System Design Basics

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

a. What are the different architectural pieces that can be used?
b. How do these pieces work with each other?
c. How can we best utilize these pieces: what are the right trade-offs?

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

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**.

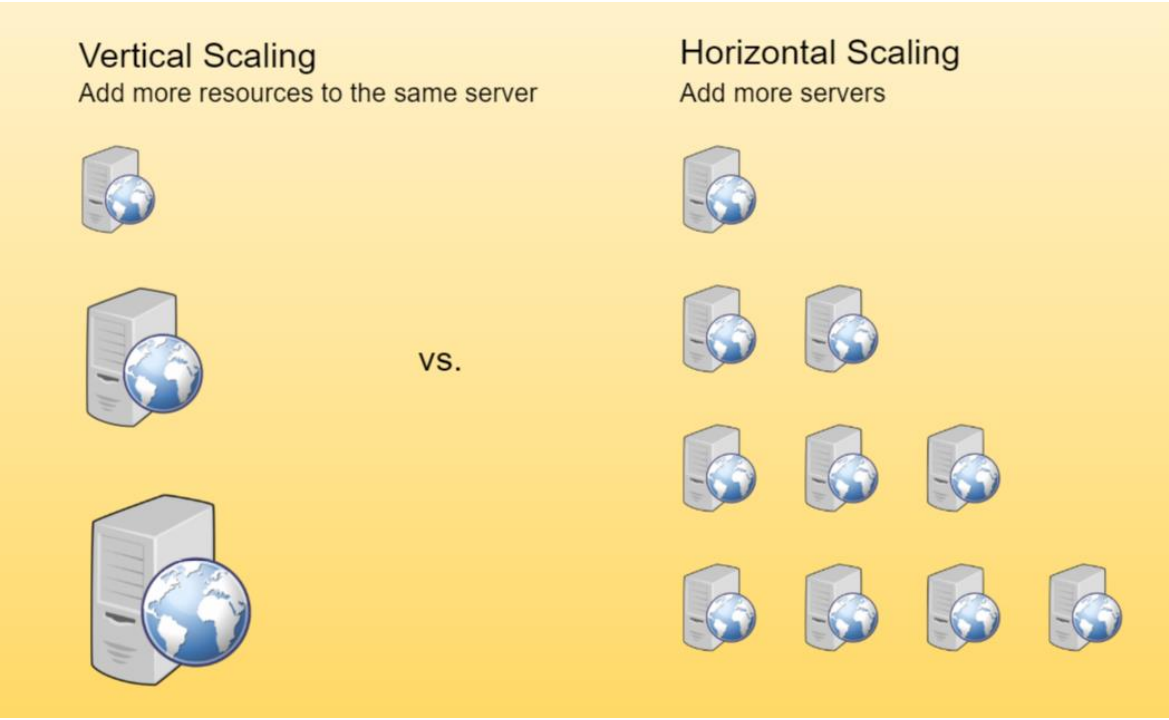

figure 1: Vertical vs Horizontal scaling

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: (figure 1) 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.

Reliability

By definition, reliability is the **probability a system will fail** in a given period. 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 requirements is that any user transaction should never be cancelled 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 considers 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 consider all these performance factors; therefore, we have to live with rough but robust estimates of the system behaviour.

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 centre (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.

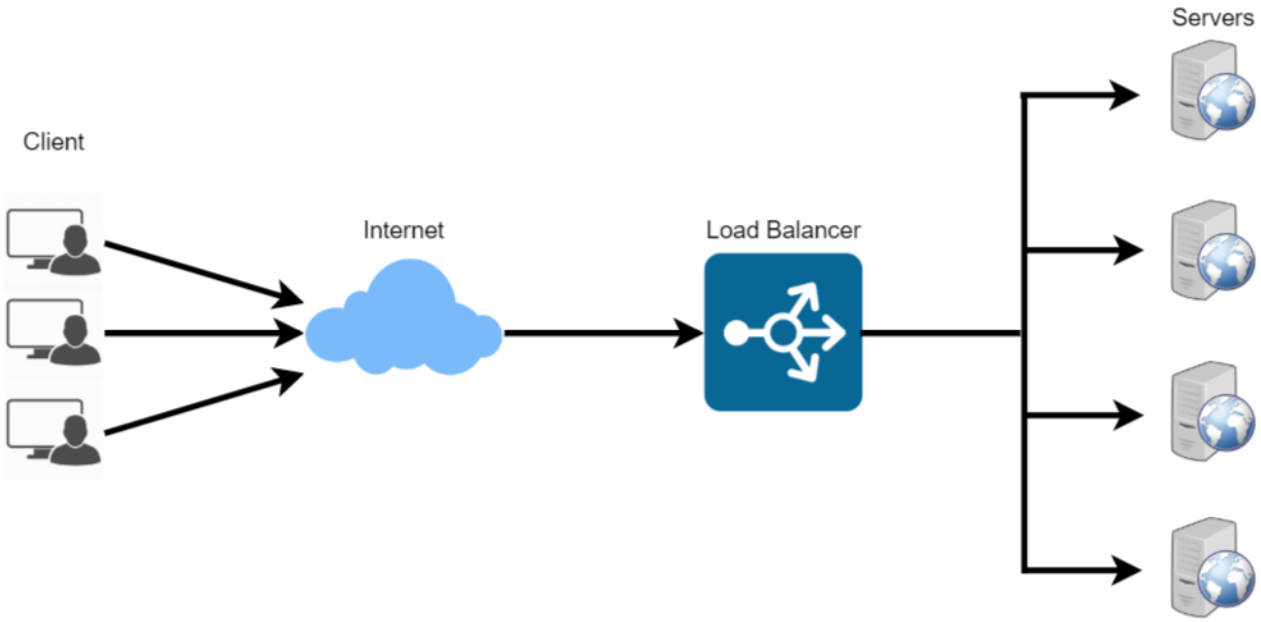

figure 1: A Load Balancer

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: (refer figure 2)

- 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.

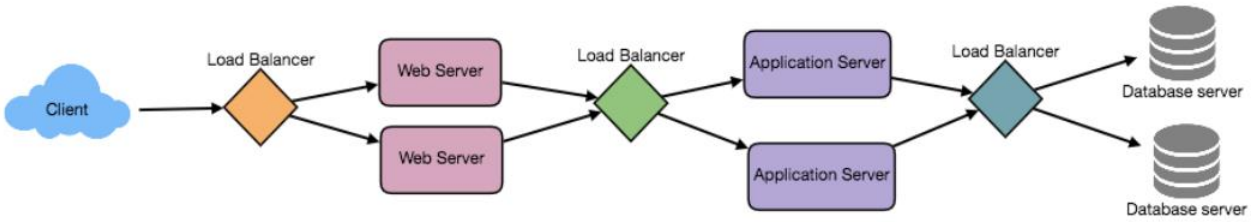

figure 2: Possible locations of LBs

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 insight. 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.

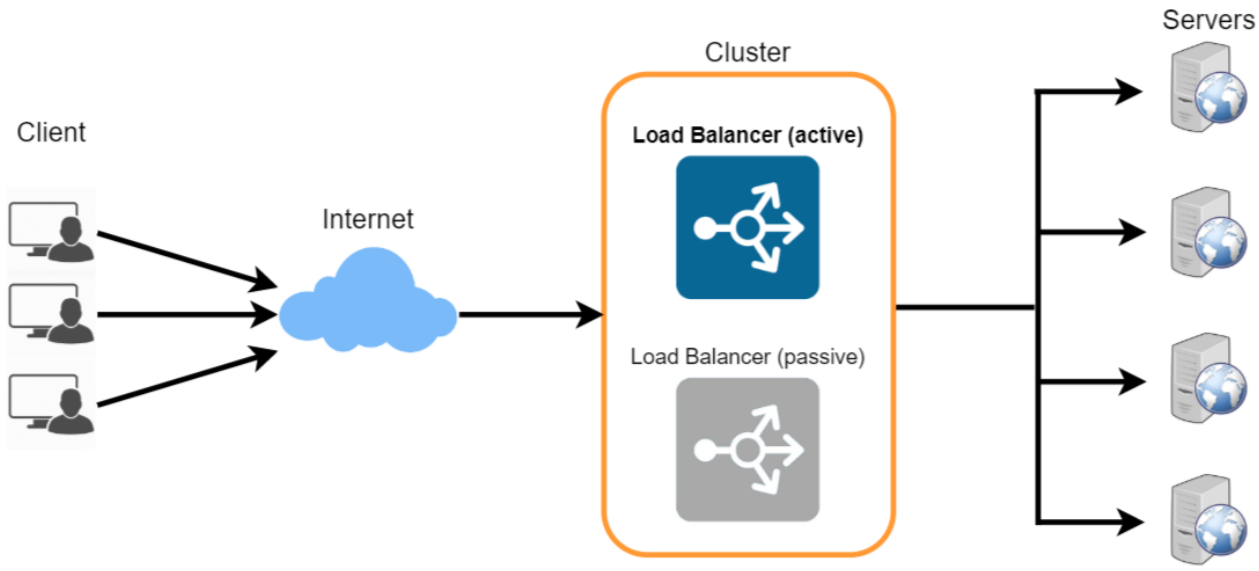

figure 3: A redundant Load Balancer

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 layer of computing: 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 local cached data if it exists. If it is not in the cache, the requesting node will query the data from 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 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 isn't yet 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 does require some maintenance for keeping 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 behaviour.

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 at the same time. 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 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 a technique to **break up a big database (DB) into many smaller parts**. It is the process of splitting up a DB/table across multiple machines to improve the manageability, performance, availability, and load balancing of an application. The justification for data partitioning is that, after a certain scale point, it is cheaper and more feasible to scale horizontally by adding more machines than to grow it vertically by adding beefier servers.

1. Partitioning Methods

There are many different schemes one could use to decide how to break up an application database into multiple smaller DBs. Below are three of the most popular schemes used by various large-scale applications.

a. Horizontal partitioning: In this scheme, we put different rows into different tables. For example, if we are storing different places in a table, we can decide that locations with ZIP codes less than 10000 are stored in one table and places with ZIP codes greater than 10000 are stored in a separate table. This is also called a range-based partitioning as we are storing different ranges of data in separate tables. Horizontal partitioning is also called as **Data Sharding**.

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. In the previous example, splitting location based on their zip codes assumes that places will be evenly distributed across the different zip codes. This assumption is not valid as there will be a lot of places in a thickly populated area like Manhattan as compared to its suburb cities.

b. Vertical Partitioning: In this scheme, we divide our data to store tables related to a specific feature in their own server. For example, if we are building *Instagram* like application - where we need to store data related to users, photos they upload, and people they follow - we can decide to place user profile information on one DB server, friend lists on another, and photos on a third server.

Vertical partitioning is straightforward to implement and has a low impact on the application. The main problem with this approach is that if our application experiences additional growth, then it may be necessary to further partition a feature specific DB across various servers (e.g. it would not be possible for a single server to handle all the metadata queries for 10 billion photos by 140 million users).

c. Directory Based Partitioning: A loosely coupled approach to work around issues mentioned in the above schemes is to create a lookup service which knows your current partitioning scheme and abstracts it away from the DB access code. So, to find out where a particular data entity resides, we query the directory server that holds the mapping between

each tuple key to its DB server. This loosely coupled approach means we can perform tasks like adding servers to the DB pool or changing our partitioning scheme without having an impact on the application.

2. Partitioning Criteria

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 which 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 de-normalize 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 all the perils of denormalization 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 of RDBMS do not support foreign keys constraints across databases on different database servers. Which means that 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 catalogue

A library catalogue is a register that contains the list of books found in a library. The catalogue is organized like a database table generally with four columns: book title, writer, subject, and date of publication. There are usually two such catalogues: 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 catalogues 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:

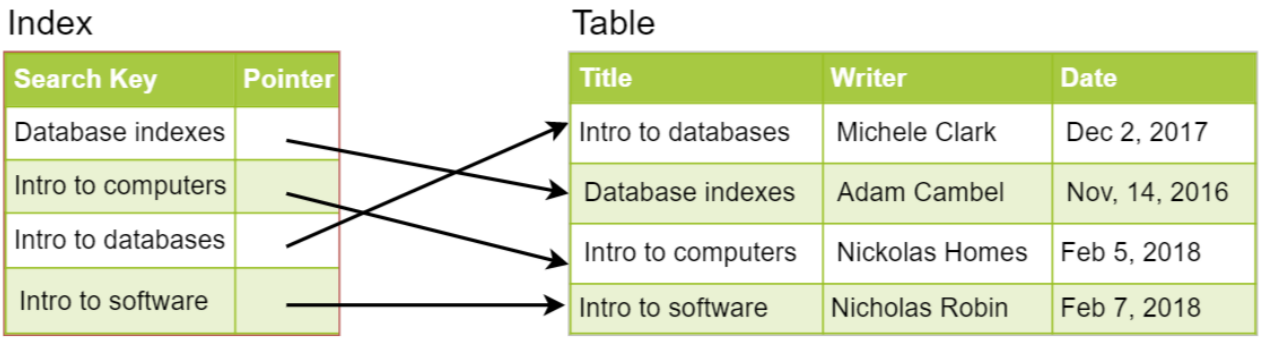

figure 1: A library catalogue index

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

A proxy server is an **intermediate server between the client and the back-end server**. Clients connect to proxy servers to make a request for a service like a web page, file, connection, etc. In short, a proxy server is a piece of software or hardware that acts as an intermediary for requests from clients seeking resources from other servers.

Typically, proxies are used to filter requests, log requests, or sometimes transform requests (by adding/removing headers, encrypting/decrypting, or compressing a resource). Another advantage of a proxy server is that **its cache can serve a lot of requests**. If multiple clients access a particular resource, the proxy server can cache it and serve it to all the clients without going to the remote server.

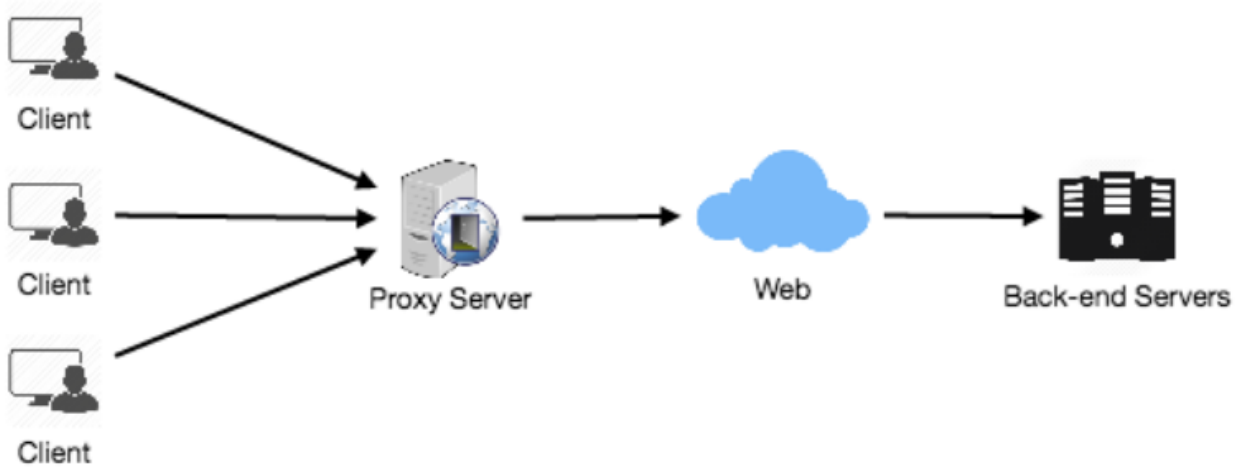

figure 1: A proxy server setup

Proxy Server Types

Proxies can reside on the client's local server or anywhere between the client and the remote servers. Here are a few famous types of proxy servers:

Open Proxy

An open proxy is a proxy server that is accessible by any Internet user. Generally, a proxy server only allows users within a network group (i.e. a closed proxy) to store and forward Internet services such as DNS or web pages to reduce and control the bandwidth used by the group. With an open proxy, however, any user on the Internet is able to use this forwarding service. There two famous open proxy types:

1. **Anonymous Proxy** - This proxy reveals its identity as a server but does not disclose the initial IP address. Though this proxy server can be discovered easily it can be beneficial for some users as it hides their IP address.
2. **Transparent Proxy** – This proxy server again identifies itself, and with the support of HTTP headers, the first IP address can be viewed. The main benefit of using this sort of server is its ability to cache the websites.

Reverse Proxy

A reverse proxy retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client, appearing as if they originated from the proxy server itself.

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.

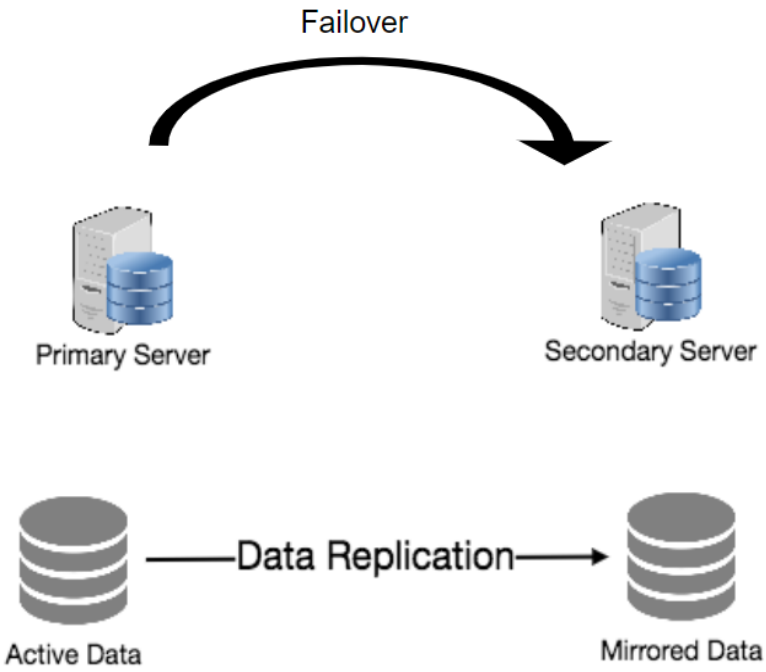

figure 1: Redundancy and Replication in practice

Replication means **sharing information to ensure consistency** between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or accessibility.

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

SQL vs. NoSQL

In the world of databases, there are two main types of solutions:

1. SQL and (relational databases)
2. NoSQL (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 analysing 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 centres 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.