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.

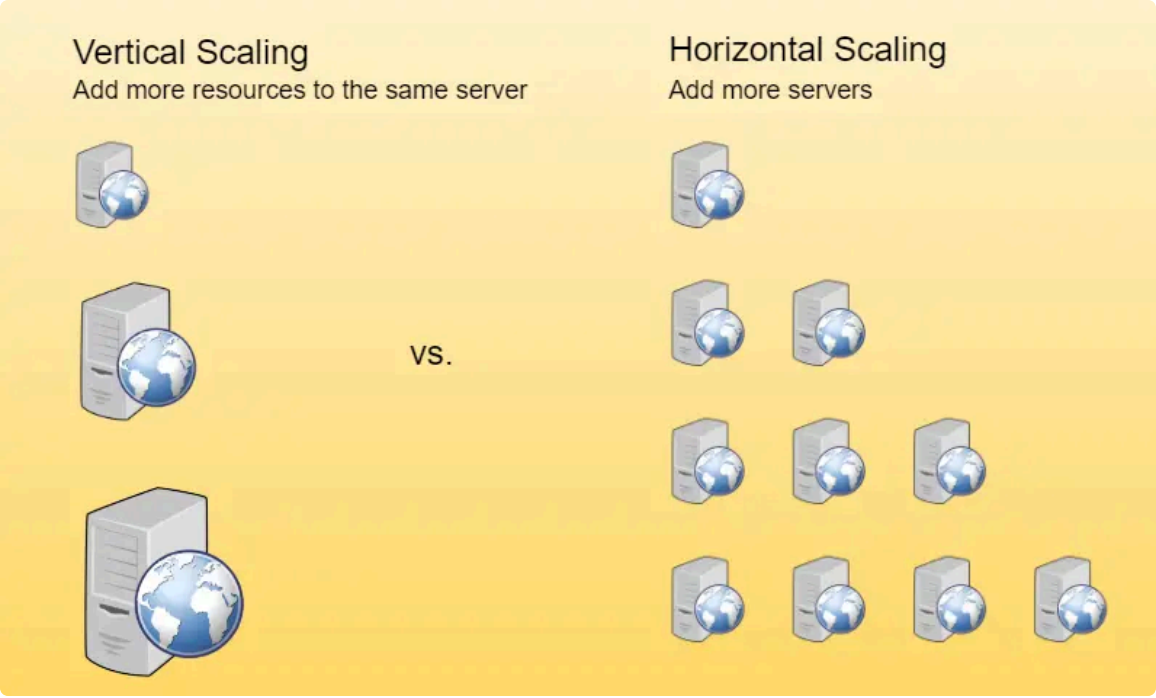
Generally, the performance of a system, although designed (or claimed) to be scalable, declines with the system size due to the management or environment cost. For instance, network speed may become slower because machines tend to be far apart from one another. More generally, some tasks may not be distributed, 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

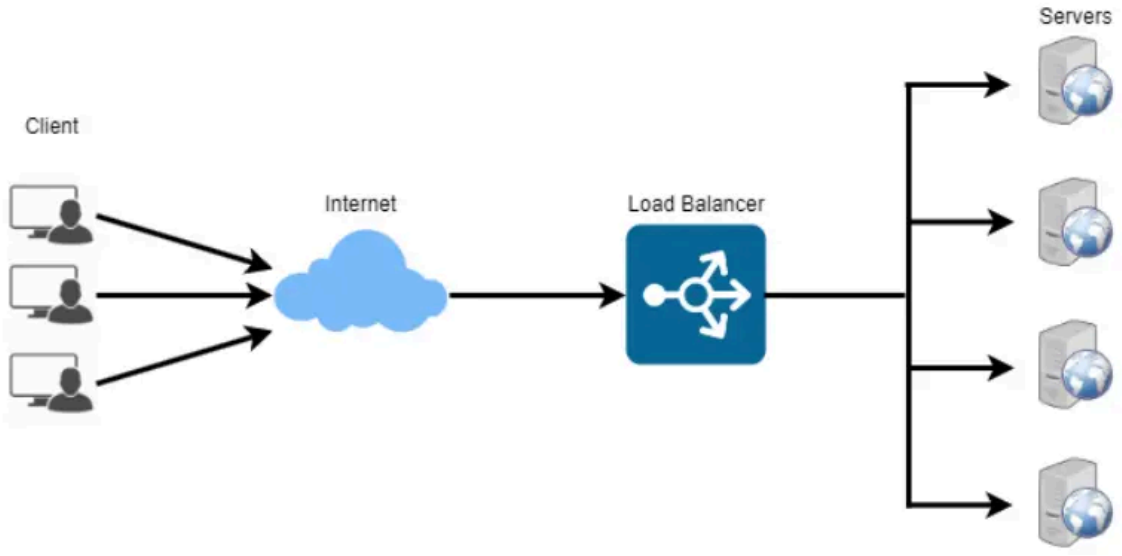
Another important consideration while designing a distributed system is how easy it is to operate and maintain. Serviceability or manageability is the simplicity and speed with which a system can be repaired or maintained; if the time to fix a failed system increases, then availability will decrease. 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.

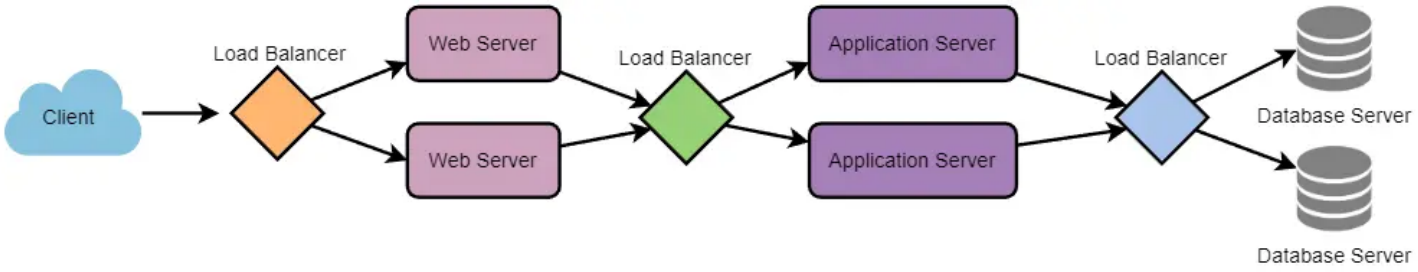
Typically a load balancer sits between the client and the server accepting incoming network and application traffic and distributing the traffic across multiple backend servers using various algorithms. By balancing application requests across multiple servers, a load balancer reduces individual server 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.

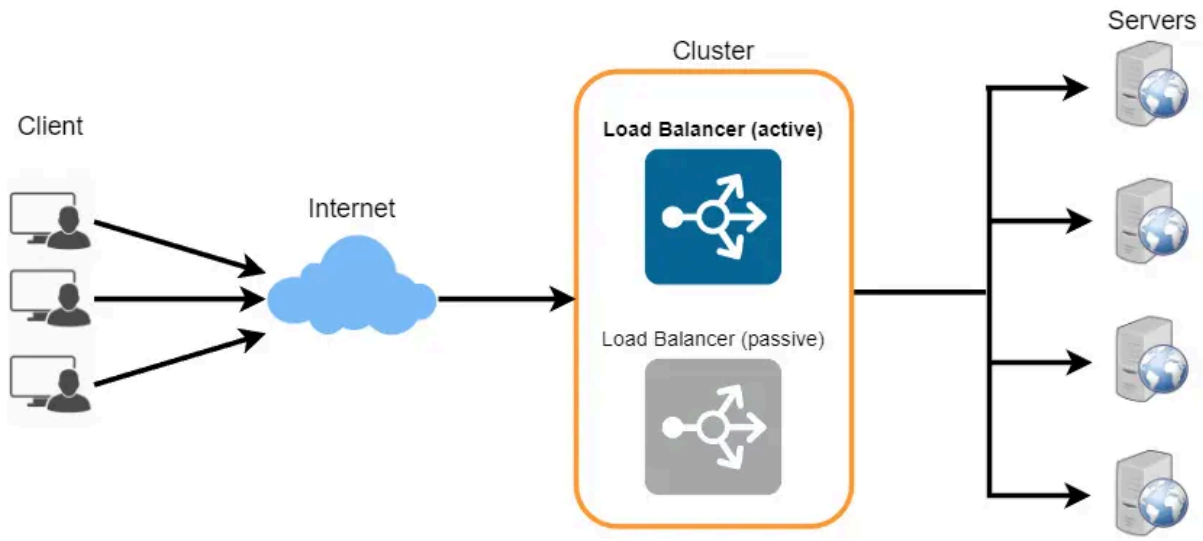
- **Least Connection Method** — This method directs traffic to the server with the fewest active connections. This approach is quite useful when there are a large number of persistent client connections which are unevenly distributed between the servers.
- **Least Response Time Method** — This algorithm 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.

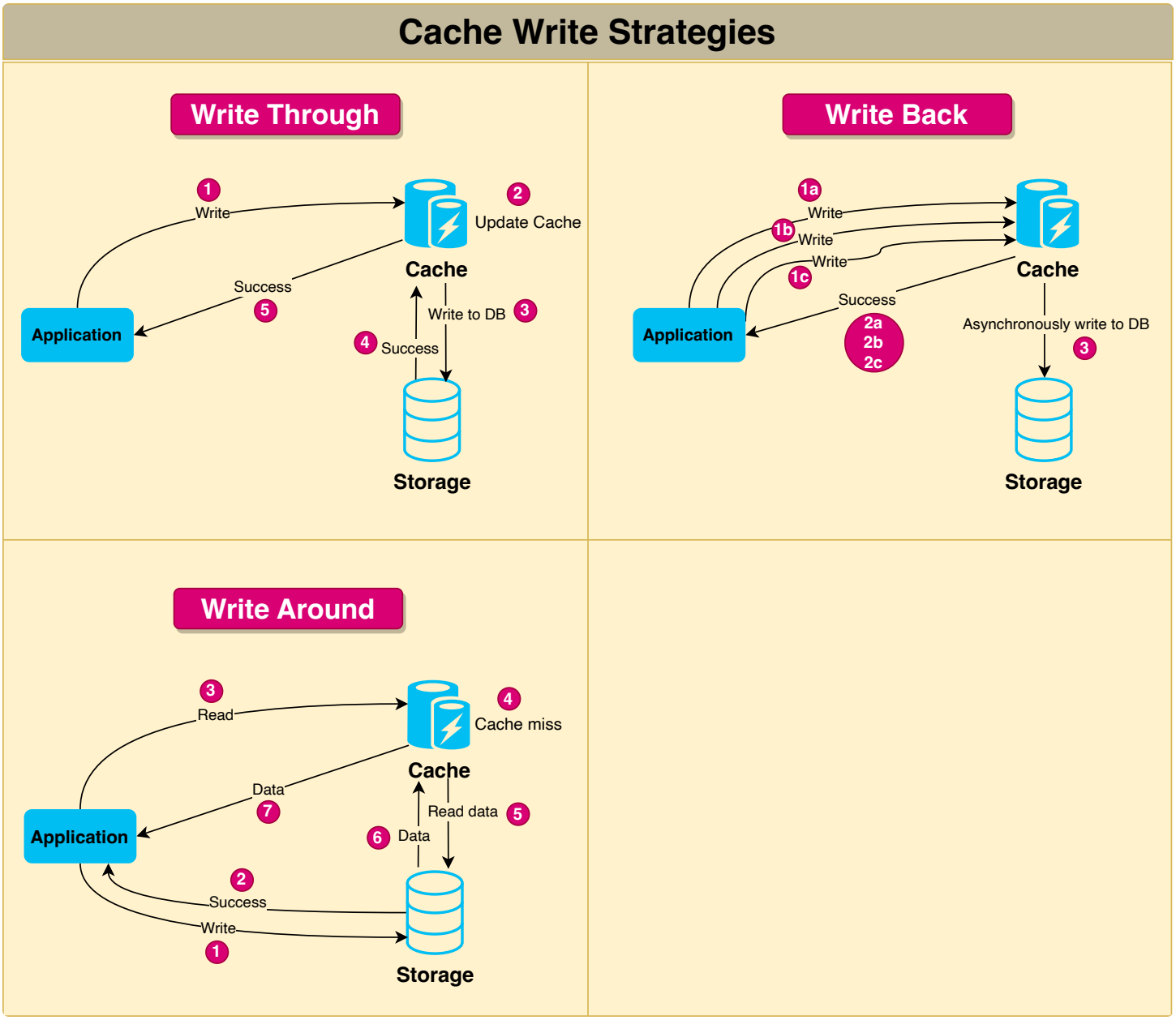
Solving this problem is known as cache invalidation; there are three main schemes that are used:

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

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

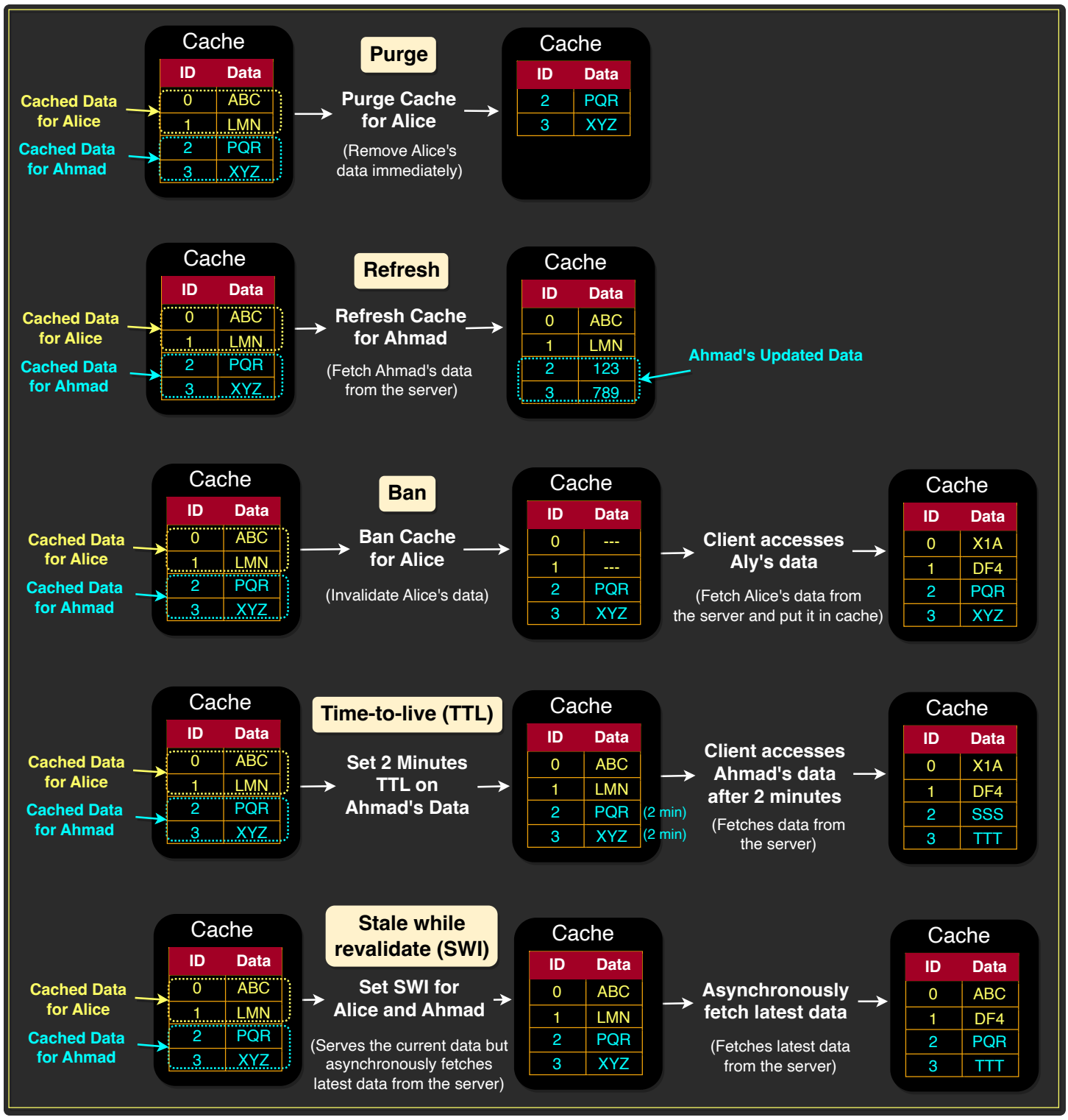
content and the cached version is no longer valid. When a purge request is received, the cached content is immediately removed, and the next request for the content will be served directly from the origin server.

- **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.

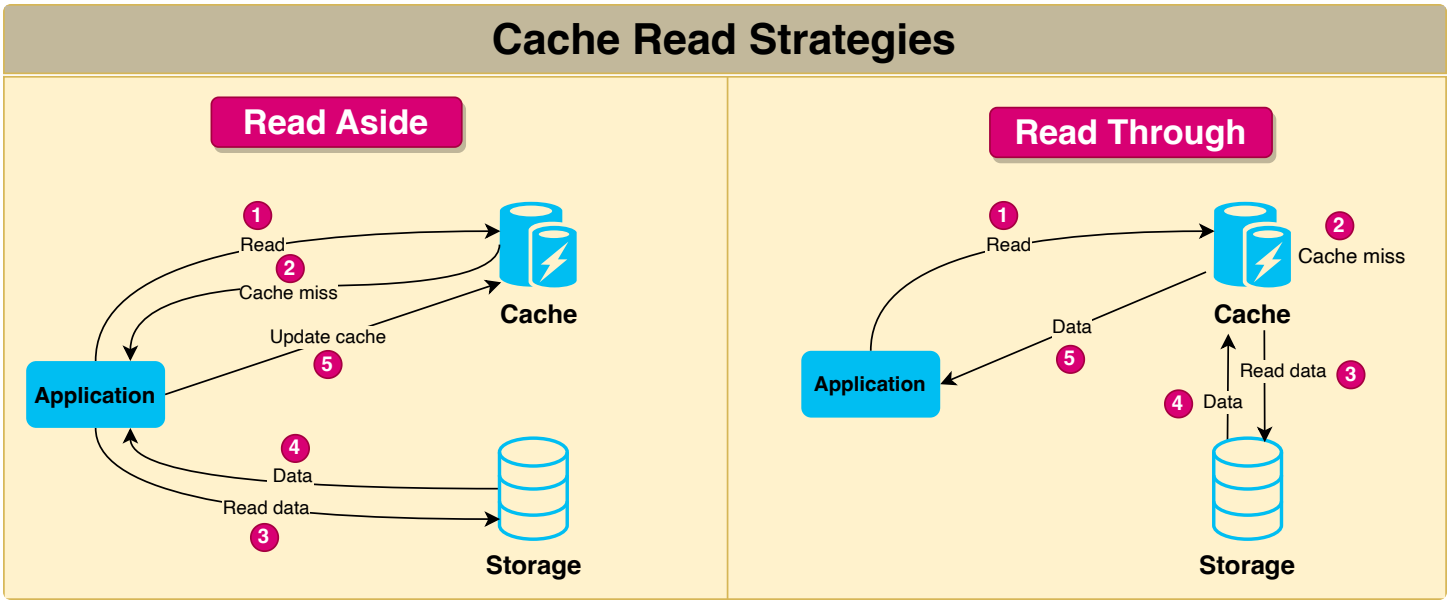
This approach helps to maintain consistency between the cache and the data store, as the cache is always responsible for retrieving and updating the data. It also simplifies the application code since the application doesn't need to handle cache misses and data retrieval logic. The read-through cache 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 Read Strategies

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.

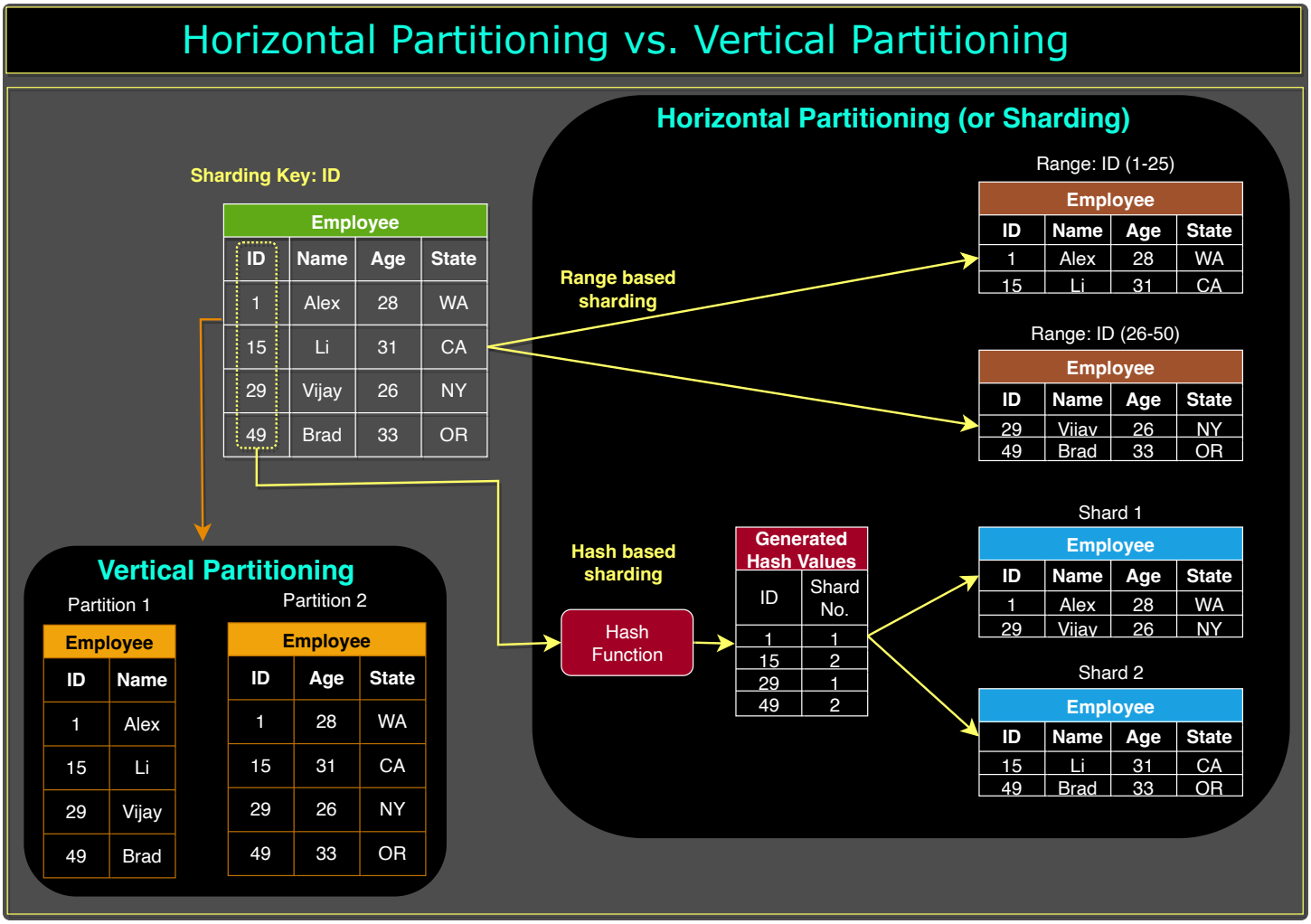
b. Vertical Partitioning: Vertical data partitioning involves splitting a database table into multiple partitions or shards, with each partition containing a subset of columns. This technique can help optimize performance by reducing the amount of data that needs to be scanned, especially when certain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

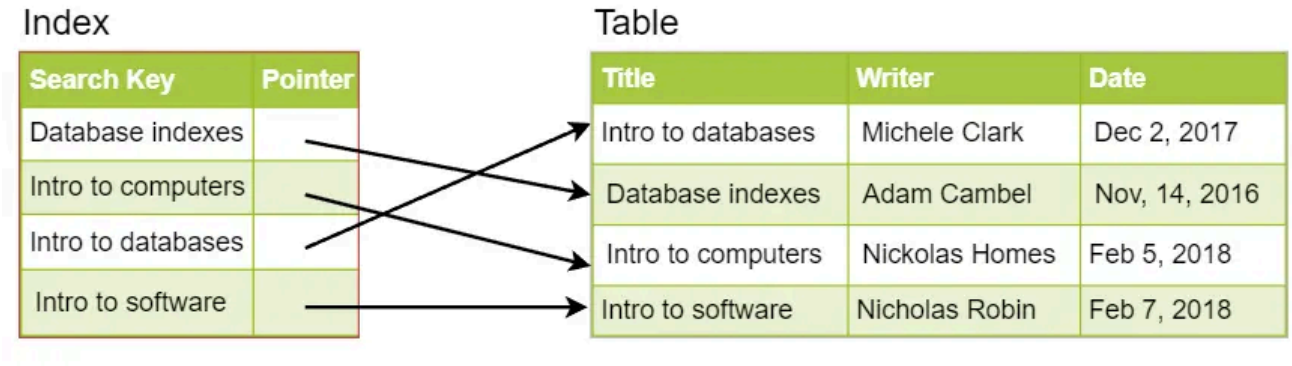
Indexes are well known when it comes to databases. Sooner or later there comes a time when database performance is no longer satisfactory. One of the very first things you should turn to when that happens is database indexing.

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

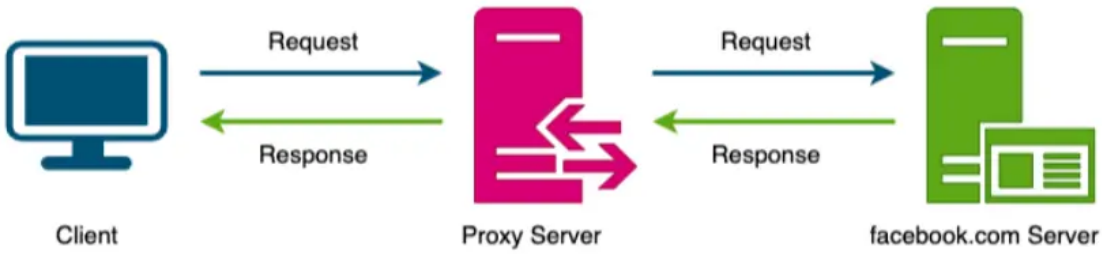
that is often written to and rarely read from, in that case, decreasing the performance of the more common operation, which is writing, is probably not worth the increase in performance we get from reading.

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.



A forward proxy

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).

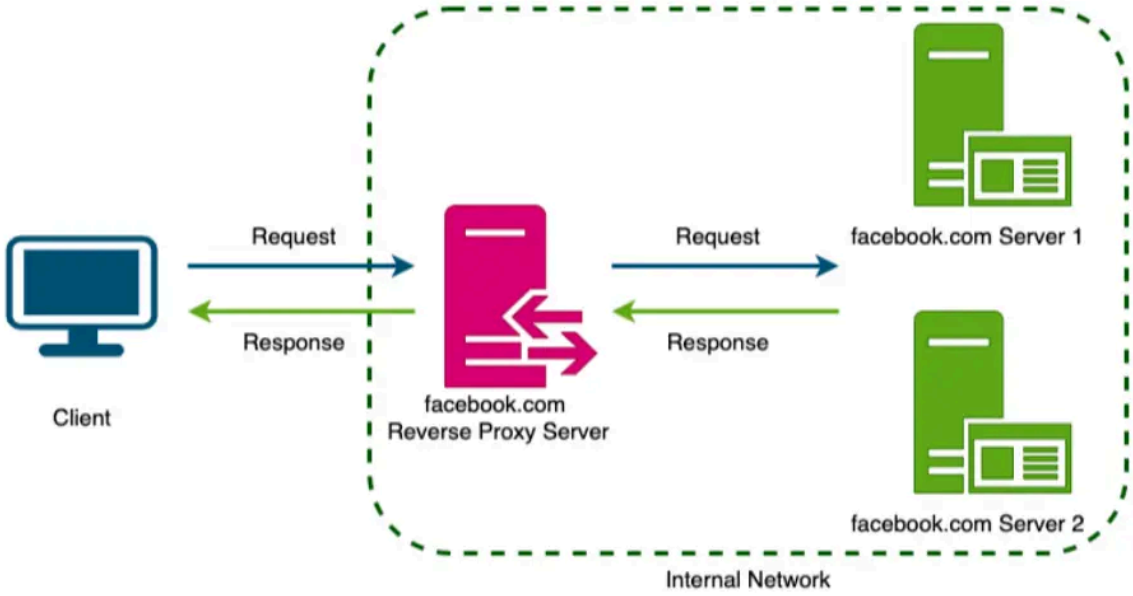
> *A forward proxy can hide the identity of the client from the server by sending requests on behalf of the client.*

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

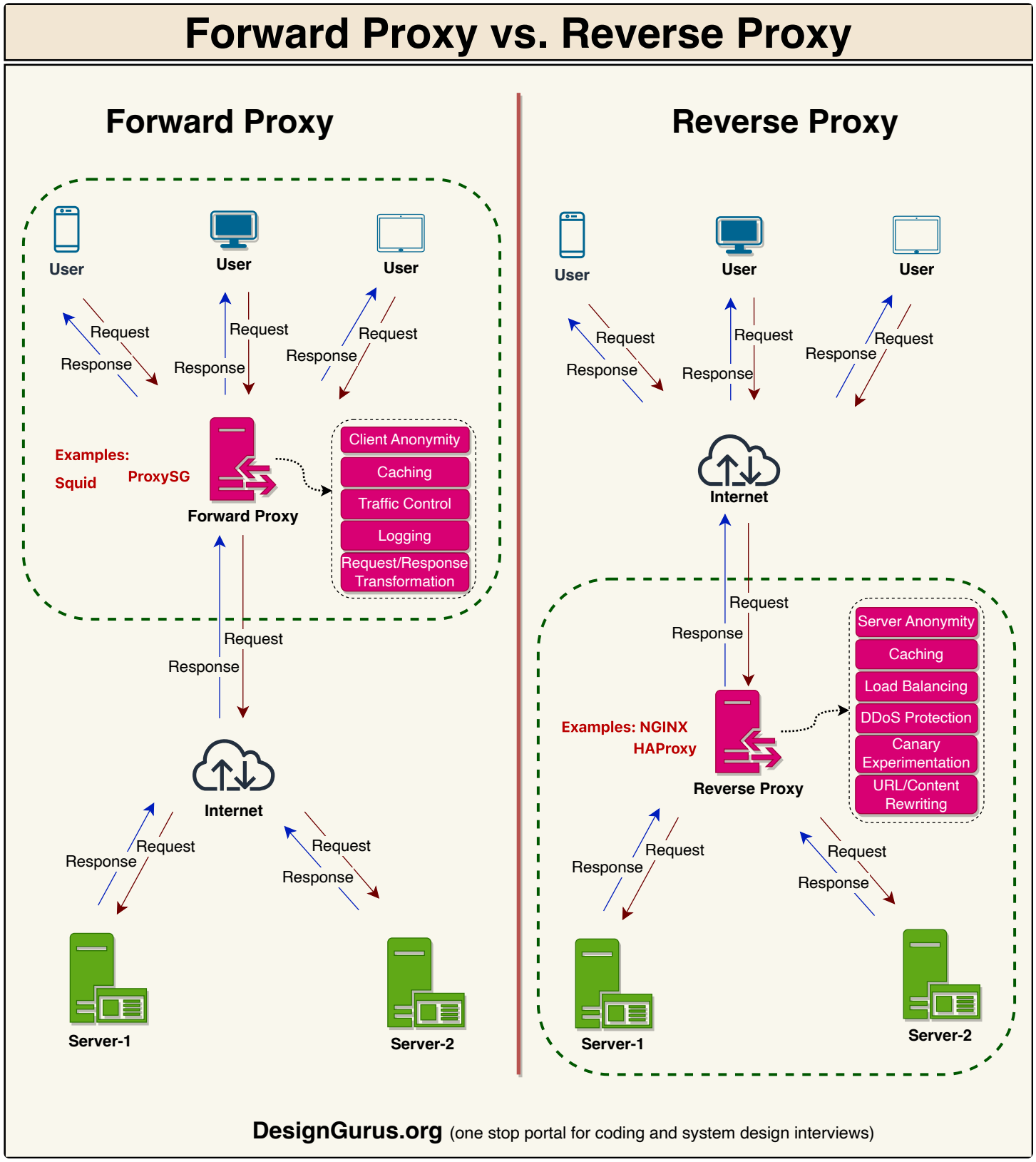
A reverse proxy retrieves resources from one or more servers on behalf of a client. These resources are then returned to the client, appearing as if they originated from the proxy server itself, thus anonymizing the server. Contrary to the forward proxy, which hides the client's identity, a reverse proxy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

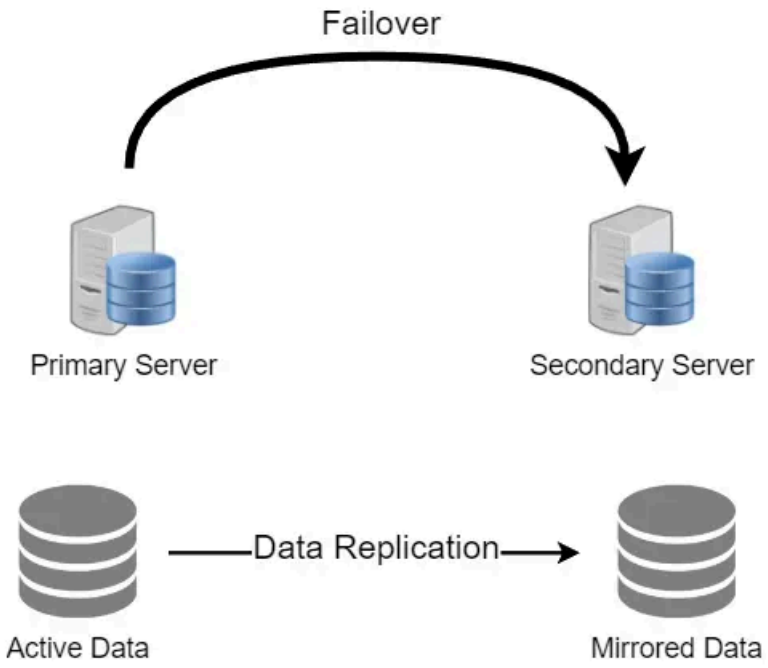
Summary

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.

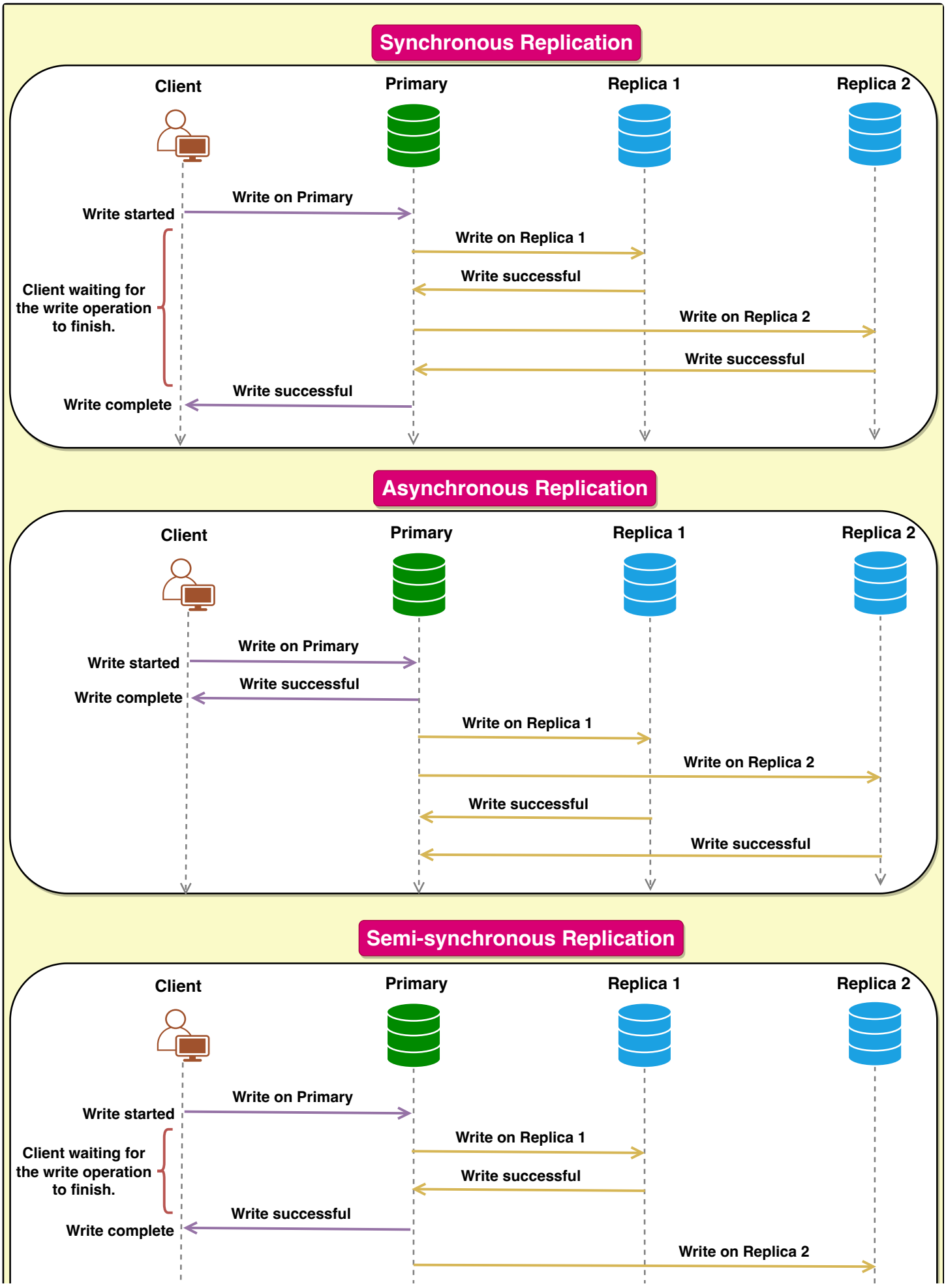
In asynchronous replication, there is a delay between the write operation on the primary database and the update on the replica databases. This delay can result in temporary inconsistencies between the primary and replica databases, as the data on the replica databases may not immediately reflect the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

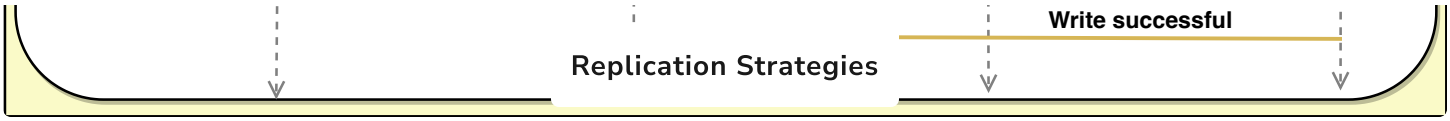
However, asynchronous replication can also have performance benefits, as write operations can be completed quickly without waiting for confirmation from the replica databases. In addition, if one or more replica databases are unavailable, the

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
Client
Primary
Replica 1
Replica 2
Write started
Write on Primary
Write on Replica 1
Write successful
Client waiting for the write operation to finish.
Write on Replica 2
Write successful
Write successful
Write complete
Asynchronous Replication
Client
Primary
Replica 1
Replica 2
Write started
Write on Primary
Write successful
Write complete
Write on Replica 1
Write on Replica 2
Write successful
Write successful
Semi-synchronous Replication
Client
Primary
Replica 1
Replica 2
Write started
Write on Primary
Write on Replica 1
Write successful
Client waiting for the write operation to finish.
Write successful
Write complete
Write on Replica 2


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:

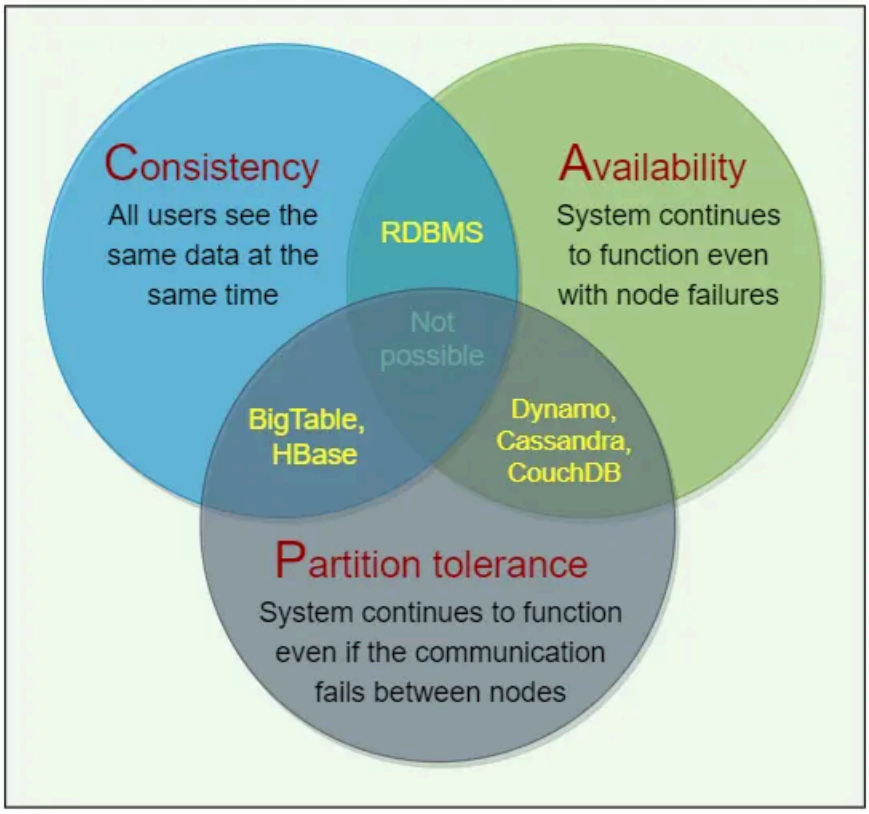
Consistency (C): All nodes see the same data at the same time. This means users can read or write from/to any node in the system and will receive the same data. It is equivalent to having a single up-to-date copy of the data.

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

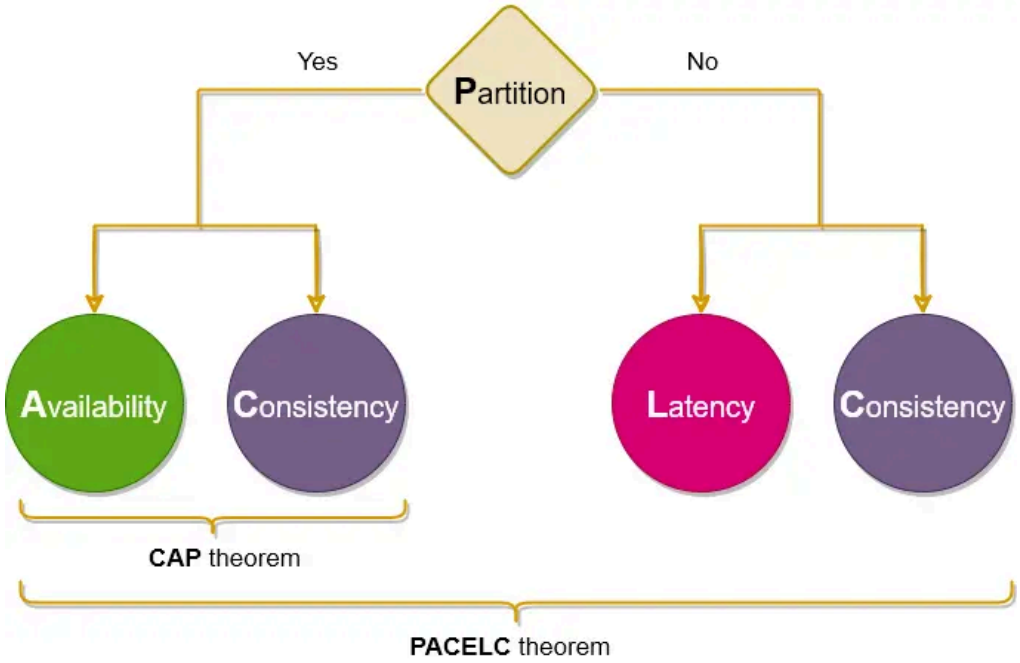
We cannot avoid partition in a distributed system, therefore, according to the CAP theorem, a distributed system should choose between consistency or availability. **ACID** (Atomicity, Consistency, Isolation, Durability) databases, such as RDBMSs like MySQL, Oracle, and Microsoft SQL Server, chose consistency (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.

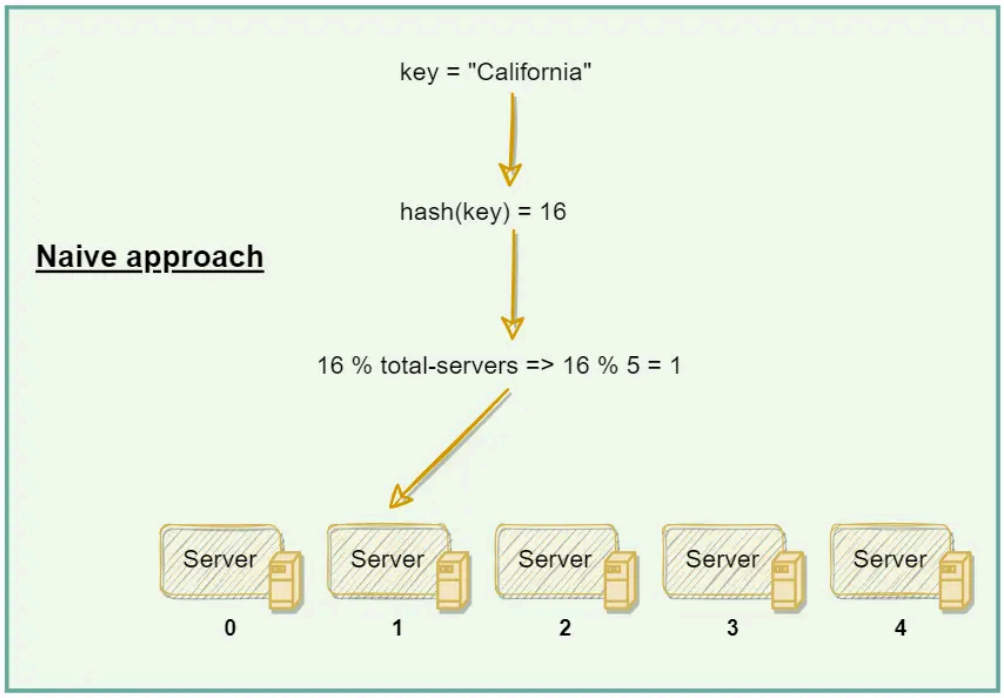
David Karger et al. first introduced Consistent Hashing in their 1997 paper and suggested its use in distributed caching. Later, Consistent Hashing was adopted and enhanced to be used across many distributed systems. In this lesson we will see how Consistent Hashing efficiently solves the problem of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

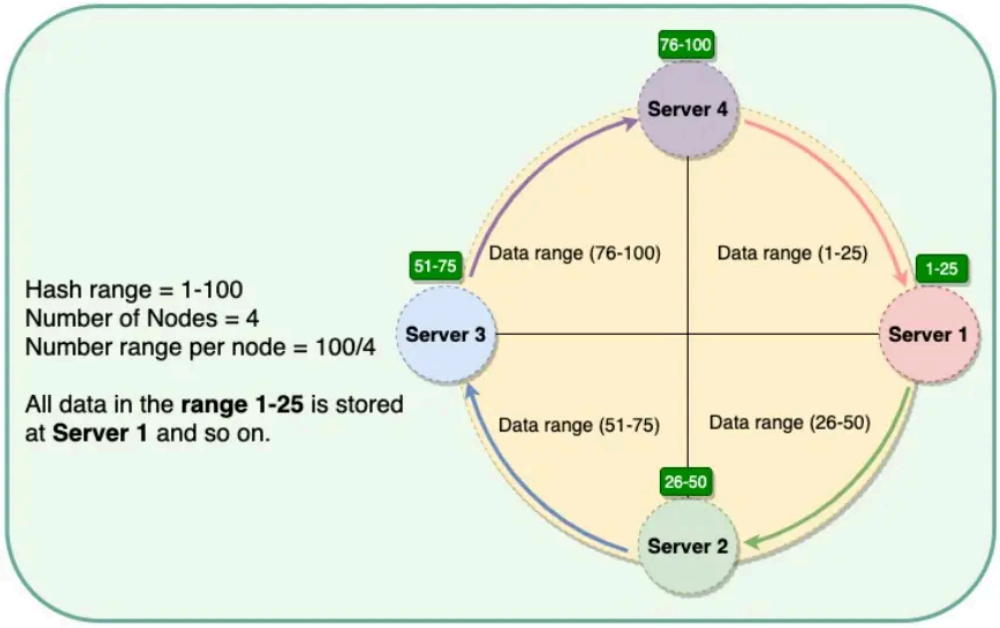
The scheme described in the above diagram solves the problem of finding a server for storing/retrieving the data. But when we add or remove a server, all our existing mappings will be broken. This is because the total number of servers will be changed, which was used to find the actual server storing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

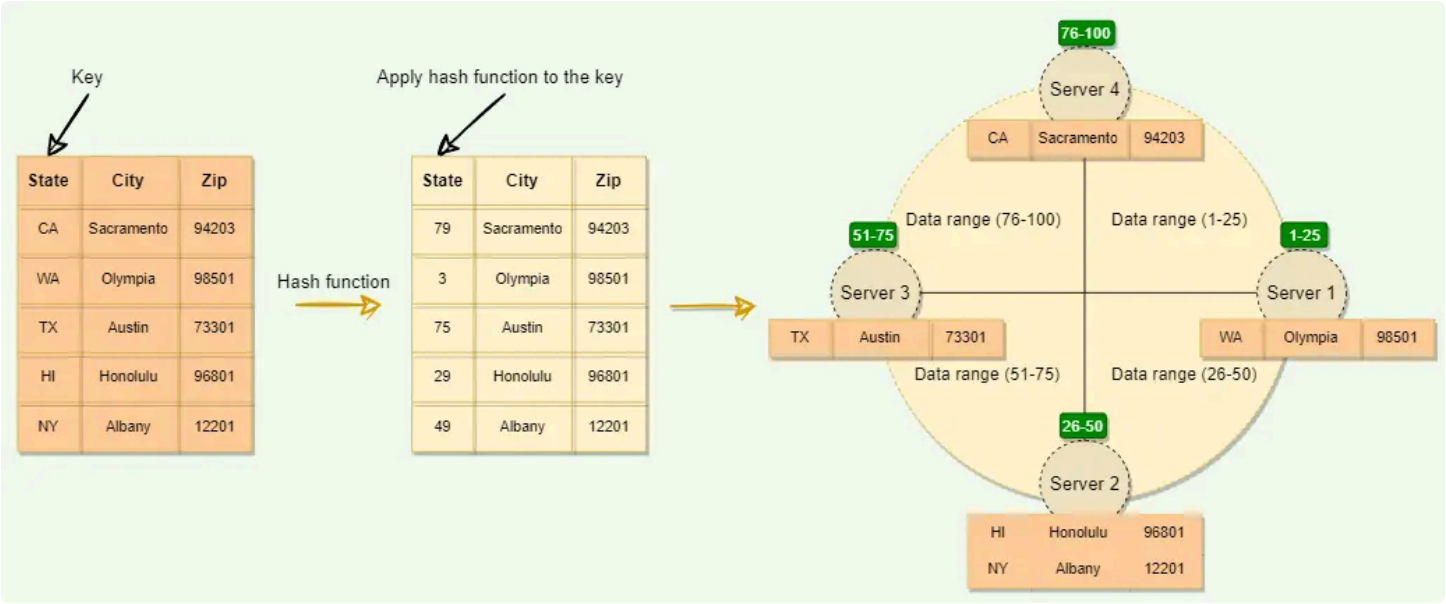
Range start: Token value

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

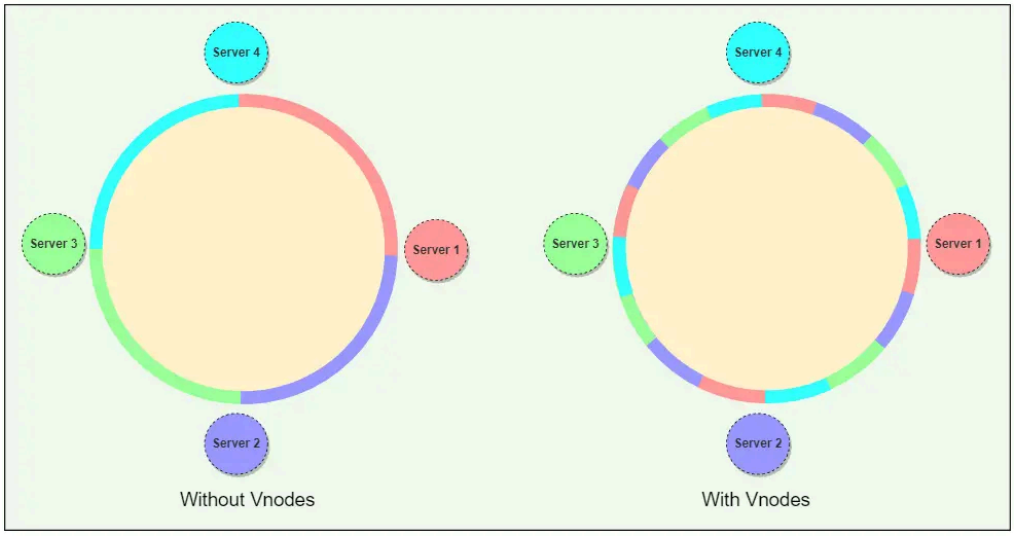
Adding and removing nodes in any distributed system is quite common. Existing nodes can die and may need to be decommissioned. Similarly, new nodes may be added to an existing cluster to meet growing demands. To efficiently handle these scenarios, Consistent Hashing makes use of virtual nodes (or Vnodes).

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

Vnodes gives the following advantages:

1. As Vnodes 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 Vnodes 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. Vnodes make it easier to maintain a cluster containing heterogeneous machines. This means, with Vnodes, we can assign a high number of sub-ranges to a powerful server and a lower number of sub-ranges to a less powerful server.
3. In contrast to one big range, since Vnodes help assign smaller ranges to each physical node, this decreases the probability of hotspots.

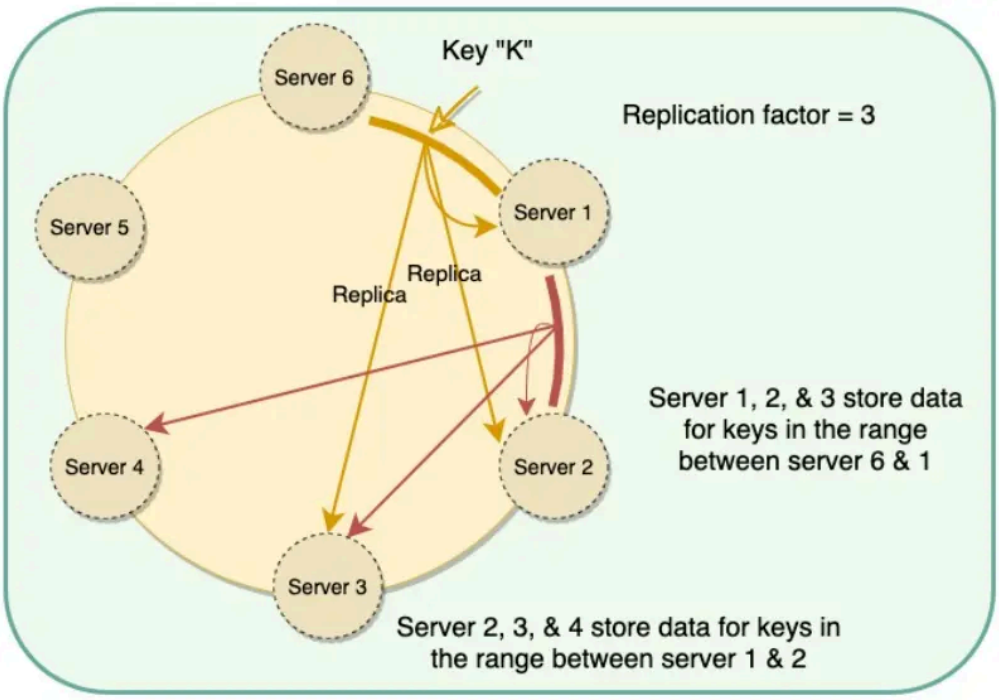
Data replication using Consistent Hashing

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 Nth 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

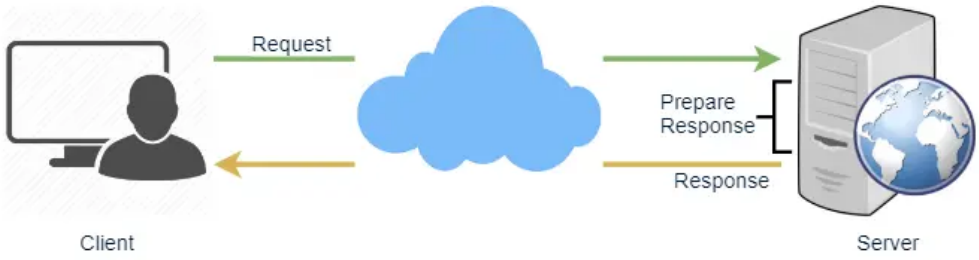
Amazon's Dynamo and Apache Cassandra use Consistent Hashing to distribute and replicate data across nodes.

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.



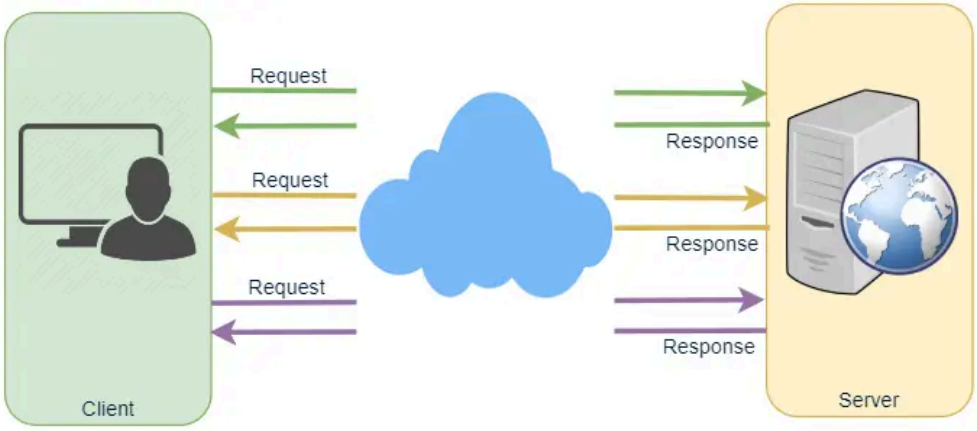
HTTP Protocol

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.



Ajax Polling Protocol

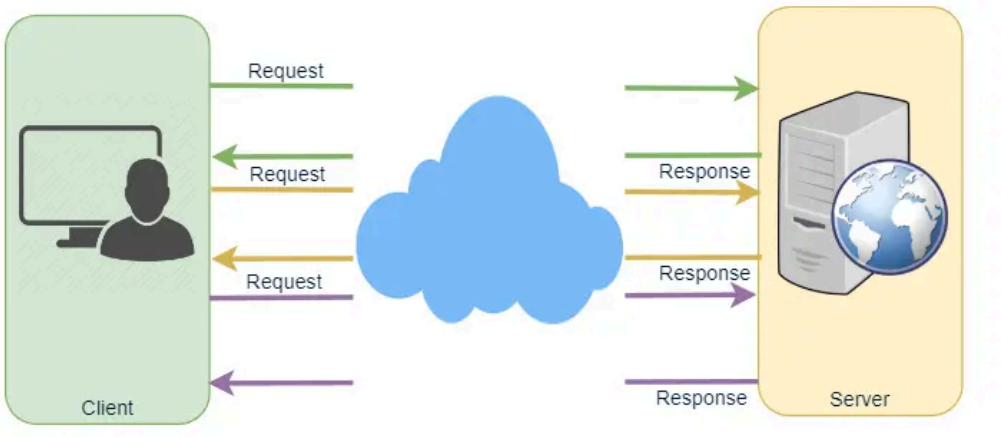
HTTP Long-Polling

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.

WebSockets Protocol

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.

Server Sent Events Protocol

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

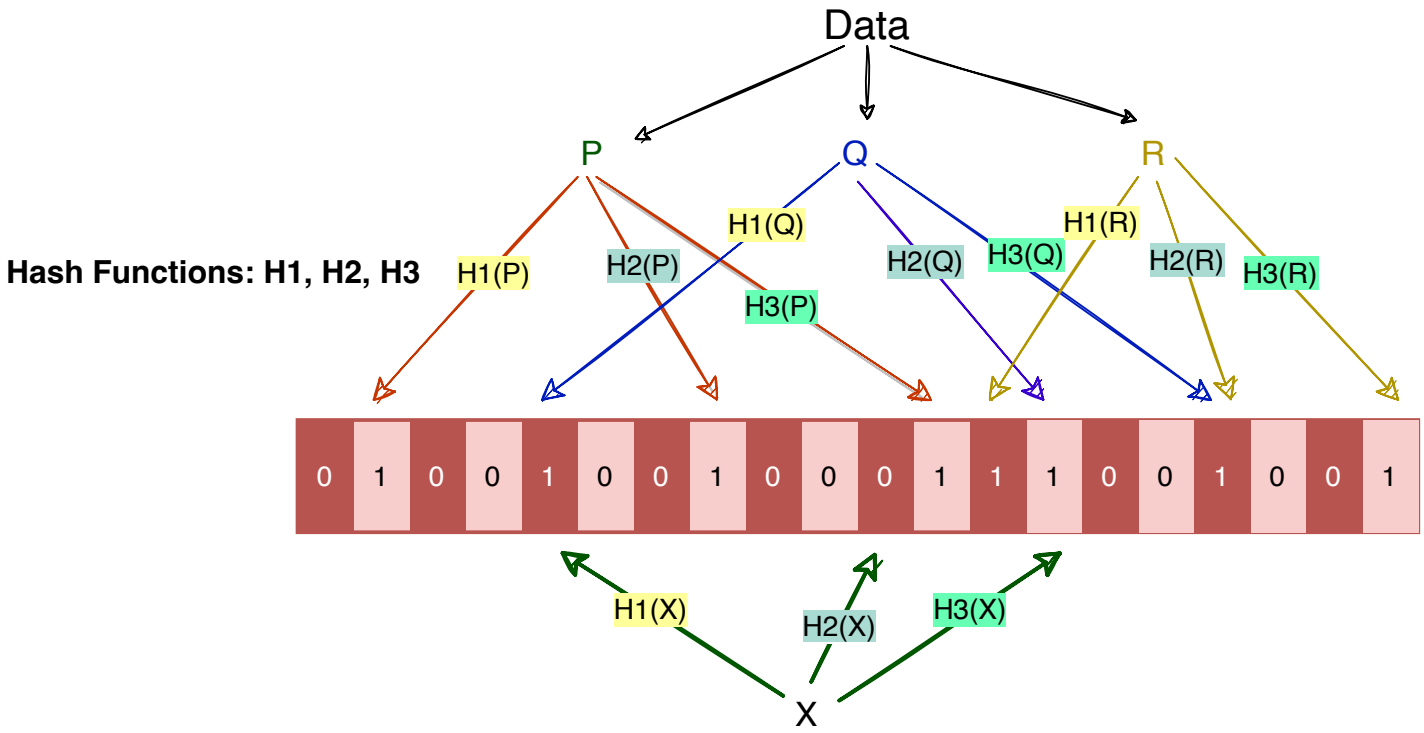
Use Bloom filters to quickly find if an element might be present in a set.

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.



Bloom Filter Using 3 Hash Functions

- 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 ⇒ 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.

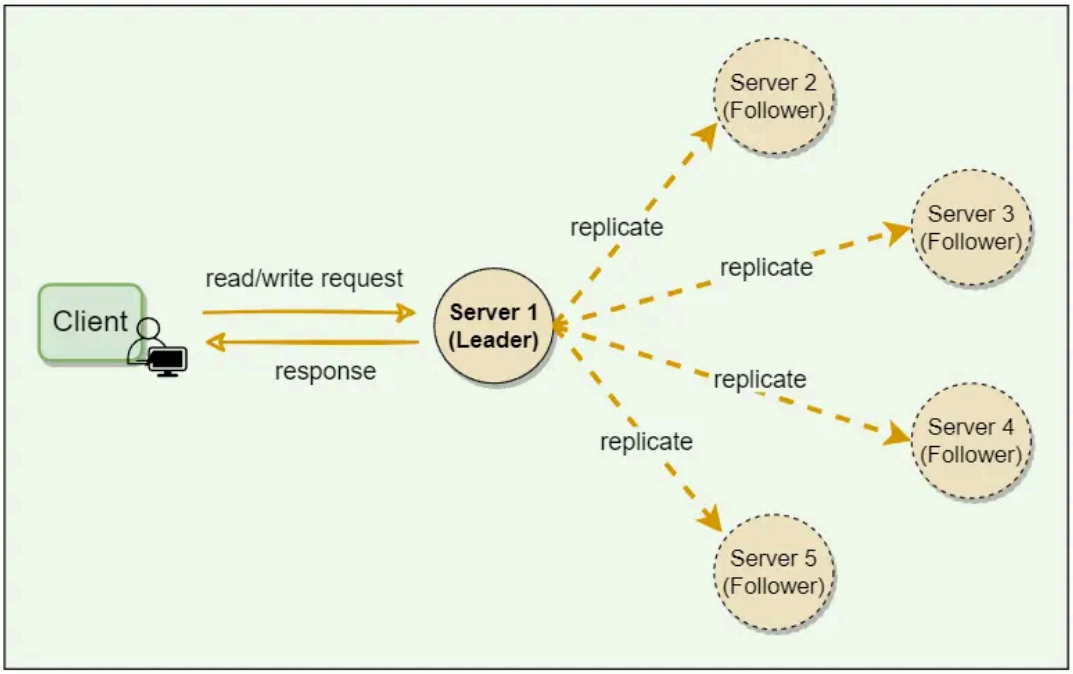
Background

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.