System design book

What's the Difference Between Throughput and Latency?

Latency and throughput are two metrics that measure the performance of a computer network. Latency is the delay in network communication. It shows the time that data takes to transfer across the network. Networks with a longer delay or lag have high latency, while those with fast response times have lower latency. In contrast, throughput refers to the average volume of data that can actually pass through the network over a specific time. It indicates the number of data packets that arrive at their destinations successfully and the data packet loss.

Read about latency »

Why are throughput and latency important?

You can determine network speed by looking at how quickly a network can transfer data packets to their destinations. This speed is the result of network performance factors like latency and throughput.

Latency determines the delay that a user experiences when they send or receive data from the network. Throughput determines the number of users that can access the network at the same time.

A network with low throughput and high latency struggles to send and process high data volume, which results in congestion and poor application performance. In contrast, a network with high throughput and low latency is responsive and efficient. Users experience improved performance and increased satisfaction.

High-performing networks directly impact revenue generation and operational efficiency. In addition, certain use cases—like real-time streaming, Internet of Things (IoT) data analytics, and high-performance computing—require certain network performance thresholds to operate optimally.

Key differences: network latency vs. throughput

Although latency and throughput both contribute to a reliable and fast network, they are not the same. These network metrics focus on distinct statistics and are different from each other.

How to measure

You can measure network latency by measuring ping time. This process is where you transmit a small data packet and receive confirmation that it arrived.

Most operating systems support a *ping* command which does this from your device. The round-trip-time (RTT) displays in milliseconds and gives you an idea of how long it takes for your network to transfer data.

You can measure throughput either with network testing tools or manually. If you wanted to test throughput manually, you would send a file and divide the file size by the time it takes to arrive. However, latency and bandwidth impact throughput. Because of this, many people use network testing tools, as the tools report throughput alongside other factors like bandwidth and latency.

Read about RTT in networking »

Unit of measurement

You measure latency in milliseconds. If you have a low number of milliseconds, your network is only experiencing a small delay. The higher the number in milliseconds, the slower the network is performing.

Originally, you would measure network throughput in bits per second (bps). But, as data transmission technologies have improved, you can now achieve much higher values. Because of this, you can measure throughput in kilobytes per second (KBps), megabytes per second (MBps), and even gigabytes per second (GBps). One byte is equal to eight bits.

Impacting factors: latency vs throughput

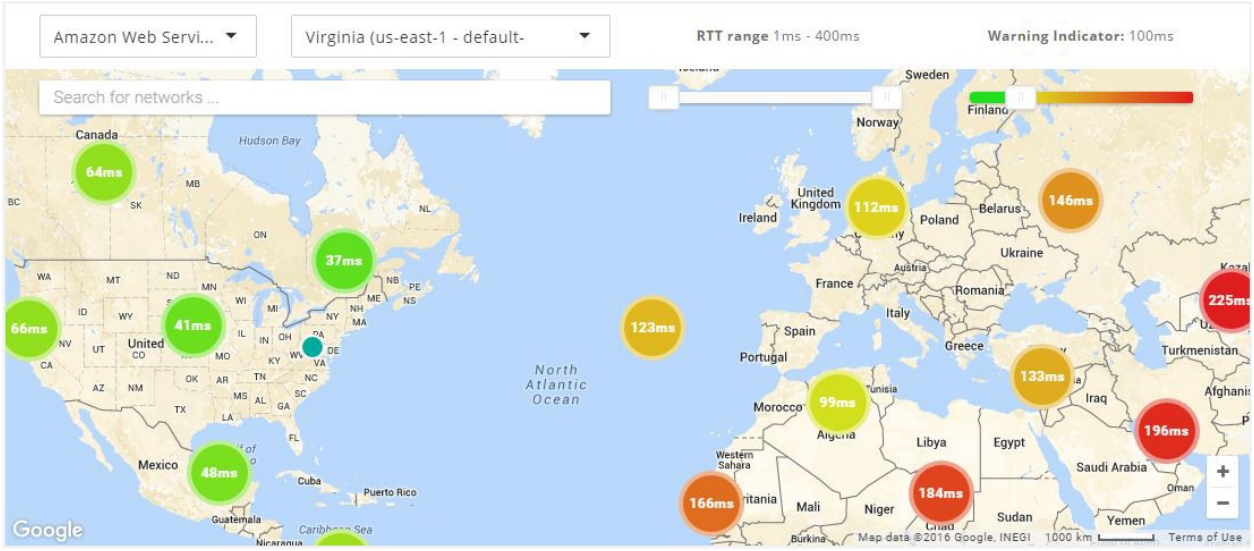
Different factors can impact your latency and throughput metrics.

Latency

Latency has several factors that contribute to it being high or low.

Location

One of the most important factors is the location of where data originates and its intended destination. If your servers are in a different geographical region from your device, the data has to travel further, which increases latency. This factor is called *propagation*.



Network congestion

Network congestion occurs when there is a high volume of data being transmitted over a network. The increased traffic on the network causes packets to take longer routes to their destination.

Protocol efficiency

Some networks require additional protocols for security. The extra handshake steps create a delay.

Network infrastructure

Network devices can become overloaded, which results in dropped packets. As packets are delayed or dropped, devices retransmit them. This adds additional latency.

Throughput

Throughput speeds are directly impacted by other factors.

Bandwidth

If your network capacity has reached the maximum bandwidth of your transmission medium, its throughput will never be able to go beyond that limit.

Processing power

Certain network devices have specialized hardware or software optimizations that improve their processing performance. Some examples are dedicated application-specific integrated circuits or software-based packet processing engines.

These optimizations enable the device to handle higher volumes of traffic and more complex packet processing tasks, which leads to higher throughput.

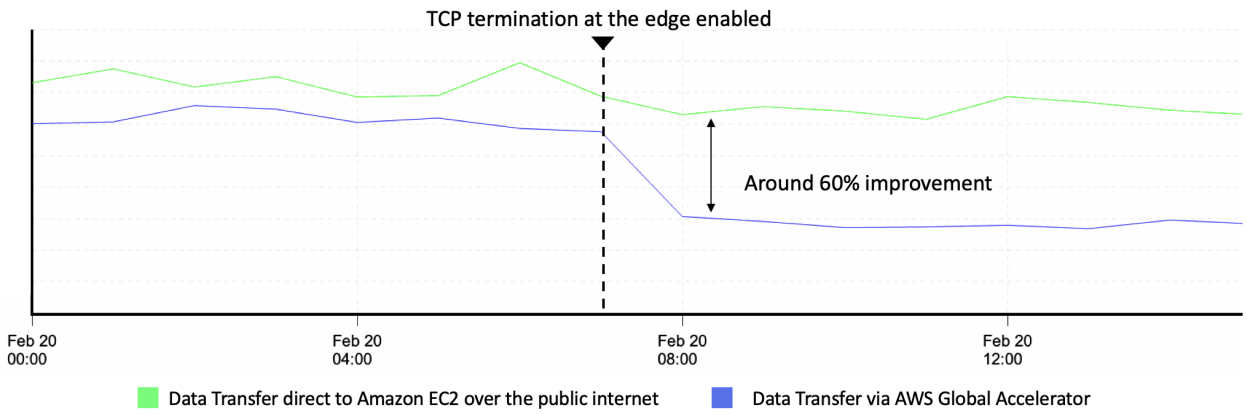
Packet loss

Packet loss can occur for a variety of reasons, including network congestion, faulty hardware, or misconfigured network devices. When packets are lost, they must be retransmitted. This results in delays and reduces the overall throughput of the network.

Network topology

Network topology refers to the number of network devices, the bandwidth of the network links, and the distance between devices in a network path.

A well-designed network topology provides multiple paths for data transmission, reduces traffic bottlenecks, and increases throughput. Networks with more devices or longer distances require complex network topologies to achieve high throughput.



Relationship between bandwidth, latency and throughput

Latency and throughput work together to deliver high network connectivity and performance. As both impact the transmission of data packets, they also affect one another.

If a network connection has high latency, it can have lower throughput, as data takes longer to transmit and arrive. Low throughput also makes it seem like a network has high latency, as it takes longer for large quantities of data to arrive.

As they are closely linked, you must monitor both latency and throughput to achieve high network performance.

Bandwidth and network throughput

Bandwidth represents the total volume of data that you can transfer over a network. Your total bandwidth refers to the theoretical maximum amount of data that you could transfer over a network. You measure it in megabytes per second (MBps). You can think of bandwidth as the theoretical maximum throughput of your network.

Bandwidth is how much data you can transfer, while throughput is the actual amount of data you transmit in any given moment based on real-world network limitations. A high bandwidth does not guarantee speed or a good network performance, but a higher bandwidth leads to higher throughput.

How can you improve latency and throughput?

To improve latency, you can shorten the propagation between the source and destination. You can improve throughput by increasing the overall network bandwidth.

Next, we give some suggestions to improve latency and throughput together.

Caching

Caching in networking refers to the process of storing frequently accessed data geographically closer to the user. For example, you can store data in proxy servers or content delivery networks (CDNs).

Your network can deliver data from the cached location much faster than if it had to be retrieved from the original source. And the user receives data much faster, improving latency. Additionally, because the data is retrieved from a cache, it reduces the load on the original source. This allows it to handle more requests at once, improving throughput.

Transport protocols

By optimizing the transport protocol that you use for specific applications, you can improve network performance.

For instance, TCP and UDP are two common network protocols. TCP establishes a connection and checks that you receive data without any errors. Because of its goal of reducing packet loss, TCP has higher latency and higher throughput. UDP does not check for packet loss or errors, transmitting several duplicate packets instead. So, it gives minimal latency but a higher throughput.

Depending on the application that you are using, TCP or UDP may be the better choice. For example, TCP is useful for transferring data, while UDP is useful for video streaming and gaming.

Quality of service

You can use a quality of service (QoS) strategy to manage and optimize network performance. QoS allows you to divide network traffic into specific categories. You can assign each category a priority level.

Your QoS configurations prioritize latency-sensitive applications. Some applications and users experience lower latency than others. Your QoS configurations can also prioritize data by type, reducing packet loss and increasing throughput for certain users

Summary of differences: throughput vs. latency

	Throughput	Latency
What does it measure?	Throughput measures the volume of data that passes through a network in a given period. Throughput impacts how much data you can transmit in a period of time.	Latency measures the time delay when sending data. A higher latency causes a network delay.
How to measure?	Manually calculate throughput by sending a file or using network testing tools.	Calculate latency by using ping times.
Unit of measurement	Megabytes per second (MBps).	Milliseconds (ms).
Impacting factors	Bandwidth, network processing power, packet loss, and network topology.	Geographical distances, network congestion, transport protocol, and network infrastructure.

NoSQL vs. SQL Databases

TL;DR: NoSQL (“non SQL” or “not only SQL”) databases were developed in the late 2000s with a focus on scaling, fast queries, allowing for frequent application changes, and making programming simpler for developers. Relational databases accessed with SQL (Structured Query Language) were developed in the 1970s with a focus on reducing data duplication as storage was much more costly than developer time. SQL databases tend to have rigid, complex, tabular schemas and typically require expensive vertical scaling.

If you’re not familiar with what NoSQL databases are or the different types of NoSQL databases, ***start here****.*

Overview

Below is an overview of what this article covers.

- **What are the differences between SQL and NoSQL?**
- **What are the benefits of NoSQL databases?**
- **What are the drawbacks of NoSQL databases?**
- **Try a NoSQL database**

Differences between SQL and NoSQL

The table below summarizes the main differences between SQL and NoSQL databases.

	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune
Primary Purpose	General purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Rigid	Flexible
Scaling	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)
Multi-Record ACID Transactions	Supported	Most do not support multi-record ACID transactions. However, some — like MongoDB — do.
Joins	Typically required	Typically not required
Data to Object Mapping	Requires ORM (object-relational mapping)	Many do not require ORMs. MongoDB documents map directly to data structures in most popular programming languages.

What are the benefits of NoSQL databases?

NoSQL databases offer many benefits over relational databases. NoSQL databases have flexible data models, scale horizontally, have incredibly fast queries, and are easy for developers to work with.

- **Flexible data models**

  NoSQL databases typically have very flexible schemas. A flexible schema allows you to easily make changes to your database as requirements change. You can iterate quickly and continuously integrate new application features to provide value to your users faster.

- **Horizontal scaling**

  Most SQL databases require you to scale-up vertically (migrate to a larger, more expensive server) when you exceed the capacity requirements of your current server. Conversely, most NoSQL databases allow you to scale-out horizontally, meaning you can add cheaper commodity servers whenever you need to.

- **Fast queries**

  Queries in NoSQL databases can be faster than SQL databases. Why? Data in SQL databases is typically normalized, so queries for a single object or entity require you to join data from multiple tables. As your tables grow in size, the joins can become expensive. However, data in NoSQL databases is typically stored in a way that is optimized for queries. The rule of thumb when you use MongoDB is **data that is accessed together should be stored together.** Queries typically do not require joins, so the queries are very fast.

- **Easy for developers**

Some NoSQL databases like MongoDB map their data structures to those of popular programming languages. This mapping allows developers to store their data in the same way that they use it in their application code. While it may seem like a trivial advantage, this mapping can allow developers to write less code, leading to faster development time and fewer bugs.

What are the drawbacks of NoSQL databases?

One of the most frequently cited drawbacks of NoSQL databases is that they don't support ACID (atomicity, consistency, isolation, durability) transactions across multiple documents. With appropriate schema design, single-record atomicity is acceptable for lots of applications. However, there are still many applications that require ACID across multiple records.

To address these use cases, MongoDB added support for **multi-document ACID transactions** in the 4.0 release, and extended them in 4.2 to span sharded clusters.

Since data models in NoSQL databases are typically optimized for queries and not for reducing data duplication, NoSQL databases can be larger than SQL databases. Storage is currently so cheap that most consider this a minor drawback, and some NoSQL databases also support compression to reduce the storage footprint.

Depending on the NoSQL database type you select, you may not be able to achieve all of your use cases in a single database. For example, graph databases are excellent for analyzing relationships in your data but may not provide what you need for everyday retrieval of the data such as range queries. When selecting a NoSQL database, consider what your use cases will be and if a general purpose database like MongoDB would be a better option.

How to try a NoSQL database

Now that you understand the basics of NoSQL databases, you're ready to give them a shot.

You can check out the **Where to Use MongoDB white paper** to help you determine if MongoDB or another database is right for your use case. Then, hop on over to **What Is a Document Database?** to learn about the document model and how it compares to the relational model.

For those who like to jump right in and learn by doing, one of the easiest ways to get started with NoSQL databases is to use **MongoDB Atlas**. Atlas is MongoDB's fully managed, global database service that is available on all of the leading cloud providers. One of the many handy things about Atlas is that it has a generous, forever-free tier so you can create a database and discover all of the benefits of NoSQL databases firsthand without providing your credit card.

For those who prefer structured learning, **MongoDB University** offers completely free online training that will walk you step by step through the process of learning MongoDB.

When you're ready to interact with MongoDB using your favorite programming language, check out the **Quick Start Tutorials**. These tutorials will help you get up and running as quickly as possible in the language of your choice.

CAP Theorem Definition

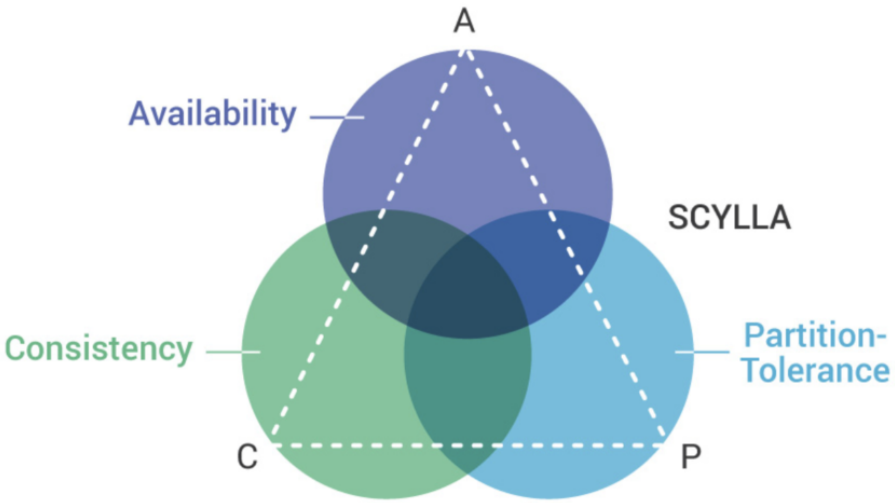
In computer science, the CAP theorem, sometimes called CAP theorem model or Brewer's theorem after its originator, Eric Brewer, states that any distributed system or data store can simultaneously provide only two of three guarantees: consistency, availability, and partition tolerance (CAP).

During times of normal operations, a data store covers all three. However, according to the CAP theorem, a distributed database system can provide **either** consistency **or** availability when it experiences a network

failure. In other words, in case of a network failure, it's a tradeoff between consistency or availability, and that choice must be made in advance.

The theorem states that a distributed database requires partition tolerance. The distributed data store, by nature, operates with network partitions, so because network failures are inevitable, partition tolerance is necessary to reliable service.This means users need to choose between C (consistency) and A (availability) in case a network failure occurs and both availability and consistency cannot be guaranteed. The cost of high availability is lower consistency. The price of high consistency is lower availability.

[See the CAP Theorem discussion in Martin Kleppmann's *Designing Data-Intensive Applications* book (free PDF)]



CAP Theorem FAQs

What is CAP Theorem?

Understanding the CAP theorem is simpler when you consider it piece by piece:

What is CAP theorem in distributed systems? A distributed network system stores data across multiple nodes—virtual or physical machines—simultaneously. It's essential to understand the CAP theorem when designing a cloud application because all cloud apps are distributed systems.

What is CAP theorem consistency? Consistency of CAP theorem means that regardless of the node they connect to, all clients see the same data at once. For the write to one node to succeed, it must also instantly be replicated or forwarded to all the other nodes in the system.

CAP theorem eventual consistency. Some NoSQL databases (for example, ScyllaDB) use a model of tunable eventual consistency to deliver multi datacenter high availability and fast and efficient write and read operations. In this case, all nodes are equal; this means any node can serve any request, there is no single point of coordination, and all nodes in the system continue to cooperatively provide service, even when nodes become unavailable. Eventual consistency supports modern workloads that are less dependent on strong consistency but rely heavily on availability.

What is availability in CAP theorem? CAP theorem availability means that even if one or more nodes are down, any client making a data request receives a response. In other words, when any request is made, without

exception, all working nodes in a distributed system return a valid response.

Partition tolerance in CAP theorem explained. In a distributed system, a partition is a break in communications —a temporarily delayed or lost connection between nodes. Partition tolerance means that in spite of any number of breakdowns in communication between nodes in the system, the cluster will continue to work.

What is CAP theorem in NoSQL? NoSQL databases and CAP theorem are closely linked. NoSQL databases are categorized based on which CAP characteristics they support: CP, AP, or CA.

A CP database offers consistency and partition tolerance but sacrifices availability. The practical result is that when a partition occurs, the system must make the inconsistent node unavailable until it can resolve the partition. MongoDB and Redis are examples of CP databases.

An AP database provides availability and partition tolerance but not consistency in the event of a failure. All nodes remain available when a partition occurs, but some might return an older version of the data. CouchDB, Cassandra, and ScyllaDB are examples of AP databases.

A CA database delivers consistency and availability, but it can't deliver fault tolerance if any two nodes in the system have a partition between them. Clearly, this is where CAP theorem and NoSQL databases collide: there are no NoSQL databases you would classify as CA under the CAP theorem. In a distributed database, there is no way to avoid system partitions. So, although CAP theorem stating a CA distributed database is possible exists, there is currently no true CA distributed database system. The modern goal of CAP theorem analysis should be for system designers to generate optimal combinations of consistency and availability for particular applications.

CAP Theorem vs ACID

Although the "C" in both ACID and CAP theorem refer to consistency, consistency in CAP is different than in ACID. In CAP, consistency means having information that is the most up-to-date. Consistency in ACID refers to the hardness of the database that protects it from corruption despite the addition of new transactions and references different database events.

Database systems such as RDBMS that are designed in part based on traditional ACID guarantees choose consistency over availability. In contrast, systems common in the NoSQL movement designed around the BASE philosophy select availability over consistency.

CAP Theorem NoSQL Database Examples

To better comprehend the role of CAP theorem in database theory, consider some CAP theorem databases examples.

MongoDB is a popular NoSQL database management system that is used for big data applications running across multiple locations. MongoDB resolves network partitions by maintaining consistency, sacrificing availability as necessary in the event of failure. One primary node receives all write operations in MongoDB. The system needs to elect a new primary node if the existing one becomes unavailable, and while it does, clients can't make any write requests so data remains consistent.

In contrast to MongoDB, Apache Cassandra is an open source NoSQL database with a peer-to-peer architecture and potentially multiple points of failure. CAP theorem in Cassandra reveals an AP database: Cassandra offers availability and partition tolerance but can't provide consistency all the time. However, by reconciling inconsistencies as quickly as possible and allowing clients to write to any nodes at any time, Cassandra provides eventual consistency. Inconsistencies are resolved quickly in most cases of network partitions, so the constant availability and high performance are often worth the Cassandra CAP theorem trade-off.

CAP Theorem in NoSQL Databases vs Distributed SQL Databases

NoSQL databases are generally considered to be AP systems, providing Availability and Partition tolerance at the expense of Consistency. In contrast, distributed SQL (NewSQL) databases provide Consistency, Availability and Partition tolerance. (According to Eric Brewer's painstaking analysis, Google Spanner is technically a CP system that can claim to be an "effectively CA" system. Such nuances, while important, are beyond the scope of this glossary entry.)

The fusion of strong consistency with distributed architecture is undeniably attractive. The question is whether distributed SQL can deliver on this promise without compromising in other critical areas – primarily performance. Notably, the traditional CAP theorem makes no provision for performance or latency. For example, according to the CAP theorem, a database can be considered Available if a query returns a response after 30 days. Obviously, such latency would be unacceptable for any real-world application. Learn more from our educational page on SQL vs. NoSQL.

PACELC vs CAP Theorem

A newer way to model databases, the PACELC theorem, builds on the advantages of CAP theorem analysis and extends it, accounting for the nuances of modern systems. According to PACELC, Like the CAP theorem, the PACELC theorem states that in case of network partitioning (P) in a distributed computer system, one has to choose between availability (A) and consistency (C). PACELC extends the CAP theorem by introducing latency and consistency as additional attributes of distributed systems. The theorem states that, "else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C)."

In other words, PACELC reveals that systems tend either towards strong consistency or latency sensitivity. Even in the absence of partitioning, a trade-off between consistency and latency exists.

Traditional CAP theorem provides for neither latency nor performance. For example, the CAP theorem considers a database available if a query returns a response after 30 days—a level of latency that users of any real-world application would find unacceptable.

Based on PACELC, a distributed SQL database that focuses on being a strongly consistent system such as CockroachDB is categorized as PC/EC. They will not sacrifice consistency for any reason, and performance may suffer as a result.

In contrast, a latency sensitive, highly available NoSQL database system like ScyllaDB is PA/EL. These systems may sacrifice consistency for availability should a partition occur.

Cloud-native applications with more modern, distributed architectures often demand predictable low latency and high availability. Strong consistency requirements are less common.

Does ScyllaDB Offer Solutions for CAP Theorem?

ScyllaDB is a PA/EL highly available, partition tolerant, low latency database system. ScyllaDB was designed to provide consistent low-latencies, not just be highly available, and it also provides tunable consistency. Under any conditions short of a complete system failure, ScyllaDB will remain highly available with predictable low latencies for mission critical applications.

ScyllaDB outperforms a distributed NewSQL database such as CockroachDB by a wide margin. Built for high availability, ScyllaDB still delivers tunable consistency, zero downtime, and high performance. Learn more.

What is load balancing?

Load balancing is the method of distributing network traffic equally across a pool of resources that support an application. Modern applications must process millions of users simultaneously and return the correct text, videos, images, and other data to each user in a fast and reliable manner. To handle such high volumes of traffic, most applications have many resource servers with duplicate data between them. A load balancer is a device that sits between the user and the server group and acts as an invisible facilitator, ensuring that all resource servers are used equally.

What are the benefits of load balancing?

Load balancing directs and controls internet traffic between the application servers and their visitors or clients. As a result, it improves an application's availability, scalability, security, and performance.

Application availability

Server failure or maintenance can increase application downtime, making your application unavailable to visitors. Load balancers increase the fault tolerance of your systems by automatically detecting server problems and redirecting client traffic to available servers. You can use load balancing to make these tasks easier:

- Run application server maintenance or upgrades without application downtime
- Provide automatic disaster recovery to backup sites
- Perform health checks and prevent issues that can cause downtime

Application scalability

You can use load balancers to direct network traffic intelligently among multiple servers. Your applications can handle thousands of client requests because load balancing does the following:

- Prevents traffic bottlenecks at any one server
- Predicts application traffic so that you can add or remove different servers, if needed
- Adds redundancy to your system so that you can scale with confidence

Application security

Load balancers come with built-in security features to add another layer of security to your internet applications. They are a useful tool to deal with distributed denial of service attacks, in which attackers flood an application server with millions of concurrent requests that cause server failure. Load balancers can also do the following:

- Monitor traffic and block malicious content
- Automatically redirect attack traffic to multiple backend servers to minimize impact
- Route traffic through a group of network firewalls for additional security

Application performance

Load balancers improve application performance by increasing response time and reducing network latency. They perform several critical tasks such as the following:

- Distribute the load evenly between servers to improve application performance
- Redirect client requests to a geographically closer server to reduce latency
- Ensure the reliability and performance of physical and virtual computing resources

What are load balancing algorithms?

A load balancing algorithm is the set of rules that a load balancer follows to determine the best server for each of the different client requests. Load balancing algorithms fall into two main categories.

Static load balancing

Static load balancing algorithms follow fixed rules and are independent of the current server state. The following are examples of static load balancing.

Roundrobin method

Servers have IP addresses that tell the client where to send requests. The IP address is a long number that is difficult to remember. To make it easy, a Domain Name System maps website names to servers. When you enter aws.amazon.com into your browser, the request first goes to our name server, which returns our IP address to your browser.

In the round-robin method, an authoritative name server does the load balancing instead of specialized hardware or software. The name server returns the IP addresses of different servers in the server farm turn by turn or in a round-robin fashion.

Weighted roundrobin method

In weighted round-robin load balancing, you can assign different weights to each server based on their priority or capacity. Servers with higher weights will receive more incoming application traffic from the name server.

IP hash method

In the IP hash method, the load balancer performs a mathematical computation, called hashing, on the client IP address. It converts the client IP address to a number, which is then mapped to individual servers.

Dynamic load balancing

Dynamic load balancing algorithms examine the current state of the servers before distributing traffic. The following are some examples of dynamic load balancing algorithms.

Least connection method

A connection is an open communication channel between a client and a server. When the client sends the first request to the server, they authenticate and establish an active connection between each other. In the least connection method, the load balancer checks which servers have the fewest active connections and sends traffic to those servers. This method assumes that all connections require equal processing power for all servers.

Weighted least connection method

Weighted least connection algorithms assume that some servers can handle more active connections than others. Therefore, you can assign different weights or capacities to each server, and the load balancer sends the new client requests to the server with the least connections by capacity.

Least response time method

The response time is the total time that the server takes to process the incoming requests and send a response. The least response time method combines the server response time and the active connections to determine the best server. Load balancers use this algorithm to ensure faster service for all users.

Resource-based method

In the resource-based method, load balancers distribute traffic by analyzing the current server load. Specialized software called an agent runs on each server and calculates usage of server resources, such as its computing capacity and memory. Then, the load balancer checks the agent for sufficient free resources before distributing traffic to that server.

How does load balancing work?

Companies usually have their application running on multiple servers. Such a server arrangement is called a server farm. User requests to the application first go to the load balancer. The load balancer then routes each request to a single server in the server farm best suited to handle the request.

Load balancing is like the work done by a manager in a restaurant. Consider a restaurant with five waiters. If customers were allowed to choose their waiters, one or two waiters could be overloaded with work while the others are idle. To avoid this scenario, the restaurant manager assigns customers to the specific waiters who are best suited to serve them.

What are the types of load balancing?

We can classify load balancing into three main categories depending on what the load balancer checks in the client request to redirect the traffic.

Application load balancing

Complex modern applications have several server farms with multiple servers dedicated to a single application function. Application load balancers look at the request content, such as HTTP headers or SSL session IDs, to redirect traffic.

For example, an ecommerce application has a product directory, shopping cart, and checkout functions. The application load balancer sends requests for browsing products to servers that contain images and videos but do not need to maintain open connections. By comparison, it sends shopping cart requests to servers that can maintain many client connections and save cart data for a long time.

Network load balancing

Network load balancers examine IP addresses and other network information to redirect traffic optimally. They track the source of the application traffic and can assign a static IP address to several servers. Network load balancers use the static and dynamic load balancing algorithms described earlier to balance server load.

Global server load balancing

Global server load balancing occurs across several geographically distributed servers. For example, companies can have servers in multiple data centers, in different countries, and in third-party cloud providers around the globe. In this case, local load balancers manage the application load within a region or zone. They attempt to redirect traffic to a server destination that is geographically closer to the client. They might redirect traffic to servers outside the client's geographic zone only in case of server failure.

DNS load balancing

In DNS load balancing, you configure your domain to route network requests across a pool of resources on your domain. A domain can correspond to a website, a mail system, a print server, or another service that is made accessible through the internet. DNS load balancing is helpful for maintaining application availability and balancing network traffic across a globally distributed pool of resources.

What are the types of load balancing technology?

Load balancers are one of two types: hardware load balancer and software load balancer.

Hardware load balancers

A hardware-based load balancer is a hardware appliance that can securely process and redirect gigabytes of traffic to hundreds of different servers. You can store it in your data centers and use virtualization to create multiple digital or virtual load balancers that you can centrally manage.

Software load balancers

Software-based load balancers are applications that perform all load balancing functions. You can install them on any server or access them as a fully managed third-party service.

Comparison of hardware balancers to software load balancers

Hardware load balancers require an initial investment, configuration, and ongoing maintenance. You might also not use them to full capacity, especially if you purchase one only to handle peak-time traffic spikes. If traffic volume increases suddenly beyond its current capacity, this will affect users until you can purchase and set up another load balancer.

In contrast, software-based load balancers are much more flexible. They can scale up or down easily and are more compatible with modern cloud computing environments. They also cost less to set up, manage, and use over time.

How can AWS help with load balancing?

Elastic Load Balancing (ELB) is a fully managed load balancing service that automatically distributes incoming application traffic to multiple targets and virtual appliances across AWS and on-premises resources. You can use it to scale modern applications without complex configurations or API gateways. You can use ELB to set up four different types of software load balancers.

- An Application Load Balancer routes traffic for HTTP-based requests.
- A Network Load Balancer routes traffic based on IP addresses. It is ideal for balancing TCP and User Datagram Protocol (UDP)-based requests.

- A Gateway Load Balancer routes traffic to third-party virtual appliances. It is ideal for incorporating a third-party appliance, such as a network firewall, into your network traffic in a scalable and easy-to-manage way.
- A Classic Load Balancer routes traffic to applications in the Amazon EC2Classic network—a single, flat network that you share with other customers.

You can select the load balancer based on your requirements. For example, Terminix, a global pest control brand, uses Gateway Load Balancer to handle 300% more throughput. Second Spectrum, a company that provides artificial intelligence-driven tracking technology for sports broadcasts, uses AWS Load Balancer Controller to reduce hosting costs by 90%. Code.org, a nonprofit dedicated to expanding access to computer science in schools, uses Application Load Balancer to handle a 400% spike in traffic efficiently during online coding events.

Get started with load balancing by creating an AWS account today!

Horizontal vs Vertical scaling: An in-depth Guide

- Modified By - nOps
- December 6, 2022

- **Feeling overwhelmed by the limitations of your existing cloud cost tool?**

Step into the future with nOps, the next-gen workload provisioner that picks the right compute at the right price in real-time.

Learn more

The primary difference between horizontal scaling and vertical scaling is that horizontal scaling involves adding more machines or nodes to a system, while vertical scaling involves adding more power (CPU, RAM, storage, etc.) to an existing machine. In other words, horizontal scaling involves scaling out, while vertical scaling involves scaling up. Horizontal scaling is typically used to handle increasing amounts of traffic or workload, while vertical scaling is typically used to handle resource-intensive tasks or applications that require more processing power.

As we expand our operations or our clientele base develops, hundreds of things change. From employee strength to resource count, we scale upwards. Similarly, when the business is growing, our cloud infrastructure

also alters. As a result, we need enhanced cloud computing power to serve the new customers and maintain the new databases. But, given scalability options, i.e., horizontal and vertical scaling, it may take time to choose what is right for your business and what advantages/disadvantages it will offer. So, let's explore the horizontal vs vertical scaling AWS offerings and explore the same.

What is horizontal scaling?

Horizontal scaling, often known as "scaling out," is the process of increasing the number of nodes and machines in the resource pool. For a sequential piece of logic to be processed in parallel across numerous devices, horizontal scaling calls for breaking it into smaller chunks and delegating the logic to the new machine. In simpler terms, scaling horizontally is about hiring new employees for an additional client/problem set.

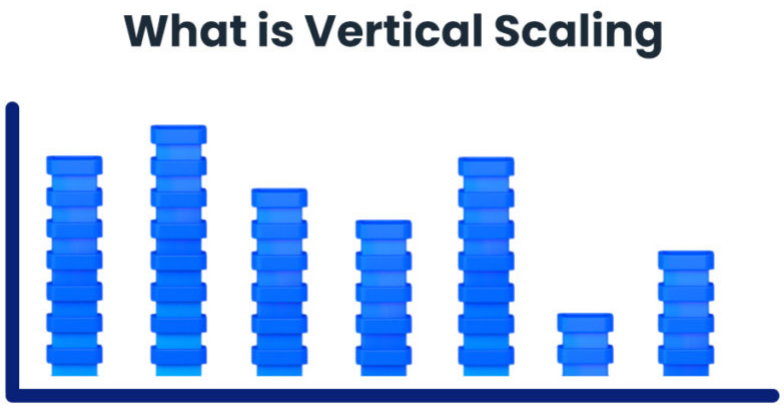
Pros:

- Easier Scaling from Hardware additions
- Enhanced Flexibility
- Lesser Downtime
- Offers Redundancy

Cons:

- Higher initial costs involved
- Harder to maintain

What is vertical scaling?



Vertical scaling, often known as “scaling up,” is the process of increasing the power of an existing system, such as the CPU or RAM, to meet the rising demands. Because there is no need to alter the logic, vertical scaling is simpler. Instead, you are only executing the same code on machines with more capacity.

The memory, storage, or network speed can be vertically scaled. Vertical scaling can also refer to completely replacing a server or shifting the workload from an outdated server to an updated one. In simpler terms, Vertical scaling is about upskilling existing employees for an additional client/problem set.

Pros:

- Cost Effective
- Less Complexity involved
- Easier to maintain

Cons:

- More Downtime possibilities
- Very less flexibility
- Single Point of Failure

Horizontal vs vertical scaling:

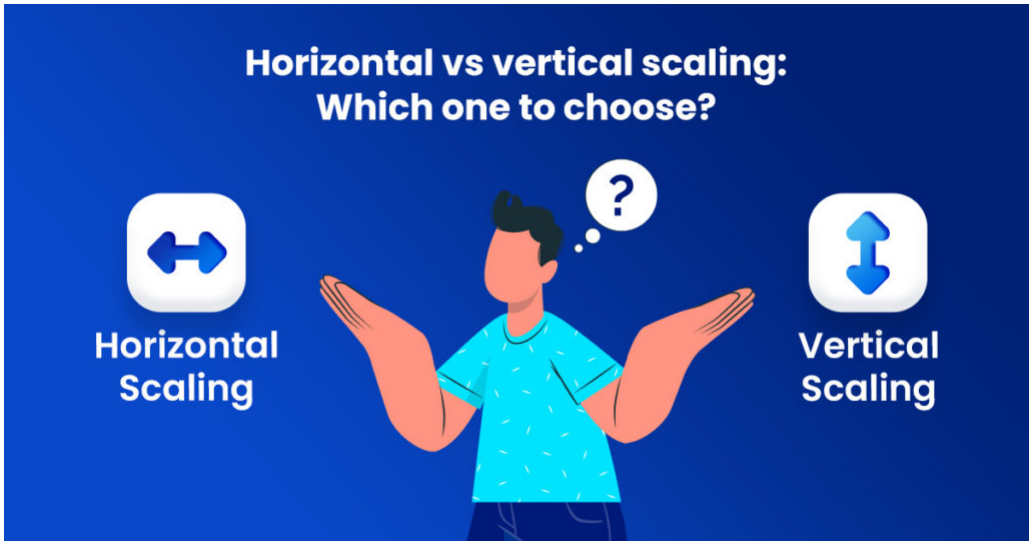
The following differences will help you delve into the deep understanding of horizontal vs vertical scaling:

Fundamentals	Horizontal Scaling	Vertical Scaling
Data Management	Scaling horizontally typically relies on data partitioning as each node only contains part of the data.	In vertical scaling, the data resides on a single node, and scaling is accomplished by multi-core, primarily by distributing the load among the machine’s CPU and RAM resources.
Examples	Cassandra, MongoDB, Google Cloud Spanner	MySQL and Amazon RDS
Downtime possibility	You can scale with less downtime by adding additional computers to the existing pool because you are no longer constrained by the capacity of a single device.	There is an upper physical limit to vertical scaling, which is the scale of the current hardware.Vertical scaling is restricted to the capacity of one machine

		because expanding over that limit can result in downtime.
Concurrency Models	As it entails distributing jobs among devices over a network is known as distributed programming. Several patterns are connected to this model: MapReduce, Master/Worker*, Blackboard, and many spaces.	It involves the Actor model: Multi-threading and in-process message forwarding are frequently used to implement concurrent programming on multi-core platforms.
Message Passing	Data sharing is more difficult in distributed computing because there isn't a shared address space. Since you will send copies of the data, it also increases the cost of sharing, transferring, or updating data.	Data sharing and message passing Can be accomplished by passing a reference in a multi-threaded scenario since it is reasonable to presume that there is a shared address space.

Horizontal vs vertical scaling: Which one to choose?

Numerous factors help you determine which type of scaling will suit your business objectives the best! So, here are the determinants helping you choose the most appropriate out of Horizontal scaling vs. Vertical Scaling-



- **Performance:** By scaling out, you can pool the computing capacity of several physical machines by combining and leveraging the computing power of all nodes. You are thus not constrained by the capability of a single unit. But if you have enough resources on a single system to meet your **cloud scalability** requirements, vertical scaling can be an appropriate fit.
- **Cost:** Horizontal upgrades have higher up-front hardware expenditures. Vertical scaling might be your best option if you have a limited budget and need to swiftly & inexpensively add more capacity to your infrastructure.
- **Future-proofing:** By horizontally scaling, you can improve the performance ceiling for your business by adding more modern equipment. A single node can only be vertically scaled so far, and it might not be able to accommodate future needs.
- **Flexibility:** Scaling out may be a preferable alternative if you want the flexibility to select the ideal configuration setting whenever you want to optimize performance and efficiency. Because vertical scaling will be constrained to the configurations of that single machine only. It will be more challenging to add or

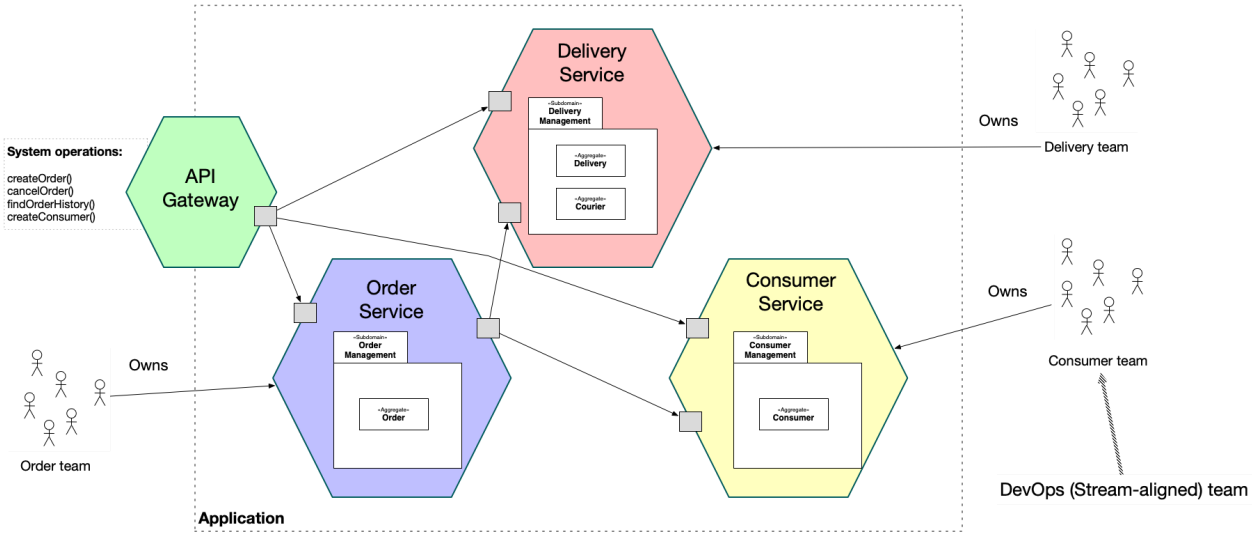
update individual lines of code without affecting the overall application if it is built as one big unit. It's simpler to decouple and horizontally grow your application to give a more continuous upgrade process.

- **Topographic distribution:** It is unrealistic to expect all of your clients to access your services from the same server in a single place if you want to have a national or international clientele. To sustain your service level agreement in this scenario, you must horizontally scale your resources.
- **Reliability and Redundancy:** Your system might be more reliable if horizontally scaled. It ensures that you do not depend on a single machine and increases redundancy. When one machine malfunctions, another one might be able to take over temporarily.

Thus, it can be tough to choose the best between both. But, once you get your objectives clear and are aware of the Horizontal vs vertical scaling factors, you surely can make a wiser decision.

If you are looking for Kubernetes Autoscaling, you can read the nOps blog at: **A Comprehensive Guide to Kubernetes Autoscaling!**

What are microservices?



Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are:

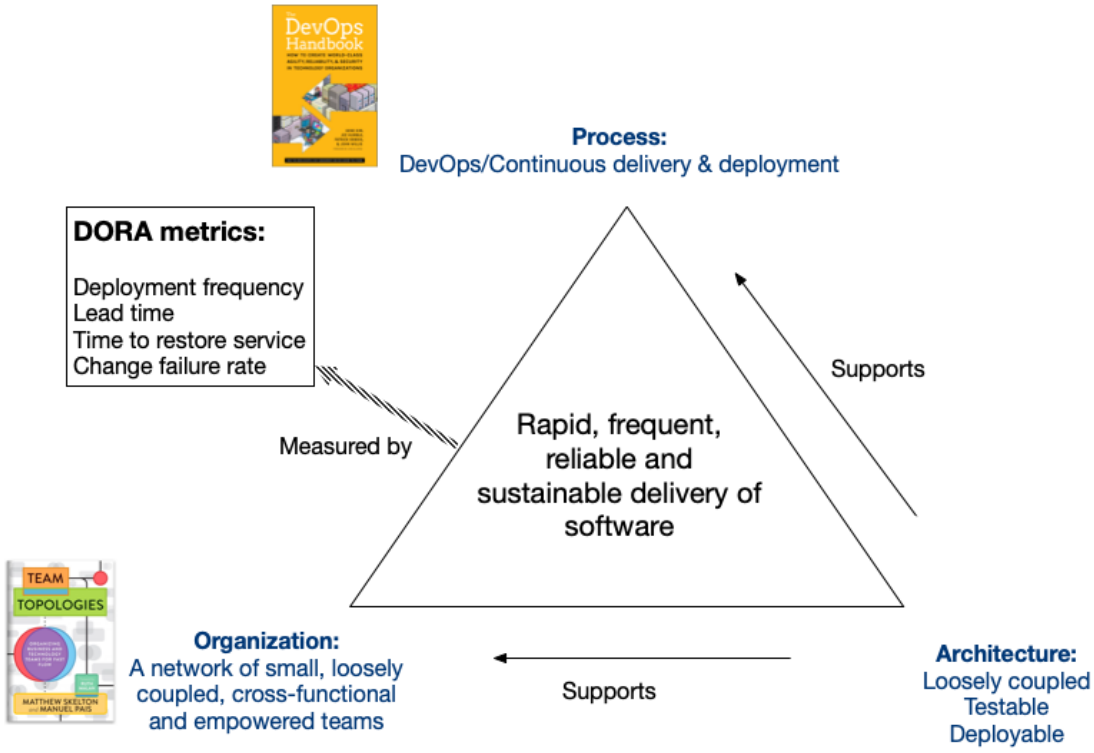
- Independently deployable
- Loosely coupled

Services are typically organized around business capabilities. Each service is often owned by a single, small team.

The microservice architecture enables an organization to deliver large, complex applications rapidly, frequently, reliably and sustainably - a necessity for competing and winning in today's world.

Enabling rapid, frequent and reliable software delivery

Success triangle



In order to thrive in today's volatile, uncertain, complex and ambiguous world, businesses must be nimble, agile and innovate faster. Moreover, since modern businesses are powered by software, IT must deliver that software rapidly, frequently and reliably - as measured by the DORA metrics.

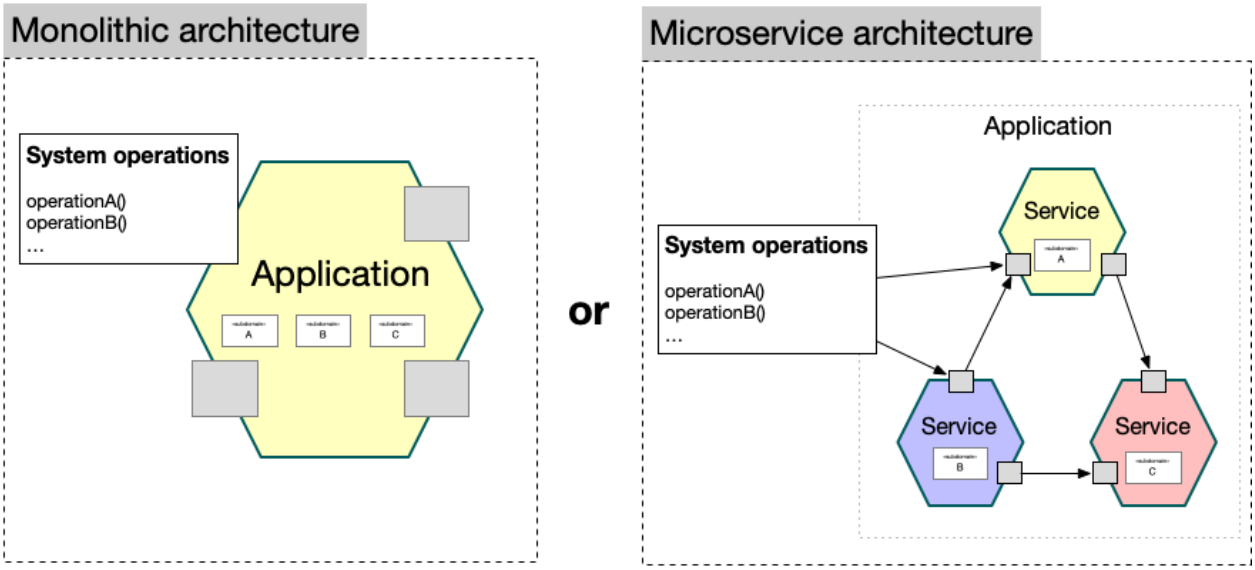
Rapid, frequent, reliable and sustainable delivery requires the success triangle, a combination of three things:

- Process - DevOps as defined by the DevOps handbook
- Organization - a network of small, loosely coupled, cross-functional teams
- Architecture - a loosely coupled, testable and deployable architecture

Teams work independently most of the time to produce a stream of small, frequent changes that are tested by an automated deployment pipeline and deployed into production.

Let's now look at when you typically need to use microservices in order to have a loosely coupled, testable and deployable architecture.

When you outgrow your monolithic architecture



Let's imagine that you responsible for a business critical business application that has a monolithic architecture and you are struggling to meet the needs of the business. Should you consider migrating to a microservice architecture? The short answer is that it depends.

It's important to make the most of your monolithic architecture, e.g. adopt DevOps, and reorganize into loosely coupled, small teams.

In many cases, once you have embraced the success triangle, your monolithic architecture is sufficiently loosely coupled, testable and deployable to enable rapid software delivery.

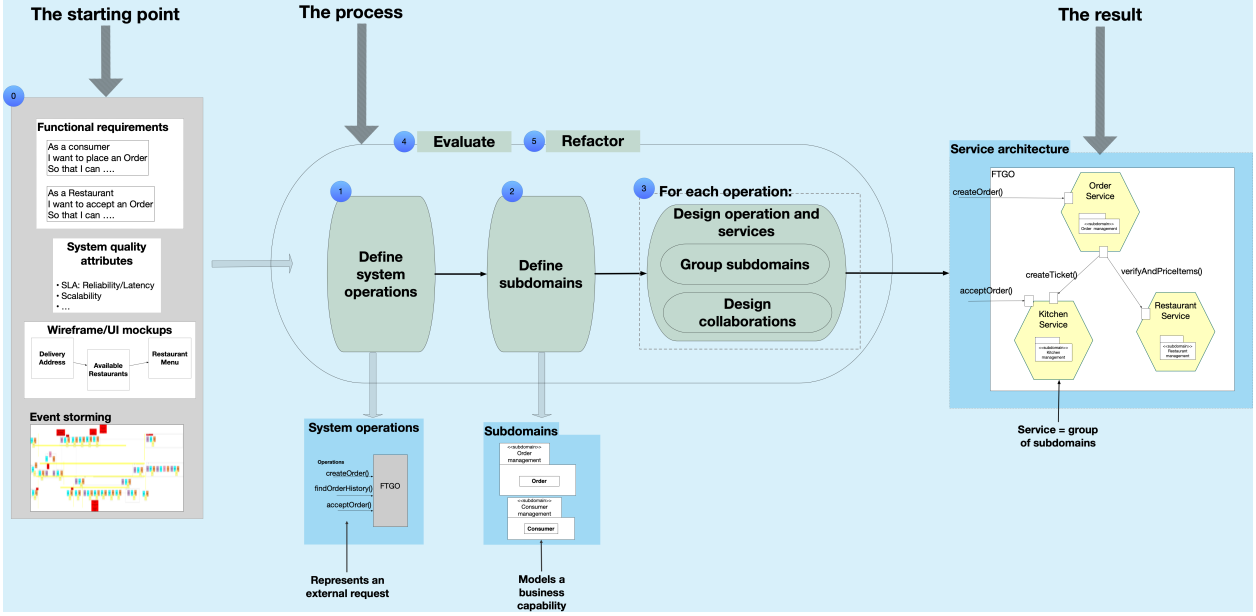
But sometimes an application can outgrow its monolithic architecture and become an obstacle to rapid, frequent and reliable software delivery. This typically happens when the application becomes large and complex and is developed by many teams. For example, its deployment pipeline become a bottleneck. When this occurs, you should consider migrating to microservices.

My presentation *Considering Migrating a Monolith to Microservices? A Dark Energy, Dark Matter Perspective* describes how to decide whether to migrate to microservices.

If you have decided to migrate to microservices then the next step is to design a target architecture.

Designing a microservice architecture

Assemblage



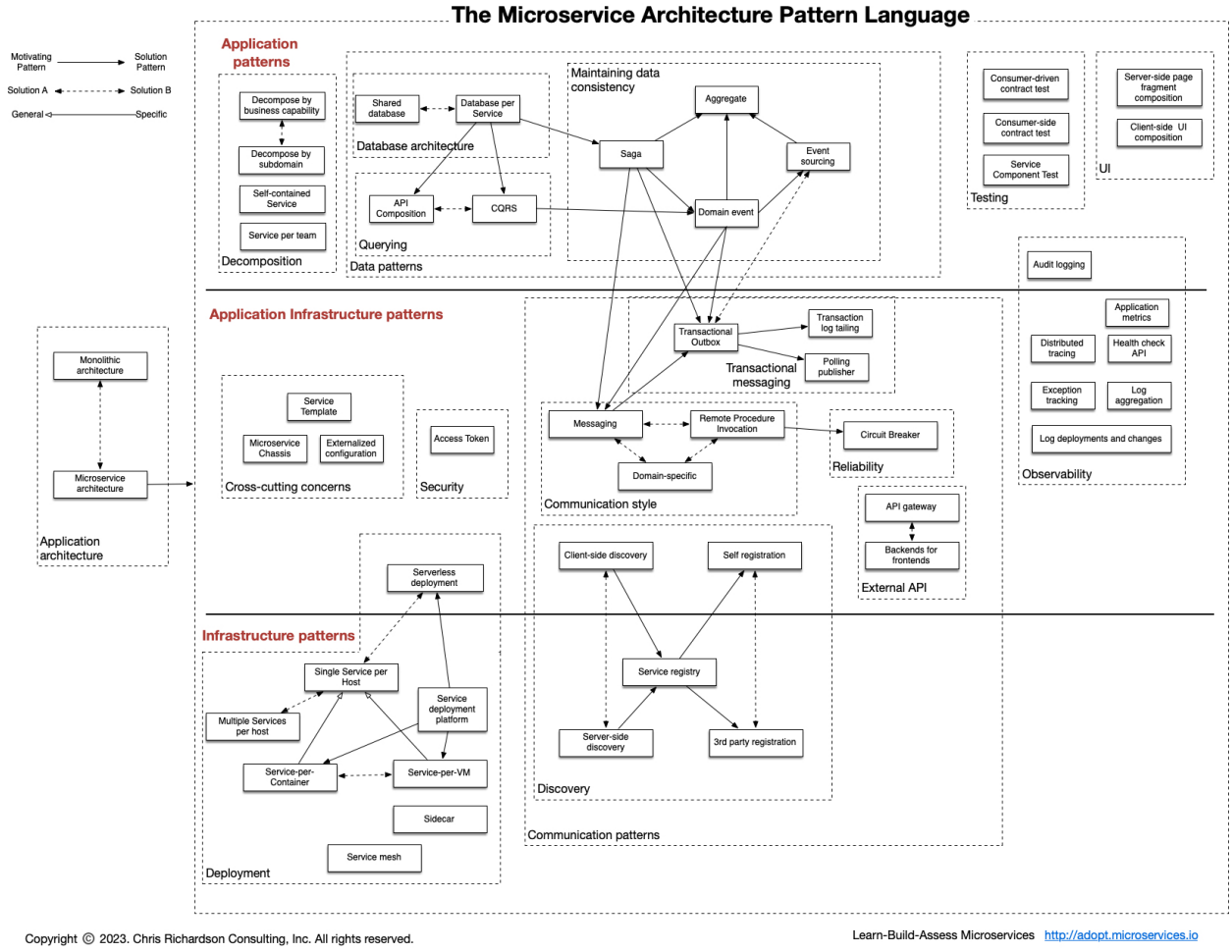
Picking technologies - Kubernetes, message broker etc - is important

But what's critically important is designing a good service architecture: identifying services; defining their responsibilities; their APIs and collaborations. If you get it wrong you risk creating a distributed monolith, which will slow down software delivery.

What's more, designing the service architecture is challenging because it's a creative activity - not something you can buy, download or read in a manual.

Assemblage is an architecture definition process that you can can use to define your microservice architecture. It distills your requirements into system operations and subdomains; uses the dark energy and dark matter forces to group the subdomains into services; and designs the distributed system operations.

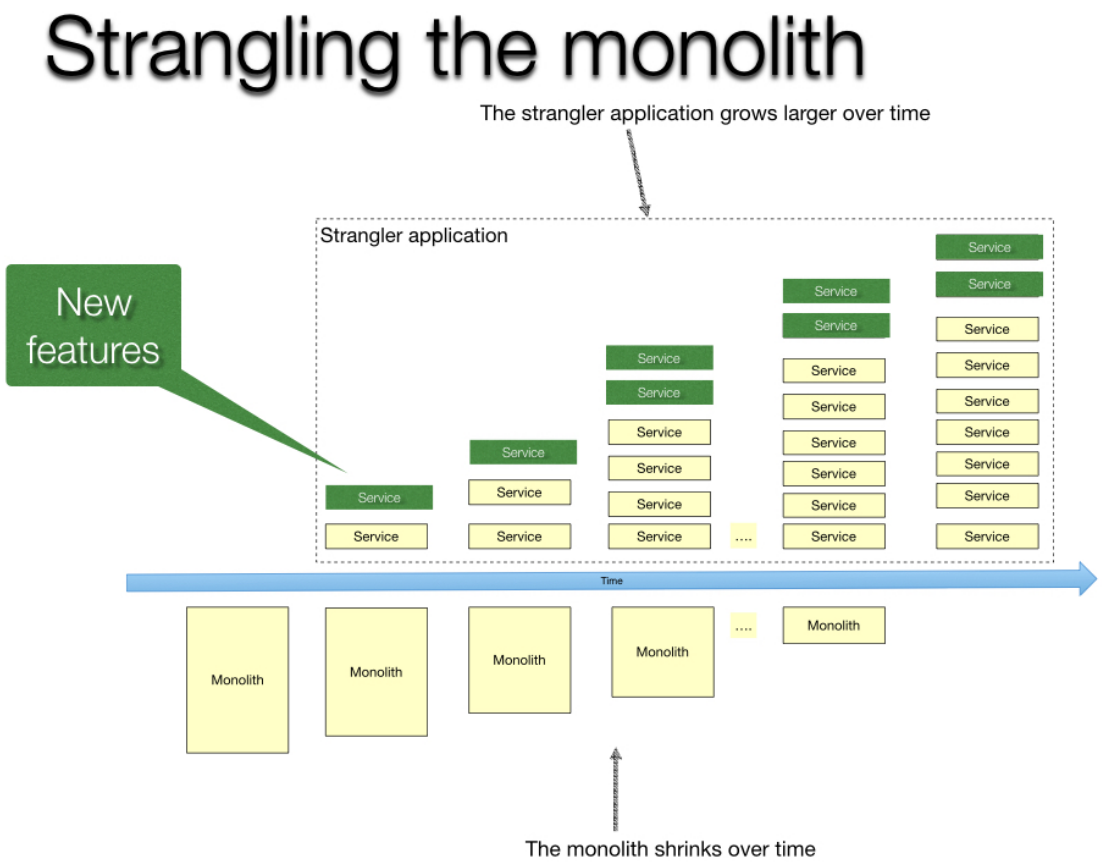
The Microservices pattern language



Assemblage works in conjunction with the Microservice architecture pattern language, which is your guide when designing a technical architecture

After defining a target microservice architecture you then need to refactor your existing monolith.

Migrating from a monolith to microservices



There are numerous principles for migrating a monolithic application to microservices.

One key principle is to incrementally migrate to microservices using the Strangler Fig pattern - no big bang rewrite. By using this pattern, you rapidly validate your design decisions and deliver new, useful functionality much earlier.

It's also important to avoid the Microservices adoption antipatterns.

Database Sharding: Concepts and Examples Get started freeLearn more about Atlas

Your application is growing. It has more active users, more features, and generates more data every day. Your database is now becoming a bottleneck for the rest of your application. Database sharding could be the solution to your problems, but many do not have a clear understanding of what it is and, especially, when to use it. In this article, we'll cover the basics of database sharding, its best use cases, and the different ways you can implement it.

Jump to:

- What is database sharding?
- Evaluating alternatives
- Advantages and disadvantages of sharding
- How does database sharding work?
- Sharding architecture and types

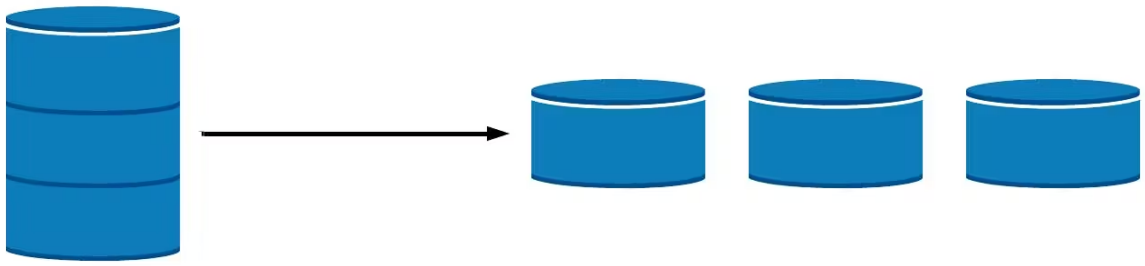
- FAQs

What is database sharding?

Sharding is a method for distributing a single dataset across multiple databases, which can then be stored on multiple machines. This allows for larger datasets to be split into smaller chunks and stored in multiple data nodes, increasing the total storage capacity of the system. See more on the basics of sharding here.

Similarly, by distributing the data across multiple machines, a sharded database can handle more requests than a single machine can.

Sharding is a form of scaling known as **horizontal scaling** or **scale-out**, as additional nodes are brought on to share the load. Horizontal scaling allows for near-limitless scalability to handle big data and intense workloads. In contrast, **vertical scaling** refers to increasing the power of a single machine or single server through a more powerful CPU, increased RAM, or increased storage capacity.

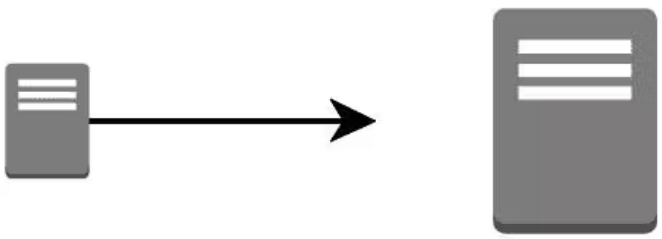


Do you need database sharding?

Database sharding, as with any distributed architecture, does not come for free. There is overhead and complexity in setting up shards, maintaining the data on each shard, and properly routing requests across those shards. Before you begin sharding, consider if one of the following alternative solutions will work for you.

Vertical scaling

By simply upgrading your machine, you can scale vertically without the complexity of sharding. Adding RAM, upgrading your computer (CPU), or increasing the storage available to your database are simple solutions that do not require you to change the design of either your database architecture or your application.



Specialized services or databases

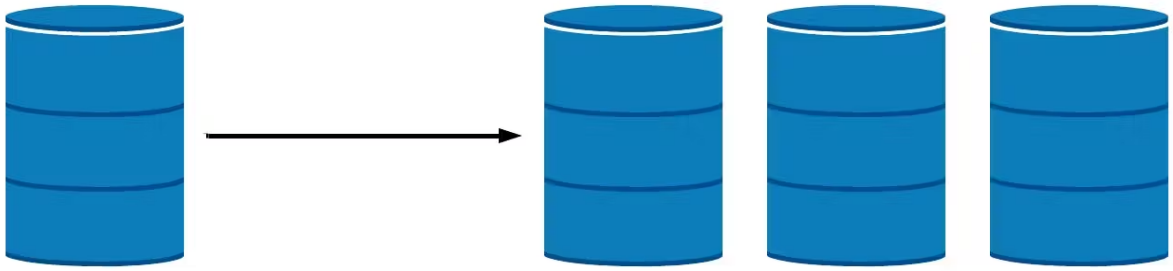
Depending on your use case, it may make more sense to simply shift a subset of the burden onto other providers or even a separate database. For example, blob or file storage can be moved directly to a cloud provider such as Amazon S3. Analytics or full-text search can be handled by specialized services or a data warehouse. Offloading this particular functionality can make more sense than trying to shard your entire database.

Replication

If your data workload is primarily read-focused,

replication

increases availability and read performance while avoiding some of the complexity of database sharding. By simply spinning up additional copies of the database, read performance can be increased either through load balancing or through geo-located query routing. However, replication introduces complexity on write-focused workloads, as each write must be copied to every replicated node.



On the other hand, if your core application database contains large amounts of data, requires high read and high write volume, and/or you have specific availability requirements, a sharded database may be the way forward. Let's look at the advantages and disadvantages of sharding.

Advantages of sharding

Sharding allows you to scale your database to handle increased load to a nearly unlimited degree by providing **increased read/write throughput**, **storage capacity**, and **high availability**. Let's look at each of those in a little more detail.

- **Increased read/write throughput** — By distributing the dataset across multiple shards, both read and write operation capacity is increased as long as read and write operations are confined to a single shard.
- **Increased storage capacity** — Similarly, by increasing the number of shards, you can also increase overall total storage capacity, allowing near-infinite scalability.
- **High availability** — Finally, shards provide high availability in two ways. First, since each shard is a replica set, every piece of data is replicated. Second, even if an entire shard becomes unavailable since the data is distributed, the database as a whole still remains partially functional, with part of the schema on different shards.

Disadvantages of sharding

Sharding does come with several drawbacks, namely **overhead in query result compilation, complexity of administration,** and **increased infrastructure costs.**

- **Query overhead** — Each sharded database must have a separate machine or service which understands how to route a querying operation to the appropriate shard. This introduces additional latency on every operation. Furthermore, if the data required for the query is horizontally partitioned across multiple shards, the router must then query each shard and merge the result together. This can make an otherwise simple operation quite expensive and slow down response times.
- **Complexity of administration** — With a single unsharded database, only the database server itself requires upkeep and maintenance. With every sharded database, on top of managing the shards themselves, there are additional service nodes to maintain. Plus, in cases where **replication** is being used, any data updates must be mirrored across each replicated node. Overall, a sharded database is a more complex system which requires more administration.
- **Increased infrastructure costs** — Sharding by its nature requires additional machines and compute power over a single database server. While this allows your database to grow beyond the limits of a single machine, each additional shard comes with higher costs. The cost of a distributed database system, especially if it is missing the proper optimization, can be significant.

Having considered the pros and cons, let's move forward and discuss implementation.

How does sharding work?

In order to shard a database, we must answer several fundamental questions. The answers will determine your implementation.

First, how will the data be distributed across shards? This is the fundamental question behind any sharded database. The answer to this question will have effects on both performance and maintenance. More detail on this can be found in the "Sharding Architectures and Types" section.

Second, what types of queries will be routed across shards? If the workload is primarily read operations, replicating data will be highly effective at increasing performance, and you may not need sharding at all. In contrast, a mixed read-write workload or even a primarily write-based workload will require a different architecture.

Finally, how will these shards be maintained? Once you have sharded a database, over time, data will need to be redistributed among the various shards, and new shards may need to be created. Depending on the distribution of data, this can be an expensive process and should be considered ahead of time.

With these questions in mind, let's consider some sharding architectures.

Sharding architectures and types

While there are many different sharding methods, we will consider four main kinds: ranged/dynamic sharding, algorithmic/hashed sharding, entity/relationship-based sharding, and geography-based sharding.

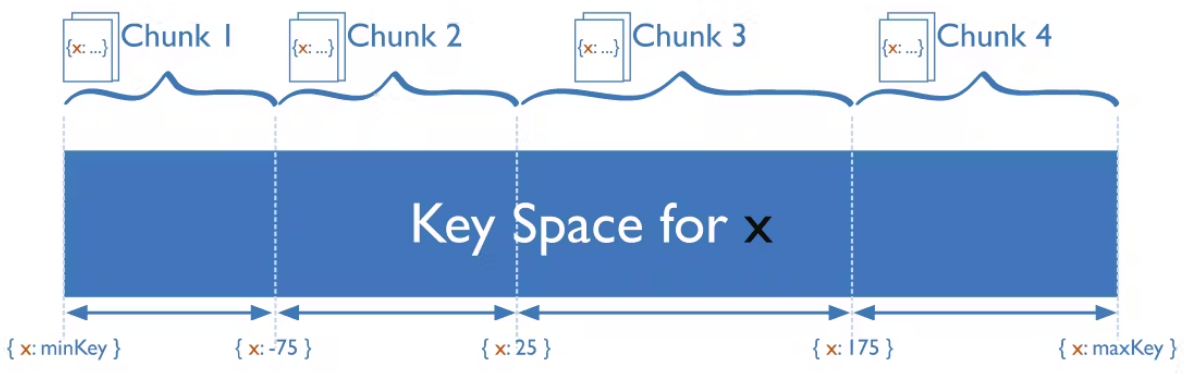
Ranged/dynamic sharding

Ranged sharding, or **dynamic sharding**, takes a field on the record as an input and, based on a predefined range, allocates that record to the appropriate shard. Ranged sharding requires there to be a lookup table or service available for all queries or writes. For example, consider a set of data with IDs that range from 0-50. A simple lookup table might look like the following:

Range	Shard ID
[0, 20)	A

[20, 40)	B
[40, 50]	C

The field on which the range is based is also known as the shard key. Naturally, the choice of shard key, as well as the ranges, are critical in making range-based sharding effective. A poor choice of shard key will lead to unbalanced shards, which leads to decreased performance. An effective shard key will allow for queries to be targeted to a minimum number of shards. In our example above, if we query for all records with IDs 10-30, then only shards A and B will need to be queried.



Two key attributes of an effective shard key are high

cardinality

and well-distributed

frequency

.

Cardinality

refers to the number of possible values of that key. If a shard key only has three possible values, then there can only be a maximum of three shards.

Frequency

refers to the distribution of the data along the possible values. If 95% of records occur with a single shard key value then, due to this hotspot, 95% of the records will be allocated to a single shard. Consider both of these attributes when selecting a shard key.

Range-based sharding is an easy-to-understand method of horizontal partitioning, but the effectiveness of it will depend heavily on the availability of a suitable shard key and the selection of appropriate ranges. Additionally, the lookup service can become a bottleneck, although the amount of data is small enough that this typically is not an issue.

Algorithmic/hashed sharding

Algorithmic sharding or hashed sharding, takes a record as an input and applies a hash function or algorithm to it which generates an output or hash value. This output is then used to allocate each record to the appropriate shard.

The function can take any subset of values on the record as inputs. Perhaps the simplest example of a hash function is to use the modulus operator with the number of shards, as follows:

Hash Value=ID % Number of Shards

This is similar to range-based sharding — a set of fields determines the allocation of the record to a given shard. Hashing the inputs allows more even distribution across shards even when there is not a suitable shard key, and no lookup table needs to be maintained. However, there are a few drawbacks.

First, query operations for multiple records are more likely to get distributed across multiple shards. Whereas ranged sharding reflects the natural structure of the data across shards, hashed sharding typically disregards the meaning of the data. This is reflected in increased broadcast operation occurrence.

Second, resharding can be expensive. Any update to the number of shards likely requires rebalancing all shards to moving around records. It will be difficult to do this while avoiding a system outage.

Entity-/relationship-based sharding

Entity-based sharding keeps related data together on a single physical shard. In a relational database (such as PostgreSQL, MySQL, or SQL Server), related data is often spread across several different tables.

For instance, consider the case of a shopping database with users and payment methods. Each user has a set of payment methods that is tied tightly with that user. As such, keeping related data together on the same shard can reduce the need for broadcast operations, increasing performance.

Geography-based sharding

Geography-based sharding, or **geosharding**, also keeps related data together on a single shard, but in this case, the data is related by geography. This is essentially ranged sharding where the shard key contains geographic information and the shards themselves are geo-located.

For example, consider a dataset where each record contains a “country” field. In this case, we can both increase overall performance and decrease system latency by creating a shard for each country or region, and storing the appropriate data on that shard. This is a simple example, and there are many other ways to allocate your geoshards which are beyond the scope of this article.

Summary

We’ve defined what sharding is, discussed when to use it, and explored different sharding architectures. Sharding is a great solution for applications with large data requirements and high-volume read/write workloads, but it does come with additional complexity. Consider whether the benefits outweigh the costs or whether there is a simpler solution before you begin implementation.

What is data replication?

Data replication also known as database replication, is a method of copying data to ensure that all information stays identical in real-time between all data resources. Think of database replication as the net that catches your information and keeps it falling through the cracks and getting lost. Data hardly stays stagnant. It’s ever-changing. It’s an ongoing process that ensures data from a primary database is mirrored in a replica, even one located on the other side of the planet.

These days, “instantaneous” isn’t quick enough. Cutting latency down to sub-millisecond intervals is the universal objective. We’ve all seen this situation before – pressing the refresh button on a website, waiting for what feels like an eternity (seconds) to see your information updated. Latency decreases productivity for a user. Achieving **near-real-time** is the goal. Zero time lag is the new ideal for any use case.

How does data replication work?

Data replication is copying data from one host to another, like, say, between two on-prem, or one on-prem to a cloud, and so forth. The point is to achieve real-time consistency with data for all users, wherever they're accessing the data from. Data-Driven Business Models (DDBMs), like the gaming industry, rely heavily on the analytics acquired through real-time data. For a clearer understanding of just how necessary real-time accessibility is for a DDBM, watch this video below:

What are the benefits of data replication?

Improved reliability and disaster recovery

In case of an emergency, should your primary instance be compromised, it's vital to have mission-critical applications safeguarded with a replica that can be swapped in its place. Disaster recovery replication methods work similarly to a backup generator; imagine blowing a critical fuse or your power grid goes dark – you won't have to worry because you have a backup generator to swoop in as a substitute and keep your lights running.

Because replica instances are exact copies of primary instances, you can guarantee performance will not falter, regardless of what happens to your primary. Even if the link between a primary and a replica breaks, performance is still assured as the primary will enact a partial resynchronization, gathering the commands that were not delivered to the replica during the disconnection. If not possible, a full resynchronization will be initiated using a snapshot.

Increased app performance

By spreading the data across multiple instances, you're helping to optimize read performance. Performance is also optimized by having your data accessible in multiple locations, thus minimizing any latency issues. Also, when replicas are directed to process most of your reads, that opens up space for your primary to tackle most of the heavy lifting of writes.

More efficient IT teams

Reduction in IT labor to manually replicate data.

Understanding full data replication vs. partial replication

Full Database Replication occurs when an entire primary database is replicated within every replica instance available. This is a holistic approach that mirrors pre-existing, new, and updated data to all destinations. Though this approach is very comprehensive, it also calls for a considerable amount of processing power and encumbers the network load because of the large size of the data being copied.

Unlike full replication, **partial replication** only mirrors some parts of the data, typically recently updated data. Partial replication isolates particular bits of data following the importance of the data at a specific location. For example, a large financial firm with headquarters in London could have many satellite offices operating around the world, with an office in Boston, another in Kuala Lumpur, and so on.

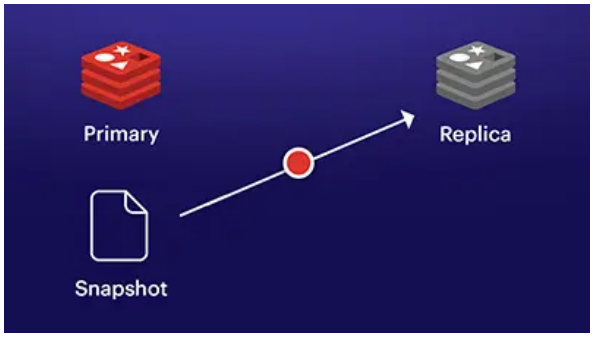
Partial replication allows the analysts in London to have only UK-pertinent data at their site and have only that data be consistently replicated for their needs. The other satellite offices in the United States and Malaysia, respectively, can do the same and not bog down any one system, which improves performance and minimizes network traffic.

Examples of data replication

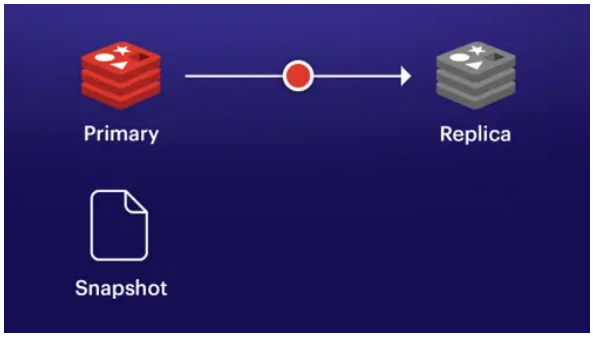
Transactional replication

This form of database replication sees data from a primary database replicating data in real-time to a replica instance by mirroring these changes in the order that they were made in the primary database. This optimizes consistency. The replication takes what is called a “snapshot” of the data in the primary and uses that snapshot as a blueprint of what needs to be replicated elsewhere. With transactional replication, you can track and distribute changes as needed.

A snapshot of the primary is shared to the replica



Primary sends data gathered after the snapshot to the replica



Given the incremental nature of this process, transactional replication isn’t the optimal choice when looking for a backup database option. Transactional replication is a useful choice in situations where you need real-time

consistency across all data locations, where each minuscule change needs to be accounted for, not just the overall impact of the changes, and if data is changing regularly from one specific location.

Snapshot replication

As its name suggests, snapshot replication takes a “snapshot” of the data from the primary as it appears at a specific moment and moves it along to the replica. Like a photograph, snapshot replication captures what data looks like at a point in time, as it looks when it moves from the primary to the replica, but doesn’t account for how it is later updated. Thus, don’t use snapshot replication to make a backup.

In the event of a storage failure, snapshot replication will have no path to updated information. To keep your information consistent, you can start with a snapshot, but then ensure that all changes made to the primary are then passed on to every replica.

On the other hand, this method is rather helpful for recoveries in the event of accidental deletion. Think of it like your Version History on Google Docs. Wish you could work on your presentation the way it looked four hours ago? If Google Docs takes a snapshot of your work at hourly intervals, you could click back on that version, or “snapshot,” from four hours ago and see what your information looked like then.

Merge replication

This method typically begins with a snapshot of the data and distributes that data to its replicas, and maintains synchronization of data between the entire system. What makes merge replication different is that it allows each node to make changes to the data independently but merges all those updates into a unified whole.

Merge replication also accounts for each change made at each node. To go back to our previous Google Docs example, if you’ve ever shared a document with coworkers who then leave comments and edits on your document, you’ll see who made what changes and at what time. Merge replication functions in a very similar way.

Key-Based replication

Also known as key-based incremental data replication, this method leverages a replication key to identify, locate and alter only the specific data that has been changed since the last update. By isolating that information, it facilitates the backup process, working with only as much load as it has to. Though key-based replication makes for a speedy method of refreshing new data, it comes with the disadvantage of failing to replicate deleted data.

Active-Active Geo-Distribution

Active-Active Geo-Distribution, also known as **peer-to-peer replication**, works somewhat like transactional replication, as it relies on constant transactional data via nodes. With active-active, all the nodes in the same network are constantly sending data to one another by synching the database with all the corresponding nodes. All the nodes are also writable, meaning anyone can change the data anywhere around the world, and it will reflect in all the other nodes. This guarantees real-time consistency, no matter where in the globe the change may occur.

Conflict-free Replicated Data Types (CRDTs) define how this data is replicated. In the event of a network failure with one of the replicas or nodes, the other replicas will have all the necessary data ready to replicate once that node comes back online. This is a solid solution for enterprises that need several data centers located across the globe. Take a look at the video below for examples of Active-Active Geo-Distribution use cases.

> Download the Under the Hood: Redis CRDTs white paper now

Synchronous and asynchronous replication

With synchronous replication, the data is written in both the primary and the replica at the same time, hence the name. Asynchronous replication, on the other hand, only copies data to the replica once the writes have already occurred in the primary. Asynchronous replication doesn't tend to happen in real-time, though it's possible. Because of the scheduled write operations that tend to happen in batches with asynchronous, sometimes data gets lost, in most cases when a fail-over event transpires. Still, asynchronous is an apt solution when having to replicate data over long distances, as the real-time component isn't a mission-critical factor.

What are common data replication implementation challenges?

Maintaining data across multiple instances requires a consistent set of resources. The **cost** of having a primary with multiple replica instances can be quite high in many instances. Maintaining these operations and ensuring that no system failures occur requires a **dedicated team** of experts. And depending on the architecture, the network **bandwidth** could get overloaded when new processes are put in place, which could affect latencies, reads, and writes.

1

SQL tuning is the attempt to diagnose and repair SQL statements that fail to meet a performance standard.

1.1 About SQL Tuning

SQL tuning is the iterative process of improving SQL statement performance to meet specific, measurable, and achievable goals.

SQL tuning implies fixing problems in deployed applications. In contrast, application design sets the security and performance goals *before* deploying an application.

See Also:

- SQL Performance Methodology
- "Guidelines for Designing Your Application" to learn how to design for SQL performance

1.2 Purpose of SQL Tuning

A SQL statement becomes a problem when it fails to perform according to a predetermined and measurable standard.

After you have identified the problem, a typical tuning session has one of the following goals:

- Reduce user response time, which means decreasing the time between when a user issues a statement and receives a response
- Improve throughput, which means using the least amount of resources necessary to process all rows accessed by a statement

For a response time problem, consider an online book seller application that hangs for three minutes after a customer updates the shopping cart. Contrast with a three-minute parallel query in a data warehouse that consumes all of the database host CPU, preventing other queries from running. In each case, the user response time is three minutes, but the cause of the problem is different, and so is the tuning goal.

1.3 Prerequisites for SQL Tuning

SQL performance tuning requires a foundation of database knowledge.

If you are tuning SQL performance, then this manual assumes that you have the knowledge and skills shown in the following table.

Table 1-1 Required Knowledge

Required Knowledge	Description	To Learn More
Database architecture	Database architecture is not the domain of administrators alone. As a developer, you want to develop applications in the least amount of time against an Oracle database, which requires exploiting the database architecture and features. For example, not understanding Oracle Database concurrency controls and multiversioning read consistency may make an application corrupt the integrity of the data, run slowly, and decrease scalability.	Oracle Database Concepts explains the basic relational data structures, transaction management, storage structures, and instance architecture of Oracle Database.
SQL and PL/SQL	Because of the existence of GUI-based tools, it is possible to create applications and administer a database without knowing SQL. However, it is impossible to tune applications or a database without knowing SQL.	Oracle Database Concepts includes an introduction to Oracle SQL and PL/SQL. You must also have a working knowledge of Oracle Database SQL Language Reference, Oracle Database PL/SQL Packages and Types Reference, and Oracle Database PL/SQL Packages and Types Reference.
SQL tuning tools	The database generates performance statistics, and provides SQL tuning tools that interpret these statistics.	Oracle Database Get Started with Performance Tuning provides an introduction to the principal SQL tuning tools.

1.4 Tasks and Tools for SQL Tuning

After you have identified the goal for a tuning session, for example, reducing user response time from three minutes to less than a second, the problem becomes how to accomplish this goal.

1.4.1 SQL Tuning Tasks

The specifics of a tuning session depend on many factors, including whether you tune proactively or reactively.

In **proactive SQL tuning**, you regularly use SQL Tuning Advisor to determine whether you can make SQL statements perform better. In **reactive SQL tuning**, you correct a SQL-related problem that a user has experienced.

Whether you tune proactively or reactively, a typical SQL tuning session involves all or most of the following tasks:

1. Identifying high-load SQL statements

   Review past execution history to find the statements responsible for a large share of the application workload and system resources.

2. Gathering performance-related data

   The optimizer statistics are crucial to SQL tuning. If these statistics do not exist or are no longer accurate, then the optimizer cannot generate the best plan. Other data relevant to SQL performance include the structure of tables and views that the statement accessed, and definitions of any indexes available to the statement.

3. Determining the causes of the problem

   Typically, causes of SQL performance problems include:

   - Inefficiently designed SQL statements

     If a SQL statement is written so that it performs unnecessary work, then the optimizer cannot do much to improve its performance. Examples of inefficient design include

     - Neglecting to add a join condition, which leads to a Cartesian join
     - Using hints to specify a large table as the driving table in a join
     - Specifying `UNION` instead of `UNION ALL`
     - Making a subquery execute for every row in an outer query

   - Suboptimal execution plans

     The query optimizer (also called the optimizer) is internal software that determines which execution plan is most efficient. Sometimes the optimizer chooses a plan with a suboptimal access path, which is the means by which the database retrieves data from the database. For example, the plan for a query predicate with low selectivity may use a full table scan on a large table instead of an index.

     You can compare the execution plan of an optimally performing SQL statement to the plan of the statement when it performs suboptimally. This comparison, along with information such as changes in data volumes, can help identify causes of performance degradation.

   - Missing SQL access structures

     Absence of SQL access structures, such as indexes and materialized views, is a typical reason for suboptimal SQL performance. The optimal set of access structures can improve SQL performance by orders of magnitude.

   - Stale optimizer statistics

     Statistics gathered by `DBMS_STATS` can become stale when the statistics maintenance operations, either automatic or manual, cannot keep up with the changes to the table data caused by DML. Because stale statistics on a table do not accurately reflect the table data, the optimizer can make decisions based on faulty information and generate suboptimal execution plans.

   - Hardware problems

     Suboptimal performance might be connected with memory, I/O, and CPU problems.

4. Defining the scope of the problem

   The scope of the solution must match the scope of the problem. Consider a problem at the database level and a problem at the statement level. For example, the shared pool is too small, which causes cursors to age out quickly, which in turn causes many hard parses. Using an initialization parameter to increase the shared pool size fixes the problem at the database level and improves performance for all sessions. However, if a single SQL statement is not using a helpful index, then changing the optimizer initialization parameters for the entire database could harm overall performance. If a single SQL statement has a problem, then an appropriately scoped solution addresses just this problem with this statement.

5. Implementing corrective actions for suboptimally performing SQL statements

   These actions vary depending on circumstances. For example, you might rewrite a SQL statement to be more efficient, avoiding unnecessary hard parsing by rewriting the statement to use bind variables. You might also use equijoins, remove functions from `WHERE` clauses, and break a complex SQL statement into multiple simple statements.

   In some cases, you improve SQL performance not by rewriting the statement, but by restructuring schema objects. For example, you might index a new access path, or reorder columns in a concatenated index. You might also partition a table, introduce derived values, or even change the database design.

6. Preventing SQL performance regressions

   To ensure optimal SQL performance, verify that execution plans continue to provide optimal performance, and choose better plans if they come available. You can achieve these goals using optimizer statistics, SQL profiles, and SQL plan baselines.

See Also:

- "Shared Pool Check"
- Oracle Database Concepts to learn more about the shared pool

1.4.2 SQL Tuning Tools

SQL tuning tools are either automated or manual.

In this context, a tool is automated if the database itself can provide diagnosis, advice, or corrective actions. A manual tool requires you to perform all of these operations.

All tuning tools depend on the basic tools of the dynamic performance views, statistics, and metrics that the database instance collects. The database itself contains the data and metadata required to tune SQL statements.

1.4.2.1 Automated SQL Tuning Tools

Oracle Database provides several advisors relevant for SQL tuning.

Additionally, SQL plan management is a mechanism that can prevent performance regressions and also help you to improve SQL performance.

All of the automated SQL tuning tools can use SQL tuning sets as input. A SQL tuning set (STS) is a database object that includes one or more SQL statements along with their execution statistics and execution context.

See Also:

- "About SQL Tuning Sets"
- Oracle Database Get Started with Performance Tuning to learn more about managing SQL tuning sets

1.4.2.1.1 Automatic Database Diagnostic Monitor (ADDM)

ADDM is self-diagnostic software built into Oracle Database.

ADDM can automatically locate the root causes of performance problems, provide recommendations for correction, and quantify the expected benefits. ADDM also identifies areas where no action is necessary.

ADDM and other advisors use Automatic Workload Repository (AWR), which is an infrastructure that provides services to database components to collect, maintain, and use statistics. ADDM examines and analyzes statistics in AWR to determine possible performance problems, including high-load SQL.

For example, you can configure ADDM to run nightly. In the morning, you can examine the latest ADDM report to see what might have caused a problem and if there is a recommended fix. The report might show that a particular `SELECT` statement consumed a huge amount of CPU, and recommend that you run SQL Tuning Advisor.

See Also:

- Oracle Database Get Started with Performance Tuning
- Oracle Database Performance Tuning Guide

1.4.2.1.2 SQL Tuning Advisor

SQL Tuning Advisor is internal diagnostic software that identifies problematic SQL statements and recommends how to improve statement performance.

When run during database maintenance windows as an automated maintenance task, SQL Tuning Advisor is known as Automatic SQL Tuning Advisor.

SQL Tuning Advisor takes one or more SQL statements as an input and invokes the Automatic Tuning Optimizer to perform SQL tuning on the statements. The advisor performs the following types of analysis:

- Checks for missing or stale statistics
- Builds SQL profiles

  A SQL profile is a set of auxiliary information specific to a SQL statement. A SQL profile contains corrections for suboptimal optimizer estimates discovered during Automatic SQL Tuning. This information can improve optimizer estimates for cardinality, which is the number of rows that is estimated to be or actually is returned by an operation in an execution plan, and selectivity. These improved estimates lead the optimizer to select better plans.
- Explores whether a different access path can significantly improve performance
- Identifies SQL statements that lend themselves to suboptimal plans

The output is in the form of advice or recommendations, along with a rationale for each recommendation and its expected benefit. The recommendation relates to a collection of statistics on objects, creation of new indexes, restructuring of the SQL statement, or creation of a SQL profile. You can choose to accept the recommendations to complete the tuning of the SQL statements.

See Also:

- "Analyzing SQL with SQL Tuning Advisor"
- Oracle Database Get Started with Performance Tuning

1.4.2.1.3 SQL Access Advisor

SQL Access Advisor is internal diagnostic software that recommends which materialized views, indexes, and materialized view logs to create, drop, or retain.

SQL Access Advisor takes an actual workload as input, or the advisor can derive a hypothetical workload from the schema. SQL Access Advisor considers the trade-offs between space usage and query performance, and

recommends the most cost-effective configuration of new and existing materialized views and indexes. The advisor also makes recommendations about partitioning.

See Also:

- "About SQL Access Advisor"
- Oracle Database Get Started with Performance Tuning
- Oracle Database Administrator's Guide to learn more about automated indexing
- Oracle Database Licensing Information User Manual for details on whether automated indexing is supported for different editions and services

1.4.2.1.4 Automatic Indexing

Oracle Database can constantly monitor the application workload, creating and managing indexes automatically.

Note:

See Oracle Database Licensing Information User Manual for details on which features are supported for different editions and services.

Creating indexes manually requires deep knowledge of the data model, application, and data distribution. Often DBAs make choices about which indexes to create, and then never revise their choices. As a result, opportunities for improvement are lost, and unnecessary indexes can become a performance liability. Automatic index management solves this problem.

1.4.2.1.4.1 How Automatic Indexing Works

The automatic indexing process runs in the background every 15 minutes and performs the following operations:

1. Automatic index candidates are identified based on the usage of table columns in SQL statements.
   - Tables that have high DML activity are excluded if the performance overhead of index maintenance during DML outweighs the benefit of improved query performance.
   - Automatic indexes can be single-column or multi-column.
   - Tables with stale or missing optimizer statistics are not considered for automatic indexes until statistics are gathered and fresh.
2. Index candidates are initially created invisible and unusable. At this stage, they are metadata only and not visible to the application workload. Index candidates can be:
   - Table columns (including virtual columns).
   - Partitioned and non-partitioned tables.
   - Selected expressions (for example, JSON expressions).
   - Single or multi-column.
3. A sample of workload SQL statements is test parsed to determine which indexes are determined by the optimizer to be beneficial. Indexes deemed useful by the optimizer are built and made valid so that the performance effect on SQL statements can be measured. All candidate indexes remain invisible during the verification step.

   Indexes that are not identified as useful by the optimizer remain invisible and unusable.
4. A sample of workload SQL statements is test executed to determine which indexes improve performance.

5. Candidate valid indexes found to improve SQL performance will be made visible and available to the application workload. Candidate indexes that do not improve SQL performance will revert to invisible and be unusable after a short delay.

   During the verification stage, if an index is found to be beneficial, but an individual SQL statement suffers a performance regression, a SQL plan baseline is created to prevent the regression when the index is made visible.

6. Unused automatic indexes are dropped by the automatic indexing process after the configurable retention period has expired.

   Note:

   By default, the unused automatic indexes are deleted after 373 days. The period for retaining the unused automatic indexes in a database can be configured using the `DBMS_AUTO_INDEX.CONFIGURE` procedure.

See Also:

Configuring Automatic Indexing in Oracle Database

1.4.2.1.4.2 Enabling and Disabling Automatic Indexing

The DBMS_AUTO_INDEX package provides options for configuring, dropping, monitoring, and reporting on automatic indexing.

You can use the DBMS_AUTO_INDEX package to do the following:

- Enable automatic indexing.

  `EXEC DBMS_AUTO_INDEX.CONFIGURE('AUTO_INDEX_MODE','IMPLEMENT')`

- Configure additional settings, such as how long to retain unused auto indexes, in days.

  `EXEC DBMS_AUTO_INDEX.CONFIGURE('AUTO_INDEX_RETENTION_FOR_AUTO','180')`

- Drop an automatic index. Carefully note the use of single and double quotation marks in the first example.

  Drop a single index owned by a schema and allow recreate.

  `Copy` `EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('SH','"SYS_AI_c0cmdvbzgyq94"',TRUE)`

  Drop all indexes owned by a schema and allow recreate.

  `Copy` `EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('SH',NULL,TRUE)`

  Drop all indexes owned by a schema and disallow recreate. Then, change the recreation status back to `allow`.

  `Copy` `EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('HR',NULL)`
  `EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('HR',NULL,TRUE)`

- Report on auto index configuration settings.

  ```
  Copy

  SELECT parameter_name, parameter_value FROM dba_auto_index_config;
  ```

Additional Controls

By setting the OPTIMIZER_SESSION_TYPE initialization parameter to ADHOC in a session, you can suspend automatic indexing for SQL statements in this session. The automatic indexing process does not identify index candidates, or create and verify indexes.

```
Copy

ALTER SESSION SET optimizer_session_type = 'ADHOC';
```

See Also:

- Oracle Database Administrator's Guide to learn more about automatic indexing
- See DBMS_AUTO_INDEX in the *Oracle Database PL/SQL Packages and Types Reference* to learn about the procedures and functions available in the `DBMS_AUTO_INDEX` package
- Oracle Database Reference to learn more about `OPTIMIZER_SESSION_TYPE`.

1.4.2.1.4.2.1 Accounting for DML Overhead

Indexes must be maintained when data is changed. This increases the overhead of DML operations such as

```
INSERT
```

,

```
UPDATE
```

, and

```
DELETE
```

. For example, improvements in query performance may be offset by the cost of DML on tables with significant write activity. Automatic indexes factors in this overhead when deciding whether a new index is beneficial or not. This functionality is controlled by the

```
CONFIGURE
```

parameter

```
AUTO_INDEX_INCLUDE_DML_COST
```

, which by default is

```
ON
```

. The feature can be disabled as follows:

```
Copy

EXEC DBMS_AUTO_INDEX.CONFIGURE('AUTO_INDEX_INCLUDE_DML_COST','ON')
```

You can use SQL to view the current setting of this parameter:

```
Copy

SELECT parameter_name,parameter_value FROM DBA_AUTO_INDEX_CONFIG WHERE parameter_name
= 'AUTO_INDEX_INCLUDE_DML_COST';
```

1.4.2.1.5 SQL Plan Management

SQL plan management is a preventative mechanism that enables the optimizer to automatically manage execution plans, ensuring that the database uses only known or verified plans.

This mechanism can build a SQL plan baseline, which contains one or more accepted plans for each SQL statement. By using baselines, SQL plan management can prevent plan regressions from environmental changes, while permitting the optimizer to discover and use better plans.

See Also:

- "Overview of SQL Plan Management"
- Oracle Database PL/SQL Packages and Types Referenceto learn about the `DBMS_SPM` package

1.4.2.1.5.1 How Automatic SQL Plan Management Works

With Automatic SPM (SQL plan management), plan performance regressions are detected and repaired automatically.

Automatic SPM is enabled using `DBMS_SPM.CONFIGURE`. The Automatic SPM Evolve Advisor itself always runs in the background, but SQL execution plan performance evaluation can be configured to run in either the foreground or the background.

When `DBMS_SPM.CONFIGURE` is set to ON:

This configures automatic SPM to use its background verification mode:

```
Copy

EXEC DBMS_SPM.CONFIGURE('AUTO_SPM_EVOLVE_TASK', 'ON');
```

In this setting, the high frequency Automatic SPM Evolve Advisor does the following.

- Inspects the Automatic Workload Repository (AWR) and the Automatic SQL Tuning Set (ASTS) to identify SQL statements that consume significant system resources.
- Locates alternative SQL execution plans located in the ASTS.
- Test executes the alternative plans and compares their performance.
- Evaluates which plans perform best and creates SQL plan baselines to enforce them.

When `DBMS_SPM.CONFIGURE` is set to AUTO:

This configures automatic SPM to use real-time mode:

```
Copy

EXEC DBMS_SPM.CONFIGURE('AUTO_SPM_EVOLVE_TASK', 'AUTO');
```

The high frequency Automatic SPM Evolve Advisor continues to operate in the background. However, SQL execution plan performance is evaluated immediately in the foreground:

- When a SQL statement is executed, and the execution plan is new, then after execution the performance of the SQL statement is compared with the performance of the plans known and stored in ASTS to check if a more optimal alternative SQL execution plan already exists.
- If a previous plan performs better than the new plan, a SQL plan baseline is created to enforce the more optimal previous plan.

1.4.2.1.6 SQL Performance Analyzer

SQL Performance Analyzer determines the effect of a change on a SQL workload by identifying performance divergence for each SQL statement.

System changes such as upgrading a database or adding an index may cause changes to execution plans, affecting SQL performance. By using SQL Performance Analyzer, you can accurately forecast the effect of system changes on SQL performance. Using this information, you can tune the database when SQL performance regresses, or validate and measure the gain when SQL performance improves.

See Also:

Oracle Database Testing Guide

1.4.2.1.7 SQL Transpiler

The SQL Transpiler automatically and wherever possible converts (*transpiles*) PL/SQL functions within SQL into SQL expressions, without user intervention.

Expressions in Oracle SQL statements are allowed to call PL/SQL functions. But these calls incur overhead because the PL/SQL runtime must be invoked. The SQL compiler automatically attempts to convert any PL/SQL function called from a SQL statement into a semantically equivalent SQL expression. Transpiling PL/SQL functions into SQL increases performance for new and existing programs and functions. When a transpiled PL/SQL function is invoked, the per row cost of executing the transpiled code within SQL is much lower than switching from the SQL runtime to the PL/SQL runtime in order to execute the original PL/SQL code.

If the transpiler cannot convert a PL/SQL function to SQL, execution of the function falls back to the PL/SQL runtime. Not all PL/SQL constructs are supported by the transpiler.

The entire operation is transparent to users

The following example shows a `SELECT` statement with a call to the PL/SQL function `GET_MONTH_ABBREVIATION`. The function extracts the three letter month abbreviation from the parameter `date_value`. In the Predicate Information section at the bottom of the plan, you can see that `GET_MONTH_ABBREVIATION` is replaced with `TO_CHAR(INTERNAL_FUNCTION("HIRE_DATE"),'MON','NLS_DATE_LANGUAGE=English')='MAY')`. This indicates that transpilation has occurred.

```
Copy

create function get_month_abbreviation (
   date_value date
) return varchar2 is
begin
  return to_char ( date_value, 'MON', 'NLS_DATE_LANGUAGE=English' );
end;
/
```

```
alter session set sql_transpiler = ON;

select employee_id, first_name, last_name
from hr.employees
where get_month_abbreviation ( hire_date ) = 'MAY';

EMPLOYEE_ID   FIRST_NAME     LAST_NAME
      ----------------------------------
        104   Bruce          Ernst
        115   Alexander      Khoo
        122   Payam          Kaufling
        174   Ellen          Abel
        178   Kimberely      Grant
        197   Kevin          Feeney

select *
from  dbms_xplan.display_cursor ( format => 'BASIC +PREDICATE' );

---------------------------------------
| Id  | Operation         | Name      |
---------------------------------------
|   0 | SELECT STATEMENT  |           |
|  *1 | TABLE ACCESS FULL | EMPLOYEES |
---------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
    1 - filter(TO_CHAR(INTERNAL_FUNCTION("HIRE_DATE"),'MON','NLS_DATE_LANGUAGE=Englis
h')='MAY')
```

If transpilation occurs, the database replaces the name of the function in the predicate section with the SQL expression. This is what is happening in the above example.

Note:

If transpilation did not occur, the predicate section in this example would show the following:

```
Copy

Predicate Information (identified by operation id):
---------------------------------------------------
    1 - filter("GET_MONTH_ABBREVIATION"("HIRE_DATE")='MAY')
```

1.4.2.1.7.1 Enabling or Disabling the SQL Transpiler

The SQL Transpiler is disabled by default. You can enable it with an ALTER SYSTEM command.

You can change the parameter value using `SQL_TRANSPILER`.

```
Copy

SQL_TRANSPILER = [ON | OFF]
```

The parameter can be modified either at the system or session level.

1.4.2.1.7.2 Eligibility of PL/SQL Constructs for Transpilation

Not all PL/SQL constructs can be transpiled to SQL.

PL/SQL Constructs Eligible for Transpilation

The following PL/SQL language elements are supported by the SQL Transpiler:

- Basic SQL scalar types: CHARACTER, DATETIME, and NUMBER
- String types (CHAR, VARCHAR, VARCHAR2, NCHAR, and others.)
- Numeric types (NUMBER, BINARY DOUBLE, and others.)
- Date types (DATE, INTERVAL, and TIMESTAMP)
- Local variables (with optional initialization at declaration) and constants
- Parameters with optional (simple) default values
- Variable assignment statements
- Expressions which can be translated into equivalent SQL expressions
- RETURN statements
- Expressions and local variables of BOOLEAN type

PL/SQL Constructs not Eligible for Transpilation at This Time

- Embedded SQL statements. A transpiled function cannot contain a cursor declaration, explicit cursors, ref cursors, or an execute-immediate statement
- Functions defined inside of a PL/SQL package.
- Package variables, both public and private.
- PL/SQL Specific Scalar Types: PLS_INTEGER
- PL/SQL Aggregate Types: Records, Collections, and Tables
- Oracle Objects (ADT/UDT), XML, and JSON
- Deprecated Datatypes: LONG
- The %TYPE and %ROWTYPE attributes
- Package state (both constants and variables)
- Locally defined PL/SQL types
- Locally defined (nested) functions
- Calls to other PL/SQL functions (both schema-level and package level). This also precludes support for recursive function calls
- Control-flow statements like LOOP, GOTO, and RAISE
- Nested DECLARE-BEGIN-EXCEPTION blocks

- CASE control-flow statements (note that this is different from the SQL CASE expressions which are supported by both SQL and PL/SQL)
- Transaction processing like COMMIT, ROLLBACK, LOCK-TABLE, PRAGMA AUTONOMOUS TRANSACTION, SELECT-FOR-UPDATE, and others

1.4.2.2 Manual SQL Tuning Tools

In some situations, you may want to run manual tools in addition to the automated tools. Alternatively, you may not have access to the automated tools.

1.4.2.2.1 Execution Plans

Execution plans are the principal diagnostic tool in manual SQL tuning. For example, you can view plans to determine whether the optimizer selects the plan you expect, or identify the effect of creating an index on a table.

You can display execution plans in multiple ways. The following tools are the most commonly used:

- `DBMS_XPLAN`

  You can use the `DBMS_XPLAN` package methods to display the execution plan generated by the `EXPLAIN PLAN` command and query of `V$SQL_PLAN`.

- `EXPLAIN PLAN`

  This SQL statement enables you to view the execution plan that the optimizer would use to execute a SQL statement without actually executing the statement. See Oracle Database SQL Language Reference.

- `V$SQL_PLAN` and related views

  These views contain information about executed SQL statements, and their execution plans, that are still in the shared pool. See Oracle Database Reference.

- `AUTOTRACE`

  The `AUTOTRACE` command in SQL*Plus generates the execution plan and statistics about the performance of a query. This command provides statistics such as disk reads and memory reads. See SQL*Plus User's Guide and Reference.

1.4.2.2.2 Real-Time SQL Monitoring and Real-Time Database Operations

The Real-Time SQL Monitoring feature of Oracle Database enables you to monitor the performance of SQL statements while they are executing. By default, SQL monitoring starts automatically when a statement runs in parallel, or when it has consumed at least 5 seconds of CPU or I/O time in a single execution.

A database operation is a set of database tasks defined by end users or application code, for example, a batch job or Extraction, Transformation, and Loading (ETL) processing. You can define, monitor, and report on database operations. Real-Time Database Operations provides the ability to monitor composite operations automatically. The database automatically monitors parallel queries, DML, and DDL statements as soon as execution begins.

Oracle Enterprise Manager Cloud Control (Cloud Control) provides easy-to-use SQL monitoring pages. Alternatively, you can monitor SQL-related statistics using the `V$SQL_MONITOR` and `V$SQL_PLAN_MONITOR` views. You can use these views with the following views to get more information about executions that you are monitoring:

- `V$ACTIVE_SESSION_HISTORY`
- `V$SESSION`
- `V$SESSION_LONGOPS`

- `V$SQL`
- `V$SQL_PLAN`

See Also:

- "About Monitoring Database Operations"
- Oracle Database Reference to learn about the `V$` views

1.4.2.2.3 Application Tracing

A **SQL trace file** provides performance information on individual SQL statements: parse counts, physical and logical reads, misses on the library cache, and so on.

Trace files are sometimes useful for diagnosing SQL performance problems. You can enable and disable SQL tracing for a specific session using the `DBMS_MONITOR` or `DBMS_SESSION` packages. Oracle Database implements tracing by generating a trace file for each server process when you enable the tracing mechanism.

Oracle Database provides the following command-line tools for analyzing trace files:

- `TKPROF`

  This utility accepts as input a trace file produced by the SQL Trace facility, and then produces a formatted output file.

- `trcsess`

  This utility consolidates trace output from multiple trace files based on criteria such as session ID, client ID, and service ID. After `trcsess` merges the trace information into a single output file, you can format the output file with `TKPROF`. `trcsess` is useful for consolidating the tracing of a particular session for performance or debugging purposes.

End-to-End Application Tracing simplifies the process of diagnosing performance problems in multitier environments. In these environments, the middle tier routes a request from an end client to different database sessions, making it difficult to track a client across database sessions. End-to-End application tracing uses a client ID to uniquely trace a specific end-client through all tiers to the database.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_MONITOR` and `DBMS_SESSION`

1.4.2.2.4 Optimizer Hints

A **hint** is an instruction passed to the optimizer through comments in a SQL statement.

Hints enable you to make decisions normally made automatically by the optimizer. In a test or development environment, hints are useful for testing the performance of a specific access path. For example, you may know that a specific index is more selective for certain queries. In this case, you may use hints to instruct the optimizer to use a better execution plan, as in the following example:

```
Copy

SELECT /*+ INDEX (employees emp_department_ix) */
       employee_id, department_id
FROM   employees
WHERE  department_id > 50;
```

Sometimes the database may not use a hint because of typos, invalid arguments, conflicting hints, and hints that are made invalid by transformations. Starting in Oracle Database 19c, you can generate a report about which hints were used or not used during plan generation.

See Also:

- "Influencing the Optimizer with Hints"
- Oracle Database SQL Language Reference to learn more about hints

1.4.3 User Interfaces to SQL Tuning Tools

Cloud Control is a system management tool that provides centralized management of a database environment. Cloud Control provides access to most tuning tools.

By combining a graphical console, Oracle Management Servers, Oracle Intelligent Agents, common services, and administrative tools, Cloud Control provides a comprehensive system management platform.

You can access all SQL tuning tools using a command-line interface. For example, the `DBMS_SQLTUNE` package is the command-line interface for SQL Tuning Advisor.

Oracle recommends Cloud Control as the best interface for database administration and tuning. In cases where the command-line interface better illustrates a particular concept or task, this manual uses command-line examples. However, in these cases the tuning tasks include a reference to the principal Cloud Control page associated with the task.

1.5 About Automatic Error Mitigation

The database attempts automatic error mitigation for SQL statements that fail with an ORA-00600 error during SQL compilation.

An ORA-00600 is a severe error. It indicates that a process has encountered a low-level, unexpected condition. When a SQL statement fails with this error during the parse phase, automatic error mitigation traps it and attempts to resolve the condition. If a resolution is found, the database generates a SQL patch in order to adjust the SQL execution plan. If this patch enables the parse to complete successfully, then the ORA-00600 error is not raised and no exception is seen by the application.

How Automatic Error Mitigation Works

These series of examples show how automatic error mitigation can transparently fix ORA-00600 errors.

1. Consider the following error condition. The query has failed and raised a fatal exception.

```
Copy

SQL> SELECT count(*)
  2  FROM emp1 e1
  3  WHERE ename = (select max(ename) from emp2 e2 where e2.empno = e1.empno)
  4    AND empno = (select max(empno) from emp2 e2 where e2.empno = e1.empno)
  5    AND job = (select max(job) from emp2 e2 where e2.empno = e1.empno);

ERROR at line 3:
ORA-00600: internal error code, arguments: [kkqctcqincf0], [0], [], [], [], [],
[], [], [], [], [], []
```

2. Automatic error mitigation is then turned on in the session.

```
Copy

SQL> alter session set sql_error_mitigation = 'on';

Session altered.
```

3. If automatic error mitigation is enabled and it successfully resolves the error, the ORA-00600 message in Step 1 is not displayed, because no exception is returned to the application. The query has been executed successfully.

```
Copy

SQL> SELECT count(*)
  2  FROM emp1 e1
  3  WHERE ename = (select max(ename) from emp2 e2 where e2.empno = e1.empno)
  4    AND empno = (select max(empno) from emp2 e2 where e2.empno = e1.empno)
  5    AND job = (select max(job) from emp2 e2 where e2.empno = e1.empno);
  COUNT(*)
----------
       999
```

4. If you now look at the explain plan for this query, you can see in the **Note** section at the bottom that a SQL patch has been created to repair the query.

```
Copy

SQL> SELECT * FROM table(dbms_xplan.display_cursor(sql_id=>'5426r24y45gz0',cursor_
child_no=>1,format=>'basic +note'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
EXPLAINED SQL STATEMENT:
------------------------
SELECT count(*) FROM emp1 e1 where ename = (select max(ename) FROM emp2 e2 WHERE e
2.empno = e1.empno) AND empno = (select max(empno) FROM
emp2 e2 WHERE e2.empno = e1.empno) AND job = (select max(job) FROM emp2 e2 WHERE e
2.empno = e1.empno);

Plan hash value: 1226419153
-----------------------------------------
| Id  | Operation               | Name    |
-----------------------------------------
|   0 | SELECT STATEMENT        |         |
|   1 |  SORT AGGREGATE         |         |
|   2 |   HASH JOIN             |         |
```

```
|   3 |    HASH JOIN            |         |
|   4 |     HASH JOIN           |         |
|   5 |      TABLE ACCESS FULL  | EMP1    |
|   6 |      VIEW               | VW_SQ_3 |
|   7 |       SORT GROUP BY     |         |
|   8 |        TABLE ACCESS FULL| EMP2    |
|   9 |     VIEW                | VW_SQ_2 |
|  10 |      SORT GROUP BY      |         |
|  11 |       TABLE ACCESS FULL | EMP2    |
|  12 |    VIEW                 | VW_SQ_1 |
|  13 |     SORT GROUP BY       |         |
|  14 |      TABLE ACCESS FULL  | EMP2    |
----------------------------------------
Note
-----
   - cpu costing is off (consider enabling it)
   -SQL patch "SYS_SQLPTCH_AUTO_dq7z4ydz3b2ug" used for this statement
```

Tip:

You can get more information about the origin and type of a patched SQL problem by querying

```
DBA_SQL_PATCHES
```

and

```
DBA_SQL_ERROR_MITIGATIONS
```

.

```
Copy

SQL> SELECT name,signature,origin FROM dba_sql_patches
  2  /

    NAME                           SIGNATURE            ORIGIN
    ------------------------------------------------------------------------------
    SYS_SQLPTCH_AUTO_dq7z4ydz3b2ug  15789590553029872463  AUTO-FOREGROUND-REPAIR

SQL> SELECT m.sql_id, m.signature, m.problem_key, m.problem_type 2
FROM dba_sql_error_mitigations m;

    SQL_ID             SIGNATURE               PROBLEM_KEY             PROBLEM_TYPE
    ------------------------------------------------------------------------------------
---
    5426r24y45gz0      15789590553029872463    ORA 600 [kkqctcqincf0]  COMPILATION ER
ROR
```

See Also:

Although automatic error mitigation repairs ORA-00600 are transparent to your application, there are views you can inspect to get more information about the process.

The Database Error Message Reference defines ORA-00600, which is the internal error number for Oracle program exceptions.

The

Oracle Database Reference Manual

provides three views related to automatic error mitigation.

- SQL_ERROR_MITIGATION describes the properties of the `SQL_ERROR_MITIGATION` initialization parameter.
- DBA_SQL_ERROR_MITIGATIONS shows the actions performed by automatic error mitigation. It describes each successful error mitigation, based on SQL ID. The `MITIGATION_DETAILS` column provides information on SQL patches created by automatic error mitigation.
- DBA_SQL_PATCHES shows details of SQL patches that have been generated (including but not limited to patches created by automatic error mitigation). The `ORIGIN` column value for patches created by automatic error mitigation is `AUTO-FOREGROUND-REPAIR`.

The *Application Packaging and Types Reference* documents the `SQL_ERROR_MITIGATION` initialization parameter.

Consistency in Distributed Systems

A distributed system provides benefits such as scalability and fault tolerance. However, maintaining consistency across the distributed system is non-trivial. Consistency is vital to achieving reliability, deterministic system state, and improved user experience 1, 2.

A distributed system replicates the data across multiple servers to attain improved fault tolerance, scalability, and reliability 3. The consistency patterns (**consistency models**) are a set of techniques for data storage and data management in a distributed system 4. The consistency pattern determines the data propagation across the distributed system. Hence, the consistency pattern will impact the scalability, and reliability of the distributed system 2.

There are numerous consistency patterns in distributed systems. The choice of the consistency pattern depends on the system requirements and use cases because each consistency pattern has its benefits and drawbacks 4. Consistency patterns must be at the crux of multi-data center system architecture as it's non-trivial to maintain consistency across multiple data centers. The consistency patterns can be broadly categorized as follows 5, 4, 6, 2:

- strong consistency
- eventual consistency
- weak consistency

The eventual consistency model is an optimal choice for distributed systems that favor high availability and performance over consistency. Strong consistency is an optimal consistency model when the same data view must be visible across the distributed system without delay. In summary, each consistency model fits a different use case and system requirements 1, 2, 4.

Strong Consistency

In the strong consistency pattern, read operations performed on any server must always retrieve the data that was included in the latest write operation. The strong consistency pattern typically replicates data synchronously

across multiple servers 6, 3. Put another way, when a write operation is executed on a server, subsequent read operations on every other server must return the latest written data 2, 4, 1.

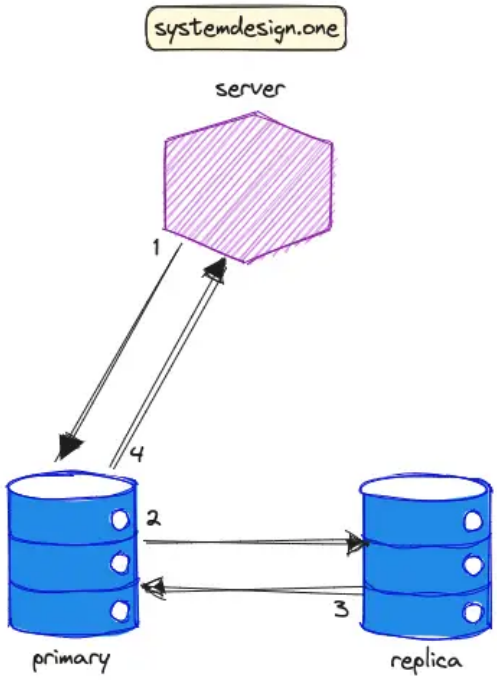
The benefits of strong consistency are the following 5:

- simplified application logic
- increased data durability
- guaranteed consistent data view across the system

The limitations of strong consistency are as follows 2, 4:

- reduced availability of the service
- degraded latency
- resource-intensive

Figure 1: Strong consistency



The workflow to reach strong consistency in data replication is the following 5:

1. the server (**client**) executes a write operation against the primary database instance
2. the primary instance propagates the written data to the replica instance
3. the replica instance sends an acknowledgment signal to the primary instance
4. the primary instance sends an acknowledgment signal to the client

The popular use cases of the strong consistency model are the following 6, 3, 4:

- File systems
- Relational databases
- Financial services such as banking

- Semi-distributed consensus protocols such as two-phase commit (**2PC**)
- Fully distributed consensus protocols such as Paxos

For instance, any changes to the user's bank account balance must be immediately replicated for improved durability and reliability. Google's Bigtable and Google's Spanner databases are real-world applications of strong consistency.

Eventual Consistency

In the eventual consistency pattern, when a write operation is executed against a server, the immediate subsequent read operations against other servers do not necessarily return the latest written data 1. The system will eventually converge to the same state and the latest data will be returned by other servers on succeeding read operations. The eventual consistency pattern typically replicates the data asynchronously across multiple servers 3, 6, 5, 4. In layman's terms, any data changes are only eventually propagated across the system and stale data views are expected until data convergence occurs.

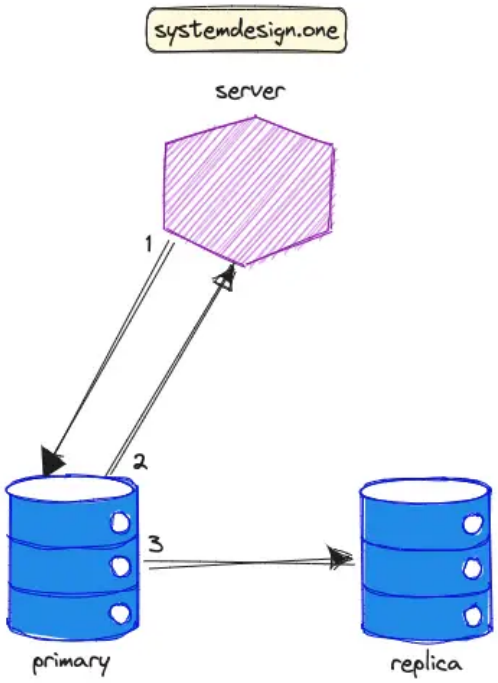
Eventual consistency can be implemented through multi-leader or leaderless replication topology. The system converges to the same state usually in a few seconds but the time frame depends on the implementation and system requirements. The benefits of eventual consistency pattern are as follows 5, 3, 2, 4, 7:

- simple
- highly available
- scalable
- low latency

The drawbacks of eventual consistency are the following 5, 4:

- weaker consistency model
- potential data loss
- potential data conflicts
- data inconsistency

Figure 2: Eventual consistency



The workflow to attain eventual consistency in data replication is the following 5:

1. the client executes a write operation against the primary database instance
2. the primary instance sends an acknowledgment signal to the client
3. the primary instance eventually propagates the written data to the replica instance

The eventual consistency pattern is a tradeoff between data staleness and scalability. The typical use cases of eventual consistency are the following 6, 3, 5, 4, 7, 1:

- search engine indexing
- URL shortener
- domain name server (**DNS**)
- simple mail transfer protocol (**SMTP**)
- object storage such as Amazon S3
- comments or posts on social media platforms such as Facebook
- distributed communication protocol such as gossip protocol
- leader-follower and multi-leader replication
- distributed counter and live comment service

For example, any changes to the domain name records are replicated eventually by DNS. Distributed databases such as Amazon Dynamo and Apache Cassandra are real-world applications of the eventual consistency pattern. Eventual consistency is not a design flaw but a feature to satisfy certain use cases. The business owner should determine whether application data is a candidate for the eventual consistency pattern 7.

Weak Consistency

In the weak consistency pattern, when a write operation is executed against a server, the subsequent read operations against other servers may or may not return the latest written data. In other words, a best-effort approach to data propagation is performed - the data may not be immediately propagated 6, 3, 4. The distributed system must meet various conditions such as the passing of time before the latest written data can be returned 1.

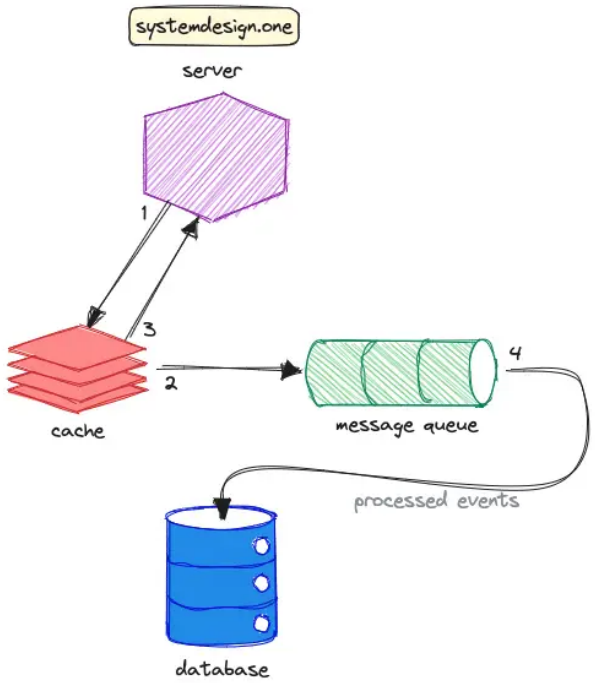
The advantages of weak consistency are the following 4:

- high availability
- low latency

The disadvantages of weak consistency are as follows 6, 4:

- potential data loss
- data inconsistency
- data conflicts

Figure 3: Weak consistency



The **write-behind** (write-back) cache pattern is an example of weak consistency. The data will be lost if the cache crashes before propagating the data to the database. The workflow of the write-behind cache pattern is the following:

1. the client executes a write operation against the cache server
2. the cache writes the received data to the message queue
3. the cache sends an acknowledgment signal to the client
4. the event processor asynchronously writes the data to the database

The common use cases of weak consistency are the following 6, 3, 4:

- Real-time multiplayer video games
- Voice over Internet Protocol (**VoIP**)
- Live streams
- Cache server
- Data backups

For instance, the lost video frames due to poor network connectivity are not retransmitted in a live stream.

Tradeoffs of Consistency Patterns

The tradeoffs associated with each consistency pattern can be outlined as the following 6:

	Backups	Leader-Follower	Multi-Leader	2PC	Paxos
Consistency	Weak	Eventual	Eventual	Strong	Strong
Transactions	No	Full	Local	Full	Full
Latency	Low	Low	Low	High	High
Throughput	High	High	High	Low	Medium
Data Loss	Lots	Some	Some	None	None
Failover	Down	Read only	Read/Write	Read/Write	Read/Write

Architecture principles: RPC vs. REST

In Remote Procedure Call (RPC), the client makes a remote function (also known as method or procedure) call on a server. Typically, one or more data values are passed to the server during the call.

In contrast, the REST client requests the server to perform an action on a specific server resource. Actions are limited to create, read, update, and delete (CRUD) only and are conveyed as HTTP verbs or HTTP methods.

RPC focuses on functions or actions, while REST focuses on resources or objects.

RPC principles

Next, we discuss some principles that RPC systems typically follow. However, these principles are not standardized like REST.

Remote invocation

An RPC call is made by a client to a function on the remote server as if it were called locally to the client.

Passing parameters

The client typically sends parameters to a server function, much the same as a local function.

Stubs

Function stubs exist on both the client and the server. On the client side, it makes the function call. On the server, it invokes the actual function.

REST principles

REST principles are standardized. A REST API must follow these principles to be classified as RESTful.

Client-server

The client-server architecture of REST decouples clients and servers. It treats them each as independent systems.

Stateless

The server keeps no record of the state of the client between client requests.

Cacheable

The client or intermediary systems may cache server responses based on whether a client specifies that the response may be cached.

Layered system

Intermediaries can exist between the client and the server. Both client and server have no knowledge of them and operate as if they were directly connected.

Uniform interface

The client and server communicate via a standardized set of instructions and messaging formats with the REST API. Resources are identified by their URL, and this URL is known as a REST API endpoint.

How they work: RPC vs. REST

In Remote Procedure Call (RPC), the client uses HTTP *POST* to call a specific function by name. Client-side developers must know the function name and parameters in advance for RPC to work.

In REST, clients and servers use HTTP verbs like *GET*, *POST*, *PATCH*, *PUT*, *DELETE*, and *OPTIONS* to perform options. Developers only need to know the server resource URLs and don't have to be concerned with individual function names.

The following table shows the type of code the client uses to perform similar actions in RPC and REST.

Action	RPC	REST	Comment
Adding a new product to a product list	POST /addProduct HTTP/1.1 HOST: api.example.com Content-Type: application/json {"name": "T-Shirt", "price": "22.00", "category": "Clothes"}	POST /products HTTP/1.1 HOST: api.example.com Content-Type: application/json {"name": "T-Shirt", "price": "22.00", "category": "Clothes"}	RPC uses *POST* on function, and REST uses *POST* on URL.
Retrieving a product's details	POST /getProduct HTTP/1.1 HOST: api.example.com Content-Type: application/json {"productID": "123"}	GET /products/123 HTTP/1.1 HOST: api.example.com	RPC uses *POST* on function and passes parameter as JSON object. REST uses *GET* on URL and passes parameter in URL.
Updating a product's price	POST /updateProductPrice HTTP/1.1 HOST: api.example.com	PUT /products/123 HTTP/1.1 HOST: api.example.com	RPC uses *POST* on function and passes parameter as JSON object. REST uses *PUT* on URL and passes parameter in URL and as JSON object.

	Content-Type: application/json {"productId": "123", "newPrice": "20.00"}	Content-Type: application/json {"price": "20.00"}	
Deleting a product	POST /deleteProduct HTTP/1.1 HOST: api.example.com Content-Type: application/json {"productId": "123""}	DELETE /products/123 HTTP/1.1 HOST: api.example.com	RPC uses *POST* on function and passes parameter as JSON object. REST uses *DELETE* on URL and passes parameter in URL.

Key differences: vs. REST

Remote Procedure Call (RPC) and REST are both ways to design corresponding client and server system interfaces for communication over the internet. However, the structure, implementation, and underlying principles differ. Systems designed with REST are known as RESTful APIs, whereas systems designed with RPC are simply RPC APIs.

Next, we give some more differences.

Time of development

RPC was developed in the late 1970s and early 1980s, while REST was a term first coined by computer scientist Roy Fielding in 2000.

Operational format

A REST API has a standardized set of server operations because of HTTP methods, but RPC APIs don’t. Some RPC implementations provide a framework for standardized operations.

Data passing format

REST can pass any data format and multiple formats, like JSON and XML, within the same API.

However, with RPC APIs, the data format is selected by the server and fixed during implementation. You can have specific JSON RPC or XML RPC implementations, and the client has no flexibility.

State

In the context of APIs, *stateless* refers to a design principle where the server does not store any information about the client's previous interactions. Each API request is treated independently, and the server does not rely on any stored client state to process the request.

REST systems must always be stateless, but RPC systems can be stateful or stateless, depending on design.

When to use: RPC vs. REST

Remote Procedure Call (RPC) is typically used to call remote functions on a server that require an action result. You can use it when you require complex calculations or want to trigger a remote procedure on the server, with the process hidden from the client.

Here are actions where RPC is a good option:

- Take a picture with a remote device's camera
- Use a machine learning algorithm on the server to identify fraud
- Transfer money from one account to another on a remote banking system
- Restart a server remotely

A REST API is typically used to perform create, read, update, and delete (CRUD) operations on a data object on a server. This makes REST APIs a good fit for cases when server data and data structures need to be exposed uniformly.

Here are actions where a REST API is a good option:

- Add a product to a database
- Retrieve the contents of a music playlist
- Update a person's address
- Delete a blog post

Why did REST replace RPC?

While REST web APIs are the norm today, Remote Procedure Call (RPC) hasn't disappeared. A REST API is typically used in applications as it is easier for developers to understand and implement. However, RPC still exists and is used when it suits the use case better.

Modern implementations of RPC, such as gRPC, are now more popular. For some use cases, gRPC performs better than RPC and REST. It allows streaming client-server communications rather than the request-and-respond data exchange pattern.

Summary of differences: RPC vs. REST

	RPC	**REST**
What is it?	A system allows a remote client to call a procedure on a server as if it were local.	A set of rules that defines structured data exchange between a client and a server.
Used for	Performing actions on a remote server.	Create, read, update, and delete (CRUD) operations on remote objects.
Best fit	When requiring complex calculations or triggering a remote process on the server.	When server data and data structures need to be exposed uniformly.
Statefulness	Stateless or stateful.	Stateless.
Data passing format	In a consistent structure defined by the server and enforced on the client.	In a structure determined independently by the server. Multiple different formats can be passed within the same API.

System Design: The Distributed Messaging Queue

Learn about the messaging queue, why we use it, and important use cases.

We'll cover the following

What is a messaging queue?

A **messaging queue** is an intermediate component between the interacting entities known as *producers* and *consumers*. The **producer** produces messages and places them in the queue, while the **consumer** retrieves the messages from the queue and processes them. There might be multiple producers and consumers interacting with the queue at the same time.

Here is an illustration of two applications interacting via a single messaging queue:

An example of two applications interacting via a single messaging queue

Motivation

A messaging queue has several advantages and use cases.

- **Improved performance:** A messaging queue enables asynchronous communication between the two interacting entities, producers and consumers, and eliminates their relative speed difference. A producer puts messages in the queue without waiting for the consumers. Similarly, a consumer processes the messages when they become available. Moreover, queues are often used to separate out slower operations from the critical path and, therefore, help reduce client-perceived latency. For example, instead of waiting for a specific task that's taking a long time to complete, the producer process sends a message, which is kept in a queue if there are multiple requests, for the required task and continues its operations. The consumer can notify us about the fate of the processing, whether a success or failure, by using another queue.
- **Better reliability:** The separation of interacting entities via a messaging queue makes the system more fault tolerant. For example, a producer or consumer can fail independently without affecting the others and restart later. Moreover, replicating the messaging queue on multiple servers ensures the system's availability if one or more servers are down.
- **Granular scalability:** Asynchronous communication makes the system more scalable. For example, many processes can communicate via a messaging queue. In addition, when the number of requests increases, we distribute the workload across several consumers. So, an application is in full control to tweak the number of producer or consumer processes according to its current need.
- **Easy decoupling:** A messaging queue decouples dependencies among different entities in a system. The interacting entities communicate via messages and are kept unaware of each other's internal working mechanisms.
- **Rate limiting:** Messaging queues also help absorb any load spikes and prevent services from becoming overloaded, acting as a rudimentary form of rate limiting when there is a need to avoid dropping any incoming request.
- **Priority queue:** Multiple queues can be used to implement different priorities—for example, one queue for each priority—and give more service time to a higher priority queue.

Messaging queue use cases

A messaging queue has many use cases, both in single-server and distributed environments. For example, it can be used for interprocess communication within one operating system. It also enables communication between processes in a distributed environment. Some of the use cases of a messaging queue are discussed below.

1. **Sending many emails:** Emails are used for numerous purposes, such as sharing information, account verification, resetting passwords, marketing campaigns, and more. All of these emails written for different purposes don't need immediate processing and, therefore, they don't disturb the system's core functionality. A messaging queue can help coordinate a large number of emails between different senders and receivers in such cases.
2. **Data post-processing:** Many multimedia applications need to process content for different viewer needs, such as for consumption on a mobile phone and a smart television. Oftentimes, applications upload the content into a store and use a messaging queue for post-processing of content offline. Doing this substantially reduces client-perceived latency and enables the service to schedule the offline work at some appropriate time—probably late at night when the compute capacity is less busy.
3. **Recommender systems:** Some platforms use recommender systems to provide preferred content or information to a user. The recommender system takes the user's historical data, processes it, and predicts relevant content or information. Since this is a time-consuming task, a messaging queue can be incorporated between the recommender system and requesting processes to increase and quicken performance.

How do we design a distributed messaging queue?

We divide the design of a distributed messaging queue into the following five lessons:

1. **Requirements:** Here, we focus on the functional and non-functional requirements of designing a distributed messaging queue. We also discuss a single server messaging queue and its drawbacks in this lesson.
2. **Design consideration:** In this lesson, we discuss some important factors that may affect the design of a distributed messaging queue—for example, the order of placing messages in a queue, their extraction, their visibility in the queue, and the concurrency of incoming messages.
3. **Design:** In this lesson, we discuss the design of a distributed messaging queue in detail. We also describe the process of replication of queues and the interaction between various building blocks involved in the design.
4. **Evaluation:** In this lesson, we evaluate the design of a distributed messaging queue based on its functional and non-functional requirements.
5. **Quiz:** At the end of the chapter, we evaluate understanding of the design of a distributed messages queue via a quiz.

TCP vs UDP: Differences between the protocols

The main difference between TCP (transmission control protocol) and UDP (user datagram protocol) is that TCP is a connection-based protocol and UDP is connectionless. While TCP is more reliable, it transfers data more slowly. UDP is less reliable but works more quickly. This makes each protocol suited to different types of data transfers.

Protocols are rules that govern how data is formatted and sent over a network. TCP and UDP are two different methods for doing the same job: transferring data via the internet. They enable servers and devices to communicate so you can send emails, watch Netflix, play games, and browse web pages.

TCP creates a secure communication line to ensure the reliable transmission of all data. Once a message is sent, the receipt is verified to make sure all the data was transferred.

UDP does not establish a connection when sending data. It sends data without confirming receipt or checking for errors. That means some or all of the data may be lost during transmission.

Here are the main differences between TCP and UDP:

Factor	TCP	UDP
Connection type	Requires an established connection before transmitting data	No connection is needed to start and end a data transfer
Data sequence	Can sequence data (send in a specific order)	Cannot sequence or arrange data
Data retransmission	Can retransmit data if packets fail to arrive	No data retransmitting. Lost data can't be retrieved
Delivery	Delivery is guaranteed	Delivery is not guaranteed
Check for errors	Thorough error-checking guarantees data arrives in its intended state	Minimal error-checking covers the basics but may not prevent all errors
Broadcasting	Not supported	Supported
Speed	Slow, but complete data delivery	Fast, but at risk of incomplete data delivery

Which protocol is better: TCP or UDP?

It depends on what you're doing online and the type of data being transferred. UDP is better if you're gaming online, because its speedy data transfer allows for mostly lag-free gaming. TCP is better if you're transferring files, like family photos, because it ensures the data arrives exactly as it was sent.

Overall, TCP and UDP are both useful protocols, so to think in terms of TCP vs UDP is a bit misleading. But depending on the type of data transfer, TCP or UDP might be better for the job. Here are some examples:

TCP is best for:

- Email or texting

- File transfers

- Web browsing

UDP is best for:

- Live streaming

- Online gaming

- Video chat

Here's a detailed breakdown of the advantages and disadvantages of TCP and UDP:

Advantages of TCP

Transmission control protocol (TCP) is the protocol to choose for maximum reliability and quality. It may not be the fastest, but it gets the job done right. Here are a few advantages of the TCP protocol:

- It sets up and maintains a connection between sender and receiver.
- It operates independently of the operating system.
- It supports many routing protocols.
- It checks for errors, guaranteeing data arrives at its destination unaltered.
- It confirms data arrival after delivery, or attempts to retransfer.
- It's able to send data in a particular sequence.
- It optimizes the pace of data transmission based on the receiver.

Disadvantages of TCP

TCP isn't suited for some types of data transfers, especially ones that require faster speeds. These are the drawbacks of TCP packet transmission:

- It uses more bandwidth and is slower than UDP.
- It's especially slow at the beginning of a file transfer.
- It can prevent data from loading if some data is lost. For example, it won't load images on a web page until all of the page data has been delivered.
- It reduces its transfer rate if the network is congested, resulting in even slower speeds.
- It's not suited for LAN and PAN networks.
- It can't multicast or broadcast.

Despite its slower speeds, TCP is the only protocol that can retransmit lost data packets. **When reliability is critical, TCP is the best option.**

Applications of TCP

When should you enable TCP data transfer? Most data transfers automatically use the best protocol option. But in certain circumstances — such as when using a VPN — you may need to choose a protocol to optimize your browsing experience. Enable TCP for the following activities:

- Email and text messaging

- Streaming pre-recorded content on sites like Netflix, Hulu, or HBO Max
- Transferring files between apps and devices
- General web browsing
- Remote device or network administration

Advantages of UDP

UDP delivers data rapidly, and it doesn't slow down or turn back to recollect lost data. This makes it an ideal protocol for delivering continuous data or broadcasting, such as for live streaming, video calling, and matching servers with IP addresses. Here are some of the advantages of UDP:

- No connection is needed to send or receive data, so apps and operating systems work faster.
- Broadcast and multicast transmission is available, meaning one UDP transmission can send data to multiple recipients.
- It endures packet loss, delivering data even if it's incomplete.
- Smaller packet size and less overhead reduce end-to-end delay.
- Operates over a larger range of network conditions than TCP.
- UDP communication is more efficient.
- It can transmit live and real-time data.

Disadvantages of UDP

While UDP provides the speed you need to live a comfortable digital life, UDP isn't as reliable as TCP. This is something to be aware of when setting up a VPN, because most VPNs run on UDP protocols to keep connection speeds high. Here are some disadvantages of using UDP:

- It's connectionless, which makes data transfer unreliable.
- There's no system in place to acknowledge a successful data transfer.
- There's no way to know if data is delivered in its original state, or at all.
- It has no error control, so it drops packets when errors are detected.
- In case of a data collision, routers will often drop UDP packets and favor TCP packets.
- Multiple users accepting UDP data can cause congestion, and there's no way to mitigate this.
- It cannot sequence data, so data can arrive in any order or out of order.

Applications of UDP

UDP is best suited for transferring a steady flow of live data. This allows many users to access data easily and quickly, if not in perfect condition. A good example is playing an online game. UDP can keep the action moving in spite of potential errors or data loss. Here are a few applications of UDP in real life.

- Online gaming
- Multicasting
- Video chatting/conferencing
- VoIP (in-app voice calling)
- Domain Name Systems (which translates domain names into IP addresses)

How does TCP work?

TCP works by using a “three-way handshake” — a three-step process that forms a connection between a device and a server. The completion of the three-step process establishes the non-stop connection, starts the transfer of data packets across the internet, delivers them intact, and acknowledges delivery.

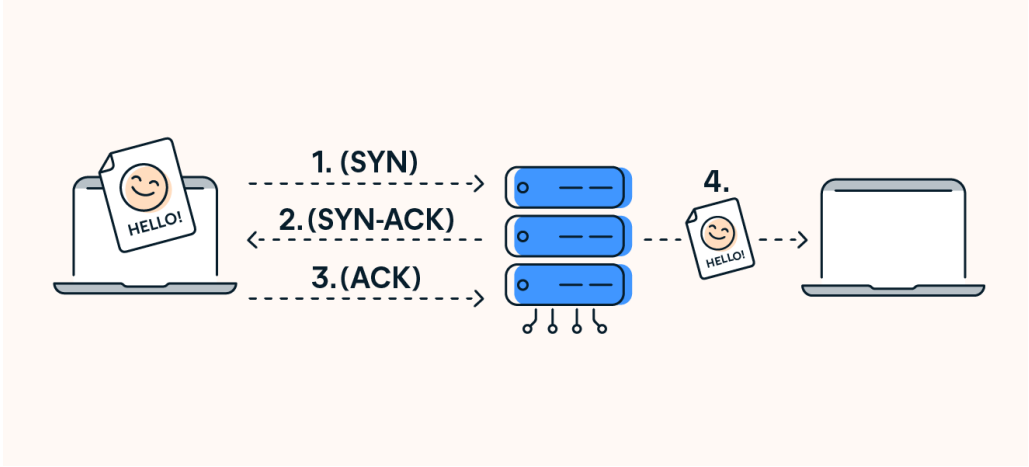
Here’s how TCP works:

1. **The client device initiating the data transfer sends a sequence number (SYN) to the server. It tells the server the number that the data packet transfer should begin with.**
2. **The server acknowledges the client SYN and sends its own SYN number. This step is often referred to as SYN-ACK (SYN acknowledgement).**
3. **The client then acknowledges (ACK) the server’s SYN-ACK, which forms a direct connection and begins the data transfer.**

The connection between the sender and receiver is maintained until the transfer is successful. Every time a data packet is sent, it requires an acknowledgment from the receiver. So, if no acknowledgment is received, the data is resent.

If an error is acknowledged, the faulty packet is discarded and the sender delivers a new one. Heavy traffic or other issues may also prevent data from being sent. In that case, the transmission is delayed (without breaking the connection).Thanks to these controls, successful data delivery is guaranteed with TCP.

TCP uses a three-step process that forms (and keeps) a connection between a device and a server.

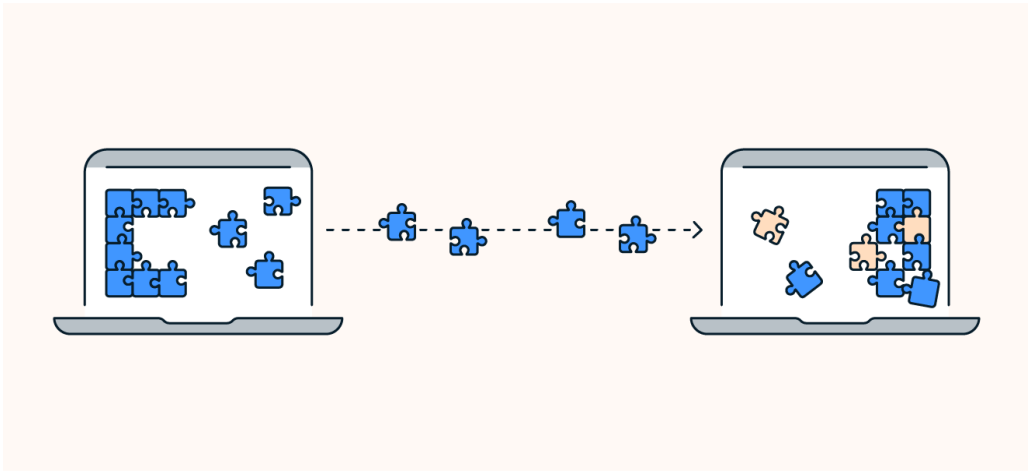


How does UDP work?

The UDP protocol works by immediately firing data at the receiver who made a data transmission request, until the transmission is complete or terminated. Sometimes called a “fire-and-forget” protocol, UDP fires data at a recipient in no particular sequence, without confirming delivery or checking if packets arrived as intended.

While TCP establishes a formal connection via its “handshake” agreement before sending data. UDP doesn’t have time for that. It speeds up data transfer by sending packets without making any agreement with a receiver. Then, it’s up to the recipient to make sense of the data.

UDP works by rapid-firing data from sender to receiver until the transfer is completed or terminated.



Here’s an analogy to help you understand how TCP and UDP work:

Imagine you’re having lunch at the office and a friend in a different cubicle asks you for half of your sandwich. You have two options: You can walk through the maze of office desks and hand it to her, guaranteeing a secure delivery. Or, you can throw the sandwich into her cubicle from across the room, leaving the quality of the delivery up to her speed and reflexes.

The first method (TCP) is reliable, but slow. The second method (UDP) is fast, but the sandwich might not arrive in its original state — or at all.

What is DNS?

The Domain Name System (DNS) is the phonebook of the Internet. Humans access information online through domain names, like nytimes.com or espn.com. Web browsers interact through Internet Protocol (IP) addresses. DNS translates domain names to IP addresses so browsers can load Internet resources.

Each device connected to the Internet has a unique IP address which other machines use to find the device. DNS servers eliminate the need for humans to memorize IP addresses such as 192.168.1.1 (in IPv4), or more complex newer alphanumeric IP addresses such as 2400:cb00:2048:1::c629:d7a2 (in IPv6).

How does DNS work?

The process of DNS resolution involves converting a hostname (such as www.example.com) into a computer-friendly IP address (such as 192.168.1.1). An IP address is given to each device on the Internet, and that address is necessary to find the appropriate Internet device - like a street address is used to find a particular home. When a user wants to load a webpage, a translation must occur between what a user types into their web browser (example.com) and the machine-friendly address necessary to locate the example.com webpage.

In order to understand the process behind the DNS resolution, it's important to learn about the different hardware components a DNS query must pass between. For the web browser, the DNS lookup occurs "behind the scenes" and requires no interaction from the user's computer apart from the initial request.

There are 4 DNS servers involved in loading a webpage:

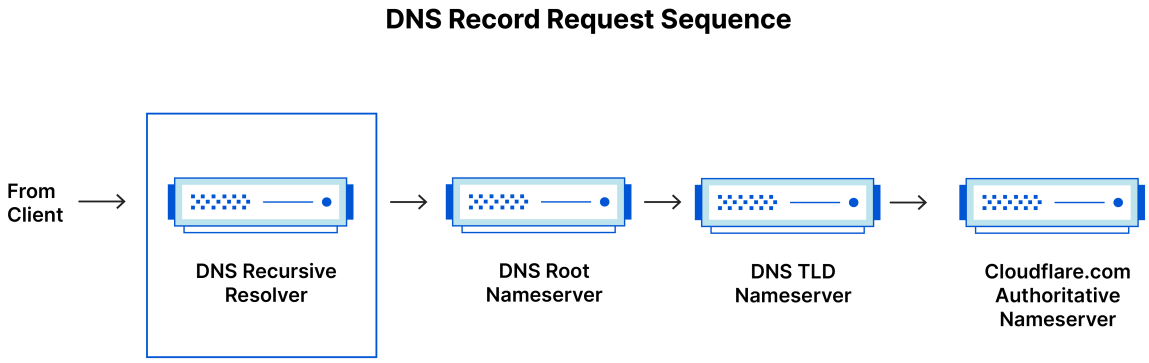
- **DNS recursor** - The recursor can be thought of as a librarian who is asked to go find a particular book somewhere in a library. The DNS recursor is a server designed to receive queries from client machines through applications such as web browsers. Typically the recursor is then responsible for making additional requests in order to satisfy the client's DNS query.
- **Root nameserver** - The root server is the first step in translating (resolving) human readable host names into IP addresses. It can be thought of like an index in a library that points to different racks of books - typically it serves as a reference to other more specific locations.
- **TLD nameserver** - The top level domain server (TLD) can be thought of as a specific rack of books in a library. This nameserver is the next step in the search for a specific IP address, and it hosts the last portion of a hostname (In example.com, the TLD server is "com").
- **Authoritative nameserver** - This final nameserver can be thought of as a dictionary on a rack of books, in which a specific name can be translated into its definition. The authoritative nameserver is the last stop in the nameserver query. If the authoritative name server has access to the requested record, it will return the IP address for the requested hostname back to the DNS Recursor (the librarian) that made the initial request.

What's the difference between an authoritative DNS server and a recursive DNS resolver?

Both concepts refer to servers (groups of servers) that are integral to the DNS infrastructure, but each performs a different role and lives in different locations inside the pipeline of a DNS query. One way to think about the difference is the recursive resolver is at the beginning of the DNS query and the authoritative nameserver is at the end.

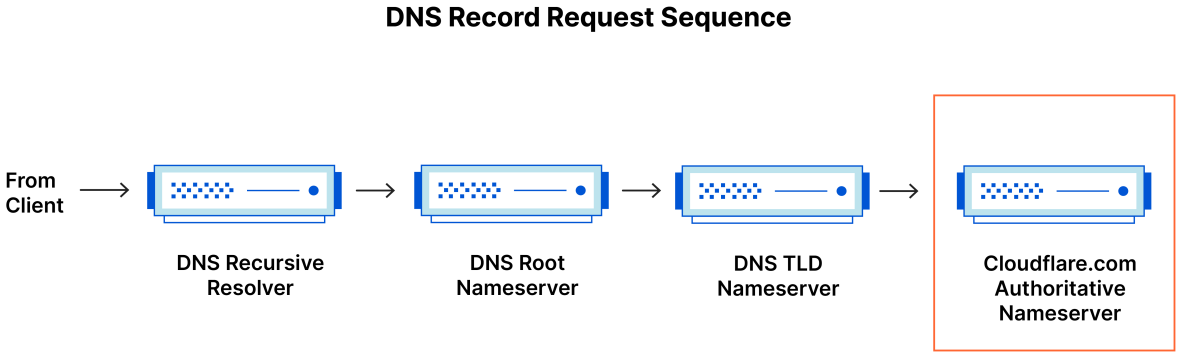
Recursive DNS resolver

The recursive resolver is the computer that responds to a recursive request from a client and takes the time to track down the DNS record. It does this by making a series of requests until it reaches the authoritative DNS nameserver for the requested record (or times out or returns an error if no record is found). Luckily, recursive DNS resolvers do not always need to make multiple requests in order to track down the records needed to respond to a client; caching is a data persistence process that helps short-circuit the necessary requests by serving the requested resource record earlier in the DNS lookup.

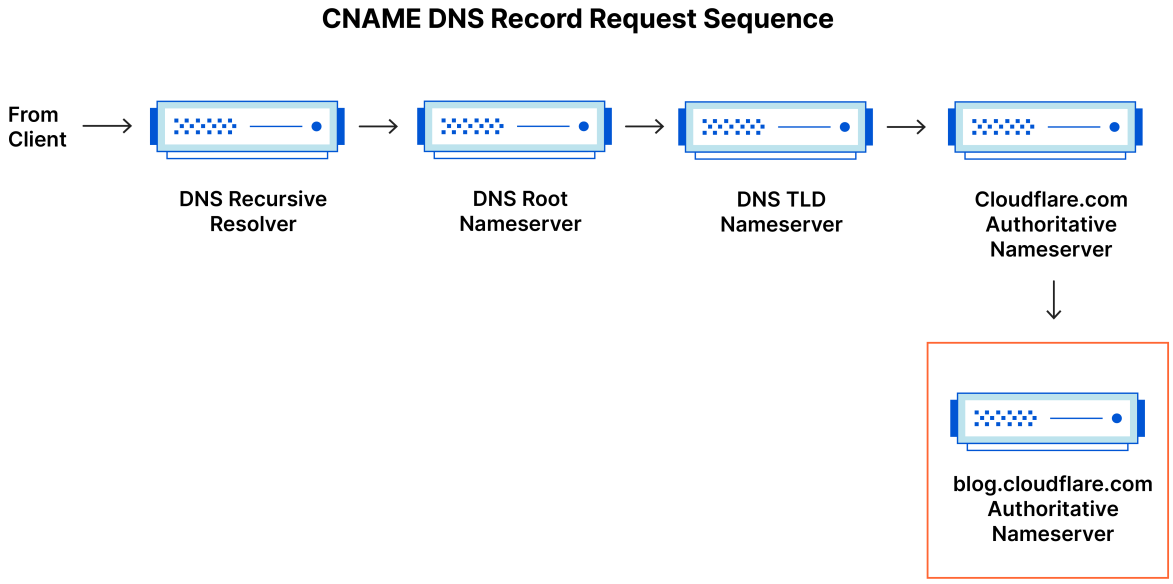


Authoritative DNS server

Put simply, an authoritative DNS server is a server that actually holds, and is responsible for, DNS resource records. This is the server at the bottom of the DNS lookup chain that will respond with the queried resource record, ultimately allowing the web browser making the request to reach the IP address needed to access a website or other web resources. An authoritative nameserver can satisfy queries from its own data without needing to query another source, as it is the final source of truth for certain DNS records.



It's worth mentioning that in instances where the query is for a subdomain such as foo.example.com or blog.cloudflare.com, an additional nameserver will be added to the sequence after the authoritative nameserver, which is responsible for storing the subdomain's CNAME record.



There is a key difference between many DNS services and the one that Cloudflare provides. Different DNS recursive resolvers such as Google DNS, OpenDNS, and providers like Comcast all maintain data center installations of DNS recursive resolvers. These resolvers allow for quick and easy queries through optimized clusters of DNS-optimized computer systems, but they are fundamentally different than the nameservers hosted by Cloudflare.

Cloudflare maintains infrastructure-level nameservers that are integral to the functioning of the Internet. One key example is the f-root server network which Cloudflare is partially responsible for hosting. The F-root is one of the root level DNS nameserver infrastructure components responsible for the billions of Internet requests per day. Our Anycast network puts us in a unique position to handle large volumes of DNS traffic without service interruption.

What are the steps in a DNS lookup?

For most situations, DNS is concerned with a domain name being translated into the appropriate IP address. To learn how this process works, it helps to follow the path of a DNS lookup as it travels from a web browser, through the DNS lookup process, and back again. Let's take a look at the steps.

Note: Often DNS lookup information will be cached either locally inside the querying computer or remotely in the DNS infrastructure. There are typically 8 steps in a DNS lookup. When DNS information is cached, steps are skipped from the DNS lookup process which makes it quicker. The example below outlines all 8 steps when nothing is cached.

The 8 steps in a DNS lookup:

1. A user types ‘example.com’ into a web browser and the query travels into the Internet and is received by a DNS recursive resolver.
2. The resolver then queries a DNS root nameserver (.).
3. The root server then responds to the resolver with the address of a Top Level Domain (TLD) DNS server (such as .com or .net), which stores the information for its domains. When searching for example.com, our

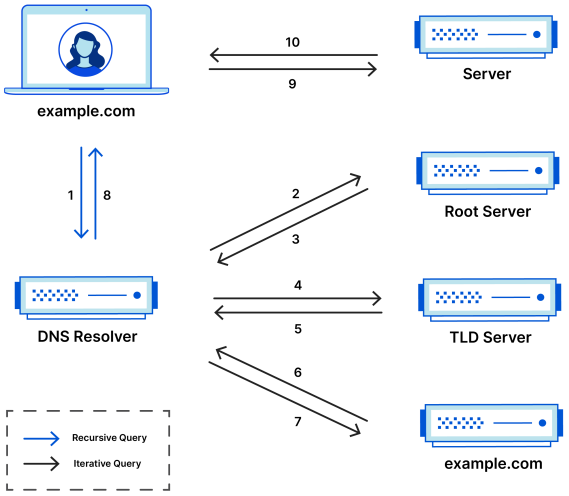
request is pointed toward the .com TLD.

4. The resolver then makes a request to the .com TLD.
5. The TLD server then responds with the IP address of the domain's nameserver, example.com.
6. Lastly, the recursive resolver sends a query to the domain's nameserver.
7. The IP address for example.com is then returned to the resolver from the nameserver.
8. The DNS resolver then responds to the web browser with the IP address of the domain requested initially.

Once the 8 steps of the DNS lookup have returned the IP address for example.com, the browser is able to make the request for the web page:

1. The browser makes a HTTP request to the IP address.
2. The server at that IP returns the webpage to be rendered in the browser (step 10).

Complete DNS Lookup and Webpage Query

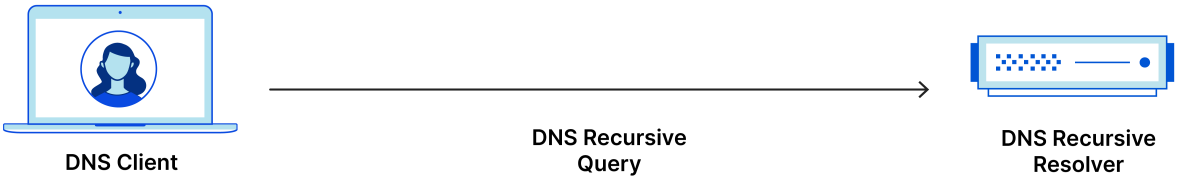


What is a DNS resolver?

The DNS resolver is the first stop in the DNS lookup, and it is responsible for dealing with the client that made the initial request. The resolver starts the sequence of queries that ultimately leads to a URL being translated into the necessary IP address.

Note: A typical uncached DNS lookup will involve both recursive and iterative queries.

It's important to differentiate between a recursive DNS query and a recursive DNS resolver. The query refers to the request made to a DNS resolver requiring the resolution of the query. A DNS recursive resolver is the computer that accepts a recursive query and processes the response by making the necessary requests.



What are the types of DNS queries?

In a typical DNS lookup three types of queries occur. By using a combination of these queries, an optimized process for DNS resolution can result in a reduction of distance traveled. In an ideal situation cached record data will be available, allowing a DNS name server to return a non-recursive query.

3 types of DNS queries:

1. **Recursive query** - In a recursive query, a DNS client requires that a DNS server (typically a DNS recursive resolver) will respond to the client with either the requested resource record or an error message if the resolver can't find the record.
2. **Iterative query** - in this situation the DNS client will allow a DNS server to return the best answer it can. If the queried DNS server does not have a match for the query name, it will return a referral to a DNS server authoritative for a lower level of the domain namespace. The DNS client will then make a query to the referral address. This process continues with additional DNS servers down the query chain until either an error or timeout occurs.
3. **Non-recursive query** - typically this will occur when a DNS resolver client queries a DNS server for a record that it has access to either because it's authoritative for the record or the record exists inside of its cache. Typically, a DNS server will cache DNS records to prevent additional bandwidth consumption and load on upstream servers.

What is DNS caching? Where does DNS caching occur?

The purpose of caching is to temporarily stored data in a location that results in improvements in performance and reliability for data requests. DNS caching involves storing data closer to the requesting client so that the DNS query can be resolved earlier and additional queries further down the DNS lookup chain can be avoided, thereby improving load times and reducing bandwidth/CPU consumption. DNS data can be cached in a variety of locations, each of which will store DNS records for a set amount of time determined by a time-to-live (TTL).

Browser DNS caching

Modern web browsers are designed by default to cache DNS records for a set amount of time. The purpose here is obvious; the closer the DNS caching occurs to the web browser, the fewer processing steps must be taken in order to check the cache and make the correct requests to an IP address. When a request is made for a DNS record, the browser cache is the first location checked for the requested record.

In Chrome, you can see the status of your DNS cache by going to chrome://net-internals/#dns.

Operating system (OS) level DNS caching

The operating system level DNS resolver is the second and last local stop before a DNS query leaves your machine. The process inside your operating system that is designed to handle this query is commonly called a “stub resolver” or DNS client. When a stub resolver gets a request from an application, it first checks its own cache to see if it has the record. If it does not, it then sends a DNS query (with a recursive flag set), outside the local network to a DNS recursive resolver inside the Internet service provider (ISP).

When the recursive resolver inside the ISP receives a DNS query, like all previous steps, it will also check to see if the requested host-to-IP-address translation is already stored inside its local persistence layer.

The recursive resolver also has additional functionality depending on the types of records it has in its cache:

1. If the resolver does not have the A records, but does have the NS records for the authoritative nameservers, it will query those name servers directly, bypassing several steps in the DNS query. This shortcut prevents lookups from the root and .com nameservers (in our search for example.com) and helps the resolution of the DNS query occur more quickly.
2. If the resolver does not have the NS records, it will send a query to the TLD servers (.com in our case), skipping the root server.
3. In the unlikely event that the resolver does not have records pointing to the TLD servers, it will then query the root servers. This event typically occurs after a DNS cache has been purged.

Caching Overview

Caching helps applications perform dramatically faster and cost significantly less at scale

Get Started with Caching

Free AWS Training | Advance your career with AWS Cloud Practitioner Essentials—a free, four-hour, foundational course »

What is Caching?

In computing, a cache is a high-speed data storage layer which stores a subset of data, typically transient in nature, so that future requests for that data are served up faster than is possible by accessing the data’s primary storage location. Caching allows you to efficiently reuse previously retrieved or computed data.

How does Caching work?

The data in a cache is generally stored in fast access hardware such as RAM (Random-access memory) and may also be used in correlation with a software component. A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer.

Trading off capacity for speed, a cache typically stores a subset of data transiently, in contrast to databases whose data is usually complete and durable.

Getting Started with Caching: Turbocharging Your Application Workloads

Caching Overview

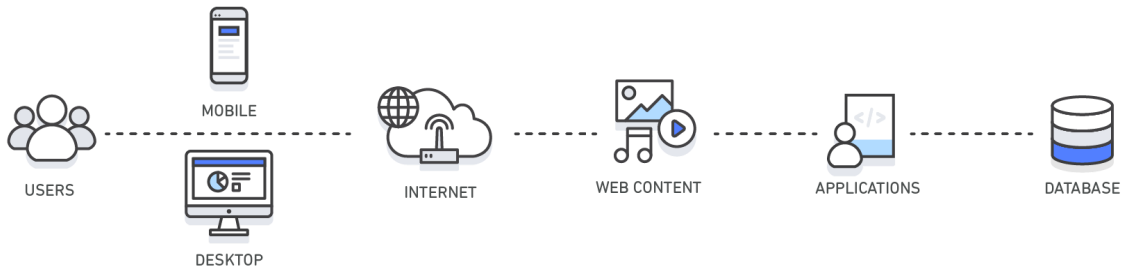
RAM and In-Memory Engines: Due to the high request rates or IOPS (Input/Output operations per second) supported by RAM and In-Memory engines, caching results in improved data retrieval performance and reduces cost at scale. To support the same scale with traditional databases and disk-based hardware, additional

resources would be required. These additional resources drive up cost and still fail to achieve the low latency performance provided by an In-Memory cache.

Applications: Caches can be applied and leveraged throughout various layers of technology including Operating Systems, Networking layers including Content Delivery Networks (CDN) and DNS, web applications, and Databases. You can use caching to significantly reduce latency and improve IOPS for many read-heavy application workloads, such as Q&A portals, gaming, media sharing, and social networking. Cached information can include the results of database queries, computationally intensive calculations, API requests/responses and web artifacts such as HTML, JavaScript, and image files. Compute-intensive workloads that manipulate data sets, such as recommendation engines and high-performance computing simulations also benefit from an In-Memory data layer acting as a cache. In these applications, very large data sets must be accessed in real-time across clusters of machines that can span hundreds of nodes. Due to the speed of the underlying hardware, manipulating this data in a disk-based store is a significant bottleneck for these applications.

Design Patterns: In a distributed computing environment, a dedicated caching layer enables systems and applications to run independently from the cache with their own lifecycles without the risk of affecting the cache. The cache serves as a central layer that can be accessed from disparate systems with its own lifecycle and architectural topology. This is especially relevant in a system where application nodes can be dynamically scaled in and out. If the cache is resident on the same node as the application or systems utilizing it, scaling may affect the integrity of the cache. In addition, when local caches are used, they only benefit the local application consuming the data. In a distributed caching environment, the data can span multiple cache servers and be stored in a central location for the benefit of all the consumers of that data.

Caching Best Practices: When implementing a cache layer, it's important to understand the validity of the data being cached. A successful cache results in a high hit rate which means the data was present when fetched. A cache miss occurs when the data fetched was not present in the cache. Controls such as TTLs (Time to live) can be applied to expire the data accordingly. Another consideration may be whether or not the cache environment needs to be Highly Available, which can be satisfied by In-Memory engines such as Redis. In some cases, an In-Memory layer can be used as a standalone data storage layer in contrast to caching data from a primary location. In this scenario, it's important to define an appropriate RTO (Recovery Time Objective--the time it takes to recover from an outage) and RPO (Recovery Point Objective--the last point or transaction captured in the recovery) on the data resident in the In-Memory engine to determine whether or not this is suitable. Design strategies and characteristics of different In-Memory engines can be applied to meet most RTO and RPO requirements.



Layer	Client-Side	DNS	Web	App	Database
Use Case	Accelerate retrieval of web content from websites (browser or device)	Domain to IP Resolution	Accelerate retrieval of web content from web/app servers. Manage Web Sessions (server side)	Accelerate application performance and data access	Reduce latency associated with database query requests

Technologies	HTTP Cache Headers, Browsers	DNS Servers	HTTP Cache Headers, CDNs, Reverse Proxies, Web Accelerators, Key/Value Stores	Key/Value data stores, Local caches	Database buffers, Key/Value data stores
Solutions	Browser Specific	Amazon Route 53	Amazon CloudFront, ElastiCache for Redis, ElastiCache for Memcached, Partner Solutions	Application Frameworks, ElastiCache for Redis, ElastiCache for Memcached, Partner Solutions	ElastiCache for Redis, ElastiCache for Memcached

Caching with Amazon ElastiCache

Amazon ElastiCache is a web service that makes it easy to deploy, operate, and scale an in-memory data store or cache in the cloud. The service improves the performance of web applications by allowing you to retrieve information from fast, managed, in-memory data stores, instead of relying entirely on slower disk-based databases. Learn how you can implement an effective caching strategy with this technical whitepaper on in-memory caching.

Benefits of Caching

Improve Application Performance

Because memory is orders of magnitude faster than disk (magnetic or SSD), reading data from in-memory cache is extremely fast (sub-millisecond). This significantly faster data access improves the overall performance of the application.

Reduce Database Cost

A single cache instance can provide hundreds of thousands of IOPS (Input/output operations per second), potentially replacing a number of database instances, thus driving the total cost down. This is especially significant if the primary database charges per throughput. In those cases the price savings could be dozens of percentage points.

Reduce the Load on the Backend

By redirecting significant parts of the read load from the backend database to the in-memory layer, caching can reduce the load on your database, and protect it from slower performance under load, or even from crashing at times of spikes.

Predictable Performance

A common challenge in modern applications is dealing with times of spikes in application usage. Examples include social apps during the Super Bowl or election day, eCommerce websites during Black Friday, etc. Increased load on the database results in higher latencies to get data, making the overall application performance unpredictable. By utilizing a high throughput in-memory cache this issue can be mitigated.

Eliminate Database Hotspots

In many applications, it is likely that a small subset of data, such as a celebrity profile or popular product, will be accessed more frequently than the rest. This can result in hot spots in your database and may require overprovisioning of database resources based on the throughput requirements for the most frequently used data. Storing common keys in an in-memory cache mitigates the need to overprovision while providing fast and predictable performance for the most commonly accessed data.

Increase Read Throughput (IOPS)

In addition to lower latency, in-memory systems also offer much higher request rates (IOPS) relative to a comparable disk-based database. A single instance used as a distributed side-cache can serve hundreds of thousands of requests per second.

Use Cases & Industries

Use CasesIndustries

- **Learn about various caching use cases**

Database Caching

The performance, both in speed and throughput that your database provides can be the most impactful factor of your application's overall performance. And despite the fact that many databases today offer relatively good performance, for a lot use cases your applications may require more. Database caching allows you to dramatically increase throughput and lower the data retrieval latency associated with backend databases, which as a result, improves the overall performance of your applications. The cache acts as an adjacent data access layer to your database that your applications can utilize in order to improve performance. A database cache layer can be applied in front of any type of database, including relational and NoSQL databases. Common techniques used to load data into you're a cache include lazy loading and write-through methods. For more information, click here.

Content Delivery Network (CDN)

When your web traffic is geo-dispersed, it's not always feasible and certainly not cost effective to replicate your entire infrastructure across the globe. A CDN provides you the ability to utilize its global network of edge locations to deliver a cached copy of web content such as videos, webpages, images and so on to your customers. To reduce response time, the CDN utilizes the nearest edge location to the customer or originating request location in order to reduce the response time. Throughput is dramatically increased given that the web assets are delivered from cache. For dynamic data, many CDNs can be configured to retrieve data from the origin servers.

Amazon CloudFront is a global CDN service that accelerates delivery of your websites, APIs, video content or other web assets. It integrates with other Amazon Web Services products to give developers and businesses an easy way to accelerate content to end users with no minimum usage commitments. To learn more about CDNs, click here.

Domain Name System (DNS) Caching

Every domain request made on the internet essentially queries DNS cache servers in order to resolve the IP address associated with the domain name. DNS caching can occur on many levels including on the OS, via ISPs and DNS servers.

Amazon Route 53 is a highly available and scalable cloud Domain Name System (DNS) web service.

Session Management

HTTP sessions contain the user data exchanged between your site users and your web applications such as login information, shopping cart lists, previously viewed items and so on. Critical to providing great user experiences on your website is managing your HTTP sessions effectively by remembering your user's preferences and providing rich user context. With modern application architectures, utilizing a centralized session management data store is the ideal solution for a number of reasons including providing, consistent

user experiences across all web servers, better session durability when your fleet of web servers is elastic and higher availability when session data is replicated across cache servers.

For more information, click here.

Application Programming Interfaces (APIs)

Today, most web applications are built upon APIs. An API generally is a RESTful web service that can be accessed over HTTP and exposes resources that allow the user to interact with the application. When designing an API, it's important to consider the expected load on the API, the authorization to it, the effects of version changes on the API consumers and most importantly the API's ease of use, among other considerations. It's not always the case that an API needs to instantiate business logic and/or make a backend requests to a database on every request. Sometimes serving a cached result of the API will deliver the most optimal and cost-effective response. This is especially true when you are able to cache the API response to match the rate of change of the underlying data. Say for example, you exposed a product listing API to your users and your product categories only change once per day. Given that the response to a product category request will be identical throughout the day every time a call to your API is made, it would be sufficient to cache your API response for the day. By caching your API response, you eliminate pressure to your infrastructure including your application servers and databases. You also gain from faster response times and deliver a more performant API.

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale.

Caching for Hybrid Environments

In a hybrid cloud environment, you may have applications that live in the cloud and require frequent access to an on-premises database. There are many network topologies that can by employed to create connectivity between your cloud and on-premises environment including VPN and Direct Connect. And while latency from the VPC to your on-premises data center may be low, it may be optimal to cache your on-premises data in your cloud environment to speed up overall data retrieval performance.

Web Caching

When delivering web content to your viewers, much of the latency involved with retrieving web assets such as images, html documents, video, etc. can be greatly reduced by caching those artifacts and eliminating disk reads and server load. Various web caching techniques can be employed both on the server and on the client side. Server side web caching typically involves utilizing a web proxy which retains web responses from the web servers it sits in front of, effectively reducing their load and latency. Client side web caching can include browser based caching which retains a cached version of the previously visited web content. For more information on Web Caching, click here.

General Cache

Accessing data from memory is orders of magnitude faster than accessing data from disk or SSD, so leveraging data in cache has a lot of advantages. For many use-cases that do not require transactional data support or disk based durability, using an in-memory key-value store as a standalone database is a great way to build highly performant applications. In addition to speed, application benefits from high throughput at a cost-effective price point. Referenceable data such product groupings, category listings, profile information, and so on are great use cases for a general cache. For more information on general cache, click here.

Integrated Cache

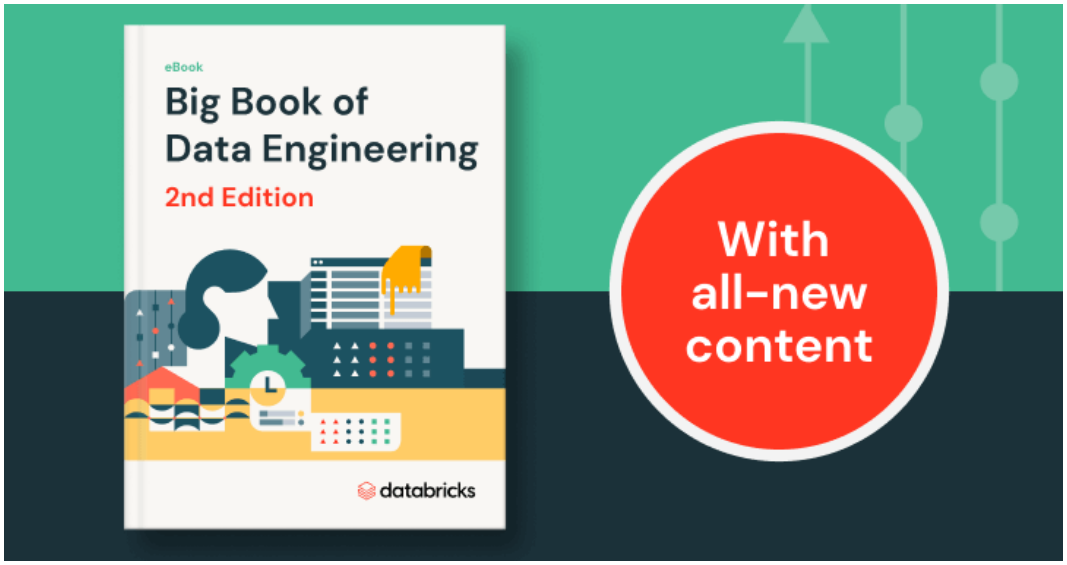
An integrated cache is an in-memory layer that automatically caches frequently accessed data from the origin database. Most commonly, the underlying database will utilize the cache to serve the response to the inbound database request given the data is resident in the cache. This dramatically increases the performance of the database by lowering the request latency and reducing CPU and memory utilization on the database engine. An important characteristic of an integrated cache is that the data cached is consistent with the data stored on disk by the database engine.

A.C.I.D. properties: Atomicity, Consistency, Isolation, and Durability

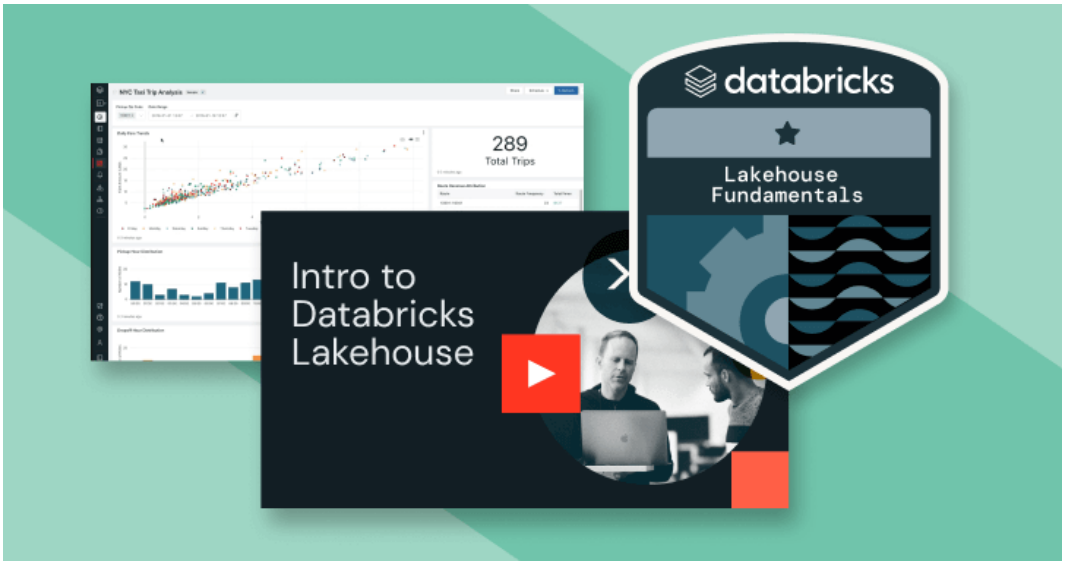
ACID is an acronym that refers to the set of 4 key properties that define a transaction: **Atomicity, Consistency, Isolation,** and **Durability.** If a database operation has these ACID properties, it can be called an ACID transaction, and data storage systems that apply these operations are called transactional systems. ACID transactions guarantee that each read, write, or modification of a table has the following properties:

- **Atomicity** - each statement in a transaction (to read, write, update or delete data) is treated as a single unit. Either the entire statement is executed, or none of it is executed. This property prevents data loss and corruption from occurring if, for example, if your streaming data source fails mid-stream.
- **Consistency** - ensures that transactions only make changes to tables in predefined, predictable ways. Transactional consistency ensures that corruption or errors in your data do not create unintended consequences for the integrity of your table.
- **Isolation** - when multiple users are reading and writing from the same table all at once, isolation of their transactions ensures that the concurrent transactions don't interfere with or affect one another. Each request can occur as though they were occurring one by one, even though they're actually occurring simultaneously.
- **Durability** - ensures that changes to your data made by successfully executed transactions will be saved, even in the event of system failure.

Here’s more to explore


eBook
Big Book of Data Engineering
2nd Edition
databricks
With all-new content


eBook
Why the Data Lakehouse Is Your Next Data Warehouse
2ND EDITION
databricks
Free Download



Why are ACID transactions a good thing to have?

ACID transactions ensure the highest possible data reliability and integrity. They ensure that your data never falls into an inconsistent state because of an operation that only partially completes. For example, without ACID transactions, if you were writing some data to a database table, but the power went out unexpectedly, it's possible that only some of your data would have been saved, while some of it would not. Now your database is in an inconsistent state that is very difficult and time-consuming to recover from.

Delta Lake: Reliable, consistent data with the guarantees of ACID transactions

DELTA LAKE

ACID transactions have long been one of the most enviable properties of data warehouses, but Delta Lake has now brought them to data lakes. They allow users to see consistent views of their data even while new data is being written to the table in real-time, because each write is an isolated transaction that is recorded in an ordered transaction log. [Delta Lake employs the highest level of isolation possible (serializable isolation), ensuring that reads and writes to a single table are consistent and reliable.] By implementing ACID transactions, Delta Lake effectively solves for several of the previously listed criticisms of Lambda architecture: its complexity, incorrect views of data, and the rework and reprocessing needed after Lambda pipelines inevitably break. Users can perform multiple concurrent transactions on their data, and in the event of an error with a data source or a stream, Delta Lake cancels execution of the transaction to ensure that the data is kept clean and intact. The beauty of ACID transactions is that users can trust the data that is stored in Delta Lake. A data analyst making use of Delta Lake tables to perform ETL on his or her data to ready it for dashboarding can count on the fact that the KPIs he or she is seeing represent the actual state of the data. A machine learning engineer using Delta Lake tables to perform feature engineering can be 100% confident that all of his or her transformations and aggregations either executed exactly as intended, or didn't execute at all (in which case, he or she would be

notified). The value of knowing that the mental model you have of your data is actually reflective of its true underlying state cannot be overstated.

Using Database Indexes Tutorial

DOWNLOAD AS PDF

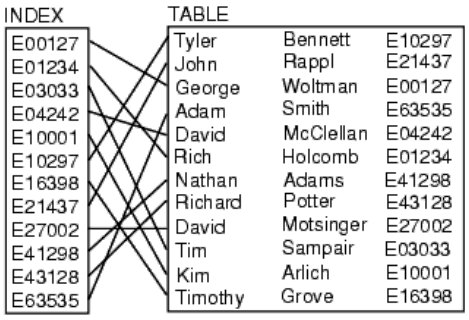
Introduction

There are a number of ways in which you can improve the performance of database activity using indexes. We provide only general guidelines that apply to most databases. Consult your database vendor's documentation for more detailed information.

For information regarding how to create and drop indexes, see your database system documentation.

An index is a database structure that you can use to improve the performance of database activity. A database table can have one or more indexes associated with it.

An index is defined by a field expression that you specify when you create the index. Typically, the field expression is a single field name, like EMP_ID. An index created on the EMP_ID field, for example, contains a sorted list of the employee ID values in the table. Each value in the list is accompanied by references to the records that contain that value.



A database driver can use indexes to find records quickly. An index on the EMP_ID field, for example, greatly reduces the time that the driver spends searching for a particular employee ID value. Consider the following Where clause:

```
WHERE emp_id = 'E10001'
```

Without an index, the driver must search the entire database table to find those records having an employee ID of E10001. By using an index on the EMP_ID field, however, the driver can quickly find those records.

Indexes may improve the performance of SQL statements. You may not notice this improvement with small tables but it can be significant for large tables; however, there can be disadvantages to having too many indexes. Indexes can slow down the performance of some inserts, updates, and deletes when the driver has to maintain the indexes as well as the database tables. Also, indexes take additional disk space.

BACK TO TOP

Improving Record Selection Performance

For indexes to improve the performance of selections, the index expression must match the selection condition exactly. For example, if you have created an index whose expression is last_name, the following Select statement

uses the index:

```
SELECT * FROM emp WHERE last_name = 'Smith'
```

This Select statement, however, does not use the index:

```
SELECT * FROM emp WHERE UPPER (last_name) = 'SMITH'
```

The second statement does not use the index because the Where clause contains UPPER(LAST_NAME), which does not match the index expression LAST_NAME. If you plan to use the UPPER function in all your Select statements and your database supports indexes on expressions, then you should define an index using the expression UPPER(LAST_NAME).

BACK TO TOP

Indexing Multiple Fields

If you often use Where clauses that involve more than one field, you may want to build an index containing multiple fields. Consider the following Where clause:

```
WHERE last_name = 'Smith' and first_name = 'Thomas'
```

For this condition, the optimal index field expression is LAST_NAME, FIRST_NAME. This creates a concatenated index.

Concatenated indexes can also be used for Where clauses that contain only the first of two concatenated fields. The LAST_NAME, FIRST_NAME index also improves the performance of the following Where clause (even though no first name value is specified):

```
last_name = 'Smith'
```

Consider the following Where clause:

```
WHERE last_name = 'Smith' and middle_name = 'Edward' and
first_name = 'Thomas'
```

If your index fields include all the conditions of the Where clause in that order, the driver can use the entire index. If, however, your index is on two nonconsecutive fields, say, LAST_NAME and FIRST_NAME, the driver can use only the LAST_NAME field of the index.

The driver uses only one index when processing Where clauses. If you have complex Where clauses that involve a number of conditions for different fields and have indexes on more than one field, the driver chooses an index to use. The driver attempts to use indexes on conditions that use the equal sign as the relational operator rather than conditions using other operators (such as greater than). Assume you have an index on the EMP_ID field as well as the LAST_NAME field and the following Where clause:

```
WHERE emp_id >= 'E10001' AND last_name = 'Smith'
```

In this case, the driver selects the index on the LAST_NAME field.

If no conditions have the equal sign, the driver first attempts to use an index on a condition that has a lower and upper bound, and then attempts to use an index on a condition that has a lower or upper bound. The driver always attempts to use the most restrictive index that satisfies the Where clause.

In most cases, the driver does not use an index if the Where clause contains an OR comparison operator. For example, the driver does not use an index for the following Where clause:

```
WHERE emp_id >= 'E10001' OR last_name = 'Smith'
```

BACK TO TOP

Deciding Which Indexes to Create

Before you create indexes for a database table, consider how you will use the table. The two most common operations on a table are to:

- Insert, update, and delete records
- Retrieve records

If you most often insert, update, and delete records, then the fewer indexes associated with the table, the better the performance. This is because the driver must maintain the indexes as well as the database tables, thus slowing down the performance of record inserts, updates, and deletes. It may be more efficient to drop all indexes before modifying a large number of records, and re-create the indexes after the modifications.

If you most often retrieve records, you must look further to define the criteria for retrieving records and create indexes to improve the performance of these retrievals. Assume you have an employee database table and you will retrieve records based on employee name, department, or hire date. You would create three indexes-one on the DEPT field, one on the HIRE_DATE field, and one on the LAST_NAME field. Or perhaps, for the retrievals based on the name field, you would want an index that concatenates the LAST_NAME and the FIRST_NAME fields (see "Indexing Multiple Fields" for details).

Here are a few rules to help you decide which indexes to create:

- If your record retrievals are based on one field at a time (for example, dept='D101'), create an index on these fields.
- If your record retrievals are based on a combination of fields, look at the combinations.
- If the comparison operator for the conditions is AND (for example, CITY = 'Raleigh' AND STATE = 'NC'), then build a concatenated index on the CITY and STATE fields. This index is also useful for retrieving records based on the CITY field.
- If the comparison operator is OR (for example, DEPT = 'D101' OR HIRE_DATE > {01/30/89}), an index does not help performance. Therefore, you need not create one.
- If the retrieval conditions contain both AND and OR comparison operators, you can use an index if the OR conditions are grouped. For example:In this case, an index on the DEPT field improves performance.

  ```
  dept = 'D101' AND (hire_date > {01/30/89} OR exempt = 1)
  ```

- If the AND conditions are grouped, an index does not improve performance. For example:

  ```
  (dept = 'D101' AND hire_date) > {01/30/89}) OR exempt = 1
  ```

BACK TO TOP

Improving Join Performance

When joining database tables, index tables can greatly improve performance. Unless the proper indexes are available, queries that use joins can take a long time.

Assume you have the following Select statement:

```
SELECT * FROM dept, emp WHERE dept.dept_id = emp.dept
```

In this example, the DEPT and EMP database tables are being joined using the department ID field. When the driver executes a query that contains a join, it processes the tables from left to right and uses an index on the second table's join field (the DEPT field of the EMP table).

To improve join performance, you need an index on the join field of the second table in the From clause. If there is a third table in the From clause, the driver also uses an index on the field in the third table that joins it to any previous table. For example:

```
SELECT * FROM dept, emp, addr
WHERE dept.dept_id = emp.dept AND emp.loc = addr.loc
```

In this case, you should have an index on the EMP.DEPT field and the ADDR.LOC field.

API Gateway

Application programming interfaces (APIs) are the most common way to connect users, applications, and services to each other in a modern IT environment. An *API gateway* is a component of the app-delivery infrastructure that sits between clients and services and provides centralized handling of API communication between them. It also delivers security, policy enforcement, and monitoring and visibility across on-premises, multi-cloud, and hybrid environments.

What Is an API Gateway?

An **API gateway** accepts API requests from a client, processes them based on defined policies, directs them to the appropriate services, and combines the responses for a simplified user experience. Typically, it handles a request by invoking multiple microservices and aggregating the results. It can also translate between protocols in legacy deployments.

For example, an e-commerce web site might use an API gateway to provide mobile clients with an endpoint for retrieving all product details with a single request. The gateway requests information from various services, like product availability and pricing, and combines the results.

API Gateway Capabilities

API gateways commonly implement capabilities that include:

- **Security policy** – Authentication, authorization, access control, and encryption
- **Routing policy** – Routing, rate limiting, request/response manipulation, circuit breaker, blue-green and canary deployments, A/B testing, load balancing, health checks, and custom error handling
- **Observability policy** – Real-time and historical metrics, logging, and tracing

For additional app- and API-level security, API gateways can be augmented with web application firewall (WAF) and denial of service (DoS) protection.

API Gateway Benefits

Deploying an API gateway for app delivery can help:

- **Reduce complexity and speed up app releases** by encapsulating the internal application architecture and providing APIs tailored for each client type
- **Streamline and simplify request processing and policy enforcement** by centralizing the point of control and offloading non-functional requirements to the infrastructure layer
- **Simplify troubleshooting** with granular real-time and historical metrics and dashboards

API Gateway and Microservices Architecture

For microservices-based applications, an API gateway acts as a single point of entry into the system. It sits in front of the microservices and simplifies both the client implementations and the microservices app by decoupling the complexity of an app from its clients.

In a microservices architecture, the API gateway is responsible for request routing, composition, and policy enforcement. It handles some requests by simply routing them to the appropriate backend service, and handles others by invoking multiple backend services and aggregating the results.

An API gateway might provide other capabilities for microservices such as authentication, authorization, monitoring, load balancing, and response handling, offloading implementation of non-functional requirements to the infrastructure layer and helping developers to focus on core business logic, speeding up app releases.

Learn more about **Building Microservices Using an API Gateway** on our blog.

API Gateway for Kubernetes

Containers are the most efficient way to run microservices, and Kubernetes is the de facto standard for deploying and managing containerized applications and workloads.

Depending on the system architecture and app delivery requirements, an API gateway can be deployed in front of the Kubernetes cluster as a load balancer (multi-cluster level), at its edge as an Ingress controller (cluster-level), or within it as a service mesh (service-level).

For API gateway deployments at the edge and within the Kubernetes cluster, it's best practice to use a Kubernetes-native tool as the API gateway. Such tools are tightly integrated with the Kubernetes API, support YAML, and can be configured through standard Kubernetes CLI; examples include **NGINX Ingress Controller** and **NGINX Service Mesh**.

Learn more about API gateways and Kubernetes in **API Gateway vs. Ingress Controller vs. Service Mesh** on our blog.

API Gateway and Ingress Gateway or Ingress Controller

Ingress gateways and **Ingress controllers** are tools that implement the **Ingress object**, a part of the Kubernetes Ingress API, to expose applications running in Kubernetes to external clients. They manage communications between users and applications (user-to-service or north-south connectivity). However, the Ingress object by itself is very limited in its capabilities. For example, it does not support defining the security policies attached to it. As a result, many vendors create custom resource definitions (CRDs) to expand their Ingress controller's capabilities and satisfy evolving customer needs and requirements, including use of the Ingress controller as an API gateway.

For example, **NGINX Ingress Controller** can be used as a full-featured API gateway at the edge of a Kubernetes cluster with its **VirtualServer and VirtualServerRoute**, **TransportServer**, and **Policy** custom resources.

API Gateway Is Not the Same as Gateway API

While their names are similar, an API gateway is not the same as the **Kubernetes Gateway API**. The Kubernetes **Gateway API** is an open source project managed by the Kubernetes community to improve and standardize service networking in Kubernetes. The Gateway API specification evolved from the **Kubernetes Ingress API** to solve various challenges around deploying Ingress resources to expose Kubernetes apps in production, including the ability to define fine-grained policies for request processing and delegate control over configuration across multiple teams and roles.

Tools built on the Gateway API specification, such as **NGINX Kubernetes Gateway**, can be used as API gateways for use cases that include routing requests to specific microservices, implementing traffic policies, and enabling canary and blue-green deployments.

Watch this quick video where NGINX's Jenn Gile explains **the difference between an API gateway and the Kubernetes Gateway API**.

Service Mesh vs API Gateway

A **service mesh** is an infrastructure layer that controls communications across services in a Kubernetes cluster (service-to-service or east-west connectivity). The service mesh delivers core capabilities for services running in Kubernetes, including load balancing, authentication, authorization, access control, encryption, observability, and advanced patterns for managing connectivity (circuit braker, A/B testing, and blue-green and canary deployments), to ensure that communication is fast, reliable, and secure.

Deployed closer to the apps and services, a service mesh can be used as a lightweight, yet comprehensive, distributed API gateway for service-to-service communications in Kubernetes.

Learn more about service mesh in **How to Choose a Service Mesh** on our blog.

API Gateway and API Management

The terms *API gateway* and *API management* are often – but incorrectly – used to describe the same

functionality.

An **API gateway** is a **data-plane** entry point for API calls that represent client requests to target applications and services. It typically performs request processing based on defined policies, including authentication, authorization, access control, SSL/TLS offloading, routing, and load balancing.

API management is the process of deploying, documenting, operating, and monitoring individual APIs. It is typically accomplished with management-plane software (for example, an API manager) that defines and applies policies to API gateways and developer portals.

Depending on business and functional requirements, an API gateway can be deployed as a standalone component in the data plane, or as part of an integrated API management solution, such as **F5 NGINX Management Suite API Connectivity Manager**.

Considerations for Choosing an API Gateway

There are several key factors to consider when deciding on requirements for your API gateway:

- **Architecture** – Where you deploy the API gateway can impact your choice of tooling, as can the decision to use built-in options from your cloud provider. Do you need the flexibility of a platform and runtime agnostic API gateway?
- **Performance** – Performance is critical for high-traffic websites and applications. Does your API gateway deliver the high throughput and low latency you need?
- **Scalability** – Your API gateway needs to easily scale to meet increasing traffic demands. Does your API gateway support scaling vertically (high throughput) and horizontally (high availability) to ensure your APIs are always fast and available?
- **Security** – API gateways are an important part of a zero-trust architecture. Does your API gateway offer access control (AuthN/AuthZ), mTLS, and other advanced security features like an integrated WAF and OpenAPI schema validation for positive security?
- **Cost** – Understand the total cost of ownership (TCO) of the API gateway. What are the costs and tradeoffs of building and maintaining a custom solution vs purchasing an enterprise-grade API gateway?

How NGINX Can Help

NGINX offers several options for deploying and operating an API gateway depending on your use cases and deployment patterns.

Kubernetes-native tools:

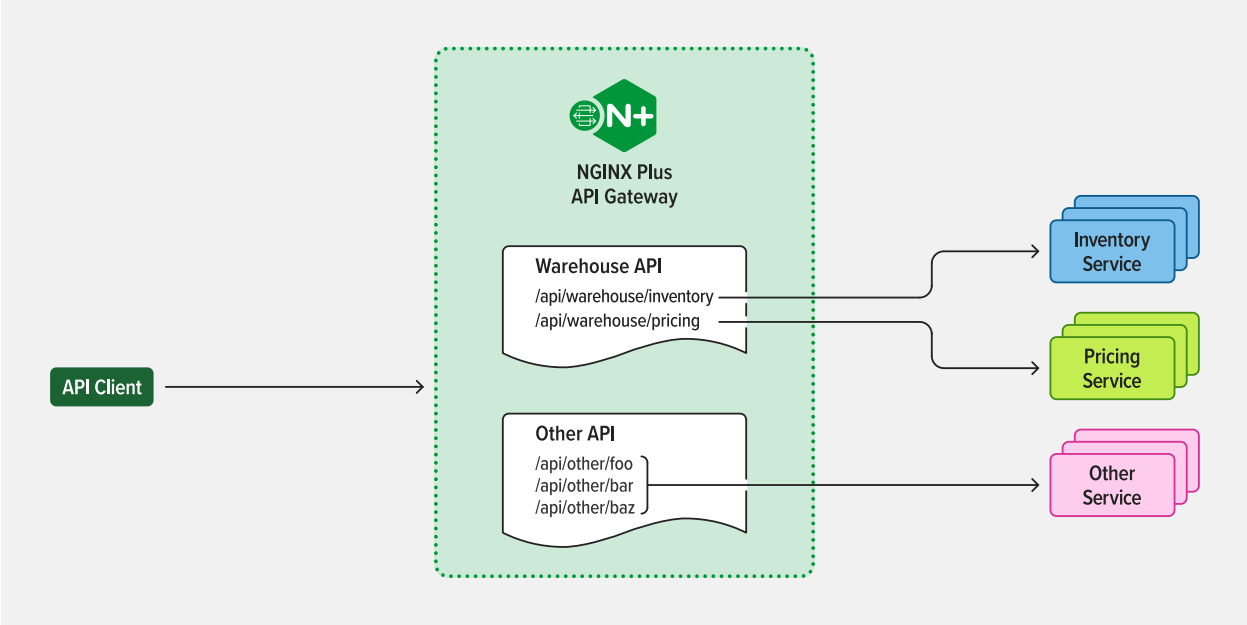
- **NGINX Ingress Controller** – Manages app connectivity at the edge of a Kubernetes cluster with API gateway, identity, and observability features
- **NGINX Service Mesh** – Developer-friendly solution for service-to-service connectivity, security, orchestration, and observability

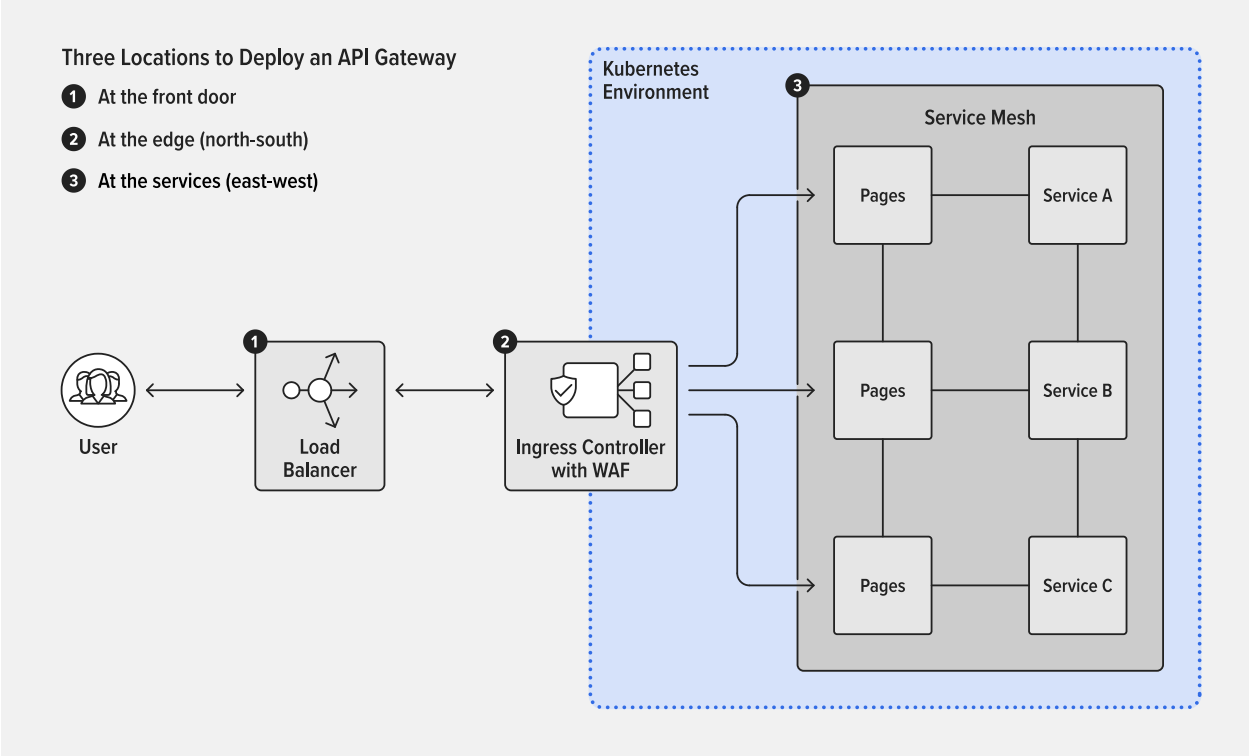
Get started by requesting your **free 30-day trial** of NGINX Ingress Controller with NGINX App Protect WAF and DoS, and **download** the always free NGINX Service Mesh.

Universal tools:

- **NGINX Plus** as an API gateway – Lightweight, high-performance API gateway that can be deployed across cloud, on-premises, and edge environments
- **F5 NGINX Management Suite API Connectivity Manager** – Deploy and operate API gateways with developer-friendly tools for API management, governance, and security

To learn more about using NGINX Plus as an API gateway, request your **free 30-day trial** and see **Deploying NGINX as an API Gateway** on our blog. To try NGINX Management Suite, request your **free 30-day trial**.

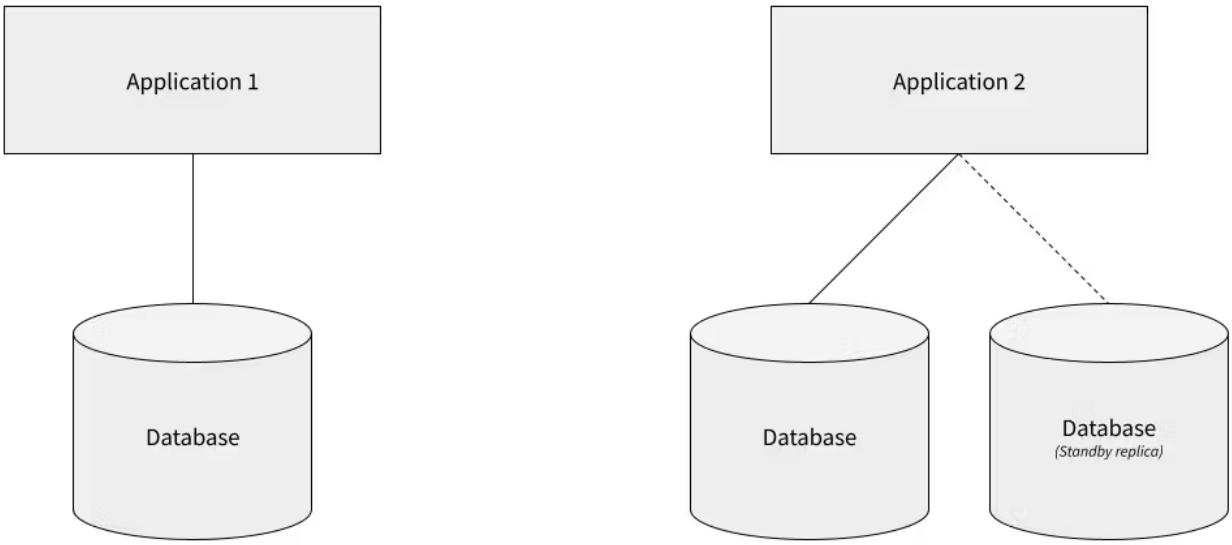




What is fault tolerance?

Fault tolerance describes a system's ability to handle errors and outages without any loss of functionality.

For example, here's a simple demonstration of comparative fault tolerance in the database layer. In the diagram below, **Application 1** is connected to a single database instance. **Application 2** is connected to two database instances — the primary database and a standby replica.



In this scenario, Application 2 is more fault tolerant. If its primary database goes offline, it can switch over to the standby replica and continue operating as usual.

Application 1 is not fault tolerant. If its database goes offline, all application features that require access to the database will cease to function.

Of course, this is just a simple example. In reality, fault tolerance must be considered in every layer of a system (not just the database), and there are degrees of fault tolerance. While Application 2 is more fault tolerant than Application 1, it's still less fault tolerant than many modern applications. (See examples of fault-tolerant application architecture.)

Fault tolerance can also be achieved in a variety of ways. These are some of the most common approaches to achieving fault tolerance:

Multiple hardware systems capable of doing the same work. For example, Application 2 in our diagram above could have its two databases located on two different physical servers, potentially in different locations. That way, if the primary database server experiences an error, a hardware failure, or a power outage, the other server might not be affected.

Multiple instances of software capable of doing the same work. For example, many modern applications make use of containerization platforms such as Kubernetes so that they can run multiple instances of software services. One reason for this is so that if one instance encounters an error or goes offline, traffic can be routed to other instances to maintain application functionality.

Backup sources of power, such as generators, are often used in on-premises systems to protect the application from being knocked offline if power to the servers is impacted by, for example, the weather. That type of outage is more common than you might expect.

Fault tolerance vs. high availability

High availability refers to a system's total uptime, and achieving high availability is one of the primary reasons architects look to build fault-tolerant systems.

Technically, fault tolerance and high availability are not exactly the same thing. Keeping an application highly available is not simply a matter of making it fault tolerant. A highly fault-tolerant application could still fail to achieve high availability if, for example, it has to be taken offline regularly to upgrade software components, change the database schema, etc.

In practice, however, the two are often closely connected, and it's difficult to achieve high availability without robust, fault-tolerant systems.

Fault tolerance goals

Building fault-tolerant systems is more complex and generally also more expensive. If we think back to our simple example from earlier, Application 2 is more fault tolerant, but it also has to pay for and maintain an additional database server. Thus, it's important to assess the level of fault tolerance your application requires and build your system accordingly.

Normal functioning vs. graceful degradation

When designing fault-tolerant systems, you may want the application to remain online and fully functional at all times. In this case, your goal is **normal functioning** — you want your application, and by extension the user's experience, to remain unchanged even if an element of your system fails or is knocked offline.

Another approach is aiming for what's called **graceful degradation,** where outages and errors are allowed to impact functionality and degrade the user experience, but not knock the application out entirely. For example, if a software instance encounters an error during a period of heavy traffic, the application experience may slow for other users, and certain features might become unavailable.

Building for normal functioning obviously provides for a superior user experience, but it's also generally more expensive. The goals for a specific application, then, might depend on what it's used for. Mission-critical applications and systems will likely need to maintain normal functioning in all but the most dire of disasters, whereas it might make economic sense to allow less essential systems to degrade gracefully.

Setting survival goals

Achieving 100% fault tolerance isn't really possible, so the question architects generally have to answer when designing fault-tolerant systems is how much they want to be able to survive.

Survival goals can vary, but here are some common ones for applications that run on one or more of the public clouds, in ascending order of resilience:

- **Survive node failure.** Running instances of your software on multiple nodes (often different physical servers) with the same AZ (data center) can allow your application to survive faults (such as hardware failures or errors) on one or more of those nodes.
- **Survive AZ failure.** Running instances of your software across multiple availability zones (data centers) within a cloud region will allow you to survive AZ outages, such as a specific data center losing power during a storm.
- **Survive region failure.** Running instances of your software across multiple cloud regions can allow you to survive an outage affecting an entire region, such as the AWS US-east-1 outage mentioned at the beginning of this post.
- **Survive cloud provider failure.** Running instances of your software both in the cloud and on-premises, or across multiple cloud providers, can allow you to survive even a full cloud provider outage.

These are not the only possible survival goals, of course, and fault tolerance is only one aspect of surviving outages and other disasters. Architects also need to consider factors such as RTO and RPO to minimize the negative impact when outages do occur. But considering your goals for fault tolerance is also important, as they will affect both the architecture of your application and its costs.

The cost of fault tolerance

When architecting fault-tolerant systems, another important consideration is cost. This is a difficult and very case-specific factor, but it's important to remember that while there are costs inherent with choosing and using more fault-tolerant architectures and tools, there are also significant costs associated with *not* choosing a high level of fault tolerance.

For example, operating multiple instances of your database across multiple cloud regions is likely to cost more on the balance sheet than operating a single instance in a single region. However, there are a few things you must also consider:

- **What does an outage cost in dollars?** For mission-critical systems, even a few minutes of downtime can lead to millions in lost revenue.
- **What does an outage cost in reputation damage?** Consumers are demanding, particularly in certain business verticals. An application outage of just a few minutes, for example, could be enough to scare millions of customers away from a bank.
- **What does an outage cost in engineering hours?** Any time your team spends recovering from an outage is time they're *not* spending building new features or doing other important work.
- **What does an outage cost in team morale and retention / hiring?** Outages also often come at inconvenient times. The US-east-1 outage, for example, came the day before Thanksgiving, when most US-based engineers were on vacation, forcing them to rush into the office on a holiday or in the middle of the night to deal with an emergency. Great engineers generally have a lot of choices when it comes to where they work, and will avoid working anywhere where those sorts of emergencies are likely to disrupt their lives.

These are just a few of the costs associated with *not* achieving a high level of fault tolerance.

The *way* you achieve fault tolerance can also have a significant impact on your costs. Here, let's consider the real-world case of a major electronics company that needed to build a more scalable, fault-tolerant version of its existing MySQL database.

The company could have made that MySQL database more fault tolerant by manually sharding it, but that approach is technically complex and requires a lot of work to execute and maintain. Instead, the company chose to migrate to CockroachDB dedicated, a managed database-as-a-service that is inherently distributed and fault tolerant.

Although CockroachDB dedicated itself is more expensive than MySQL (which is free), migrating to CockroachDB enabled the company to save millions in labor costs because it automates the labor-intensive manual sharding process and resolves many of the technical complexities that manually sharding would introduce.

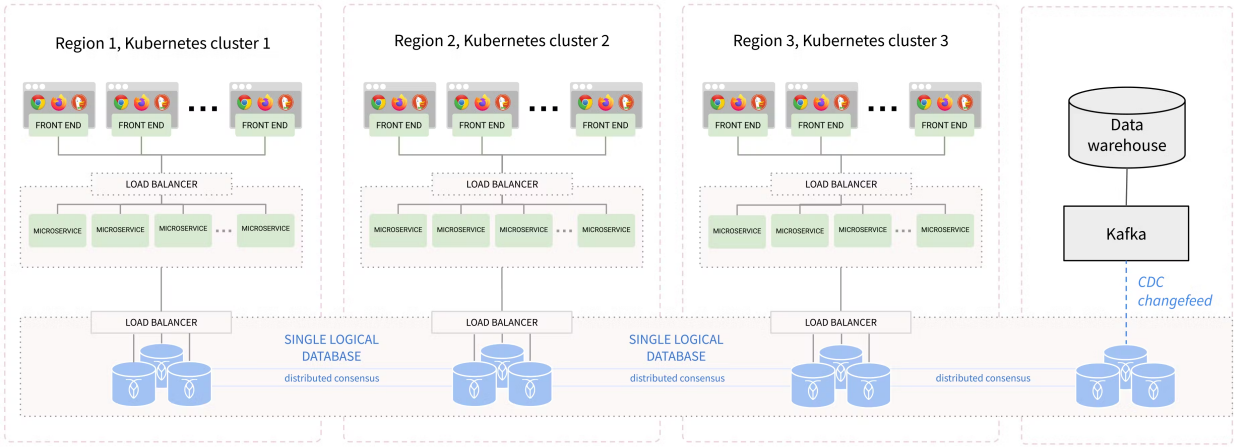
Ultimately, the company achieved a database that is as or *more* fault tolerant than manually sharded MySQL while spending millions of dollars less than what manually sharded MySQL would ultimately have cost them.

This's not to say that CockroachDB or *any* specific tool or platform will be the most affordable option for all use cases. However, it's important to recognize that the methods you choose for achieving your fault tolerance goals can have a significant impact on your costs in both the short and long term.

What does manually sharding legacy RDBMS really cost?

Fault-tolerant architecture examples

There are many ways to achieve fault tolerance, but let's take a look at a very common approach for modern applications: adopting a cloud-based, multi-region architecture built around containerization services such as Kubernetes.



An example of a fault-tolerant multi-region architecture. Click to enlarge.

This application could survive a node, AZ, or even region failure affecting its application layer, its database layer, or both. Let's take a closer look at how that's possible.

Achieving fault tolerance in the application layer

In the diagram above, the application is spread across multiple regions, with each region having its own Kubernetes cluster.

Within each region, the application is built with microservices that execute specific tasks, and these microservices are typically operated inside Kubernetes pods. This allows for much greater fault tolerance, since a new pod with a new instance can be started up whenever an existing pod encounters an error. This approach also makes the application easier to scale horizontally — as the load on a specific service increases, additional instances of that service can be added in real time to handle the load, and then removed when the load dies down again and they're no longer needed.

Message queuing and solving the dual-write problem in microservice architectures.

Achieving fault tolerance in the persistence (database) layer

The application in the diagram above takes a similar approach in the database layer. Here, CockroachDB is chosen because its distributed, node-based nature naturally provides a high level of fault tolerance and the same flexibility when it comes to scaling up and down horizontally. Being a distributed SQL database, it also allows for strong consistency guarantees, which is important for most transactional workloads.

CockroachDB also makes sense for this architecture because although it's a distributed database, it can be treated like a single-instance Postgres database by the application — almost all the complexity of distributing the data to meet your application's availability and survival goals happens under the hood.

What is a CDN?

A content delivery network (CDN) is a network of interconnected servers that speeds up webpage loading for data-heavy applications. CDN can stand for content delivery network or content distribution network. When a user visits a website, data from that website's server has to travel across the internet to reach the user's computer. If the user is located far from that server, it will take a long time to load a large file, such as a video or website image. Instead, the website content is stored on CDN servers geographically closer to the users and reaches their computers much faster.

Why is a CDN important?

The primary purpose of a content delivery network (CDN) is to reduce latency, or reduce the delay in communication created by a network's design. Because of the global and complex nature of the internet, communication traffic between websites (servers) and their users (clients) has to move over large physical distances. The communication is also two-way, with requests going from the client to the server and responses coming back.

A CDN improves efficiency by introducing intermediary servers between the client and the website server. These CDN servers manage some of the client-server communications. They decrease web traffic to the web server, reduce bandwidth consumption, and improve the user experience of your applications.

What are the benefits of CDNs?

Content delivery networks (CDNs) provide many benefits that improve website performance and support core network infrastructure. For example, a CDN can do the following tasks:

Reduce page load time

Website traffic can decrease if your page load times are too slow. A CDN can reduce bounce rates and increase the time users spend on your site.

Reduce bandwidth costs

Bandwidth costs are a significant expense because every incoming website request consumes network bandwidth. Through caching and other optimizations, CDNs can reduce the amount of data an origin server must provide, reducing the costs of hosting for website owners.

Increase content availability

Too many visitors at one time or network hardware failures can cause a website to crash. CDN services can handle more web traffic and reduce the load on web servers. Also, if one or more CDN servers go offline, other operational servers can replace them to ensure uninterrupted service.

Improve website security

Distributed denial-of-service (DDoS) attacks attempt to take down applications by sending large amounts of fake traffic to the website. CDNs can handle such traffic spikes by distributing the load between several intermediary servers, reducing the impact on the origin server.

What is the history of CDN technology?

Content delivery network (CDN) technology emerged in the late 1990s with the focus on faster content delivery over the internet:

First generation

First-generation CDN services focused on networking principles of intelligent network traffic management and data centers for replication.

Second generation

Second-generation CDNs arose in response to the rise of audio and video streaming services, especially video on demand and news on demand. The technology also evolved to solve new challenges in content delivery on mobile devices. Companies used cloud computing techniques and peer-to-peer networks to accelerate content delivery.

Third generation

Third-generation CDNs are still evolving. AWS is driving innovation as one of the leading CDN service providers in the world. With most web services centralized in the cloud, the focus is now on edge computing—managing bandwidth consumption using smart devices that communicate intelligently. Autonomous and self-managed edge networks might be the next step in CDN technology.

What internet content can a CDN deliver?

A content delivery network (CDN) can deliver two types of content: static content and dynamic content.

Static content

Static content is website data that does not change from user to user. Website header images, logos, and font styles remain the same across all users, and the business does not change them frequently. Static data does not need to be modified, processed, or generated and is ideal for storage on a CDN.

Dynamic content

Dynamic content such as social media news feeds, weather reports, login status, and chat messages vary among website users. This data changes based on the user's location, login time, or user preferences, and the website must generate the data for every user and every user interaction.

How does a CDN work?

Content delivery networks (CDNs) work by establishing a point of presence (POP) or a group of CDN edge servers at multiple geographical locations. This geographically distributed network works on the principles of caching, dynamic acceleration, and edge logic computations.

Caching

Caching is the process of storing multiple copies of the same data for faster data access. In computing, the principle of caching applies to all types of memory and storage management. In CDN technology, the term refers to the process of storing static website content on multiple servers in the network. Caching in CDN works as follows:

1. A geographically remote website visitor makes the first request for static web content from your site.
2. The request reaches your web application server or origin server. The origin server sends the response to the remote visitor. At the same time, it also sends a copy of the response to the CDN POP geographically closest to that visitor.
3. The CDN POP server stores the copy as a cached file.
4. The next time this visitor, or any other visitor in that location, makes the same request, the caching server, not the origin server, sends the response.

Dynamic acceleration

Dynamic acceleration is the reduction in server response time for dynamic web content requests because of an intermediary CDN server between the web applications and the client. Caching doesn't work well with dynamic web content because the content can change with every user request. CDN servers have to reconnect with the origin server for every dynamic request, but they accelerate the process by optimizing the connection between themselves and the origin servers.

If the client sends a dynamic request directly to the web server over the internet, the request might get lost or delayed due to network latency. Time might also be spent opening and closing the connection for security verification. On the other hand, if the nearby CDN server forwards the request to the origin server, they would already have an ongoing, trusted connection established. For example, the following features could further optimize the connection between them:

- Intelligent routing algorithms
- Geographic proximity to the origin
- The ability to process the client request, which reduces its size

Edge logic computations

You can program the CDN edge server to perform logical computations that simplify communication between the client and server. For example, this server can do the following:

- Inspect user requests and modify caching behavior.
- Validate and handle incorrect user requests.
- Modify or optimize content before responding.

Distribution of application logic between the web servers and the network edge helps developers offload origin servers' compute requirements and improve website performance.

What is a CDN used for?

A content delivery network (CDN) improves normal website functions and increases customer satisfaction. The following are some example use cases.

High-speed content delivery

By combining static and dynamic internet content delivery, you can use CDNs to provide your customers with a global, high-performing, whole-site experience. For example, Reuters is the world's largest news wholesaler to top channels such as BBC, CNN, the New York Times, and the Washington Post. The news media challenge for Reuters is to deliver news content promptly to customers around the globe. Reuters uses Amazon's CDN service, Amazon CloudFront, with Amazon Simple Storage Service (Amazon S3) to minimize dependence on satellite link communication and create a cheaper, highly available, and secure globally distributed network platform.

Real-time streaming

CDNs help reliably and cost-effectively deliver rich and high-quality media files. Companies streaming video and audio use CDNs to overcome three challenges: reduce bandwidth costs, increase scale, and decrease delivery time. For example, Hulu is an online video streaming platform owned by the Walt Disney Company. It uses Amazon CloudFront to consistently stream more than 20 GBps of data to its growing customer base.

Multi-user scaling

CDNs help support a large number of concurrent users. Website resources can manage only a limited number of client connections at a time. CDNs can rapidly scale this number by taking some of the load from the application server. For instance, King is a gaming company that builds socially connected, cross-platform games that can be played anytime, anywhere, and from any device. King has over 350 million players at any time, and they play 10.6 billion games a day on the platform.

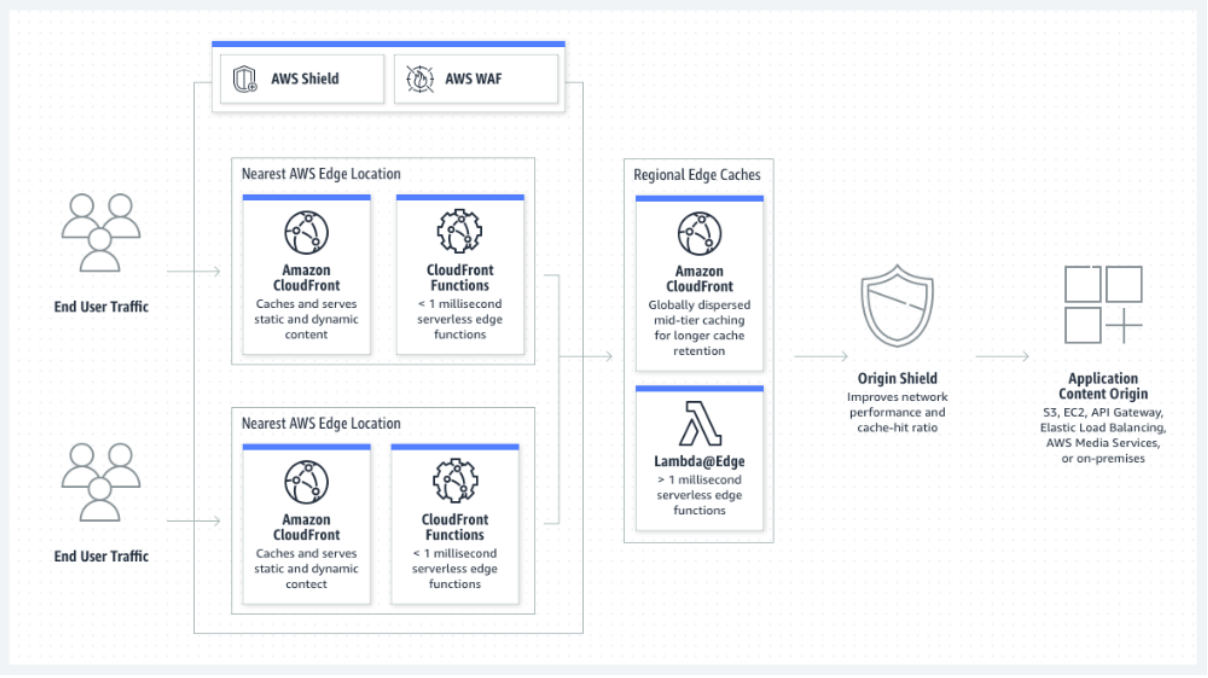
King's game applications record users' game data on central data centers, allowing them to play on different devices without losing progress. The data centers aim to give users a consistent experience, even if users join the game on old machines with limited bandwidth.

King uses Amazon CloudFront to deliver hundreds of terabytes of content daily, with spikes to half a petabyte or more when it launches a new game or initiates a large-scale marketing program.

What is Amazon CloudFront?

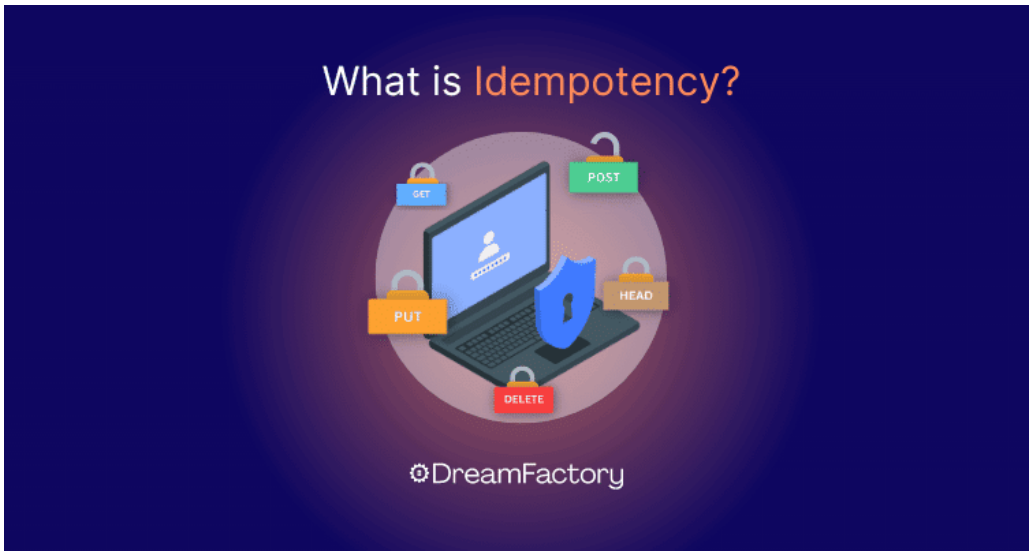
Amazon CloudFront is a content delivery network (CDN) service built for high performance, security, and developer convenience. You can use Amazon CloudFront to do these tasks:

- Deliver data through 450+ globally dispersed points of presence (POPs) with automated network mapping and intelligent routing.
- Improve security with traffic encryption and access controls, and use AWS Shield Standard to defend against distributed denial-of-service (DDoS) attacks at no additional charge.
- Customize the code you run at the AWS network edge using serverless compute features to balance cost, performance, and security.
- Scale automatically to deliver software, game patches, and IoT updates with high transfer rates.



What is Idempotency?

Terence Bennett - August 22, 2023



Idempotent operations produce the same result even when the operation is repeated many times. The result of the 2nd, 3rd, and 1,000th repeat of the operation will return exactly the same result as the 1st time. In this post, we will demystify the concept of idempotency—a fundamental property that ensures consistency, predictability, and reliability in APIs and distributed systems. Join us as we unravel the importance of idempotency, its implications, and best practices to implement it in your own systems, paving the way for more robust and user-friendly applications.

Here's the main facts you need to know about idempotency:

- **Idempotency is a property of operations or API requests that ensures repeating the operation multiple times produces the same result as executing it once.**
- **Safe methods are idempotent but not all idempotent methods are safe.**
- **HTTP methods like GET, HEAD, PUT, DELETE, OPTIONS, and TRACE are idempotent, while POST and PATCH are generally non-idempotent.**
- **Understanding and leveraging the idempotent nature of HTTP methods helps create more consistent, reliable, and predictable web applications and APIs.**
- **Most HTTP methods used in REST APIs are idempotent, except for POST, and following REST principles can ensure proper usage of idempotent methods.**

Table of Contents:

Need an API? Did you know you can generate a full-featured, documented, and secure REST API in minutes using DreamFactory? Sign up for our **free 14 day hosted trial** to learn how! Our guided tour will show you how to create an API using an example MySQL database provided to you as part of the trial!

Create Your REST API Now

What is Idempotency?

Idempotency is a property of certain operations or API requests, which guarantees that performing the operation multiple times will yield the same result as if it was executed only once. This principle is especially vital in **Distributed Systems** and APIs, as it helps maintain consistency and predictability in situations such as network issues, request retries, or duplicated requests.

Idempotency is a crucial property of certain operations or API requests that guarantees consistent outcomes, regardless of the number of times an operation is performed. This principle simplifies error handling, concurrency management, debugging, and monitoring, while also enhancing the overall user experience.

For example, simple mathematical examples of idempotency include:

x + 0;

x = 5;

In the first example, adding zero will never change the result, regardless of how many times you do it. In the second, x is always 5. Again, this is the case, regardless of how many times you perform the operation. Both of these examples describe an operation that is idempotent.

Why is Idempotency Important?

Idempotency is important in **APIs** because a resource may be called multiple times if the network is interrupted. In this scenario, non-idempotent operations can cause significant unintended side-effects by creating additional resources or changing them unexpectedly. When a business relies on the accuracy of its data, non-idempotency posts a significant risk.

Imagine a scenario where an API facilitates monetary transactions between accounts. A user sends a request to transfer $100 from Account A to Account B. Due to network latency or other factors, the request is unintentionally duplicated. Without idempotency, the API would process both requests, resulting in an unintended transfer of $200. However, if the operation is idempotent, only one transfer of $100 will occur, regardless of the number of duplicate requests.

Idempotency is crucial for ensuring the following:

1. Consistency: The system maintains a predictable state, even when faced with request duplication or retries.
2. Error Handling: Idempotent operations simplify error handling, as clients can safely retry requests without the risk of unintended side effects.
3. Fault Tolerance: Idempotent APIs can better cope with network issues and other disruptions, ensuring a more reliable user experience.

Is hand-coding APIs taking up too much of your development time? DreamFactory enables non-coders to create REST APIs in minutes. **Start Your Free 14-Day Hosted DreamFactory Trial Today**.

Idempotent vs. Safe

The concepts of ‘idempotent methods’ and ‘safe methods’ are often confused. A safe method does not change the value that is returned, it reads – but it never writes. Going back to our previous examples:

x + 0;

x = 5;

The first of these, adding zero, will return the same value every time (it is idempotent), and adding zero will have no effect on that value (it is also safe). The second example will return the same value every time (it is idempotent) but is not safe (if x is anything other than 5 before the operation runs, it changes x).

Therefore, all safe methods are idempotent, but not all idempotent methods are safe.

Idempotent Methods in REST

REST APIs use HTTP methods such as POST, PUT, and GET to interact with resources such as an image, customer name, or document. When using an idempotent method, the method can be called multiple times without changing the result.

For example, using GET, an API can retrieve a REST resource. This has no effect on the resource and can be repeated over and over – it is both idempotent and safe.

HTTP Methods

HTTP methods, or verbs, are the actions used to interact with resources in web-based systems, defining the type of operation a client wants to perform. Idempotency is an important property of some HTTP methods, ensuring that executing the same request multiple times produces the same result as if it was performed only once. GET, HEAD, PUT, DELETE, OPTIONS, and TRACE are idempotent methods, meaning they are safe to be retried or executed multiple times without causing unintended side effects. In contrast, POST and PATCH are generally considered non-idempotent, as their outcomes may vary with each request. Understanding and leveraging the idempotent nature of specific HTTP methods helps developers create more consistent, reliable, and predictable web applications and APIs.

Here are the HTTP methods and whether each one is idempotent or not.

POST

POST is an HTTP (Hypertext Transfer Protocol) method used to submit data to a specified resource, typically for creating or updating content on a server. It is often employed when submitting form data, uploading files, or sending JSON payloads to an API. Unlike other HTTP methods like GET, PUT, or DELETE, POST is not inherently idempotent. This means that sending the same POST request multiple times can result in different outcomes or create multiple instances of the submitted data.

Consequently, developers need to be cautious when handling POST requests, implementing mechanisms such as idempotency keys or other safeguards to prevent unintended side effects in cases where duplicate requests are submitted or retried.

GET

The GET method is one of the fundamental HTTP methods used to request data from a specific resource on the web. When a client sends a GET request to a server, it is asking the server to retrieve and return the information associated with the specified resource, such as a webpage, image, or document. GET requests are designed to be read-only, meaning they should not modify the state of the resource or cause any side effects on the server.

Due to their read-only nature, GET requests are considered idempotent. This means that making multiple identical GET requests to the same resource will yield the same result without causing any additional changes, ensuring consistency and predictability in **Web-Based Applications And Services**.

HEAD

The HEAD method in HTTP is a request method used to retrieve metadata about a specific resource without actually downloading the resource's content. Similar to the GET method, the HEAD method is used to request information about a resource, but it only returns the HTTP headers, which include information such as content

type, content length, and last modification date. This method is particularly useful when clients want to check if a resource exists or retrieve its metadata without incurring the cost of downloading the entire resource.

The HEAD method is idempotent, meaning that multiple requests using HEAD on the same resource will yield the same outcome without any side effects. Consequently, it is safe to perform HEAD requests multiple times without affecting the state of the resource or the server.

PUT

PUT is an HTTP method used to update or replace an existing resource on the server with new data provided by the client. The request includes a unique identifier, such as a URI, which the server uses to locate the target resource. The client also provides a payload, containing the updated data to be applied to the resource. As an HTTP method, PUT is inherently idempotent. This means that making multiple identical PUT requests will have the same effect as making a single request. If a resource already exists at the specified URI, the server will replace it with the new data. If the resource does not exist, the server may create it, depending on the implementation.

The idempotent nature of PUT ensures consistency and predictability in the face of network issues or duplicate requests, making it well-suited for updating resources in a distributed system.

PATCH

PATCH is an HTTP method used for partially updating resources on a server. Unlike PUT, which typically requires the client to send a complete representation of the resource, PATCH allows clients to send only the changes that need to be applied. This makes it more efficient for updating specific attributes or elements of a resource without affecting the rest of the data. The idempotency of PATCH depends on the implementation and the semantics of the operation.

While PATCH can be idempotent if the changes are applied in a consistent and deterministic manner, it is not inherently idempotent like GET, PUT, and DELETE. In practice, PATCH can be made idempotent by carefully designing the API and the underlying operations, ensuring that repeated PATCH requests yield the same outcome as a single request.

DELETE

DELETE is an HTTP method used to remove a specified resource from a server. It is part of the HTTP/1.1 standard and allows clients to request the deletion of a resource identified by a specific Uniform Resource Identifier (URI). When a DELETE request is successfully processed, the server typically responds with a 204 No Content status code, indicating that the resource has been removed. The DELETE method is considered idempotent, as performing the same request multiple times results in the same outcome. After the initial successful deletion, any subsequent DELETE requests to the same URI should have no additional effect on the server's state, since the resource is no longer present.

This idempotent nature ensures consistency and predictability, making the DELETE method a reliable choice for managing resource deletion in APIs and web services.

TRACE

TRACE is an HTTP method that allows clients to retrieve a diagnostic representation of the request and response messages for a specific resource on the server. When a client sends a TRACE request, the server returns the entire HTTP request, including headers, as the response body. This method is primarily used for debugging purposes, enabling developers to examine and diagnose potential issues in the request and response cycle.

TRACE is considered idempotent because, by design, it does not alter the resource or its state on the server. When issuing multiple TRACE requests, the server will consistently return the same request data, making it a

safe and predictable operation. However, due to security concerns, TRACE is often disabled on web servers to prevent potential information leaks.

As can be seen from this information, most of the methods used in a REST API are idempotent. The exception is POST. However, it is worth remembering that if used incorrectly, other methods like GET and HEAD can still be **Called In A Non-Idempotent Way**. Developers can avoid this by following REST principles.

Need an API? Did you know you can generate a full-featured, documented, and secure REST API in minutes using DreamFactory? Sign up for our **free 14 day hosted trial** to learn how! Our guided tour will show you how to create an API using an example MySQL database provided to you as part of the trial!

Create Your REST API Now

Getting Started with DreamFactory

DreamFactory is a full-service SaaS platform that helps with API management at all stages of the API life cycle. We can help you create an API, convert and update **Existing Web Services**, or maintain your existing APIs. There is a library of resources made available by our API experts. DreamFactory's no-code platform is the perfect starting point if your considering how APIs might contribute to enhancing better organizational agility within your business or team.

Sign Up For A 14-Day Free Trial and start creating your APIs today!

Frequently Asked Questions: Idempotentcy

What is idempotency?

Idempotency is a property of certain operations or API requests that ensures performing the operation multiple times yields the same result as if it were executed only once. This principle is essential in distributed systems and APIs to maintain consistency and predictability in scenarios like network issues, request retries, or duplicated requests.

Why is idempotency important in APIs?

Idempotency is crucial in APIs because it guarantees consistent outcomes even if a resource is called multiple times due to network interruptions or request duplications. Non-idempotent operations can lead to unintended side-effects like creating additional resources or unexpected changes, posing a significant risk when **Data Accuracy Is Crucial.**

Can you provide an example of idempotency in practice?

Consider an API for monetary transactions. If a user requests to transfer $100 from Account A to Account B, and due to network issues, the request is duplicated, idempotency ensures that only one $100 transfer occurs, preventing the unintended transfer of $200.

What benefits does idempotency offer?

Idempotency simplifies error handling, concurrency management, debugging, and monitoring in operations and **APIs**. It enhances user experience by maintaining consistency and predictability, even in the face of network disruptions and request retries.

What's the difference between idempotency and safety?

Idempotency ensures that repeated operations yield the same outcome. Safety, on the other hand, refers to operations that don't modify the returned value. While all safe methods are idempotent, not all idempotent methods are safe.

Which HTTP methods are idempotent?

HTTP methods such as GET, HEAD, PUT, DELETE, OPTIONS, and TRACE are considered idempotent. They can be retried or executed multiple times without causing unintended side effects.

Is POST an idempotent method?

No, POST is not inherently idempotent. Making the same POST request multiple times can result in different outcomes or create multiple instances of submitted data.

What Is Hashing?

What is "hashing" all about? Merriam-Webster defines the noun *hash* as "chopped meat mixed with potatoes and browned," and the verb as "to chop (as meat and potatoes) into small pieces." So, culinary details aside, hash roughly means "chop and mix"—and that's precisely where the technical term comes from.

A hash function is a function that maps one piece of data—typically describing some kind of object, often of arbitrary size—to another piece of data, typically an integer, known as *hash code*, or simply *hash*.

For instance, some hash function designed to hash strings, with an output range of `0 .. 100`, may map the string `Hello` to, say, the number `57`, `Hasta la vista, baby` to the number `33`, and any other possible string to some number within that range. Since there are way more possible inputs than outputs, any given number will have many different strings mapped to it, a phenomenon known as *collision*. Good hash functions should somehow "chop and mix" (hence the term) the input data in such a way that the outputs for different input values are spread as evenly as possible over the output range.

Hash functions have many uses and for each one, different properties may be desired. There is a type of hash function known as *cryptographic hash functions*, which must meet a restrictive set of properties and are used for security purposes—including applications such as password protection, integrity checking and fingerprinting of messages, and data corruption detection, among others, but those are outside the scope of this article.

Non-cryptographic hash functions have several uses as well, the most common being their use in *hash tables*, which is the one that concerns us and which we'll explore in more detail.

Introducing Hash Tables (Hash Maps)

Imagine we needed to keep a list of all the members of some club while being able to search for any specific member. We could handle it by keeping the list in an array (or linked list) and, to perform a search, iterate the elements until we find the desired one (we might be searching based on their name, for instance). In the worst case, that would mean checking all members (if the one we're searching for is last, or not present at all), or half of them on average. In complexity theory terms, the search would then have complexity `O(n)`, and it would be reasonably fast for a small list, but it would get slower and slower in direct proportion to the number of members.

How could that be improved? Let's suppose all these club members had a member `ID`, which happened to be a sequential number reflecting the order in which they joined the club.

Assuming that searching by `ID` were acceptable, we could place all members in an array, with their indexes matching their `ID`s (for example, a member with `ID=10` would be at the index `10` in the array). This would allow us to access each member directly, with no search at all. That would be very efficient, in fact, as efficient as it can possibly be, corresponding to the lowest complexity possible, `O(1)`, also known as *constant time*.

But, admittedly, our club member `ID` scenario is somewhat contrived. What if `ID`s were big, non-sequential or random numbers? Or, if searching by `ID` were not acceptable, and we needed to search by name (or some other field) instead? It would certainly be useful to keep our fast direct access (or something close) while at the same time being able to handle arbitrary datasets and less restrictive search criteria.

Here's where hash functions come to the rescue. A suitable hash function can be used to map an arbitrary piece of data to an integer, which will play a similar role to that of our club member `ID`, albeit with a few important differences.

First, a good hash function generally has a wide output range (typically, the whole range of a 32 or 64-bit integer), so building an array for all possible indices would be either impractical or plain impossible, and a colossal waste of memory. To overcome that, we can have a reasonably sized array (say, just twice the number of elements we expect to store) and perform a *modulo* operation on the hash to get the array index. So, the index would be `index = hash(object) mod N`, where `N` is the size of the array.

Second, object hashes will not be unique (unless we're working with a fixed dataset and a custom-built perfect hash function, but we won't discuss that here). There will be *collisions* (further increased by the modulo operation), and therefore a simple direct index access won't work. There are several ways to handle this, but a typical one is to attach a list, commonly known as a *bucket*, to each array index to hold all the objects sharing a given index.

So, we have an array of size `N`, with each entry pointing to an object bucket. To add a new object, we need to calculate its `hash modulo N`, and check the bucket at the resulting index, adding the object if it's not already there. To search for an object, we do the same, just looking into the bucket to check if the object is there. Such a structure is called a *hash table*, and although the searches within buckets are linear, a properly sized hash table should have a reasonably small number of objects per bucket, resulting in *almost* constant time access (an average complexity of `O(N/k)`, where `k` is the number of buckets).

With complex objects, the hash function is typically not applied to the whole object, but to a *key* instead. In our club member example, each object might contain several fields (like name, age, address, email, phone), but we could pick, say, the email to act as the key so that the hash function would be applied to the email only. In fact, the key need not be part of the object; it is common to store key/value pairs, where the key is usually a relatively short string, and the value can be an arbitrary piece of data. In such cases, the hash table or hash map is used as a *dictionary*, and that's the way some high-level languages implement objects or associative arrays.

Scaling Out: Distributed Hashing

Now that we have discussed hashing, we're ready to look into *distributed hashing*.

In some situations, it may be necessary or desirable to split a hash table into several parts, hosted by different servers. One of the main motivations for this is to bypass the memory limitations of using a single computer, allowing for the construction of arbitrarily large hash tables (given enough servers).

In such a scenario, the objects (and their keys) are *distributed* among several servers, hence the name.

A typical use case for this is the implementation of in-memory caches, such as Memcached.

Such setups consist of a pool of caching servers that host many key/value pairs and are used to provide fast access to data originally stored (or computed) elsewhere. For example, to reduce the load on a database server and at the same time improve performance, an application can be designed to first fetch data from the cache servers, and only if it's not present there—a situation known as *cache miss*—resort to the database, running the relevant query and caching the results with an appropriate key, so that it can be found next time it's needed.

Now, how does distribution take place? What criteria are used to determine which keys to host in which servers?

The simplest way is to take the hash *modulo* of the number of servers. That is, `server = hash(key) mod N`, where `N` is the size of the pool. To store or retrieve a key, the client first computes the hash, applies a `modulo N` operation, and uses the resulting index to contact the appropriate server (probably by using a lookup table of IP addresses). Note that the hash function used for key distribution must be the same one across all clients, but it need not be the same one used internally by the caching servers.

Let's see an example. Say we have three servers, `A`, `B` and `C`, and we have some string keys with their hashes:

KEY	HASH	HASH mod 3
"john"	1633428562	2
"bill"	7594634739	0
"jane"	5000799124	1
"steve"	9787173343	0
"kate"	3421657995	2

A client wants to retrieve the value for key `john`. Its `hash modulo 3` is `2`, so it must contact server `C`. The key is not found there, so the client fetches the data from the source and adds it. The pool looks like this:

A	B	C
		"john"

Next another client (or the same one) wants to retrieve the value for key `bill`. Its `hash modulo 3` is `0`, so it must contact server `A`. The key is not found there, so the client fetches the data from the source and adds it. The pool looks like this now:

A	B	C
"bill"		"john"

After the remaining keys are added, the pool looks like this:

A	B	C
"bill"	"jane"	"john"
"steve"		"kate"

The Rehashing Problem

This distribution scheme is simple, intuitive, and works fine. That is, until the number of servers changes. What happens if one of the servers crashes or becomes unavailable? Keys need to be redistributed to account for the missing server, of course. The same applies if one or more new servers are added to the pool;keys need to be redistributed to include the new servers. This is true for any distribution scheme, but the problem with our simple modulo distribution is that when the number of servers changes, most `hashes modulo N` will change, so most keys will need to be moved to a different server. So, even if a single server is removed or added, all keys will likely need to be rehashed into a different server.

From our previous example, if we removed server `C`, we'd have to rehash all the keys using `hash modulo 2` instead of `hash modulo 3`, and the new locations for the keys would become:

KEY	HASH	HASH mod 2
"john"	1633428562	0
"bill"	7594634739	1
"jane"	5000799124	0
"steve"	9787173343	1
"kate"	3421657995	1

A	B

"john"	"bill"
"jane"	"steve"
	"kate"

Note that all key locations changed, not only the ones from server `C`.

In the typical use case we mentioned before (caching), this would mean that, all of a sudden, the keys won't be found because they won't yet be present at their new location.

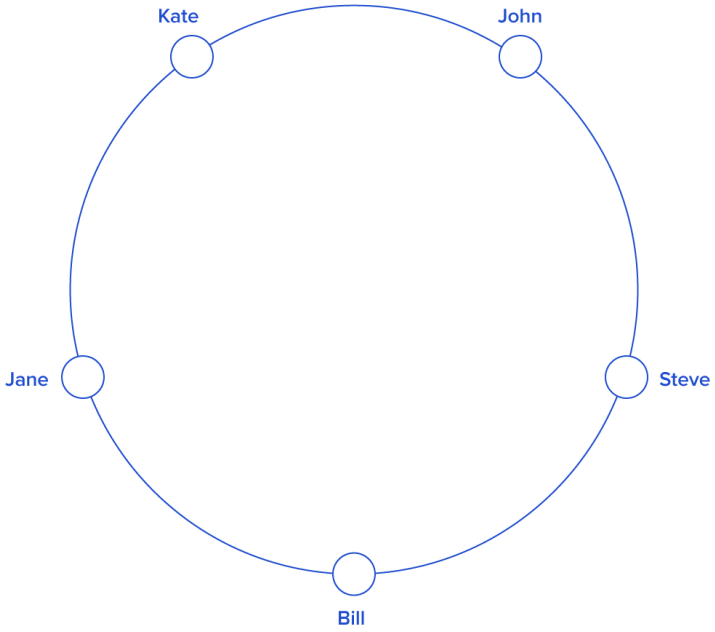
So, most queries will result in misses, and the original data will likely need retrieving again from the source to be rehashed, thus placing a heavy load on the origin server(s) (typically a database). This may very well degrade performance severely and possibly crash the origin servers.

The Solution: Consistent Hashing

So, how can this problem be solved? We need a distribution scheme that does *not* depend directly on the number of servers, so that, when adding or removing servers, the number of keys that need to be relocated is minimized. One such scheme—a clever, yet surprisingly simple one—is called *consistent hashing*, and was first described by Karger et al. at MIT in an academic paper from 1997 (according to Wikipedia).

Consistent Hashing is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed *hash table* by assigning them a position on an abstract circle, or *hash ring*. This allows servers and objects to scale without affecting the overall system.

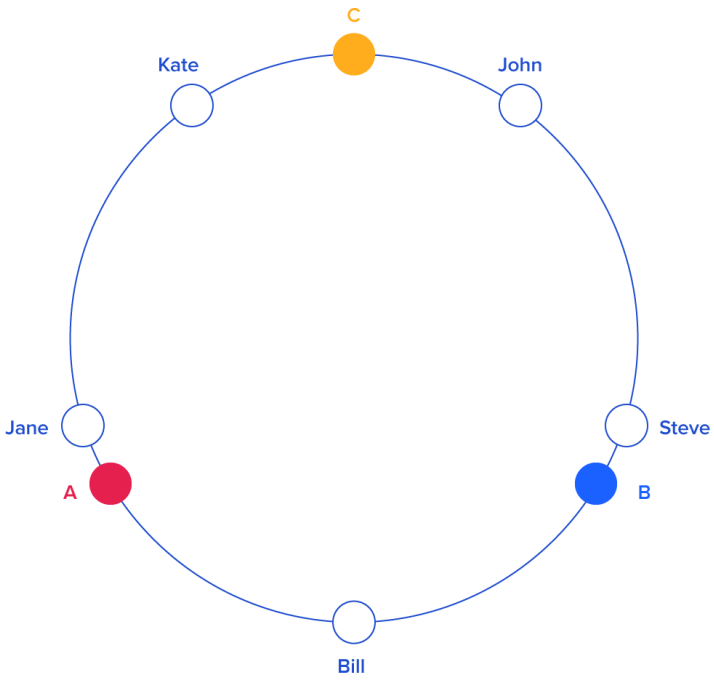
Imagine we mapped the hash output range onto the edge of a circle. That means that the minimum possible hash value, zero, would correspond to an angle of zero, the maximum possible value (some big integer we'll call `INT_MAX`) would correspond to an angle of 2π radians (or 360 degrees), and all other hash values would linearly fit somewhere in between. So, we could take a key, compute its hash, and find out where it lies on the circle's edge. Assuming an `INT_MAX` of 1010 (for example's sake), the keys from our previous example would look like this:



KEY	HASH	ANGLE (DEG)
"john"	1633428562	58.8
"bill"	7594634739	273.4
"jane"	5000799124	180
"steve"	9787173343	352.3
"kate"	3421657995	123.2

Now imagine we also placed the servers on the edge of the circle, by pseudo-randomly assigning them angles too. This should be done in a repeatable way (or at least in such a way that all clients agree on the servers' angles). A convenient way of doing this is by hashing the server name (or IP address, or some ID)—as we'd do with any other key—to come up with its angle.

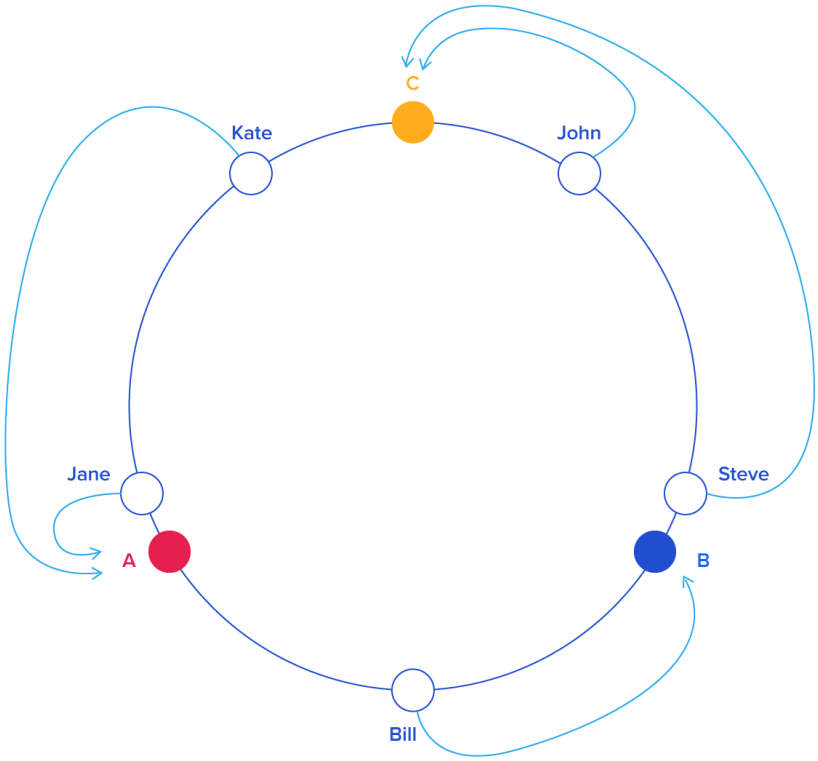
In our example, things might look like this:



KEY	HASH	ANGLE (DEG)
"john"	1633428562	58.8
"bill"	7594634739	273.4
"jane"	5000799124	180
"steve"	9787173343	352.3
"kate"	3421657995	123.2
"A"	5572014558	200.6
"B"	8077113362	290.8
"C"	2269549488	81.7

Since we have the keys for both the objects and the servers on the same circle, we may define a simple rule to associate the former with the latter: Each object key will belong in the server whose key is closest, in a counterclockwise direction (or clockwise, depending on the conventions used). In other words, to find out which server to ask for a given key, we need to locate the key on the circle and move in the ascending angle direction until we find a server.

In our example:



KEY	HASH	ANGLE (DEG)
"john"	1633428562	58.7
"C"	2269549488	81.7
"kate"	3421657995	123.1
"jane"	5000799124	180
"A"	5572014557	200.5
"bill"	7594634739	273.4
"B"	8077113361	290.7
"steve"	787173343	352.3

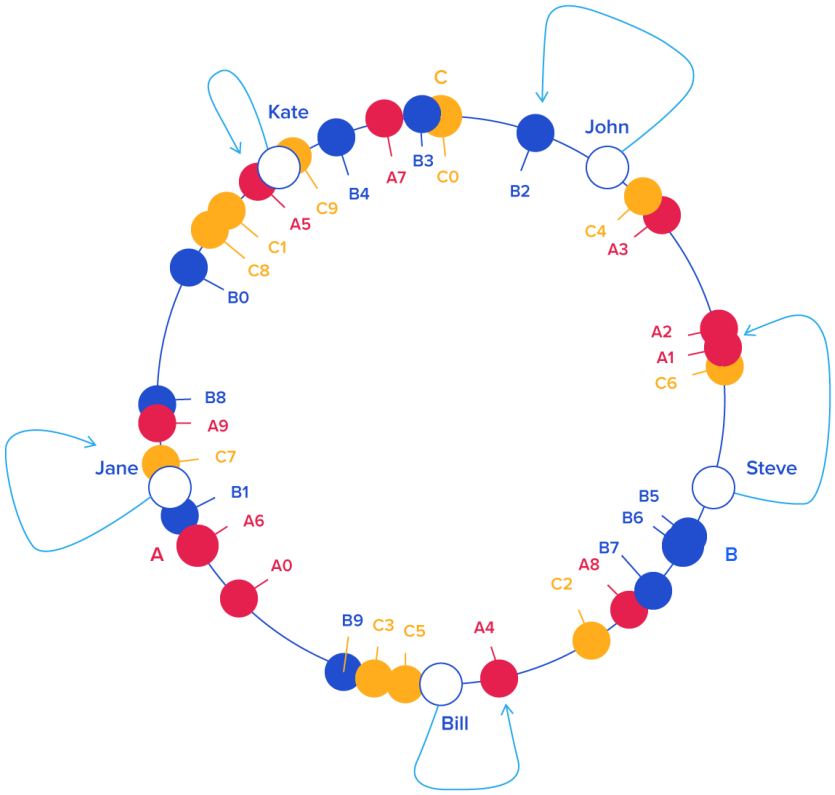
KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"C"	C
"kate"	3421831276	123.1	"A"	A
"jane"	5000648311	180	"A"	A
"bill"	7594873884	273.4	"B"	B
"steve"	9786437450	352.3	"C"	C

From a programming perspective, what we would do is keep a sorted list of server values (which could be angles or numbers in any real interval), and walk this list (or use a binary search) to find the first server with a value

greater than, or equal to, that of the desired key. If no such value is found, we need to wrap around, taking the first one from the list.

To ensure object keys are evenly distributed among servers, we need to apply a simple trick: To assign not one, but many labels (angles) to each server. So instead of having labels `A`, `B` and `C`, we could have, say, `A0 .. A9`, `B0 .. B9` and `C0 .. C9`, all interspersed along the circle. The factor by which to increase the number of labels (server keys), known as *weight*, depends on the situation (and may even be different for each server) to adjust the probability of keys ending up on each. For example, if server `B` were twice as powerful as the rest, it could be assigned twice as many labels, and as a result, it would end up holding twice as many objects (on average).

For our example we'll assume all three servers have an equal weight of 10 (this works well for three servers, for 10 to 50 servers, a weight in the range 100 to 500 would work better, and bigger pools may need even higher weights):



KEY	HASH	ANGLE (DEG)
"C6"	408965526	14.7
"A1"	473914830	17
"A2"	548798874	19.7
"A3"	1466730567	52.8
"C4"	1493080938	53.7
"john"	1633428562	58.7

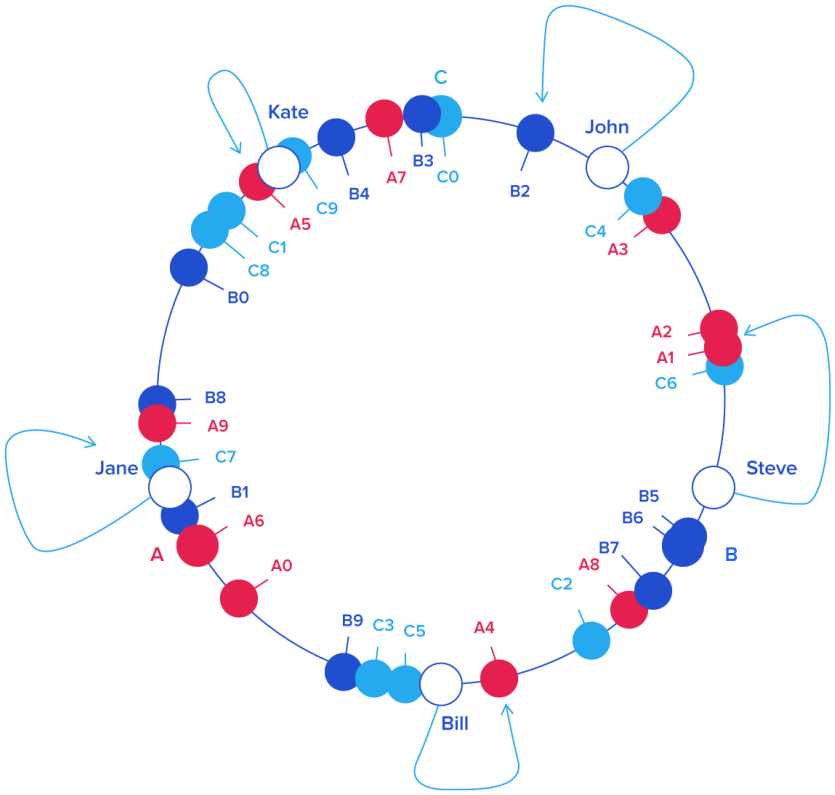
"B2"	1808009038	65
"C0"	1982701318	71.3
"B3"	2058758486	74.1
"A7"	2162578920	77.8
"B4"	2660265921	95.7
"C9"	3359725419	120.9
"kate"	3421657995	123.1
"A5"	3434972143	123.6
"C1"	3672205973	132.1
"C8"	3750588567	135
"B0"	4049028775	145.7
"B8"	4755525684	171.1
"A9"	4769549830	171.7
"jane"	5000799124	180
"C7"	5014097839	180.5
"B1"	5444659173	196
"A6"	6210502707	223.5
"A0"	6511384141	234.4
"B9"	7292819872	262.5
"C3"	7330467663	263.8
"C5"	7502566333	270
"bill"	7594634739	273.4
"A4"	8047401090	289.7
"C2"	8605012288	309.7
"A8"	8997397092	323.9
"B7"	9038880553	325.3
"B5"	9368225254	337.2
"B6"	9379713761	337.6
"steve"	9787173343	352.3

KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"B2"	B
"kate"	3421831276	123.1	"A5"	A
"jane"	5000648311	180	"C7"	C
"bill"	7594873884	273.4	"A4"	A
"steve"	9786437450	352.3	"C6"	C

So, what's the benefit of all this circle approach? Imagine server `C` is removed. To account for this, we must remove labels `C0 .. C9` from the circle. This results in the object keys formerly adjacent to the deleted labels now being randomly labeled `Ax` and `Bx`, reassigning them to servers `A` and `B`.

But what happens with the other object keys, the ones that originally belonged in `A` and `B`? Nothing! That's the beauty of it: The absence of `Cx` labels does not affect those keys in any way. So, removing a server results in its

object keys being randomly reassigned to the rest of the servers, **leaving all other keys untouched**:

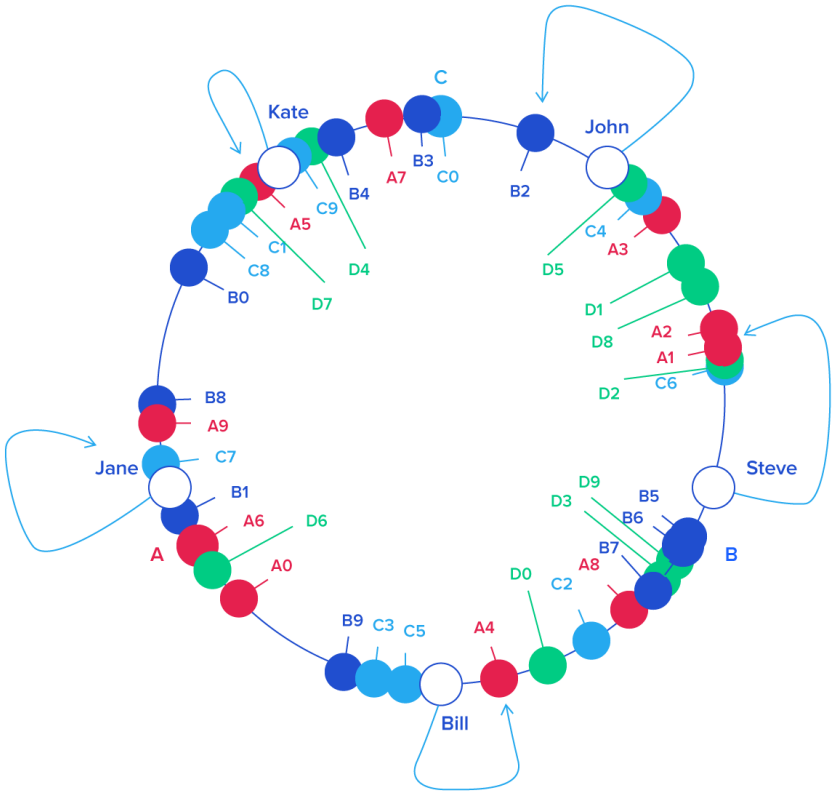


KEY	HASH	ANGLE (DEG)
"A1"	473914830	17
"A2"	548798874	19.7
"A3"	1466730567	52.8
"john"	1633428562	58.7
"B2"	1808009038	65
"B3"	2058758486	74.1
"A7"	2162578920	77.8
"B4"	2660265921	95.7
"kate"	3421657995	123.1
"A5"	3434972143	123.6
"B0"	4049028775	145.7
"B8"	4755525684	171.1
"A9"	4769549830	171.7
"jane"	5000799124	180
"B1"	5444659173	196

"A6"	6210502707	223.5
"A0"	6511384141	234.4
"B9"	7292819872	262.5
"bill"	7594634739	273.4
"A4"	8047401090	289.7
"A8"	8997397092	323.9
"B7"	9038880553	325.3
"B5"	9368225254	337.2
"B6"	9379713761	337.6
"steve"	9787173343	352.3

KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"B2"	B
"kate"	3421831276	123.1	"A5"	A
"jane"	5000648311	180	"B1"	B
"bill"	7594873884	273.4	"A4"	A
"steve"	9786437450	352.3	"A1"	A

Something similar happens if, instead of removing a server, we add one. If we wanted to add server `D` to our example (say, as a replacement for `C`), we would need to add labels `D0 .. D9`. The result would be that roughly one-third of the existing keys (all belonging to `A` or `B`) would be reassigned to `D`, and, again, the rest would stay the same:



KEY	HASH	ANGLE (DEG)
"D2"	439890723	15.8
"A1"	473914830	17
"A2"	548798874	19.7
"D8"	796709216	28.6
"D1"	1008580939	36.3
"A3"	1466730567	52.8
"D5"	1587548309	57.1
"john"	1633428562	58.7
"B2"	1808009038	65
"B3"	2058758486	74.1
"A7"	2162578920	77.8
"B4"	2660265921	95.7
"D4"	2909395217	104.7
"kate"	3421657995	123.1
"A5"	3434972143	123.6
"D7"	3567129743	128.4
"B0"	4049028775	145.7

"B8"	4755525684	171.1
"A9"	4769549830	171.7
"jane"	5000799124	180
"B1"	5444659173	196
"D6"	5703092354	205.3
"A6"	6210502707	223.5
"A0"	6511384141	234.4
"B9"	7292819872	262.5
"bill"	7594634739	273.4
"A4"	8047401090	289.7
"D0"	8272587142	297.8
"A8"	8997397092	323.9
"B7"	9038880553	325.3
"D3"	9048608874	325.7
"D9"	9314459653	335.3
"B5"	9368225254	337.2
"B6"	9379713761	337.6
"steve"	9787173343	352.3

KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"B2"	B
"kate"	3421831276	123.1	"A5"	A
"jane"	5000648311	180	"B1"	B
"bill"	7594873884	273.4	"A4"	A
"steve"	9786437450	352.3	"D2"	D

This is how consistent hashing solves the rehashing problem.

In general, only `k/N` keys need to be remapped when `k` is the number of keys and `N` is the number of servers (more specifically, the maximum of the initial and final number of servers).

What is a reverse proxy?

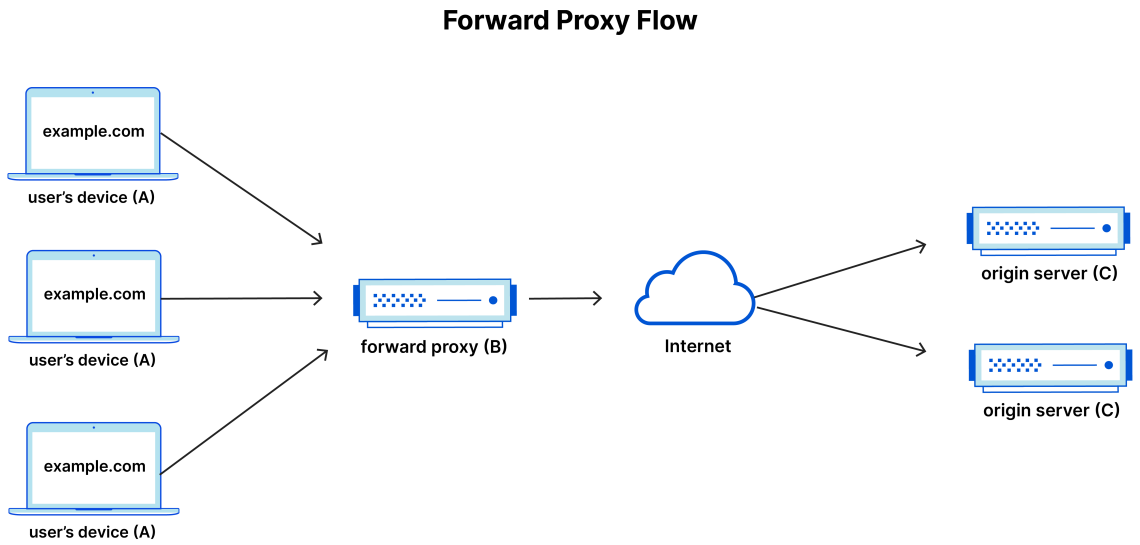
A reverse proxy is a server that sits in front of web servers and forwards client (e.g. web browser) requests to those web servers. Reverse proxies are typically implemented to help increase security, performance, and reliability. In order to better understand how a reverse proxy works and the benefits it can provide, let's first define what a proxy server is.

What is a proxy server?

A forward proxy, often called a proxy, proxy server, or web proxy, is a server that sits in front of a group of client machines. When those computers make requests to sites and services on the Internet, the proxy server intercepts those requests and then communicates with web servers on behalf of those clients, like a middleman.

For example, let's name 3 computers involved in a typical forward proxy communication:

- A: This is a user's home computer
- B: This is a forward proxy server
- C: This is a website's origin server (where the website data is stored)



In a standard Internet communication, computer A would reach out directly to computer C, with the client sending requests to the origin server and the origin server responding to the client. When a forward proxy is in place, A will instead send requests to B, which will then forward the request to C. C will then send a response to B, which will forward the response back to A.

Why would anyone add this extra middleman to their Internet activity? There are a few reasons one might want to use a forward proxy:

- **To avoid state or institutional browsing restrictions** - Some governments, schools, and other organizations use firewalls to give their users access to a limited version of the Internet. A forward proxy can be used to get around these restrictions, as they let the user connect to the proxy rather than directly to the sites they are visiting.
- **To block access to certain content** - Conversely, proxies can also be set up to block a group of users from accessing certain sites. For example, a school network might be configured to connect to the web through a

proxy which enables content filtering rules, refusing to forward responses from Facebook and other social media sites.

- **To protect their identity online** - In some cases, regular Internet users simply desire increased anonymity online, but in other cases, Internet users live in places where the government can impose serious consequences to political dissidents. Criticizing the government in a web forum or on social media can lead to fines or imprisonment for these users. If one of these dissidents uses a forward proxy to connect to a website where they post politically sensitive comments, the IP address used to post the comments will be harder to trace back to the dissident. Only the IP address of the proxy server will be visible.

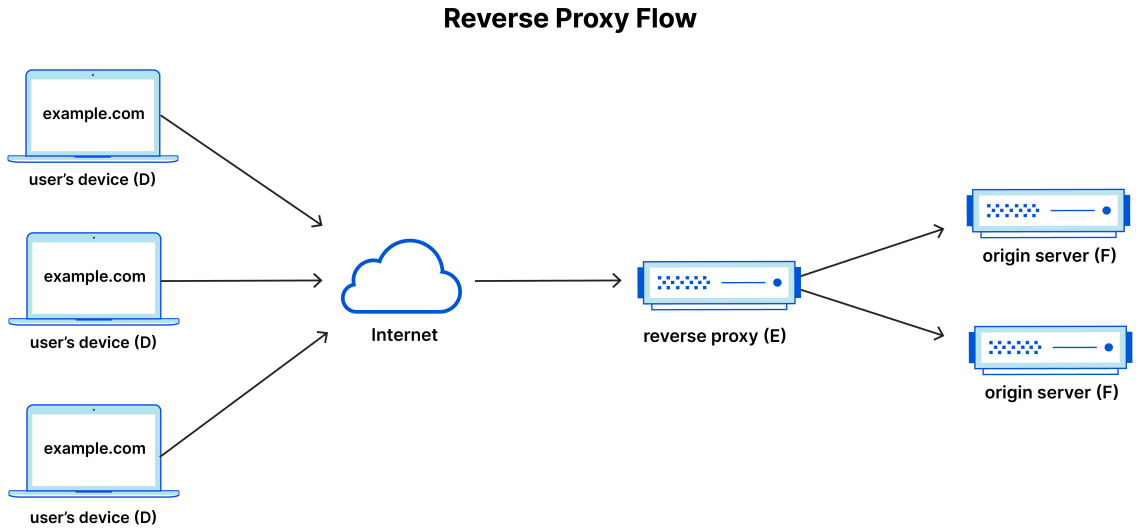
How is a reverse proxy different?

A reverse proxy is a server that sits in front of one or more web servers, intercepting requests from clients. This is different from a forward proxy, where the proxy sits in front of the clients. With a reverse proxy, when clients send requests to the origin server of a website, those requests are intercepted at the network edge by the reverse proxy server. The reverse proxy server will then send requests to and receive responses from the origin server.

The difference between a forward and reverse proxy is subtle but important. A simplified way to sum it up would be to say that a forward proxy sits in front of a client and ensures that no origin server ever communicates directly with that specific client. On the other hand, a reverse proxy sits in front of an origin server and ensures that no client ever communicates directly with that origin server.

Once again, let's illustrate by naming the computers involved:

- D: Any number of users' home computers
- E: This is a reverse proxy server
- F: One or more origin servers



Typically all requests from D would go directly to F, and F would send responses directly to D. With a reverse proxy, all requests from D will go directly to E, and E will send its requests to and receive responses from F. E will then pass along the appropriate responses to D.

Below we outline some of the benefits of a reverse proxy:

- **Load balancing** - A popular website that gets millions of users every day may not be able to handle all of its incoming site traffic with a single origin server. Instead, the site can be distributed among a pool of different servers, all handling requests for the same site. In this case, a reverse proxy can provide a load balancing solution which will distribute the incoming traffic evenly among the different servers to prevent any single server from becoming overloaded. In the event that a server fails completely, other servers can step up to handle the traffic.
- **Protection from attacks** - With a reverse proxy in place, a web site or service never needs to reveal the IP address of their origin server(s). This makes it much harder for attackers to leverage a targeted attack against them, such as a DDoS attack. Instead the attackers will only be able to target the reverse proxy, such as Cloudflare's CDN, which will have tighter security and more resources to fend off a cyber attack.
- **Global server load balancing (GSLB)** - In this form of load balancing, a website can be distributed on several servers around the globe and the reverse proxy will send clients to the server that's geographically closest to them. This decreases the distances that requests and responses need to travel, minimizing load times.
- **Caching** - A reverse proxy can also cache content, resulting in faster performance. For example, if a user in Paris visits a reverse-proxied website with web servers in Los Angeles, the user might actually connect to a local reverse proxy server in Paris, which will then have to communicate with an origin server in L.A. The proxy server can then cache (or temporarily save) the response data. Subsequent Parisian users who browse the site will then get the locally cached version from the Parisian reverse proxy server, resulting in much faster performance.
- **SSL encryption** - Encrypting and decrypting SSL (or TLS) communications for each client can be computationally expensive for an origin server. A reverse proxy can be configured to decrypt all incoming requests and encrypt all outgoing responses, freeing up valuable resources on the origin server.

How to implement a reverse proxy

Some companies build their own reverse proxies, but this requires intensive software and hardware engineering resources, as well as a significant investment in physical hardware. One of the easiest and most cost-effective ways to reap all the benefits of a reverse proxy is by signing up for a CDN service. For example, the Cloudflare CDN provides all the performance and security features listed above, as well as many others.

What are WebSockets?

A WebSocket is a communication protocol that provides full-duplex communication channels over a single TCP connection. It enables real-time, event-driven communication between a client and a server.

Unlike traditional HTTP, which follows a request-response model, WebSockets allow bi-directional communication. This means that the client and the server can send data to each other anytime without continuous polling.

What are WebSockets Used For?

WebSockets are used for real-time, event-driven communication between clients and servers. They are particularly useful for building applications requiring instant updates, such as real-time chat, messaging, and multiplayer games.

In traditional HTTP, the client sends a request to the server, and the server responds with the requested data. This request-response model requires continuous polling from the client to the server, which can result in

increased latency and decreased efficiency.

On the other hand, WebSockets establish a persistent connection between the client and the server. This means that once the connection is established, the client and the server can send data to each other at any time without continuous polling. This allows realtime communication, where updates can be sent and received instantly.

For example, when a user sends a message in a chat application, it can be instantly delivered to all other users without refreshing the page or making frequent HTTP requests. This results in a more seamless and efficient user experience.

Furthermore, Web Sockets also allow for bi-directional communication, meaning that both the client and the server can send data to each other. This opens up possibilities for more interactive and engaging applications, where the server can push updates or notifications to the client without the client explicitly requesting them.

Drawbacks of Web Sockets

The drawbacks of WebSockets include:

- Browser Support: Although most modern browsers support WebSockets, some older ones do not. This can limit the reach of your application and require additional fallback mechanisms for older browsers.
- Proxy and Firewall Limitations: Some proxy servers and firewalls may block or interfere with WebSocket connections. This can cause connectivity issues, especially in corporate or restricted network environments.
- Scalability: Web Sockets maintain a persistent connection between the client and the server, which can strain server resources when dealing with many concurrent connections. Proper load balancing and resource management techniques must be implemented to ensure scalability. Open-source resources, like Socket.io, are not great for large-scale operations or quick growth.
- Stateful Nature: Unlike traditional HTTP, which is stateless, WebSockets are stateful. This means that the server needs to maintain the connection state for each client, leading to increased memory usage and potential scalability challenges.
- Security Considerations: With the persistent connection established by WebSockets, there is a need for proper security measures to protect against potential vulnerabilities, such as cross-site scripting (XSS) and cross-site request forgery (CSRF). Secure WebSocket connections (wss://) using SSL/TLS encryption should be implemented to ensure data privacy and integrity.
- If a connection over Web Sockets is lost, there are no included load balancing or reconnecting mechanisms.
- It is still necessary to have fallback options, like HTTP streaming or long polling, in environments where Web Sockets may not be supported.
- Features like Presence do not work well over WebSocket connections because disconnections are hard to detect.

WebSockets vs. HTTP vs. web servers vs. polling

HTTP connections vs. WebSockets

To understand the WebSocket API, it is also important to understand the foundation it was built on – HTTP (Hypertext Transfer Protocol) and its request/response model. HTTP is an application layer protocol and the basis for all web-based communication and data transfers.

When using HTTP, clients—such as web browsers—send requests to servers, and then the servers send messages back, known as responses. The web as we know it today was built on this basic client-server cycle, although there have been many additions and updates to HTTP to make it more interactive. There are currently a few viable and supported versions of HTTP—HTTP/1.1 and HTTP/2—and a secure version known as HTTPS.

Basic HTTP requests work well for many use cases, such as when someone needs to search on a web page and receive relevant, non-time-sensitive information. However, it is not always best suited for web applications requiring real-time communication or data that needs to update quickly with minimal latency.

Whenever the client makes a new HTTP server request, the default behavior is to open a new HTTP connection. This is inefficient because it uses bandwidth on recurring non-payload data and increases latency between the data transfers.

Additionally, HTTP requests can only flow in one direction—from the client side. There is traditionally no mechanism for the server to initiate communication with the client. The server cannot send data to the client unless it requests it first. This can create issues for use cases where messaging needs to go out in real time from the server side.

Short polling vs. WebSockets

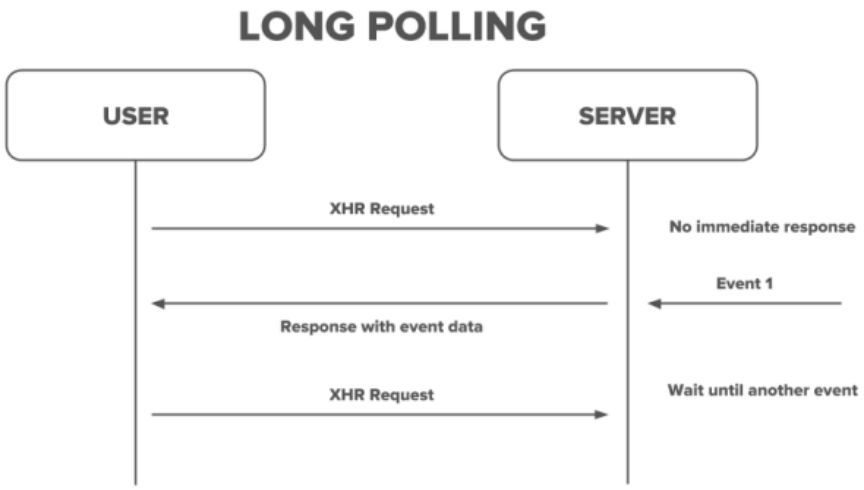
One of the first solutions for receiving regular data updates was HTTP polling. Polling is a technique where the client repeatedly sends requests to the server until it updates. For example, all modern web browsers offer support for XMLHttpRequest, one of the original methods of polling servers.

These earlier solutions were still not ideal for efficient real-time communication—short polling is intensive because, for every request, the non-payload data is re-sent and must be parsed, including the header HTML, the web URL, and other repetitive information that wastes resources.

Long polling vs. Web Sockets

The next logical step to improve latency was HTTP long polling. When long polling, the client polls the server, and that connection remains open until the server has new data. The server sends the response with the relevant information, and then the client immediately opens another request, holding again until the next update. Long polling can hold a connection open for a maximum of 280 seconds before automatically sending another request. This method effectively emulates an HTTP server push.

Long polling provides fast communication in many environments and is widely used, often as opposed to true push-based methods like WebSocket connections or Server Side Events (SSE). Long polling can seem intensive on the server side, as it requires continuous resources to hold a connection open, but it uses much less than repeatedly sending polling requests.



Read more on: Long Polling vs Websockets

What are WebSockets used for?

Developers invented WebSockets to facilitate real-time results effectively. WebSockets initiate continuous, full-duplex communication between a client and WebSocket server. This reduces unnecessary network traffic, as data can immediately travel both ways through a single open connection. This provides speed and realtime capability on the web. Websockets also enable servers to keep track of clients and “push” data to them as needed, which was not possible using only HTTP.

WebSocket connections enable the streaming of text strings and binary data via messages. WebSocket messages include a frame, payload, and data portion. Very little non-payload data gets sent across the existing network connection this way, which helps to reduce latency and overhead, especially when compared to HTTP request and streaming models.

Google Chrome was the first browser to include standard support for WebSockets in 2009. RFC 6455—The WebSocket Protocol—was officially published online in 2011. The WebSocket Protocol and WebSocket API are standardized by the W3C and the IETF, and support across browsers is very common.

How do WebSockets work (and their connections)

Before a client and server exchange data, they must use the TCP (Transport Control Protocol) layer to establish the connection. Using their WebSocket protocol, webSockets effectively run as a transport layer over the TCP connection.

Once connected through an HTTP request/response pair, the clients can use an HTTP/1.1 upgrade header to switch their connection from HTTP to WebSockets. WebSocket connections are fully asynchronous, unlike

HTTP/1.1, however. WebSocket connection is established through a WebSocket handshake over the TCP. During a new WebSocket handshake, the client and server also communicate which subprotocol will be used for subsequent interactions. After this is established, the connection will run on the WebSocket protocol.

It is important to note that when running on the WebSocket protocol layer, WebSockets require a uniform resource identifier (URI) to use a “ws:” or “wss:” scheme, similar to how HTTP URLs will always use a “http:” or “https:” scheme.

What libraries are available for implementing WebSockets?

Several libraries can provide the necessary tools and functionalities when implementing WebSockets in your realtime chat and messaging applications. These libraries offer a wide range of features and support for different programming languages, making it easier for developers to integrate WebSockets into their applications. Here are some of the popular libraries that you can consider:

1. Socket.IO: Socket.IO is a widely used library that provides realtime bidirectional event-based communication between the browser and the server. It offers features like automatic reconnection, fallback options, and support for various transports, making it an excellent choice for building scalable and reliable applications. Socket.IO supports multiple programming languages, including JavaScript, Python, and Java.

2. SignalR: SignalR is a realtime communication library developed by Microsoft. It allows you to build realtime web applications by providing a simple API for creating WebSockets connections. SignalR supports server-side and client-side implementations and can be used with .NET, JavaScript, and other languages. It also offers automatic connection management, broadcasting messages, and scaling across multiple servers.

3. SockJS: SockJS is a JavaScript library that provides a WebSocket-like object in the browser, even if the server doesn't support WebSockets. It offers a fallback mechanism that uses alternative transport protocols, such as HTTP long-polling, allowing your application to work in environments where websockets are unavailable. SockJS can be used with various backends and programming languages, including Node.js, Java, and Python.

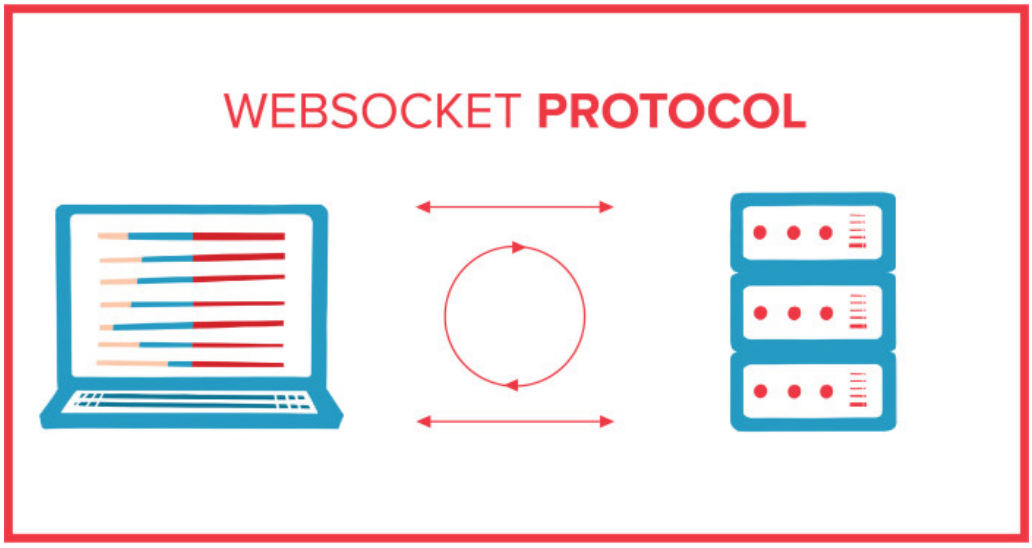
4. ws: ws is a simple and lightweight WebSocket implementation for Node.js. It provides a straightforward API for creating WebSocket servers and clients, making it easy to integrate websockets into your Node.js applications. ws offers per-message compression, automatic reconnection, and customizable options for handling incoming and outgoing messages.

5. Django Channels: Django Channels is a library that extends the capabilities of the Django web framework to handle real-time applications. It supports websockets and other protocols like HTTP long-polling and Server-Sent Events. Django Channels allows you to build real-time chat and messaging applications using the familiar Django syntax and tools.

Reasons to consider WebSockets for real-time communication

- Websockets provide real-time updates and open lines of communication.
- Websockets are HTML5 compliant and offer backward compatibility with older HTML documents. Therefore, all modern web browsers—Google Chrome, Mozilla Firefox, Apple Safari, and more- support them.
- WebSockets are compatible across Android, iOS, web, and desktop platforms.
- A single server can have multiple WebSocket connections open simultaneously and multiple connections with the same client, which opens the door for scalability.
- WebSockets can stream through many proxies and firewalls.

- There are many open-source resources and tutorials for incorporating WebSockets in an application, like the Javascript library Socket.io.



PubNub's Take on WebSockets vs. Long Polling

PubNub takes a protocol-agnostic stance, but in our current operations, we have found that long polling is the best bet for most use cases. This is partly because of the maintenance and upkeep required to scale WebSockets on the backend and potential issues that can arise when you can not easily identify a disconnection. WebSockets are a great tool, but long polling works reliably in every situation.

PubNub uses long polling to ensure reliability, security, and scalability in all networking environments, not just most. Long polling can be as efficient as WebSockets in many real-world, real-time implementations. We have developed a method for efficient long polling – written in C and with multiple kernel optimizations for scale.

PubNub is a real-time communications platform that provides the foundation for authentic virtual experiences, like live updates, in-app chat, push notifications, and more. The building block structure of our platform allows for extra features like Presence, operational dashboards, or geolocation to be incorporated. PubNub also makes it extremely easy to scale, especially compared to socket frameworks like Socket.io or SocksJS.

Summary

To conclude, WebSockets are a very useful protocol for building real-time functionality across web, mobile, and desktop variants, but they are not a one-size-fits-all approach. WebSockets are just one tool that fits into a larger arsenal when developing real-time, communication-based applications that require low latency. It is possible to build off of basic WebSocket protocol, incorporate other methods like SSE or long polling, and construct an even

better, more scalable real-time application. The problem is that when you use WebSockets, the shortcomings can be difficult to manage if you are not an expert in building real-time systems.

Using PubNub provides a better user experience, saves significant development time and maintenance costs, speeds up time to market, and reduces the complexity of what your engineering and web development team will need to develop, manage, and grow.

With over 15 points of presence worldwide supporting 800 million monthly active users and 99.999% reliability, you'll never have to worry about outages, concurrency limits, or any latency issues caused by traffic spikes. PubNub is perfect for any application that requires real-time data.

Sign up for a free trial and get up to 200 MAUs or 1M total transactions per month included.