs. This method deviates from the conventional monolithic architecture, where an application is constructed as a single, tightly-coupled unit.

The primary characteristics of microservices include:

1. Single Responsibility: Adhering to the Single Responsibility Principle, each microservice focuses on a specific function or domain, making the services more straightforward to comprehend, develop, and maintain.
2. Independence: Microservices can be independently developed, deployed, and scaled, offering increased flexibility and agility in the development process. Teams can work on various services simultaneously without impacting the entire system.
3. Decentralization: Typically, microservices are decentralized, with each service possessing its data and business logic. This approach fosters separation of concerns and empowers teams to make decisions and select technologies tailored to their unique requirements.
4. Communication: Microservices interact with each other using lightweight protocols, such as HTTP/REST, gRPC, or message queues. This fosters interoperability and facilitates the integration of new services or the replacement of existing ones.
5. Fault Tolerance: As microservices are independent, the failure of one service does not necessarily result in the collapse of the entire system, enhancing the application's overall resiliency.

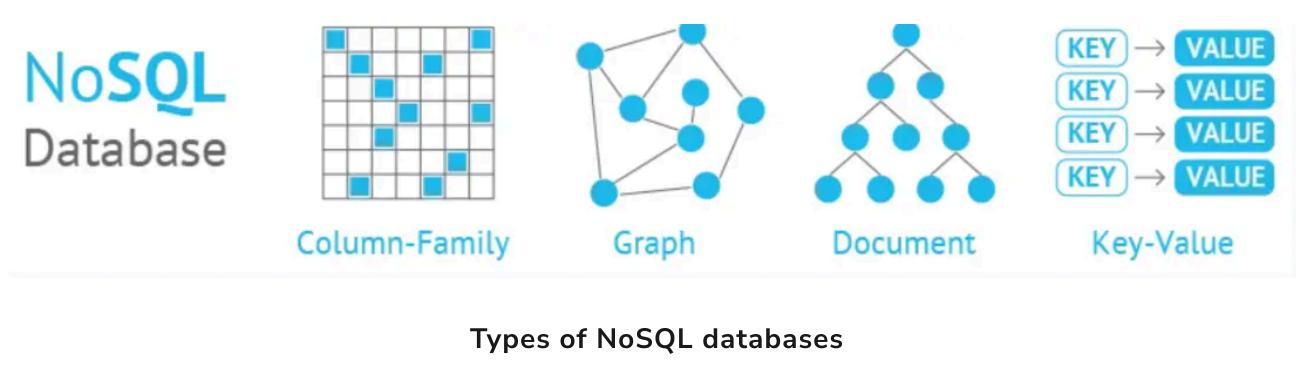
11. NoSQL Databases

NoSQL databases, or “Not Only SQL” databases, are non-relational databases designed to store, manage, and retrieve unstructured or semi-structured data. They offer an alternative to traditional relational databases, which rely on structured data and predefined schemas. NoSQL databases have become popular due to their flexibility, scalability, and ability to handle large volumes of data, making them well-suited for modern applications, big data processing, and real-time analytics.

NoSQL databases can be categorized into four main types:

1. Document-Based: These databases store data in document-like structures, such as JSON or BSON. Each document is self-contained and can have its own unique structure, making them suitable for handling heterogeneous data. Examples of document-based NoSQL databases include MongoDB and Couchbase.
2. Key-Value: These databases store data as key-value pairs, where the key acts as a unique identifier, and the value holds the associated data. Key-value databases are highly efficient for simple read and write operations, and they can be easily partitioned and scaled horizontally. Examples of key-value NoSQL databases include Redis and Amazon DynamoDB.
3. Column-Family: These databases store data in column families, which are groups of related columns. They are designed to handle write-heavy workloads and are highly efficient for querying data with a known row and column keys. Examples of column-family NoSQL databases include Apache Cassandra and HBase.
4. Graph-Based: These databases are designed for storing and querying data that has complex relationships and interconnected structures, such as social networks or recommendation systems. Graph databases use nodes, edges, and properties to represent and store data, making it easier to perform complex

traversals and relationship-based queries. Examples of graph-based NoSQL databases include Neo4j and Amazon Neptune.



12. Database Index

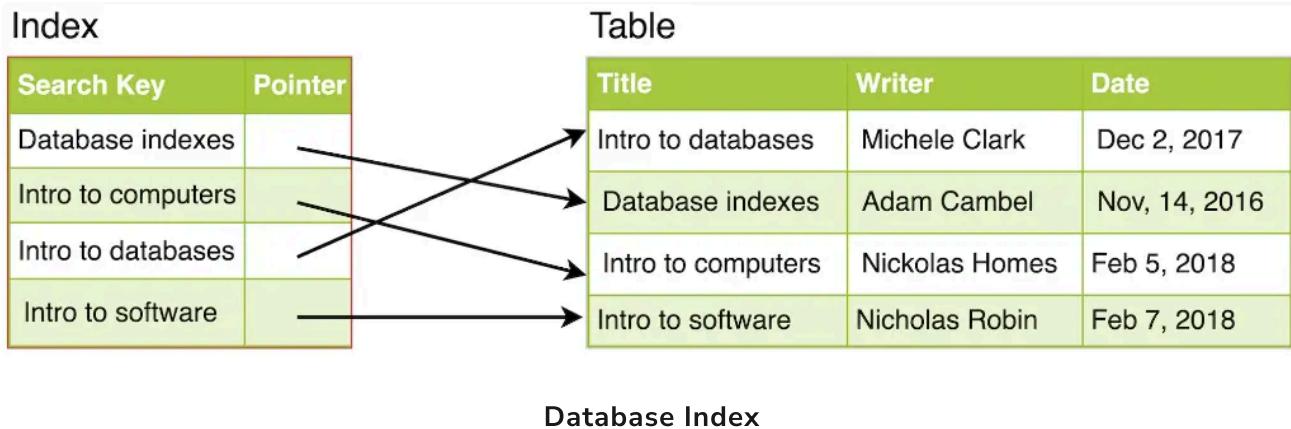
Database indexes are data structures that enhance the speed and efficiency of query operations within a database. They function similarly to an index in a book, enabling the database management system (DBMS) to swiftly locate data associated with a specific value or group of values, without the need to search through every row in a table. By offering a more direct route to the desired data, indexes can considerably decrease the time required to retrieve information from a database.

Indexes are typically constructed on one or more columns of a database table. The B-tree index is the most prevalent type, organizing data in a hierarchical tree structure, which allows for rapid search, insertion, and deletion operations. Other types of indexes, such as bitmap indexes and hash indexes, exist as well, each with their particular use cases and advantages.

Although indexes can significantly enhance query performance, they also involve certain trade-offs:

- **Storage Space:** Indexes require additional storage space since they generate and maintain separate data structures alongside the original table data.

- **Write Performance:** When data is inserted, updated, or deleted in a table, the corresponding indexes must also be updated, which may slow down write operations.



13. Distributed File Systems

Distributed file systems are storage systems designed to manage and grant access to files and directories across multiple servers, nodes, or machines, frequently distributed across a network. They allow users and applications to access and modify files as though they were situated on a local file system, despite the fact that the actual files may be physically located on various remote servers. Distributed file systems are commonly employed in large-scale or distributed computing environments to offer fault tolerance, high availability, and enhanced performance.

14. Notification System

These are used to send notifications or alerts to users, such as emails, push notifications, or text messages.

15. Full-text Search

Full-text search allows users to search for particular words or phrases within an application or website. Upon receiving a user query, the application or website delivers the most relevant results. To accomplish this rapidly and effectively, full-text search utilizes an inverted index, a data structure that associates words or phrases with the documents where they are found. Elastic Search is an example of such systems.

16. Distributed Coordination Services

Distributed coordination services are systems engineered to regulate and synchronize the actions of distributed applications, services, or nodes in a dependable, efficient, and fault-tolerant way. They assist in maintaining consistency, addressing distributed synchronization, and overseeing the configuration and state of diverse components in a distributed setting. Distributed coordination services are especially valuable in large-scale or intricate systems, like those encountered in microservices architectures, distributed computing environments, or clustered databases. Apache ZooKeeper, etcd, and Consul are examples of such services.

17. Heartbeat

In a distributed environment, work/data is distributed among servers. To efficiently route requests in such a setup, servers need to know what other servers are part of the system. Furthermore, servers should know if other servers are alive and working. In a decentralized system, whenever a request arrives at a server, the server should have enough information to decide which server is responsible for entertaining that request. This makes the timely detection of server failure an

important task, which also enables the system to take corrective actions and move the data/work to another healthy server and stop the environment from further deterioration.

To solve this, each server periodically sends a heartbeat message to a central monitoring server or other servers in the system to show that it is still alive and functioning.

Heartbeating is one of the mechanisms for detecting failures in a distributed system. If there is a central server, all servers periodically send a heartbeat message to it. If there is no central server, all servers randomly choose a set of servers and send them a heartbeat message every few seconds. This way, if no heartbeat message is received from a server for a while, the system can suspect that the server might have crashed. If there is no heartbeat within a configured timeout period, the system can conclude that the server is not alive anymore and stop sending requests to it and start working on its replacement.

18. Checksum

In a distributed system, while moving data between components, it is possible that the data fetched from a node may arrive corrupted. This corruption can occur because of faults in a storage device, network, software, etc. How can a distributed system ensure data integrity, so that the client receives an error instead of corrupt data?

To solve this, we can calculate a checksum and store it with data.

To calculate a checksum, a cryptographic hash-function like MD5 , SHA-1 , SHA-256 , or SHA-512 is used. The hash function takes the input data and produces a string (containing letters and numbers) of fixed length; this string is called the checksum.

When a system is storing some data, it computes a checksum of the data and stores the checksum with the data. When a client retrieves data, it verifies that the data it received from the server matches the checksum stored. If not, then the client can opt to retrieve that data from another replica.

With this, let's proceed to solve our first system design problem: 'Designing a URL Shortening Service'.

Designing a URL Shortening Service like TinyURL

Let's design a URL shortening service like TinyURL. This service will provide short aliases redirecting to long URLs.

Similar services - bit.ly, ow.ly, short.io

Difficulty Level - Easy

1. Why do we need URL shortening?

URL shortening is used to create shorter aliases for long URLs. We call these shortened aliases “short links.” Users are redirected to the original URL when they hit these short links. Short links save a lot of space when displayed, printed, messaged, or tweeted. Additionally, users are less likely to mistype shorter URLs.

For example, if we shorten the following URL through TinyURL:

<https://www.designgurus.org/course/grokking-the-system-design-interview>

We would get:

<https://tinyurl.com/vzet59pa>

The shortened URL is nearly one-third the size of the actual URL.

URL shortening is used to optimize links across devices, track individual links to analyze audience, measure ad campaigns' performance, or hide affiliated original URLs.

If you haven't used tinyurl.com before, please try creating a new shortened URL and spend some time going through the various options their service offers. This will help you a lot in understanding this chapter.

Designing URL Shortener (video)

Here is a video discussing how to design URL Shortner:

1:39:09

Designing URL Shortener

2. Requirements and Goals of the System

 You should always clarify requirements at the beginning of the interview. Be sure to ask questions to find the exact scope of the system that the interviewer has in mind.

Our URL shortening system should meet the following requirements:

Functional Requirements:

1. Given a URL, our service should generate a shorter and unique alias of it. This is called a short link. This link should be short enough to be easily copied and pasted into applications.
2. When users access a short link, our service should redirect them to the original link.
3. Users should optionally be able to pick a custom short link for their URL.
4. Links will expire after a standard default timespan. Users should be able to specify the expiration time.

Non-Functional Requirements:

1. The system should be highly available. This is required because, if our service is down, all the URL redirections will start failing.
2. URL redirection should happen in real-time with minimal latency.
3. Shortened links should not be guessable (not predictable).

Extended Requirements:

1. Analytics; e.g., how many times a redirection happened?
2. Our service should also be accessible through REST APIs by other services.

3. Capacity Estimation and Constraints

Our system will be read-heavy. There will be lots of redirection requests compared to new URL shortenings. Let's assume a 100:1 ratio between read and write.

Traffic estimates: Assuming, we will have 500 million new URL shortenings per month, with 100:1 read/write ratio, we can expect 50 billion redirections during the same period:

$$100 * 500M \Rightarrow 50B$$

What would be Queries Per Second (QPS) for our system? New URLs shortenings per second:

$$500 \text{ million} / (30 \text{ days} * 24 \text{ hours} * 3600 \text{ seconds}) = \sim 200 \text{ URLs/s}$$

Considering 100:1 read/write ratio, URLs redirections per second will be:

$$100 * 200 \text{ URLs/s} = 20K/s$$

Storage estimates: Let's assume we store every URL shortening request (and associated shortened link) for 5 years. Since we expect to have 500 million new URLs every month, the total number of objects we expect to store will be 30 billion:

$$500 \text{ million} * 5 \text{ years} * 12 \text{ months} = 30 \text{ billion}$$

Let's assume that each stored object will be approximately 500 bytes (just a ballpark estimate--we will dig into it later). We will need 15TB of total storage:

$$30 \text{ billion} * 500 \text{ bytes} = 15 \text{ TB}$$

Bandwidth estimates: For write requests, since we expect 200 new URLs every second, total incoming data for our service will be 100KB per second:

$$200 * 500 \text{ bytes} = 100 \text{ KB/s}$$

For read requests, since every second we expect ~20K URLs redirections, total outgoing data for our service would be 10MB per second:

$$20K * 500 \text{ bytes} = \sim 10 \text{ MB/s}$$

Memory estimates: If we want to cache some of the hot URLs that are frequently accessed, how much memory will we need to store them? If we follow the 80-20 rule, meaning 20% of URLs generate 80% of traffic, we would like to cache these 20% hot URLs.

Since we have 20K requests per second, we will be getting 1.7 billion requests per day:

$$20K * 3600 \text{ seconds} * 24 \text{ hours} = \sim 1.7 \text{ billion}$$

To cache 20% of these requests, we will need 170GB of memory.

$$0.2 * 1.7 \text{ billion} * 500 \text{ bytes} = \sim 170\text{GB}$$

One thing to note here is that since there will be many duplicate requests (of the same URL), our actual memory usage will be less than 170GB.

High-level estimates: Assuming 500 million new URLs per month and 100:1 read:write ratio, following is the summary of the high level estimates for our service:

New URLs	200/s
URL redirections	20K/s
Incoming data	100KB/s
Outgoing data	10MB/s
Storage for 5 years	15TB
Memory for cache	170GB

4. System Interface Definition

💡 Once we've finalized the requirements, it's always a good idea to define the system APIs. This should explicitly state what is expected from the system.

We can have SOAP or REST APIs to expose the functionality of our service.

Following could be the definitions of the APIs for creating and deleting URLs:

Here are the different APIs that could be part of a URL shortening service, along with their parameters and definitions:

1. Create Short URL API

Generates a shortened URL from a long URL, with optional custom alias and expiration date.

- **Endpoint:** POST /shorten
- **Parameters:**
 - **original_url (string, required):** The original long URL that needs to be shortened.
 - **custom_alias (string, optional):** A custom alias for the shortened URL if the user wants to specify one.
 - **expiration_date (timestamp, optional):** The date and time when the shortened URL should expire.
 - **user_id (string, optional):** The ID of the user creating the shortened URL, if user accounts are supported.
- **Response:**
 - **shortened_url (string):** The shortened URL generated by the service.

- **creation_date (timestamp)**: The date and time when the URL was shortened.
- **expiration_date (timestamp, if provided)**: The expiration date of the shortened URL.

2. Redirect API

Redirects users from a shortened URL to the original long URL.

- **Endpoint:** GET /{shortened_url}
- **Parameters:**
 - **shortened_url (string, required)**: The shortened URL that needs to be resolved to the original URL.
- **Response:**
 - Redirects to the `original_url`.

3. Analytics API

Provides detailed analytics for a shortened URL, including click count and user demographics.

- **Endpoint:** GET /analytics/{shortened_url}
- **Parameters:**
 - **shortened_url (string, required)**: The shortened URL for which analytics data is requested.
 - **start_date (timestamp, optional)**: The start date for filtering analytics data.
 - **end_date (timestamp, optional)**: The end date for filtering analytics data.

- **Response:**

- **click_count (integer):** The total number of times the shortened URL has been clicked.
- **unique_clicks (integer):** The number of unique users who clicked the shortened URL.
- **referring_sites (list):** The sites that referred traffic to the shortened URL.
- **location_data (map):** Geographic distribution of users who clicked the URL.
- **device_data (map):** Breakdown of devices (mobile, desktop, etc.) used to click the URL.

4. URL Management API

Retrieves a list of all URLs shortened by a specific user, with metadata.

- **Endpoint:** GET /user/urls

- **Parameters:**

- **user_id (string, required):** The ID of the user whose URLs are being requested.
- **page (integer, optional):** The page number for paginated results.
- **page_size (integer, optional):** The number of results per page.

- **Response:**

- **urls (list):** A list of URLs shortened by the user, including metadata like creation date and expiration date.

5. Delete Short URL API

Deletes a specified shortened URL from the service.

- **Endpoint:** DELETE /{shortened_url}
- **Parameters:**
 - **shortened_url (string, required):** The shortened URL that needs to be deleted.
 - **user_id (string, required):** The ID of the user requesting the deletion.
- **Response:**
 - **status (string):** Confirmation of deletion or error message if the operation fails.

These APIs cover the essential operations for a URL shortening service, allowing users to create, manage, and track shortened URLs while also handling redirection and security features like deletion.

How do we detect and prevent abuse? A malicious user can put our service out of business by consuming all URL keys in the current design. To prevent abuse, we can limit users to a certain number of URL creations and redirections per some time period (which may be set to a different duration for different user types).

5. Database Design

💡 Defining the DB schema in the early stages of the interview would help to understand the data flow among various components and later would guide towards data partitioning.

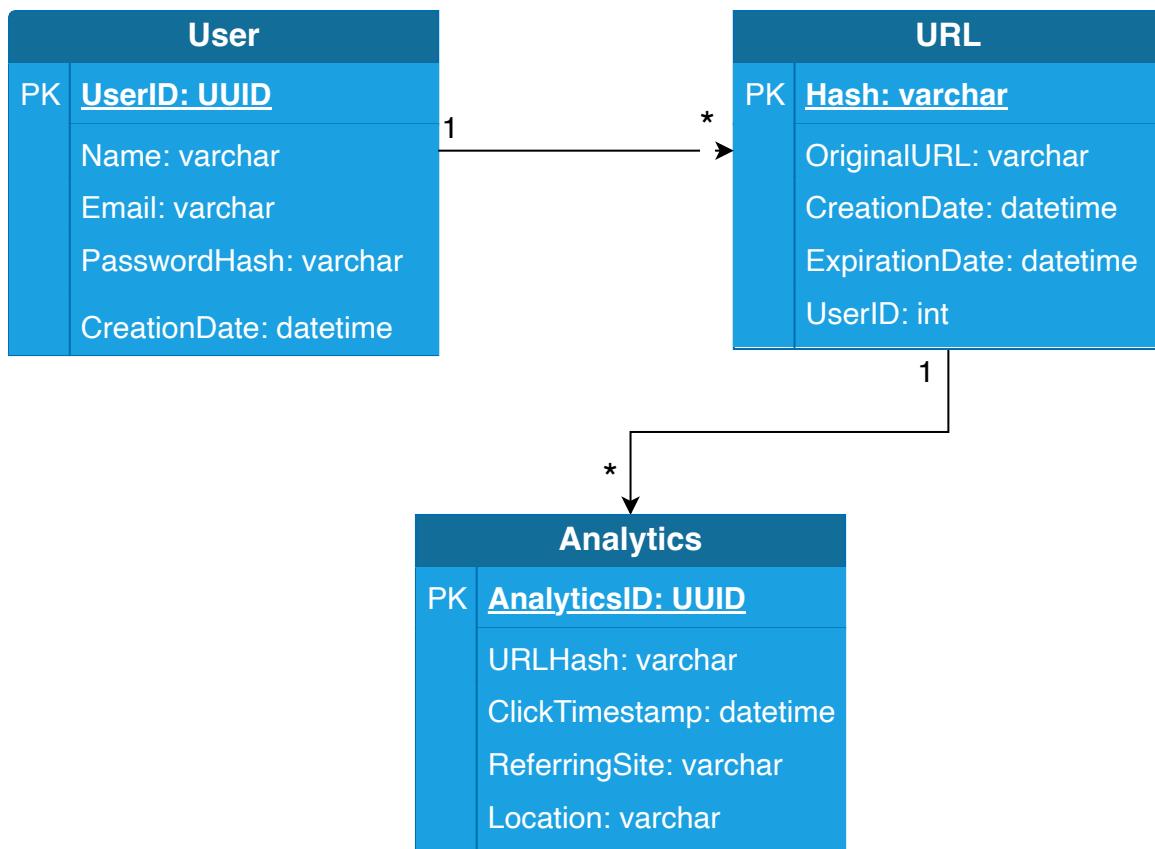
A few observations about the nature of the data we will store:

1. We need to store billions of records.

2. Each object we store is small (less than 1K).
3. There are no relationships between records—other than storing which user created a URL.
4. Our service is read-heavy.

Database Schema:

We would need two tables: one for storing information about the URL mappings and one for the user's data who created the short link.



Database Schema

What kind of database should we use? Since we anticipate storing billions of rows, and we don't need to use relationships between objects – a NoSQL store like [DynamoDB](#), [Cassandra](#) or [Riak](#) is a better choice. A NoSQL choice would also be easier to scale. Please see '[SQL vs NoSQL](#)' for more details.

6. Basic System Design and Algorithm

The problem we are solving here is how to generate a short and unique key for a given URL.

In the TinyURL example in Section 1, the shortened URL is "<https://tinyurl.com/vzet59pa>". The last eight characters of this URL constitute the short key we want to generate. 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. A reasonable question would be, what should be the length of the short key? 6, 8, or 10 characters?

Using base64 encoding, a 6 letters long key would result in $64^6 = \sim 68.7$ billion possible strings.

Using base64 encoding, an 8 letters long key would result in $64^8 = \sim 281$ trillion possible strings.

With 68.7B unique strings, let's assume six letter keys would suffice for our system.

If we use the MD5 algorithm as our hash function, it will produce a 128-bit hash value. After base64 encoding, we'll get a string having more than 21 characters (since each base64 character encodes 6 bits of the hash value). Now we only have space for 6 (or 8) characters per short key; how will we choose our key then? We can take the first 6 (or 8) letters for the key. This could result in key duplication; to resolve that, we can choose some other characters out of the encoding string or swap some characters.

What are the different issues with our solution? We have the following couple of problems with our encoding scheme:

1. If multiple users enter the same URL, they can get the same shortened URL, which is not acceptable.

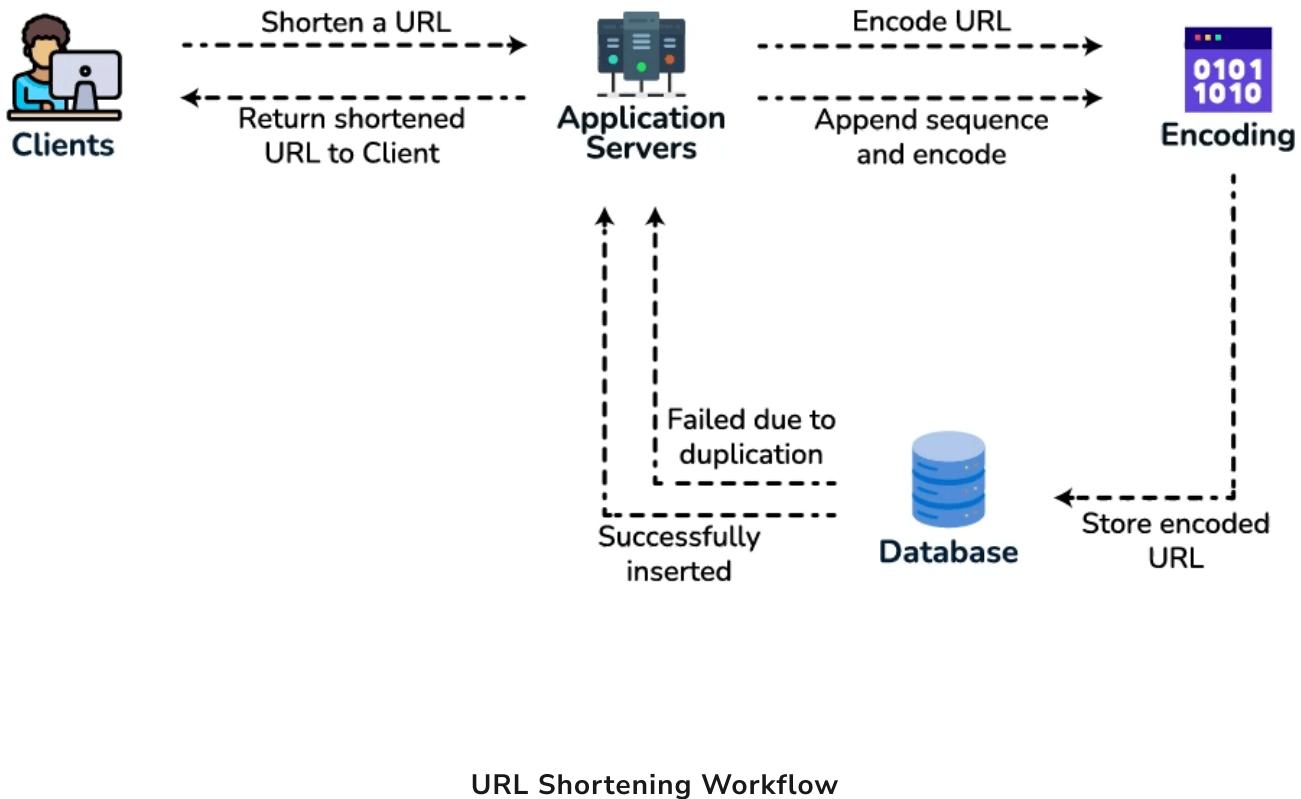
2. What if parts of the URL are URL-encoded? e.g.,

<http://www.desingngurus.org/distributed.php?id=design>, and

<http://www.desingngurus.org/distributed.php%3Fid%3Ddesign> are identical except for the URL encoding.

Workaround for the issues: We can append an increasing sequence number to each input URL to make it unique and then generate its hash. We don't need to store this sequence number in the databases, though. Possible problems with this approach could be an ever-increasing sequence number. Can it overflow? Appending an increasing sequence number will also impact the performance of the service.

Another solution could be to append the user id (which should be unique) to the input URL. However, if the user has not signed in, we would have to ask the user to choose a uniqueness key. Even after this, if we have a conflict, we have to keep generating a key until we get a unique one.



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. This approach will make things quite simple and fast. Not only are we not encoding the URL, but we won't have to worry about duplications or collisions. KGS will make sure all the keys inserted into key-DB are unique.

Can concurrency cause problems? As soon as a key is used, it should be marked in the database to ensure that it is not used again. If there are multiple servers reading keys concurrently, we might get a scenario where two or more servers try

to read the same key from the database. How can we solve this concurrency problem?

Servers can use KGS to read/mark keys in the database. KGS can use two tables to store keys: one for keys that are not used yet, and one for all the used keys. As soon as KGS gives keys to one of the servers, it can move them to the used keys table. KGS can always keep some keys in memory to quickly provide them whenever a server needs them.

For simplicity, as soon as KGS loads some keys in memory, it can move them to the used keys table. This ensures each server gets unique keys. If KGS dies before assigning all the loaded keys to some server, we will be wasting those keys--which could be acceptable, given the huge number of keys we have.

KGS also has to make sure not to give the same key to multiple servers. For that, it must synchronize (or get a lock on) the data structure holding the keys before removing keys from it and giving them to a server.

What would be the key-DB size? With base64 encoding, we can generate 68.7B unique six letters keys. If we need one byte to store one alpha-numeric character, we can store all these keys in:

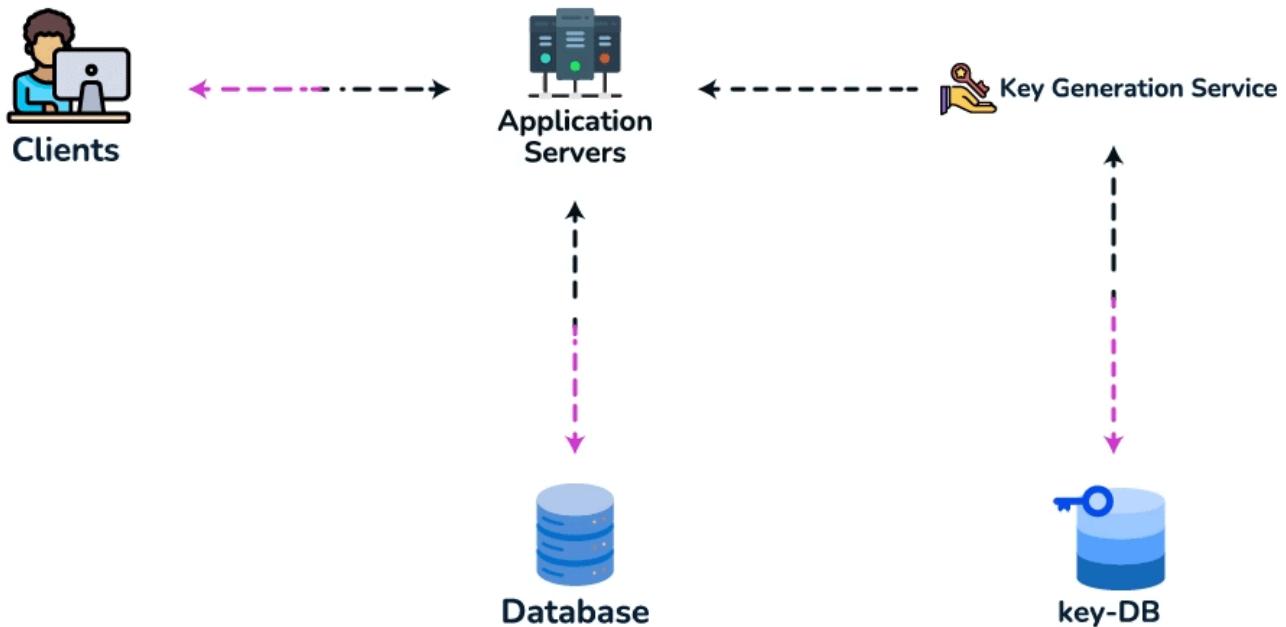
$$6 \text{ (characters per key)} * 68.7\text{B} \text{ (unique keys)} = 412 \text{ GB.}$$

Isn't KGS a single point of failure? Yes, it is. To solve this, we can have a standby replica of KGS. Whenever the primary server dies, the standby server can take over to generate and provide keys.

Can each app server cache some keys from key-DB? Yes, this can surely speed things up. Although, in this case, if the application server dies before consuming all the keys, we will end up losing those keys. This can be acceptable since we have 68B unique six-letter keys.

How would we perform a key lookup? We can look up the key in our database to get the full URL. If it's present in the DB, issue an "HTTP 302 Redirect" status back to the browser, passing the stored URL in the "Location" field of the request. If that key is not present in our system, issue an "HTTP 404 Not Found" status or redirect the user back to the homepage.

Should we impose size limits on custom aliases? Our service supports custom aliases. Users can pick any 'key' they like, but providing a custom alias is not mandatory. However, it is reasonable (and often desirable) to impose a size limit on a custom alias to ensure we have a consistent URL database. Let's assume users can specify a maximum of 16 characters per customer key (as reflected in the above database schema).



High Level Design - URL Shortening

7. Data Partitioning and Replication

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. This approach is called range-based partitioning. We can even combine certain less frequently occurring letters into one database partition. Thus, we should develop a static partitioning scheme to always store/find a URL in a predictable manner.

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 hash. In our case, we can take the hash of the 'key' or the short link to determine the partition in which we store the data object.

Our hashing function will randomly distribute URLs into different partitions (e.g., our hashing function can always map any 'key' to a number between [1...256]). This number would represent the partition in which we store our object.

This approach can still lead to overloaded partitions, which can be solved using 'Consistent Hashing'.

8. Cache

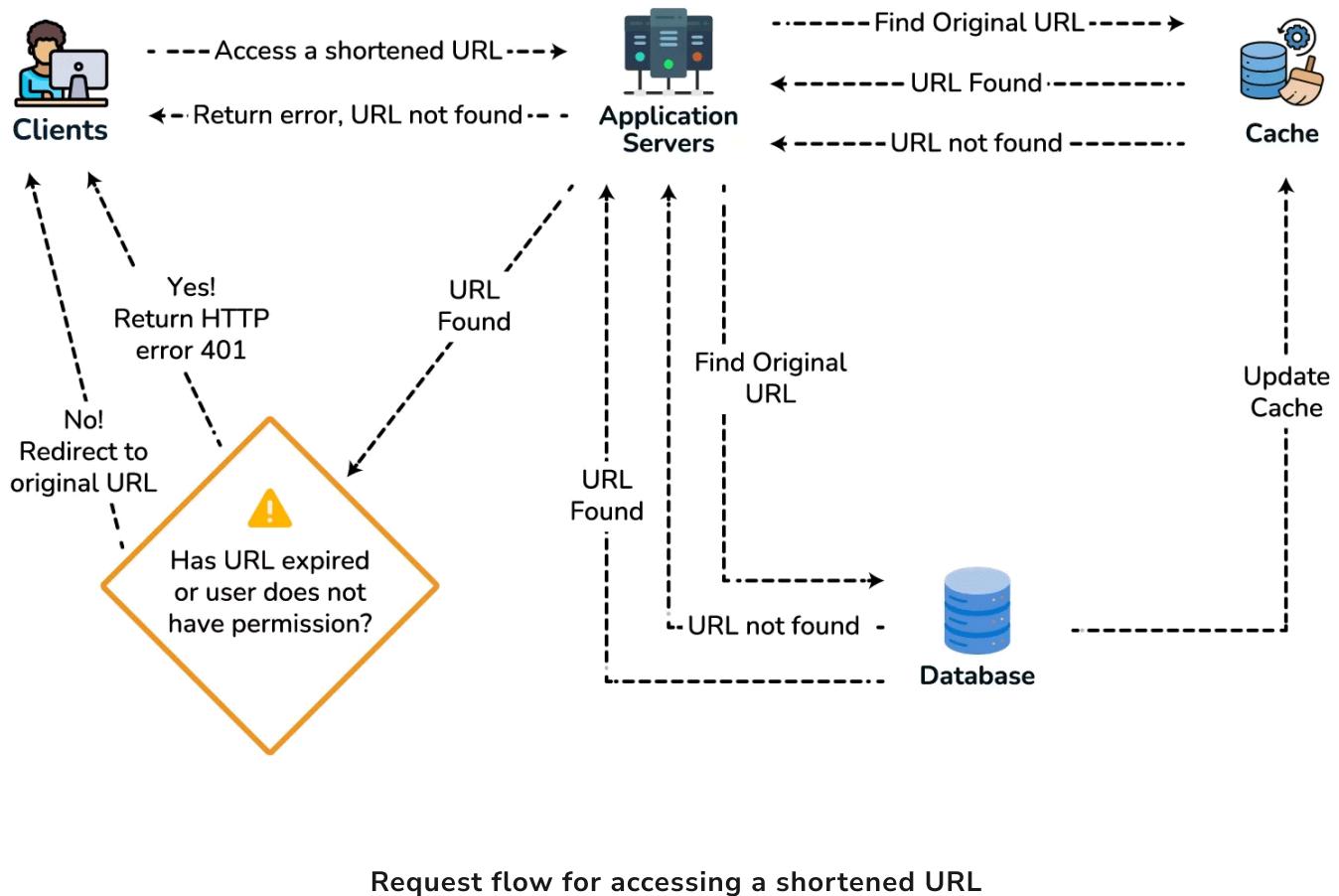
We can cache URLs that are frequently accessed. We can use any off-the-shelf solution like [Memcached](#), which can store full URLs with their respective hashes. Thus, the application servers, before hitting the backend storage, can quickly check if the cache has the desired URL.

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. Since a modern-day server can have 256GB of memory, we can easily fit all the cache into one machine. Alternatively, we can use a couple of smaller servers to store all these hot URLs.

Which cache eviction policy would best fit our needs? When the cache is full, and we want to replace a link with a newer/hotter URL, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system. Under this policy, we discard the least recently used URL first. We can use a [Linked Hash Map](#) or a similar data structure to store our URLs and Hashes, which will also keep track of the URLs that have been accessed recently.

To further increase the efficiency, we can replicate our caching servers to distribute the load between them.

How can each cache replica be updated? Whenever there is a cache miss, our servers would be hitting a backend database. Whenever this happens, we can update the cache and pass the new entry to all the cache replicas. Each replica can update its cache by adding the new entry. If a replica already has that entry, it can simply ignore it.



9. 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.

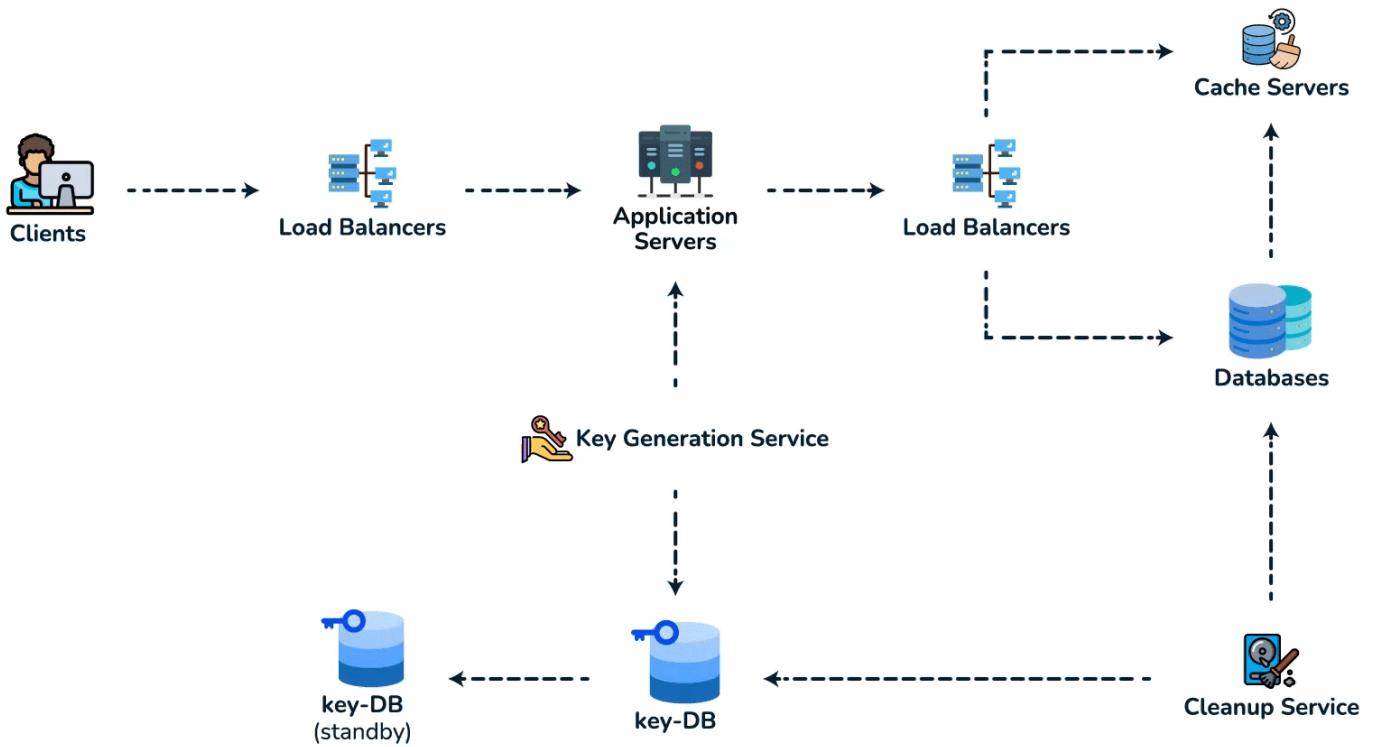
A problem with Round Robin LB is that we do not consider the server load. As a result, if a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about its load and adjusts traffic based on that.

10. 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).
- After removing an expired link, we can put the key back in the key-DB to be reused.
- Should we remove links that haven't been visited in some length of time, say six months? This could be tricky. Since storage is getting cheap, we can decide to keep links forever.



Detailed component design for URL shortening

11. Telemetry

How many times a short URL has been used, what were user locations, etc.? How would we store these statistics? If it is part of a DB row that gets updated on each view, what will happen when a popular URL is slammed with a large number of concurrent requests?

Some statistics worth tracking: country of the visitor, date and time of access, web page that referred the click, browser, or platform from where the page was accessed.

12. Security and Permissions

Can users create private URLs or allow a particular set of users to access a URL?

We can store the permission level (public/private) with each URL in the database. We can also create a separate table to store UserIDs that have permission to see a specific URL. If a user does not have permission and tries to access a URL, we can send an error (HTTP 401) back. Given that we are storing our data in a NoSQL wide-column database like Cassandra, the key for the table storing permissions would be the 'Hash' (or the KGS generated 'key'). The columns will store the UserIDs of those users that have permission to see the URL.

Designing Pastebin

Let's design a Pastebin like web service, where users can store plain text. Users of the service will enter a piece of text and get a randomly generated URL to access it.

Similar Services: pastebin.com, controlc.com, hastebin.com, privatebin.net

Difficulty Level: Easy

1. What is Pastebin?

Pastebin like services enable users to store plain text or images over the network (typically the Internet) and generate unique URLs to access the uploaded data.

Such services are also used to share data over the network quickly, as users would just need to pass the URL to let other users see it.

If you haven't used pastebin.com before, please try creating a new 'Paste' there and spend some time going through the different options their service offers. This will help you a lot in understanding this chapter.

2. Requirements and Goals of the System

Our Pastebin service should meet the following requirements:

Functional Requirements:

1. Users should be able to upload or “paste” their data and get a unique URL to access it.
2. Users will only be able to upload text.
3. Data and links will expire after a specific timespan automatically; users should also be able to specify expiration time.

4. Users should optionally be able to pick a custom alias for their paste.

Non-Functional Requirements:

1. The system should be highly reliable, any data uploaded should not be lost.
2. The system should be highly available. This is required because if our service is down, users will not be able to access their Pastes.
3. Users should be able to access their Pastes in real-time with minimum latency.
4. Paste links should not be guessable (not predictable).

Extended Requirements:

1. Analytics, e.g., how many times a paste was accessed?
2. Our service should also be accessible through REST APIs by other services.

3. Some Design Considerations

Pastebin shares some requirements with 'URL Shortening service', but there are some additional design considerations we should keep in mind.

What should be the limit on the amount of text user can paste at a time? We can limit users not to have Pastes bigger than 10MB to stop the abuse of the service.

Should we impose size limits on custom URLs? Since our service supports custom URLs, users can pick any URL that they like, but providing a custom URL is not mandatory. However, it is reasonable (and often desirable) to impose a size limit on custom URLs, so that we have a consistent URL database.

4. Capacity Estimation and Constraints

Our services will be read-heavy; there will be more read requests compared to new Paste creation. We can assume a 5:1 ratio between the read and write.

Traffic estimates: Pastebin services are not expected to have traffic similar to Twitter or Facebook, let's assume here that we get one million new pastes added to our system every day. This leaves us with five million reads per day.

New Pastes per second:

$$1M / (24 \text{ hours} * 3600 \text{ seconds}) \approx 12 \text{ pastes/sec}$$

Paste reads per second:

$$5M / (24 \text{ hours} * 3600 \text{ seconds}) \approx 58 \text{ reads/sec}$$

Storage estimates: Users can upload maximum 10MB of data; commonly Pastebin like services are used to share source code, configs, or logs. Such texts are not huge, so let's assume that each paste on average contains 10KB.

At this rate, we will be storing 10GB of data per day.

$$1M * 10KB \Rightarrow 10 \text{ GB/day}$$

If we want to store this data for ten years we would need a total storage capacity of 36TB.

With 1M pastes every day we will have 3.6 billion Pastes in 10 years. We need to generate and store keys to uniquely identify these pastes. If we use base64 encoding ([A-Z, a-z, 0-9, ., -]) we would need six letters strings:

$$64^6 \approx 68.7 \text{ billion unique strings}$$

If it takes one byte to store one character, total size required to store 3.6B keys would be:

$$3.6B * 6 \Rightarrow 22 \text{ GB}$$

22GB is negligible compared to 36TB. To keep some margin, we will assume a 70% capacity model (meaning we don't want to use more than 70% of our total storage

capacity at any point), which raises our storage needs to 51.4TB.

Bandwidth estimates: For write requests, we expect 12 new pastes per second, resulting in 120KB of ingress per second.

$$12 * 10\text{KB} \Rightarrow 120 \text{ KB/s}$$

As for the read request, we expect 58 requests per second. Therefore, total data egress (sent to users) will be 0.6 MB/s.

$$58 * 10\text{KB} \Rightarrow 0.6 \text{ MB/s}$$

Although total ingress and egress are not big, we should keep these numbers in mind while designing our service.

Memory estimates: We can cache some of the hot pastes that are frequently accessed. Following the 80-20 rule, meaning 20% of hot pastes generate 80% of traffic, we would like to cache these 20% pastes.

Since we have 5M read requests per day, to cache 20% of these requests, we would need:

$$0.2 * 5\text{M} * 10\text{KB} \approx 10 \text{ GB}$$

5. System APIs

We can have SOAP or REST APIs to expose the functionality of our service.

Following could be the definitions of the APIs to create/retrieve/delete Pastes:

```
addPaste(api_dev_key, paste_data, custom_url=None
         user_name=None, paste_name=None, expire_date=None)
```

Parameters:

api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.

paste_data (string): Textual data of the paste.

custom_url (string): Optional custom URL.

user_name (string): Optional user name to be used to generate URL.

paste_name (string): Optional name of the paste

expire_date (string): Optional expiration date for the paste.

Returns: (string)

A successful insertion returns the URL through which the paste can be accessed, otherwise, it will return an error code.

Similarly, we can have Retrieve and Delete Paste APIs:

`getPaste(api_dev_key, api_paste_key)`

Where "api_paste_key" is a string representing the Paste Key of the paste to be retrieved. This API will return the textual data of the paste.

`deletePaste(api_dev_key, api_paste_key)`

A successful deletion returns 'true', otherwise returns 'false'.

6. Database Design

A few observations about the nature of the data we are storing:

1. We need to store billions of records.

2. Each metadata object we are storing would be small (less than 1KB).
3. Each paste object we are storing can be of medium size (it can be a few MB).
4. There are no relationships between records, except if we want to store which user created what Paste.
5. Our service is read-heavy.

Database Schema:

We would need two tables, one for storing information about the Pastes and the other for users' data.

Paste		User	
PK	<u>URLHash: varchar(16)</u>	PK	<u>UserID: int</u>
	ContentKey: varchar(512) ExpirationDate: datetime UserID: int CreationDate: datetime		Name: varchar(20) Email: varchar(32) CreationDate: datetime LastLogin: datetime

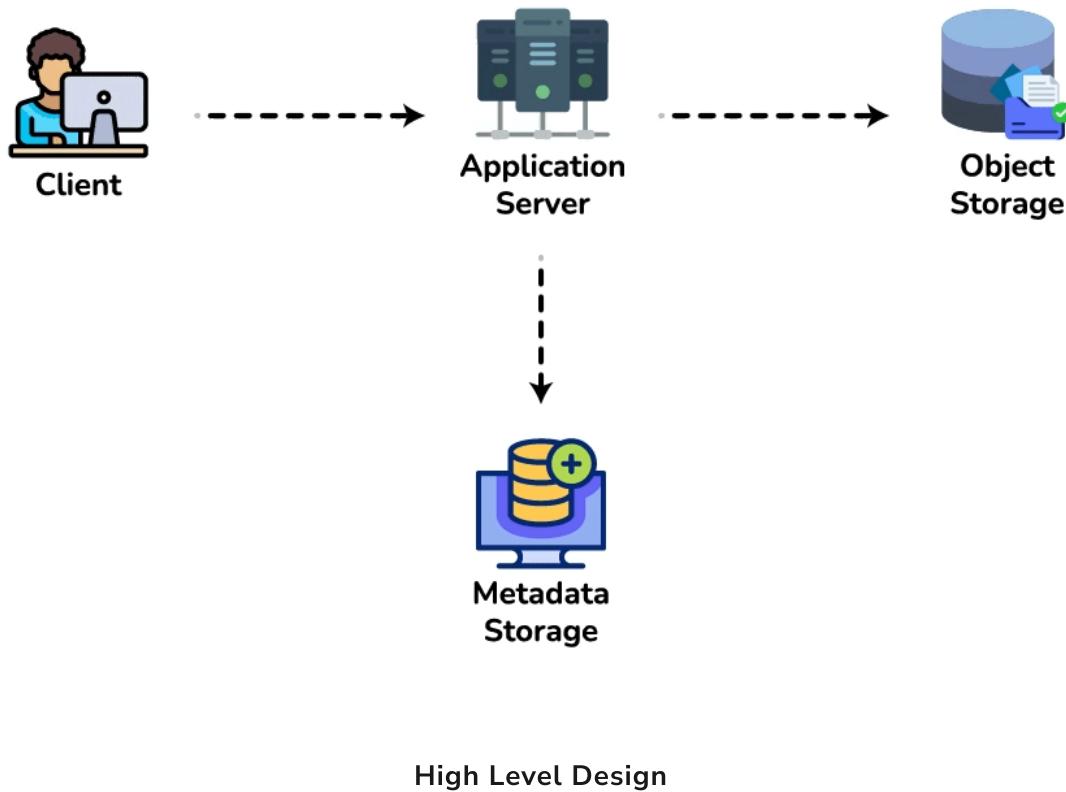
DB Schema

Here, 'URLHash' is the URL equivalent of the TinyURL, and 'ContentKey' is a reference to an external object storing the contents of the paste; we'll discuss the external storage of the paste contents later in the chapter.

7. High Level Design

At a high level, we need an application layer that will serve all the read and write requests. Application layer will talk to a storage layer to store and retrieve data. We can segregate our storage layer with one database storing metadata related to each

paste, users, etc., while the other storing the paste contents in some object storage (like [Amazon S3](#)). This division of data will also allow us to scale them individually.



High Level Design

8. Component Design

a. Application layer

Our application layer will process all incoming and outgoing requests. The application servers will be talking to the backend data store components to serve the requests.

How to handle a write request? Upon receiving a write-request, our application server will generate a six-letter random string, which would serve as the key of the paste (if the user has not provided a custom key). The application server will then

store the contents of the paste and the generated key in the database. After the successful insertion, the server can return the key to the user. One possible problem here could be that the insertion fails because of a duplicate key. Since we are generating a random key, there is a possibility that the newly generated key could match an existing one. In that case, we should regenerate a new key and try again. We should keep retrying until we don't see failure due to the duplicate key. We should return an error to the user if the custom key they have provided is already present in our database.

Another solution for the above problem could be to run a standalone **Key Generation Service** (KGS) that generates random six letters strings beforehand and stores them in a database (let's call it key-DB). Whenever we want to store a new paste, we will just take one of the already generated keys and use it. This approach will make things quite simple and fast since we will not be worrying about duplications or collisions. KGS will make sure all the keys inserted in key-DB are unique. KGS can use two tables to store keys, one for keys that are not used yet and one for all the used keys. As soon as KGS gives some keys to an application server, it can move these to the used keys table. KGS can always keep some keys in memory so that whenever a server needs them, it can quickly provide them. As soon as KGS loads some keys in memory, it can move them to the used keys table; this way we can make sure each server gets unique keys. If KGS dies before using all the keys loaded in memory, we will be wasting those keys. We can ignore these keys given that we have a huge number of them.

Isn't KGS a single point of failure? Yes, it is. To solve this, we can have a standby replica of KGS and whenever the primary server dies it can take over to generate and provide keys.

Can each app server cache some keys from key-DB? Yes, this can surely speed things up. Although in this case, if the application server dies before consuming all the

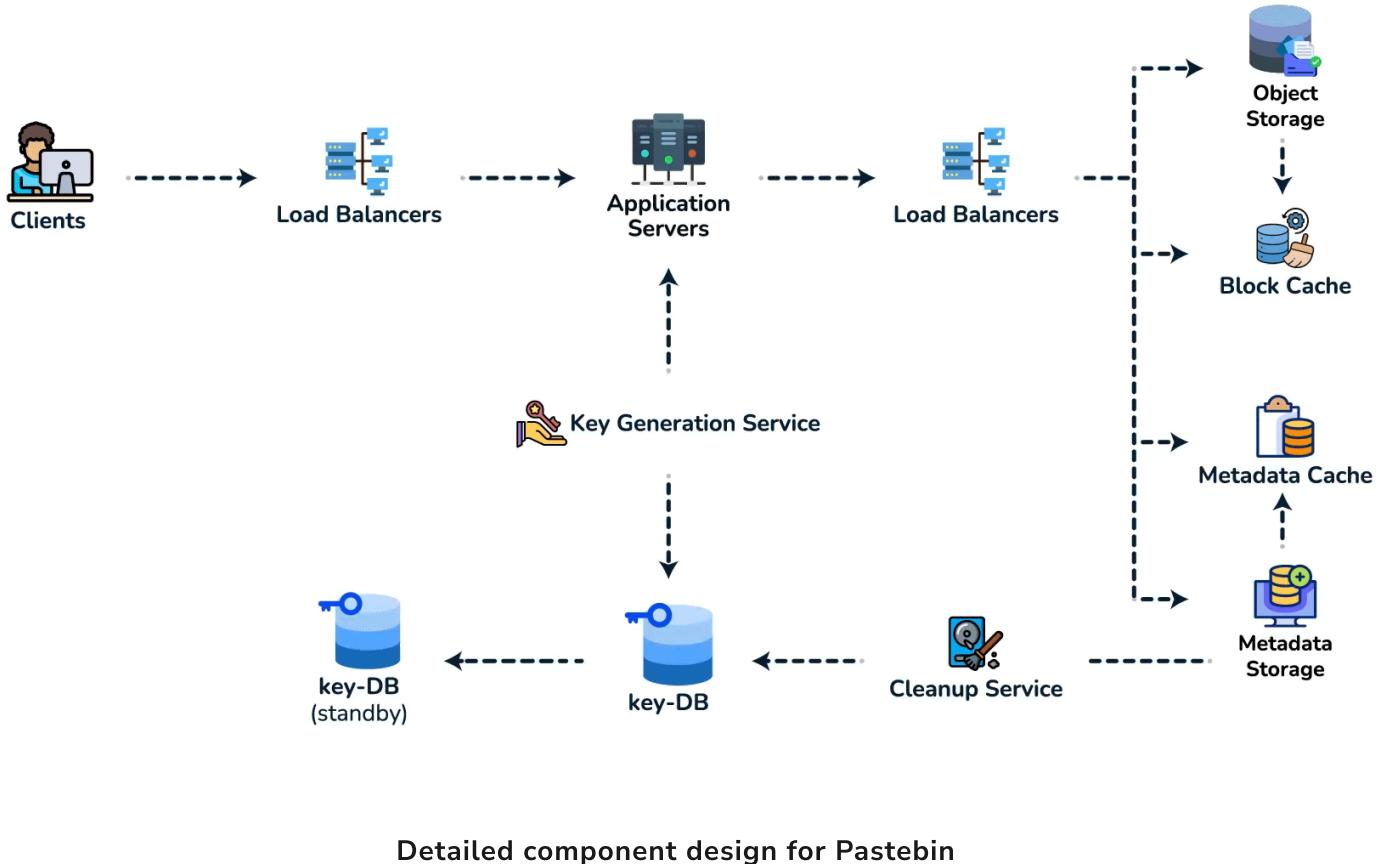
keys, we will end up losing those keys. This could be acceptable since we have 68B unique six letters keys, which are a lot more than we require.

How does it handle a paste read request? Upon receiving a read paste request, the application service layer contacts the datastore. The datastore searches for the key, and if it is found, it returns the paste's contents. Otherwise, an error code is returned.

b. Datastore layer

We can divide our datastore layer into two:

1. Metadata database: We can use a relational database like MySQL or a Distributed Key-Value store like Dynamo or Cassandra.
2. Object storage: We can store our contents in an Object Storage like Amazon's S3. Whenever we feel like hitting our full capacity on content storage, we can easily increase it by adding more servers.



9. Purging or DB Cleanup

Please see '[Designing a URL Shortening service](#)'.

10. Data Partitioning and Replication

Please see '[Designing a URL Shortening service](#)'.

11. Cache and Load Balancer

Please see '[Designing a URL Shortening service](#)'.

12. Security and Permissions

Please see '[Designing a URL Shortening service](#)'.

Designing Instagram

Let's design a photo-sharing service like Instagram, where users can upload photos to share them with other users.

Similar Services: Flickr, Picasa

Difficulty Level: Medium

1. What is Instagram?

Instagram is a social networking service that enables its users to upload and share their photos and videos with other users. Instagram users can choose to share information either publicly or privately. Anything shared publicly can be seen by any other user, whereas privately shared content can only be accessed by the specified set of people. Instagram also enables its users to share through many other social networking platforms, such as Facebook, Twitter, Flickr, and Tumblr.

We plan to design a simpler version of Instagram for this design problem, where a user can share photos and follow other users. The 'News Feed' for each user will consist of top photos of all the people the user follows.

Designing Instagram (video)

Here is a video discussing how to design Instagram:

1:41:50

Designing Instagram

2. Requirements and Goals of the System

We'll focus on the following set of requirements while designing Instagram:

Functional Requirements

1. Users should be able to upload/download/view photos.
2. Users can perform searches based on photo/video titles.
3. Users can follow other users.
4. The system should generate and display a user's News Feed consisting of top photos from all the people the user follows.

Non-functional Requirements

1. Our service needs to be highly available.
2. The acceptable latency of the system is 200ms for News Feed generation.
3. Consistency can take a hit (in the interest of availability) if a user doesn't see a photo for a while; it should be fine.
4. The system should be highly reliable; any uploaded photo or video should never be lost.

Not in scope: Adding tags to photos, searching photos on tags, commenting on photos, tagging users to photos, who to follow, etc.

3. Some Design Considerations

The system would be read-heavy, so we will focus on building a system that can retrieve photos quickly.

1. Practically, users can upload as many photos as they like; therefore, efficient management of storage should be a crucial factor in designing this system.
2. Low latency is expected while viewing photos.
3. Data should be 100% reliable. If a user uploads a photo, the system will guarantee that it will never be lost.

4. Capacity Estimation and Constraints

- Let's assume we have 500M total users, with 1M daily active users.
- 2M new photos every day, 23 new photos every second.
- Average photo file size => 200KB
- Total space required for 1 day of photos

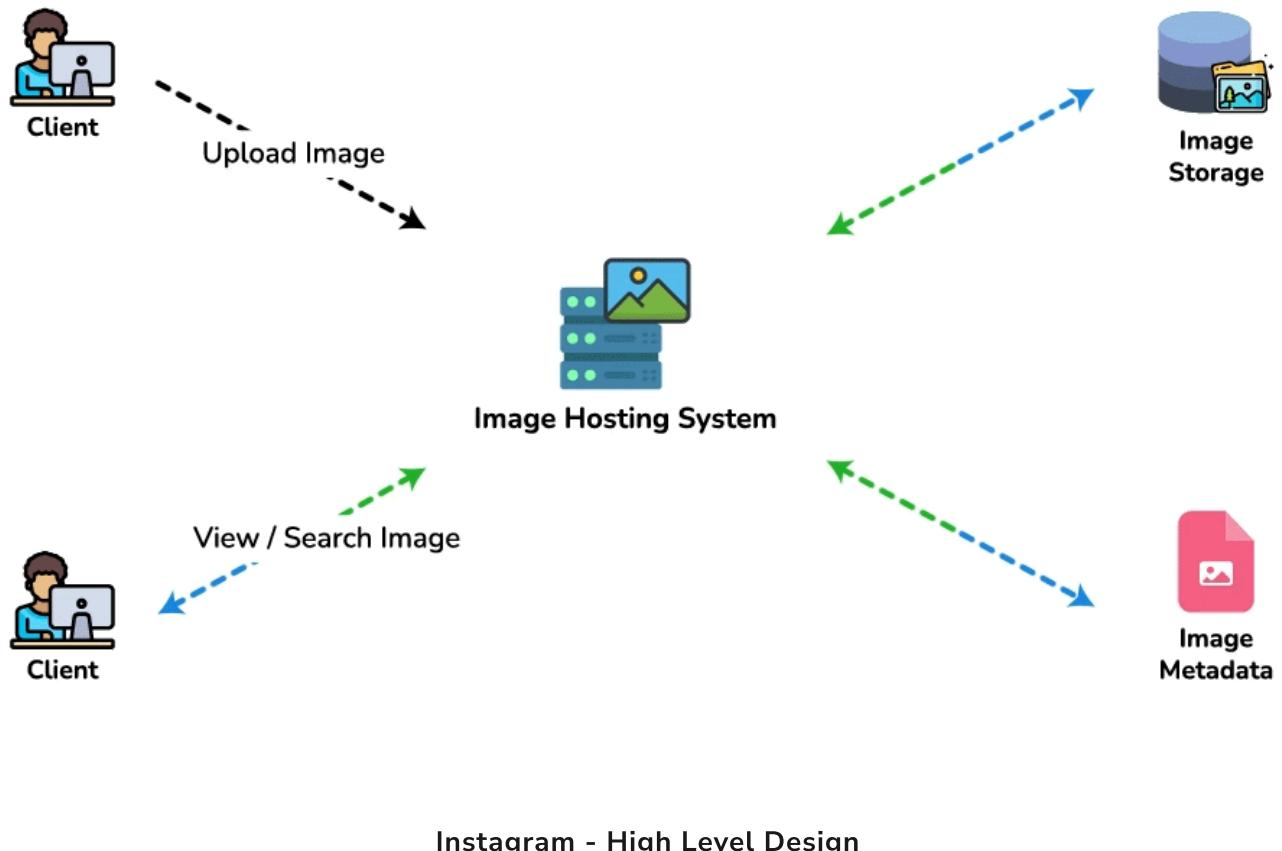
$$2M * 200KB \Rightarrow 400 \text{ GB}$$

- Total space required for 10 years:

$$400\text{GB} * 365 \text{ (days a year)} * 10 \text{ (years)} \approx 1425\text{TB}$$

5. High Level System Design

At a high-level, we need to support two scenarios, one to upload photos and the other to view/search photos. Our service would need some **object storage** servers to store photos and some database servers to store metadata information about the photos.



6. Database Schema

 Defining the DB schema in the early stages of the interview would help to understand the data flow among various components and later would guide towards data partitioning.

We need to store data about users, their uploaded photos, and the people they follow. The Photo table will store all data related to a photo; we need to have an index on (PhotoID, CreationDate) since we need to fetch recent photos first.

Photo		User		UserFollow	
PK	PhotoID: int	PK	UserID: int	PK	FollowerID: int FolloweeID: int
	UserID: int PhotoPath: varchar PhotoLatitude: int PhotoLongitude: int UserLatitude: int UserLongitude: int CreationDate: datetime		Name: varchar Email: varchar DateOfBirth: datetime CreationDate: datetime LastLogin: datetime		

DB Schema

A straightforward approach for storing the above schema would be to use an RDBMS like MySQL since we require joins. But relational databases come with their challenges, especially when we need to scale them. For details, please take a look at [SQL vs. NoSQL](#) chapter.

We can store photos in a distributed file storage like [HDFS](#) or [S3](#).

We can store the above schema in a distributed key-value store to enjoy the benefits offered by NoSQL. All the metadata related to photos can go to a table where the 'key' would be the 'PhotoID' and the 'value' would be an object containing PhotoLocation, UserLocation, CreationTimestamp, etc.

NoSQL stores, in general, always maintain a certain number of replicas to offer reliability. Also, in such data stores, deletes don't get applied instantly; data is retained for certain days (to support undeleting) before getting removed from the system permanently.

7. Data Size Estimation

Let's estimate how much data will be going into each table and how much total storage we will need for 10 years.

User: Assuming each "int" and "dateTime" is four bytes, each row in the User's table will be of 68 bytes:

$$\text{UserID (4 bytes)} + \text{Name (20 bytes)} + \text{Email (32 bytes)} + \text{DateOfBirth (4 bytes)} + \text{CreationDate (4 bytes)} + \text{LastLogin (4 bytes)} = 68 \text{ bytes}$$

If we have 500 million users, we will need 32GB of total storage.

$$500 \text{ million} * 68 \approx 32 \text{ GB}$$

Photo: Each row in Photo's table will be of 284 bytes:

$$\text{PhotoID (4 bytes)} + \text{UserID (4 bytes)} + \text{PhotoPath (256 bytes)} + \text{PhotoLatitude (4 bytes)} + \text{PhotoLongitude (4 bytes)} + \text{UserLatitude (4 bytes)} + \text{UserLongitude (4 bytes)} + \text{CreationDate (4 bytes)} = 284 \text{ bytes}$$

If 2M new photos get uploaded every day, we will need 0.5GB of storage for one day:

2M * 284 bytes \approx 0.5GB per day

For 10 years we will need 1.88TB of storage.

UserFollow: Each row in the UserFollow table will consist of 8 bytes. If we have 500 million users and on average each user follows 500 users. We would need 1.82TB of storage for the UserFollow table:

$$500 \text{ million users} * 500 \text{ followers} * 8 \text{ bytes} \approx 1.82\text{TB}$$

Total space required for all tables for 10 years will be 3.7TB:

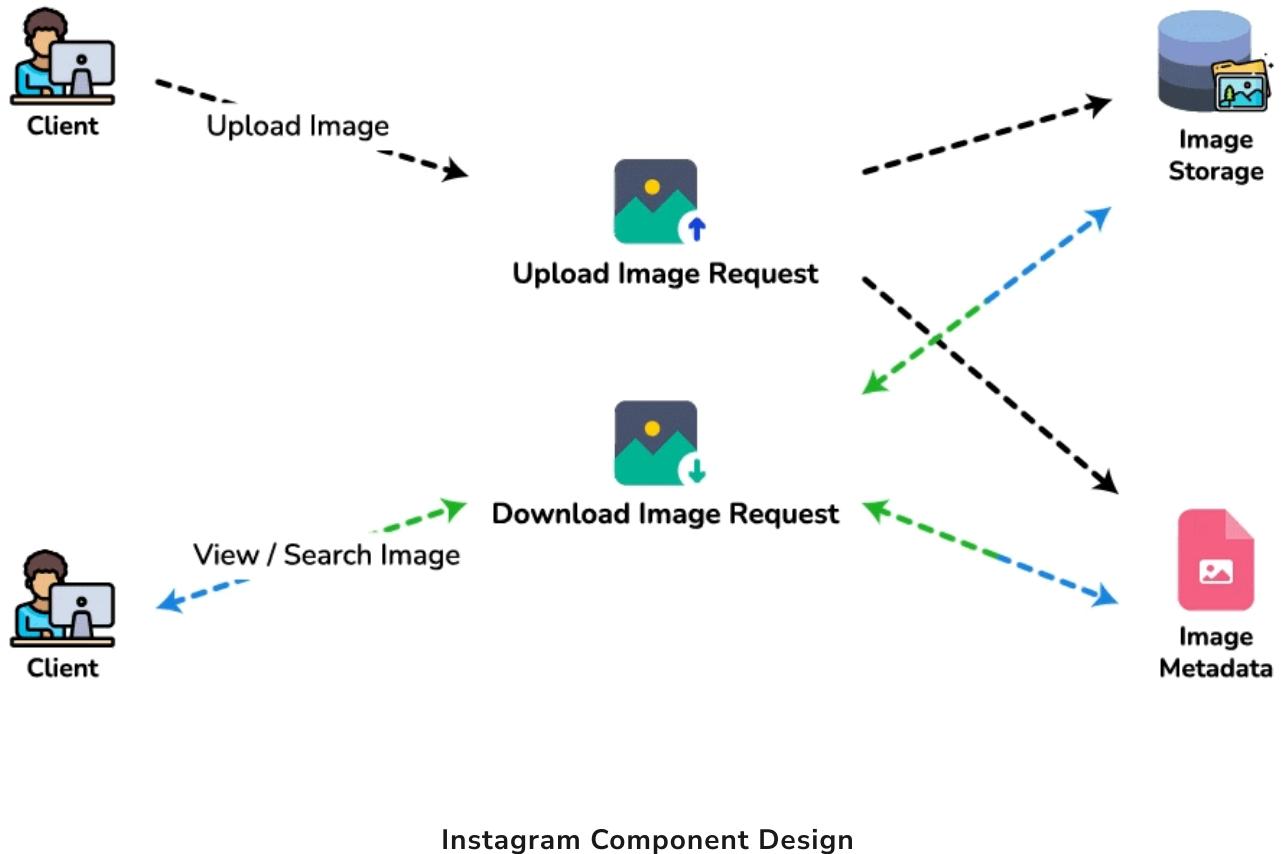
$$32\text{GB} + 1.88\text{TB} + 1.82\text{TB} \approx 3.7\text{TB}$$

8. Component Design

Photo uploads (or writes) can be slow as they have to go to the disk, whereas reads will be faster, especially if they are being served from cache.

Uploading users can consume all the available connections, as uploading is a slow process. This means that 'reads' cannot be served if the system gets busy with all the 'write' requests. We should keep in mind that web servers have a connection limit before designing our system. If we assume that a web server can have a maximum of 500 connections at any time, then it can't have more than 500 concurrent uploads or reads. To handle this bottleneck, we can split reads and writes into separate services. We will have dedicated servers for reads and different servers for writes to ensure that uploads don't hog the system.

Separating photos' read and write requests will also allow us to scale and optimize each of these operations independently.



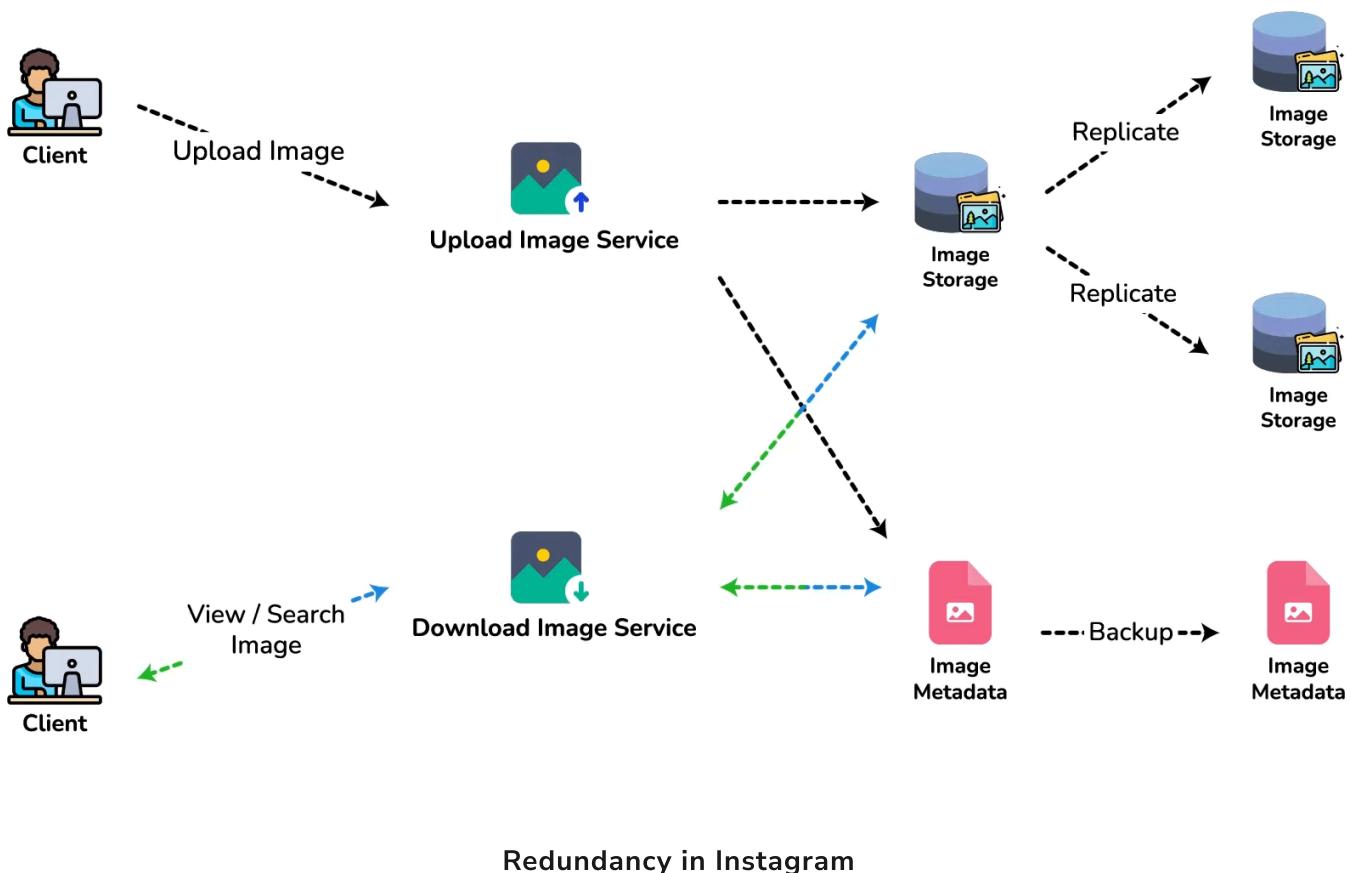
9. Reliability and Redundancy

Losing files is not an option for our service. Therefore, we will store multiple copies of each file so that if one storage server dies, we can retrieve the photo from the other copy present on a different storage server.

This same principle also applies to other components of the system. If we want to have high availability of the system, we need to have multiple replicas of services running in the system so that even if a few services die down, the system remains available and running. Redundancy removes the single point of failure in the system.

If only one instance of a service is required to run at any point, we can run a redundant secondary copy of the service that is not serving any traffic, but it can take control after the failover when the primary has a problem.

Creating redundancy in a system can remove single points of failure and provide a backup or spare functionality if needed in a crisis. For example, if there are two instances of the same service running in production and one fails or degrades, the system can failover to the healthy copy. Failover can happen automatically or require manual intervention.



10. Data Sharding

Let's discuss different schemes for metadata sharding:

a. Partitioning based on UserID Let's assume we shard based on the 'UserID' so that we can keep all photos of a user on the same shard. If one DB shard is 1TB, we will need four shards to store 3.7TB of data. Let's assume, for better performance and scalability, we keep 10 shards.

So we'll find the shard number by **UserID % 10** and then store the data there. To uniquely identify any photo in our system, we can append the shard number with each PhotoID.

How can we generate PhotoIDs? Each DB shard can have its own auto-increment sequence for PhotoIDs, and since we will append ShardID with each PhotoID, it will make it unique throughout our system.

What are the different issues with this partitioning scheme?

1. How would we handle hot users? Several people follow such hot users, and a lot of other people see any photo they upload.
2. Some users will have a lot of photos compared to others, thus making a non-uniform distribution of storage.
3. What if we cannot store all pictures of a user on one shard? If we distribute photos of a user onto multiple shards, will it cause higher latencies?
4. Storing all photos of a user on one shard can cause issues like unavailability of all of the user's data if that shard is down or higher latency if it is serving high load etc.

b. Partitioning based on PhotoID If we can generate unique PhotoIDs first and then find a shard number through "PhotoID % 10", the above problems will have been solved. We would not need to append ShardID with PhotoID in this case, as PhotoID will itself be unique throughout the system.

How can we generate PhotoIDs? Here, we cannot have an auto-incrementing sequence in each shard to define PhotoID because we need to know PhotoID first to find the shard where it will be stored. One solution could be that we dedicate a

separate database instance to generate auto-incrementing IDs. If our PhotoID can fit into 64 bits, we can define a table containing only a 64 bit ID field. So whenever we would like to add a photo in our system, we can insert a new row in this table and take that ID to be our PhotoID of the new photo.

Wouldn't this key generating DB be a single point of failure? Yes, it would be. A workaround for that could be to define two such databases, one generating even-numbered IDs and the other odd-numbered. For MySQL, the following script can define such sequences:

KeyGeneratingServer1:

```
auto-increment-increment = 2  
auto-increment-offset = 1
```

KeyGeneratingServer2:

```
auto-increment-increment = 2  
auto-increment-offset = 2
```

We can put a load balancer in front of both of these databases to round-robin between them and to deal with downtime. Both these servers could be out of sync, with one generating more keys than the other, but this will not cause any issue in our system. We can extend this design by defining separate ID tables for Users, Photo-Comments, or other objects present in our system.

Alternately, we can implement a 'key' generation scheme similar to what we have discussed in 'Designing a URL Shortening service like TinyURL'.

How can we plan for the future growth of our system? We can have a large number of logical partitions to accommodate future data growth, such that in the beginning, multiple logical partitions reside on a single physical database server. Since each database server can have multiple database instances running on it, we can have

separate databases for each logical partition on any server. So whenever we feel that a particular database server has a lot of data, we can migrate some logical partitions from it to another server. We can maintain a config file (or a separate database) that can map our logical partitions to database servers; this will enable us to move partitions around easily. Whenever we want to move a partition, we only have to update the config file to announce the change.

11. Ranking and News Feed Generation

To create the News Feed for any given user, we need to fetch the latest, most popular, and relevant photos of the people the user follows.

For simplicity, let's assume we need to fetch the top 100 photos for a user's News Feed. Our application server will first get a list of people the user follows and then fetch metadata info of each user's latest 100 photos. In the final step, the server will submit all these photos to our ranking algorithm, which will determine the top 100 photos (based on recency, likeness, etc.) and return them to the user. A possible problem with this approach would be higher latency as we have to query multiple tables and perform sorting/merging/ranking on the results. To improve the efficiency, we can pre-generate the News Feed and store it in a separate table.

Pre-generating the News Feed: We can have dedicated servers that are continuously generating users' News Feeds and storing them in a 'UserNewsFeed' table. So whenever any user needs the latest photos for their News-Feed, we will simply query this table and return the results to the user.

Whenever these servers need to generate the News Feed of a user, they will first query the UserNewsFeed table to find the last time the News Feed was generated for that user. Then, new News-Feed data will be generated from that time onwards (following the steps mentioned above).

What are the different approaches for sending News Feed contents to the users?

1. Pull: Clients can pull the News-Feed contents from the server at a regular interval or manually whenever they need it. Possible problems with this approach are a) New data might not be shown to the users until clients issue a pull request b) Most of the time, pull requests will result in an empty response if there is no new data.

2. Push: Servers can push new data to the users as soon as it is available. To efficiently manage this, users have to maintain a **Long Poll** request with the server for receiving the updates. A possible problem with this approach is a user who follows a lot of people or a celebrity user who has millions of followers; in this case, the server has to push updates quite frequently.

3. Hybrid: We can adopt a hybrid approach. We can move all the users who have a high number of followers to a pull-based model and only push data to those who have a few hundred (or thousand) follows. Another approach could be that the server pushes updates to all the users not more than a certain frequency and letting users with a lot of updates to pull data regularly.

For a detailed discussion about News-Feed generation, take a look at 'Designing Facebook's Newsfeed'.

12. News Feed Creation with Sharded Data

One of the most important requirements to create the News Feed for any given user is to fetch the latest photos from all people the user follows. For this, we need to have a mechanism to sort photos on their time of creation. To efficiently do this, we can make photo creation time part of the PhotoID. As we will have a primary index on PhotoID, it will be quite quick to find the latest PhotoIDs.

We can use epoch time for this. Let's say our PhotoID will have two parts; the first part will be representing epoch time, and the second part will be an auto-

incrementing sequence. So to make a new PhotoID, we can take the current epoch time and append an auto-incrementing ID from our key-generating DB. We can figure out the shard number from this PhotoID (PhotoID \% 10) and store the photo there.

What could be the size of our PhotoID? Let's say our epoch time starts today; how many bits we would need to store the number of seconds for the next 50 years?

$$86400 \text{ sec/day} * 365 \text{ (days a year)} * 50 \text{ (years)} \Rightarrow 1.6 \text{ billion seconds}$$

We would need 31 bits to store this number. Since, on average, we are expecting 23 new photos per second, we can allocate 9 additional bits to store the auto-incremented sequence. So every second, we can store ($2^9 \Rightarrow 512$) new photos. We are allocating 9 bits for the sequence number which is more than what we require; we are doing this to get a full byte number (as $40\text{bits} = 5\text{bytes}$). We can reset our auto-incrementing sequence every second.

We will discuss this technique under 'Data Sharding' in 'Designing Twitter'.

13. Cache and Load balancing

Our service would need a massive-scale photo delivery system to serve globally distributed users. Our service should push its content closer to the user using a large number of geographically distributed photo cache servers and use CDNs (for details, see 'Caching').

We can introduce a cache for metadata servers to cache hot database rows. We can use Memcache to cache the data, and Application servers before hitting the database can quickly check if the cache has desired rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system. Under this policy, we discard the least recently viewed row first.

How can we build a more intelligent cache? If we go with the eighty-twenty rule, i.e., 20% of daily read volume for photos is generating 80% of the traffic, which means that certain photos are so popular that most people read them. This dictates that we can try caching 20% of the daily read volume of photos and metadata.

Designing Dropbox

Let's design a file hosting service like Dropbox or Google Drive. Cloud file storage enables users to store their data on remote servers. Usually, these servers are maintained by cloud storage providers and made available to users over a network (typically through the Internet). Users pay for their cloud data storage on a monthly basis.

Similar Services: OneDrive, Google Drive

Difficulty Level: Medium

1. Why Cloud Storage?

Cloud file storage services have become very popular recently as they simplify the storage and exchange of digital resources among multiple devices. The shift from using single personal computers to using multiple devices with different platforms and operating systems such as smartphones and tablets each with portable access from various geographical locations at any time, is believed to be accountable for the huge popularity of cloud storage services. Following are some of the top benefits of such services:

Availability: The motto of cloud storage services is to have data availability anywhere, anytime. Users can access their files/photos from any device whenever and wherever they like.

Reliability and Durability: Another benefit of cloud storage is that it offers 100% reliability and durability of data. Cloud storage ensures that users will never lose their data by keeping multiple copies of the data stored on different geographically located servers.

Scalability: Users will never have to worry about getting out of storage space. With cloud storage you have unlimited storage as long as you are ready to pay for it.

If you haven't used dropbox.com before, we would highly recommend creating an account there and uploading/editing a file and also going through the different options their service offers. This will help you a lot in understanding this chapter.

Designing Dropbox (video)

Here is a video discussing how to design Dropbox:

1:39:45

Designing Dropbox

2. Requirements and Goals of the System

What do we wish to achieve from a Cloud Storage system? Here are the top-level requirements for our system:

1. Users should be able to upload and download their files/photos from any device.
2. Users should be able to share files or folders with other users.
3. Our service should support automatic synchronization between devices, i.e., after updating a file on one device, it should get synchronized on all devices.
4. The system should support storing large files up to a GB.
5. ACID-ity is required. Atomicity, Consistency, Isolation and Durability of all file operations should be guaranteed.
6. Our system should support offline editing. Users should be able to add/delete/modify files while offline, and as soon as they come online, all their changes should be synced to the remote servers and other online devices.

Extended Requirements

The system should support snapshotting of the data, so that users can go back to any version of the files.

3. Some Design Considerations

- We should expect huge read and write volumes.
- Read to write ratio is expected to be nearly the same.
- Internally, files can be stored in small parts or chunks (say 4MB); this can provide a lot of benefits i.e. all failed operations shall only be retried for

smaller parts of a file. If a user fails to upload a file, then only the failing chunk will be retried.

- We can reduce the amount of data exchange by transferring updated chunks only.
- By removing duplicate chunks, we can save storage space and bandwidth usage.
- Keeping a local copy of the metadata (file name, size, etc.) with the client can save us a lot of round trips to the server.
- For small changes, clients can intelligently upload the diffs instead of the whole chunk.

4. Capacity Estimation and Constraints

- Let's assume that we have 500M total users, and 100M daily active users (DAU).
- Let's assume that on average each user connects from three different devices.
- On average if a user has 200 files/photos, we will have 100 billion total files.
- Let's assume that average file size is 100KB, this would give us ten petabytes of total storage.

$$100B * 100KB \Rightarrow 10PB$$

- Let's also assume that we will have one million active connections per minute.

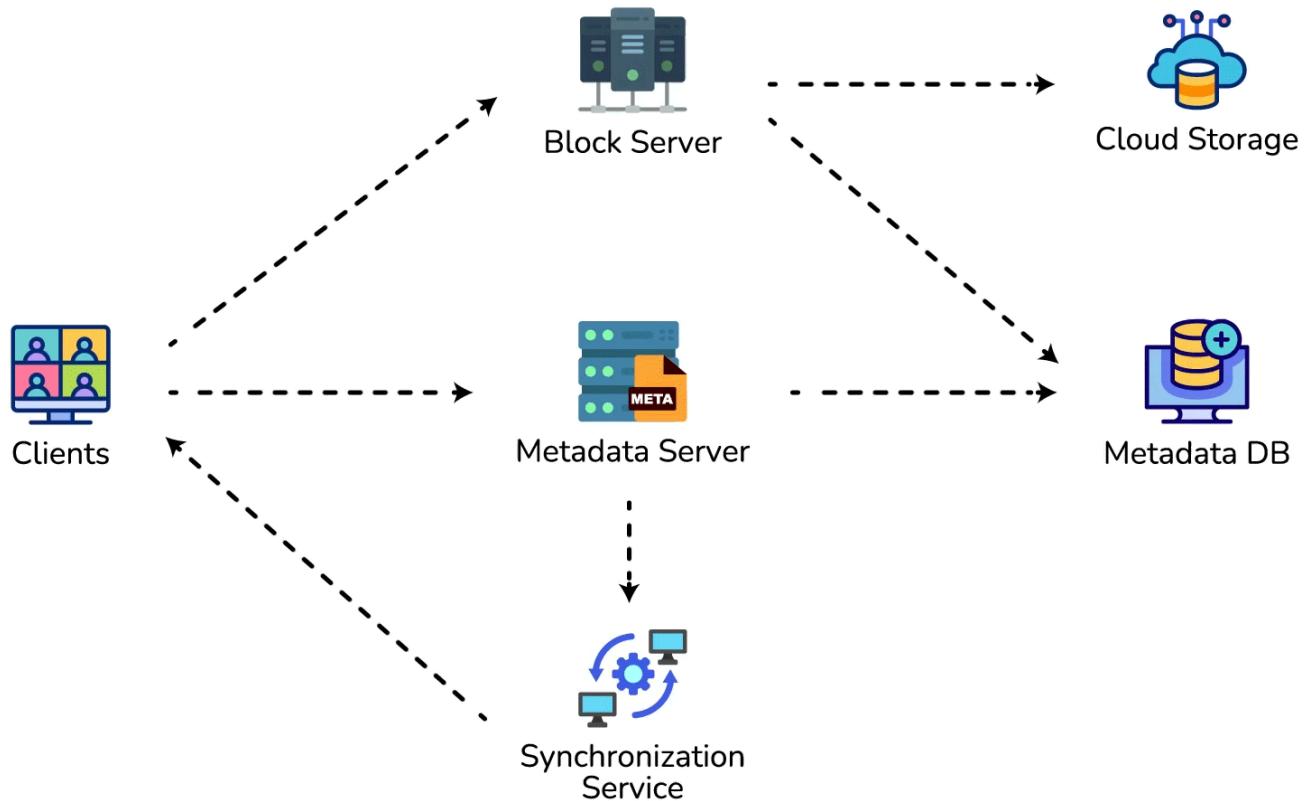
5. High Level Design

The user will specify a folder as the workspace on their device. Any file/photo/folder placed in this folder will be uploaded to the cloud, and whenever a file is modified or deleted, it will be reflected in the same way in the cloud storage. The user can specify similar workspaces on all their devices and any

modification done on one device will be propagated to all other devices to have the same view of the workspace everywhere.

At a high level, we need to store files and their metadata information like File Name, File Size, Directory, etc., and who this file is shared with. So, we need some servers that can help the clients to upload/download files to Cloud Storage and some servers that can facilitate updating metadata about files and users. We also need some mechanism to notify all clients whenever an update happens so they can synchronize their files.

As shown in the diagram below, Block servers will work with the clients to upload/download files from cloud storage and Metadata servers will keep metadata of files updated in a SQL or NoSQL database. Synchronization servers will handle the workflow of notifying all clients about different changes for synchronization.



High Level Design

6. Component Design

Let's go through the major components of our system one by one:

a. Client

The Client Application monitors the workspace folder on the user's machine and syncs all files/folders in it with the remote Cloud Storage. The client application will work with the storage servers to upload, download, and modify actual files to backend Cloud Storage. The client also interacts with the remote Synchronization Service to handle any file metadata updates, e.g., change in the file name, size, modification date, etc.

Here are some of the essential operations for the client:

1. Upload and download files.
2. Detect file changes in the workspace folder.
3. Handle conflict due to offline or concurrent updates.

How do we handle file transfer efficiently? As mentioned above, we can break each file into smaller chunks so that we transfer only those chunks that are modified and not the whole file. Let's say we divide each file into fixed sizes of 4MB chunks. We can statically calculate what could be an optimal chunk size based on 1) Storage devices we use in the cloud to optimize space utilization and input/output operations per second (IOPS) 2) Network bandwidth 3) Average file size in the storage etc. In our metadata, we should also keep a record of each file and the chunks that constitute it.

Should we keep a copy of metadata with Clients? Keeping a local copy of metadata not only enables us to do offline updates but also saves a lot of round trips to update

remote metadata.

How can clients efficiently listen to changes happening with other clients? One solution could be that the clients periodically check with the server if there are any changes. The problem with this approach is that we will have a delay in reflecting changes locally as clients will be checking for changes periodically compared to a server notifying whenever there is some change. If the client frequently checks the server for changes, it will not only be wasting bandwidth, as the server has to return an empty response most of the time, but will also be keeping the server busy. Pulling information in this manner is not scalable.

A solution to the above problem could be to use HTTP long polling. With long polling, the client requests information from the server with the expectation that the server may not respond immediately. If the server has no new data for the client when the poll is received, instead of sending an empty response, the server holds the request open and waits for response information to become available. Once it does have new information, the server immediately sends an HTTP/S response to the client, completing the open HTTP/S Request. Upon receipt of the server response, the client can immediately issue another server request for future updates.

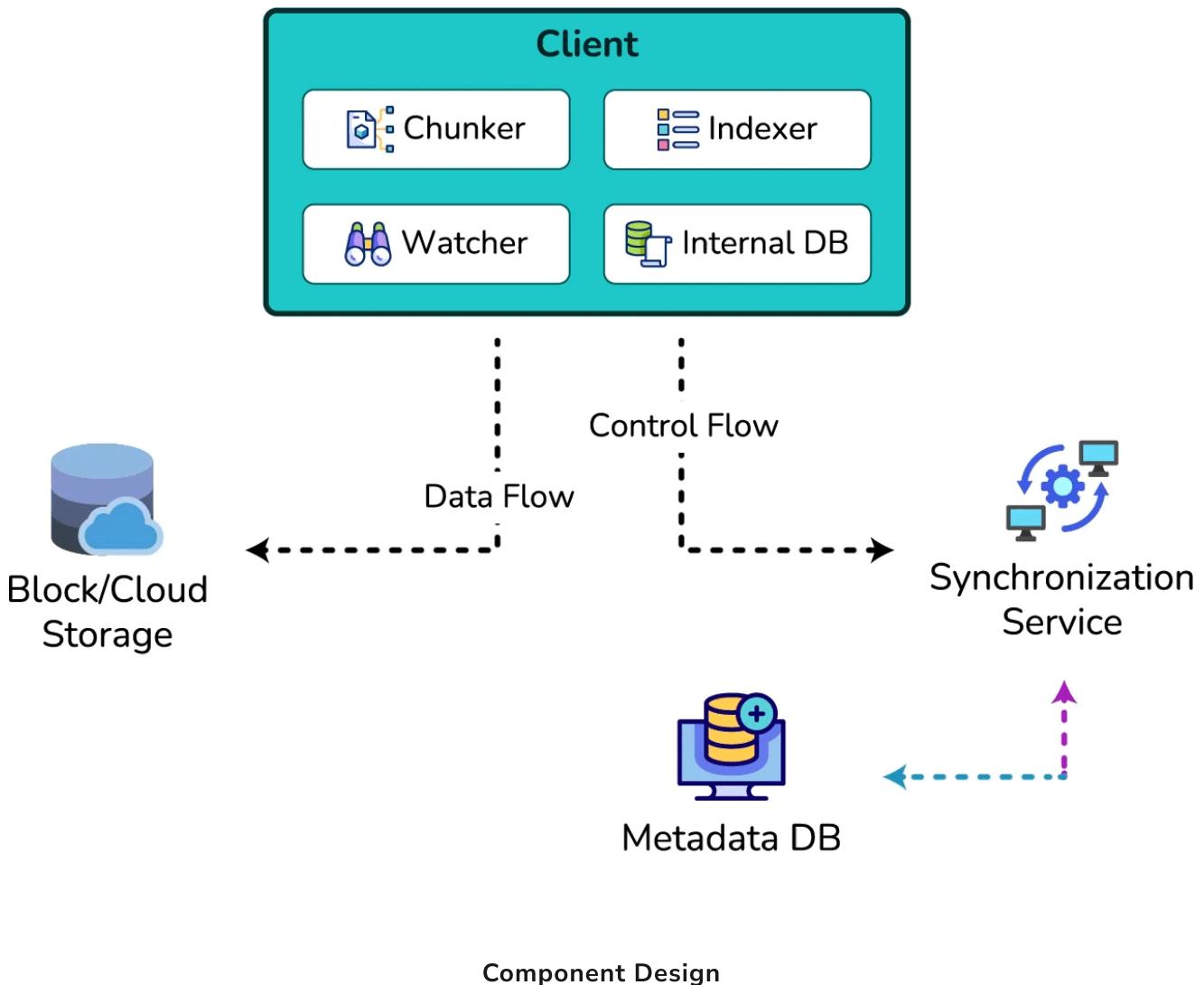
Based on the above considerations, we can divide our client into four parts:

I. Internal Metadata Database will keep track of all the files, chunks, their versions, and their location in the file system.

II. Chunker will split the files into smaller pieces called chunks. It will also be responsible for reconstructing a file from its chunks. Our chunking algorithm will detect the parts of the files that have been modified by the user and only transfer those parts to the Cloud Storage; this will save us bandwidth and synchronization time.

III. Watcher will monitor the local workspace folders and notify the **Indexer** (discussed below) of any action performed by the users, e.g. when users create, delete, or update files or folders. Watcher also listens to any changes happening on other clients that are broadcasted by Synchronization service.

IV. Indexer will process the events received from the Watcher and update the internal metadata database with information about the chunks of the modified files. Once the chunks are successfully submitted/downloaded to the Cloud Storage, the Indexer will communicate with the remote Synchronization Service to broadcast changes to other clients and update the remote metadata database.



How should clients handle slow servers? Clients should exponentially back-off if the server is busy/not-responding. Meaning, if a server is too slow to respond, clients should delay their retries, and this delay should increase exponentially.

Should mobile clients sync remote changes immediately? Unlike desktop or web clients, mobile clients usually sync on-demand to save user's bandwidth and space.

b. Metadata Database

The Metadata Database is responsible for maintaining the versioning and metadata information about files/chunks, users, and workspaces. The Metadata Database can be a relational database such as MySQL or a NoSQL database service such as DynamoDB. Regardless of the type of the database, the Synchronization Service should be able to provide a consistent view of the files using a database, especially if more than one user is working with the same file simultaneously. Since NoSQL data stores do not support ACID properties in favor of scalability and performance, we need to incorporate the support for ACID properties programmatically in the logic of our Synchronization Service in case we opt for this kind of database. However, using a relational database can simplify the implementation of the Synchronization Service as they natively support ACID properties.

The metadata Database should be storing information about following objects:

1. Chunks
2. Files
3. User
4. Devices
5. Workspace (sync folders)

c. Synchronization Service

The Synchronization Service is the component that processes file updates made by a client and applies these changes to other subscribed clients. It also synchronizes clients' local databases with the information stored in the remote Metadata DB. The Synchronization Service is the most important part of the system architecture due to its critical role in managing the metadata and synchronizing users' files.

Desktop clients communicate with the Synchronization Service to either obtain updates from the Cloud Storage or send files and updates to the Cloud Storage and, potentially, other users. If a client was offline for a period, it polls the system for new updates as soon as they come online. When the Synchronization Service receives an update request, it checks with the Metadata Database for consistency and then proceeds with the update. Subsequently, a notification is sent to all subscribed users or devices to report the file update.

The Synchronization Service should be designed to transmit less data between clients and the Cloud Storage to achieve a better response time. To meet this design goal, the Synchronization Service can employ a differencing algorithm to reduce the amount of data that needs to be synchronized. Instead of transmitting entire files from clients to the server or vice versa, we can just transmit the difference between two versions of a file. Therefore, only the part of the file that has been changed is transmitted. This also decreases bandwidth consumption and cloud data storage for the end-user. As described above, we will be dividing our files into 4MB chunks and will be transferring modified chunks only. Server and clients can calculate a hash (e.g., SHA-256) to see whether to update the local copy of a chunk or not. On the server, if we already have a chunk with a similar hash (even from another user), we don't need to create another copy; we can use the same chunk. This is discussed in detail later under Data Deduplication.

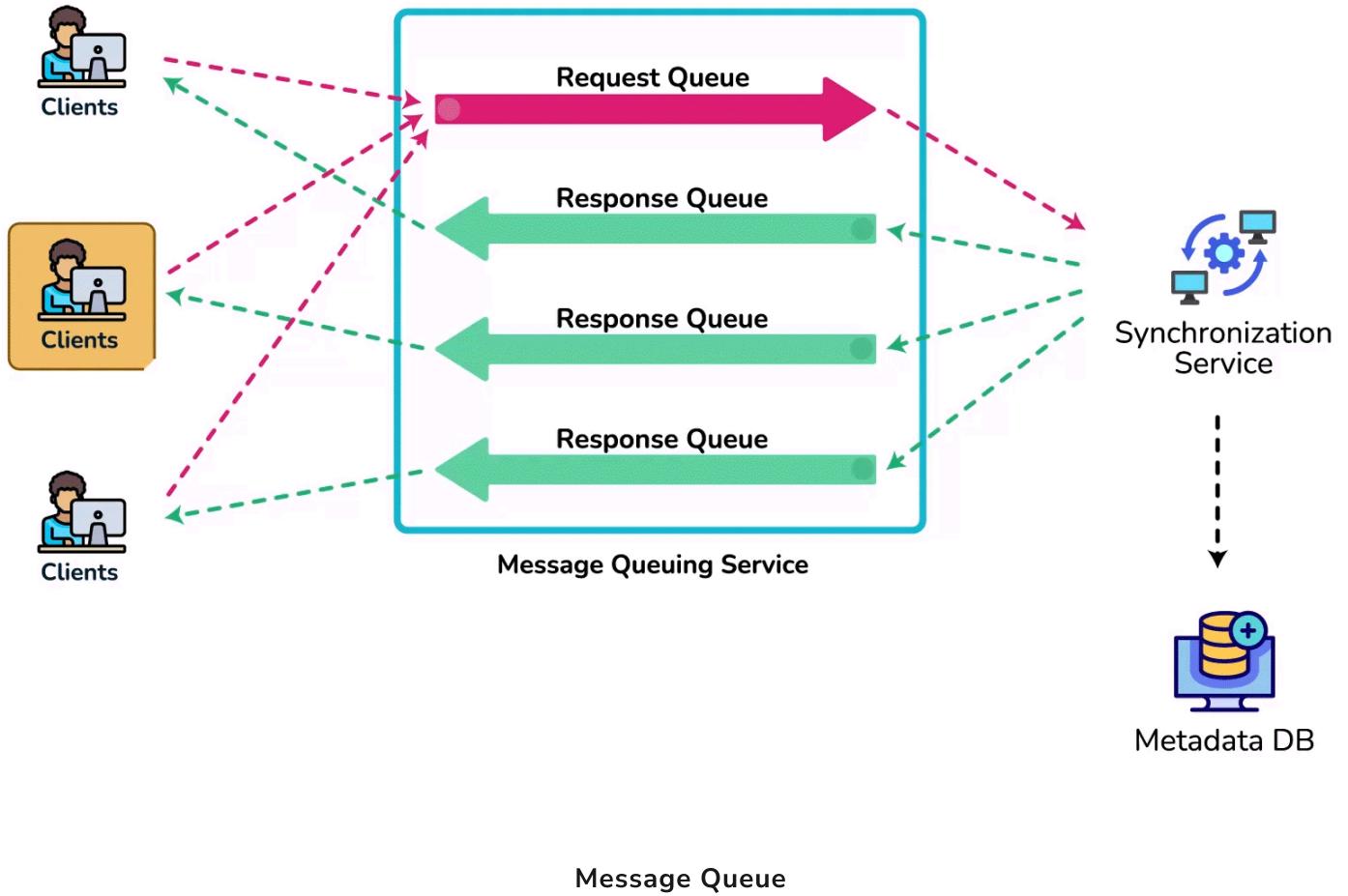
To be able to provide an efficient and scalable synchronization protocol, we can consider using a communication middleware between clients and the Synchronization Service. The messaging middleware should provide scalable message queuing and change notifications to support a high number of clients

using pull or push strategies. This way, multiple Synchronization Service instances can receive requests from a global request [Queue](#), and the communication middleware will be able to balance its load.

d. Message Queuing Service

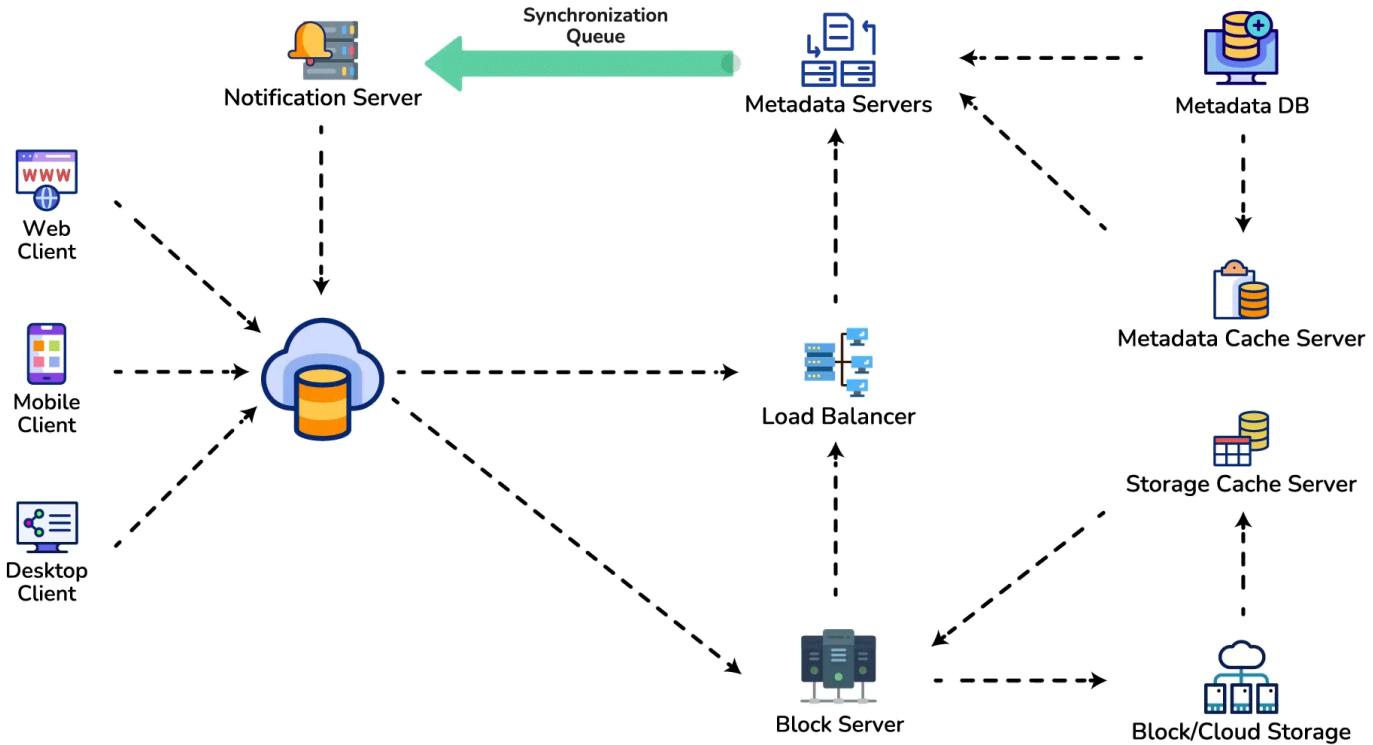
An important part of our architecture is a messaging middleware that should be able to handle a substantial number of requests. A scalable Message Queuing Service that supports asynchronous message-based communication between clients and the Synchronization Service best fits the requirements of our application. The Message Queuing Service supports asynchronous and loosely coupled message-based communication between distributed components of the system. The Message Queuing Service should be able to efficiently store any number of messages in a highly available, reliable, and scalable queue.

The Message Queuing Service will implement two types of queues in our system. The Request Queue is a global queue and all clients will share it. Clients' requests to update the Metadata Database will be sent to the Request Queue first; from there, the Synchronization Service will take it to update metadata. The Response Queues that correspond to individual subscribed clients are responsible for delivering the update messages to each client. Since a message will be deleted from the queue once received by a client, we need to create separate Response Queues for each subscribed client to share update messages.



e. Cloud/Block Storage

Cloud/Block Storage stores chunks of files uploaded by the users. Clients directly interact with the storage to send and receive objects from it. Separation of the metadata from storage enables us to use any storage either in the cloud or in-house.



Detailed component design for Dropbox

7. File Processing Workflow

The sequence below shows the interaction between the components of the application in a scenario when Client A updates a file that is shared with Client B and C, so they should receive the update too. If the other clients are not online at the time of the update, the Message Queuing Service keeps the update notifications in separate response queues for them until they come online later.

1. Client A uploads chunks to cloud storage.
2. Client A updates metadata and commits changes.
3. Client A gets confirmation and notifications are sent to Clients B and C about the changes.
4. Client B and C receive metadata changes and download updated chunks.

8. Data Deduplication

Data deduplication is a technique used for eliminating duplicate copies of data to improve storage utilization. It can also be applied to network data transfers to reduce the number of bytes that must be sent. For each new incoming chunk, we can calculate a hash of it and compare that hash with all the hashes of the existing chunks to see if we already have the same chunk present in our storage.

We can implement deduplication in two ways in our system:

a. Post-process deduplication

With post-process deduplication, new chunks are first stored on the storage device and later some process analyzes the data looking for duplication. The benefit is that clients will not need to wait for the hash calculation or lookup to complete before storing the data, thereby ensuring that there is no degradation in storage performance. Drawbacks of this approach are 1) We will unnecessarily be storing duplicate data, though for a short time, 2) Duplicate data will be transferred consuming bandwidth.

b. In-line deduplication

Alternatively, deduplication hash calculations can be done in real-time as the clients are entering data on their device. If our system identifies a chunk that it has already stored, only a reference to the existing chunk will be added in the metadata, rather than a full copy of the chunk. This approach will give us optimal network and storage usage.

9. Metadata Partitioning

To scale out metadata DB, we need to partition it so that it can store information about millions of users and billions of files/chunks. We need to come up with a partitioning scheme that would divide and store our data in different DB servers.

1. Vertical Partitioning: We can partition our database in such a way that we store tables related to one particular feature on one server. For example, we can store all the user-related tables in one database and all files/chunks related tables in another database. Although this approach is straightforward to implement it has some issues:

1. Will we still have scale issues? What if we have trillions of chunks to be stored and our database cannot support storing such a huge number of records? How would we further partition such tables?
2. Joining two tables in two separate databases can cause performance and consistency issues. How frequently do we have to join user and file tables?

2. Range Based Partitioning: What if we store files/chunks in separate partitions based on the first letter of the File Path? In that case, we save all the files starting with the letter 'A' in one partition and those that start with the letter 'B' into another partition and so on. This approach is called range-based partitioning. We can even combine certain less frequently occurring letters into one database partition. We should come up with this partitioning scheme statically so that we can always store/find a file in a predictable manner.

The main problem with this approach is that it can lead to unbalanced servers. For example, if we decide to put all files starting with the letter 'E' into a DB partition, and later we realize that we have too many files that start with the letter 'E', to such an extent that we cannot fit them into one DB partition.

3. Hash-Based Partitioning: In this scheme we take a hash of the object we are storing and based on this hash we figure out the DB partition to which this object should go. In our case, we can take the hash of the 'FileID' of the File object we are storing to determine the partition the file will be stored. Our hashing function will randomly distribute objects into different partitions, e.g., our hashing function can always map any ID to a number between [1...256], and this number would be the partition we will store our object.

This approach can still lead to overloaded partitions, which can be solved by using 'Consistent Hashing'.

10. Caching

We can have two kinds of caches in our system. To deal with hot files/chunks we can introduce a cache for Block storage. We can use an off-the-shelf solution like [Memcached](#) that can store whole chunks with its respective IDs/Hashes and Block servers before hitting Block storage can quickly check if the cache has desired chunk. Based on clients' usage patterns we can determine how many cache servers we need. A high-end commercial server can have 144GB of memory; one such server can cache 36K chunks.

Which cache replacement policy would best fit our needs? When the cache is full, and we want to replace a chunk with a newer/hotter chunk, how would we choose? **Least Recently Used (LRU)** can be a reasonable policy for our system. Under this policy, we discard the least recently used chunk first. Similarly, we can have a cache for Metadata DB.

11. Load Balancer (LB)

We can add the Load balancing layer at two places in our system: 1) Between Clients and Block servers and 2) Between Clients and Metadata servers. Initially, a simple Round Robin approach can be adopted 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 if a server is dead, LB will take it out of the rotation and will stop sending any traffic to it. A problem with Round Robin LB is, it won't take server load into consideration. If a server is overloaded or slow, the LB will not stop sending new requests to that server. To

handle this, a more intelligent LB solution can be placed that periodically queries backend servers about their load and adjusts traffic based on that.

12. Security, Permissions and File Sharing

One of the primary concerns users will have while storing their files in the cloud is the privacy and security of their data, especially since in our system users can share their files with other users or even make them public to share them with everyone. To handle this, we will be storing the permissions of each file in our metadata DB to reflect what files are visible or modifiable by any user.

Designing Facebook Messenger

Let's design an instant messaging service like Facebook Messenger where users can send text messages to each other through web and mobile interfaces.

Difficulty Level: Medium

1. What is Facebook Messenger?

Facebook Messenger is a software application that provides text-based instant messaging services to its users. Messenger users can chat with their Facebook friends both from cell phones and Facebook's website.

Designing Messenger (video)

Here is a video discussing how to design Facebook Messenger:

1:43:33

Designing Messenger

2. Requirements and Goals of the System

Our Messenger should meet the following requirements:

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.

Extended Requirements:

- Group Chats: Messenger should support multiple people talking to each other in a group.
- Push notifications: Messenger should be able to notify users of new messages when they are offline.

3. Capacity Estimation and Constraints

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 store all the messages for one day, we would need 2TB of storage.

$$20 \text{ billion messages} * 100 \text{ bytes} \Rightarrow 2 \text{ TB/day}$$

To store five years of chat history, we would need 3.6 petabytes of storage.

$$2 \text{ TB} * 365 \text{ days} * 5 \text{ years} \approx 3.6 \text{ PB}$$

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.

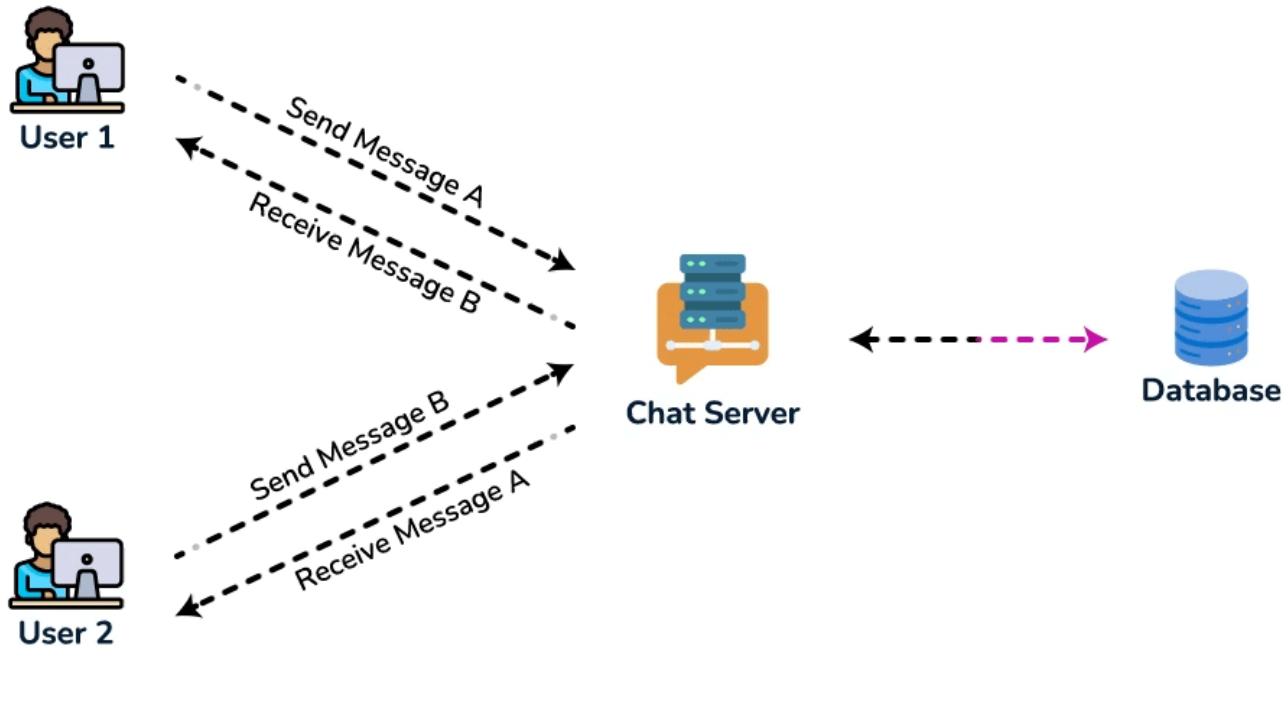
Bandwidth Estimation: If our service is getting 2TB of data every day, this will give us 25MB of incoming data for each second.

2 TB / 86400 sec \approx 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.

4. High Level Design

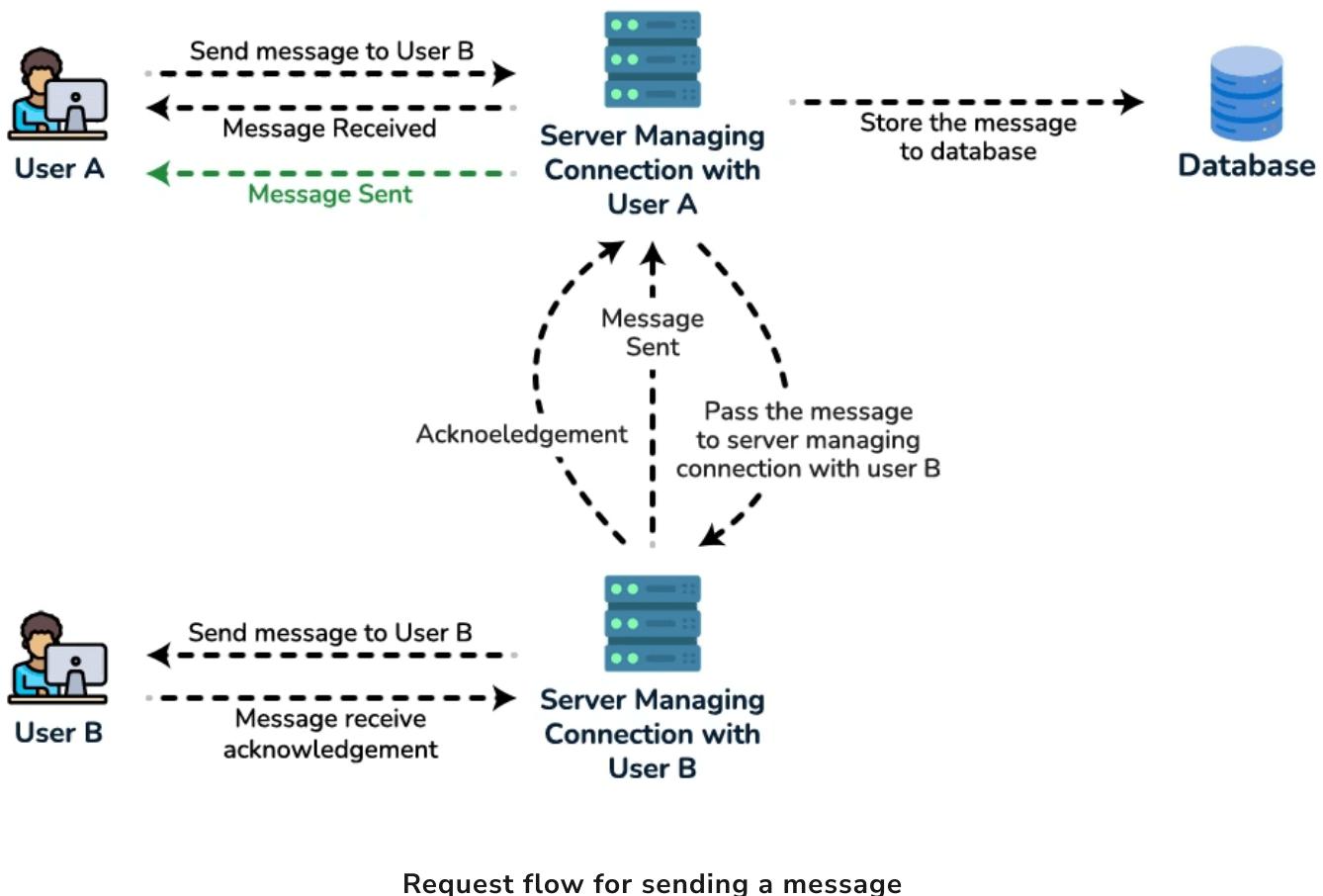
At a high level, we will need a chat server that will be the central piece orchestrating all the communications between users. For example, when a user wants to send a message to another user, they will connect to the chat server and send the message to the server; the server then passes that message to the other user and also stores it in the database.



High Level Design

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.



5. Detailed Component Design

Let's try to build a simple solution first where everything runs on one server.

At the high level, our system needs to handle the following use cases:

1. Receive incoming messages and deliver outgoing messages.
2. Store and retrieve messages from the database.
3. Keep a record of which user is online or has gone offline, and notify all the relevant users about these status changes.

Let's talk about these scenarios one by one:

a. 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.
2. **Push model:** Users can keep a connection open with the server and can depend upon the server to notify them whenever there are new messages.

In the first approach, 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 messages. This will waste a lot of resources and does not look like an efficient solution.

If we go with our second approach, where all the active users keep a connection open with the server, then as soon as the server receives a message, it can immediately pass the message to the intended user. This way, the server does not need to keep track of the pending messages, and we will have minimum latency, as the messages are delivered instantly on the opened connection.

How will clients maintain an open connection with the server? We can use HTTP 'Long Polling' or 'WebSockets'. In long polling, clients can request information from the server with the expectation that the server may not respond immediately. If the server has no new data for the client when the poll is received, instead of sending an empty response, the server holds the request open and waits for response information to become available. Once it does have new information, the server immediately sends the response to the client, completing the open request. Upon receipt of the server response, the client can immediately issue another server request for future updates. This gives a lot of improvements in latencies, throughputs, and performance. However, the long polling request can timeout or receive a disconnect from the server; in that case, the client has to open a new request.

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.

How should the server process a 'deliver message' request? The server needs to do the following things upon receiving a new message: 1) Store the message in the database, 2) Send the message to the receiver, and 3) Send an acknowledgment to the sender.

The chat server will first find the server that holds the connection for the receiver and pass the message to that server to send it to the receiver. The chat server can then send the acknowledgment to the sender; we don't need to wait to store the message in the database (this can happen in the background). Storing the message is discussed in the next section.

How does the messenger maintain the sequencing of the messages? We can store a timestamp with each message, which is the time when the server receives the message. However, this will still not ensure the correct ordering of messages for clients. The scenario where the server timestamp cannot determine the exact order of messages would look like this:

1. User-1 sends a message M1 to the server for User-2.
2. The server receives M1 at T1.
3. Meanwhile, User-2 sends a message M2 to the server for User-1.
4. The server receives the message M2 at T2, such that $T2 > T1$.
5. The server sends the message M1 to User-2 and M2 to User-1.

So User-1 will see M1 first and then M2, whereas User-2 will see M2 first and then M1.

To resolve this, we need to keep a sequence number with every message for each client. This sequence number will determine the exact ordering of messages for

EACH user. With this solution, both clients will see a different view of the message sequence, but this view will be consistent for them on all devices.

b. 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.

We have to keep certain things in mind while designing our database:

1. How to efficiently work with the database connection pool.
2. How to retry failed requests.
3. Where to log those requests that failed even after some retries.
4. How to retry these logged requests (that failed after the retry) when all the issues have been resolved.

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.

We cannot use RDBMS like MySQL or NoSQL like MongoDB because we cannot afford to read/write a row from the database every time a user receives/sends a message. This will not only make the basic operations of our service run with high latency but also create a huge load on databases.

Both of our requirements can be easily met with a wide-column database solution like **HBase**. HBase is a column-oriented key-value NoSQL database that can store multiple values against one key into multiple columns. HBase is modeled after

Google's [BigTable](#) and runs on top of Hadoop Distributed File System ([HDFS](#)). HBase groups data together to store new data in a memory buffer and, once the buffer is full, it dumps the data to the disk. This way of storage not only helps to store a lot of small data quickly but also fetching rows by the key or scanning ranges of rows. HBase is also an efficient database to store variable-sized data, which is also required by our service.

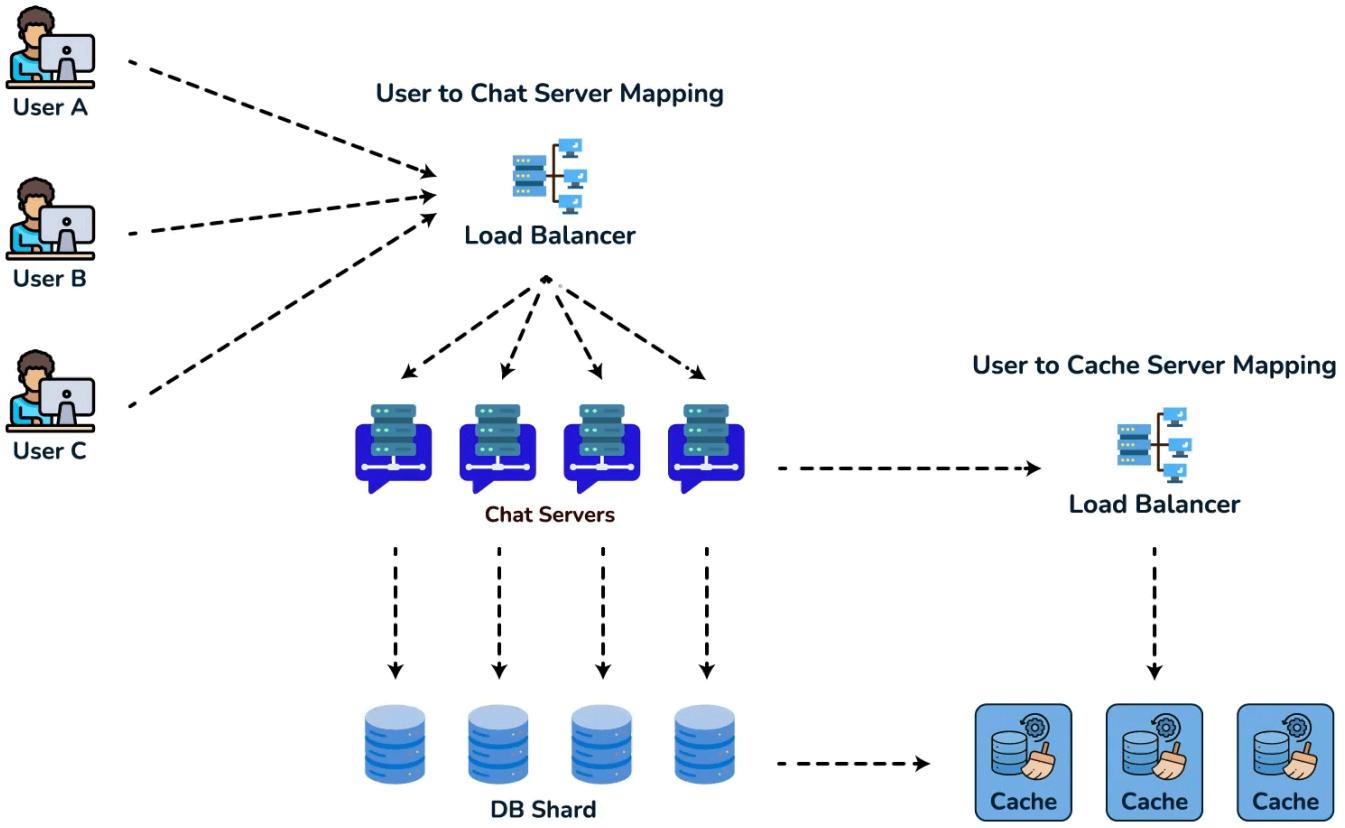
How should clients efficiently fetch data from the server? Clients should paginate while fetching data from the server. Page size could be different for different clients, e.g., cell phones have smaller screens, so we need fewer messages/conversations in the viewport.

c. 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 500M active users at any time, if we have to broadcast each status change to all the relevant active users, it will consume a lot of resources. We can do the following optimization around this:

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.

- Whenever the client starts a new chat with another user, we can pull the status at that time.



Detailed component design for Facebook messenger

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 range-based searches. The servers can broadcast the online status of a user to other relevant users. Clients can pull status updates for users who are visible in the client's viewport on a less frequent basis.

6. 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 " $3.6\text{PB}/4\text{TB} \approx 900$ " shards for five years. For simplicity, let's assume we keep 1K shards. So we will find the shard number by " $\text{hash}(\text{UserID}) \% 1000$ " and then store/retrieve the data from there. This partitioning scheme will also be very quick to fetch chat history for any user.

In the beginning, we can start with fewer database servers with multiple shards residing on one physical server. Since we can have multiple database instances on a server, we can easily store multiple partitions on a single server. Our hash function needs to understand this logical partitioning scheme so that it can map multiple logical partitions on one physical server.

Since we will store an unlimited history of messages, we can start with a large number of logical partitions that will be mapped to fewer physical servers. Then, as our storage demand increases, we can add more physical servers to distribute our logical partitions.

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.

7. 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 messages on one shard, the cache for a user should entirely reside on one machine too.

8. Load balancing

We will need a load balancer in front of our chat servers that can map each UserID to a server that holds the connection for the user and then direct the request to that server. Similarly, we would need a load balancer for our cache servers.

9. Fault tolerance and Replication

What will happen when a chat server fails? Our chat servers are holding connections with the users. If a server goes down, should we devise a mechanism to transfer those connections to some other server? It's extremely hard to failover TCP connections to other servers; an easier approach can be to have clients automatically reconnect if the connection is lost.

Should we store multiple copies of user messages? We cannot store only one copy of the user's data because if the server holding the data crashes or is down permanently, we don't have any mechanism to recover that data. For this, either we have to store multiple copies of the data on different servers or use techniques like Reed-Solomon encoding to distribute and replicate it.

10. Extended Requirements

a. Group chat

We can have separate group-chat objects in our system that can be stored on the chat servers. A group-chat object is identified by `GroupChatID` and will also maintain a list of people who are part of that chat. Our load balancer can direct each group chat message based on `GroupChatID`, and the server handling that

group chat can iterate through all the users of the chat to find the server handling the connection of each user to deliver the message.

In databases, we can store all the group chats in a separate table partitioned based on GroupChatID .

b. Push notifications

In our current design, users can only send messages to online users; if the receiving user is offline, we send a failure to the sending user. Push notifications will enable our system to send messages to offline users.

For Push notifications, each user can opt-in from their device (or a web browser) to get notifications whenever there is a new message or event. Each manufacturer maintains a set of servers that handles pushing these notifications to the user.

To have push notifications in our system, we would need to set up a Notification server, which will take the messages for offline users and send them to the manufacturer's push notification server, which will then send them to the user's device.

Designing Twitter

Let's design a Twitter-like social networking service. Users of the service will be able to post tweets, follow other people, and favorite tweets.

Difficulty Level: Medium

1. What is Twitter?

Twitter is an online social networking service where users post and read short messages called "tweets." Registered users can post and read tweets, but those who are not registered can only read them. Users access Twitter through their website interface, SMS, or mobile app.

2. Requirements and Goals of the System

We will be designing a simpler version of Twitter with the following requirements:

Functional Requirements

1. Users should be able to post new tweets.
2. A user should be able to follow other users.
3. Users should be able to mark tweets as favorites.
4. The service should be able to create and display a user's timeline consisting of top tweets from all the people the user follows.
5. Tweets can contain photos and videos.

Non-functional Requirements

1. Our service needs to be highly available.
2. Acceptable latency of the system is 200ms for timeline generation.
3. Consistency can take a hit (in the interest of availability); if a user doesn't see a tweet for a while, it should be fine.

Extended Requirements

1. Searching for tweets.
2. Replying to a tweet.
3. Trending topics – current hot topics/searches.
4. Tagging other users.
5. Tweet Notification.
6. Who to follow? Suggestions?
7. Moments.

3. Capacity Estimation and Constraints

Let's assume we have one billion total users with 200 million daily active users (DAU). Also assume we have 100 million new tweets every day and on average each user follows 200 people.

How many favorites per day? If, on average, each user favorites five tweets per day we will have:

$$200M \text{ users} * 5 \text{ favorites} \Rightarrow 1B \text{ favorites}$$

How many total tweet-views will our system generate? Let's assume on average a user visits their timeline two times a day and visits five other people's pages. On each page if a user sees 20 tweets, then our system will generate 28B/day total tweet-views:

$$200M \text{ DAU} * ((2 + 5) * 20 \text{ tweets}) \Rightarrow 28B/\text{day}$$

Storage Estimates Let's say each tweet has 140 characters and we need two bytes to store a character without compression. Let's assume we need 30 bytes to store metadata with each tweet (like ID, timestamp, user ID, etc.). Total storage we would need:

$$100M * (280 + 30) \text{ bytes} \Rightarrow 30\text{GB/day}$$

How much storage space would we need for five years? How much storage we would need for users' data, follows, favorites? We will leave this for the exercise.

Not all tweets will have media, let's assume that on average every fifth tweet has a photo and every tenth has a video. Let's also assume on average a photo is 200KB and a video is 2MB. This will lead us to have 24TB of new media every day.

$$(100M/5 \text{ photos} * 200\text{KB}) + (100M/10 \text{ videos} * 2\text{MB}) \approx 24\text{TB/day}$$

Bandwidth Estimates Since total ingress is 24TB per day, this would translate into 290MB/sec.

Remember that we have 28B tweet views per day. We must show the photo of every tweet (if it has a photo), but let's assume that the users watch every 3rd video they see in their timeline. So, total egress will be:

$$\begin{aligned} & (28B * 280 \text{ bytes}) / 86400\text{s of text} \Rightarrow 93\text{MB/s} \\ & + (28B/5 * 200\text{KB}) / 86400\text{s of photos} \Rightarrow 13\text{GB/S} \\ & + (28B/10/3 * 2\text{MB}) / 86400\text{s of Videos} \Rightarrow 22\text{GB/s} \\ & \text{Total } \approx 35\text{GB/s} \end{aligned}$$

4. System APIs

💡 Once we've finalized the requirements, it's always a good idea to define the

system APIs. This should explicitly state what is expected from the system.

We can have SOAP or REST APIs to expose the functionality of our service.

Following could be the definition of the API for posting a new tweet:

```
tweet(api_dev_key, tweet_data, tweet_location,  
user_location, media_ids)
```

Parameters:

api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.

tweet_data (string): The text of the tweet, typically up to 140 characters.

tweet_location (string): Optional location (longitude, latitude) this Tweet refers to.

user_location (string): Optional location (longitude, latitude) of the user adding the tweet.

media_ids (number[]): Optional list of media_ids to be associated with the Tweet.
(all the media photo, video, etc. need to be uploaded separately).

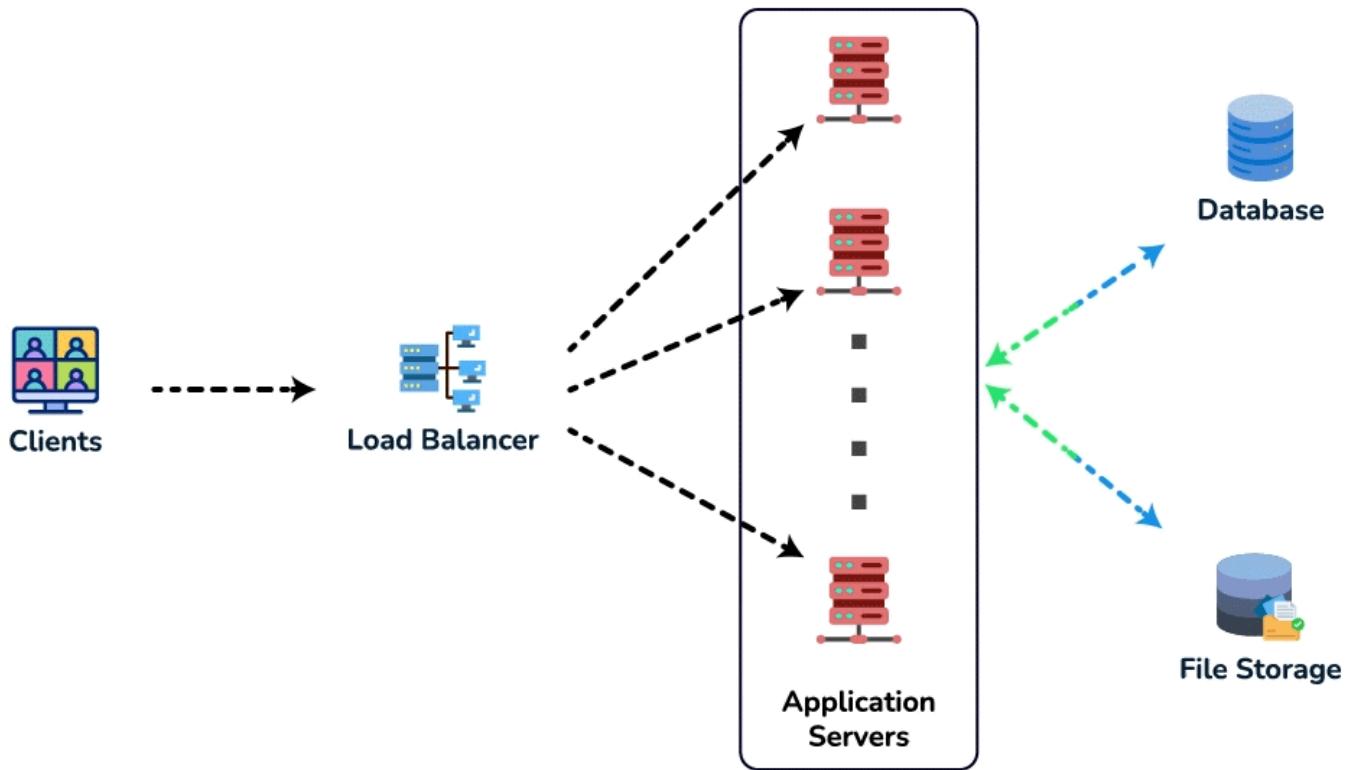
Returns: (string)

A successful post will return the URL to access that tweet. Otherwise, an appropriate HTTP error is returned.

5. High Level System Design

We need a system that can efficiently store all the new tweets, $100M/86400s \Rightarrow 1150$ tweets per second and read $28B/86400s \Rightarrow 325K$ tweets per second. It is clear from the requirements that this will be a read-heavy system.

At a high level, we need multiple application servers to serve all these requests with load balancers in front of them for traffic distributions. On the backend, we need an efficient database that can store all the new tweets and can support a huge number of reads. We also need some file storage to store photos and videos.



High Level Design

Although our expected daily write load is 100 million and read load is 28 billion tweets. This means on average our system will receive around 1160 new tweets and 325K read requests per second. This traffic will be distributed unevenly throughout the day, though, at peak time we should expect at least a few thousand write requests and around 1M read requests per second. We should keep this in mind while designing the architecture of our system.

6. Database Schema

We need to store data about users, their tweets, their favorite tweets, and people they follow.

Tweet		User		UserFollow	
PK	<u>TweetID: int</u>	PK	<u>UserID: int</u>	PK	<u>FollowerID: int</u> <u>FolloweeID: int</u>
	UserID: int Content: varchar TweetLatitude: int TweetLongitude: int UserLatitude: int UserLongitude: int CreationDate: datetime NumFavorites: int		Name: varchar Email: varchar DateOfBirth: datetime CreationDate: datetime LastLogin: datetime		

DB Schema

For choosing between SQL and NoSQL databases to store the above schema, please see 'Database schema' under 'Designing Instagram'.

7. Data Sharding

Since we have a huge number of new tweets every day and our read load is extremely high too, we need to distribute our data onto multiple machines such that we can read/write it efficiently. We have many options to shard our data; let's go through them one by one:

Sharding based on UserID: We can try storing all the data of a user on one server. While storing, we can pass the UserID to our hash function that will map the user to a database server where we will store all of the user's tweets, favorites, follows, etc. While querying for tweets/follows/favorites of a user, we can ask our hash

function where can we find the data of a user and then read it from there. This approach has a couple of issues:

1. What if a user becomes hot? There could be a lot of queries on the server holding the user. This high load will affect the performance of our service.
2. Over time some users can end up storing a lot of tweets or having a lot of follows compared to others. Maintaining a uniform distribution of growing user data is quite difficult.

To recover from these situations either we have to repartition/redistribute our data or use consistent hashing.

Sharding based on TweetID: Our hash function will map each TweetID to a random server where we will store that Tweet. To search for tweets, we have to query all servers, and each server will return a set of tweets. A centralized server will aggregate these results to return them to the user. Let's look into timeline generation example; here are the number of steps our system has to perform to generate a user's timeline:

1. Our application (app) server will find all the people the user follows.
2. App server will send the query to all database servers to find tweets from these people.
3. Each database server will find the tweets for each user, sort them by recency and return the top tweets.
4. App server will merge all the results and sort them again to return the top results to the user.

This approach solves the problem of hot users, but, in contrast to sharding by UserID, we have to query all database partitions to find tweets of a user, which can result in higher latencies.

We can further improve our performance by introducing cache to store hot tweets in front of the database servers.

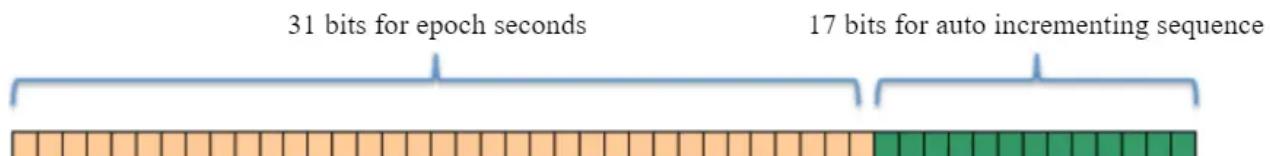
Sharding based on Tweet creation time: Storing tweets based on creation time will give us the advantage of fetching all the top tweets quickly and we only have to query a very small set of servers. The problem here is that the traffic load will not be distributed, e.g., while writing, all new tweets will be going to one server and the remaining servers will be sitting idle. Similarly, while reading, the server holding the latest data will have a very high load as compared to servers holding old data.

What if we can combine sharding by TweetID and Tweet creation time? If we don't store tweet creation time separately and use TweetID to reflect that, we can get benefits of both the approaches. This way it will be quite quick to find the latest Tweets. For this, we must make each TweetID universally unique in our system and each TweetID should contain a timestamp too.

We can use epoch time for this. Let's say our TweetID will have two parts: the first part will be representing epoch seconds and the second part will be an auto-incrementing sequence. So, to make a new TweetID, we can take the current epoch time and append an auto-incrementing number to it. We can figure out the shard number from this TweetID and store it there.

What could be the size of our TweetID? Let's say our epoch time starts today, how many bits we would need to store the number of seconds for the next 50 years?

$$86400 \text{ sec/day} * 365 \text{ (days a year)} * 50 \text{ (years)} \Rightarrow 1.6B$$



We would need 31 bits to store this number. Since on average we are expecting 1150 new tweets per second, we can allocate 17 bits to store auto incremented sequence; this will make our TweetID 48 bits long. So, every second we can store ($2^{17} \Rightarrow 130K$) new tweets. We can reset our auto incrementing sequence every second. For

fault tolerance and better performance, we can have two database servers to generate auto-incrementing keys for us, one generating even numbered keys and the other generating odd numbered keys.

If we assume our current epoch seconds are "1483228800," our TweetID will look like this:

```
1483228800 000001  
1483228800 000002  
1483228800 000003  
1483228800 000004  
...
```

If we make our TweetID 64bits (8 bytes) long, we can easily store tweets for the next 100 years and also store them for mili-seconds granularity.

In the above approach, we still have to query all the servers for timeline generation, but our reads (and writes) will be substantially quicker.

1. Since we don't have any secondary index (on creation time) this will reduce our write latency.
2. While reading, we don't need to filter on creation-time as our primary key has epoch time included in it.

8. Cache

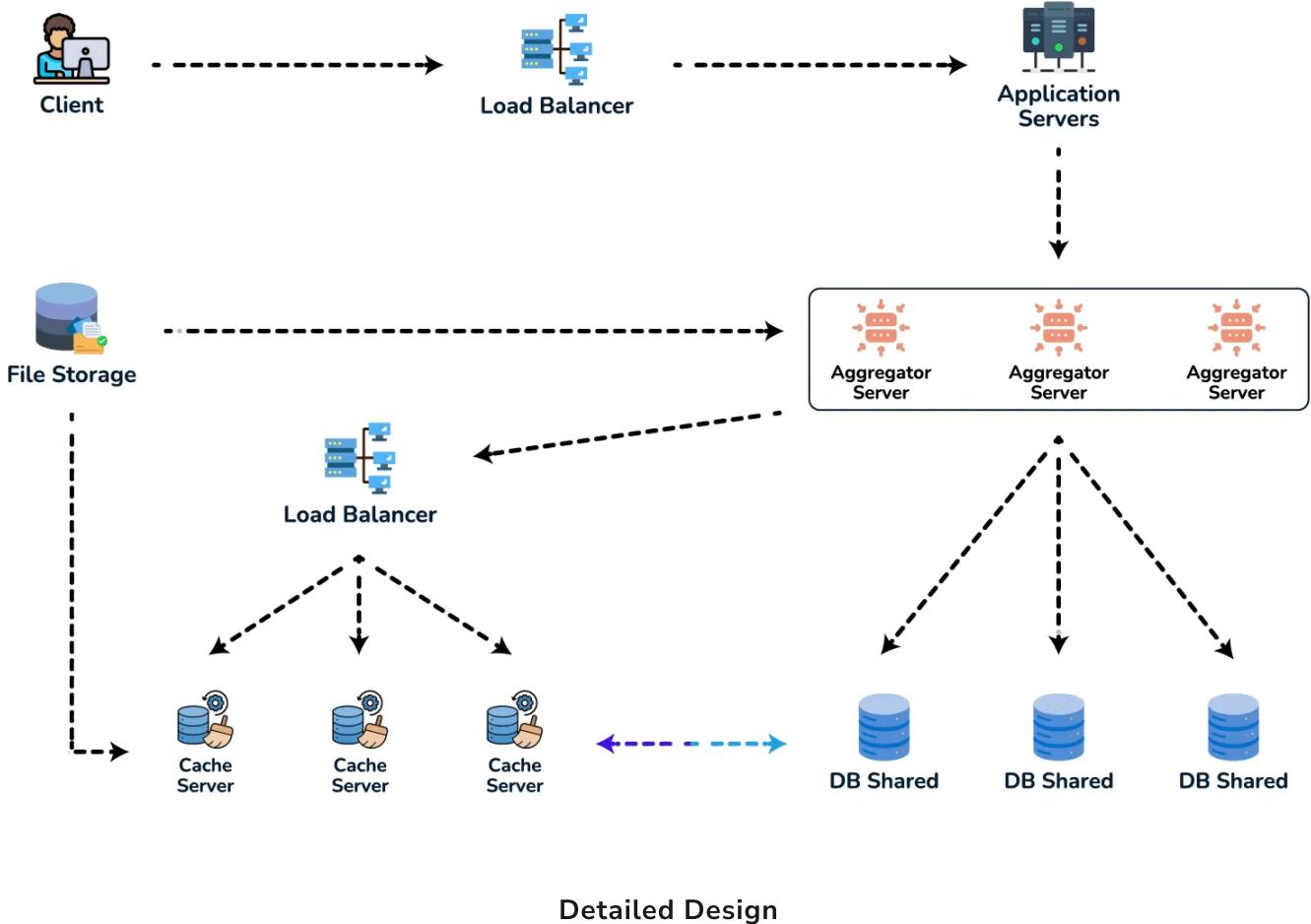
We can introduce a cache for database servers to cache hot tweets and users. We can use an off-the-shelf solution like Memcache that can store the whole tweet objects. Application servers, before hitting database, can quickly check if the cache has desired tweets. Based on clients' usage patterns we can determine how many cache servers we need.

Which cache replacement policy would best fit our needs? When the cache is full and we want to replace a tweet with a newer/hotter tweet, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system. Under this policy, we discard the least recently viewed tweet first.

How can we have a more intelligent cache? If we go with 80-20 rule, that is 20% of tweets generating 80% of read traffic which means that certain tweets are so popular that a majority of people read them. This dictates that we can try to cache 20% of daily read volume from each shard.

What if we cache the latest data? Our service can benefit from this approach. Let's say if 80% of our users see tweets from the past three days only; we can try to cache all the tweets from the past three days. Let's say we have dedicated cache servers that cache all the tweets from all the users from the past three days. As estimated above, we are getting 100 million new tweets or 30GB of new data every day (without photos and videos). If we want to store all the tweets from last three days, we will need less than 100GB of memory. This data can easily fit into one server, but we should replicate it onto multiple servers to distribute all the read traffic to reduce the load on cache servers. So whenever we are generating a user's timeline, we can ask the cache servers if they have all the recent tweets for that user. If yes, we can simply return all the data from the cache. If we don't have enough tweets in the cache, we have to query the backend server to fetch that data. On a similar design, we can try caching photos and videos from the last three days.

Our cache would be like a hash table where 'key' would be 'OwnerID' and 'value' would be a doubly linked list containing all the tweets from that user in the past three days. Since we want to retrieve the most recent data first, we can always insert new tweets at the head of the linked list, which means all the older tweets will be near the tail of the linked list. Therefore, we can remove tweets from the tail to make space for newer tweets.



9. Timeline Generation

For a detailed discussion about timeline generation, take a look at 'Designing Facebook's Newsfeed'.

10. Replication and Fault Tolerance

Since our system is read-heavy, we can have multiple secondary database servers for each DB partition. Secondary servers will be used for read traffic only. All writes will first go to the primary server and then will be replicated to secondary

servers. This scheme will also give us fault tolerance, since whenever the primary server goes down we can failover to a secondary server.

11. Load Balancing

We can add Load balancing layer at three places in our system 1) Between Clients and Application servers 2) Between Application servers and database replication servers and 3) Between Aggregation servers and Cache server. Initially, a simple Round Robin approach can be adopted; that distributes incoming requests equally among 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 will stop sending any traffic to it. A problem with Round Robin LB is that it won't take servers load into consideration. If a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries backend server about their load and adjusts traffic based on that.

12. Monitoring

Having the ability to monitor our systems is crucial. We should constantly collect data to get an instant insight into how our system is doing. We can collect following metrics/counters to get an understanding of the performance of our service:

1. New tweets per day/second, what is the daily peak?
2. Timeline delivery stats, how many tweets per day/second our service is delivering.
3. Average latency that is seen by the user to refresh timeline.

By monitoring these counters, we will realize if we need more replication, load balancing, or caching.

13. Extended Requirements

How do we serve feeds? Get all the latest tweets from the people someone follows and merge/sort them by time. Use pagination to fetch/show tweets. Only fetch top N tweets from all the people someone follows. This N will depend on the client's Viewport, since on a mobile we show fewer tweets compared to a Web client. We can also cache next top tweets to speed things up.

Alternately, we can pre-generate the feed to improve efficiency; for details please see 'Ranking and timeline generation' under 'Designing Instagram'.

Retweet: With each Tweet object in the database, we can store the ID of the original Tweet and not store any contents on this retweet object.

Trending Topics: We can cache most frequently occurring hashtags or search queries in the last N seconds and keep updating them after every M seconds. We can rank trending topics based on the frequency of tweets or search queries or retweets or likes. We can give more weight to topics which are shown to more people.

Who to follow? How to give suggestions? This feature will improve user engagement. We can suggest friends of people someone follows. We can go two or three levels down to find famous people for the suggestions. We can give preference to people with more followers.

As only a few suggestions can be made at any time, use Machine Learning (ML) to shuffle and re-prioritize. ML signals could include people with recently increased follow-ship, common followers if the other person is following this user, common location or interests, etc.

Moments: Get top news for different websites for past 1 or 2 hours, figure out related tweets, prioritize them, categorize them (news, support, financial, entertainment, etc.) using ML – supervised learning or Clustering. Then we can show these articles as trending topics in Moments.

Search: Search involves Indexing, Ranking, and Retrieval of tweets. A similar solution is discussed in our next problem 'Design Twitter Search'.

Designing Youtube or Netflix

Let's design a video sharing service like Youtube, where users will be able to upload/view/search videos.

Similar Services: netflix.com, vimeo.com, dailymotion.com, veoh.com

Difficulty Level: Medium

1. Why Youtube?

Youtube is one of the most popular video sharing websites in the world. Users of the service can upload, view, share, rate, and report videos as well as add comments on videos.

Designing Youtube (video)

Here is a video discussing how to design Youtube:

1:37:59

Designing Youtube

2. Requirements and Goals of the System

For the sake of this exercise, we plan to design a simpler version of Youtube with following requirements:

Functional Requirements:

1. Users should be able to upload videos.
2. Users should be able to share and view videos.
3. Users should be able to perform searches based on 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 fine.
3. Users should have a real-time experience while watching videos and should not feel any lag.

Not in scope: Video recommendations, most popular videos, channels, subscriptions, watch later, favorites, etc.

3. Capacity Estimation and Constraints

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:

$$800M * 5 / 86400 \text{ sec} \Rightarrow 46K \text{ videos/sec}$$

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.

$$46K / 200 \Rightarrow 230 \text{ videos/sec}$$

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:

$$500 \text{ hours} * 60 \text{ min} * 50\text{MB} \Rightarrow 1500 \text{ GB/min (25 GB/sec)}$$

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.

$$500 \text{ hours} * 60 \text{ mins} * 10\text{MB} \Rightarrow 300\text{GB/min (5GB/sec)}$$

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

4. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. The following could be the definitions of the APIs for uploading and searching videos:

```
uploadVideo(api_dev_key, video_title, video_description,
tags[], category_id, default_language, recording_details,
video_contents)
```

Parameters:

api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.

video_title (string): Title of the video.

video_description (string): Optional description of the video.

tags (string[]): Optional tags for the video.

category_id (string): Category of the video, e.g., Film, Song, People, etc.

default_language (string): For example English, Mandarin, Hindi, etc.

`recording_details` (string): Location where the video was recorded.

`video_contents` (stream): Video to be uploaded.

Returns: (string)

A successful upload will return HTTP 202 (request accepted) and once the video encoding is completed the user is notified through email with a link to access the video. We can also expose a queryable API to let users know the current status of their uploaded video.

```
searchVideo(api_dev_key, search_query, user_location,  
maximum_videos_to_return, page_token)
```

Parameters:

`api_dev_key` (string): The API developer key of a registered account of our service.

`search_query` (string): A string containing the search terms.

`user_location` (string): Optional location of the user performing the search.

`maximum_videos_to_return` (number): Maximum number of results returned in one request.

`page_token` (string): This token will specify a page in the result set that should be returned.

Returns: (JSON)

A JSON containing information about the list of video resources matching the search query. Each video resource will have a video title, a thumbnail, a video creation date, and a view count.

```
streamVideo(api_dev_key, video_id, offset, codec,  
resolution)
```

Parameters:

api_dev_key (string): The API developer key of a registered account of our service.

video_id (string): A string to identify the video.

offset (number): We should be able to stream video from any offset; this offset would be a time in seconds from the beginning of the video. If we support playing/pausing a video from multiple devices, we will need to store the offset on the server. This will enable the users to start watching a video on any device from the same point where they left off.

codec (string) & resolution(string): We should send the codec and resolution info in the API from the client to support play/pause from multiple devices. Imagine you are watching a video on your TV's Netflix app, paused it, and started watching it on your phone's Netflix app. In this case, you would need codec and resolution, as both these devices have a different resolution and use a different codec.

Returns: (STREAM)

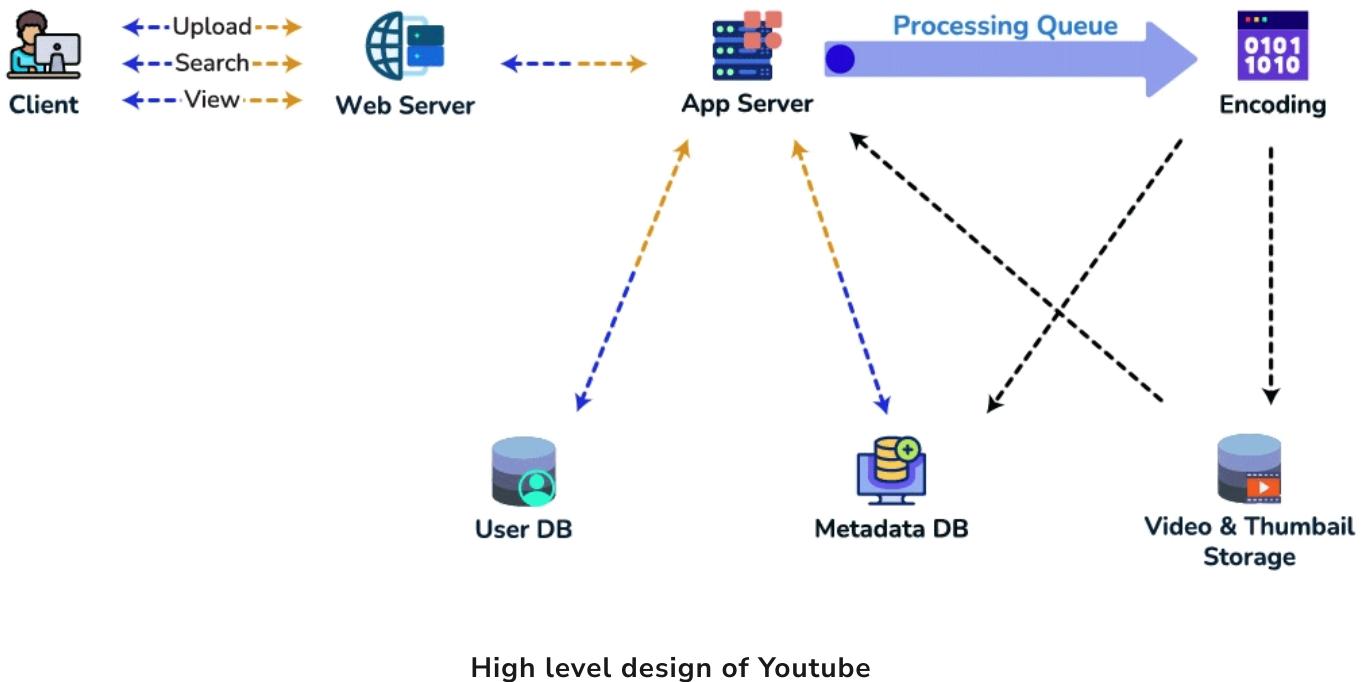
A media stream (a video chunk) from the given offset.

5. High Level Design

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.



6. Database Schema

Video metadata storage - MySql

Videos metadata can be stored in a SQL database. The following information should be stored with each video:

- VideoID
- Title
- Description
- Size
- Thumbnail
- Uploader/User
- Total number of likes
- Total number of dislikes

- Total number of views

For each video comment, we need to store following information:

- CommentID
- VideoID
- UserID
- Comment
- TimeOfCreation

User data storage - MySql

- UserID, Name, email, address, age, registration details, etc.

7. 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](#) or [GlusterFS](#).

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. Such configurations can cause some staleness in data, e.g., when a new video is added, its metadata would be inserted in the primary first, and before it gets applied to the secondary, our secondaries would not be able to see it; and therefore, it will be returning stale results to the user. This

staleness might be acceptable in our system as it would be very short-lived, and the user would be able to see the new videos after a few milliseconds.

Where would thumbnails be stored? There will be a lot more thumbnails than videos. If we assume that every video will have five thumbnails, we need to have a very efficient storage system that can serve huge read traffic. There will be two considerations before deciding which storage system should be used for thumbnails:

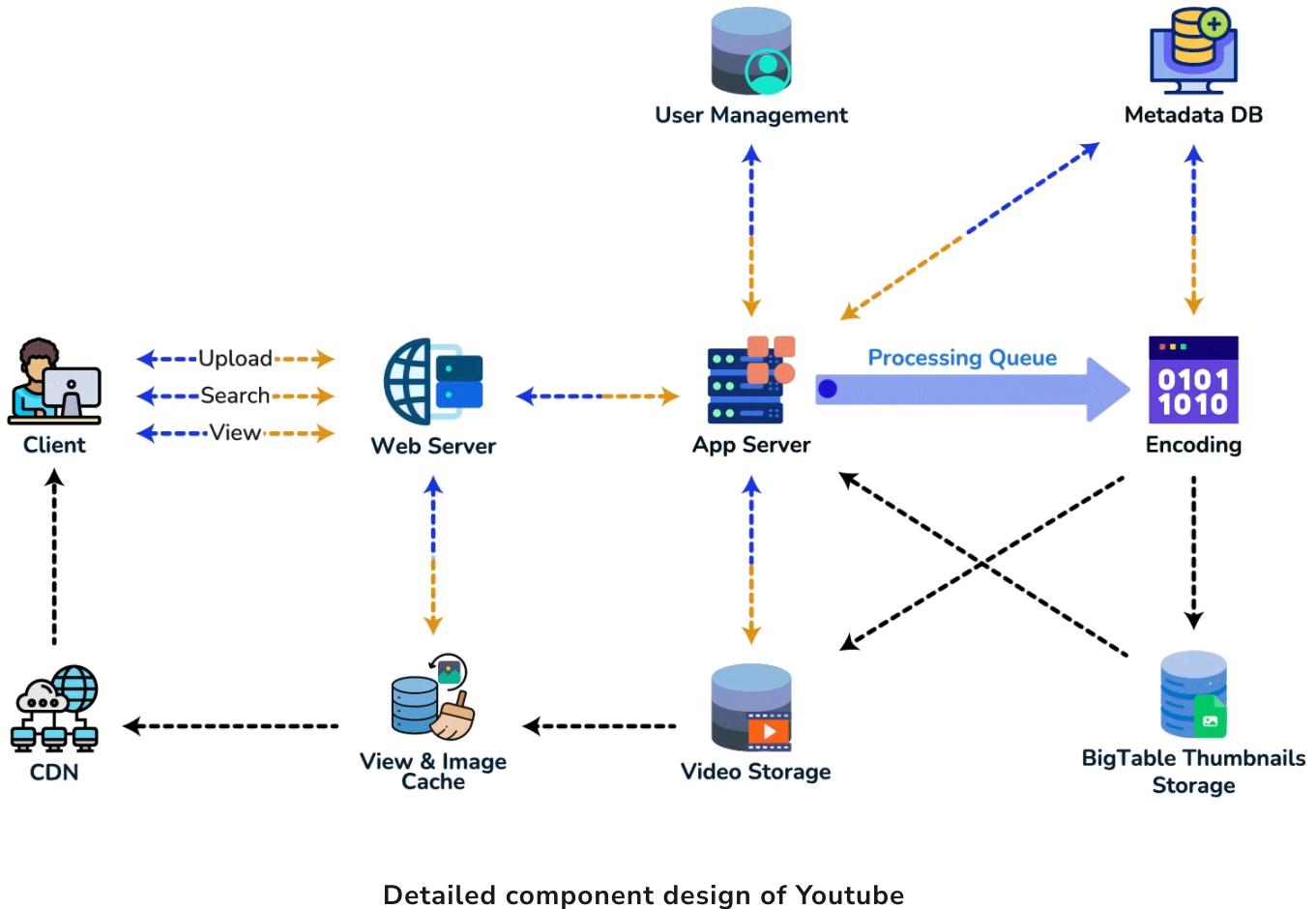
1. Thumbnails are small files, say, a maximum of 5KB each.
2. Read traffic for thumbnails will be huge compared to videos. Users will be watching one video at a time, but they might be looking at a page with 20 thumbnails of other videos.

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 multiple files into one block to store on the disk and is very efficient in reading a small amount of data. Both of these are the two most significant requirements for our service. Keeping hot thumbnails in the cache will also help improve the latencies and, given that thumbnails files are small in size, we can easily cache a large number of such files in memory.

Video Uploads: Since videos could be huge, if while uploading, the connection drops, we should support resuming from the same point.

Video Encoding: Newly uploaded videos are stored on the server, and a new task is added to the processing queue to encode the video into multiple formats. Once all the encoding is completed, the uploader will be notified, and the video is made available for view/sharing.



8. 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. While querying for videos of a user, we can ask our hash function to find

the server holding the user's data and then read it from there. To search videos by titles, we will have to query all servers, and each server will return a set of videos. A centralized server will then aggregate and rank these results before returning them to the user.

This approach has a couple of issues:

1. 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.
2. 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.

To recover from these situations, either we have to repartition/redistribute our data or used consistent hashing to balance the load between servers.

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.

We can further improve our performance by introducing a cache to store hot videos in front of the database servers.

9. Video Deduplication

With a huge number of users uploading a massive amount of video data, our service will have to deal with widespread video duplication. Duplicate videos often differ in aspect ratios or encodings, contain overlays or additional borders, or be excerpts from a longer original video. The proliferation of duplicate videos can have an impact on many levels:

1. Data Storage: We could be wasting storage space by keeping multiple copies of the same video.
2. Caching: Duplicate videos would result in degraded cache efficiency by taking up space that could be used for unique content.
3. Network usage: Duplicate videos will also increase the amount of data that must be sent over the network to in-network caching systems.
4. Energy consumption: Higher storage, inefficient cache, and network usage could result in energy wastage.

For the end-user, these inefficiencies will be realized in the form of duplicate search results, longer video startup times, and interrupted streaming.

For our service, deduplication makes most sense early; when a user is uploading a video as compared to post-processing it to find duplicate videos later. Inline deduplication will save us a lot of resources that can be used to encode, transfer, and store the duplicate copy of the video. As soon as any user starts uploading a video, our service can run video matching algorithms (e.g., [Block Matching](#), [Phase Correlation](#), etc.) to find duplications. If we already have a copy of the video being uploaded, we can either stop the upload and use the existing copy or continue the upload and use the newly uploaded video if it is of higher quality. If the newly uploaded video is a subpart of an existing video or vice versa, we can intelligently divide the video into smaller chunks so that we only upload the parts that are missing.

10. 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 an additional HTTP request; it also leads to higher delays before the video starts playing back. Moreover, inter-tier (or cross data-center) redirections lead a client to a distant cache location because the higher tier caches are only present at a small number of locations.

11. 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 reasonable cache eviction policy for our system. Under this policy, we discard the least recently viewed row first.

How can we build a more intelligent cache? If we go with the 80-20 rule, i.e., 20% of daily read volume for videos is generating 80% of traffic, meaning that certain

videos are so popular that the majority of people view them; it follows that we can try caching 20% of daily read volume of videos and metadata.

12. Content Delivery Network (CDN)

A CDN is a system of distributed servers that deliver web content to a user based on the user's geographic locations, the origin of the web page, and a content delivery server. Take a look at the 'CDN' section in 'Caching' chapter.

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.

13. Fault Tolerance

We should use 'Consistent Hashing' for distribution among database servers.

Consistent hashing will not only help in replacing a dead server but also help in distributing load among servers.

Designing Typeahead Suggestion

Let's design a real-time suggestion service, which will recommend terms to users as they enter text for searching.

Similar Services: Auto-suggestions, Typeahead search

Difficulty: Medium

1. What is Typeahead Suggestion?

Typeahead suggestions enable users to search for known and frequently searched terms. As the user types into the search box, it tries to predict the query based on the characters the user has entered and gives a list of suggestions to complete the query. Typeahead suggestions help the user to articulate their search queries better. It's not about speeding up the search process but rather about guiding the users and lending them a helping hand in constructing their search query.

Designing Typeahead Suggestion (video)

Here is a video discussing how to design Typeahead Suggestion:

1:11:33

Designing Typeahead Suggestion

2. Requirements and Goals of the System

Functional Requirements: As the user types in their query, our service should suggest top 10 terms starting with whatever the user has typed.

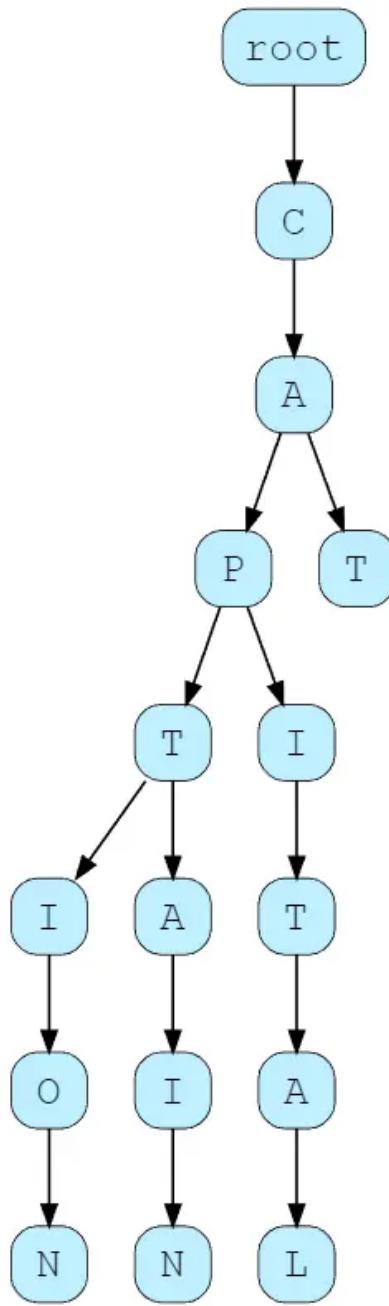
Non-function Requirements: The suggestions should appear in real-time. The user should be able to see the suggestions within 200ms.

3. Basic System Design and Algorithm

The problem we are trying to solve is that we have a lot of 'strings' that we need to store in such a way that users can search with any prefix. Our service will suggest the next terms matching the given prefix. For example, if our database contains the following terms: cap, cat, captain, or capital, and the user has typed in 'cap', our system should suggest 'cap', 'captain' and 'capital'.

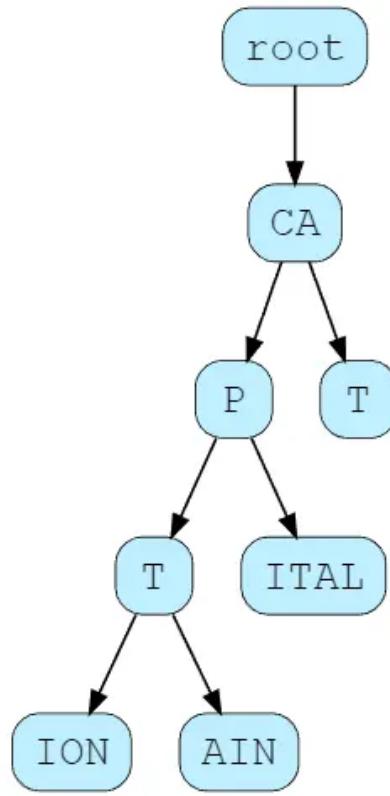
As we have to serve a lot of queries with minimum latency, we need to come up with a scheme that can efficiently store our data such that it can be queried quickly. We can't depend upon some database for this; we need to store our index in memory in a highly efficient data structure.

One of the most appropriate data structures that can serve our purpose is the Trie (pronounced “try”). A trie is a tree-like data structure used to store phrases where each node stores a character of the phrase in a sequential manner. For example, if we need to store 'cap, cat, caption, captain, capital' in the trie, it would look like:



Now if the user has typed 'cap', our service can traverse the trie to go to the node 'P' to find all the terms that start with this prefix (e.g., cap-tion, cap-it-al etc).

We can merge nodes that have only one branch to save storage space. The above trie can be stored like this:



Should we have case insensitive trie? For simplicity and search use-case, let's assume our data is case insensitive.

How to find top suggestion? Now that we can find all the terms for a given prefix, how can we find the top 10 terms for the given prefix? One simple solution could be to store the count of searches that terminated at each node, e.g., if users have searched about 'CAPTAIN' 100 times and 'CAPTION' 500 times, we can store this number with the last character of the phrase. Now if the user types 'CAP' we know the top most searched word under the prefix 'CAP' is 'CAPTION'. So, to find the top suggestions for a given prefix, we can traverse the sub-tree under it.

Given a prefix, how much time will it take to traverse its sub-tree? Given the amount of data we need to index, we should expect a huge tree. Even traversing a sub-tree would take really long, e.g., the phrase 'system design interview questions' is 30 levels deep. Since we have very strict latency requirements we do need to improve the efficiency of our solution.

Can we store top suggestions with each node? This can surely speed up our searches but will require a lot of extra storage. We can store top 10 suggestions at each node that we can return to the user. We have to bear the big increase in our storage capacity to achieve the required efficiency.

We can optimize our storage by storing only references of the terminal nodes rather than storing the entire phrase. To find the suggested terms we need to traverse back using the parent reference from the terminal node. We will also need to store the frequency with each reference to keep track of top suggestions.

How would we build this trie? We can efficiently build our trie bottom up. Each parent node will recursively call all the child nodes to calculate their top suggestions and their counts. Parent nodes will combine top suggestions from all of their children to determine their top suggestions.

How to update the trie? Assuming five billion searches every day, which would give us approximately 60K queries per second. If we try to update our trie for every query it'll be extremely resource intensive and this can hamper our read requests, too. One solution to handle this could be to update our trie offline after a certain interval.

As the new queries come in we can log them and also track their frequencies. Either we can log every query or do sampling and log every 1000th query. For example, if we don't want to show a term which is searched for less than 1000 times, it's safe to log every 1000th searched term.

We can have a [Map-Reduce \(MR\)](#) set-up to process all the logging data periodically say every hour. These MR jobs will calculate frequencies of all searched terms in the past hour. We can then update our trie with this new data. We can take the current snapshot of the trie and update it with all the new terms and their frequencies. We should do this offline as we don't want our read queries to be blocked by update trie requests. We can have two options:

1. We can make a copy of the trie on each server to update it offline. Once done we can switch to start using it and discard the old one.
2. Another option is we can have a primary-secondary configuration for each trie server. We can update the secondary while the primary is serving traffic. Once the update is complete, we can make the secondary our new primary. We can later update our old primary, which can then start serving traffic, too.

How can we update the frequencies of typeahead suggestions? Since we are storing frequencies of our typeahead suggestions with each node, we need to update them too! We can update only differences in frequencies rather than recounting all search terms from scratch. If we're keeping count of all the terms searched in the last 10 days, we'll need to subtract the counts from the time period no longer included and add the counts for the new time period being included. We can add and subtract frequencies based on [Exponential Moving Average \(EMA\)](#) of each term. In EMA, we give more weight to the latest data. It's also known as the exponentially weighted moving average.

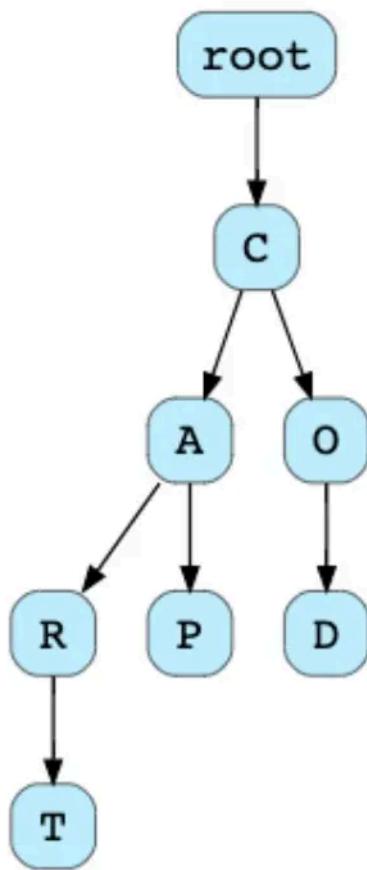
After inserting a new term in the trie, we'll go to the terminal node of the phrase and increase its frequency. Since we're storing the top 10 queries in each node, it is possible that this particular search term jumped into the top 10 queries of a few other nodes. So, we need to update the top 10 queries of those nodes then. We have to traverse back from the node to all the way up to the root. For every parent, we check if the current query is part of the top 10. If so, we update the corresponding frequency. If not, we check if the current query's frequency is high enough to be a part of the top 10. If so, we insert this new term and remove the term with the lowest frequency.

How can we remove a term from the trie? Let's say we have to remove a term from the trie because of some legal issue or hate or piracy etc. We can completely remove such terms from the trie when the regular update happens, meanwhile, we can add a filtering layer on each server which will remove any such term before sending them to users.

What could be different ranking criteria for suggestions? In addition to a simple count, for terms ranking, we have to consider other factors too, e.g., freshness, user location, language, demographics, personal history, etc.

4. Permanent Storage of the Trie

How to store trie in a file so that we can rebuild our trie easily - this will be needed when a machine restarts? We can take a snapshot of our trie periodically and store it in a file. This will enable us to rebuild a trie if the server goes down. To store, we can start with the root node and save the trie level-by-level. With each node, we can store what character it contains and how many children it has. Right after each node, we should put all of its children. Let's assume we have the following trie:



If we store this trie in a file with the above-mentioned scheme, we will have: "C2,A2,R1,T,P,O1,D". From this, we can easily rebuild our trie.

If you've noticed, we are not storing top suggestions and their counts with each node. It is hard to store this information; as our trie is being stored top-down, we don't have child nodes created before the parent, so there is no easy way to store their references. For this, we have to recalculate all the top terms with counts. This can be done while we are building the trie. Each node will calculate its top suggestions and pass it to its parent. Each parent node will merge results from all of its children to figure out its top suggestions.

5. Scale Estimation

If we are building a service that has the same scale as that of Google we can expect 5 billion searches every day, which would give us approximately 60K queries per second.

Since there will be a lot of duplicates in 5 billion queries, we can assume that only 20% of these will be unique. If we only want to index the top 50% of the search terms, we can get rid of a lot of less frequently searched queries. Let's assume we will have 100 million unique terms for which we want to build an index.

Storage Estimation: If on the average each query consists of 3 words and if the average length of a word is 5 characters, this will give us 15 characters of average query size. Assuming we need 2 bytes to store a character, we will need 30 bytes to store an average query. So total storage we will need:

$$100 \text{ million} * 30 \text{ bytes} \Rightarrow 3 \text{ GB}$$

We can expect some growth in this data every day, but we should also be removing some terms that are not searched anymore. If we assume we have 2% new queries every day and if we are maintaining our index for the last one year, total storage we should expect:

$$3\text{GB} + (0.02 * 3 \text{ GB} * 365 \text{ days}) \Rightarrow 25 \text{ GB}$$

6. Data Partition

Although our index can easily fit on one server, we can still partition it in order to meet our requirements of higher efficiency and lower latencies. How can we efficiently partition our data to distribute it onto multiple servers?

a. Range Based Partitioning: What if we store our phrases in separate partitions based on their first letter. So we save all the terms starting with the letter 'A' in one partition and those that start with the letter 'B' into another partition and so on. We can even combine certain less frequently occurring letters into one partition. We should come up with this partitioning scheme statically so that we can always store and search terms in a predictable manner.

The main problem with this approach is that it can lead to unbalanced servers, for instance, if we decide to put all terms starting with the letter 'E' into one partition, but later we realize that we have too many terms that start with letter 'E' that we can't fit into one partition.

We can see that the above problem will happen with every statically defined scheme. It is not possible to calculate if each of our partitions will fit on one server statically.

b. Partition based on the maximum capacity of the server: Let's say we partition our trie based on the maximum memory capacity of the servers. We can keep storing data on a server as long as it has memory available. Whenever a sub-tree cannot fit into a server, we break our partition there to assign that range to this server and move on to the next server to repeat this process. Let's say if our first trie server can store all terms from 'A' to 'AABC', which mean our next server will store from 'AABD' onwards. If our second server could store up to 'BXA', the next server will start from 'BXB', and so on. We can keep a hash table to quickly access this partitioning scheme:

Server 1, A-AABC

Server 2, AABD-BXA

Server 3, BXB-CDA

For querying, if the user has typed 'A' we have to query both servers 1 and 2 to find the top suggestions. When the user has typed 'AA', we still have to query server 1 and 2, but when the user has typed 'AAA' we only need to query server 1.

We can have a load balancer in front of our trie servers which can store this mapping and redirect traffic. Also, if we are querying from multiple servers, either we need to merge the results on the server-side to calculate the overall top results or make our clients do that. If we prefer to do this on the server-side, we need to introduce another layer of servers between load balancers and trie servers (let's call them aggregator). These servers will aggregate results from multiple trie servers and return the top results to the client.

Partitioning based on the maximum capacity can still lead us to hotspots, e.g., if there are a lot of queries for terms starting with 'cap', the server holding it will have a high load compared to others.

c. Partition based on the hash of the term: Each term will be passed to a hash function, which will generate a server number and we will store the term on that server. This will make our term distribution random and hence minimize hotspots. The disadvantage of this scheme is, to find typeahead suggestions for a term we have to ask all the servers and then aggregate the results.

7. Cache

We should realize that caching the top searched terms will be extremely helpful in our service. There will be a small percentage of queries that will be responsible for most of the traffic. We can have separate cache servers in front of the trie servers

holding the most frequently searched terms and their typeahead suggestions. Application servers should check these cache servers before hitting the trie servers to see if they have the desired searched terms. This will save us time to traverse the trie.

We can also build a simple Machine Learning (ML) model that can try to predict the engagement on each suggestion based on simple counting, personalization, or trending data, and cache these terms beforehand.

8. Replication and Load Balancer

We should have replicas for our trie servers both for load balancing and also for fault tolerance. We also need a load balancer that keeps track of our data partitioning scheme and redirects traffic based on the prefixes.

9. Fault Tolerance

What will happen when a trie server goes down? As discussed above we can have a primary-secondary configuration; if the primary dies, the secondary can take over after failover. Any server that comes back up, can rebuild the trie based on the last snapshot.

10. Typeahead Client

We can perform the following optimizations on the client-side to improve user's experience:

1. The client should only try hitting the server if the user has not pressed any key for 50ms.

2. If the user is constantly typing, the client can cancel the in-progress requests.
3. Initially, the client can wait until the user enters a couple of characters.
4. Clients can pre-fetch some data from the server to save future requests.
5. Clients can store the recent history of suggestions locally. Recent history has a very high rate of being reused.
6. Establishing an early connection with the server turns out to be one of the most important factors. As soon as the user opens the search engine website, the client can open a connection with the server. So when a user types in the first character, the client doesn't waste time in establishing the connection.
7. The server can push some part of their cache to CDNs and Internet Service Providers (ISPs) for efficiency.

11. Personalization

Users will receive some typeahead suggestions based on their historical searches, location, language, etc. We can store the personal history of each user separately on the server and also cache them on the client. The server can add these personalized terms in the final set before sending it to the user. Personalized searches should always come before others.

Designing an API Rate Limiter

Let's design an API Rate Limiter which will throttle users based upon the number of the requests they are sending.

Difficulty Level: Medium

1. What is a Rate Limiter?

Imagine we have a service which is receiving a huge number of requests, but it can only serve a limited number of requests per second. To handle this problem we would need some kind of throttling or rate limiting mechanism that would allow only a certain number of requests so our service can respond to all of them. A rate limiter, at a high-level, limits the number of events an entity (user, device, IP, etc.) can perform in a particular time window. For example:

- A user can send only one message per second.
- A user is allowed only three failed credit card transactions per day.
- A single IP can only create twenty accounts per day.

In general, a rate limiter caps how many requests a sender can issue in a specific time window. It then blocks requests once the cap is reached.

2. Why do we need API rate limiting?

Rate Limiting helps to protect services against abusive behaviors targeting the application layer like **Denial-of-service (DOS)** attacks, brute-force password attempts, brute-force credit card transactions, etc. These attacks are usually a

barrage of HTTP/S requests which may look like they are coming from real users, but are typically generated by machines (or bots). As a result, these attacks are often harder to detect and can more easily bring down a service, application, or an API.

Rate limiting is also used to prevent revenue loss, to reduce infrastructure costs, to stop spam, and to stop online harassment. Following is a list of scenarios that can benefit from Rate limiting by making a service (or API) more reliable:

- **Misbehaving clients/scripts:** Either intentionally or unintentionally, some entities can overwhelm a service by sending a large number of requests. Another scenario could be when a user is sending a lot of lower-priority requests and we want to make sure that it doesn't affect the high-priority traffic. For example, users sending a high volume of requests for analytics data should not be allowed to hamper critical transactions for other users.
- **Security:** By limiting the number of the second-factor attempts (in 2-factor auth) that the users are allowed to perform, for example, the number of times they're allowed to try with a wrong password.
- **To prevent abusive behavior and bad design practices:** Without API limits, developers of client applications would use sloppy development tactics, for example, requesting the same information over and over again.
- **To keep costs and resource usage under control:** Services are generally designed for normal input behavior, for example, a user writing a single post in a minute. Computers could easily push thousands/second through an API. Rate limiter enables controls on service APIs.
- **Revenue:** Certain services might want to limit operations based on the tier of their customer's service and thus create a revenue model based on rate limiting. There could be default limits for all the APIs a service offers. To go beyond that, the user has to buy higher limits
- **To eliminate spikiness in traffic:** Make sure the service stays up for everyone else.

3. Requirements and Goals of the System

Our Rate Limiter should meet the following requirements:

Functional Requirements:

1. Limit the number of requests an entity can send to an API within a time window, e.g., 15 requests per second.
2. The APIs are accessible through a cluster, so the rate limit should be considered across different servers. The user should get an error message whenever the defined threshold is crossed within a single server or across a combination of servers.

Non-Functional Requirements:

1. The system should be highly available. The rate limiter should always work since it protects our service from external attacks.
2. Our rate limiter should not introduce substantial latencies affecting the user experience.

4. How to do Rate Limiting?

Rate Limiting is a process that is used to define the rate and speed at which consumers can access APIs. **Throttling** is the process of controlling the usage of the APIs by customers during a given period. Throttling can be defined at the application level and/or API level. When a throttle limit is crossed, the server returns HTTP status “429 - Too many requests”.

5. What are different types of throttling?

Here are the three famous throttling types that are used by different services:

Hard Throttling: The number of API requests cannot exceed the throttle limit.

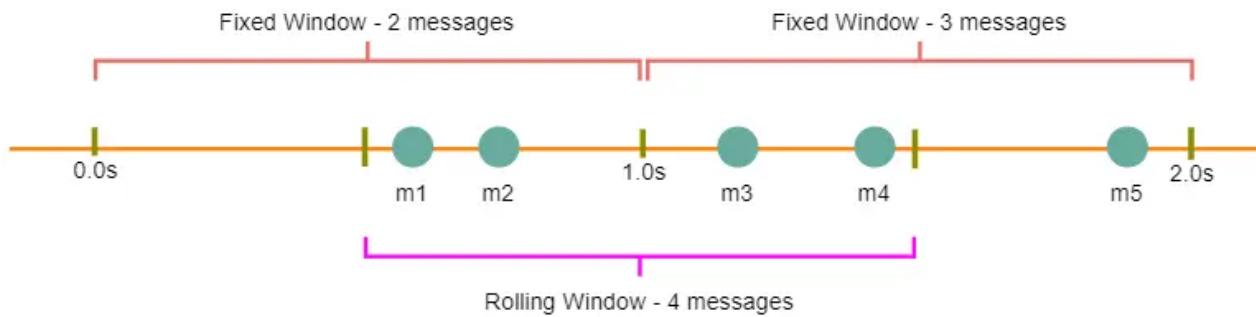
Soft Throttling: In this type, we can set the API request limit to exceed a certain percentage. For example, if we have rate-limit of 100 messages a minute and 10% exceed-limit, our rate limiter will allow up to 110 messages per minute.

Elastic or Dynamic Throttling: Under Elastic throttling, the number of requests can go beyond the threshold if the system has some resources available. For example, if a user is allowed only 100 messages a minute, we can let the user send more than 100 messages a minute when there are free resources available in the system.

6. What are different types of algorithms used for Rate Limiting?

Following are the two types of algorithms used for Rate Limiting:

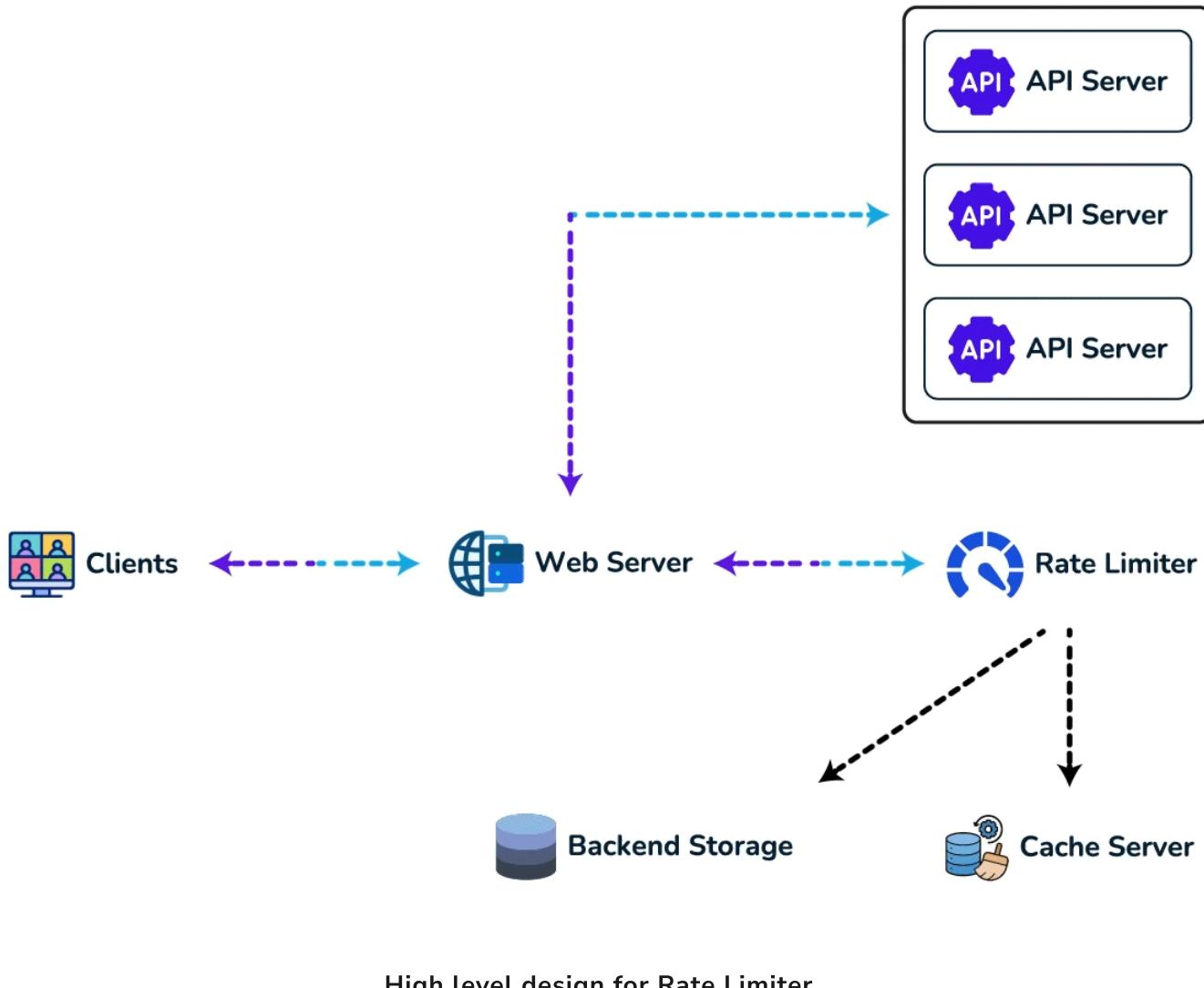
Fixed Window Algorithm: In this algorithm, the time window is considered from the start of the time-unit to the end of the time-unit. For example, a period would be considered 0-60 seconds for a minute irrespective of the time frame at which the API request has been made. In the diagram below, there are two messages between 0-1 second and three messages between 1-2 seconds. If we have a rate limiting of two messages a second, this algorithm will throttle only 'm5'.



Rolling Window Algorithm: In this algorithm, the time window is considered from the fraction of the time at which the request is made plus the time window length. For example, if there are two messages sent at the 300th millisecond and 400th millisecond of a second, we'll count them as two messages from the 300th millisecond of that second up to the 300th millisecond of next second. In the above diagram, keeping two messages a second, we'll throttle 'm3' and 'm4'.

7. High level design for Rate Limiter

Rate Limiter will be responsible for deciding which request will be served by the API servers and which request will be declined. Once a new request arrives, the Web Server first asks the Rate Limiter to decide if it will be served or throttled. If the request is not throttled, then it'll be passed to the API servers.



8. Basic System Design and Algorithm

Let's take the example where we want to limit the number of requests per user. Under this scenario, for each unique user, we would keep a count representing how many requests the user has made and a timestamp when we started counting the requests. We can keep it in a hashtable, where the 'key' would be the 'UserID' and 'value' would be a structure containing an integer for the 'Count' and an integer for the Epoch time:

Key : Value

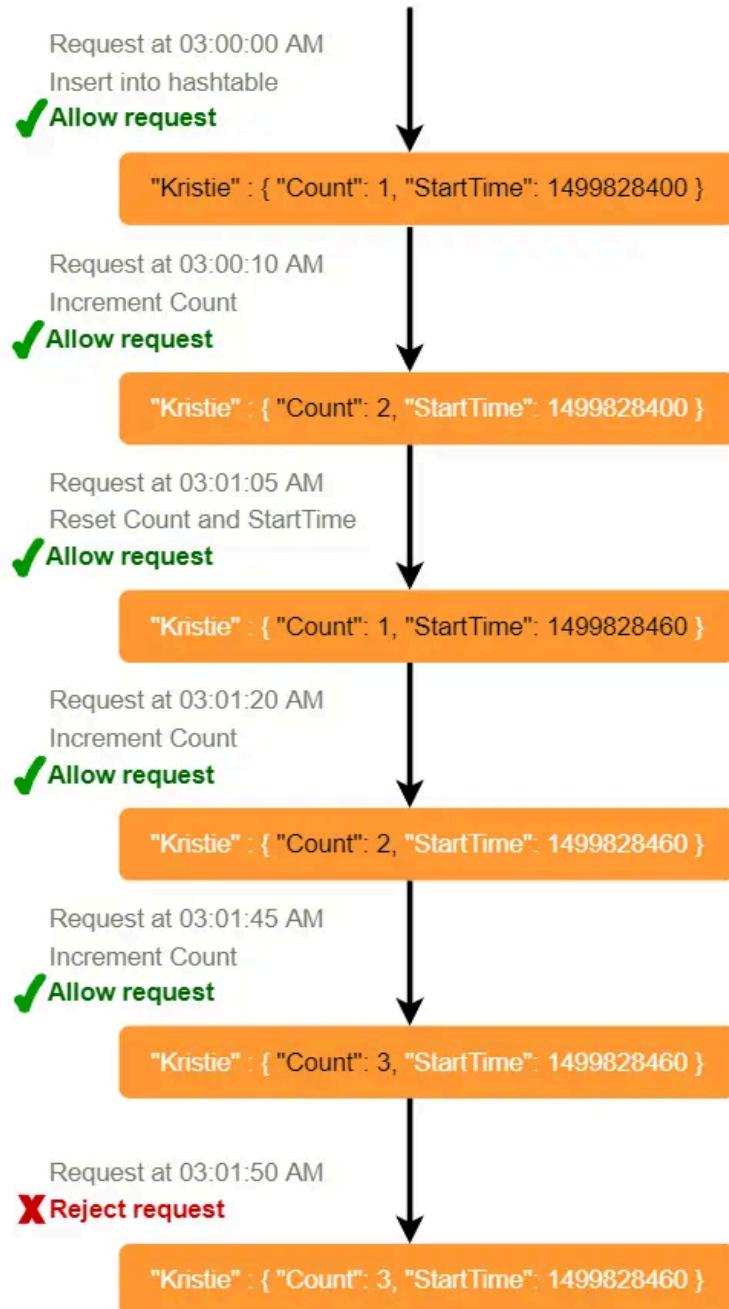
E.g., **UserID : { Count, StartTime }**

Kristie : { 3, 1499818564 }

Let's assume our rate limiter is allowing three requests per minute per user, so whenever a new request comes in, our rate limiter will perform the following steps:

1. If the 'UserID' is not present in the hash-table, insert it, set the 'Count' to 1, set 'StartTime' to the current time (normalized to a minute), and allow the request.
2. Otherwise, find the record of the 'UserID' and if
 $\text{CurrentTime} - \text{StartTime} \geq 1 \text{ min}$, set the 'StartTime' to the current time, 'Count' to 1, and allow the request.
3. If $\text{CurrentTime} - \text{StartTime} \leq 1 \text{ min}$ and
 - If 'Count < 3', increment the Count and allow the request.
 - If 'Count \geq 3', reject the request.

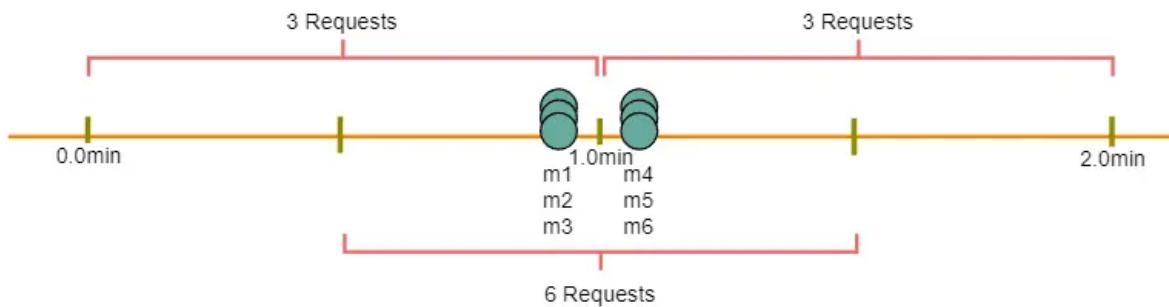
Rate Limiter allowing three requests per minute for user "Kristie"



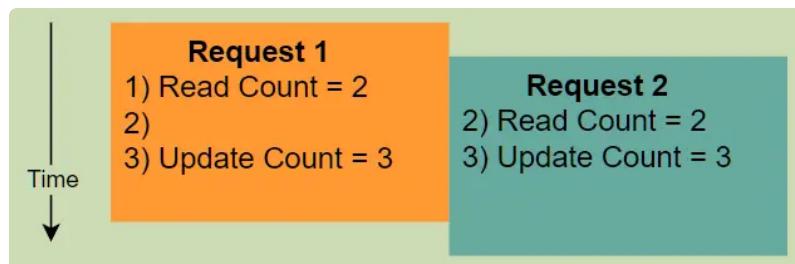
What are some of the problems with our algorithm?

1. This is a **Fixed Window** algorithm since we're resetting the 'StartTime' at the end of every minute, which means it can potentially allow twice the number of requests per minute. Imagine if Kristie sends three requests at the last second of a minute, then she can immediately send three more requests at the very first second of the next minute, resulting in 6 requests in the span of two minutes.

seconds. The solution to this problem would be a sliding window algorithm which we'll discuss later.



2. Atomicity: In a distributed environment, the “read-and-then-write” behavior can create a race condition. Imagine if Kristie's current 'Count' is "2" and that she issues two more requests. If two separate processes served each of these requests and concurrently read the Count before either of them updated it, each process would think that Kristie could have one more request and that she had not hit the rate limit.



If we are using [Redis](#) to store our key-value, one solution to resolve the atomicity problem is to use [Redis lock](#) for the duration of the read-update operation. This, however, would come at the expense of slowing down concurrent requests from the same user and introducing another layer of complexity. We can use [Memcached](#), but it would have comparable complications.

If we are using a simple hash-table, we can have a custom implementation for 'locking' each record to solve our atomicity problems.

How much memory would we need to store all of the user data? Let's assume the simple solution where we are keeping all of the data in a hash-table.

Let's assume 'UserID' takes 8 bytes. Let's also assume a 2 byte 'Count', which can count up to 65k, is sufficient for our use case. Although epoch time will need 4 bytes, we can choose to store only the minute and second part, which can fit into 2 bytes. Hence, we need a total of 12 bytes to store a user's data:

$$8 + 2 + 2 = 12 \text{ bytes}$$

Let's assume our hash-table has an overhead of 20 bytes for each record. If we need to track one million users at any time, the total memory we would need would be 32MB:

$$(12 + 20) \text{ bytes} * 1 \text{ million} \Rightarrow 32\text{MB}$$

If we assume that we would need a 4-byte number to lock each user's record to resolve our atomicity problems, we would require a total 36MB memory.

This can easily fit on a single server; however we would not like to route all of our traffic through a single machine. Also, if we assume a rate limit of 10 requests per second, this would translate into 10 million QPS for our rate limiter! This would be too much for a single server. Practically, we can assume we would use a Redis or Memcached kind of a solution in a distributed setup. We'll be storing all the data in the remote Redis servers and all the Rate Limiter servers will read (and update) these servers before serving or throttling any request.

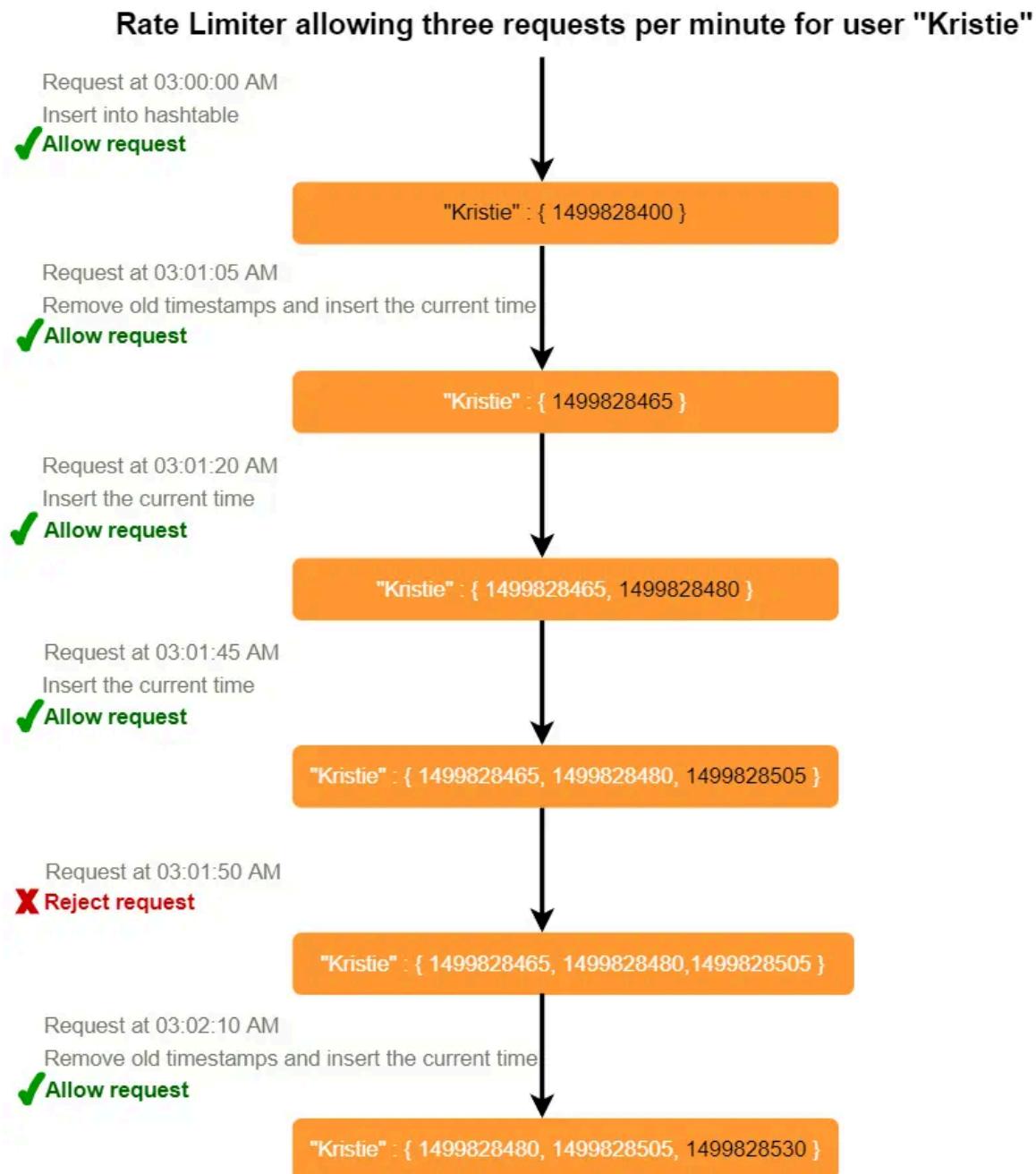
9. Sliding Window algorithm

We can maintain a sliding window if we can keep track of each request per user. We can store the timestamp of each request in a Redis **Sorted Set** in our 'value' field of hash-table.

Key :	Value
E.g.,	UserID : { Sorted Set <UnixTime> }
	Kristie : { 1499818000, 1499818500, 1499818860 }

Let's assume our rate limiter is allowing three requests per minute per user, so, whenever a new request comes in, the Rate Limiter will perform following steps:

1. Remove all the timestamps from the Sorted Set that are older than "CurrentTime - 1 minute".
2. Count the total number of elements in the sorted set. Reject the request if this count is greater than our throttling limit of "3".
3. Insert the current time in the sorted set and accept the request.



How much memory would we need to store all of the user data for sliding window? Let's assume 'UserID' takes 8 bytes. Each epoch time will require 4 bytes. Let's suppose we need a rate limiting of 500 requests per hour. Let's assume 20 bytes overhead for hash-table and 20 bytes overhead for the Sorted Set. At max, we would need a total of 12KB to store one user's data:

$$8 + (4 + 20 \text{ (sorted set overhead)}) * 500 + 20 \text{ (hash-table overhead)} = 12\text{KB}$$

Here we are reserving 20 bytes overhead per element. In a sorted set, we can assume that we need at least two pointers to maintain order among elements – one pointer to the previous element and one to the next element. On a 64bit machine, each pointer will cost 8 bytes. So we will need 16 bytes for pointers. We added an extra word (4 bytes) for storing other overhead.

If we need to track one million users at any time, total memory we would need would be 12GB:

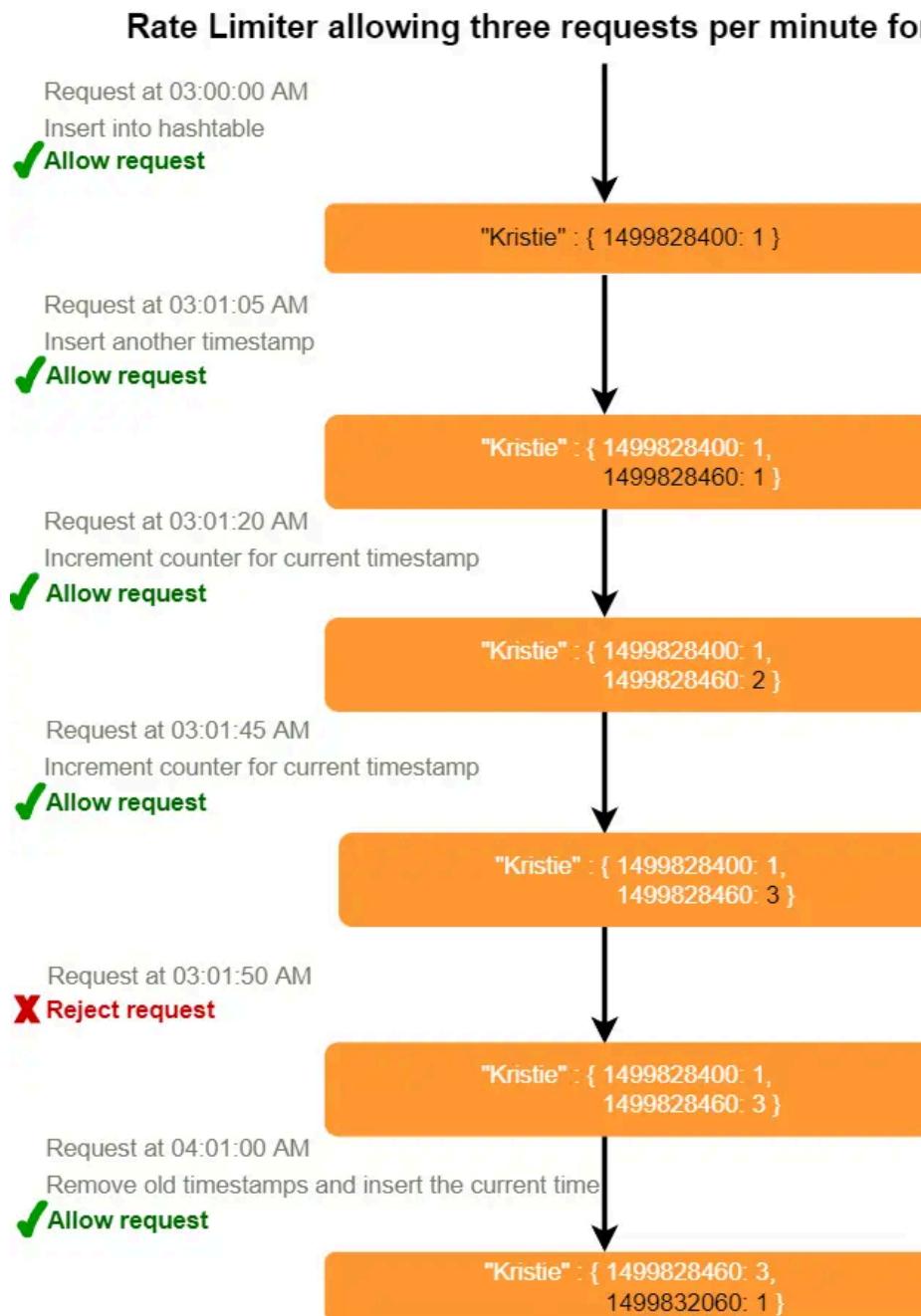
$$12\text{KB} * 1 \text{ million} \approx 12\text{GB}$$

Sliding Window Algorithm takes a lot of memory compared to the Fixed Window; this would be a scalability issue. What if we can combine the above two algorithms to optimize our memory usage?

10. Sliding Window with Counters

What if we keep track of request counts for each user using multiple fixed time windows, e.g., 1/60th the size of our rate limit's time window. For example, if we have an hourly rate limit we can keep a count for each minute and calculate the sum of all counters in the past hour when we receive a new request to calculate the throttling limit. This would reduce our memory footprint. Let's take an example where we rate-limit at 500 requests per hour with an additional limit of 10 requests per minute. This means that when the sum of the counters with timestamps in the past hour exceeds the request threshold (500), Kristie has exceeded the rate limit. In addition to that, she can't send more than ten requests per minute. This would be a reasonable and practical consideration, as none of the real users would send frequent requests. Even if they do, they will see success with retries since their limits get reset every minute.

We can store our counters in a [Redis Hash](#) since it offers incredibly efficient storage for fewer than 100 keys. When each request increments a counter in the hash, it also sets the hash to [expire](#) an hour later. We will normalize each 'time' to a minute.



How much memory we would need to store all the user data for sliding window with counters? Let's assume 'UserID' takes 8 bytes. Each epoch time will need 4 bytes, and the Counter would need 2 bytes. Let's suppose we need a rate limiting of 500

requests per hour. Assume 20 bytes overhead for hash-table and 20 bytes for Redis hash. Since we'll keep a count for each minute, at max, we would need 60 entries for each user. We would need a total of 1.6KB to store one user's data:

$$8 + (4 + 2 + 20 \text{ (Redis hash overhead)}) * 60 + 20 \text{ (hash-table overhead)} = 1.6\text{KB}$$

If we need to track one million users at any time, total memory we would need would be 1.6GB:

$$1.6\text{KB} * 1 \text{ million} \approx 1.6\text{GB}$$

So, our 'Sliding Window with Counters' algorithm uses 86% less memory than the simple sliding window algorithm.

11. Data Sharding and Caching

We can shard based on the 'UserID' to distribute the user's data. For fault tolerance and replication we should use 'Consistent Hashing'. If we want to have different throttling limits for different APIs, we can choose to shard per user per API. Take the example of 'URL Shortener'; we can have different rate limiter for `createUrl()` and `deleteURL()` APIs for each user or IP.

If our APIs are partitioned, a practical consideration could be to have a separate (somewhat smaller) rate limiter for each API shard as well. Let's take the example of our URL Shortener where we want to limit each user not to create more than 100 short URLs per hour. Assuming we are using **Hash-Based Partitioning** for our `createUrl()` API, we can rate limit each partition to allow a user to create not more than three short URLs per minute in addition to 100 short URLs per hour.

Our system can get huge benefits from caching recent active users. Application servers can quickly check if the cache has the desired record before hitting backend servers. Our rate limiter can significantly benefit from the **Write-back**

cache by updating all counters and timestamps in cache only. The write to the permanent storage can be done at fixed intervals. This way we can ensure minimum latency added to the user's requests by the rate limiter. The reads can always hit the cache first; which will be extremely useful once the user has hit their maximum limit and the rate limiter will only be reading data without any updates.

Least Recently Used (LRU) can be a reasonable cache eviction policy for our system.

12. Should we rate limit by IP or by user?

Let's discuss the pros and cons of using each one of these schemes:

IP: In this scheme, we throttle requests per-IP; although it's not optimal in terms of differentiating between 'good' and 'bad' actors, it's still better than not have rate limiting at all. The biggest problem with IP based throttling is when multiple users share a single public IP like in an internet cafe or smartphone users that are using the same gateway. One bad user can cause throttling to other users. Another issue could arise while caching IP-based limits, as there are a huge number of IPv6 addresses available to a hacker from even one computer, it's trivial to make a server run out of memory tracking IPv6 addresses!

User: Rate limiting can be done on APIs after user authentication. Once authenticated, the user will be provided with a token which the user will pass with each request. This will ensure that we will rate limit against a particular API that has a valid authentication token. But what if we have to rate limit on the login API itself? The weakness of this rate-limiting would be that a hacker can perform a denial of service attack against a user by entering wrong credentials up to the limit; after that the actual user will not be able to log-in.

How about if we combine the above two schemes?

Hybrid: A right approach could be to do both per-IP and per-user rate limiting, as they both have weaknesses when implemented alone, though, this will result in more cache entries with more details per entry, hence requiring more memory and storage.

Designing Twitter Search

Twitter is one of the largest social networking service where users can share photos, news, and text-based messages. In this chapter, we will design a service that can store and search user tweets.

Similar Problems: Tweet search, Facebook status search

Difficulty Level: Medium

1. What is Twitter Search?

Twitter users can update their status whenever they like. Each status (called a tweet) consists of plain text and our goal is to design a system that allows searching over all the user tweets.

2. Requirements and Goals of the System

- Let's assume Twitter has 1.5 billion total users with 800 million daily active users.
- On average Twitter gets 400 million tweets every day.
- The average size of a tweet is 300 bytes.
- Let's assume there will be 500M searches every day.
- The search query will consist of multiple words combined with AND/OR.

We need to design a system that can efficiently store and query tweets.

3. Capacity Estimation and Constraints

Storage Capacity: Since we have 400 million new tweets every day and each tweet on average is 300 bytes, the total storage we need, will be:

$$400M * 300 \Rightarrow 120\text{GB/day}$$

Total storage per second:

$$120\text{GB} / 24\text{hours} / 3600\text{sec} \approx 1.38\text{MB/second}$$

4. System APIs

We can have SOAP or REST APIs to expose the functionality of our service; following could be the definition of the search API:

```
search(api_dev_key, search_terms, maximum_results_to_return,  
sort, page_token)
```

Parameters:

`api_dev_key` (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.

`search_terms` (string): A string containing the search terms.

`maximum_results_to_return` (number): Number of tweets to return.

`sort` (number): Optional sort mode: Latest first (0 - default), Best matched (1), Most liked (2).

`page_token` (string): This token will specify a page in the result set that should be returned.

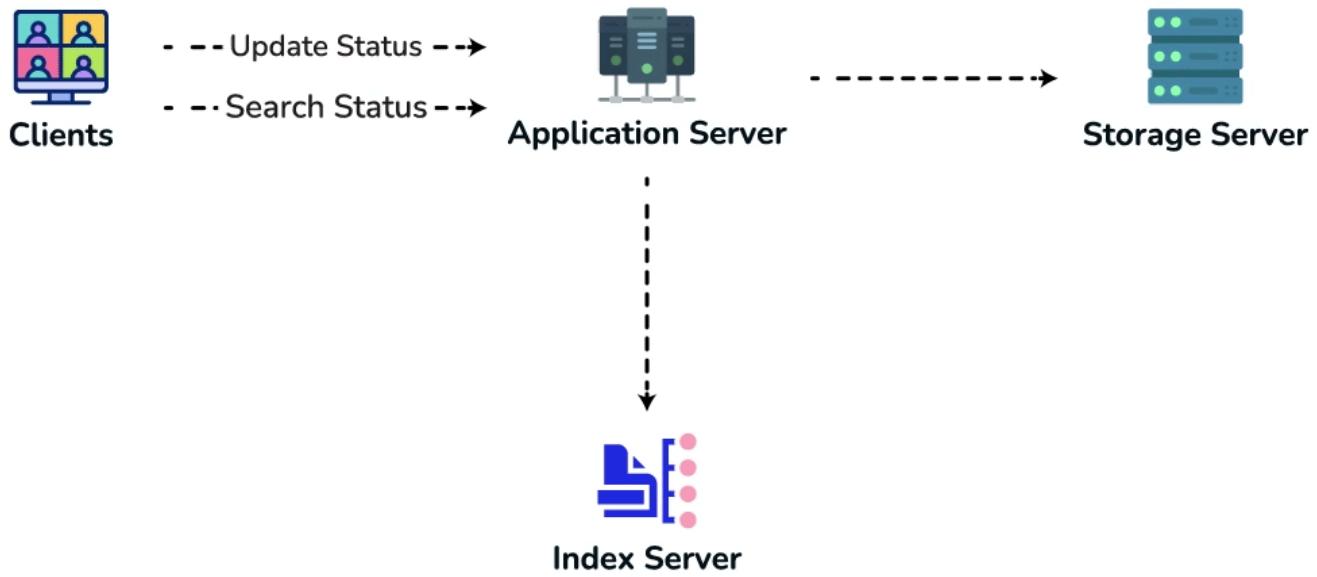
Returns: (JSON)

A JSON containing information about a list of tweets matching the search query.

Each result entry can have the user ID & name, tweet text, tweet ID, creation time, number of likes, etc.

5. High Level Design

At the high level, we need to store all the tweets in a database and also build an index that can keep track of which word appears in which tweet. This index will help us quickly find tweets that the users are trying to search for.



High level design for Twitter search

6. Detailed Component Design

1. Storage: We need to store 120GB of new data every day. Given this huge amount of data, we need to come up with a data partitioning scheme that will be efficiently distributing the data onto multiple servers. If we plan for next five years, we will need the following storage:

$$120\text{GB} * 365\text{days} * 5\text{years} \approx 200\text{TB}$$

If we never want to be more than 80% full at any time, we approximately will need 250TB of total storage. Let's assume that we want to keep an extra copy of all tweets for fault tolerance; then, our total storage requirement will be 500TB. If we assume a modern server can store up to 4TB of data, we would need 125 such servers to hold all of the required data for the next five years.

Let's start with a simplistic design where we store the tweets in a MySQL database. We can assume that we store the tweets in a table having two columns, TweetID and TweetText. Let's assume we partition our data based on TweetID. If our TweetIDs are unique system-wide, we can define a hash function that can map a TweetID to a storage server where we can store that tweet object.

How can we create system-wide unique TweetIDs? If we are getting 400M new tweets each day, then how many tweet objects we can expect in five years?

$$400\text{M} * 365 \text{ days} * 5 \text{ years} \Rightarrow 730 \text{ billion}$$

This means we would need a five bytes number to identify TweetIDs uniquely. Let's assume we have a service that can generate a unique TweetID whenever we need to store an object (The TweetID discussed here will be similar to TweetID discussed in 'Designing Twitter'). We can feed the TweetID to our hash function to find the storage server and store our tweet object there.

2. Index: What should our index look like? Since our tweet queries will consist of words, let's build the index that can tell us which word comes in which tweet object. Let's first estimate how big our index will be. If we want to build an index for all the English words and some famous nouns like people names, city names, etc., and if we assume that we have around 300K English words and 200K nouns, then we will have 500k total words in our index. Let's assume that the average length of a word is five characters. If we are keeping our index in memory, we need 2.5MB of memory to store all the words:

$$500K * 5 \Rightarrow 2.5 \text{ MB}$$

Let's assume that we want to keep the index in memory for all the tweets from only past two years. Since we will be getting 730B tweets in 5 years, this will give us 292B tweets in two years. Given that each TweetID will be 5 bytes, how much memory will we need to store all the TweetIDs?

$$292B * 5 \Rightarrow 1460 \text{ GB}$$

So our index would be like a big distributed hash table, where 'key' would be the word and 'value' will be a list of TweetIDs of all those tweets which contain that word. Assuming on average we have 40 words in each tweet and since we will not be indexing prepositions and other small words like 'the', 'an', 'and' etc., let's assume we will have around 15 words in each tweet that need to be indexed. This means each TweetID will be stored 15 times in our index. So total memory we will need to store our index:

$$(1460 * 15) + 2.5\text{MB} \approx 21 \text{ TB}$$

Assuming a high-end server has 144GB of memory, we would need 152 such servers to hold our index.

We can partition our data based on two criteria:

Sharding based on Words: While building our index, we will iterate through all the words of a tweet and calculate the hash of each word to find the server where it would be indexed. To find all tweets containing a specific word we have to query only the server which contains this word.

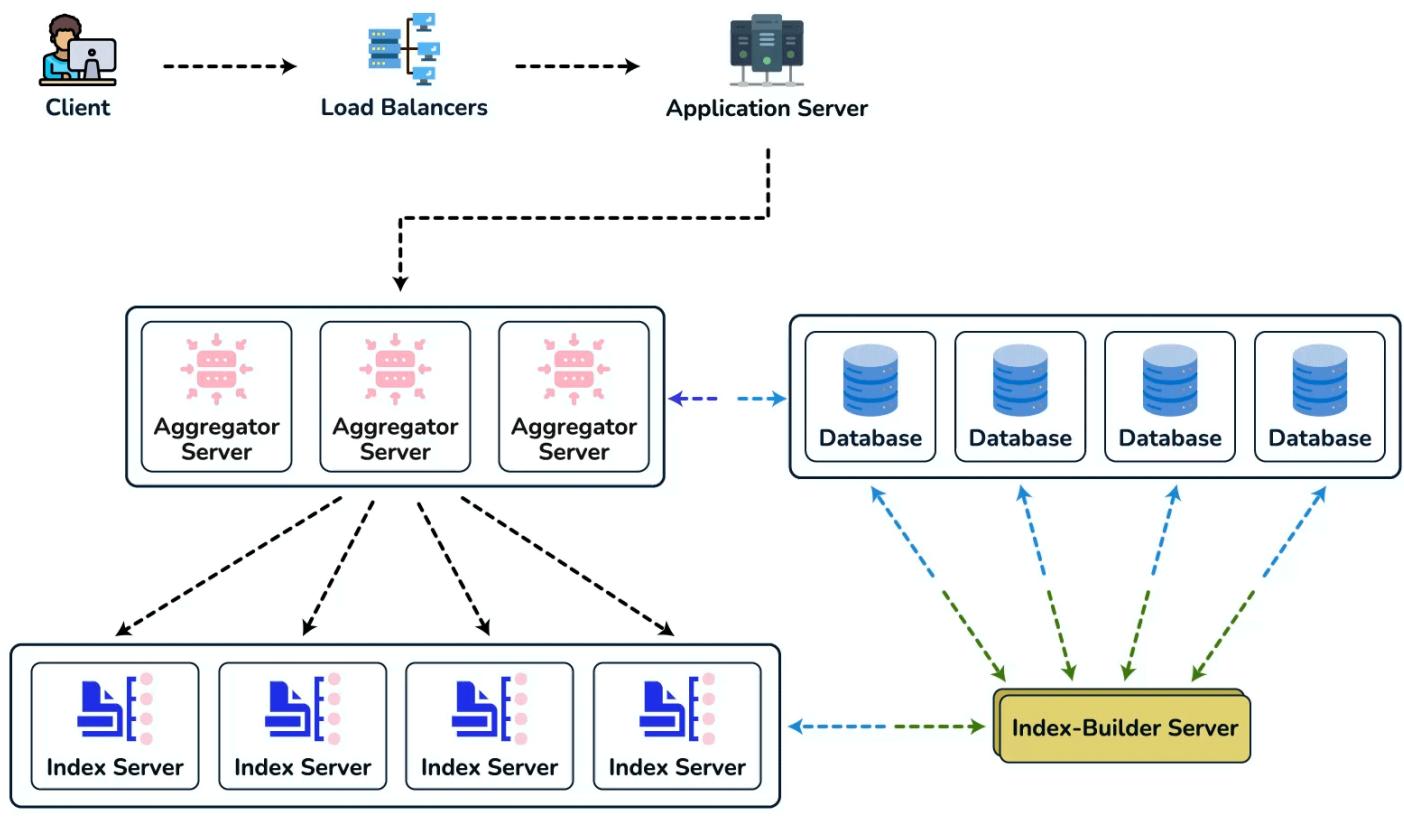
We have a couple of issues with this approach:

1. What if a word becomes hot? Then there will be a lot of queries on the server holding that word. This high load will affect the performance of our service.

2. Over time, some words can end up storing a lot of TweetIDs compared to others, therefore, maintaining a uniform distribution of words while tweets are growing is quite tricky.

To recover from these situations we either have to repartition our data or use 'Consistent Hashing'.

Sharding based on the tweet object: While storing, we will pass the TweetID to our hash function to find the server and index all the words of the tweet on that server. While querying for a particular word, we have to query all the servers, and each server will return a set of TweetIDs. A centralized server will aggregate these results to return them to the user.



Detailed component design

7. Fault Tolerance

What will happen when an index server dies? We can have a secondary replica of each server and if the primary server dies it can take control after the failover. Both primary and secondary servers will have the same copy of the index.

What if both primary and secondary servers die at the same time? We have to allocate a new server and rebuild the same index on it. How can we do that? We don't know what words/tweets were kept on this server. If we were using 'Sharding based on the tweet object', the brute-force solution would be to iterate through the whole database and filter TweetIDs using our hash function to figure out all the required tweets that would be stored on this server. This would be inefficient and also during the time when the server was being rebuilt we would not be able to serve any query from it, thus missing some tweets that should have been seen by the user.

How can we efficiently retrieve a mapping between tweets and the index server? We have to build a reverse index that will map all the TweetID to their index server. Our Index-Builder server can hold this information. We will need to build a Hashtable where the 'key' will be the index server number and the 'value' will be a HashSet containing all the TweetIDs being kept at that index server. Notice that we are keeping all the TweetIDs in a HashSet; this will enable us to add/remove tweets from our index quickly. So now, whenever an index server has to rebuild itself, it can simply ask the Index-Builder server for all the tweets it needs to store and then fetch those tweets to build the index. This approach will surely be fast. We should also have a replica of the Index-Builder server for fault tolerance.

8. Cache

To deal with hot tweets we can introduce a cache in front of our database. We can use [Memcached](#), which can store all such hot tweets in memory. Application servers, before hitting the backend database, can quickly check if the cache has that tweet. Based on clients' usage patterns, we can adjust how many cache servers we need. For cache eviction policy, Least Recently Used (LRU) seems suitable for our system.

9. Load Balancing

We can add a load balancing layer at two places in our system 1) Between Clients and Application servers and 2) Between Application servers and Backend servers. Initially, a simple Round Robin approach can be adopted; 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 LB will take dead servers out of the rotation and will stop sending any traffic to it. A problem with Round Robin LB is it won't take server load into consideration. If a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about their load and adjust traffic based on that.

10. Ranking

How about if we want to rank the search results by social graph distance, popularity, relevance, etc?

Let's assume we want to rank tweets by popularity, like how many likes or comments a tweet is getting, etc. In such a case, our ranking algorithm can calculate a 'popularity number' (based on the number of likes, etc.) and store it with the index. Each partition can sort the results based on this popularity number

before returning results to the aggregator server. The aggregator server combines all these results, sorts them based on the popularity number, and sends the top results to the user.

Designing a Web Crawler

Let's design a Web Crawler that will systematically browse and download the World Wide Web. Web crawlers are also known as web spiders, robots, worms, walkers, and bots.

Difficulty Level: Hard

1. What is a Web Crawler?

A web crawler is a software program that browses the World Wide Web in a methodical and automated manner. It collects documents by recursively fetching links from a set of starting pages. Many sites, particularly search engines, use web crawling as a means of providing up-to-date data. Search engines download all the pages to create an index on them to perform faster searches.

Some other uses of web crawlers are:

- To test web pages and links for valid syntax and structure.
- To monitor sites to see when their structure or contents change.
- To maintain mirror sites for popular Web sites.
- To search for copyright infringements.
- To build a special-purpose index, e.g., one that has some understanding of the content stored in multimedia files on the Web.

Designing Web Crawler (video)

Here is a video discussing how to design a Web Crawler:

1:07:51

Designing Web Crawler

2. Requirements and Goals of the System

Let's assume we need to crawl all the web.

Scalability: Our service needs to be scalable such that it can crawl the entire Web and can be used to fetch hundreds of millions of Web documents.

Extensibility: Our service should be designed in a modular way with the expectation that new functionality will be added to it. There could be newer document types that need to be downloaded and processed in the future.

3. Some Design Considerations

Crawling the web is a complex task, and there are many ways to go about it. We should be asking a few questions before going any further:

Is it a crawler for HTML pages only? Or should we fetch and store other types of media, such as sound files, images, videos, etc.? This is important because the answer can change the design. If we are writing a general-purpose crawler to download different media types, we might want to break down the parsing module into different sets of modules: one for HTML, another for images, or another for videos, where each module extracts what is considered interesting for that media type.

Let's assume for now that our crawler is going to deal with HTML only, but it should be extensible and make it easy to add support for new media types.

What protocols are we looking at? HTTP? What about FTP links? What different protocols should our crawler handle? For the sake of the exercise, we will assume HTTP. Again, it shouldn't be hard to extend the design to use FTP and other protocols later.

What is the expected number of pages we will crawl? How big will the URL database become? Let's assume we need to crawl one billion websites. Since a website can contain many, many URLs, let's assume an upper bound of 15 billion different web pages that will be reached by our crawler.

What is 'RobotsExclusion' and how should we deal with it? Courteous Web crawlers implement the Robots Exclusion Protocol, which allows Webmasters to declare parts of their sites off-limits to crawlers. The Robots Exclusion Protocol requires a Web crawler to fetch a special document called robot.txt which contains these declarations from a Web site before downloading any real content from it.

4. Capacity Estimation and Constraints

If we want to crawl 15 billion pages within four weeks, how many pages do we need to fetch per second?

$$15B / (4 \text{ weeks} * 7 \text{ days} * 86400 \text{ sec}) \approx 6200 \text{ pages/sec}$$

What about storage? Page sizes vary a lot, but, as mentioned above since, we will be dealing with HTML text only, let's assume an average page size of 100KB. With each page, if we are storing 500 bytes of metadata, total storage we would need:

$$15B * (100KB + 500) \approx 1.5 \text{ petabytes}$$

Assuming a 70% capacity model (we don't want to go above 70% of the total capacity of our storage system), total storage we will need:

$$1.5 \text{ petabytes} / 0.7 \approx 2.14 \text{ petabytes}$$

5. High Level Design

The basic algorithm executed by any Web crawler is to take a list of seed URLs as its input and repeatedly execute the following steps.

1. Pick a URL from the unvisited URL list.
2. Determine the IP Address of its host-name.
3. Establish a connection to the host to download the corresponding document.
4. Parse the document contents to look for new URLs.
5. Add the new URLs to the list of unvisited URLs.
6. Process the downloaded document, e.g., store it or index its contents, etc.
7. Go back to step 1

How to crawl?

Breadth-first or depth-first? Breadth First Search (BFS) is usually used. However, Depth First Search (DFS) is also utilized in some situations, such as, if your crawler has already established a connection with the website, it might just DFS all the URLs within this website to save some handshaking overhead.

Path-ascending crawling: Path-ascending crawling can help discover a lot of isolated resources or resources for which no inbound link would have been found in regular crawling of a particular Web site. In this scheme, a crawler would ascend to every path in each URL that it intends to crawl. For example, when given a seed URL of <http://foo.com/a/b/page.html>, it will attempt to crawl /a/b/, /a/, and /.

Difficulties in implementing an efficient web crawler

There are two important characteristics of the Web that makes Web crawling a very difficult task:

1. Large volume of Web pages: A large volume of web pages implies that web crawler can only download a fraction of the web pages at any time and hence it is critical that web crawler should be intelligent enough to prioritize download.

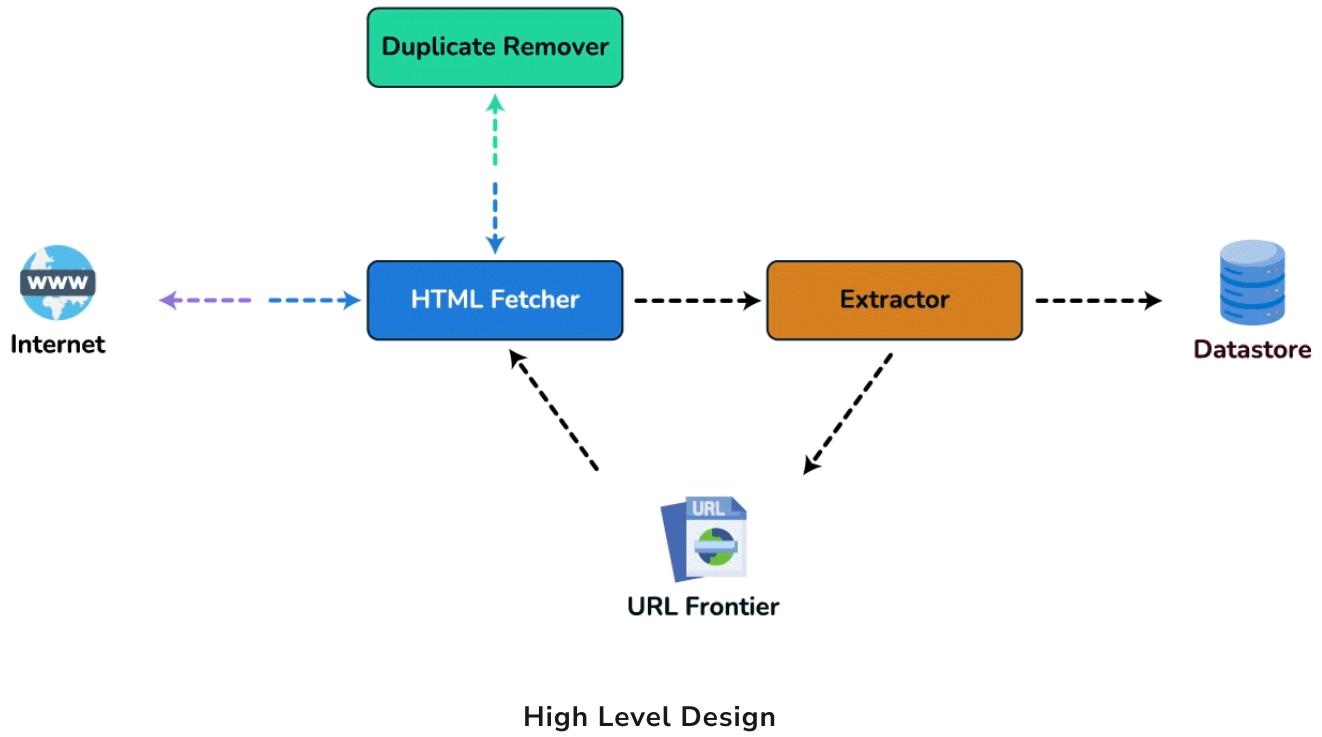
2. Rate of change on web pages. Another problem with today's dynamic world is that web pages on the internet change very frequently. As a result, by the time the crawler is downloading the last page from a site, the page may change, or a new page may be added to the site.

A bare minimum crawler needs at least these components:

1. URL frontier: To store the list of URLs to download and also prioritize which URLs should be crawled first.

2. HTML Fetcher: To retrieve a web page from the server.

- 3. Extractor:** To extract links from HTML documents.
- 4. Duplicate Eliminator:** To make sure the same content is not extracted twice unintentionally.
- 5. Datastore:** To store retrieved pages, URLs, and other metadata.



6. Detailed Component Design

Let's assume our crawler is running on one server and all the crawling is done by multiple working threads where each working thread performs all the steps needed to download and process a document in a loop.

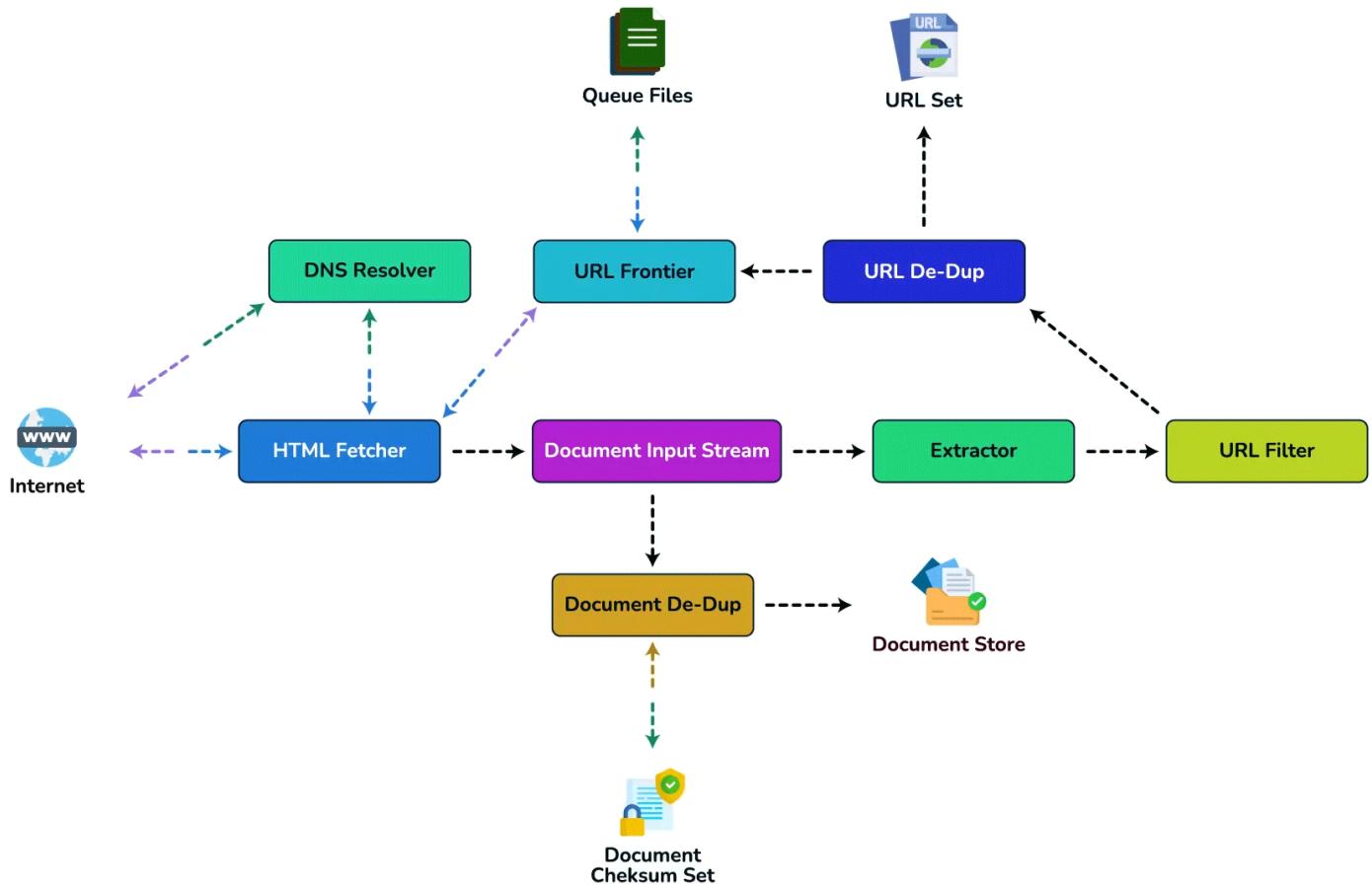
The first step of this loop is to remove an absolute URL from the shared URL frontier for downloading. An absolute URL begins with a scheme (e.g., “HTTP”) which identifies the network protocol that should be used to download it. We can implement these protocols in a modular way for extensibility, so that later if our crawler needs to support more protocols, it can be easily done. Based on the URL’s

scheme, the worker calls the appropriate protocol module to download the document. After downloading, the document is placed into a Document Input Stream (DIS). Putting documents into DIS will enable other modules to re-read the document multiple times.

Once the document has been written to the DIS, the worker thread invokes the dedupe test to determine whether this document (associated with a different URL) has been seen before. If so, the document is not processed any further and the worker thread removes the next URL from the frontier.

Next, our crawler needs to process the downloaded document. Each document can have a different MIME type like HTML page, Image, Video, etc. We can implement these MIME schemes in a modular way, so that later if our crawler needs to support more types, we can easily implement them. Based on the downloaded document's MIME type, the worker invokes the process method of each processing module associated with that MIME type.

Furthermore, our HTML processing module will extract all links from the page. Each link is converted into an absolute URL and tested against a user-supplied URL filter to determine if it should be downloaded. If the URL passes the filter, the worker performs the URL-seen test, which checks if the URL has been seen before, namely, if it is in the URL frontier or has already been downloaded. If the URL is new, it is added to the frontier.



Detailed Design

Let's discuss these components one by one, and see how they can be distributed onto multiple machines:

1. The URL frontier: The URL frontier is the data structure that contains all the URLs that remain to be downloaded. We can crawl by performing a breadth-first traversal of the Web, starting from the pages in the seed set. Such traversals are easily implemented by using a FIFO queue.

Since we'll be having a huge list of URLs to crawl, we can distribute our URL frontier into multiple servers. Let's assume on each server we have multiple worker threads performing the crawling tasks. Let's also assume that our hash function maps each URL to a server which will be responsible for crawling it.

Following politeness requirements must be kept in mind while designing a distributed URL frontier:

1. Our crawler should not overload a server by downloading a lot of pages from it.
2. We should not have multiple machines connecting a web server.

To implement this politeness constraint our crawler can have a collection of distinct FIFO sub-queues on each server. Each worker thread will have its separate sub-queue, from which it removes URLs for crawling. When a new URL needs to be added, the FIFO sub-queue in which it is placed will be determined by the URL's canonical hostname. Our hash function can map each hostname to a thread number. Together, these two points imply that, at most, one worker thread will download documents from a given Web server, and also, by using the FIFO queue, it'll not overload a Web server.

How big will our URL frontier be? The size would be in the hundreds of millions of URLs. Hence, we need to store our URLs on a disk. We can implement our queues in such a way that they have separate buffers for enqueueing and dequeuing. Enqueue buffer, once filled, will be dumped to the disk, whereas dequeue buffer will keep a cache of URLs that need to be visited; it can periodically read from disk to fill the buffer.

2. The fetcher module: The purpose of a fetcher module is to download the document corresponding to a given URL using the appropriate network protocol like HTTP. As discussed above, webmasters create robot.txt to make certain parts of their websites off-limits for the crawler. To avoid downloading this file on every request, our crawler's HTTP protocol module can maintain a fixed-sized cache mapping host-names to their robot's exclusion rules.

3. Document input stream: Our crawler's design enables the same document to be processed by multiple processing modules. To avoid downloading a document

multiple times, we cache the document locally using an abstraction called a Document Input Stream (DIS).

A DIS is an input stream that caches the entire contents of the document read from the internet. It also provides methods to re-read the document. The DIS can cache small documents (64 KB or less) entirely in memory, while larger documents can be temporarily written to a backing file.

Each worker thread has an associated DIS, which it reuses from document to document. After extracting a URL from the frontier, the worker passes that URL to the relevant protocol module, which initializes the DIS from a network connection to contain the document's contents. The worker then passes the DIS to all relevant processing modules.

4. Document Dedupe test: Many documents on the Web are available under multiple, different URLs. There are also many cases in which documents are mirrored on various servers. Both of these effects will cause any Web crawler to download the same document multiple times. To prevent the processing of a document more than once, we perform a dedupe test on each document to remove duplication.

To perform this test, we can calculate a 64-bit checksum of every processed document and store it in a database. For every new document, we can compare its checksum to all the previously calculated checksums to see if the document has been seen before. We can use MD5 or SHA to calculate these checksums.

How big would the checksum store be? If the whole purpose of our checksum store is to do dedupe, then we just need to keep a unique set containing checksums of all previously processed documents. Considering 15 billion distinct web pages, we would need:

$$15B * 8 \text{ bytes} \Rightarrow 120 \text{ GB}$$

Although this can fit into a modern-day server's memory, if we don't have enough memory available, we can keep smaller LRU based cache on each server with everything backed by persistent storage. The dedupe test first checks if the checksum is present in the cache. If not, it has to check if the checksum resides in the back storage. If the checksum is found, we will ignore the document. Otherwise, it will be added to the cache and back storage.

5. URL filters: The URL filtering mechanism provides a customizable way to control the set of URLs that are downloaded. This is used to block websites so that our crawler can ignore them. Before adding each URL to the frontier, the worker thread consults the user-supplied URL filter. We can define filters to restrict URLs by domain, prefix, or protocol type.

6. Domain name resolution: Before contacting a Web server, a Web crawler must use the Domain Name Service (DNS) to map the Web server's hostname into an IP address. DNS name resolution will be a big bottleneck for our crawlers given the huge number of URLs we will be working with. To avoid repeated requests, we can start caching DNS results by building our local DNS server.

7. URL dedupe test: While extracting links, any Web crawler will encounter multiple links to the same document. To avoid downloading and processing a document multiple times, a URL dedupe test must be performed on each extracted link before adding it to the URL frontier.

To perform the URL dedupe test, we can store all the URLs seen by our crawler in canonical form in a database. To save space, we do not store the textual representation of each URL in the URL set, but rather a fixed-sized checksum.

To reduce the number of operations on the database store, we can keep an in-memory cache of popular URLs on each host shared by all threads. The reason to have this cache is that links to some URLs are quite common, so caching the popular ones in memory will lead to a high in-memory hit rate.

How much storage we would need for URL's store? If the whole purpose of our checksum is to do URL dedupe, then we just need to keep a unique set containing checksums of all previously seen URLs. Considering 15 billion distinct URLs and 8 bytes for checksum, we would need:

$$15B * 8 \text{ bytes} \Rightarrow 120 \text{ GB}$$

Can we use bloom filters for deduping? Bloom filters are a probabilistic data structure for set membership testing that may yield false positives. A large bit vector represents the set. An element is added to the set by computing n hash functions of the element and setting the corresponding bits. An element is deemed to be in the set if the bits at all ' n ' of the element's hash locations are set. Hence, a document may incorrectly be deemed to be in the set, but false negatives are not possible.

The disadvantage of using a bloom filter for the URL seen test is that each false positive will cause the URL not to be added to the frontier and, therefore, the document will never be downloaded. The chance of a false positive can be reduced by making the bit vector larger.

8. Checkpointing: A crawl of the entire Web takes weeks to complete. To guard against failures, our crawler can write regular snapshots of its state to the disk. An interrupted or aborted crawl can easily be restarted from the latest checkpoint.

7. Fault Tolerance

We should use consistent hashing for distribution among crawling servers. Consistent hashing will not only help in replacing a dead host but also help in distributing load among crawling servers.

All our crawling servers will be performing regular checkpointing and storing their FIFO queues to disks. If a server goes down, we can replace it. Meanwhile,

consistent hashing should shift the load to other servers.

8. Data Partitioning

Our crawler will be dealing with three kinds of data: 1) URLs to visit 2) URL checksums for dedupe 3) Document checksums for dedupe.

Since we are distributing URLs based on the hostnames, we can store these data on the same host. So, each host will store its set of URLs that need to be visited, checksums of all the previously visited URLs, and checksums of all the downloaded documents. Since we will be using consistent hashing, we can assume that URLs will be redistributed from overloaded hosts.

Each host will perform checkpointing periodically and dump a snapshot of all the data it is holding onto a remote server. This will ensure that if a server dies down another server can replace it by taking its data from the last snapshot.

9. Crawler Traps

There are many crawler traps, spam sites, and cloaked content. A crawler trap is a URL or set of URLs that cause a crawler to crawl indefinitely. Some crawler traps are unintentional. For example, a symbolic link within a file system can create a cycle. Other crawler traps are introduced intentionally. For example, people have written traps that dynamically generate an infinite Web of documents. The motivations behind such traps vary. Anti-spam traps are designed to catch crawlers used by spammers looking for email addresses, while other sites use traps to catch search engine crawlers to boost their search ratings.

Designing Facebook's Newsfeed

Let's design Facebook's Newsfeed, which would contain posts, photos, videos, and status updates from all the people and pages a user follows.

Similar Services: Twitter Newsfeed, Instagram Newsfeed, Quora Newsfeed

Difficulty Level: Hard

1. What is Facebook's newsfeed?

A Newsfeed is the constantly updating list of stories in the middle of Facebook's homepage. It includes status updates, photos, videos, links, app activity, and 'likes' from people, pages, and groups that a user follows on Facebook. In other words, it is a compilation of a complete scrollable version of your friends' and your life story from photos, videos, locations, status updates, and other activities.

For any social media site you design - Twitter, Instagram, or Facebook - you will need some newsfeed system to display updates from friends and followers.

2. Requirements and Goals of the System

Let's design a newsfeed for Facebook with the following requirements:

Functional requirements:

1. Newsfeed will be generated based on the posts from the people, pages, and groups that a user follows.
2. A user may have many friends and follow a large number of pages/groups.
3. Feeds may contain images, videos, or just text.

4. Our service should support appending new posts as they arrive to the newsfeed for all active users.

Non-functional requirements:

1. Our system should be able to generate any user's newsfeed in real-time - maximum latency seen by the end user would be 2s.
2. A post shouldn't take more than 5s to make it to a user's feed assuming a new newsfeed request comes in.

3. Capacity Estimation and Constraints

Let's assume on average a user has 300 friends and follows 200 pages.

Traffic estimates: Let's assume 300M daily active users with each user fetching their timeline an average of five times a day. This will result in 1.5B newsfeed requests per day or approximately 17,500 requests per second.

Storage estimates: On average, let's assume we need to have around 500 posts in every user's feed that we want to keep in memory for a quick fetch. Let's also assume that on average each post would be 1KB in size. This would mean that we need to store roughly 500KB of data per user. To store all this data for all the active users we would need 150TB of memory. If a server can hold 100GB we would need around 1500 machines to keep the top 500 posts in memory for all active users.

4. System APIs

- 💡 Once we have finalized the requirements, it's always a good idea to define the system APIs. This should explicitly state what is expected from the system.

We can have SOAP or REST APIs to expose the functionality of our service. The following could be the definition of the API for getting the newsfeed:

```
getUserFeed(api_dev_key, user_id, since_id, count, max_id,  
exclude_replies)
```

Parameters:

api_dev_key (string): The API developer key of a registered can be used to, among other things, throttle users based on their allocated quota.

user_id (number): The ID of the user for whom the system will generate the newsfeed.

since_id (number): Optional; returns results with an ID higher than (that is, more recent than) the specified ID.

count (number): Optional; specifies the number of feed items to try and retrieve up to a maximum of 200 per distinct request.

max_id (number): Optional; returns results with an ID less than (that is, older than) or equal to the specified ID.

exclude_replies(boolean): Optional; this parameter will prevent replies from appearing in the returned timeline.

Returns: (JSON) Returns a JSON object containing a list of feed items.

5. Database Design

There are three primary objects: User, Entity (e.g. page, group, etc.), and FeedItem (or Post). Here are some observations about the relationships between these entities:

- A User can follow other entities and can become friends with other users.

- Both users and entities can post FeedItems which can contain text, images, or videos.
- Each FeedItem will have a UserID which will point to the User who created it. For simplicity, let's assume that only users can create feed items, although, on Facebook Pages can post feed item too.
- Each FeedItem can optionally have an EntityID pointing to the page or the group where that post was created.

If we are using a relational database, we would need to model two relations: User-Entity relation and FeedItem-Media relation. Since each user can be friends with many people and follow a lot of entities, we can store this relation in a separate table. The “Type” column in “UserFollow” identifies if the entity being followed is a User or Entity. Similarly, we can have a table for FeedMedia relation.

User	
PK	<u>UserID: int</u>
	Name: varchar(20)
	Email: varchar(32)
	DateOfBirth: datetime
	CreationDate: datetime
	LastLogin: datetime

Entity	
PK	<u>EntityID: int</u>
	Name: varchar(20)
	Type: tinyint
	Description: varchar(512)
	CreationDate: datetime
	Category: smallint
	Phone: varchar(12)
	Email: varchar(20)

UserFollow	
PK	<u>UserID: int</u> <u>EntityOrFriendID: int</u>
	Type: tinyint

FeedItem	
PK	<u>FeedItemID: int</u>
	UserID: int
	Contents: varchar(256)
	EntityID: int
	LocationLatitude: int
	LocationLongitude: int
	CreationDate: datetime
	NumLikes: int

FeedMedia	
PK	<u>FeedItemID: int</u> <u>MediaID: int</u>

Media	
PK	<u>MediaID: int</u>
	Type: smallint
	Description: varchar(256)
	Path: varchar(256)
	LocationLatitude: int
	LocationLongitude: int
	CreationDate: datetime

DB Schema

6. High Level System Design

At a high level this problem can be divided into two parts:

Feed generation: Newsfeed is generated from the posts (or feed items) of users and entities (pages and groups) that a user follows. So, whenever our system receives a request to generate the feed for a user (say Jane), we will perform the following steps:

1. Retrieve IDs of all users and entities that Jane follows.
2. Retrieve latest, most popular and relevant posts for those IDs. These are the potential posts that we can show in Jane's newsfeed.
3. Rank these posts based on the relevance to Jane. This represents Jane's current feed.
4. Store this feed in the cache and return top posts (say 20) to be rendered on Jane's feed.
5. On the front-end, when Jane reaches the end of her current feed, she can fetch the next 20 posts from the server and so on.

One thing to notice here is that we generated the feed once and stored it in the cache. What about new incoming posts from people that Jane follows? If Jane is online, we should have a mechanism to rank and add those new posts to her feed. We can periodically (say every five minutes) perform the above steps to rank and add the newer posts to her feed. Jane can then be notified that there are newer items in her feed that she can fetch.

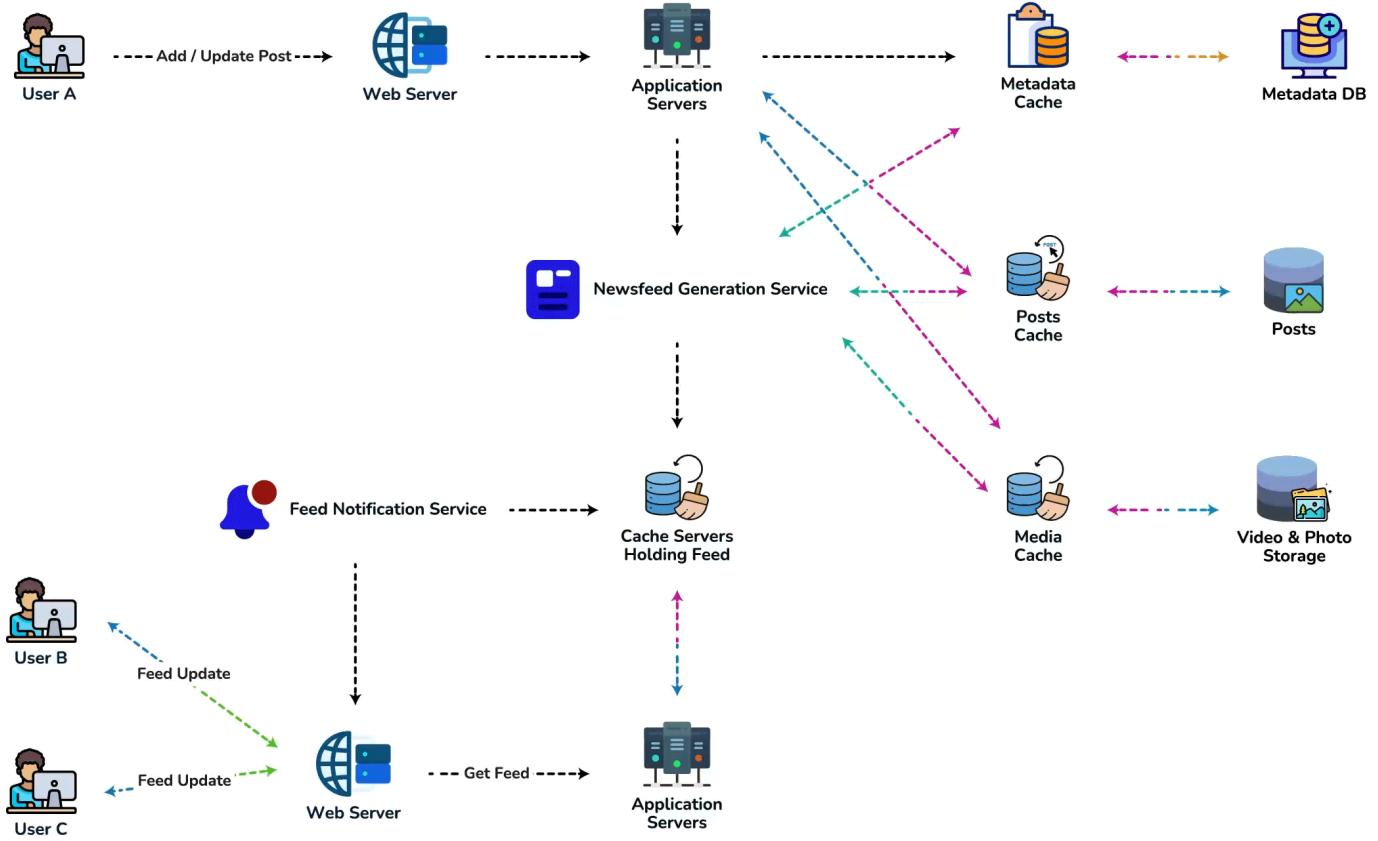
Feed publishing: Whenever Jane loads her newsfeed page, she has to request and pull feed items from the server. When she reaches the end of her current feed, she can pull more data from the server. For newer items either the server can notify

Jane and then she can pull, or the server can push, these new posts. We will discuss these options in detail later.

At a high level, we will need following components in our Newsfeed service:

- 1. Web servers:** To maintain a connection with the user. This connection will be used to transfer data between the user and the server.
- 2. Application server:** To execute the workflows of storing new posts in the database servers. We will also need some application servers to retrieve and to push the newsfeed to the end user.
- 3. Metadata database and cache:** To store the metadata about Users, Pages, and Groups.
- 4. Posts database and cache:** To store metadata about posts and their contents.
- 5. Video and photo storage, and cache:** Blob storage, to store all the media included in the posts.
- 6. Newsfeed generation service:** To gather and rank all the relevant posts for a user to generate newsfeed and store in the cache. This service will also receive live updates and will add these newer feed items to any user's timeline.
- 7. Feed notification service:** To notify the user that there are newer items available for their newsfeed.

Following is the high-level architecture diagram of our system. User B and C are following User A.



Facebook Newsfeed Architecture

7. Detailed Component Design

Let's discuss different components of our system in detail.

a. Feed generation

Let's take the simple case of the newsfeed generation service fetching most recent posts from all the users and entities that Jane follows; the query would look like this:

```
(SELECT FeedItemID FROM FeedItem WHERE UserID in (
    SELECT EntityOrFriendID FROM UserFollow WHERE UserID =
```

```

<current_user_id> and type = 0(user))
)
UNION
(SELECT FeedItemID FROM FeedItem WHERE EntityID in (
    SELECT EntityOrFriendID FROM UserFollow WHERE UserID =
<current_user_id> and type = 1(entity))
)
ORDER BY CreationDate DESC
LIMIT 100

```

Here are issues with this design for the feed generation service:

1. Crazy slow for users with a lot of friends/follows as we have to perform sorting/merging/ranking of a huge number of posts.
2. We generate the timeline when a user loads their page. This would be quite slow and have a high latency.
3. For live updates, each status update will result in feed updates for all followers. This could result in high backlogs in our Newsfeed Generation Service.
4. For live updates, the server pushing (or notifying about) newer posts to users could lead to very heavy loads, especially for people or pages that have a lot of followers. To improve the efficiency, we can pre-generate the timeline and store it in a memory.

Offline generation for newsfeed: We can have dedicated servers that are continuously generating users' newsfeed and storing them in memory. So, whenever a user requests for the new posts for their feed, we can simply serve it from the pre-generated, stored location. Using this scheme, user's newsfeed is not compiled on load, but rather on a regular basis and returned to users whenever they request for it.

Whenever these servers need to generate the feed for a user, they will first query to see what was the last time the feed was generated for that user. Then, new feed

data would be generated from that time onwards. We can store this data in a hash table where the “key” would be UserID and “value” would be a STRUCT like this:

```
Struct {  
    LinkedHashMap<FeedItemID, FeedItem> feedItems;  
    DateTime lastGenerated;  
}
```

We can store FeedItemIDs in a data structure similar to [Linked HashMap](#) or [TreeMap](#), which can allow us to not only jump to any feed item but also iterate through the map easily. Whenever users want to fetch more feed items, they can send the last FeedItemID they currently see in their newsfeed, we can then jump to that FeedItemID in our hash-map and return next batch/page of feed items from there.

How many feed items should we store in memory for a user’s feed? Initially, we can decide to store 500 feed items per user, but this number can be adjusted later based on the usage pattern. For example, if we assume that one page of a user’s feed has 20 posts and most of the users never browse more than ten pages of their feed, we can decide to store only 200 posts per user. For any user who wants to see more posts (more than what is stored in memory), we can always query backend servers.

Should we generate (and keep in memory) newsfeeds for all users? There will be a lot of users that don’t log-in frequently. Here are a few things we can do to handle this; 1) a more straightforward approach could be, to use an LRU based cache that can remove users from memory that haven’t accessed their newsfeed for a long time 2) a smarter solution can figure out the login pattern of users to pre-generate their newsfeed, e.g., at what time of the day a user is active and which days of the week does a user access their newsfeed? etc.

Let's now discuss some solutions to our "live updates" problems in the following section.

b. Feed publishing

The process of pushing a post to all the followers is called fanout. By analogy, the push approach is called fanout-on-write, while the pull approach is called fanout-on-load. Let's discuss different options for publishing feed data to users.

1. **"Pull" model or Fan-out-on-load:** This method involves keeping all the recent feed data in memory so that users can pull it from the server whenever they need it. Clients can pull the feed data on a regular basis or manually whenever they need it. Possible problems with this approach are a) New data might not be shown to the users until they issue a pull request, b) It's hard to find the right pull cadence, as most of the time pull requests will result in an empty response if there is no new data, causing waste of resources.
2. **"Push" model or Fan-out-on-write:** For a push system, once a user has published a post, we can immediately push this post to all the followers. The advantage is that when fetching feed you don't need to go through your friend's list and get feeds for each of them. It significantly reduces read operations. To efficiently handle this, users have to maintain a **Long Poll** request with the server for receiving the updates. A possible problem with this approach is that when a user has millions of followers (a celebrity-user) the server has to push updates to a lot of people.
3. **Hybrid:** An alternate method to handle feed data could be to use a hybrid approach, i.e., to do a combination of fan-out-on-write and fan-out-on-load. Specifically, we can stop pushing posts from users with a high number of followers (a celebrity user) and only push data for those users who have a few hundred (or thousand) followers. For celebrity users, we can let the followers pull the updates. Since the push operation can be extremely costly for users who have a lot of friends or followers, by disabling fanout for them, we can

save a huge number of resources. Another alternate approach could be that, once a user publishes a post, we can limit the fanout to only her online friends. Also, to get benefits from both the approaches, a combination of 'push to notify' and 'pull for serving' end-users is a great way to go. Purely a push or pull model is less versatile.

How many feed items can we return to the client in each request? We should have a maximum limit for the number of items a user can fetch in one request (say 20). But, we should let the client specify how many feed items they want with each request as the user may like to fetch a different number of posts depending on the device (mobile vs. desktop).

Should we always notify users if there are new posts available for their newsfeed? It could be useful for users to get notified whenever new data is available. However, on mobile devices, where data usage is relatively expensive, it can consume unnecessary bandwidth. Hence, at least for mobile devices, we can choose not to push data, instead, let users “Pull to Refresh” to get new posts.

8. Feed Ranking

The most straightforward way to rank posts in a newsfeed is by the creation time of the posts, but today’s ranking algorithms are doing a lot more than that to ensure “important” posts are ranked higher. The high-level idea of ranking is first to select key “signals” that make a post important and then to find out how to combine them to calculate a final ranking score.

More specifically, we can select features that are relevant to the importance of any feed item, e.g., number of likes, comments, shares, time of the update, whether the post has images/videos, etc., and then, a score can be calculated using these features. This is generally enough for a simple ranking system. A better ranking

system can significantly improve itself by constantly evaluating if we are making progress in user stickiness, retention, ads revenue, etc.

9. Data Partitioning

a. Sharding posts and metadata

Since we have a huge number of new posts every day and our read load is extremely high too, we need to distribute our data onto multiple machines such that we can read/write it efficiently. For sharding our databases that are storing posts and their metadata, we can have a similar design as discussed under 'Designing Twitter'.

b. Sharding feed data

For feed data, which is being stored in memory, we can partition it based on UserID. We can try storing all the data of a user on one server. When storing, we can pass the UserID to our hash function that will map the user to a cache server where we will store the user's feed objects. Also, for any given user, since we don't expect to store more than 500 FeedItemIDs, we will not run into a scenario where feed data for a user doesn't fit on a single server. To get the feed of a user, we would always have to query only one server. For future growth and replication, we must use 'Consistent Hashing'.

Designing Yelp or Nearby Friends

Let's design a Yelp like service, where users can search for nearby places like restaurants, theaters, or shopping malls, etc., and can also add/view reviews of places.

Similar Services: Proximity server.

Difficulty Level: Hard

1. Why Yelp or Proximity Server?

Proximity servers are used to discover nearby attractions like places, events, etc. If you haven't used yelp.com before, please try it before proceeding (you can search for nearby restaurants, theaters, etc.) and spend some time understanding different options that the website offers. This will help you a lot in understanding this chapter better.

Designing Yelp (video)

Here is a video discussing how to design Yelp or Proximity Server:

1:43:04

Designing Yelp

2. Requirements and Goals of the System

What do we wish to achieve from a Yelp like service? Our service will be storing information about different places so that users can perform a search on them. Upon querying, our service will return a list of places around the user.

Our Yelp-like service should meet the following requirements:

Functional Requirements:

1. Users should be able to add/delete/update Places.

2. Given their location (longitude/latitude), users should be able to find all nearby places within a given radius.
3. Users should be able to add feedback/review about a place. The feedback can have pictures, text, and a rating.

Non-functional Requirements:

1. Users should have a real-time search experience with minimum latency.
2. Our service should support a heavy search load. There will be a lot of search requests compared to adding a new place.

3. Scale Estimation

Let's build our system assuming that we have 500M places and 100K queries per second (QPS). Let's also assume a 20% growth in the number of places and QPS each year.

4. Database Schema

Each Place can have the following fields:

1. LocationID (8 bytes): Uniquely identifies a location.
2. Name (256 bytes)
3. Latitude (8 bytes)
4. Longitude (8 bytes)
5. Description (512 bytes)
6. Category (1 byte): E.g., coffee shop, restaurant, theater, etc.

Although a four bytes number can uniquely identify 500M locations, with future growth in mind, we will go with 8 bytes for LocationID.

Total size: $8 + 256 + 8 + 8 + 512 + 1 \Rightarrow 793$ bytes

We also need to store reviews, photos, and ratings of a Place. We can have a separate table to store reviews for Places:

1. LocationID (8 bytes)
2. ReviewID (4 bytes): Uniquely identifies a review, assuming any location will not have more than 2^{32} reviews.
3. ReviewText (512 bytes)
4. Rating (1 byte): how many stars a place gets out of ten.

Similarly, we can have a separate table to store photos for Places and Reviews.

5. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. The following could be the definition of the API for searching:

```
search(api_dev_key, search_terms, user_location,  
radius_filter, maximum_results_to_return, category_filter,  
sort, page_token)
```

Parameters:

api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.

search_terms (string): A string containing the search terms.

user_location (string): Location of the user performing the search.

radius_filter (number): Optional search radius in meters.

maximum_results_to_return (number): Number of business results to return.

category_filter (string): Optional category to filter search results, e.g., Restaurants,

Shopping Centers, etc.

sort (number): Optional sort mode: Best matched (0 - default), Minimum distance (1), Highest rated (2).

page_token (string): This token will specify a page in the result set that should be returned.

Returns: (JSON)

A JSON containing information about a list of businesses matching the search query. Each result entry will have the business name, address, category, rating, and thumbnail.

6. Basic System Design and Algorithm

At a high level, we need to store and index each dataset described above (places, reviews, etc.). For users to query this massive database, the indexing should be read efficient, since while searching for the nearby places users expect to see the results in real-time.

Given that the location of a place doesn't change that often, we don't need to worry about frequent updates of the data. As a contrast, if we intend to build a service where objects do change their location frequently, e.g., people or taxis, then we might come up with a very different design.

Let's see what are different ways to store this data and also find out which method will suit best for our use cases:

a. SQL solution

One simple solution could be to store all the data in a database like MySQL. Each place will be stored in a separate row, uniquely identified by LocationID. Each place will have its longitude and latitude stored separately in two different

columns, and to perform a fast search; we should have indexes on both these fields.

To find all the nearby places of a given location (X, Y) within a radius 'D', we can query like this:

```
Select * from Places where Latitude between X-D and X+D and Longitude between Y-D  
and Y+D
```

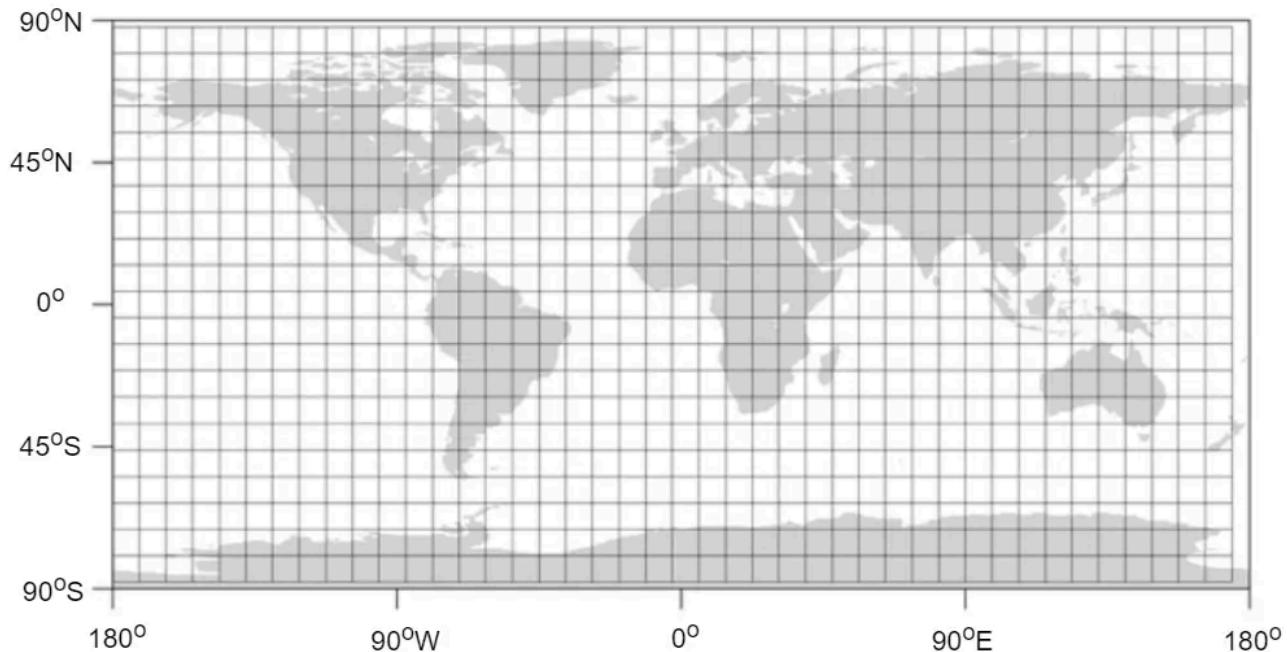
The above query is not completely accurate, as we know that to find the distance between two points we have to use the distance formula (Pythagorean theorem), but for simplicity let's take this.

How efficient would this query be? We have estimated 500M places to be stored in our service. Since we have two separate indexes, each index can return a huge list of places and performing an intersection on those two lists won't be efficient. Another way to look at this problem is that there could be too many locations between 'X-D' and 'X+D', and similarly between 'Y-D' and 'Y+D'. If we can somehow shorten these lists, it can improve the performance of our query.

b. Grids

We can divide the whole map into smaller grids to group locations into smaller sets. Each grid will store all the Places residing within a specific range of longitude and latitude. This scheme would enable us to query only a few grids to find nearby places. Based on a given location and radius, we can find all the neighboring grids and then query these grids to find nearby places.

Grid of two dimensional data



Let's assume that GridID (a four bytes number) would uniquely identify grids in our system.

What could be a reasonable grid size? Grid size could be equal to the distance we would like to query since we also want to reduce the number of grids. If the grid size is equal to the distance we want to query, then we only need to search within the grid which contains the given location and neighboring eight grids. Since our grids would be statically defined (from the fixed grid size), we can easily find the grid number of any location (lat, long) and its neighboring grids.

In the database, we can store the GridID with each location and have an index on it, too, for faster searching. Now, our query will look like:

```
Select * from Places where Latitude between X-D and X+D and Longitude between Y-D  
and Y+D and GridID in (GridID, GridID1, GridID2, ..., GridID8)
```

This will undoubtedly improve the runtime of our query.

Should we keep our index in memory? Maintaining the index in memory will improve the performance of our service. We can keep our index in a hash table where 'key' is the grid number and 'value' is the list of places contained in that grid.

How much memory will we need to store the index? Let's assume our search radius is 10 miles; given that the total area of the earth is around 200 million square miles, we will have 20 million grids. We would need a four bytes number to uniquely identify each grid and, since LocationID is 8 bytes, we would need 4GB of memory (ignoring hash table overhead) to store the index.

$$(4 * 20M) + (8 * 500M) \approx 4 \text{ GB}$$

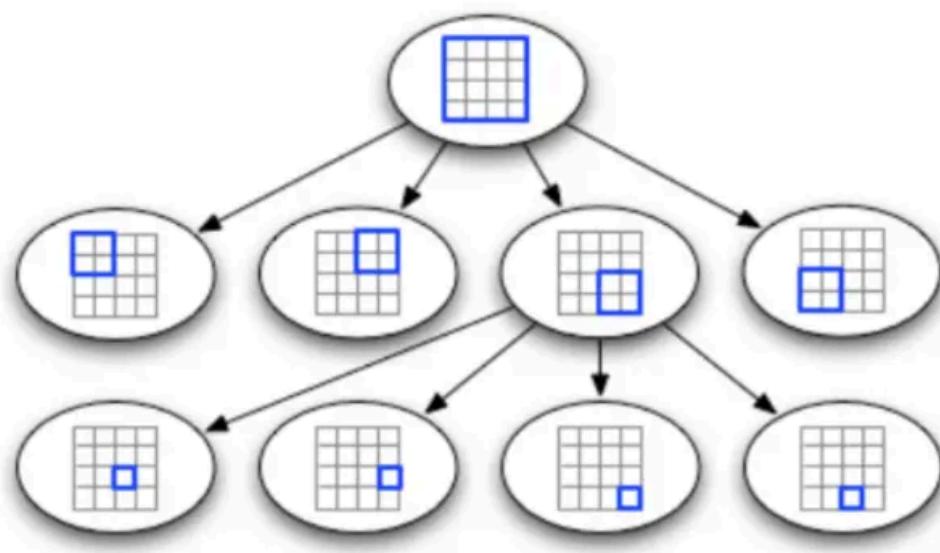
This solution can still run slow for those grids that have a lot of places since our places are not uniformly distributed among grids. We can have a thickly dense area with a lot of places, and on the other hand, we can have areas which are sparsely populated.

This problem can be solved if we can dynamically adjust our grid size such that whenever we have a grid with a lot of places we break it down to create smaller grids. A couple of challenges with this approach could be: 1) how to map these grids to locations and 2) how to find all the neighboring grids of a grid.

c. Dynamic size grids

Let's assume we don't want to have more than 500 places in a grid so that we can have a faster searching. So, whenever a grid reaches this limit, we break it down into four grids of equal size and distribute places among them. This means thickly populated areas like downtown San Francisco will have a lot of grids, and sparsely populated area like the Pacific Ocean will have large grids with places only around the coastal lines.

What data-structure can hold this information? A tree in which each node has four children can serve our purpose. Each node will represent a grid and will contain information about all the places in that grid. If a node reaches our limit of 500 places, we will break it down to create four child nodes under it and distribute places among them. In this way, all the leaf nodes will represent the grids that cannot be further broken down. So leaf nodes will keep a list of places with them. This tree structure in which each node can have four children is called a [QuadTree](#).



QuadTree

How will we build a QuadTree? We will start with one node that will represent the whole world in one grid. Since it will have more than 500 locations, we will break it down into four nodes and distribute locations among them. We will keep repeating this process with each child node until there are no nodes left with more than 500 locations.

How will we find the grid for a given location? We will start with the root node and search downward to find our required node/grid. At each step, we will see if the current node we are visiting has children. If it has, we will move to the child node that contains our desired location and repeat this process. If the node does not have any children, then that is our desired node.

How will we find neighboring grids of a given grid? Since only leaf nodes contain a list of locations, we can connect all leaf nodes with a doubly linked list. This way we can iterate forward or backward among the neighboring leaf nodes to find out our desired locations. Another approach for finding adjacent grids would be through parent nodes. We can keep a pointer in each node to access its parent, and since each parent node has pointers to all of its children, we can easily find siblings of a node. We can keep expanding our search for neighboring grids by going up through the parent pointers.

Once we have nearby LocationIDs, we can query the backend database to find details about those places.

What will be the search workflow? We will first find the node that contains the user's location. If that node has enough desired places, we can return them to the user. If not, we will keep expanding to the neighboring nodes (either through the parent pointers or doubly linked list) until either we find the required number of places or exhaust our search based on the maximum radius.

How much memory will be needed to store the QuadTree? For each Place, if we cache only LocationID and Lat/Long, we would need 12GB to store all places.

$$24 * 500M \Rightarrow 12 \text{ GB}$$

Since each grid can have a maximum of 500 places, and we have 500M locations, how many total grids we will have?

$$500M / 500 \Rightarrow 1M \text{ grids}$$

Which means we will have 1M leaf nodes and they will be holding 12GB of location data. A QuadTree with 1M leaf nodes will have approximately 1/3rd internal nodes, and each internal node will have 4 pointers (for its children). If each pointer is 8 bytes, then the memory we need to store all internal nodes would be:

$$1M * 1/3 * 4 * 8 = 10 \text{ MB}$$

So, total memory required to hold the whole QuadTree would be 12.01GB. This can easily fit into a modern-day server.

How would we insert a new Place into our system? Whenever a new Place is added by a user, we need to insert it into the databases as well as in the QuadTree. If our tree resides on one server, it is easy to add a new Place, but if the QuadTree is distributed among different servers, first we need to find the grid/server of the new Place and then add it there (discussed in the next section).

7. Data Partitioning

What if we have a huge number of places such that our index does not fit into a single machine's memory? With 20% growth each year we will reach the memory limit of the server in the future. Also, what if one server cannot serve the desired read traffic? To resolve these issues, we must partition our QuadTree!

We will explore two solutions here (both of these partitioning schemes can be applied to databases, too):

a. Sharding based on regions: We can divide our places into regions (like zip codes), such that all places belonging to a region will be stored on a fixed node. To store a place we will find the server through its region and, similarly, while querying for nearby places we will ask the region server that contains user's location. This approach has a couple of issues:

1. What if a region becomes hot? There would be a lot of queries on the server holding that region, making it perform slow. This will affect the performance of our service.
2. Over time, some regions can end up storing a lot of places compared to others. Hence, maintaining a uniform distribution of places, while regions are growing is quite difficult.

To recover from these situations, either we have to repartition our data or use consistent hashing.

b. Sharding based on LocationID: Our hash function will map each LocationID to a server where we will store that place. While building our QuadTree, we will iterate through all the places and calculate the hash of each LocationID to find a server where it would be stored. To find places near a location, we have to query all servers and each server will return a set of nearby places. A centralized server will aggregate these results to return them to the user.

Will we have different QuadTree structure on different partitions? Yes, this can happen since it is not guaranteed that we will have an equal number of places in any given grid on all partitions. However, we do make sure that all servers have approximately an equal number of Places. This different tree structure on different servers will not cause any issue though, as we will be searching all the neighboring grids within the given radius on all partitions.

The remaining part of this chapter assumes that we have partitioned our data based on LocationID.

8. Replication and Fault Tolerance

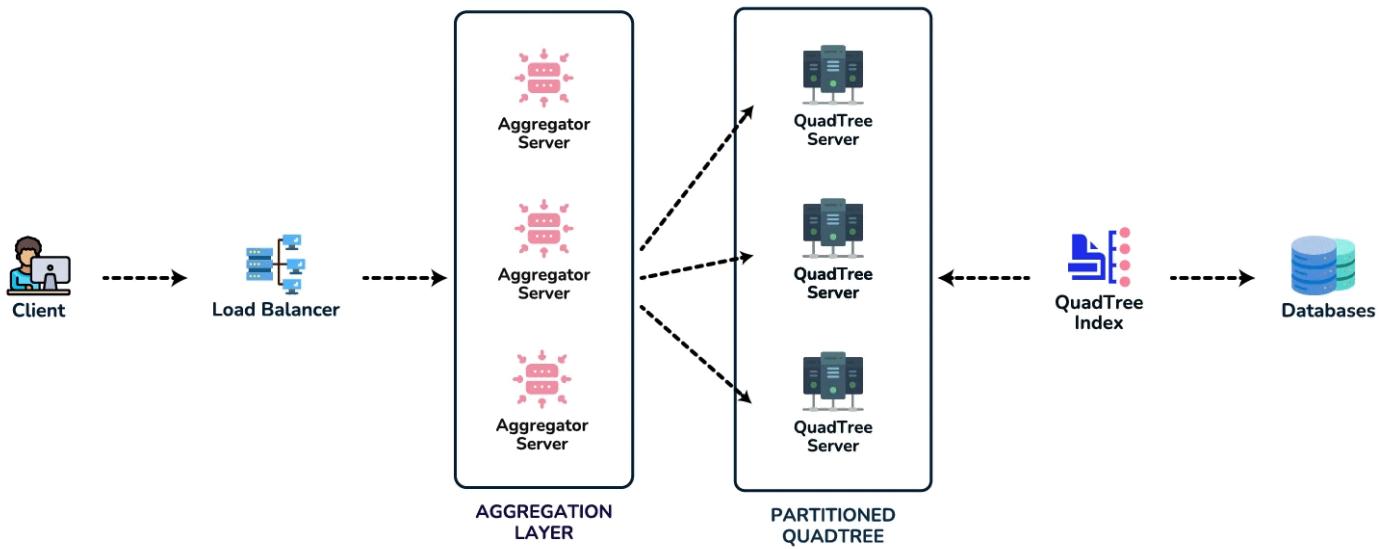
Having replicas of QuadTree servers can provide an alternate to data partitioning. To distribute read traffic, we can have replicas of each QuadTree server. We can have a primary-secondary configuration where replicas (secondaries) will only serve read traffic; all write traffic will first go to the primary and then applied to secondaries. Secondaries might not have some recently inserted places (a few milliseconds delay will be there), but this could be acceptable.

What will happen when a QuadTree server dies? We can have a secondary replica of each server and, if the primary dies, it can take control after the failover. Both

primary and secondary servers will have the same QuadTree structure.

What if both primary and secondary servers die at the same time? We have to allocate a new server and rebuild the same QuadTree on it. How can we do that, since we don't know what places were kept on this server? The brute-force solution would be to iterate through the whole database and filter LocationIDs using our hash function to figure out all the required places that will be stored on this server. This would be inefficient and slow; also, during the time when the server is being rebuilt, we will not be able to serve any query from it, thus missing some places that should have been seen by users.

How can we efficiently retrieve a mapping between Places and QuadTree server? We have to build a reverse index that will map all the Places to their QuadTree server. We can have a separate QuadTree Index server that will hold this information. We will need to build a HashMap where the 'key' is the QuadTree server number and the 'value' is a HashSet containing all the Places being kept on that QuadTree server. We need to store LocationID and Lat/Long with each place because information servers can build their QuadTrees through this. Notice that we are keeping Places' data in a HashSet, this will enable us to add/remove Places from our index quickly. So now, whenever a QuadTree server needs to rebuild itself, it can simply ask the QuadTree Index server for all the Places it needs to store. This approach will surely be quite fast. We should also have a replica of the QuadTree Index server for fault tolerance. If a QuadTree Index server dies, it can always rebuild its index from iterating through the database.



9. Cache

To deal with hot Places, we can introduce a cache in front of our database. We can use an off-the-shelf solution like Memcache, which can store all data about hot places. Application servers, before hitting the backend database, can quickly check if the cache has that Place. Based on clients' usage pattern, we can adjust how many cache servers we need. For cache eviction policy, Least Recently Used (LRU) seems suitable for our system.

10. Load Balancing (LB)

We can add LB layer at two places in our system 1) Between Clients and Application servers and 2) Between Application servers and Backend server. Initially, a simple Round Robin approach can be adopted; that will distribute all incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is if a server is dead the load balancer will take it out of the rotation and will stop sending any traffic to it.

A problem with Round Robin LB is, it won't take server load into consideration. If a server is overloaded or slow, the load balancer will not stop sending new requests to that server. To handle this, a more intelligent LB solution would be needed that periodically queries backend server about their load and adjusts traffic based on that.

11. Ranking

How about if we want to rank the search results not just by proximity but also by popularity or relevance?

How can we return most popular places within a given radius? Let's assume we keep track of the overall popularity of each place. An aggregated number can represent this popularity in our system, e.g., how many stars a place gets out of ten (this would be an average of different rankings given by users)? We will store this number in the database as well as in the QuadTree. While searching for the top 100 places within a given radius, we can ask each partition of the QuadTree to return the top 100 places with maximum popularity. Then the aggregator server can determine the top 100 places among all the places returned by different partitions.

Remember that we didn't build our system to update place's data frequently. With this design, how can we modify the popularity of a place in our QuadTree?

Although we can search a place and update its popularity in the QuadTree, it would take a lot of resources and can affect search requests and system throughput. Assuming the popularity of a place is not expected to reflect in the system within a few hours, we can decide to update it once or twice a day, especially when the load on the system is minimum.

Our next problem, 'Designing Uber backend', discusses dynamic updates of the QuadTree in detail.

Designing Uber backend

Let's design a ride-sharing service like Uber, which connects passengers who need a ride with drivers who have a car.

Similar Services: Lyft, Didi, Via, Sidecar, etc.

Difficulty level: Hard

Prerequisite: Designing Yelp

1. What is Uber?

Uber enables its customers to book drivers for taxi rides. Uber drivers use their personal cars to drive customers around. Both customers and drivers communicate with each other through their smartphones using the Uber app.

Designing Uber (video)

Here is a video discussing how to design Uber:

1:43:04

Designing Uber

2. Requirements and Goals of the System

Let's start with building a simpler version of Uber.

There are two types of users in our system: 1) Drivers 2) Customers.

- Drivers need to regularly notify the service about their current location and their availability to pick passengers.
- Passengers get to see all the nearby available drivers.
- Customer can request a ride; nearby drivers are notified that a customer is ready to be picked up.

- Once a driver and a customer accept a ride, they can constantly see each other's current location until the trip finishes.
- Upon reaching the destination, the driver marks the journey complete to become available for the next ride.

3. Capacity Estimation and Constraints

- Let's assume we have 300M customers and 1M drivers with 1M daily active customers and 500K daily active drivers.
- Let's assume 1M daily rides.
- Let's assume that all active drivers notify their current location every three seconds.
- Once a customer puts in a request for a ride, the system should be able to contact drivers in real-time.

4. Basic System Design and Algorithm

We will take the solution discussed in 'Designing Yelp' and modify it to make it work for the above-mentioned "Uber" use cases. The biggest difference we have is that our QuadTree was not built keeping in mind that there would be frequent updates to it. So, we have two issues with our Dynamic Grid solution:

- Since all active drivers are reporting their locations every three seconds, we need to update our data structures to reflect that. If we have to update the QuadTree for every change in the driver's position, it will take a lot of time and resources. To update a driver to its new location, we must find the right grid based on the driver's previous location. If the new position does not belong to the current grid, we must remove the driver from the current grid and

move/reinsert the user to the correct grid. After this move, if the new grid reaches the maximum limit of drivers, we have to repartition it.

- We need to have a quick mechanism to propagate the current location of all the nearby drivers to any active customer in that area. Also, when a ride is in progress, our system needs to notify both the driver and passenger about the current location of the car.

Although our QuadTree helps us find nearby drivers quickly, a fast update in the tree is not guaranteed.

Do we need to modify our QuadTree every time a driver reports their location? If we don't update our QuadTree with every update from the driver, it will have some old data and will not reflect the current location of drivers correctly. If you recall, our purpose of building the QuadTree was to find nearby drivers (or places) efficiently. Since all active drivers report their location every three seconds, therefore there will be a lot more updates happening to our tree than querying for nearby drivers. So, what if we keep the latest position reported by all drivers in a hash table and update our QuadTree a little less frequently? Let's assume we guarantee that a driver's current location will be reflected in the QuadTree within 15 seconds. Meanwhile, we will maintain a hash table that will store the current location reported by drivers; let's call this DriverLocationHT.

How much memory we need for DriverLocationHT? We need to store DriveID, their present and old location, in the hash table. So, we need a total of 35 bytes to store one record:

1. DriverID (3 bytes - 1 million drivers)
2. Old latitude (8 bytes)
3. Old longitude (8 bytes)
4. New latitude (8 bytes)
5. New longitude (8 bytes) Total = 35 bytes

If we have 1 million total drivers, we need the following memory (ignoring hash table overhead):

$$1 \text{ million} * 35 \text{ bytes} \Rightarrow 35 \text{ MB}$$

How much bandwidth will our service consume to receive location updates from all drivers? If we get DriverID and their location, it will be (3+16 => 19 bytes). If we receive this information every three seconds from 500K daily active drivers, we will be getting 9.5MB per three seconds.

Do we need to distribute DriverLocationHT onto multiple servers? Although our memory and bandwidth requirements don't require this, since all this information can easily be stored on one server but, for scalability, performance, and fault tolerance, we should distribute DriverLocationHT onto multiple servers. We can distribute based on the DriverID to make the distribution completely random. Let's call the machines holding DriverLocationHT the Driver Location server. Other than storing the driver's location, each of these servers will do two things:

1. As soon as the server receives an update for a driver's location, they will broadcast that information to all the interested customers.
2. The server needs to notify the respective QuadTree server to refresh the driver's location. As discussed above, this can happen every 15 seconds.

How can we efficiently broadcast the driver's location to customers? We can have a **Push Model** where the server will push the positions to all the relevant users. We can have a dedicated Notification Service that can broadcast drivers' current location to all the interested customers. We can build our Notification service on a publisher/subscriber model. When customers open the Uber app on their cell phones, they query the server to find nearby drivers. On the server-side, before returning the list of drivers to the customer, we will subscribe the customer for all the updates from those drivers. We can maintain a list of customers (subscribers) interested in knowing the location of a driver and, whenever we have an update in

DriverLocationHT for that driver, we can broadcast the current location of the driver to all subscribed customers. This way, our system makes sure that we always show the driver's current position to the customer.

How much memory will we need to store all these subscriptions? As we have estimated above, we will have 1M daily active customers and 500K daily active drivers. On average, let's assume that five customers subscribe to one driver. Let's assume we store all this information in a hash table so that we can update it efficiently. We need to store driver and customer IDs to maintain the subscriptions. Assuming we will need 3 bytes for DriverID and 8 bytes for CustomerID, we will need 21MB of memory.

$$(500K * 3) + (500K * 5 * 8) \approx 21 \text{ MB}$$

How much bandwidth will we need to broadcast the driver's location to customers? For every active driver, we have five subscribers, so the total subscribers we have:

$$5 * 500K \Rightarrow 2.5M$$

To all these customers we need to send DriverID (3 bytes) and their location (16 bytes) every second, so, we need the following bandwidth:

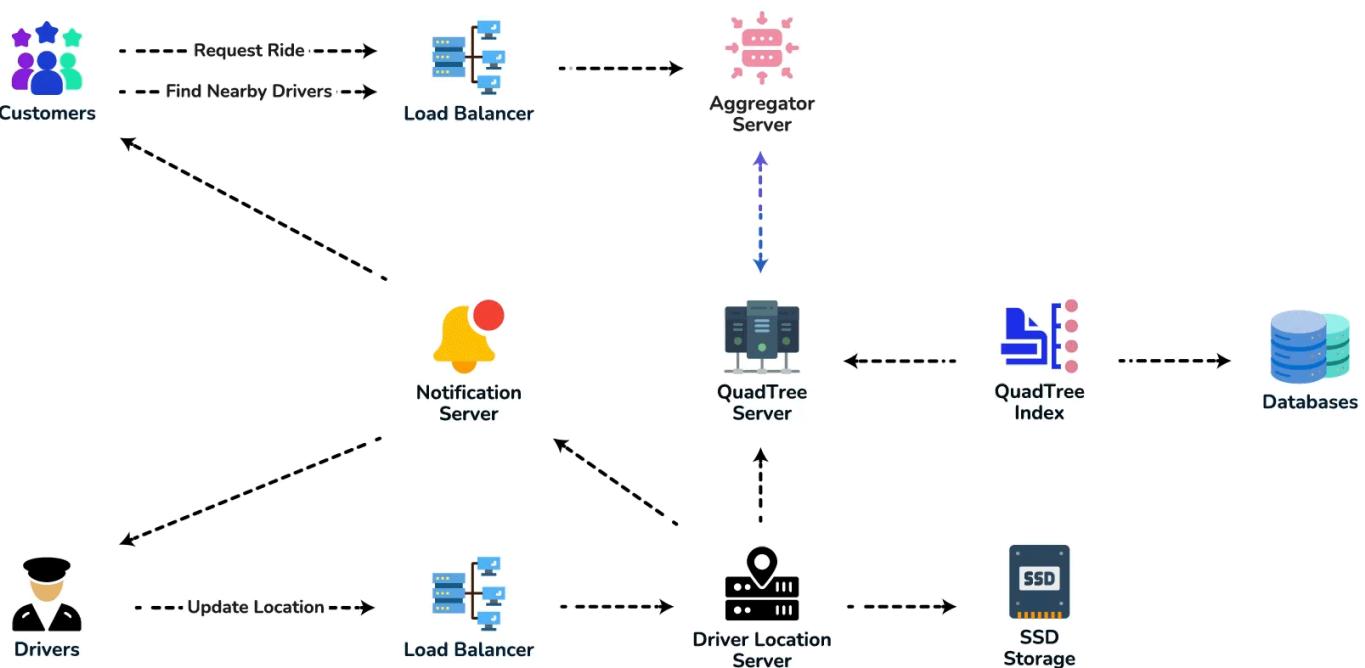
$$2.5M * 19 \text{ bytes} \Rightarrow 47.5 \text{ MB/s}$$

How can we efficiently implement the Notification service? We can either use HTTP long polling or push notifications.

How will the new publishers/drivers get added for a current customer? As we have proposed above, customers will be subscribed to nearby drivers when they open the Uber app for the first time; what will happen when a new driver enters the area the customer is looking at? To add a new customer/driver subscription dynamically, we need to keep track of the area the customer is watching. This will make our solution complicated; what if, instead of pushing this information, clients pull it from the server?

How about if clients pull information about nearby drivers from the server? Clients can send their current location, and the server will find all the nearby drivers from the QuadTree to return them to the client. Upon receiving this information, the client can update their screen to reflect the current positions of the drivers. Clients can query every five seconds to limit the number of round trips to the server. This solution looks simpler compared to the push model described above.

Do we need to repartition a grid as soon as it reaches the maximum limit? We can have a cushion to let each grid grow a little bigger beyond the limit before we decide to partition it. Let's say our grids can grow/shrink an extra 10% before we partition/merge them. This should decrease the load for a grid partition or merge on high traffic grids.



How would "Request Ride" use case work?

1. The customer will put a request for a ride.
2. One of the Aggregator servers will take the request and asks QuadTree servers to return nearby drivers.
3. The Aggregator server collects all the results and sorts them by ratings.

4. The Aggregator server will send a notification to the top (say three) drivers simultaneously, whichever driver accepts the request first will be assigned the ride. The other drivers will receive a cancellation request. If none of the three drivers respond, the Aggregator will request a ride from the next three drivers from the list.
5. Once a driver accepts a request, the customer is notified.

5. Fault Tolerance and Replication

What if a Driver Location server or Notification server dies? We would need replicas of these servers, so that if the primary dies the secondary can take control. Also, we can store this data in some persistent storage like SSDs that can provide fast IOs; this will ensure that if both primary and secondary servers die we can recover the data from the persistent storage.

6. Ranking

How about if we want to rank the search results not just by proximity but also by popularity or relevance?

How can we return top rated drivers within a given radius? Let's assume we keep track of the overall ratings of each driver in our database and QuadTree. An aggregated number can represent this popularity in our system, e.g., how many stars does a driver get out of ten? While searching for the top 10 drivers within a given radius, we can ask each partition of the QuadTree to return the top 10 drivers with a maximum rating. The aggregator server can then determine the top 10 drivers among all the drivers returned by different partitions.

7. Advanced Issues

1. How will we handle clients on slow and disconnecting networks?
2. What if a client gets disconnected when they are a part of a ride? How will we handle billing in such a scenario?
3. How about if clients pull all the information, compared to servers always pushing it?

Designing Ticketmaster

Let's design an online ticketing system that sells movie tickets like Ticketmaster or BookMyShow.

Similar Services: bookmyshow.com, ticketmaster.com

Difficulty Level: Hard

1. What is an online movie ticket booking system?

A movie ticket booking system provides its customers the ability to purchase theatre seats online. E-ticketing systems allow the customers to browse through movies currently being played and to book seats, anywhere anytime.

Designing Ticketmaster (video)

Here is a video discussing how to design Ticketmaster:

1:34:20

Designing Ticketmaster

2. Requirements and Goals of the System

Our ticket booking service should meet the following requirements:

Functional Requirements:

1. Our ticket booking service should be able to list different cities where its affiliate cinemas are located.
2. Once the user selects the city, the service should display the movies released in that particular city.

3. Once the user selects a movie, the service should display the cinemas running that movie and its available showtimes.
4. The user should be able to choose a show at a particular cinema and book their tickets.
5. The service should be able to show the user the seating arrangement of the cinema hall. The user should be able to select multiple seats according to their preference.
6. The user should be able to distinguish available seats from booked ones.
7. Users should be able to put a hold on the seats for five minutes before they make a payment to finalize the booking.
8. The user should be able to wait if there is a chance that the seats might become available, e.g., when holds by other users expire.
9. Waiting customers should be serviced in a fair, first come, first serve manner.

Non-Functional Requirements:

1. The system would need to be highly concurrent. There will be multiple booking requests for the same seat at any particular point in time. The service should handle this gracefully and fairly.
2. The core thing of the service is ticket booking, which means financial transactions. This means that the system should be secure and the database ACID compliant.

3. Some Design Considerations

1. For simplicity, let's assume our service does not require any user authentication.
2. The system will not handle partial ticket orders. Either user gets all the tickets they want or they get nothing.
3. Fairness is mandatory for the system.

4. To stop system abuse, we can restrict users from booking more than ten seats at a time.
5. We can assume that traffic would spike on popular/much-awaited movie releases and the seats would fill up pretty fast. The system should be scalable and highly available to keep up with the surge in traffic.

4. Capacity Estimation

Traffic estimates: Let's assume that our service has 3 billion page views per month and sells 10 million tickets a month.

Storage estimates: Let's assume that we have 500 cities and, on average each city has ten cinemas. If there are 2000 seats in each cinema and on average, there are two shows every day.

Let's assume each seat booking needs 50 bytes (IDs, NumberOfSeats, ShowID, MovieID, SeatNumbers, SeatStatus, Timestamp, etc.) to store in the database. We would also need to store information about movies and cinemas; let's assume it'll take 50 bytes. So, to store all the data about all shows of all cinemas of all cities for a day:

$$500 \text{ cities} * 10 \text{ cinemas} * 2000 \text{ seats} * 2 \text{ shows} * (50+50) \text{ bytes} = 2\text{GB / day}$$

To store five years of this data, we would need around 3.6TB.

5. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. The following could be the definition of the APIs to search movie shows and reserve seats.

```
SearchMovies(api_dev_key, keyword, city, lat_long, radius,  
start_datetime, end_datetime, postal_code,  
includeSpellcheck, results_per_page, sorting_order)
```

Parameters:

api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.

keyword (string): Keyword to search on.

city (string): City to filter movies by.

lat_long (string): Latitude and longitude to filter by.

radius (number): Radius of the area in which we want to search for events.

start_datetime (string): Filter movies with a starting datetime.

end_datetime (string): Filter movies with an ending datetime.

postal_code (string): Filter movies by postal code / zipcode.

includeSpellcheck (Enum: "yes" or "no"): Yes, to include spell check suggestions in the response.

results_per_page (number): Number of results to return per page. Maximum is 30.

sorting_order (string): Sorting order of the search result. Some allowable values :
'name,asc', 'name,desc', 'date,asc', 'date,desc', 'distance,asc', 'name,date,asc',
'name,date,desc', 'date,name,asc', 'date,name,desc'.

Returns: (JSON)

Here is a sample list of movies and their shows:

```
{  
  "MovieID": 1,  
  "ShowID": 1,  
  "Title": "Cars 2",  
  "Description": "About cars",
```

```
"Duration": 120,  
"Genre": "Animation",  
"Language": "English",  
"ReleaseDate": "8th Oct. 2014",  
"Country": USA,  
"StartTime": "14:00",  
"EndTime": "16:00",  
"Seats":  
[  
{  
    "Type": "Regular"  
    "Price": 14.99  
    "Status": "Almost Full"  
},  
{  
    "Type": "Premium"  
    "Price": 24.99  
    "Status": "Available"  
}  
]  
,  
{  
    "MovieID": 1,  
    "ShowID": 2,  
    "Title": "Cars 2",  
    "Description": "About cars",  
    "Duration": 120,  
    "Genre": "Animation",  
    "Language": "English",  
    "ReleaseDate": "8th Oct. 2014",  
    "Country": USA,  
    "StartTime": "16:30",  
    "EndTime": "18:30",  
    "Seats":  
[  
{
```

```

    "Type": "Regular"
    "Price": 14.99
    "Status": "Full"
  },
  {
    "Type": "Premium"
    "Price": 24.99
    "Status": "Almost Full"
  }
]
}

```

```
ReserveSeats(api_dev_key, session_id, movie_id, show_id,
seats_to_reserve[])
```

Parameters:

api_dev_key (string): same as above

session_id (string): User's session ID to track this reservation. Once the reservation time expires, user's reservation on the server will be removed using this ID.

movie_id (string): Movie to reserve.

show_id (string): Show to reserve.

seats_to_reserve (number): An array containing seat IDs to reserve.

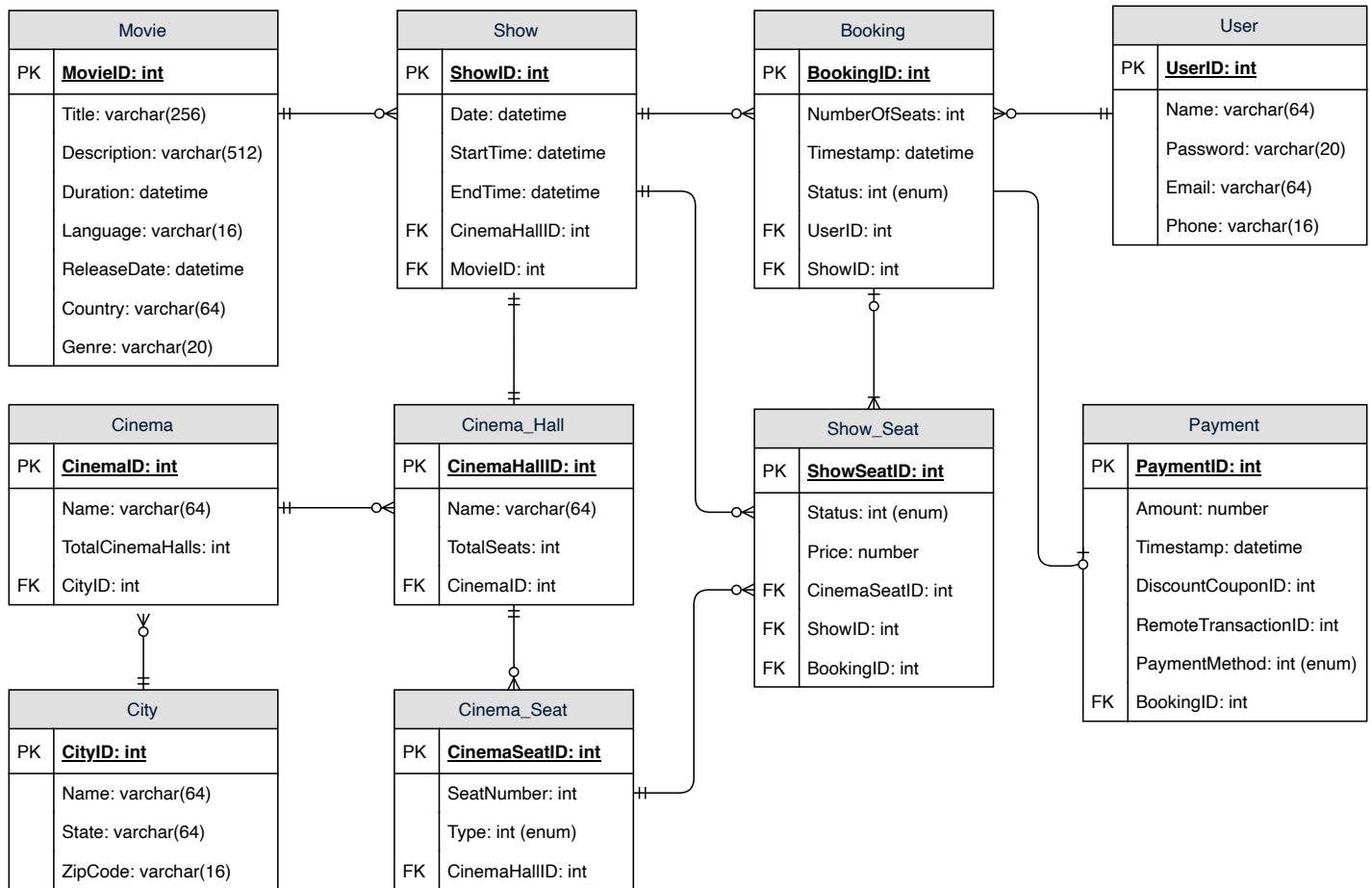
Returns: (JSON)

Returns the status of the reservation, which would be one of the following: 1) "Reservation Successful" 2) "Reservation Failed - Show Full," 3) "Reservation Failed - Retry, as other users are holding reserved seats".

6. Database Design

Here are a few observations about the data we are going to store:

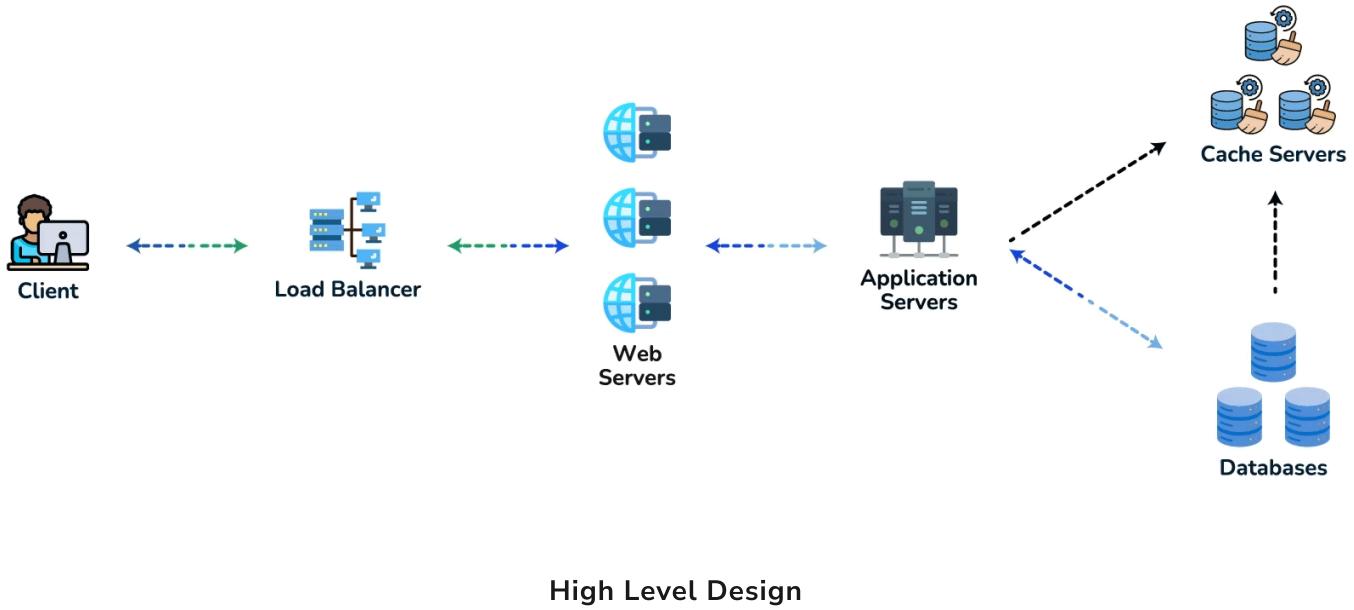
1. Each City can have multiple Cinemas.
2. Each Cinema will have multiple halls.
3. Each Movie will have many Shows and each Show will have multiple Bookings.
4. A user can have multiple bookings.



DB Schema

7. High Level Design

At a high-level, our web servers will manage users' sessions and application servers will handle all the ticket management, storing data in the databases as well as working with the cache servers to process reservations.



8. Detailed Component Design

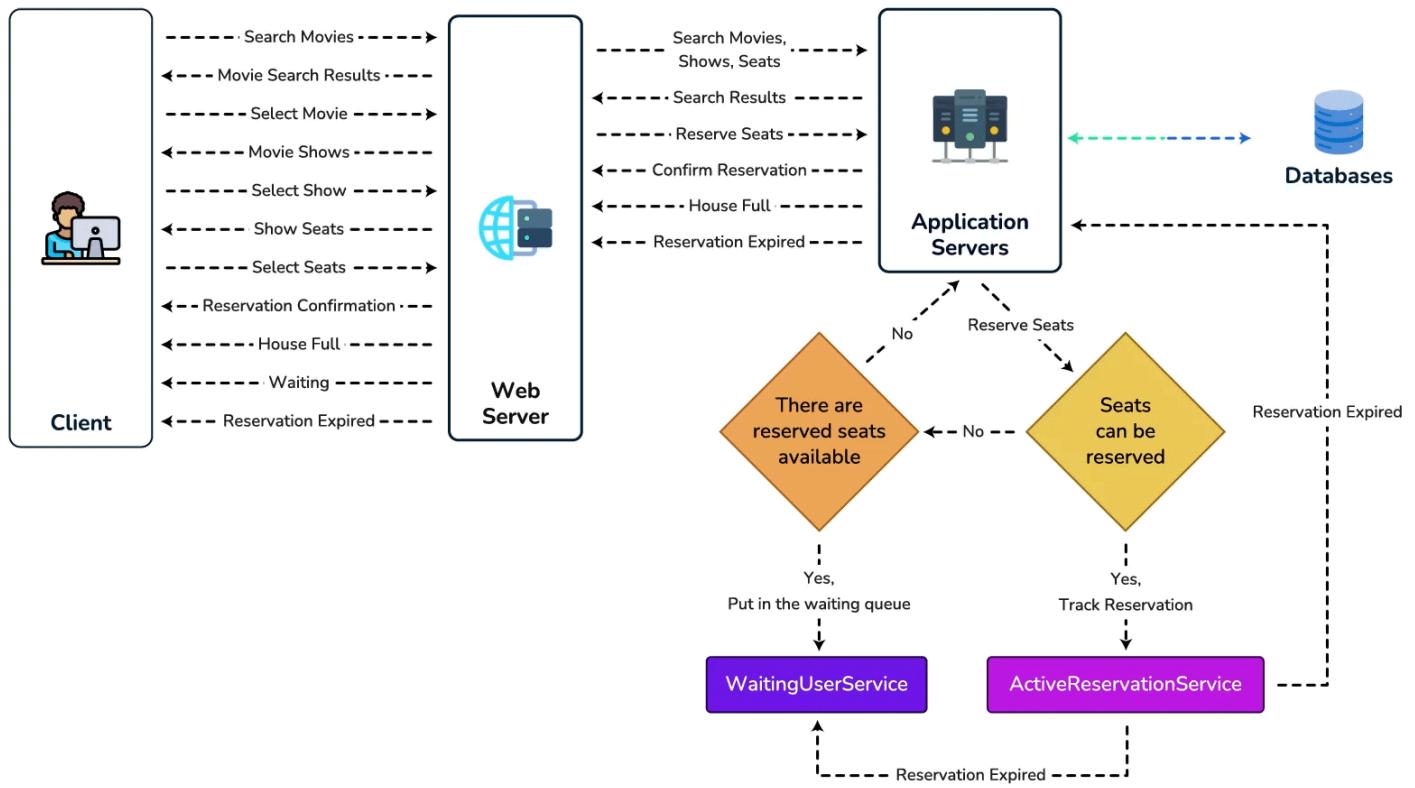
First, let's try to build our service assuming it is being served from a single server.

Ticket Booking Workflow: The following would be a typical ticket booking workflow:

1. The user searches for a movie.
2. The user selects a movie.
3. The user is shown the available shows of the movie.
4. The user selects a show.
5. The user selects the number of seats to be reserved.
6. If the required number of seats are available, the user is shown a map of the theater to select seats. If not, the user is taken to 'step 8' below.
7. Once the user selects the seat, the system will try to reserve those selected seats.
8. If seats can't be reserved, we have the following options:

- Show is full; the user is shown the error message.
- The seats the user wants to reserve are no longer available, but there are other seats available, so the user is taken back to the theater map to choose different seats.
- There are no seats available to reserve, but all the seats are not booked yet, as there are some seats that other users are holding in the reservation pool and have not booked yet. The user will be taken to a waiting page where they can wait until the required seats get freed from the reservation pool. This waiting could result in the following options:
 - If the required number of seats become available, the user is taken to the theater map page where they can choose seats.
 - While waiting, if all seats get booked or there are fewer seats in the reservation pool than the user intends to book, the user is shown the error message.
 - User cancels the waiting and is taken back to the movie search page.
 - At maximum, a user can wait one hour, after that user's session gets expired and the user is taken back to the movie search page.

9. If seats are reserved successfully, the user has five minutes to pay for the reservation. After payment, the booking is marked complete. If the user is not able to pay within five minutes, all their reserved seats are freed to become available to other users.



Detailed Workflow

How would the server keep track of all the active reservations that haven't been booked yet? And how would the server keep track of all the waiting customers?

We need two daemon services, one to keep track of all active reservations and remove any expired reservation from the system; let's call it **ActiveReservationService**. The other service would be keeping track of all the waiting user requests and, as soon as the required number of seats become available, it will notify the (the longest waiting) user to choose the seats; let's call it **WaitingUserService**.

a. ActiveReservationsService

We can keep all the reservations of a 'show' in memory in a data structure similar to [Linked HashMap](#) or a [TreeMap](#) in addition to keeping all the data in the database. We will need a linked HashMap kind of data structure that allows us to

jump to any reservation to remove it when the booking is complete. Also, since we will have expiry time associated with each reservation, the head of the HashMap will always point to the oldest reservation record so that the reservation can be expired when the timeout is reached.

To store every reservation for every show, we can have a HashTable where the ‘key’ would be ‘ShowID’, and the ‘value’ would be the Linked HashMap containing ‘BookingID’ and creation ‘Timestamp’.

In the database, we will store the reservation in the ‘Booking’ table and the expiry time will be in the Timestamp column. The ‘Status’ field will have a value of ‘Reserved (1)’ and, as soon as a booking is complete, the system will update the ‘Status’ to ‘Booked (2)’ and remove the reservation record from the Linked HashMap of the relevant show. When the reservation is expired, we can either remove it from the Booking table or mark it ‘Expired (3)’ in addition to removing it from memory.

ActiveReservationsService will also work with the external financial service to process user payments. Whenever a booking is completed, or a reservation gets expired, WaitingUserService will get a signal so that any waiting customer can be served.

Key : Value

E.g., ShowID : LinkedHashMap< BookingID, TimeStamp >

123 : { (1, 1499818500), (2, 1499818700), (3, 1499818800) }

ActiveReservationsService keeping track of all active reservations

b. WaitingUserService

Just like ActiveReservationsService, we can keep all the waiting users of a show in memory in a Linked HashMap or a TreeMap. We need a data structure similar to Linked HashMap so that we can jump to any user to remove them from the HashMap when the user cancels their request. Also, since we are serving in a first-come-first-serve manner, the head of the Linked HashMap would always be pointing to the longest waiting user, so that whenever seats become available, we can serve users in a fair manner.

We will have a HashTable to store all the waiting users for every Show. The ‘key’ would be 'ShowID', and the ‘value’ would be a Linked HashMap containing ‘UserIDs’ and their wait-start-time.

Clients can use [Long Polling](#) for keeping themselves updated for their reservation status. Whenever seats become available, the server can use this request to notify the user.

Reservation Expiration

On the server, ActiveReservationsService keeps track of expiry (based on reservation time) of active reservations. As the client will be shown a timer (for the expiration time), which could be a little out of sync with the server, we can add a buffer of five seconds on the server to safeguard from a broken experience, such that the client never times out after the server, preventing a successful purchase.

9. Concurrency

How to handle concurrency, such that no two users are able to book the same seat. We can use transactions in SQL databases to avoid any clashes. For example, if we are using an SQL server we can utilize [Transaction Isolation Levels](#) to lock the rows before we can update them. Here is the sample code:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
  
BEGIN TRANSACTION;  
  
    -- Suppose we intend to reserve three seats (IDs: 54,  
    55, 56) for ShowID=99  
    Select * From Show_Seat where ShowID=99 && ShowSeatID in  
    (54, 55, 56) && Status=0 -- free  
  
    -- if the number of rows returned by the above statement  
    is three, we can update to  
    -- return success otherwise return failure to the user.  
    update Show_Seat ...  
    update Booking ...  
  
COMMIT TRANSACTION;
```

'Serializable' is the highest isolation level and guarantees safety from **Dirty**, **Nonrepeatable**, and **Phantoms** reads. One thing to note here; within a transaction, if we read rows, we get a write lock on them so that they can't be updated by anyone else.

Once the above database transaction is successful, we can start tracking the reservation in ActiveReservationService.

10. Fault Tolerance

What happens when ActiveReservationsService or WaitingUsersService crashes?

Whenever ActiveReservationsService crashes, we can read all the active reservations from the 'Booking' table. Remember that we keep the 'Status' column as 'Reserved (1)' until a reservation gets booked. Another option is to have a

primary-secondary configuration so that, when the primary crashes, the secondary can take over. We are not storing the waiting users in the database, so, when WaitingUserService crashes, we don't have any means to recover that data unless we have a primary-secondary setup.

Similarly, we'll have a primary-secondary setup for databases to make them fault-tolerant.

11. Data Partitioning

Database partitioning: If we partition by 'MovieID', then all the Shows of a movie will be on a single server. For a very hot movie, this could cause a lot of load on that server. A better approach would be to partition based on ShowID; this way, the load gets distributed among different servers.

ActiveReservationService and WaitingUserService partitioning: Our web servers will manage all the active users' sessions and handle all the communication with the users. We can use the Consistent Hashing to allocate application servers for both ActiveReservationService and WaitingUserService based upon the 'ShowID'. This way, all reservations and waiting users of a particular show will be handled by a certain set of servers. Let's assume for load balancing our 'Consistent Hashing' allocates three servers for any Show, so whenever a reservation is expired, the server holding that reservation will do the following things:

1. Update the database to remove the Booking (or mark it expired) and update the seats' Status in 'Show_Seats' table.
2. Remove the reservation from the Linked HashMap.
3. Notify the user that their reservation has expired.
4. Broadcast a message to all WaitingUserService servers that are holding waiting users of that Show to figure out the longest waiting user. Consistent Hashing scheme will tell what servers are holding these users.

5. Send a message to the WaitingUserService server holding the longest waiting user to process their request if required seats have become available.

Whenever a reservation is successful, following things will happen:

1. The server holding that reservation sends a message to all servers holding the waiting users of that Show, so that those servers can expire all the waiting users that need more seats than the available seats.
2. Upon receiving the above message, all servers holding the waiting users will query the database to find how many free seats are available now. A database cache would greatly help here to run this query only once.
3. Expire all waiting users who want to reserve more seats than the available seats. For this, WaitingUserService has to iterate through the Linked HashMap of all the waiting users.

Additional Resources

Here are some useful links for further reading (*we have discussed most of these systems in Grokking the Advanced System Design Interview'*):

1. **Dynamo** - Highly Available Key-value Store
2. **Kafka** - A Distributed Messaging System for Log Processing
3. **Consistent Hashing** - Data distribution with minimal disruption during scaling.
4. **Paxos** - Protocol for distributed consensus
5. **Bigtable** - A Distributed Storage System for Structured Data
6. **Gossip protocol** - For failure detection and more.
7. **Chubby** - Lock service for loosely-coupled distributed systems
8. **ZooKeeper** - Wait-free coordination for Internet-scale systems
9. **MapReduce** - Simplified Data Processing on Large Clusters
10. **HDFS** - A Distributed File System
11. **Cassandra** - A distributed, decentralized, scalable, and highly available NoSQL database.

Importance of Discussing Trade-offs

Presenting trade-offs in a system design interview is highly significant for several reasons as it demonstrates a depth of understanding and maturity in design. Here's why discussing trade-offs is important:

1. Shows Comprehensive Understanding

- **Balanced Perspective:** Discussing trade-offs indicates that you understand there are multiple ways to approach a problem, each with its advantages and disadvantages.
- **Depth of Knowledge:** It shows that you're aware of different technologies, architectures, and methodologies, and understand how choices impact a system's behavior and performance.

2. Highlights Critical Thinking and Decision-Making Skills

- **Analytical Approach:** By evaluating trade-offs, you demonstrate an ability to analyze various aspects of a system, considering factors like scalability, performance, maintainability, and cost.
- **Informed Decision-Making:** It shows that your design decisions are thoughtful and informed, rather than arbitrary.

3. Demonstrates Real-World Problem-Solving Skills

- **Practical Solutions:** In the real world, every system design decision comes with trade-offs. Demonstrating this understanding aligns with practical, real-world scenarios where perfect solutions rarely exist.

- **Prioritization:** Discussing trade-offs shows that you can prioritize certain aspects over others based on the requirements and constraints, which is a critical skill in system design.

4. Reveals Awareness of Business and Technical Constraints

- **Business Acumen:** Understanding trade-offs indicates that you're considering not just the technical but also the business implications of your design choices (like cost implications, time to market).
- **Adaptability:** It shows you can adapt your design to meet different priorities and constraints, which is key in a dynamic business environment.

5. Facilitates Better Team Collaboration and Communication

- **Communication Skills:** Clearly articulating trade-offs is a vital part of effective technical communication, crucial for collaborating with team members and stakeholders.
- **Expectation Management:** It helps in setting realistic expectations and preparing for potential challenges in implementation.

6. Prepares for Scalability and Future Growth

- **Long-term Vision:** Discussing trade-offs shows that you're thinking about how the system will evolve over time and how early decisions might impact future changes or scalability.

7. Shows Maturity and Experience

- **Professional Maturity:** Recognizing that every decision has pros and cons reflects professional maturity and experience in handling complex projects.
- **Learning from Experience:** It can also indicate that you've learned from past experiences, applying these lessons to make better design choices.

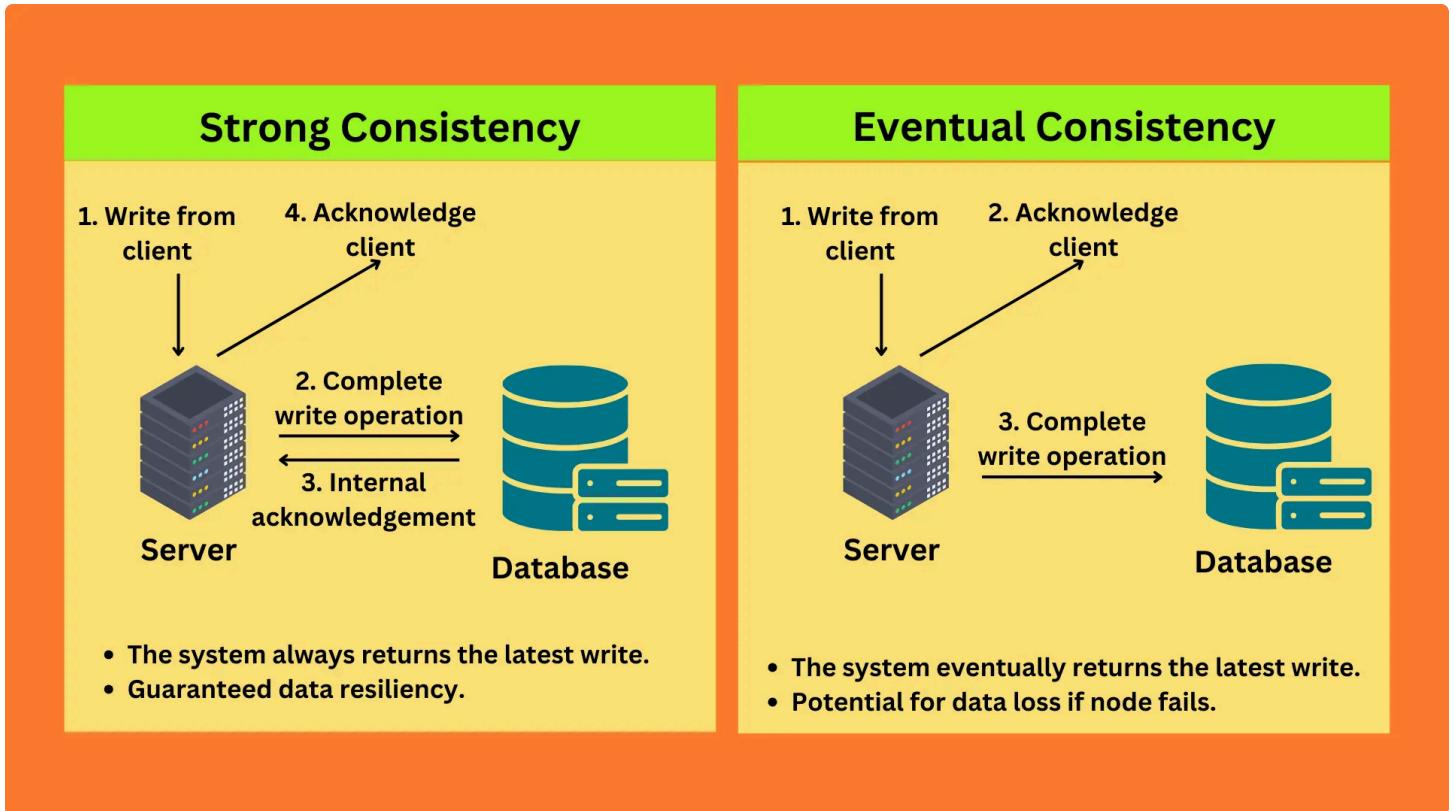
Conclusion

In system design interviews, discussing trade-offs is not just about acknowledging that they exist, but about demonstrating a well-rounded and mature approach to system design. It reflects a candidate's ability to make informed decisions, a deep understanding of technical principles, and an appreciation of the broader business context.

Up next, let's discuss some essential trade-offs that you can explore during a system design interview.

Strong vs Eventual Consistency

Strong consistency and eventual consistency are two different models used to manage data consistency in distributed systems, particularly in database systems and data storage services.



Strong vs Eventual Consistency

Strong Consistency

- **Definition:** In a strong consistency model, a system guarantees that once a write operation is completed, any subsequent read operation will reflect that write. In other words, all users see the same data at the same time.
- **Characteristics:**

- **Immediate Consistency:** Ensures that all clients see the same data as soon as it's updated or written.
- **Read-Write Synchronization:** Read operations might have to wait for a write operation to complete to ensure consistent data is returned.
- **Example:** Consider a banking system where a user transfers money between accounts. With strong consistency, as soon as the transfer is processed, any query on the account balance will reflect the transfer. There's no period where different users see different balances.
- **Pros:**
 - **Data Reliability:** Ensures high data integrity and reliability.
 - **Simplicity for Users:** Easier for users to understand and work with.
- **Cons:**
 - **Potential Latency:** Can introduce latency, especially in distributed systems, as the system needs to ensure data is consistent across all nodes before proceeding.
 - **Scalability Challenges:** More challenging to scale, as ensuring immediate consistency across distributed nodes can be complex.

Eventual Consistency

- **Definition:** In an eventual consistency model, the system guarantees that if no new updates are made to a given piece of data, eventually all accesses will return the last updated value. However, for a time after a write operation, reads might return an older value.
- **Characteristics:**
 - **Delayed Consistency:** The system eventually becomes consistent but allows for periods where different users might see different data.
 - **Higher Performance:** Typically offers higher performance and availability than strong consistency.

- **Example:** A social media platform's distributed database that uses eventual consistency might show different users different versions of a post's like count for a short period after it's updated. Over time, all users will see the correct count.
- **Pros:**
 - **Scalability:** Easier to scale across multiple nodes, as it doesn't require immediate consistency across all nodes.
 - **High Availability:** Offers higher availability, even in the presence of network partitions.
- **Cons:**
 - **Data Inconsistency Window:** There's a window of time where data might be inconsistent.
 - **Complexity for Users:** Users might be confused or make incorrect decisions based on outdated information.

Key Differences

- **Consistency Guarantee:** Strong consistency ensures that all users see the same data at the same time, while eventual consistency allows for a period where data can be inconsistent but eventually becomes uniform.
- **Performance vs. Consistency:** Strong consistency prioritizes consistency which can affect performance and scalability. Eventual consistency prioritizes performance and availability, with a trade-off in immediate data consistency.

Conclusion

The choice between strong and eventual consistency depends on the specific requirements of the application. Applications that require strict data accuracy (like

financial systems) typically opt for strong consistency, while applications that can tolerate some temporary inconsistency for better performance and availability (like social media feeds) might choose eventual consistency.

Latency vs Throughput

Latency and throughput are two critical performance metrics in software systems, but they measure different aspects of the system's performance.

Latency

- **Definition:** Latency is the time it takes for a piece of data to travel from its source to its destination. In other words, it's the delay between the initiation of a request and the receipt of the response.
- **Characteristics:**
 - Measured in units of time (milliseconds, seconds).
 - Lower latency indicates a more responsive system.
- **Impact:** Latency is particularly important in scenarios where real-time or near-real-time interaction or data transfer is crucial, such as in online gaming, video conferencing, or high-frequency trading.
- **Example:** If you click a link on a website, the latency would be the time it takes from the moment you click the link to when the page starts loading.

Throughput

- **Definition:** Throughput refers to the amount of data transferred over a network or processed by a system in a given amount of time. It's a measure of how much work or data processing is completed over a specific period.
- **Characteristics:**
 - Measured in units of data per time (e.g., Mbps - Megabits per second).
 - Higher throughput indicates a higher data processing capacity.

- **Impact:** Throughput is a critical measure in systems where the volume of data processing is significant, such as in data backup systems, bulk data processing, or video streaming services.
- **Example:** In a video streaming service, throughput would be the rate at which video data is transferred from the server to your device.

Latency vs Throughput - Key Differences

- **Focus:** Latency is about the delay or time, focusing on speed. Throughput is about the volume of work or data, focusing on capacity.
- **Influence on User Experience:** High latency can lead to a sluggish user experience, while low throughput can result in slow data transfer rates, affecting the efficiency of data-intensive operations.
- **Trade-offs:** In some systems, improving throughput may increase latency, and vice versa. For instance, sending data in larger batches may improve throughput but could also result in higher latency.

Improving latency and throughput often involves different strategies, as optimizing for one can sometimes impact the other. However, there are several techniques that can enhance both metrics:

How to Improve Latency

1. **Optimize Network Routes:** Use Content Delivery Networks (CDNs) to serve content from locations geographically closer to the user. This reduces the distance data must travel, decreasing latency.
2. **Upgrade Hardware:** Faster processors, more memory, and quicker storage (like SSDs) can reduce processing time.

3. **Use Faster Communication Protocols:** Protocols like HTTP/2 can reduce latency through features like multiplexing and header compression.
4. **Database Optimization:** Use indexing, optimized queries, and in-memory databases to reduce data access and processing time.
5. **Load Balancing:** Distribute incoming requests efficiently among servers to prevent any single server from becoming a bottleneck.
6. **Code Optimization:** Optimize algorithms and remove unnecessary computations to speed up execution.
7. **Minimize External Calls:** Reduce the number of API calls or external dependencies in your application.

How to Improve Throughput

1. **Scale Horizontally:** Add more servers to handle increased load. This is often more effective than vertical scaling (upgrading the capacity of a single server).
2. **Implement Caching:** Cache frequently accessed data in memory to reduce the need for repeated data processing.
3. **Parallel Processing:** Use parallel computing techniques where tasks are divided and processed simultaneously.
4. **Batch Processing:** For non-real-time data, processing in batches can be more efficient than processing each item individually.
5. **Optimize Database Performance:** Ensure efficient data storage and retrieval. This may include techniques like partitioning and sharding.
6. **Asynchronous Processing:** Use asynchronous processes for tasks that don't need to be completed immediately.
7. **Network Bandwidth:** Increase the network bandwidth to accommodate higher data transfer rates.

Conclusion

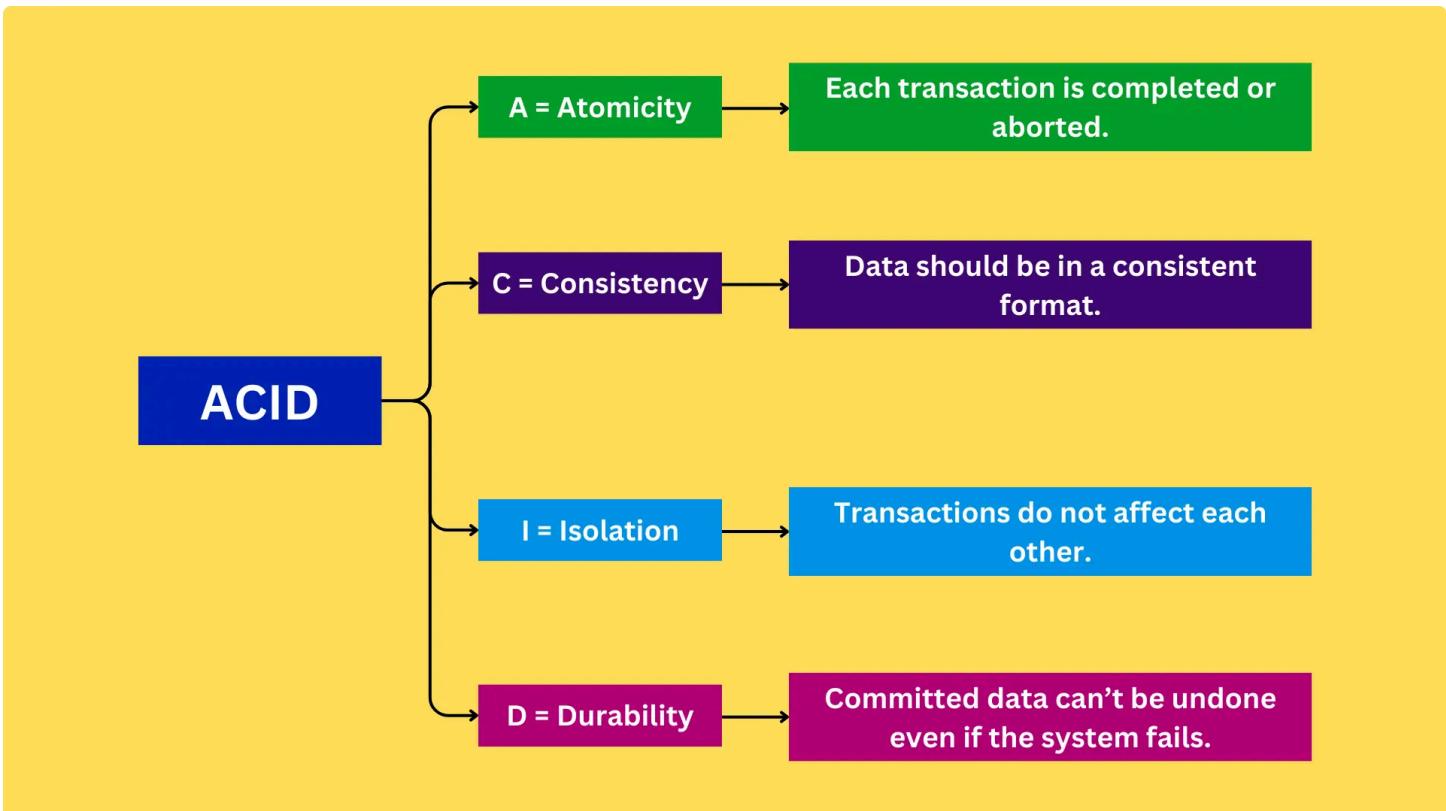
Low latency is crucial for applications requiring fast response times, while high throughput is vital for systems dealing with large volumes of data.

ACID vs BASE Properties in Databases

ACID and BASE are two sets of properties that represent different approaches to handling transactions in database systems. They reflect trade-offs between consistency, availability, and partition tolerance, especially in distributed databases.

ACID Properties

- **Definition:** ACID stands for Atomicity, Consistency, Isolation, and Durability. It's a set of properties that guarantee reliable processing of database transactions.
- **Components:**
 - **Atomicity:** Ensures that a transaction is either fully completed or not executed at all.
 - **Consistency:** Guarantees that a transaction brings the database from one valid state to another.
 - **Isolation:** Ensures that concurrent transactions do not interfere with each other.
 - **Durability:** Once a transaction is committed, it remains so, even in the event of a system failure.
- **Example:** Consider a bank transfer from one account to another. The transfer operation (debit from one account and credit to another) must be atomic, maintain the consistency of total funds, be isolated from other transactions, and changes must be permanent.
- **Use Cases:** Ideal for systems requiring high reliability and data integrity, like banking or financial systems.

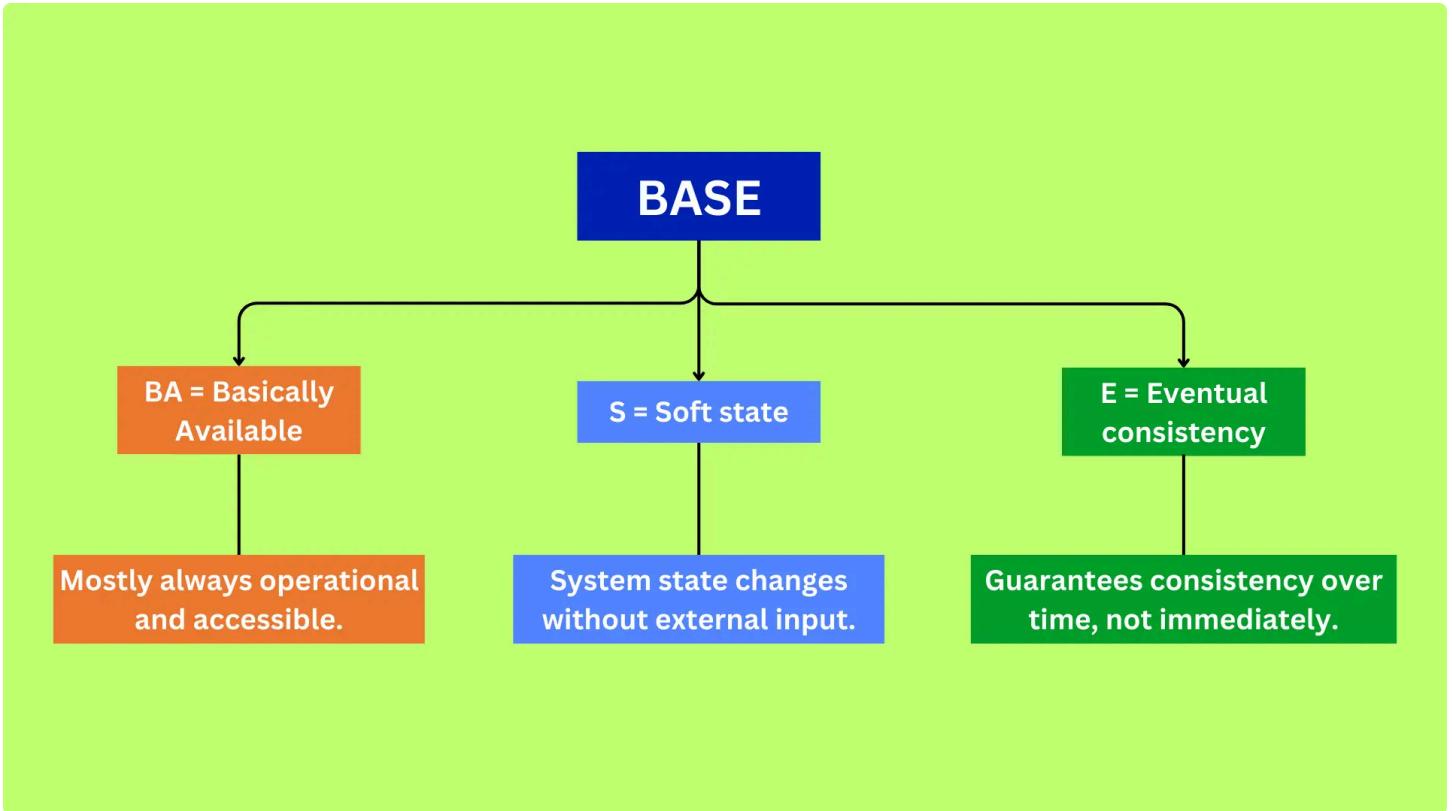


ACID

BASE Properties

- **Definition:** BASE stands for Basically Available, Soft state, and Eventual consistency. It's an alternative to ACID in distributed systems, favoring availability over consistency.
- **Components:**
 - **Basically Available:** Indicates that the system is available most of the time.
 - **Soft State:** The state of the system may change over time, even without input.
 - **Eventual Consistency:** The system will eventually become consistent, given enough time.

- **Example:** A social media platform using a BASE model may show different users different counts of likes on a post for a short period but eventually, all users will see the correct count.
- **Use Cases:** Suitable for distributed systems where availability and partition tolerance are more critical than immediate consistency, like social networks or e-commerce product catalogs.



BASE

Key Differences

- **Consistency and Availability:** ACID prioritizes consistency and reliability of each transaction, while BASE prioritizes system availability and partition tolerance, allowing for some level of data inconsistency.

- **System Design:** ACID is generally used in traditional relational databases, while BASE is often associated with NoSQL and distributed databases.
- **Use Case Alignment:** ACID is well-suited for applications requiring strong data integrity, whereas BASE is better for large-scale applications needing high availability and scalability.

Conclusion

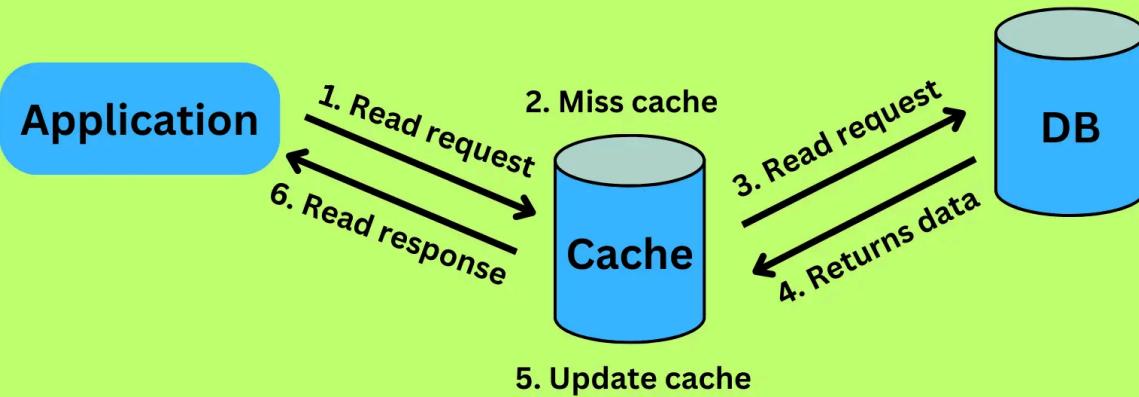
ACID is critical for systems where transactions must be reliable and consistent, while BASE is beneficial in environments where high availability and scalability are necessary, and some degree of data inconsistency is acceptable.

Read-Through vs Write-Through Cache

Read-through and write-through caching are two caching strategies used to manage how data is synchronized between a cache and a primary storage system. They play crucial roles in system performance optimization, especially in applications where data access speed is critical.

Read-Through Cache

Read-Through Cache



Read-Through Cache

- **Definition:** In a read-through cache, data is loaded into the cache on demand, typically when a read request occurs for data that is not already in the cache.
- **Process:**

- When a read request is made, the cache first checks if the data is available (cache hit).
 - If the data is not in the cache (cache miss), the cache system reads the data from the primary storage, stores it in the cache, and then returns the data to the client.
 - Subsequent read requests for the same data will be served directly from the cache until the data expires or is evicted.
- Pros:
 - **Data Consistency:** Ensures consistency between the cache and the primary storage.
 - **Reduces Load on Primary Storage:** Frequent read operations are offloaded from the primary storage.
 - Cons:
 - **Latency on Cache Miss:** Initial read requests (cache misses) incur latency due to data fetching from the primary storage.

Read-Through Cache Example: Online Product Catalog

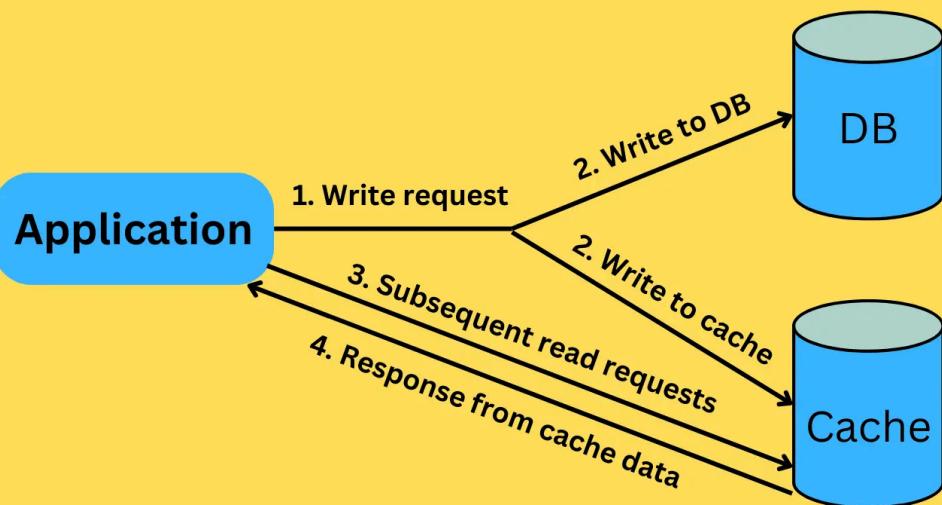
- **Scenario:** Imagine an e-commerce website with an extensive online product catalog.
- **Read-Through Process:**
 - **Cache Miss:** When a customer searches for a product that is not currently in the cache, the system experiences a cache miss.
 - **Fetching and Caching:** The system then fetches the product details from the primary database (like a SQL database) and stores this information in the cache.
 - **Subsequent Requests:** The next time any customer searches for the same product, the system delivers the product information directly from the cache, significantly faster than querying the primary database.

- Benefits in this Scenario:

- **Reduced Database Load:** Frequent queries for popular products are served from the cache, reducing the load on the primary database.
- **Improved Read Performance:** After initial caching, product information retrieval is much faster.

Write-Through Cache

Write-Through Cache



Write-Through Cache

- **Definition:** In a write-through cache, data is written simultaneously to the cache and the primary storage system. This approach ensures that the cache always contains the most recent data.
- **Process:**
 - When a write request is made, the data is first written to the cache.
 - Simultaneously, the same data is written to the primary storage.

- Read requests can then be served from the cache, which contains the up-to-date data.
- Pros:
 - **Data Consistency:** Provides strong consistency between the cache and the primary storage.
 - **No Data Loss on Crash:** Since data is written to the primary storage, there's no risk of data loss if the cache fails.
- Cons:
 - **Latency on Write Operations:** Each write operation incurs latency as it requires simultaneous writing to both the cache and the primary storage.
 - **Higher Load on Primary Storage:** Every write request impacts the primary storage.

Write-Through Cache Example: Banking System Transaction

- Scenario: Consider a banking system processing financial transactions.
- Write-Through Process:
 - **Transaction Execution:** When a user makes a transaction, such as a deposit, the transaction details are written to the cache.
 - **Simultaneous Database Write:** Simultaneously, the transaction is also recorded in the primary database.
 - **Consistent Data:** This ensures that the cached data is always up-to-date with the database. If the user immediately checks their balance, the updated balance is already in the cache for fast retrieval.
- Benefits in this Scenario:
 - **Data Integrity:** Crucial in banking, as it ensures that the cache and the primary database are always synchronized, reducing the risk of discrepancies.

- **Reliability:** In the event of a cache system failure, the data is safe in the primary database.

Key Differences

- In the **Read-Through Cache** (Product Catalog), the emphasis is on efficiently loading and serving read-heavy data after the initial request, which is ideal for data that is read frequently but updated less often.
- In the **Write-Through Cache** (Banking System), the focus is on maintaining high data integrity and consistency between the cache and the database, which is essential for transactional data where every write is critical.
- **Data Synchronization Point:** Read-through caching synchronizes data at the point of reading, while write-through caching synchronizes data at the point of writing.
- **Performance Impact:** Read-through caching improves read performance after the initial load, whereas write-through caching ensures write reliability but may have slower write performance.
- **Use Case Alignment:** Read-through is ideal for read-heavy workloads with infrequent data updates, whereas write-through is suitable for environments where data integrity and consistency are crucial, especially for write operations.

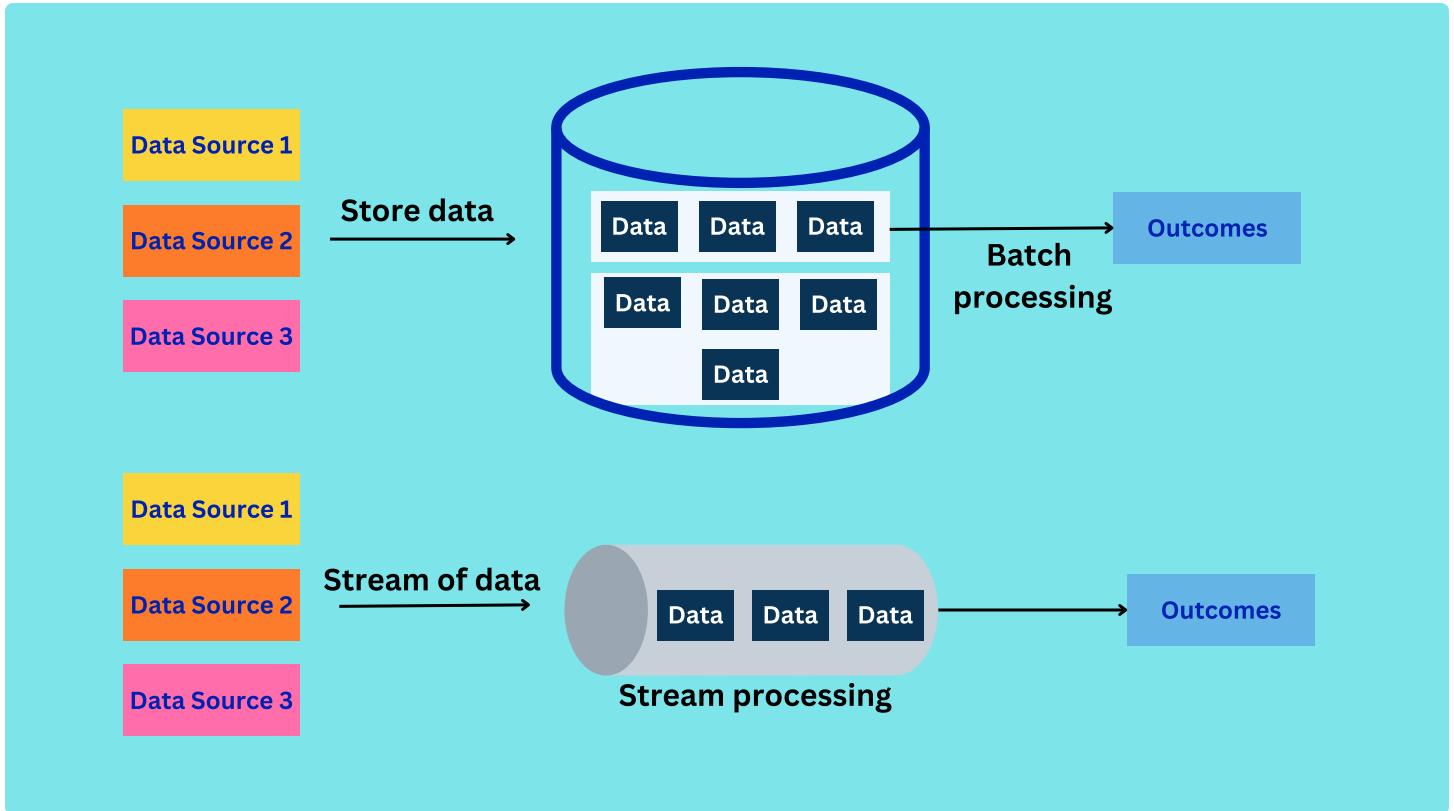
Conclusion

Read-through caching is optimal for scenarios where read performance is crucial and the data can be loaded into the cache on the first read request. Write-through caching is suited for applications where data integrity and consistency on write operations are paramount. Both strategies enhance performance but in different

aspects of data handling – read-through for read efficiency, and write-through for reliable writes.

Batch Processing vs Stream Processing

Batch processing and stream processing are two methods used for processing large volumes of data, each suited for different scenarios and data processing needs.



Batch Processing vs Stream Processing

Batch Processing

- **Definition:** Batch processing refers to processing data in large, discrete blocks (batches) at scheduled intervals or after accumulating a certain amount of data.
- **Characteristics:**
 - **Delayed Processing:** Data is collected over a period and processed all at once.
 - **High Throughput:** Efficient for processing large volumes of data where immediate action is not necessary.

- **Example:** Payroll processing in a company. Salary calculations are done at the end of each pay period (e.g., monthly). All employee data over the month is processed in one large batch to calculate salaries, taxes, and other deductions.
- **Pros:**
 - **Resource Efficient:** Can be more resource-efficient as the system can optimize for large data volumes.
 - **Simplicity:** Often simpler to implement and maintain than stream processing systems.
- **Cons:**
 - **Delay in Insights:** Not suitable for scenarios requiring real-time data processing and action.
 - **Inflexibility:** Less flexible in handling real-time data or immediate changes.

Stream Processing

- **Definition:** Stream processing involves continuously processing data in real-time as it arrives.
- **Characteristics:**
 - **Immediate Processing:** Data is processed immediately as it is generated or received.
 - **Suitable for Real-Time Applications:** Ideal for applications that require instantaneous data processing and decision-making.
- **Example:** Fraud detection in credit card transactions. Each transaction is immediately analyzed in real-time for suspicious patterns. If a transaction is flagged as fraudulent, the system can trigger an alert and take action immediately.
- **Pros:**
 - **Real-Time Analysis:** Enables immediate insights and actions.

- **Dynamic Data Handling:** More adaptable to changing data and conditions.
- **Cons:**
 - **Complexity:** Generally more complex to implement and manage than batch processing.
 - **Resource Intensive:** Can require significant resources to process data as it streams.

Key Differences

- **Data Handling:** Batch processing handles data in large chunks after accumulating it over time, while stream processing handles data continuously and in real-time.
- **Timeliness:** Batch processing is suited for scenarios where there's no immediate need for data processing, whereas stream processing is used when immediate action is required based on the incoming data.
- **Complexity and Resources:** Stream processing is generally more complex and resource-intensive, catering to real-time data, compared to the more straightforward and scheduled nature of batch processing.

Conclusion

The choice between batch and stream processing depends on specific application requirements. Batch processing is suitable for large-scale data processing tasks that don't require immediate action, like financial reporting. Stream processing is essential for real-time applications, like monitoring systems or real-time analytics, where immediate data processing and quick decision-making are crucial.

Load Balancer vs. API Gateway

Load Balancer and API Gateway are two crucial components in modern web architectures, often used to manage incoming traffic and requests to web applications. While they have some overlapping functionalities, their primary purposes and use cases are distinct.

Load Balancer

- **Purpose:** A Load Balancer is primarily used to distribute network or application traffic across multiple servers. This distribution helps to optimize resource use, maximize throughput, reduce response time, and ensure reliability.
- **How It Works:** It accepts incoming requests and then routes them to one of several backend servers based on factors like the number of current connections, server response times, or server health.
- **Types:** There are different types of load balancers, such as hardware-based or software-based, and they can operate at various layers of the OSI model (Layer 4 - transport level or Layer 7 - application level).

Example of Load Balancer:

Imagine an e-commerce website experiencing high volumes of traffic. A load balancer sits in front of the website's servers and evenly distributes incoming user requests to prevent any single server from becoming overloaded. This setup increases the website's capacity and reliability, ensuring all users have a smooth experience.

API Gateway

- **Purpose:** An API Gateway is an API management tool that sits between a client and a collection of backend services. It acts as a reverse proxy to route requests, simplify the API, and aggregate the results from various services.
- **Functionality:** The API Gateway can handle a variety of tasks, including request routing, API composition, rate limiting, authentication, and authorization.
- **Usage:** Commonly used in microservices architectures to provide a unified interface to a set of microservices, making it easier for clients to consume the services.

Example of API Gateway:

Consider a mobile banking application that needs to interact with different services like account details, transaction history, and currency exchange rates. An API Gateway sits between the app and these services. When the app requests user account information, the Gateway routes this request to the appropriate service, handles authentication, aggregates data from different services if needed, and returns a consolidated response to the app.

Key Differences:

- **Focus:** Load balancers are focused on distributing traffic to prevent overloading servers and ensure high availability and redundancy. API Gateways are more about providing a central point for managing, securing, and routing API calls.
- **Functionality:** While both can route requests, the API Gateway offers more functionalities like API transformation, composition, and security.

Is it Possible to Use a Load Balancer and an API Gateway Together?

Yes, you can use a Load Balancer and an API Gateway together in a system architecture, and they often complement each other in managing traffic and providing efficient service delivery. The typical arrangement is to place the Load Balancer in front of the API Gateway, but the actual setup can vary based on specific requirements and architecture decisions. Here's how they can work together:

Load Balancer Before API Gateway

- **Most Common Setup:** The Load Balancer is placed in front of the API Gateway. This is the typical configuration in many architectures.
- **Functionality:** The Load Balancer distributes incoming traffic across multiple instances of the API Gateway, ensuring that no single gateway instance becomes a bottleneck.
- **Benefits:**
 - **High Availability:** This setup enhances the availability and reliability of the API Gateway.
 - **Scalability:** Facilitates horizontal scaling of API Gateway instances.
- **Example:** In a cloud-based microservices architecture, external traffic first hits the Load Balancer, which then routes requests to one of the several API Gateway instances. The chosen API Gateway instance then processes the request, communicates with the appropriate microservices, and returns the response.

Load Balancer After API Gateway

- **Alternative Configuration:** In some cases, the API Gateway can be placed in front of the Load Balancer, especially when the Load Balancer is used to distribute traffic to various microservices or backend services.

- **Functionality:** The API Gateway first processes and routes the request to an internal Load Balancer, which then distributes the request to the appropriate service instances.
- **Use Case:** Useful when different services behind the API Gateway require their own load balancing logic.

Combination of Both

- **Hybrid Approach:** Some architectures might have Load Balancers at both ends – before and after the API Gateway.
- **Reasoning:** External traffic is first balanced across API Gateway instances for initial processing (authentication, rate limiting, etc.), and then further balanced among microservices or backend services.

Conclusion:

In a complex web architecture:

- A **Load Balancer** would be used to distribute incoming traffic across multiple servers or services, enhancing performance and reliability.
- An **API Gateway** would be the entry point for clients to interact with your backend APIs or microservices, providing a unified interface, handling various cross-cutting concerns, and reducing the complexity for the client applications.

In many real-world architectures, both of these components work together, where the Load Balancer effectively manages traffic across multiple instances of API Gateways or directly to services, depending on the setup.

API Gateway vs Direct Service Exposure

API Gateway and Direct Service Exposure are two approaches to exposing services and APIs in a microservices architecture or a distributed system. Each approach has its own benefits and is suitable for different scenarios.

API Gateway

- **Definition:** An API Gateway is a single entry point for all clients to access various services in a microservices architecture. It acts as a reverse proxy, routing requests from clients to the appropriate backend services.
- **Characteristics:**
 - **Aggregation:** The gateway aggregates requests and responses from various services.
 - **Cross-Cutting Concerns:** Handles cross-cutting concerns like authentication, authorization, rate limiting, and logging.
 - **Simplifies Client Interaction:** Clients interact with one endpoint, simplifying the client-side logic.
- **Example:** An e-commerce platform where the API Gateway routes user requests to appropriate services like product catalog, user accounts, or order processing. A mobile app client makes a single request to the API Gateway to get a user's profile and order history, and the gateway routes this request to the respective services.
- **Pros:**
 - **Centralized Management:** Simplifies management of cross-cutting functionalities.
 - **Reduced Complexity for Clients:** Clients need to know only the endpoint of the API Gateway, not the individual services.

- **Enhanced Security:** Provides an additional layer of security by offering centralized authentication and SSL termination.
- **Cons:**
 - **Single Point of Failure:** Can become a bottleneck if not properly managed.
 - **Increased Latency:** Adding an extra network hop can increase response times.

Direct Service Exposure

- **Definition:** In direct service exposure, each microservice or service is directly exposed to clients. Clients interact with each service through its own endpoint.
- **Characteristics:**
 - **Direct Access:** Clients access services directly using individual service endpoints.
 - **Decentralized:** Each service manages its own cross-cutting concerns.
- **Example:** In a cloud storage service, clients might directly interact with separate endpoints for file uploads, downloads, and metadata retrieval. The client application has to manage multiple endpoints and handle cross-service functionalities like authentication for each service separately.
- **Pros:**
 - **Eliminates Single Point of Failure:** Avoids the bottleneck of a central gateway.
 - **Potentially Lower Latency:** Can offer reduced latency as requests do not go through an additional layer.
- **Cons:**
 - **Increased Client Complexity:** Clients must handle interactions with multiple services.
 - **Redundant Implementations:** Cross-cutting concerns may need to be implemented in each service.

Key Differences

- **Point of Contact:** API Gateway provides a single point of contact for accessing multiple services, while direct service exposure requires clients to interact with multiple endpoints.
- **Cross-Cutting Concerns:** API Gateway centralizes common functionalities like security and rate limiting, whereas in direct service exposure, these concerns are handled by each service.

Conclusion

The choice between using an API Gateway and direct service exposure depends on the specific requirements of the architecture and the trade-offs in terms of complexity, latency, and single points of failure. API Gateways are beneficial for unifying access to a distributed system and simplifying client interactions, making them suitable for complex, large-scale microservices architectures. Direct service exposure can be more efficient in terms of latency and is simpler architecturally but places more burden on the client to manage interactions with multiple services.

Proxy vs. Reverse Proxy

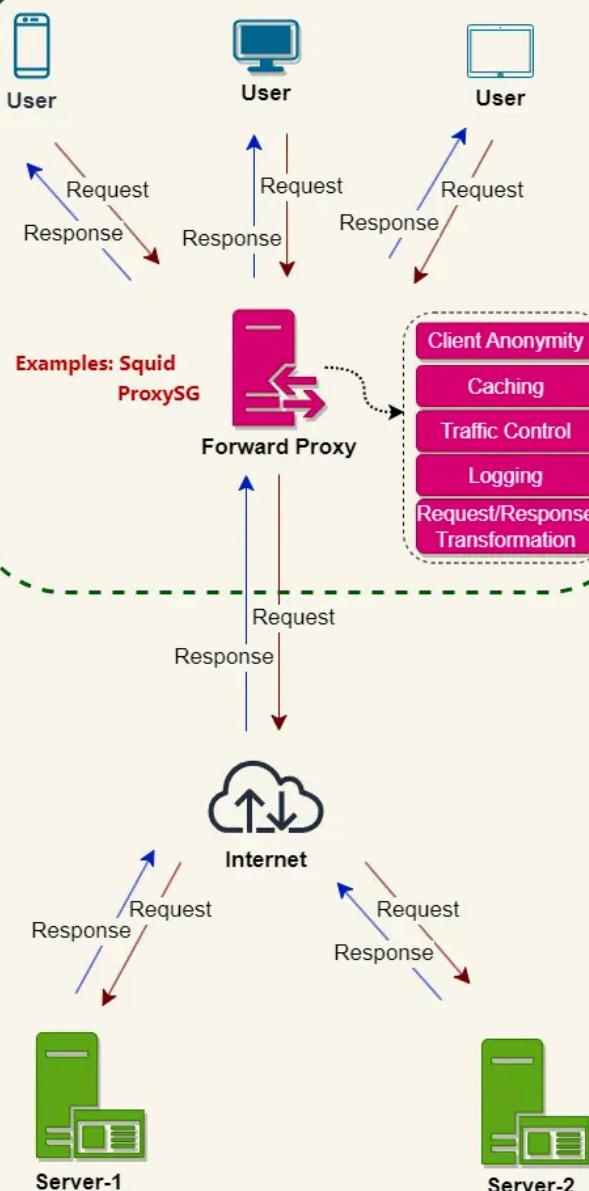
Proxy and Reverse Proxy are both intermediary entities in a network that manage and redirect traffic, but they differ in terms of their operational setup and the direction in which they handle traffic.

Proxy (Forward Proxy)

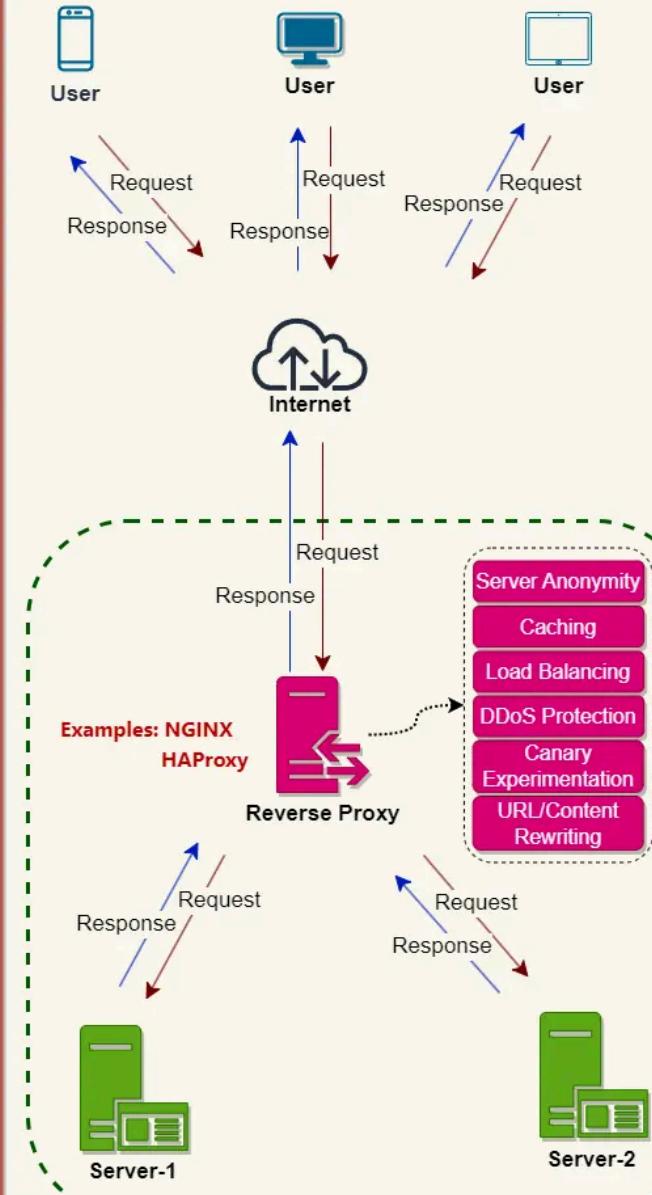
- **Operational Direction:** A Proxy, often referred to as a Forward Proxy, serves as an intermediary for requests from clients (like browsers) seeking resources from other servers. The clients connect to the proxy server, which then forwards the request to the destination server on behalf of the client.
- **Functionality:**
 - **Privacy and Anonymity:** Hides the identity of the client from the internet servers they are accessing. The server sees the proxy's IP address, not the client's.
 - **Content Filtering and Censorship:** Can be used to control and restrict access to specific websites or content.
 - **Caching:** May cache responses to reduce loading times and bandwidth usage for frequently accessed resources.
- **Use Case Example:** In an organizational network, a forward proxy is used to control internet access, where all employee web requests go through the proxy. The proxy blocks certain websites and caches frequently accessed resources to improve speed and reduce external bandwidth usage.

Forward Proxy vs. Reverse Proxy

Forward Proxy



Reverse Proxy



DesignGurus.io (one stop portal for coding and system design interviews)

Proxy vs. Reverse Proxy

Reverse Proxy

- **Operational Direction:** A Reverse Proxy, in contrast, is an intermediary for requests from clients (external or internal) directed to one or more servers. The clients connect to the reverse proxy server, which then forwards the request to the appropriate backend server.
- **Functionality:**
 - **Load Balancing:** Distributes incoming requests evenly among multiple servers to balance the load.
 - **Security and Anonymity for Servers:** Hides the identities of the backend servers from the clients, providing an additional security layer.
 - **SSL Termination:** Handles SSL encryption and decryption, offloading that responsibility from the backend servers.
 - **Caching and Compression:** Improves performance by caching content and compressing server responses.
- **Use Case Example:** A high-traffic website uses a reverse proxy to manage incoming user requests. The proxy distributes traffic among various servers (load balancing), caches content for faster retrieval, and provides SSL encryption for secure communications.

Key Differences

- **Direction of Traffic:**
 - A **Proxy (Forward Proxy)** acts on behalf of clients (users), managing outbound requests to the internet or other networks.
 - A **Reverse Proxy** acts on behalf of servers, managing inbound requests from the outside to the server infrastructure.

- **Intended Purpose:**

- **Proxies** are typically used for client privacy, internet access control, and caching.
- **Reverse Proxies** are used for server load balancing, security, performance enhancement, and as an additional layer in the server architecture.

Conclusion

While both Proxy and Reverse Proxy serve as intermediaries in network traffic, their roles are essentially opposite. A Proxy is client-facing, managing outgoing traffic and user access, while a Reverse Proxy is server-facing, managing incoming traffic to the server infrastructure. Their deployment and specific functionalities reflect these distinct roles.

API Gateway vs. Reverse Proxy

API Gateway and Reverse Proxy are both architectural components that manage incoming requests, but they serve different purposes and operate in somewhat different contexts.

API Gateway

- **Purpose:** An API Gateway is a management tool that acts as a single entry point for a defined group of microservices, handling requests and routing them to the appropriate service.
- **Functionality:**
 - **Routing:** Routes requests to the correct microservice.
 - **Aggregation:** Aggregates results from multiple microservices.
 - **Cross-Cutting Concerns:** Handles cross-cutting concerns like authentication, authorization, rate limiting, and logging.
 - **Protocol Translation:** Can translate between web protocols (HTTP, WebSockets) and backend protocols.
- **Use Cases:** Typically used in microservices architectures to provide a unified interface to a set of independently deployable services.
- **Example:** In a microservices-based e-commerce application, the API Gateway would be the single entry point for all client requests. It would handle user authentication, then route product search requests to the search service, cart management requests to the cart service, etc.

Reverse Proxy

- **Purpose:** A Reverse Proxy is a type of proxy server that retrieves resources on behalf of a client from one or more servers. It sits between the client and the backend services or servers.
- **Functionality:**
 - **Load Balancing:** Distributes client requests across multiple servers to balance load and ensure reliability.
 - **Security:** Provides an additional layer of defense (hides the identities of backend servers).
 - **Caching:** Can cache content to reduce server load and improve performance.
 - **SSL Termination:** Handles SSL encryption and decryption, offloading that responsibility from backend servers.
- **Use Cases:** Commonly used in both monolithic and microservices architectures to enhance security, load balancing, and caching.
- **Example:** A website with high traffic might use a reverse proxy to distribute requests across multiple application servers, cache content for faster retrieval, and manage SSL connections.

Key Differences

- **Primary Role:**
 - An **API Gateway** primarily facilitates and manages application-level traffic, acting as a gatekeeper for microservices.
 - A **Reverse Proxy** focuses more on network-level concerns like load balancing, security, and caching for a wider range of applications.
- **Complexity and Functionality:**
 - **API Gateways** are more sophisticated in functionality, often providing additional features like request transformation, API orchestration, and rate limiting.

- **Reverse Proxies** tend to be simpler and more focused on network and server efficiency and security.

Conclusion

While both API Gateways and Reverse Proxies manage traffic, they cater to different needs. An API Gateway is more about managing, routing, and orchestrating API calls in a microservices architecture, whereas a Reverse Proxy is about general server efficiency, security, and network traffic management. In practice, many modern architectures might use both, with an API Gateway handling application-specific routing and a Reverse Proxy managing general traffic and security concerns.

SQL vs. NoSQL

Let's explore the differences between SQL and NoSQL. Think of them like two different storage cabinets, each with its unique way of organizing and accessing the stuff you put inside.

SQL (Structured Query Language) Databases:

What They Are:

- SQL databases are relational databases. They use structured query language (SQL) for defining and manipulating data.

How They Work:

- Data is stored in tables, and these tables are related to each other.
- They follow a schema, a defined structure for how data is organized.

Key Features:

- **ACID Compliance:** Ensures reliable transactions (Atomicity, Consistency, Isolation, Durability).
- **Structured Data:** Ideal for data that fits well into tables and rows.
- **Complex Queries:** Powerful for complex queries and joining data from multiple tables.

Popular Examples:

- MySQL, PostgreSQL, Oracle, Microsoft SQL Server.

Best For:

- Applications requiring complex transactions, like banking systems.

- Situations where data structure won't change frequently.

NoSQL (Not Only SQL) Databases:

What They Are:

- NoSQL databases are non-relational or distributed databases. They can handle a wide variety of data models, including document, key-value, wide-column, and graph formats.

How They Work:

- They don't require a fixed schema, allowing the structure of the data to change over time.
- They are designed to scale out by using distributed clusters of hardware, which is ideal for large data sets or cloud computing.

Key Features:

- **Flexibility:** Can store different types of data together without a fixed schema.
- **Scalability:** Designed to scale out and handle very large amounts of data.
- **Speed:** Can be faster than SQL databases for certain queries, especially in big data and real-time web applications.

Popular Examples:

- MongoDB (Document), Redis (Key-Value), Cassandra (Wide-Column), Neo4j (Graph).

Best For:

- Systems needing to handle large amounts of diverse data.
- Projects where the data structure can change over time.

SQL vs NoSQL – The Difference:

1. **Data Structure:** SQL requires a predefined schema; NoSQL is more flexible.
2. **Scaling:** SQL scales vertically (requires more powerful hardware), while NoSQL scales horizontally (across many servers).
3. **Transactions:** SQL offers robust transaction capabilities, ideal for complex queries. NoSQL offers limited transaction support but excels in speed and scalability.
4. **Complexity:** SQL can handle complex queries, while NoSQL is optimized for speed and simplicity of queries.

Choosing Between Them:

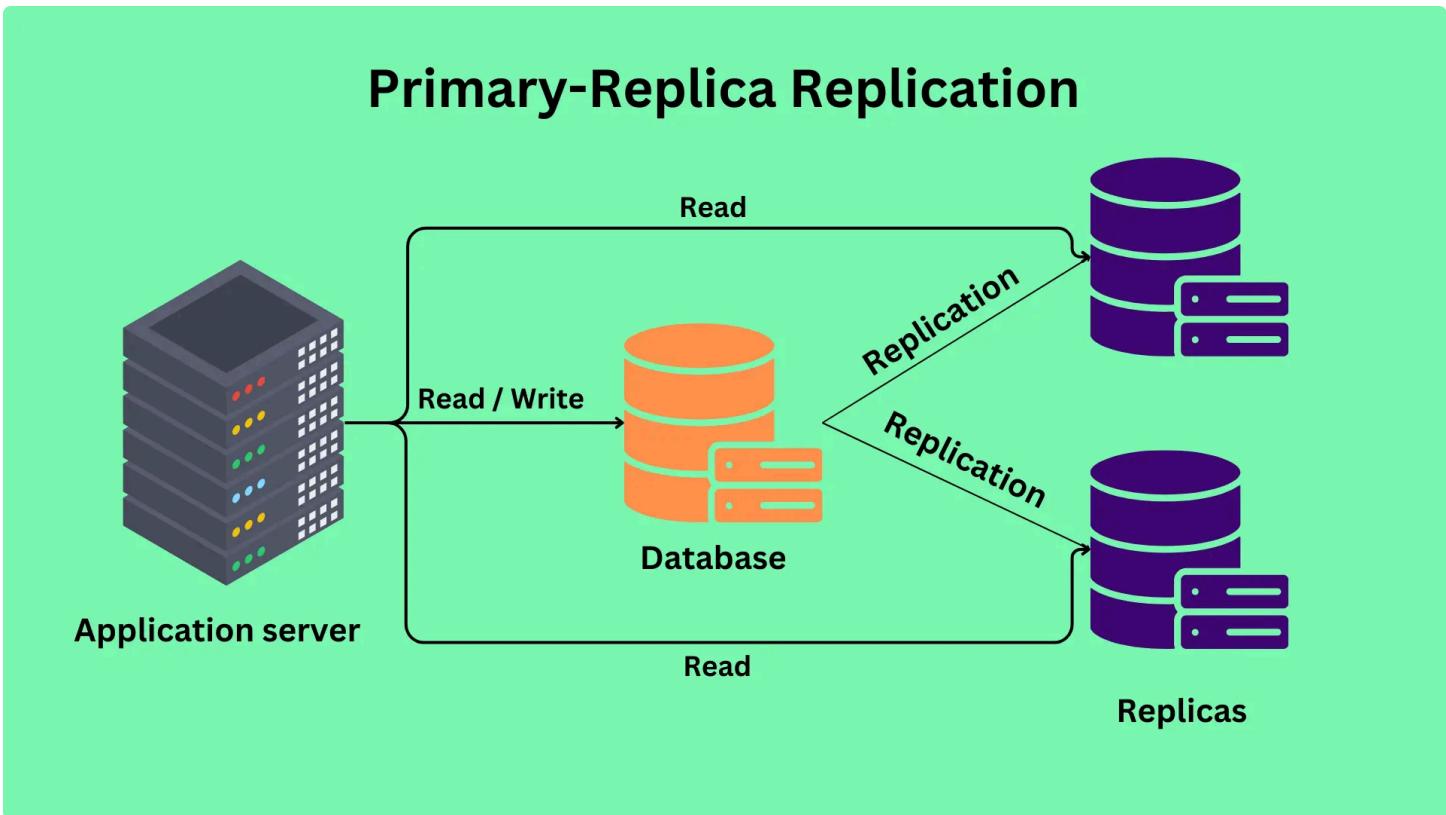
- **Use SQL when** you need strong ACID compliance, and your data structure is clear and consistent.
- **Use NoSQL when** you're dealing with massive volumes of data or need flexibility in the data model.

Both SQL and NoSQL have their unique strengths and are suited to different types of applications. The choice largely depends on the specific requirements of your project, including the data structure, scalability needs, and the complexity of the data operations.

Primary-Replica vs Peer-to-Peer Replication

Primary-Replica and Peer-to-Peer Replication are two distinct approaches to data replication in distributed systems, each with its own use cases, benefits, and challenges.

Primary-Replica Replication



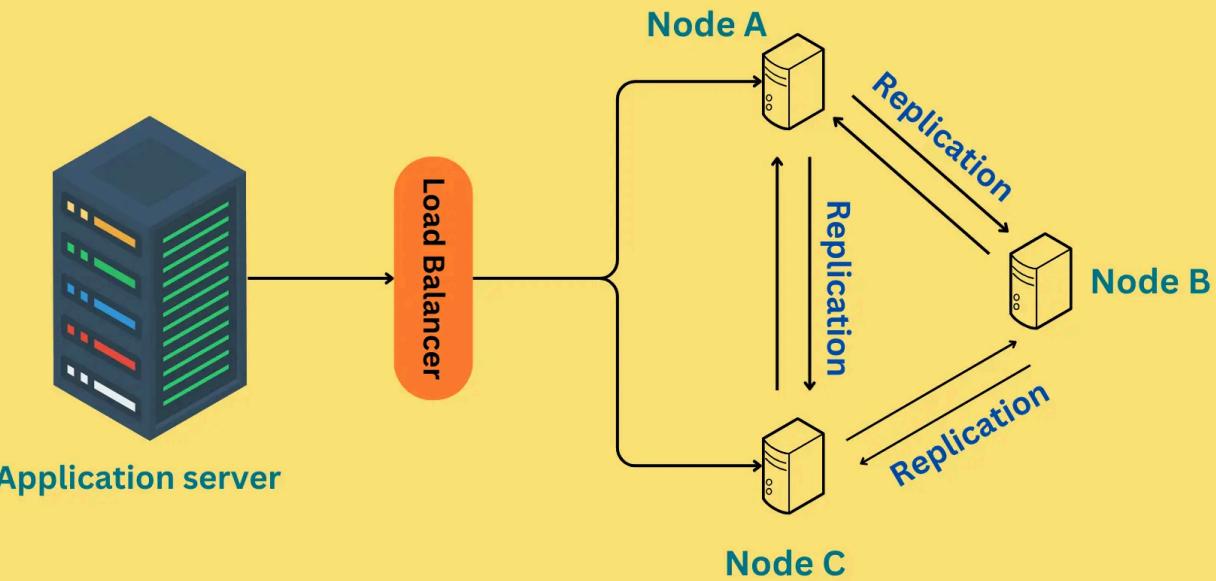
Primary-Replica Replication

- **Definition:** In Primary-Replica (also known as Master-Slave) replication, one server (the primary/master) handles all the write operations, and the changes are then replicated to one or more other servers (replicas/slaves).
- **Characteristics:**
 - **Unidirectional:** Data flows from the primary to the replicas.

- **Read and Write Split:** The primary handles writes, while replicas handle read queries, thereby distributing the load.
- **Example:** A popular example is a web application with a database backend. The primary database handles all write operations (like new user signups or content postings), while multiple replica databases handle read operations (like users browsing the site).
- **Pros:**
 - **Simplicity:** Easier to maintain and ensure consistency.
 - **Read Scalability:** Can scale read operations by adding more replicas.
- **Cons:**
 - **Single Point of Failure:** The primary is a critical point; if it fails, the system cannot process write operations.
 - **Replication Lag:** Changes made to the primary might take time to propagate to the replicas.

Peer-to-Peer Replication

Peer-to-Peer Replication



Peer-to-Peer Replication

- **Definition:** In Peer-to-Peer replication, each node (peer) in the network can act both as a client and a server. Peers are equally privileged and can initiate or complete replication processes.
- **Characteristics:**
 - **Multi-Directional:** Any node can replicate its data to any other node, and vice versa.
 - **Autonomy:** Each peer maintains its copy of the data and can independently respond to read and write requests.
- **Example:** A file-sharing application like BitTorrent uses peer-to-peer replication. Each user (peer) in the network can download files (data) from others and simultaneously upload files to others.
- **Pros:**
 - **Decentralization:** Eliminates single points of failure and bottlenecks.

- **Load Distribution:** Spreads the load evenly across the network.
- **Cons:**
 - **Complexity:** More complex to manage and ensure data consistency across all nodes.
 - **Conflict Resolution:** Handling data conflicts can be challenging due to simultaneous updates from multiple peers.

Key Differences

- **Control and Flow:** In Primary-Replica replication, the primary has control over write operations, with a clear flow of data from the primary to replicas. In Peer-to-Peer, every node can perform read and write operations, and data flow is multi-directional.
- **Architecture:** Primary-Replica follows a more centralized architecture, whereas Peer-to-Peer is decentralized.
- **Use Cases:** Primary-Replica is ideal for applications where read-heavy operations need to be scaled. Peer-to-Peer is suited for distributed networks where decentralization and load distribution are critical, such as in file sharing or blockchain technologies.

Conclusion

The choice between Primary-Replica and Peer-to-Peer replication depends on the specific requirements of the application, such as the need for scalability, fault tolerance, and the desired level of decentralization. Primary-Replica offers simplicity and read scalability, making it suitable for traditional database applications. In contrast, Peer-to-Peer provides robustness against failures and load distribution, ideal for decentralized networks.

Data Compression vs Data Deduplication

Data compression and data deduplication are two techniques used to optimize data storage, but they function in different ways and are suited for different scenarios.

Data Compression

- **Definition:** Data compression involves encoding information using fewer bits than the original representation. It reduces the size of data by removing redundancies and is often used to save storage space or decrease transmission times.
- **Types:**
 - **Lossless Compression:** Reduces file size without losing any data (e.g., ZIP files). You can restore data to its original state.
 - **Lossy Compression:** Reduces file size by permanently eliminating certain information, especially in media files (e.g., JPEG images, MP3 audio).
- **Example:** When you compress a text document using a ZIP file format, it uses algorithms to find and eliminate redundancies, reducing the file size. The original document can be perfectly reconstructed when the ZIP file is decompressed.
- **Pros:**
 - **Efficient Storage:** Saves storage space.
 - **Faster Transmission:** Reduces data transmission time over networks.
- **Cons:**
 - **Processing Overhead:** Requires computational resources for compressing and decompressing data.
 - **Quality Loss in Lossy Compression:** Can lead to quality degradation in media files.

Data Deduplication

- **Definition:** Data deduplication is a technique for eliminating duplicate copies of repeating data. It is used in data storage and backup systems to reduce the amount of storage space needed.
- **Process:**
 - **Identify Duplicates:** The system identifies and removes redundant data segments, keeping only one copy of each segment.
 - **Reference Links:** Subsequent copies are replaced with pointers to the stored segment.
- **Example:** In a corporate backup system, many employees might have the same file saved on their computers. Instead of storing each copy, deduplication stores one copy and then references to that copy for all subsequent identical files.
- **Pros:**
 - **Significantly Reduces Storage Needs:** Particularly effective in environments with lots of redundant data, like backup systems.
 - **Optimizes Backup Processes:** Makes backups more efficient by reducing the amount of data to be backed up.
- **Cons:**
 - **Limited to Identical Data:** Only reduces data that is exactly the same.
 - **Resource Intensive:** Requires processing power to identify duplicates.

Key Differences

- **Method of Reduction:** Data compression reduces file size by eliminating redundancies within a file, whereas data deduplication eliminates redundant files or data blocks across a system.

- **Scope:** Compression works on a single file or data stream, while deduplication works across a larger dataset or storage system.
- **Restoration:** Compressed data can be decompressed to its original form, but deduplicated data relies on references to the original data for restoration.

Conclusion

Data compression is useful for reducing the size of individual files for storage and transmission efficiency. In contrast, data deduplication is ideal for large-scale storage systems where the same data is stored or backed up multiple times. Both techniques can significantly improve storage efficiency, but they are used in different contexts and often complement each other in comprehensive data storage and management strategies.

Server-Side Caching vs Client-Side Caching

Server-side caching and client-side caching are two strategies used to store data temporarily to improve the performance and efficiency of applications. Both serve the purpose of reducing load times and bandwidth usage but operate at different points in the data retrieval and rendering process.

Server-Side Caching

- **Definition:** Server-side caching involves storing frequently accessed data on the server. When a client requests data, the server first checks its cache. If the data is present (cache hit), it is served from the cache; otherwise, the server processes the request and may cache the result for future requests.
- **Characteristics:**
 - **Location:** Cache is maintained on the server-side.
 - **Control:** Fully controlled by the server.
 - **Types:** Includes database query caching, page caching, and object caching.
- **Examples:**
 - **Database Query Results:** A web application server caches the results of common database queries. When a user requests data, such as product information, the server quickly retrieves the data from the cache instead of querying the database again.
 - **Full HTML Pages:** A news website caches entire HTML pages on the server. When a user requests to view an article, the server delivers the cached page, reducing the time taken to generate the page dynamically.
- **Pros:**
 - **Reduced Load Times:** Faster response times for users as data is quickly retrieved from the cache.

- **Decreased Server Load:** Reduces the load on databases and backend systems.
- **Cons:**
 - **Resource Usage:** Requires additional server resources (memory, disk space).
 - **Cache Management:** Requires effective cache invalidation strategies to ensure data consistency.

Client-Side Caching

- **Definition:** Client-side caching stores data on the client's device, such as a web browser or a mobile app. This cache is used to quickly load data without sending a request to the server.
- **Characteristics:**
 - **Location:** Cache is maintained on the client's device (e.g., browser, mobile app).
 - **Control:** Controlled by the client, with some influence from server settings.
 - **Types:** Includes browser caching of images, scripts, stylesheets, and application data caching.
- **Examples:**
 - **Browser Caching of Website Assets:** When a user first visits a website, resources like images, CSS, and JavaScript files are stored in the browser's cache. On subsequent visits, these assets load from the cache, speeding up page rendering.
 - **Mobile App Data:** A weather app on a phone caches the latest weather data. When the user reopens the app, it displays cached data until it refreshes the information.
- **Pros:**
 - **Reduced Network Traffic:** Decreases load times and bandwidth usage as fewer data needs to be downloaded from the server.

- **Offline Access:** Allows users to access cached data even when offline.
- **Cons:**
 - **Storage Limitations:** Limited by the client device's storage capacity.
 - **Stale Data:** Can lead to users viewing outdated information if not synchronized properly with the server.

Key Differences

- **Cache Location:** Server-side caching occurs on the server, benefiting all users, while client-side caching is specific to an individual user's device.
- **Data Freshness:** Server-side caching can centrally manage data freshness, while client-side caching may serve stale data if not properly updated.
- **Resource Utilization:** Server-side caching uses server resources and is ideal for data used by multiple users. Client-side caching uses the client's resources and is ideal for user-specific or static data.

Conclusion

Both server-side and client-side caching are essential for optimizing application performance. Server-side caching is effective for reducing server load and speeding up data delivery from the server. In contrast, client-side caching enhances the end-user experience by reducing load times and enabling offline content access. The choice of caching strategy depends on the specific needs of the application, the type of data being handled, and the desired balance between server load and client experience. For example, a website might use server-side caching for frequently accessed database content and client-side caching for static assets like images and stylesheets. By combining both methods, applications can provide a fast, efficient, and seamless user experience.

REST vs RPC

REST (Representational State Transfer) and RPC (Remote Procedure Call) are two architectural approaches used for designing networked applications, particularly for web services and APIs. Each has its distinct style and is suited for different use cases.

REST (Representational State Transfer)

- **Concept:** REST is an architectural style that uses HTTP requests to access and manipulate data. It treats server data as resources that can be created, read, updated, or deleted (CRUD operations) using standard HTTP methods (GET, POST, PUT, DELETE).
- **Stateless:** Each request from client to server must contain all the necessary information to understand and complete the request. The server does not store any client context between requests.
- **Data and Resources:** Emphasizes on resources, identified by URLs, and their state transferred over HTTP in a textual representation like JSON or XML.
- **Example:** A RESTful web service for a blog might provide a URL like `http://example.com/articles` for accessing articles. A GET request to that URL would retrieve articles, and a POST request would create a new article.

Advantages of REST

- **Scalability:** Stateless interactions improve scalability and visibility.
- **Performance:** Can leverage HTTP caching infrastructure.
- **Simplicity and Flexibility:** Uses standard HTTP methods, making it easy to understand and implement.

Disadvantages of REST

- **Over-fetching or Under-fetching:** Sometimes, it retrieves more or less data than needed.
- **Standardization:** Lacks a strict standard, leading to different interpretations and implementations.

RPC (Remote Procedure Call)

- **Concept:** RPC is a protocol that allows one program to execute a procedure (subroutine) in another address space (commonly on another computer on a shared network). The programmer defines specific procedures.
- **Procedure-Oriented:** Clients and servers communicate with each other through explicit remote procedure calls. The client invokes a remote method, and the server returns the results of the executed procedure.
- **Data Transmission:** Can use various formats like JSON (JSON-RPC) or XML (XML-RPC), or binary formats like Protocol Buffers (gRPC).
- **Example:** A client invoking a method `getArticle(articleId)` on a remote server. The server executes the method and returns the article's details to the client.

Advantages of RPC

- **Tight Coupling:** Allows for a more straightforward mapping of actions (procedures) to server-side operations.
- **Efficiency:** Binary RPC (like gRPC) can be more efficient in data transfer and faster in performance.
- **Clear Contract:** Procedure definitions create a clear contract between the client and server.

Disadvantages of RPC

- **Less Flexible:** Tightly coupled to the methods defined on the server.
- **Stateful Interactions:** Can maintain state, which might reduce scalability.

Conclusion

- **REST** is generally more suited for web services and public APIs where scalability, caching, and a uniform interface are important.
- **RPC** is often chosen for actions that are tightly coupled to server-side operations, especially when efficiency and speed are critical, as in internal microservices communication.

Polling vs Long-Polling vs Webhooks

Polling, long-polling, and webhooks are three techniques used in applications for getting updates or information, each with its own mechanism and use case.

Polling

- **Definition:** Polling is a technique where the client repeatedly requests (polls) a server at regular intervals to get new or updated data.
- **Characteristics:**
 - **Regular Requests:** The client makes requests at fixed intervals (e.g., every 5 seconds).
 - **Client-Initiated:** The client initiates each request.
- **Example:** A weather app that checks for updated weather information every 15 minutes by sending a request to the weather server.
- **Pros:**
 - **Simple to Implement:** Easy to set up on the client side.
- **Cons:**
 - **Inefficient:** Generates a lot of unnecessary traffic and server load, especially if there are no new updates.
 - **Delay in Updates:** There's always a delay between the actual update and the client receiving it.

Long-Polling

- **Definition:** Long-polling is an enhanced version of polling where the server holds the request open until new data is available to send back to the client.

- **Characteristics:**

- **Open Connection:** The server keeps the connection open for a period until there's new data or a timeout occurs.
- **Reduced Traffic:** Less frequent requests compared to traditional polling.
- **Example:** A chat application where the client sends a request to the server and the server holds the request until new messages are available. Once new messages arrive, the server responds, and the client immediately sends another request.
- **Pros:**
 - **More Timely Updates:** Clients can receive updates more quickly after they occur.
 - **Reduced Network Traffic:** Less frequent requests than standard polling.
- **Cons:**
 - **Resource Intensive on the Server:** Holding connections open can consume server resources.

Webhooks

- **Definition:** Webhooks are user-defined HTTP callbacks that are triggered by specific events. When the event occurs, the source site makes an HTTP request to the URL configured for the webhook.
- **Characteristics:**
 - **Server-Initiated:** The server sends data when there's a new update, without the client needing to request it.
 - **Event-Driven:** Triggered by specific events in the server.
- **Example:** A project management tool where a webhook is set up to notify a team's chat application whenever a new task is created. The creation of the task triggers a webhook that sends data directly to the chat app.

- **Pros:**

- **Real-Time:** Provides real-time updates.
- **Efficient:** Eliminates the need for polling, reducing network traffic and load.

- **Cons:**

- **Complexity in Handling:** The client needs to be capable of receiving and handling incoming HTTP requests.
- **Security Considerations:** Requires secure handling to prevent malicious data reception.

Key Differences

- **Initiation and Traffic:** Polling is client-initiated with frequent traffic, long-polling also starts with the client but reduces traffic by keeping the request open, and webhooks are server-initiated, requiring no polling.
- **Real-Time Updates:** Webhooks offer the most real-time updates, while polling and long-polling have inherent delays.

Conclusion

The choice between polling, long-polling, and webhooks depends on the application's requirements for real-time updates, server and client capabilities, and efficiency considerations. Polling is simple but can be inefficient, long-polling offers a middle ground with more timely updates, and webhooks provide real-time updates efficiently but require the client to handle incoming requests.

CDN Usage vs Direct Server Serving

CDN (Content Delivery Network) usage and direct server serving are two different approaches to delivering content to end-users over the internet. Understanding their differences is crucial for optimizing website performance, especially for content-heavy and globally accessed websites.

CDN Usage

- **Definition:** A Content Delivery Network (CDN) is a network of distributed servers that deliver web content to users based on their geographic location. CDNs cache content in multiple locations closer to the end-users.
- **Characteristics:**
 - **Geographical Distribution:** Consists of servers located in various geographic locations to reduce latency.
 - **Content Caching:** Stores copies of web content (like HTML pages, images, videos) for faster delivery.
- **Example:** A global news website uses a CDN to serve news articles and videos. When a user from London accesses the website, they are served content from the nearest CDN server in the UK, rather than from the main server located in the USA.
- **Pros:**
 - **Reduced Latency:** Faster content delivery by serving users from a nearby CDN server.
 - **Scalability:** Effectively handles high traffic loads and spikes.
 - **Bandwidth Optimization:** Reduces the load on the origin server, saving bandwidth.
- **Cons:**

- **Costs:** Can incur additional costs, depending on the CDN provider and traffic volume.
- **Complexity:** Requires configuration and maintenance of CDN settings.

Direct Server Serving

- **Definition:** In direct server serving, all user requests are handled directly by the main server (origin server) where the website is hosted, without intermediary CDN servers.
- **Characteristics:**
 - **Single Location:** The server is typically located in a single geographic location.
 - **Direct Delivery:** All content is served directly from this server to the end-user.
- **Example:** A local restaurant website hosted on a single server. All users, regardless of their location, are served directly from this server. If the server is in New York, both New York and Tokyo users access the website through the same server.
- **Pros:**
 - **Simplicity:** Easier to set up and manage, as it involves a single hosting environment.
 - **Cost-Effective for Small Scale:** Can be more cost-effective for websites with low traffic or those serving a localized audience.
- **Cons:**
 - **Potential Latency:** Users far from the server location may experience slower access.
 - **Scalability Limits:** Might struggle to handle traffic spikes or high global traffic volumes efficiently.

Key Differences

- **Content Delivery:** CDN spreads content across multiple servers globally for faster delivery, while direct server serving relies on a single location for all content delivery.
- **Performance and Scalability:** CDN offers enhanced performance and scalability, especially for a global audience, whereas direct server serving may be sufficient for small-scale or localized websites.
- **User Experience:** CDN generally provides a better user experience in terms of speed, especially for users located far from the origin server.

Conclusion

Using a CDN is ideal for websites with a global audience and those serving heavy content (like media files), as it significantly improves loading times and handles traffic efficiently. Direct server serving might be adequate for smaller websites with a predominantly local user base or limited content, where the simplicity and lower costs are more beneficial than the performance gains of a CDN.

Serverless Architecture vs Traditional Server-based

Serverless architecture and traditional server-based architecture represent two different approaches to deploying and managing applications and services, especially in cloud computing.

Serverless Architecture

- **Definition:** In serverless architecture, the cloud provider dynamically manages the allocation and provisioning of servers. Developers write and deploy code without worrying about the underlying infrastructure.
- **Characteristics:**
 - **Dynamic Scaling:** Automatically scales up or down based on the demand.
 - **Billing Model:** Costs are based on actual usage — for instance, the number of function executions or execution time.
 - **Stateless:** Functions are typically stateless and executed in response to events.
- **Example:** A photo sharing application where the backend (like resizing uploaded images or processing metadata) is handled by serverless functions. These functions run in response to events (like an image upload) and the developer doesn't need to maintain or scale servers.
- **Pros:**
 - **Reduced Operational Overhead:** Eliminates the need for managing servers.
 - **Cost-Effective:** Pay only for what you use, which can reduce costs.
 - **High Scalability:** Automatically scales with the application load.
- **Cons:**
 - **Limited Control:** Less control over the environment and underlying infrastructure.

- **Cold Starts:** Can experience latency issues due to cold starts (initializing a function).

Traditional Server-based Architecture

- **Definition:** In traditional server-based architecture, applications are deployed on servers which must be provisioned, maintained, and scaled by the developer or the operations team.
- **Characteristics:**
 - **Fixed Resources:** Servers have fixed resources and need to be manually scaled.
 - **Continuous Operation:** Servers run continuously, irrespective of demand.
 - **Billing Model:** Typically involves ongoing costs regardless of usage, including server maintenance and operation.
- **Example:** A company website hosted on a dedicated server or a shared hosting service. The server runs continuously, and the team is responsible for installing updates, managing server security, and scaling resources during traffic spikes.
- **Pros:**
 - **Full Control:** Complete control over the server environment and infrastructure.
 - **Flexibility:** More flexibility in configuring and optimizing the server.
- **Cons:**
 - **Higher Costs:** Involves costs for unused capacity and continuous server maintenance.
 - **Operational Complexity:** Requires active management of the server infrastructure.

Key Differences

- **Infrastructure Management:** Serverless abstracts away the server management, while traditional architecture requires active management of servers.
- **Scaling:** Serverless automatically scales with demand, while traditional architecture requires manual scaling.
- **Cost Model:** Serverless has a pay-as-you-go model, whereas traditional architecture typically involves continuous costs for server operation.

Conclusion

Serverless architecture is ideal for applications with variable or unpredictable workloads, where simplifying operational management and reducing costs are priorities. Traditional server-based architecture is suitable for applications requiring extensive control over the environment and predictable performance. The choice depends on specific application requirements, workload patterns, and operational preferences.

Stateful vs Stateless Architecture

Stateful and Stateless architectures are two approaches to managing user information and data processing in software applications, particularly in web services and APIs.

Stateful Architecture

- **Definition:** In a stateful architecture, the server retains information (or state) about the client's session. This state is used to remember previous interactions and respond accordingly in future interactions.
- **Characteristics:**
 - **Session Memory:** The server remembers past session data, which influences its responses to future requests.
 - **Dependency on Context:** The response to a request can depend on previous interactions.
- **Example:** An online banking application is a typical example of a stateful application. Once you log in, the server maintains your session data (like authentication, your interactions). This data influences how the server responds to your subsequent actions, such as displaying your account balance or transaction history.
- **Pros:**
 - **Personalized Interaction:** Enables more personalized user experiences based on previous interactions.
 - **Easier to Manage Continuous Transactions:** Convenient for transactions that require multiple steps.
- **Cons:**
 - **Resource Intensive:** Maintaining state can consume more server resources.

- **Scalability Challenges:** Scaling a stateful application can be more complex due to session data dependencies.

Stateless Architecture

- **Definition:** In a stateless architecture, each request from the client to the server must contain all the information needed to understand and complete the request. The server doesn't rely on information from previous interactions.
- **Characteristics:**
 - **No Session Memory:** The server does not store any state about the client's session.
 - **Self-contained Requests:** Each request is independent and must include all necessary data.
- **Example:** RESTful APIs are a classic example of stateless architecture. Each HTTP request to a RESTful API contains all the information the server needs to process it (like user authentication, required data), and the response to each request doesn't depend on past requests.
- **Pros:**
 - **Simplicity and Scalability:** Easier to scale as there is no need to maintain session state.
 - **Predictability:** Each request is processed independently, making the system more predictable and easier to debug.
- **Cons:**
 - **Redundancy:** Can lead to redundancy in data sent with each request.
 - **Potentially More Complex Requests:** Clients may need to handle more complexities in preparing requests.

Key Differences

- **Session Memory:** Stateful retains user session information, influencing future interactions, whereas stateless treats each request as an isolated transaction, independent of previous requests.
- **Server Design:** Stateful servers maintain state, making them more complex and resource-intensive. Stateless servers are simpler and more scalable.
- **Use Cases:** Stateful is suitable for applications requiring continuous user interactions and personalization. Stateless is ideal for services where each request can be processed independently, like many web APIs.

Conclusion

Stateful and stateless architectures offer different approaches to handling user sessions and data processing. The choice between them depends on the specific requirements of the application, such as the need for personalization, resource availability, and scalability. Stateful provides a more personalized user experience but at the cost of higher complexity and resource usage, while stateless offers simplicity and scalability, suitable for distributed systems where each request is independent.

Hybrid Cloud Storage vs All-Cloud Storage

Hybrid cloud storage and all-cloud (or fully cloud) storage are two cloud computing models that businesses use for storing data, each with its own architecture and use cases.

Hybrid Cloud Storage

- **Definition:** Hybrid cloud storage combines on-premises data storage (private cloud) with public cloud storage. Data and applications can move between private and public clouds, providing greater flexibility and data deployment options.
- **Characteristics:**
 - **Integration:** Involves a mix of on-premises, private cloud, and public cloud services with orchestration between the platforms.
 - **Flexibility:** Allows businesses to store sensitive data on a private cloud or on-premises while leveraging the expansive power and scalability of the public cloud for less sensitive operations.
- **Example:**
 - **Healthcare System:** A hospital uses on-premises storage for sensitive patient records (due to compliance and security reasons) but leverages public cloud resources for managing less sensitive data like administrative tasks, patient portals, and large medical datasets for research.
 - **Retail Business:** A retail company stores its transactional and customer data in a private cloud for security but uses public cloud services for its e-commerce website and inventory management, benefiting from the scalability and advanced analytics capabilities of the public cloud.
- **Pros:**

- **Enhanced Security:** Sensitive data can be kept on-premises or in a private cloud, ensuring better control and security.
- **Scalability:** Easy to scale resources up or down using the public cloud component.

- **Cons:**

- **Complexity:** Managing and integrating different environments can be complex.
- **Potentially Higher Costs:** Can be more expensive than using only public cloud services due to the maintenance of on-premises infrastructure.

All-Cloud Storage (Fully Cloud)

- **Definition:** All-cloud storage, or fully cloud storage, involves using cloud-based services for all data storage needs without on-premises infrastructure. It typically utilizes public cloud services like AWS, Azure, or Google Cloud Platform.
- **Characteristics:**
 - **Cloud-Dependent:** All data is stored and managed in the cloud.
 - **Vendor Management:** Relies on third-party cloud service providers for infrastructure, security, and maintenance.
- **Example:**
 - **Startup Company:** A tech startup uses AWS for all its storage and computing needs, leveraging AWS's scalable infrastructure to handle varying loads and store vast amounts of user data without any on-premises hardware.
 - **Media Company:** A media streaming service uses Google Cloud Platform to store and distribute its extensive library of videos, benefiting from Google's global reach and advanced data analytics tools.
- **Pros:**

- **Low Maintenance:** Eliminates the need for physical infrastructure and its maintenance.
 - **High Scalability:** Easy to scale resources to meet demand.
- **Cons:**
 - **Potential Security Concerns:** Relying solely on external providers can raise concerns about data security and compliance.
 - **Internet Dependency:** Fully reliant on internet connectivity.

Key Differences

- **Infrastructure:** Hybrid cloud storage involves a mix of on-premises/private and public cloud storage, offering a balance of control and scalability. In contrast, all-cloud storage exclusively uses cloud services, relying entirely on external providers for data storage.
- **Flexibility vs. Simplicity:** Hybrid cloud offers flexibility in data deployment and security, ideal for businesses with varying compliance needs. All-cloud storage provides simplicity and ease of use, suitable for businesses that prefer to outsource their entire infrastructure.

Conclusion

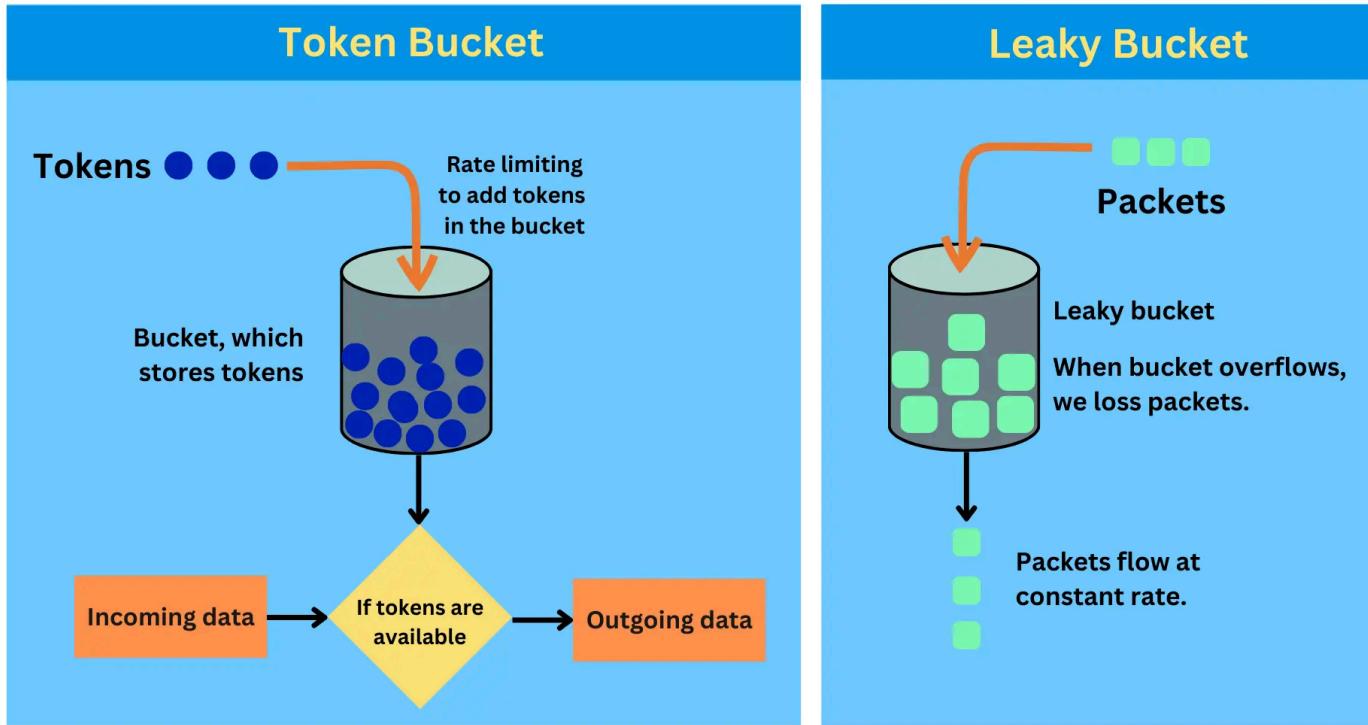
The choice between hybrid cloud storage and all-cloud storage depends on the specific needs and constraints of the organization. Hybrid cloud storage is optimal for businesses that require a balance of security, control, and compliance along with the scalability of cloud services. It's particularly suitable for organizations that handle sensitive data or are subject to strict regulatory requirements but also need the scalability and advanced services offered by public clouds.

On the other hand, all-cloud storage is ideal for businesses that prioritize scalability, flexibility, and minimal infrastructure maintenance. Startups, small businesses, or companies whose operations are entirely online may find all-cloud storage more advantageous due to its ease of use and lower upfront costs.

In summary, hybrid cloud storage offers the best of both worlds but at the cost of increased complexity, while all-cloud storage offers simplicity and scalability but with potential trade-offs in terms of control and data security.

Token Bucket vs Leaky Bucket

Token Bucket and Leaky Bucket are two algorithms used for network traffic shaping and rate limiting. They help manage the rate of traffic flow in a network, but they do so in slightly different ways.



Token Bucket vs Leaky Bucket

Token Bucket Algorithm

- Mechanism:** The token bucket algorithm is based on tokens being added to a bucket at a fixed rate. Each token represents permission to send a certain amount of data. When a packet (data) needs to be sent, it can only be transmitted if there is a token available, which is then removed from the bucket.

- **Characteristics:**

- **Burst Allowance:** Can handle bursty traffic because the bucket can store tokens, allowing for temporary bursts of data as long as there are tokens in the bucket.
- **Flexibility:** The rate of token addition and the size of the bucket can be adjusted to control the data rate.
- **Example:** Think of a video streaming service. The service allows data bursts for fast initial streaming (buffering) as long as tokens are available in the bucket. Once the tokens are used up, the streaming rate is limited to the rate of token replenishment.
- **Pros:**
 - Allows for flexibility in handling bursts of traffic.
 - Useful for applications where occasional bursts are acceptable.
- **Cons:**
 - Requires monitoring the number of available tokens, which might add complexity.

Leaky Bucket Algorithm

- **Mechanism:** In the leaky bucket algorithm, packets are added to a queue (bucket), and they are released at a steady, constant rate. If the bucket (buffer) is full, incoming packets are discarded or queued for later transmission.
- **Characteristics:**
 - **Smooth Traffic:** Ensures a steady, uniform output rate regardless of the input burstiness.
 - **Overflow:** Can result in packet loss if the bucket overflows.
- **Example:** Imagine an ISP limiting internet speed. The ISP uses a leaky bucket to smooth out the internet traffic. Regardless of how bursty the incoming traffic

is, the data flow to the user is at a consistent, predetermined rate. If the data comes in too fast and the bucket fills up, excess packets are dropped.

- **Pros:**

- Simple to implement and understand.
- Ensures a steady, consistent flow of traffic.

- **Cons:**

- Does not allow for much flexibility in handling traffic bursts.
- Can lead to packet loss if incoming rate exceeds the bucket's capacity.

Key Differences

- **Traffic Burst Handling:** Token bucket allows for bursts of data until the bucket's tokens are exhausted, making it suitable for applications where such bursts are common. In contrast, the leaky bucket smooths out the data flow, releasing packets at a steady, constant rate.
- **Use Cases:** Token bucket is ideal for applications that require flexibility and can tolerate bursts, like video streaming. Leaky bucket is suited for scenarios where a steady, continuous data flow is required, like voice over IP (VoIP) or real-time streaming.

Conclusion

Choosing between Token Bucket and Leaky Bucket depends on the specific requirements for traffic management in a network. Token Bucket offers more flexibility and is better suited for bursty traffic scenarios, while Leaky Bucket is ideal for maintaining a uniform output rate.

Read Heavy vs Write Heavy System

Designing systems for read-heavy versus write-heavy workloads involves different strategies, as each type of system has unique demands and challenges.

Designing for Read-Heavy Systems

Read-heavy systems are characterized by a high volume of read operations compared to writes. Common in scenarios like content delivery networks, reporting systems, or read-intensive APIs.

Key Strategies

1. Caching:

- Implement extensive caching mechanisms to reduce database read operations. Technologies like Redis or Memcached can be used to cache frequent queries or results.
- Cache at different levels (application level, database level, or using a dedicated caching service).
- **Example:** A news website experiences high traffic with users frequently accessing the same articles. Implementing a caching layer using a technology like Redis or Memcached stores the most accessed articles in memory. When a user requests an article, the system first checks the cache. If the article is there, it's served directly from the cache, significantly reducing database read operations.

2. Database Replication:

- Use database replication to create read replicas of the primary database. Read operations are distributed across these replicas, while write operations are directed to the primary database.
- Ensure eventual consistency between the primary database and the replicas.
- **Example:** An e-commerce platform uses a primary database for all transactions. To optimize for read operations (like browsing products), it replicates its database across multiple read replicas. User queries for product information are handled by these replicas, distributing the load and preserving the primary database for write operations.

3. Content Delivery Network (CDN):

- Use CDNs to cache static content geographically closer to users, reducing latency and offloading traffic from the origin server.
- **Example:** An online content provider uses a CDN to store static assets like images, videos, and CSS files. When a user accesses this content, it is delivered from the nearest CDN node rather than the origin server, enhancing speed and efficiency.

4. Load Balancing:

- Employ load balancers to distribute incoming read requests evenly across multiple servers or replicas.
- **Example:** A cloud-based application service uses a load balancer to distribute user requests across a cluster of servers, each capable of handling read operations. This setup ensures that no single server becomes a performance bottleneck.

5. Optimized Data Retrieval:

- Design efficient data access patterns and optimize queries for read operations.

- Use data indexing to speed up searches and retrievals.
- **Example:** An analytics dashboard that aggregates data for reports optimizes its SQL queries to fetch only relevant data, use proper indexes, and avoid costly join operations whenever possible.

6. Data Partitioning:

- Partition data to distribute the load across different servers or databases (sharding or horizontal partitioning).
- **Example:** A social media platform with millions of users implements database sharding. User data is partitioned based on user IDs or geographic location, allowing read queries to be directed to specific shards, thus reducing the read load on any single database server.

7. Asynchronous Processing

- Use asynchronous processing for operations that don't need to be done in real-time.
- **Example:** A financial application performs complex data aggregation and reporting. It uses asynchronous processing to pre-compute and store these reports, which can then be quickly retrieved on demand.

Designing for Write-Heavy Systems

Write-heavy systems are characterized by a high volume of write operations, such as logging systems, real-time data collection systems, or transactional databases.

Key Strategies

1. Database Optimization for Writes:

- Choose a database optimized for high write throughput (like NoSQL databases: Cassandra, MongoDB).
- Optimize database schema and indexes to improve write performance.
- **Example:** For a real-time analytics system, using a NoSQL database like Cassandra, which is optimized for high write throughput, can be more effective than a traditional SQL database. Cassandra's distributed architecture allows it to handle large write volumes efficiently.

2. Write Batching and Buffering:

- Batch multiple write operations together to reduce the number of write requests.
- **Example:** In a logging system where numerous log entries are generated every second, instead of writing each entry to the database individually, the system batches multiple log entries together and writes them in a single transaction, reducing the overhead of database writes.

3. Asynchronous Processing:

- Handle write operations asynchronously, allowing the application to continue processing without waiting for the write operation to complete.
- **Example:** A video sharing platform like YouTube processes user uploaded videos asynchronously. When a video is uploaded, it's added to a queue, and the user receives an immediate confirmation. The video processing, including encoding and thumbnails generations, happens in the background.

4. CQRS (Command Query Responsibility Segregation)

- Separate the write (command) and read (query) operations into different models.
- **Example:** In a financial system, transaction processing (writes) is handled separately from account balance inquiries (reads). This separation allows

optimizing the write model for transactional integrity and the read model for performance.

5. Data Partitioning:

- Use sharding or partitioning to distribute write operations across different database instances or servers.
- **Example:** A social media application uses sharding to distribute user data across multiple databases based on user IDs. When new posts are created, they are written to the shard corresponding to the user's ID, distributing the write load across the database infrastructure.

6. Use of Write-Ahead Logging (WAL)

- First write changes to a log before applying them to the database. This ensures data integrity and improves write performance.
- **Example:** A database management system uses WAL to handle transactions. Changes are first written to a log file, ensuring that in case of a crash, the database can recover and apply missing writes, thus maintaining data integrity.

7. Event Sourcing:

- Persist changes as a sequence of immutable events rather than modifying the database state directly.
- **Example:** In an order management system, instead of updating an order record directly, each change (like order placed, order shipped) is stored as a separate event. This stream of events can be processed and persisted efficiently and replayed to reconstruct the order state.

Conclusion

Read-heavy systems benefit significantly from caching and data replication to reduce database read operations and latency. Write-heavy systems, on the other hand, require optimized database writes, effective data distribution, and asynchronous processing to handle high volumes of write operations efficiently. The choice of technologies and architecture patterns should align with the specific demands of the workload.