Contents

Performance vs scalability

A service is **scalable** if it results in increased **performance** in a manner proportional to resources added. Generally, increasing performance means serving more units of work, but it can also be to handle larger units of work, such as when datasets grow.[1]

Another way to look at performance vs scalability:

- If you have a **performance** problem, your system is slow for a single user.
- If you have a **scalability** problem, your system is fast for a single user but slow under heavy load.

Latency vs throughput

Latency is the time to perform some action or to produce some result.

Throughput is the number of such actions or results per unit of time.

Generally, you should aim for **maximal throughput** with **acceptable latency**.

Availability vs consistency

CAP theorem

In a distributed computer system, you can only support two of the following guarantees:

- **Consistency** - Every read receives the most recent write or an error
- **Availability** - Every request receives a response, without guarantee that it contains the most recent version of the information
- **Partition Tolerance** - The system continues to operate despite arbitrary partitioning due to network failures
- **Consistency –**
  Consistency means that the nodes will have the same copies of a replicated data item visible for various transactions. A guarantee that every node in a distributed cluster returns the same, most recent and a successful write. Consistency refers to every client having the same view of the data. There are various types of consistency models. Consistency in CAP refers to sequential consistency, a very strong form of consistency.

- **Availability –**
  Availability means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed. Every non-failing node returns a response for all the read and write requests in a reasonable amount of time. The key word here is “every”. In simple terms, every node (on either side of a network partition) must be able to respond in a reasonable amount of time.

- **Partition Tolerance –**
  Partition tolerance means that the system can continue operating even if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other. That means, the system continues to function and upholds its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the

partition heals.

The use of the word consistency in CAP and its use in ACID do not refer to the same identical concept.

In CAP, the term consistency refers to the consistency of the values in different copies of the same data item in a replicated distributed system. In ACID, it refers to the fact that a transaction will not violate the integrity constraints specified on the database schema.

The CAP theorem states that it is not possible to guarantee all three of the desirable properties – consistency, availability, and partition tolerance at the same time in a distributed system with data replication.

The theorem states that networked shared-data systems can only strongly support two of the following three properties:

Consistency patterns

With multiple copies of the same data, we are faced with options on how to synchronize them so clients have a consistent view of the data. Recall the definition of consistency from the CAP theorem - Every read receives the most recent write or an error.

Weak consistency

After a write, reads may or may not see it. A best effort approach is taken.

This approach is seen in systems such as memcached. Weak consistency works well in real time use cases such as VoIP, video chat, and realtime multiplayer games. For example, if you are on a phone call and lose reception for a few seconds, when you regain connection you do not hear what was spoken during connection loss.

Eventual consistency

After a write, reads will eventually see it (typically within milliseconds). Data is replicated asynchronously.

This approach is seen in systems such as DNS and email. Eventual consistency works well in highly available systems.

Strong consistency

After a write, reads will see it. Data is replicated synchronously.

This approach is seen in file systems and RDBMSes. Strong consistency works well in systems that need transactions.

Availability patterns

There are two complementary patterns to support high availability: **fail-over** and **replication**.

Fail-over

Active-passive

With active-passive fail-over, heartbeats are sent between the active and the passive server on standby. If the heartbeat is interrupted, the passive server takes over the active's IP address and resumes service.

The length of downtime is determined by whether the passive server is already running in 'hot' standby or whether it needs to start up from 'cold' standby. Only the active server handles traffic.

Active-passive failover can also be referred to as master-slave failover.

Active-active

In active-active, both servers are managing traffic, spreading the load between them.

If the servers are public-facing, the DNS would need to know about the public IPs of both servers. If the servers are internal-facing, application logic would need to know about both servers.

Active-active failover can also be referred to as master-master failover.

Disadvantage(s): failover

- Fail-over adds more hardware and additional complexity.
- There is a potential for loss of data if the active system fails before any newly written data can be replicated to the passive.

Replication

Master-slave and master-master

This topic is further discussed in the Database section:

- Master-slave replication
- Master-master replication

Availability in numbers

Availability is often quantified by uptime (or downtime) as a percentage of time the service is available. Availability is generally measured in number of 9s--a service with 99.99% availability is described as having four 9s.

99.9% availability - three 9s

Duration	Acceptable downtime
Downtime per year	8h 45min 57s
Downtime per month	43m 49.7s
Downtime per week	10m 4.8s
Downtime per day	1m 26.4s

99.99% availability - four 9s

Duration	Acceptable downtime
Downtime per year	52min 35.7s
Downtime per month	4m 23s
Downtime per week	1m 5s
Downtime per day	8.6s

Availability in parallel vs in sequence

If a service consists of multiple components prone to failure, the service's overall availability depends on whether the components are in sequence or in parallel.

In sequence

Overall availability decreases when two components with availability < 100% are in sequence:

`Availability (Total) = Availability (Foo) * Availability (Bar)`
If both `Foo` and `Bar` each had 99.9% availability, their total availability in sequence would be 99.8%.

In parallel

Overall availability increases when two components with availability < 100% are in parallel:

`Availability (Total) = 1 - (1 - Availability (Foo)) * (1 - Availability (Bar))`
If both `Foo` and `Bar` each had 99.9% availability, their total availability in parallel would be 99.9999%.

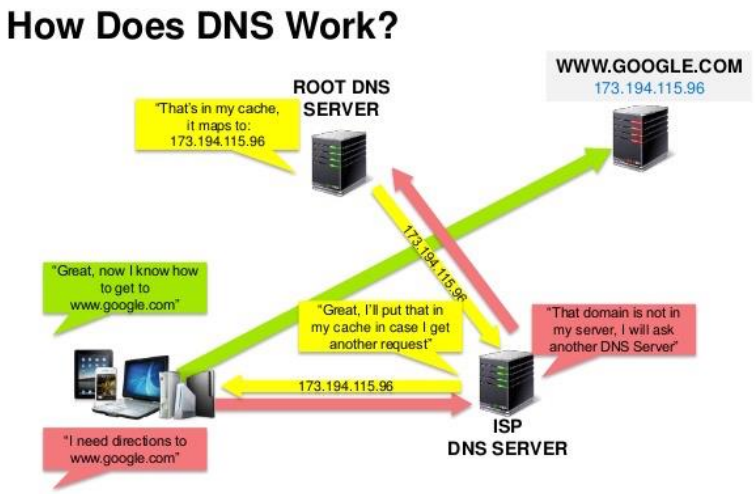
TCP vs UDP

TCP is a connection-oriented protocol, whereas UDP is a connectionless protocol. A key difference between TCP and UDP is speed, as TCP is comparatively slower than UDP. Overall, UDP is a much faster, simpler, and more efficient protocol, however, retransmission of lost data packets is only possible with TCP.

TCP provides ordered delivery of data from user to server (and vice versa), whereas UDP is not dedicated to end-to-end communications, nor does it check the readiness of the receiver.

Feature	TCP	UDP
Connection	Requires an established connection	Connectionless protocol
Guaranteed delivery	Can guarantee delivery of data	Cannot guarantee delivery of data
Re-transmission	Re-transmission of lost packets is possible	No re-transmission of lost packets
Speed	Slower than UDP	Faster than TCP
Broadcasting	Does not support broadcasting	Supports broadcasting
Use cases	HTTPS, HTTP, SMTP, POP, FTP, etc	Video streaming, DNS, VoIP, etc

Domain name system



Source: DNS security presentation

A Domain Name System (DNS) translates a domain name such as www.example.com to an IP address.

DNS is hierarchical, with a few authoritative servers at the top level. Your router or ISP provides information about which DNS server(s) to contact when doing a lookup. Lower level DNS servers cache mappings, which could become stale due to DNS propagation delays. DNS results can also be cached by your browser or OS for a certain period of time, determined by the time to live (TTL).

- **NS record (name server)** - Specifies the DNS servers for your domain/subdomain.
- **MX record (mail exchange)** - Specifies the mail servers for accepting messages.
- **A record (address)** - Points a name to an IP address.
- **CNAME (canonical)** - Points a name to another name or `CNAME` (example.com to www.example.com) or to an `A` record.

Services such as CloudFlare and Route 53 provide managed DNS services. Some DNS services can route traffic through various methods:

- Weighted round robin
  - Prevent traffic from going to servers under maintenance
  - Balance between varying cluster sizes
  - A/B testing
- Latency-based
- Geolocation-based

Disadvantage(s): DNS

- Accessing a DNS server introduces a slight delay, although mitigated by caching described above.

- DNS server management could be complex and is generally managed by governments, ISPs, and large companies.
- DNS services have recently come under DDoS attack, preventing users from accessing websites such as Twitter without knowing Twitter's IP address(es).

Examples

These are some widely used managed DNS solutions:

- Route53
- Cloudflare DNS
- Google Cloud DNS
- Azure DNS
- NS1

Content delivery network

Source: Why use a CDN

A content delivery network (CDN) is a globally distributed network of proxy servers, serving content from locations closer to the user. Generally, static files such as HTML/CSS/JS, photos, and videos are served from CDN, although some CDNs such as Amazon's CloudFront support dynamic content. The site's DNS resolution will tell clients which server to contact.

Serving content from CDNs can significantly improve performance in two ways:

- Users receive content from data centers close to them
- Your servers do not have to serve requests that the CDN fulfills

Push CDNs

Push CDNs receive new content whenever changes occur on your server. You take full responsibility for providing content, uploading directly to the CDN and rewriting URLs to point to the CDN. You can configure when content expires and when it is updated. Content is uploaded only when it is new or changed, minimizing traffic, but maximizing storage.

Sites with a small amount of traffic or sites with content that isn't often updated work well with push CDNs. Content is placed on the CDNs once, instead of being re-pulled at regular intervals.

Pull CDNs

Pull CDNs grab new content from your server when the first user requests the content. You leave the content on your server and rewrite URLs to point to the CDN. This results in a slower request until the content is cached on the CDN.

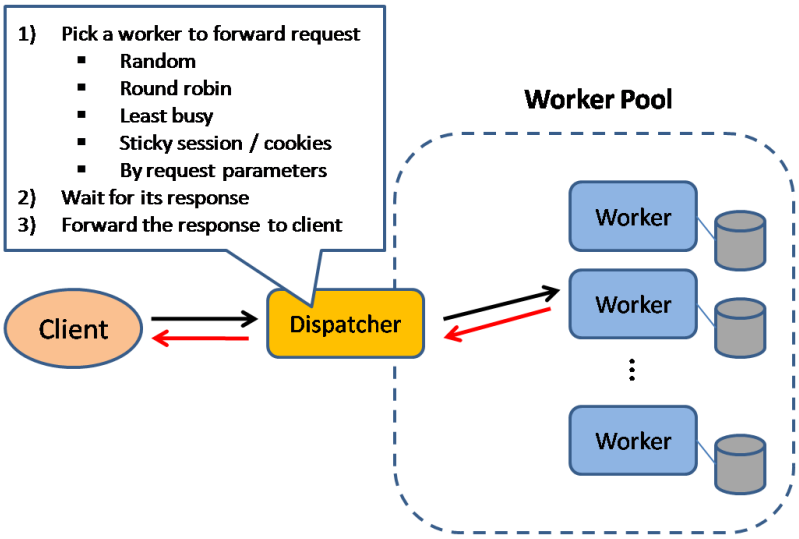
A time-to-live (TTL) determines how long content is cached. Pull CDNs minimize storage space on the CDN, but can create redundant traffic if files expire and are pulled before they have actually changed.

Sites with heavy traffic work well with pull CDNs, as traffic is spread out more evenly with only recently-requested content remaining on the CDN.

Disadvantage(s): CDN

- CDN costs could be significant depending on traffic, although this should be weighed with additional costs you would incur not using a CDN.
- Content might be stale if it is updated before the TTL expires it.
- CDNs require changing URLs for static content to point to the CDN.

Load balancer



Source: Scalable system design patterns

Load balancers distribute incoming client requests to computing resources such as application servers and databases. In each case, the load balancer returns the response from the computing resource to the appropriate client. Load balancers are effective at:

- Preventing requests from going to unhealthy servers
- Preventing overloading resources
- Helping to eliminate a single point of failure

Load balancers can be implemented with hardware (expensive) or with software such as HAProxy.

Additional benefits include:

- **SSL termination** - Decrypt incoming requests and encrypt server responses so backend servers do not have to perform these potentially expensive operations
    - Removes the need to install X.509 certificates on each server
- **Session persistence** - Issue cookies and route a specific client's requests to same instance if the web apps do not keep track of sessions

To protect against failures, it's common to set up multiple load balancers, either in active-passive or active-active mode.

Load balancers can route traffic based on various metrics, including:

- Random
- Least loaded
- Session/cookies
- Round robin or weighted round robin
- Layer 4
- Layer 7

Layer 4 load balancing

Layer 4 load balancers look at info at the transport layer to decide how to distribute requests. Generally, this involves the source, destination IP addresses, and ports in the header, but not the contents of the packet. Layer 4 load balancers forward network packets to and from the upstream server, performing Network Address Translation (NAT).

Layer 7 load balancing

Layer 7 load balancers look at the application layer to decide how to distribute requests. This can involve contents of the header, message, and cookies. Layer 7 load balancers terminate network traffic, reads the message, makes a load-balancing decision, then opens a connection to the selected server. For example, a layer 7 load balancer can direct video traffic to servers that host videos while directing more sensitive user billing traffic to security-hardened servers.

At the cost of flexibility, layer 4 load balancing requires less time and computing resources than Layer 7, although the performance impact can be minimal on modern commodity hardware.

Horizontal scaling

Load balancers can also help with horizontal scaling, improving performance and availability. Scaling out using commodity machines is more cost efficient and results in higher availability than scaling up a single server on more expensive hardware, called **Vertical Scaling**. It is also easier to hire for talent working on commodity hardware than it is for specialized enterprise systems.

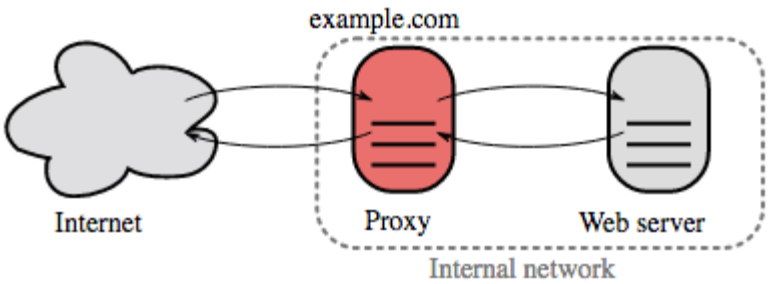
Disadvantage(s): horizontal scaling

- Scaling horizontally introduces complexity and involves cloning servers
  - Servers should be stateless: they should not contain any user-related data like sessions or profile pictures
  - Sessions can be stored in a centralized data store such as a database (SQL, NoSQL) or a persistent cache (Redis, Memcached)
- Downstream servers such as caches and databases need to handle more simultaneous connections as upstream servers scale out

Disadvantage(s): load balancer

- The load balancer can become a performance bottleneck if it does not have enough resources or if it is not configured properly.
- Introducing a load balancer to help eliminate a single point of failure results in increased complexity.
- A single load balancer is a single point of failure, configuring multiple load balancers further increases complexity.

Reverse proxy (web server)



Source: Wikipedia

A reverse proxy is a web server that centralizes internal services and provides unified interfaces to the public. Requests from clients are forwarded to a server that can fulfill it before the reverse proxy returns the server's response to the client.

Additional benefits include:

- **Increased security** - Hide information about backend servers, blacklist IPs, limit number of connections per client

- **Increased scalability and flexibility** - Clients only see the reverse proxy's IP, allowing you to scale servers or change their configuration
- **SSL termination** - Decrypt incoming requests and encrypt server responses so backend servers do not have to perform these potentially expensive operations
    - Removes the need to install X.509 certificates on each server
- **Compression** - Compress server responses
- **Caching** - Return the response for cached requests
- **Static content** - Serve static content directly
    - HTML/CSS/JS
    - Photos
    - Videos
    - Etc

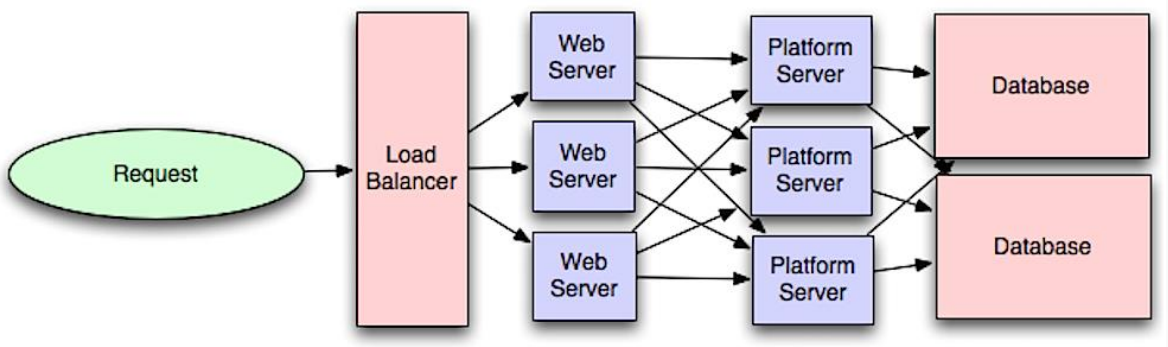
Load balancer vs reverse proxy

- Deploying a load balancer is useful when you have multiple servers. Often, load balancers route traffic to a set of servers serving the same function.
- Reverse proxies can be useful even with just one web server or application server, opening up the benefits described in the previous section.
- Solutions such as NGINX and HAProxy can support both layer 7 reverse proxying and load balancing.

Disadvantage(s): reverse proxy

- Introducing a reverse proxy results in increased complexity.
- A single reverse proxy is a single point of failure, configuring multiple reverse proxies (ie a failover) further increases complexity.

Application layer



Source: Intro to architecting systems for scale

Separating out the web layer from the application layer (also known as platform layer) allows you to scale and configure both layers independently. Adding a new API results in adding application servers without necessarily adding additional web servers. The **single responsibility principle** advocates for

small and autonomous services that work together. Small teams with small services can plan more aggressively for rapid growth.

Workers in the application layer also help enable asynchronism.

Microservices

Related to this discussion are microservices, which can be described as a suite of independently deployable, small, modular services. Each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal. [1]

Pinterest, for example, could have the following microservices: user profile, follower, feed, search, photo upload, etc.

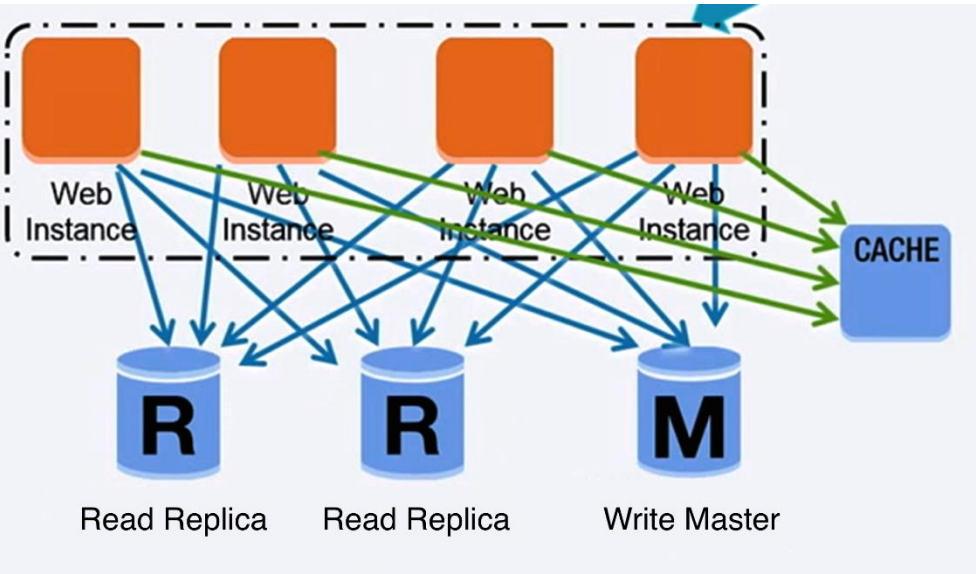
Service Discovery

Systems such as Consul, Etcd, and Zookeeper can help services find each other by keeping track of registered names, addresses, and ports. Health checks help verify service integrity and are often done using an HTTP endpoint. Both Consul and Etcd have a built in key-value store that can be useful for storing config values and other shared data.

Disadvantage(s): application layer

- Adding an application layer with loosely coupled services requires a different approach from an architectural, operations, and process viewpoint (vs a monolithic system).
- Microservices can add complexity in terms of deployments and operations.

Database



Source: Scaling up to your first 10 million users

Relational database management system (RDBMS)

A relational database like SQL is a collection of data items organized in tables.

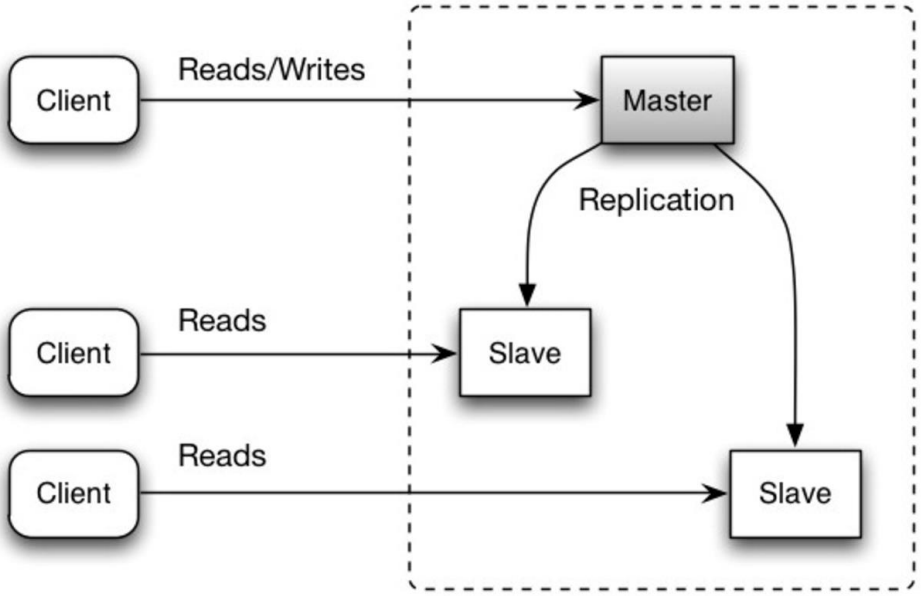
ACID is a set of properties of relational database transactions.

- **Atomicity** - Each transaction is all or nothing
- **Consistency** - Any transaction will bring the database from one valid state to another
- **Isolation** - Executing transactions concurrently has the same results as if the transactions were executed serially
- **Durability** - Once a transaction has been committed, it will remain so

There are many techniques to scale a relational database: **master-slave replication**, **master-master replication**, **federation**, **sharding**, **denormalization**, and **SQL tuning**.

Master-slave replication

The master serves reads and writes, replicating writes to one or more slaves, which serve only reads. Slaves can also replicate to additional slaves in a tree-like fashion. If the master goes offline, the system can continue to operate in read-only mode until a slave is promoted to a master or a new master is provisioned.



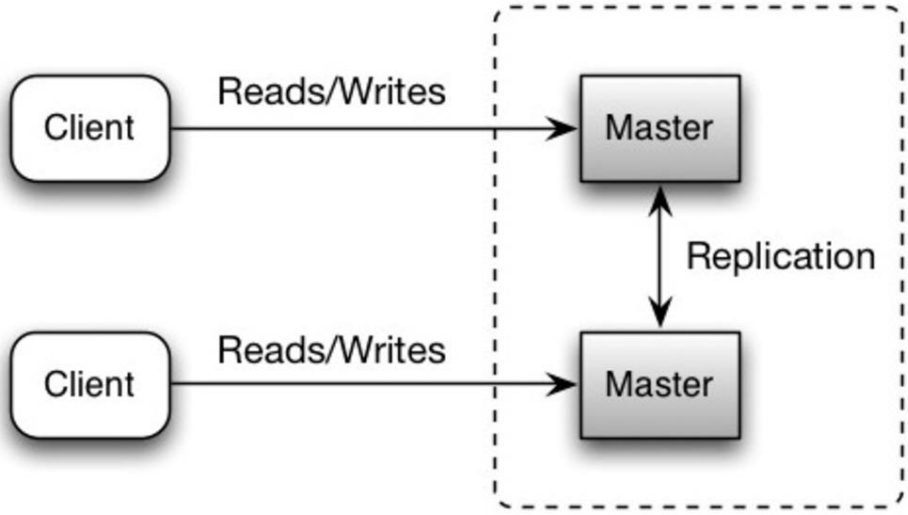
Source: Scalability, availability, stability, patterns

Disadvantage(s): master-slave replication

- Additional logic is needed to promote a slave to a master.
- See Disadvantage(s): replication for points related to **both** master-slave and master-master.

Master-master replication

Both masters serve reads and writes and coordinate with each other on writes. If either master goes down, the system can continue to operate with both reads and writes.



Source: Scalability, availability, stability, patterns

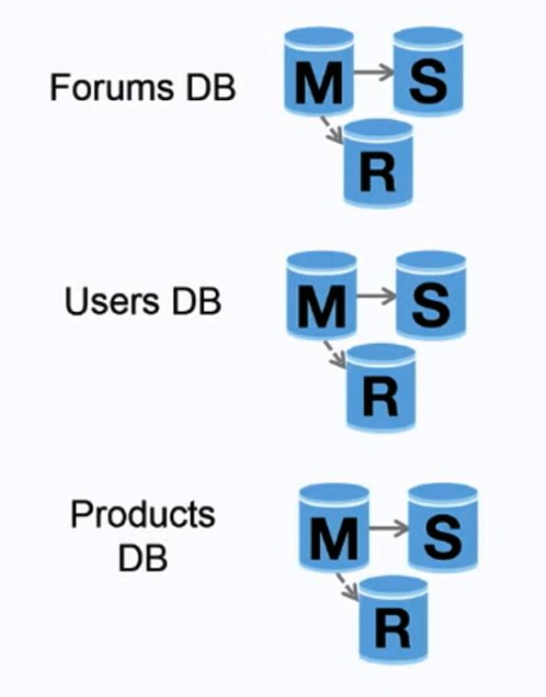
Disadvantage(s): master-master replication

- You'll need a load balancer or you'll need to make changes to your application logic to determine where to write.
- Most master-master systems are either loosely consistent (violating ACID) or have increased write latency due to synchronization.
- Conflict resolution comes more into play as more write nodes are added and as latency increases.
- See Disadvantage(s): replication for points related to **both** master-slave and master-master.

Disadvantage(s): replication

- There is a potential for loss of data if the master fails before any newly written data can be replicated to other nodes.
- Writes are replayed to the read replicas. If there are a lot of writes, the read replicas can get bogged down with replaying writes and can't do as many reads.
- The more read slaves, the more you have to replicate, which leads to greater replication lag.
- On some systems, writing to the master can spawn multiple threads to write in parallel, whereas read replicas only support writing sequentially with a single thread.
- Replication adds more hardware and additional complexity.

Federation



Source: Scaling up to your first 10 million users

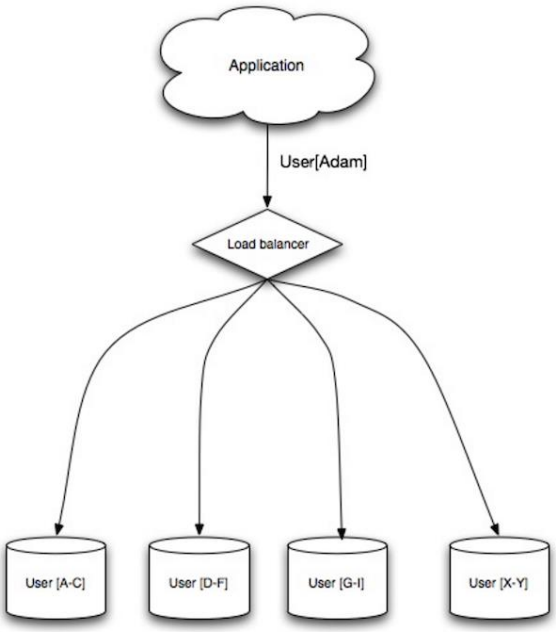
Federation (or functional partitioning) splits up databases by function. For example, instead of a single, monolithic database, you could have three databases: **forums**, **users**, and **products**, resulting in less read and write traffic to each database and therefore less replication lag. Smaller databases

result in more data that can fit in memory, which in turn results in more cache hits due to improved cache locality. With no single central master serializing writes you can write in parallel, increasing throughput.

Disadvantage(s): federation

- Federation is not effective if your schema requires huge functions or tables.
- You'll need to update your application logic to determine which database to read and write.
- Joining data from two databases is more complex with a server link.
- Federation adds more hardware and additional complexity.

Sharding



Source: Scalability, availability, stability, patterns

Sharding distributes data across different databases such that each database can only manage a subset of the data. Taking a users database as an example, as the number of users increases, more shards are added to the cluster.

Similar to the advantages of federation, sharding results in less read and write traffic, less replication, and more cache hits. Index size is also reduced, which generally improves performance with faster queries. If one shard goes down, the other shards are still operational, although you'll want to add some form of replication to avoid data loss. Like federation, there is no single central master serializing writes, allowing you to write in parallel with increased throughput.

Common ways to shard a table of users is either through the user's last name initial or the user's geographic location.

Disadvantage(s): sharding

- You'll need to update your application logic to work with shards, which could result in complex SQL queries.
- Data distribution can become lopsided in a shard. For example, a set of power users on a shard could result in increased load to that shard compared to others.
    - Rebalancing adds additional complexity. A sharding function based on consistent hashing can reduce the amount of transferred data.
- Joining data from multiple shards is more complex.
- Sharding adds more hardware and additional complexity.

Source(s) and further reading: sharding

- The coming of the shard
- Shard database architecture
- Consistent hashing

Denormalization

Denormalization attempts to improve read performance at the expense of some write performance. Redundant copies of the data are written in multiple tables to avoid expensive joins. Some RDBMS such as PostgreSQL and Oracle support materialized views which handle the work of storing redundant information and keeping redundant copies consistent.

Once data becomes distributed with techniques such as federation and sharding, managing joins across data centers further increases complexity. Denormalization might circumvent the need for such complex joins.

In most systems, reads can heavily outnumber writes 100:1 or even 1000:1. A read resulting in a complex database join can be very expensive, spending a significant amount of time on disk operations.

Disadvantage(s): denormalization

- Data is duplicated.
- Constraints can help redundant copies of information stay in sync, which increases complexity of the database design.
- A denormalized database under heavy write load might perform worse than its normalized counterpart.

SQL tuning

SQL tuning is a broad topic and many books have been written as reference.

It's important to **benchmark** and **profile** to simulate and uncover bottlenecks.

- **Benchmark** - Simulate high-load situations with tools such as ab.
- **Profile** - Enable tools such as the slow query log to help track performance issues.

Benchmarking and profiling might point you to the following optimizations.

Tighten up the schema

- MySQL dumps to disk in contiguous blocks for fast access.
- Use `CHAR` instead of `VARCHAR` for fixed-length fields.
    - `CHAR` effectively allows for fast, random access, whereas with `VARCHAR`, you must find the end of a string before moving onto the next one.
- Use `TEXT` for large blocks of text such as blog posts. `TEXT` also allows for boolean searches. Using a `TEXT` field results in storing a pointer on disk that is used to locate the text block.
- Use `INT` for larger numbers up to 2^32 or 4 billion.
- Use `DECIMAL` for currency to avoid floating point representation errors.
- Avoid storing large `BLOBS`, store the location of where to get the object instead.
- `VARCHAR(255)` is the largest number of characters that can be counted in an 8 bit number, often maximizing the use of a byte in some RDBMS.
- Set the `NOT NULL` constraint where applicable to improve search performance.

Use good indices

- Columns that you are querying (`SELECT`, `GROUP BY`, `ORDER BY`, `JOIN`) could be faster with indices.
- Indices are usually represented as self-balancing B-tree that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
- Placing an index can keep the data in memory, requiring more space.
- Writes could also be slower since the index also needs to be updated.
- When loading large amounts of data, it might be faster to disable indices, load the data, then rebuild the indices.

Avoid expensive joins

- Denormalize where performance demands it.

Partition tables

- Break up a table by putting hot spots in a separate table to help keep it in memory.

Tune the query cache

- In some cases, the query cache could lead to performance issues.

NoSQL

NoSQL is a collection of data items represented in a **key-value store**, **document store**, **wide column store**, or a **graph database**. Data is denormalized, and joins are generally done in the application code. Most NoSQL stores lack true ACID transactions and favor eventual consistency.

BASE is often used to describe the properties of NoSQL databases. In comparison with the CAP Theorem, BASE chooses availability over consistency.

- **Basically available** - the system guarantees availability.
- **Soft state** - the state of the system may change over time, even without input.
- **Eventual consistency** - the system will become consistent over a period of time, given that the system doesn't receive input during that period.

In addition to choosing between SQL or NoSQL, it is helpful to understand which type of NoSQL database best fits your use case(s). We'll review **key-value stores**, **document stores**, **wide column stores**, and **graph databases** in the next section.

Key-value store

Abstraction: hash table
A key-value store generally allows for O(1) reads and writes and is often backed by memory or SSD. Data stores can maintain keys in lexicographic order, allowing efficient retrieval of key ranges. Key-value stores can allow for storing of metadata with a value.

Key-value stores provide high performance and are often used for simple data models or for rapidly-changing data, such as an in-memory cache layer. Since they offer only a limited set of operations, complexity is shifted to the application layer if additional operations are needed.

A key-value store is the basis for more complex systems such as a document store, and in some cases, a graph database.

Source(s) and further reading: key-value store

- Key-value database
- Disadvantages of key-value stores
- Redis architecture
- Memcached architecture

Document store

Abstraction: key-value store with documents stored as values
A document store is centered around documents (XML, JSON, binary, etc), where a document stores all information for a given object. Document stores provide APIs or a query language to query based on the internal structure of the document itself. *Note, many key-value stores include features for working with a value's metadata, blurring the lines between these two storage types.*

Based on the underlying implementation, documents are organized by collections, tags, metadata, or directories. Although documents can be organized or grouped together, documents may have fields that are completely different from each other.

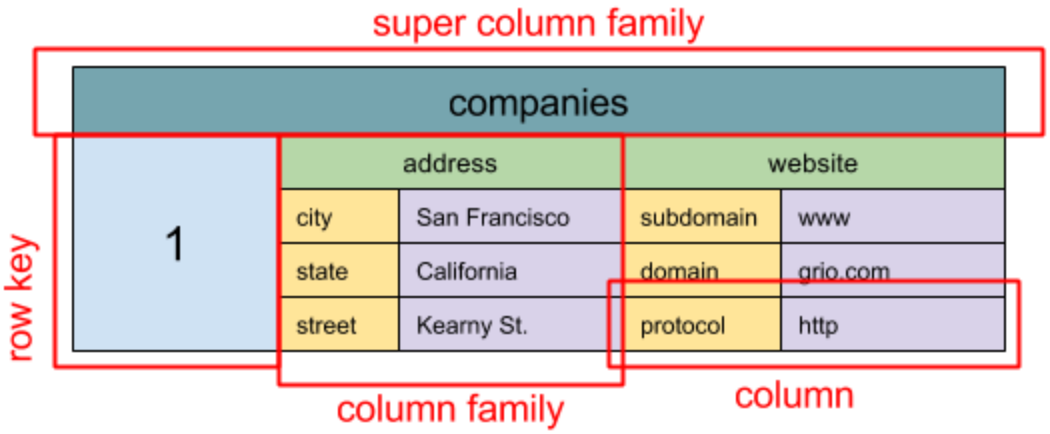
Some document stores like MongoDB and CouchDB also provide a SQL-like language to perform complex queries. DynamoDB supports both key-values and documents.

Document stores provide high flexibility and are often used for working with occasionally changing data.

Source(s) and further reading: document store

- Document-oriented database
- MongoDB architecture
- CouchDB architecture
- Elasticsearch architecture

Wide column store



Source: SQL & NoSQL, a brief history

Abstraction: nested map `ColumnFamily<RowKey, Columns<ColKey, Value, Timestamp>>`
A wide column store's basic unit of data is a column (name/value pair). A column can be grouped in column families (analogous to a SQL table). Super column families further group column families. You can access each column independently with a row key, and columns with the same row key form a row. Each value contains a timestamp for versioning and for conflict resolution.

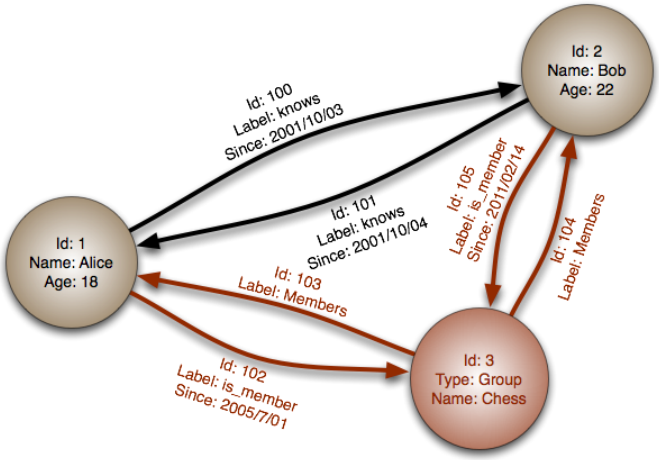
Google introduced Bigtable as the first wide column store, which influenced the open-source HBase often-used in the Hadoop ecosystem, and Cassandra from Facebook. Stores such as BigTable, HBase, and Cassandra maintain keys in lexicographic order, allowing efficient retrieval of selective key ranges.

Wide column stores offer high availability and high scalability. They are often used for very large data sets.

Source(s) and further reading: wide column store

- SQL & NoSQL, a brief history
- Bigtable architecture
- HBase architecture
- Cassandra architecture

Graph database



Source: Graph database

Abstraction: graph
In a graph database, each node is a record and each arc is a relationship between two nodes. Graph databases are optimized to represent complex relationships with many foreign keys or many-to-many relationships.

Graphs databases offer high performance for data models with complex relationships, such as a social network. They are relatively new and are not yet widely-used; it might be more difficult to find development tools and resources. Many graphs can only be accessed with REST APIs.

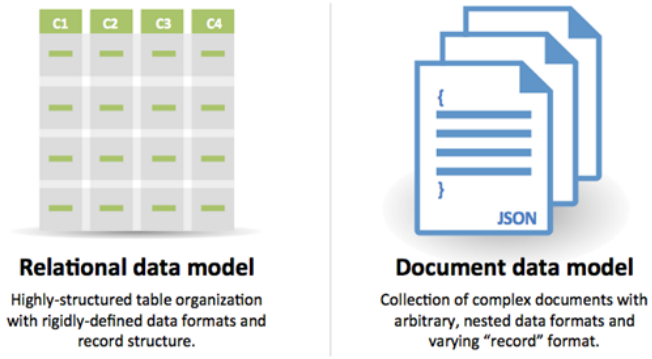
Source(s) and further reading: graph

- Graph database
- Neo4j
- FlockDB

Source(s) and further reading: NoSQL

- Explanation of base terminology
- NoSQL databases a survey and decision guidance
- Scalability
- Introduction to NoSQL
- NoSQL patterns

SQL or NoSQL



Source: Transitioning from RDBMS to NoSQL

Reasons for **SQL**:

- Structured data
- Strict schema
- Relational data
- Need for complex joins
- Transactions
- Clear patterns for scaling
- More established: developers, community, code, tools, etc
- Lookups by index are very fast

Reasons for **NoSQL**:

- Semi-structured data
- Dynamic or flexible schema
- Non-relational data
- No need for complex joins
- Store many TB (or PB) of data
- Very data intensive workload
- Very high throughput for IOPS

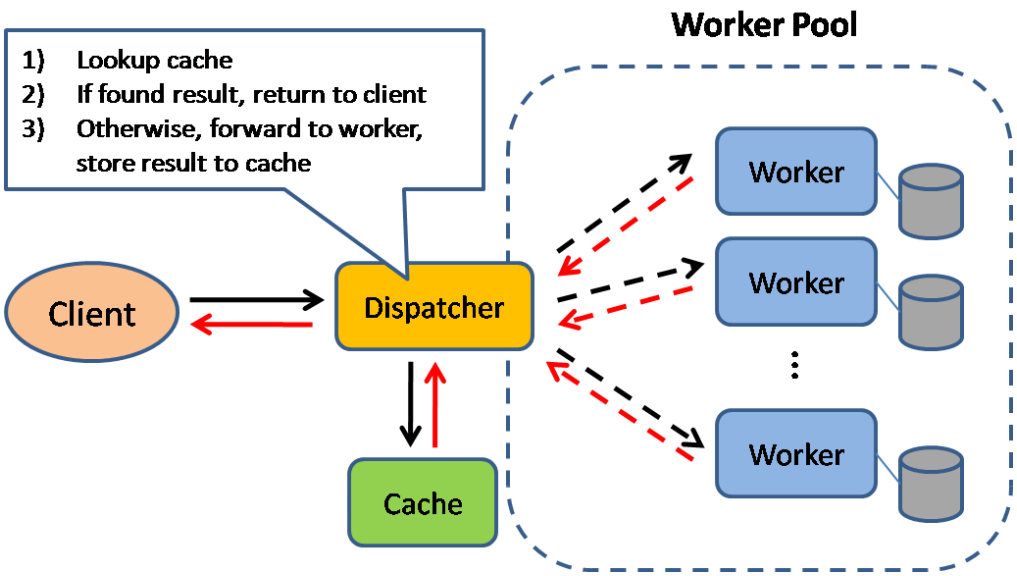
Sample data well-suited for NoSQL:

- Rapid ingest of clickstream and log data
- Leaderboard or scoring data
- Temporary data, such as a shopping cart
- Frequently accessed ('hot') tables
- Metadata/lookup tables

Source(s) and further reading: SQL or NoSQL

- Scaling up to your first 10 million users
- SQL vs NoSQL differences

Cache



Source: Scalable system design patterns

Caching improves page load times and can reduce the load on your servers and databases. In this model, the dispatcher will first lookup if the request has been made before and try to find the previous result to return, in order to save the actual execution.

Databases often benefit from a uniform distribution of reads and writes across its partitions. Popular items can skew the distribution, causing bottlenecks. Putting a cache in front of a database can help absorb uneven loads and spikes in traffic.

Client caching

Caches can be located on the client side (OS or browser), server side, or in a distinct cache layer.

CDN caching

CDNs are considered a type of cache.

Web server caching

Reverse proxies and caches such as Varnish can serve static and dynamic content directly. Web servers can also cache requests, returning responses without having to contact application servers.

Database caching

Your database usually includes some level of caching in a default configuration, optimized for a generic use case. Tweaking these settings for specific usage patterns can further boost performance.

Application caching

In-memory caches such as Memcached and Redis are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk. RAM is more limited than disk, so cache invalidation algorithms such as least recently used (LRU) can help invalidate 'cold' entries and keep 'hot' data in RAM.

Redis has the following additional features:

- Persistence option
- Built-in data structures such as sorted sets and lists

There are multiple levels you can cache that fall into two general categories: **database queries** and **objects**:

- Row level
- Query-level
- Fully-formed serializable objects
- Fully-rendered HTML

Generally, you should try to avoid file-based caching, as it makes cloning and auto-scaling more difficult.

Caching at the database query level

Whenever you query the database, hash the query as a key and store the result to the cache. This approach suffers from expiration issues:

- Hard to delete a cached result with complex queries

- If one piece of data changes such as a table cell, you need to delete all cached queries that might include the changed cell

Caching at the object level

See your data as an object, similar to what you do with your application code. Have your application assemble the dataset from the database into a class instance or a data structure(s):

- Remove the object from cache if its underlying data has changed
- Allows for asynchronous processing: workers assemble objects by consuming the latest cached object

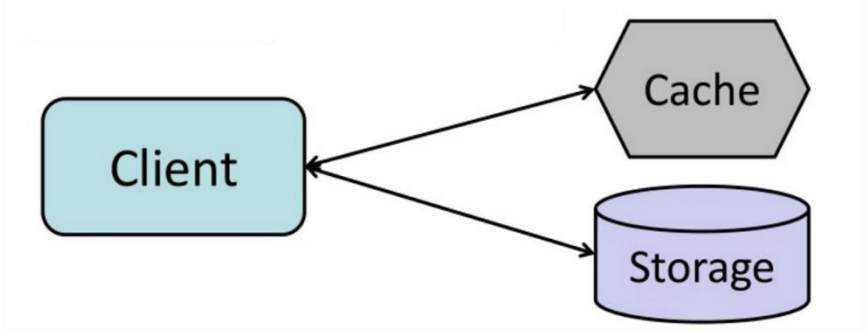
Suggestions of what to cache:

- User sessions
- Fully rendered web pages
- Activity streams
- User graph data

When to update the cache

Since you can only store a limited amount of data in cache, you'll need to determine which cache update strategy works best for your use case.

Cache-aside



Source: From cache to in-memory data grid

The application is responsible for reading and writing from storage. The cache does not interact with storage directly. The application does the following:

- Look for entry in cache, resulting in a cache miss
- Load entry from the database
- Add entry to cache
- Return entry

```
def get_user(self, user_id):
```

```
    user = cache.get("user.{0}", user_id)
    if user is None:
        user = db.query("SELECT * FROM users WHERE user_id = {0}", user_id)
        if user is not None:
            key = "user.{0}".format(user_id)
            cache.set(key, json.dumps(user))
    return user
```

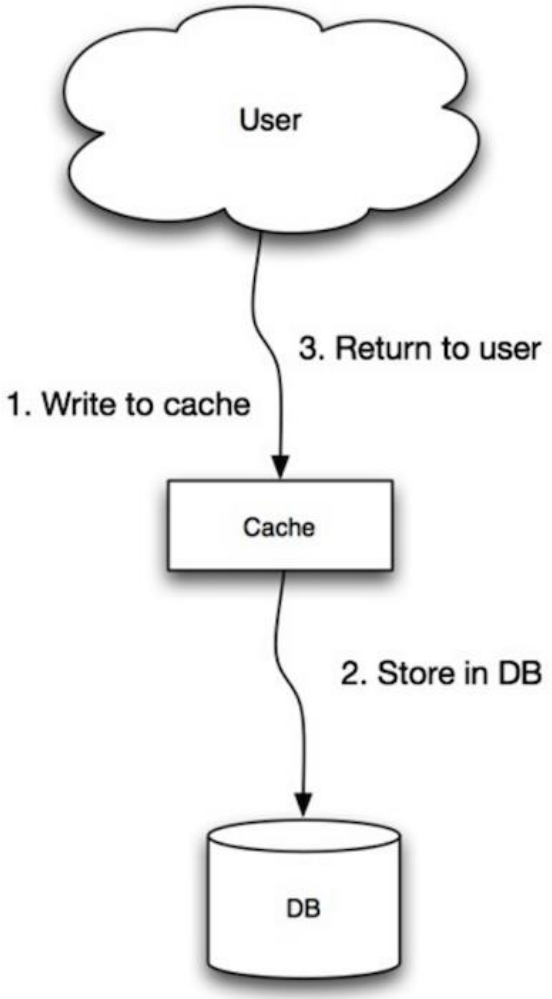
Memcached is generally used in this manner.

Subsequent reads of data added to cache are fast. Cache-aside is also referred to as lazy loading. Only requested data is cached, which avoids filling up the cache with data that isn't requested.

Disadvantage(s): cache-aside

- Each cache miss results in three trips, which can cause a noticeable delay.
- Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through.
- When a node fails, it is replaced by a new, empty node, increasing latency.

Write-through



Source: Scalability, availability, stability, patterns

The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:

- Application adds/updates entry in cache
- Cache synchronously writes entry to data store
- Return

Application code:

```
set_user(12345, {"foo":"bar"})
```

Cache code:

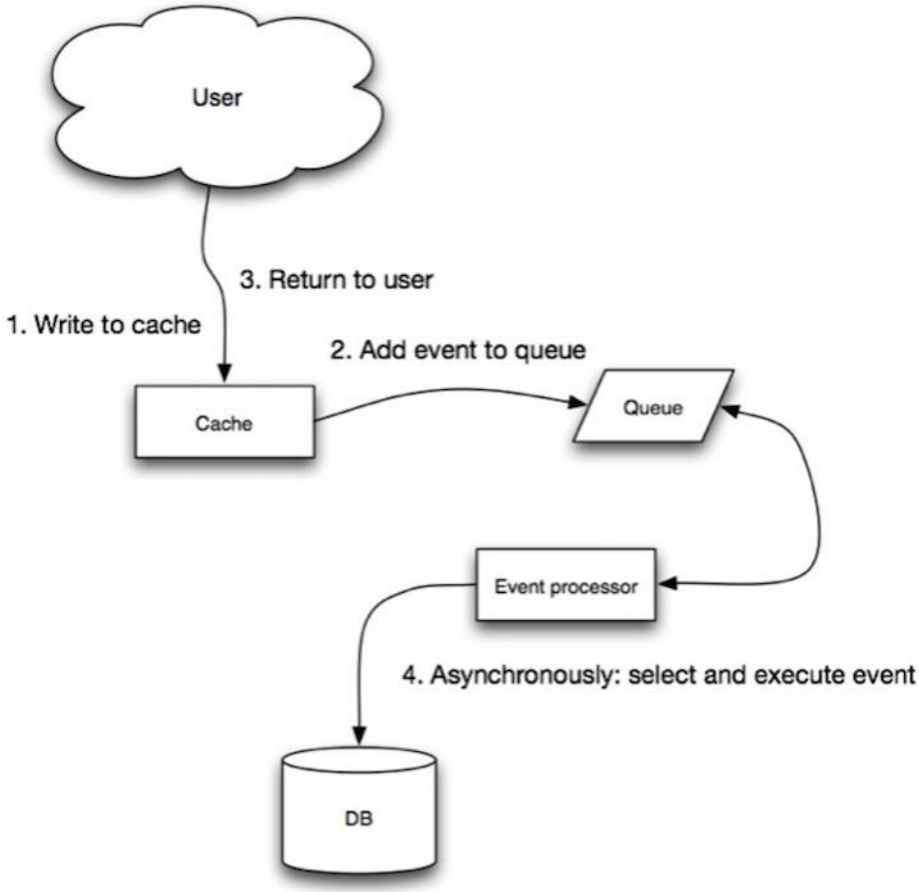
```
def set_user(user_id, values):
    user = db.query("UPDATE Users WHERE id = {0}", user_id, values)
    cache.set(user_id, user)
```

Write-through is a slow overall operation due to the write operation, but subsequent reads of just written data are fast. Users are generally more tolerant of latency when updating data than reading data. Data in the cache is not stale.

Disadvantage(s): write through

- When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database. Cache-aside in conjunction with write through can mitigate this issue.
- Most data written might never be read, which can be minimized with a TTL.

Write-behind (write-back)



Source: Scalability, availability, stability, patterns

In write-behind, the application does the following:

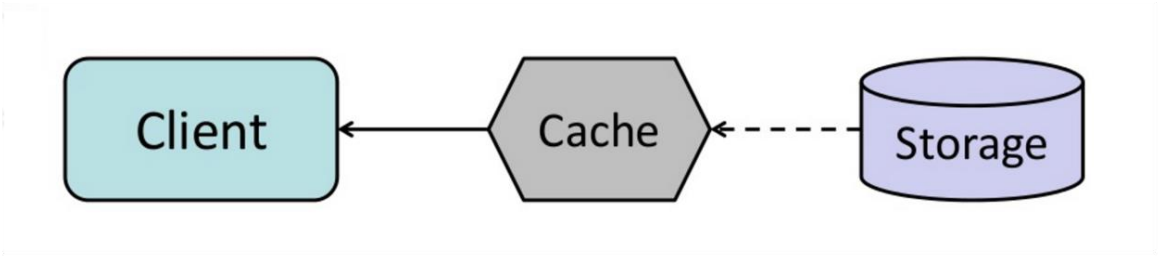
- Add/update entry in cache
- Asynchronously write entry to the data store, improving write performance

Disadvantage(s): write-behind

- There could be data loss if the cache goes down prior to its contents hitting the data store.

- It is more complex to implement write-behind than it is to implement cache-aside or write-through.

Refresh-ahead



Source: From cache to in-memory data grid

You can configure the cache to automatically refresh any recently accessed cache entry prior to its expiration.

Refresh-ahead can result in reduced latency vs read-through if the cache can accurately predict which items are likely to be needed in the future.

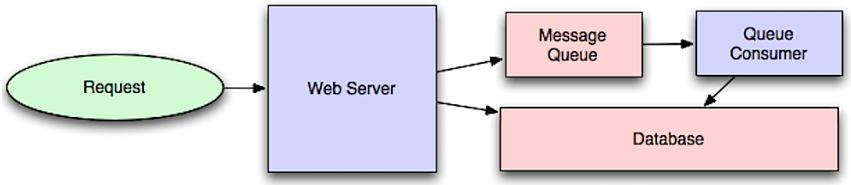
Disadvantage(s): refresh-ahead

- Not accurately predicting which items are likely to be needed in the future can result in reduced performance than without refresh-ahead.

Disadvantage(s): cache

- Need to maintain consistency between caches and the source of truth such as the database through cache invalidation.
- Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
- Need to make application changes such as adding Redis or memcached.

Asynchronism



Source: Intro to architecting systems for scale

Asynchronous workflows help reduce request times for expensive operations that would otherwise be performed in-line. They can also help by doing time-consuming work in advance, such as periodic aggregation of data.

Message queues

Message queues receive, hold, and deliver messages. If an operation is too slow to perform inline, you can use a message queue with the following workflow:

- An application publishes a job to the queue, then notifies the user of job status
- A worker picks up the job from the queue, processes it, then signals the job is complete

The user is not blocked and the job is processed in the background. During this time, the client might optionally do a small amount of processing to make it seem like the task has completed. For example, if posting a tweet, the tweet could be instantly posted to your timeline, but it could take some time before your tweet is actually delivered to all of your followers.

Redis is useful as a simple message broker but messages can be lost.

RabbitMQ is popular but requires you to adapt to the 'AMQP' protocol and manage your own nodes.

Amazon SQS is hosted but can have high latency and has the possibility of messages being delivered twice.

Task queues

Tasks queues receive tasks and their related data, runs them, then delivers their results. They can support scheduling and can be used to run computationally-intensive jobs in the background.

Celery has support for scheduling and primarily has python support.

Back pressure

If queues start to grow significantly, the queue size can become larger than memory, resulting in cache misses, disk reads, and even slower performance. Back pressure can help by limiting the queue size, thereby maintaining a high throughput rate and good response times for jobs already in the queue. Once the queue fills up, clients get a server busy or HTTP 503 status code to try again later. Clients can retry the request at a later time, perhaps with exponential backoff.

Disadvantage(s): asynchronism

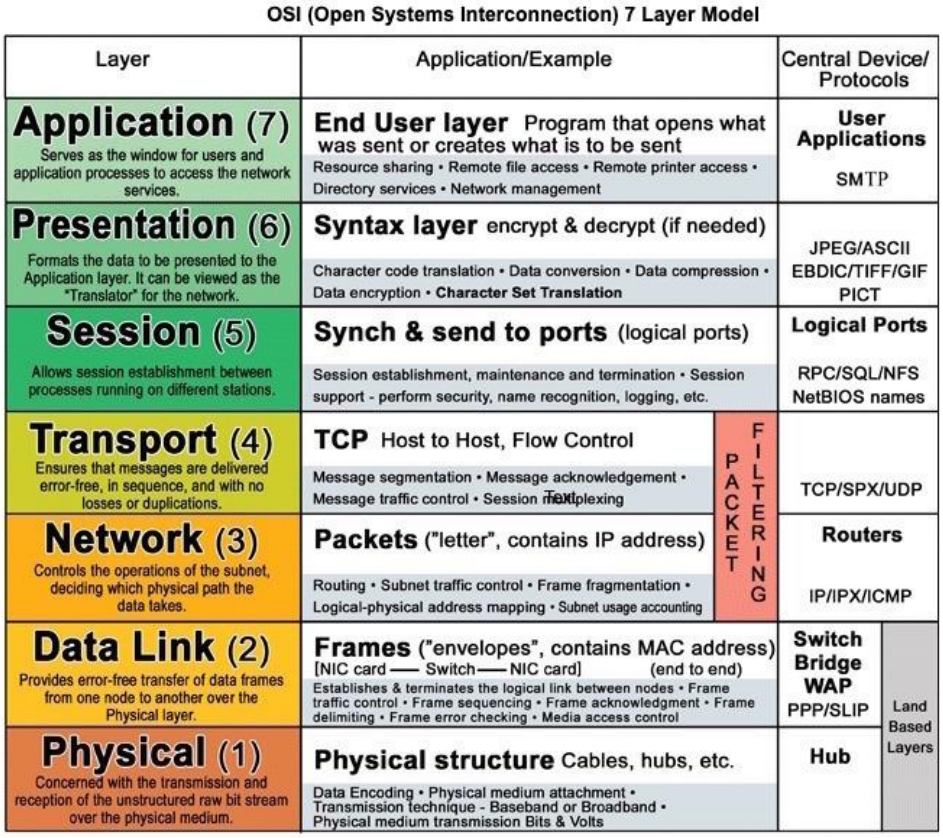
- Use cases such as inexpensive calculations and realtime workflows might be better suited for synchronous operations, as introducing queues can add delays and complexity.

Source(s) and further reading

- It's all a numbers game
- Applying back pressure when overloaded

- Little's law
- What is the difference between a message queue and a task queue?

Communication



Source: OSI 7 layer model

Hypertext transfer protocol (HTTP)

HTTP is a method for encoding and transporting data between a client and a server. It is a request/response protocol: clients issue requests and servers issue responses with relevant content and completion status info about the request. HTTP is self-contained, allowing requests and responses to flow through many intermediate routers and servers that perform load balancing, caching, encryption, and compression.

A basic HTTP request consists of a verb (method) and a resource (endpoint). Below are common HTTP verbs:

Verb	Description	Idempotent*	Safe	Cacheable
GET	Reads a resource	Yes	Yes	Yes
POST	Creates a resource or trigger a process that handles data	No	No	Yes if response contains freshness info
PUT	Creates or replace a resource	Yes	No	No
PATCH	Partially updates a resource	No	No	Yes if response contains freshness info
DELETE	Deletes a resource	Yes	No	No

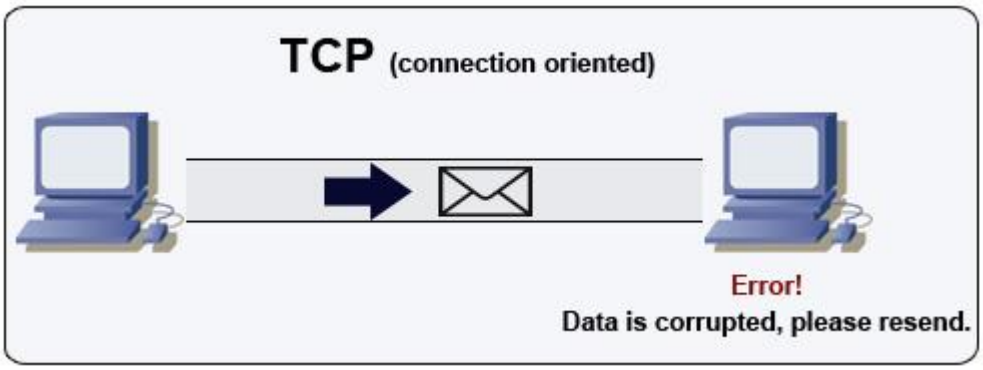
*Can be called many times without different outcomes.

HTTP is an application layer protocol relying on lower-level protocols such as **TCP** and **UDP**.

Source(s) and further reading: HTTP

- What is HTTP?
- Difference between HTTP and TCP
- Difference between PUT and PATCH

Transmission control protocol (TCP)



Source: How to make a multiplayer game

TCP is a connection-oriented protocol over an IP network. Connection is established and terminated using a handshake. All packets sent are guaranteed to reach the destination in the original order and without corruption through:

- Sequence numbers and checksum fields for each packet

- Acknowledgement packets and automatic retransmission

If the sender does not receive a correct response, it will resend the packets. If there are multiple timeouts, the connection is dropped. TCP also implements flow control and congestion control. These guarantees cause delays and generally result in less efficient transmission than UDP.

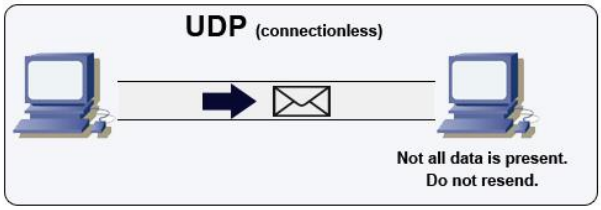
To ensure high throughput, web servers can keep a large number of TCP connections open, resulting in high memory usage. It can be expensive to have a large number of open connections between web server threads and say, a memcached server. Connection pooling can help in addition to switching to UDP where applicable.

TCP is useful for applications that require high reliability but are less time critical. Some examples include web servers, database info, SMTP, FTP, and SSH.

Use TCP over UDP when:

- You need all of the data to arrive intact
- You want to automatically make a best estimate use of the network throughput

User datagram protocol (UDP)



Source: How to make a multiplayer game

UDP is connectionless. Datagrams (analogous to packets) are guaranteed only at the datagram level. Datagrams might reach their destination out of order or not at all. UDP does not support congestion control. Without the guarantees that TCP support, UDP is generally more efficient.

UDP can broadcast, sending datagrams to all devices on the subnet. This is useful with DHCP because the client has not yet received an IP address, thus preventing a way for TCP to stream without the IP address.

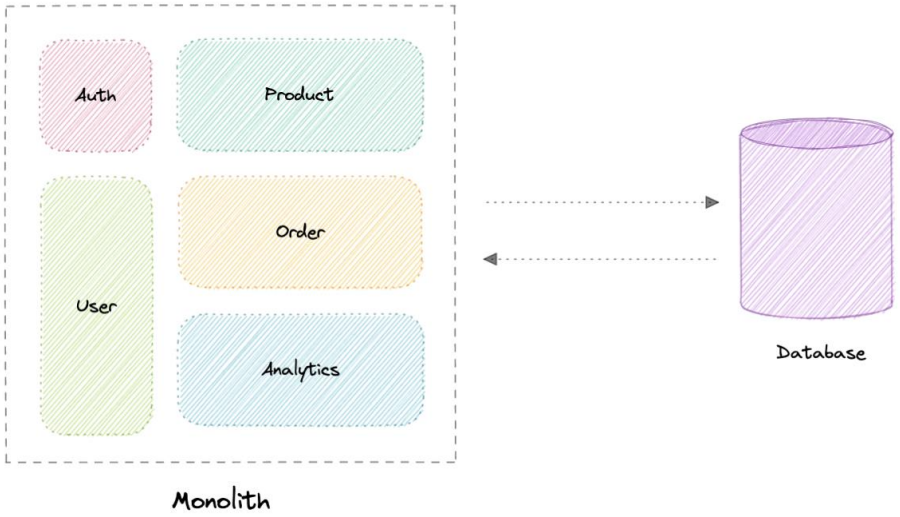
UDP is less reliable but works well in real time use cases such as VoIP, video chat, streaming, and realtime multiplayer games.

Use UDP over TCP when:

- You need the lowest latency
- Late data is worse than loss of data
- You want to implement your own error correction

Monoliths

A monolith is a self-contained and independent application. It is built as a single unit and is responsible for not just a particular task, but can perform every step needed to satisfy a business need.



Advantages

Following are some advantages of monoliths:

- Simple to develop or debug.
- Fast and reliable communication.
- Easy monitoring and testing.
- Supports ACID transactions.

Disadvantages

Some common disadvantages of monoliths are:

- Maintenance becomes hard as the codebase grows.
- Tightly coupled application, hard to extend.
- Requires commitment to a particular technology stack.
- On each update, the entire application is redeployed.

- Reduced reliability as a single bug can bring down the entire system.
- Difficult to scale or adopt technologies new technologies.

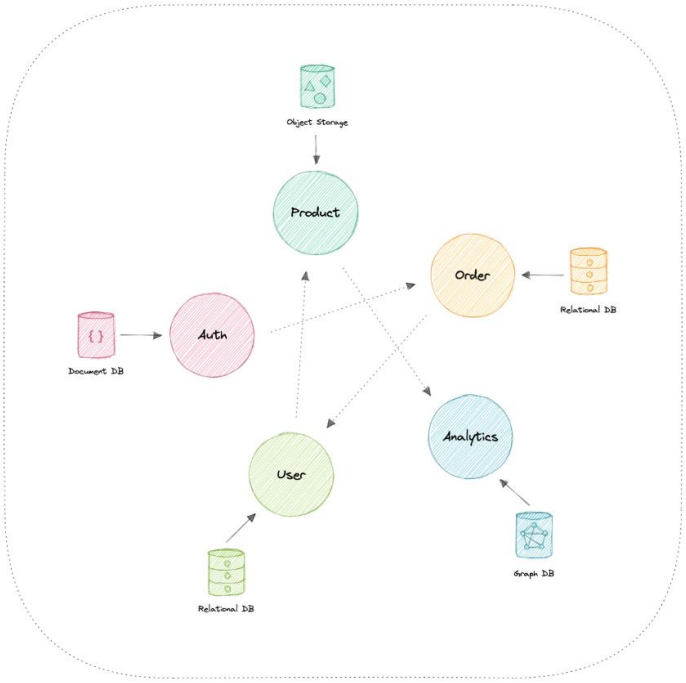
Modular monoliths

A Modular Monolith is an approach where we build and deploy a single application (that's the *Monolith* part), but we build it in a way that breaks up the code into independent modules for each of the features needed in our application.

This approach reduces the dependencies of a module in such as way that we can enhance or change a module without affecting other modules. When done right, this can be really beneficial in the long term as it reduces the complexity that comes with maintaining a monolith as the system grows.

Microservices

A microservices architecture consists of a collection of small, autonomous services where each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division of business logic that provides an explicit boundary within which a domain model exists.



Microservices

Each service has a separate codebase, which can be managed by a small development team. Services can be deployed independently and a team can update an existing service without rebuilding and redeploying the entire application.

Services are responsible for persisting their own data or external state (database per service). This differs from the traditional model, where a separate data layer handles data persistence.

Characteristics

The microservices architecture style has the following characteristics:

- **Loosely coupled**: Services should be loosely coupled so that they can be independently deployed and scaled. This will lead to the decentralization of development teams and thus, enabling them to develop and deploy faster with minimal constraints and operational dependencies.
- **Small but focused**: It's about scope and responsibilities and not size, a service should be focused on a specific problem. Basically, "*It does one thing and does it well*". Ideally, they can be independent of the underlying architecture.
- **Built for businesses**: The microservices architecture is usually organized around business capabilities and priorities.
- **Resilience & Fault tolerance**: Services should be designed in such a way that they still function in case of failure or errors. In environments with independently deployable services, failure tolerance is of the highest importance.
- **Highly maintainable**: Service should be easy to maintainable and test because services that cannot be maintained will be re-written.

Advantages

Here are some advantages of microservices architecture:

- Loosely coupled services.
- Services can be deployed independently.
- Highly agile for multiple development teams.
- Improves fault tolerance and data isolation.
- Better scalability as each service can be scaled independently.
- Eliminates any long-term commitment to a particular technology stack.

Disadvantages

Microservices architecture brings its own set of challenges:

- Complexity of a distributed system.
- Testing is more difficult.
- Expensive to maintain (individual servers, databases, etc.).
- Inter-service communication has its own challenges.
- Data integrity and consistency.
- Network congestion and latency.

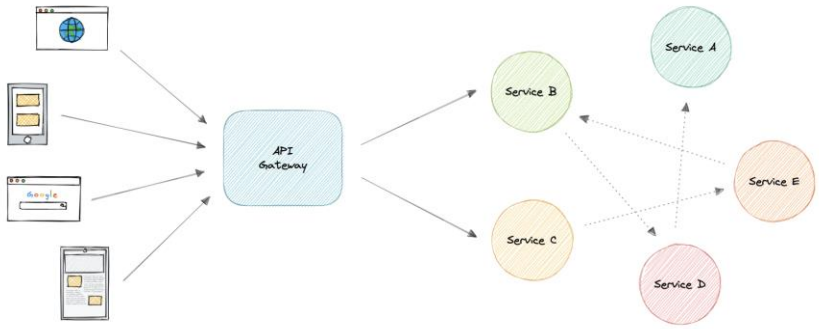
Best practices

Let's discuss some microservices best practices:

- Model services around the business domain.
- Services should have loose coupling and high functional cohesion.
- Isolate failures and use resiliency strategies to prevent failures within a service from cascading.
- Services should only communicate through well-designed APIs. Avoid leaking implementation details.
- Data storage should be private to the service that owns the data
- Avoid coupling between services. Causes of coupling include shared database schemas and rigid communication protocols.
- Decentralize everything. Individual teams are responsible for designing and building services. Avoid sharing code or data schemas.
- Fail fast by using a circuit breaker to achieve fault tolerance.
- Ensure that the API changes are backward compatible.

API Gateway

The API Gateway is an API management tool that sits between a client and a collection of backend services. It is a single entry point into a system that encapsulates the internal system architecture and provides an API that is tailored to each client. It also has other responsibilities such as authentication, monitoring, load balancing, caching, throttling, logging, etc.



Why do we need an API Gateway?

The granularity of APIs provided by microservices is often different than what a client needs. Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services. Hence, an API gateway can provide a single entry point for all clients with some additional features and better management.

Features

Below are some desired features of an API Gateway:

- Authentication and Authorization
- Service discovery
- Reverse Proxy
- Caching
- Security
- Retry and Circuit breaking
- Load balancing
- Logging, Tracing
- API composition
- Rate limiting and throttling
- Versioning
- Routing
- IP whitelisting or blacklisting

Advantages

Let's look at some advantages of using an API Gateway:

- Encapsulates the internal structure of an API.
- Provides a centralized view of the API.
- Simplifies the client code.
- Monitoring, analytics, tracing, and other such features.

SLA, SLO, SLI

Let's briefly discuss SLA, SLO, and SLI. These are mostly related to the business and site reliability side of things but good to know nonetheless.

Why are they important?

SLAs, SLOs, and SLIs allow companies to define, track and monitor the promises made for a service to its users. Together, SLAs, SLOs, and SLIs should help teams generate more user trust in their services with an added emphasis on continuous improvement to incident management and response processes.

SLA

An SLA, or Service Level Agreement, is an agreement made between a company and its users of a given service. The SLA defines the different promises that the company makes to users regarding specific metrics, such as service availability.

SLAs are often written by a company's business or legal team.

SLO

An SLO, or Service Level Objective, is the promise that a company makes to users regarding a specific metric such as incident response or uptime. SLOs exist within an SLA as individual promises contained within the full user agreement. The SLO is the specific goal that the service must meet in order to comply with the SLA. SLOs should always be simple, clearly defined, and easily measured to determine whether or not the objective is being fulfilled.

SLI

An SLI, or Service Level Indicator, is a key metric used to determine whether or not the SLO is being met. It is the measured value of the metric described within the SLO. In order to remain in compliance with the SLA, the SLI's value must always meet or exceed the value determined by the SLO.

Disaster recovery

Disaster recovery (DR) is a process of regaining access and functionality of the infrastructure after events like a natural disaster, cyber attack, or even business disruptions.

Disaster recovery relies upon the replication of data and computer processing in an off-premises location not affected by the disaster. When servers go down because of a disaster, a business needs to recover lost data from a second location where the data is backed up. Ideally, an organization can transfer its computer processing to that remote location as well in order to continue operations.

Disaster Recovery is often not actively discussed during system design interviews but it's important to have some basic understanding of this topic. You can learn more about disaster recovery from AWS Well-Architected Framework.

Why is disaster recovery important?

Disaster recovery can have the following benefits:

- Minimize interruption and downtime
- Limit damages
- Fast restoration
- Better customer retention

Terms

Let's discuss some important terms relevantly for disaster recovery:

RTO

Recovery Time Objective (RTO) is the maximum acceptable delay between the interruption of service and restoration of service. This determines what is considered an acceptable time window when service is unavailable.

RPO

Recovery Point Objective (RPO) is the maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

Strategies

A variety of disaster recovery (DR) strategies can be part of a disaster recovery plan.

Back-up

This is the simplest type of disaster recovery and involves storing data off-site or on a removable drive.

Cold Site

In this type of disaster recovery, an organization sets up basic infrastructure in a second site.

Hot site

A hot site maintains up-to-date copies of data at all times. Hot sites are time-consuming to set up and more expensive than cold sites, but they dramatically reduce downtime.