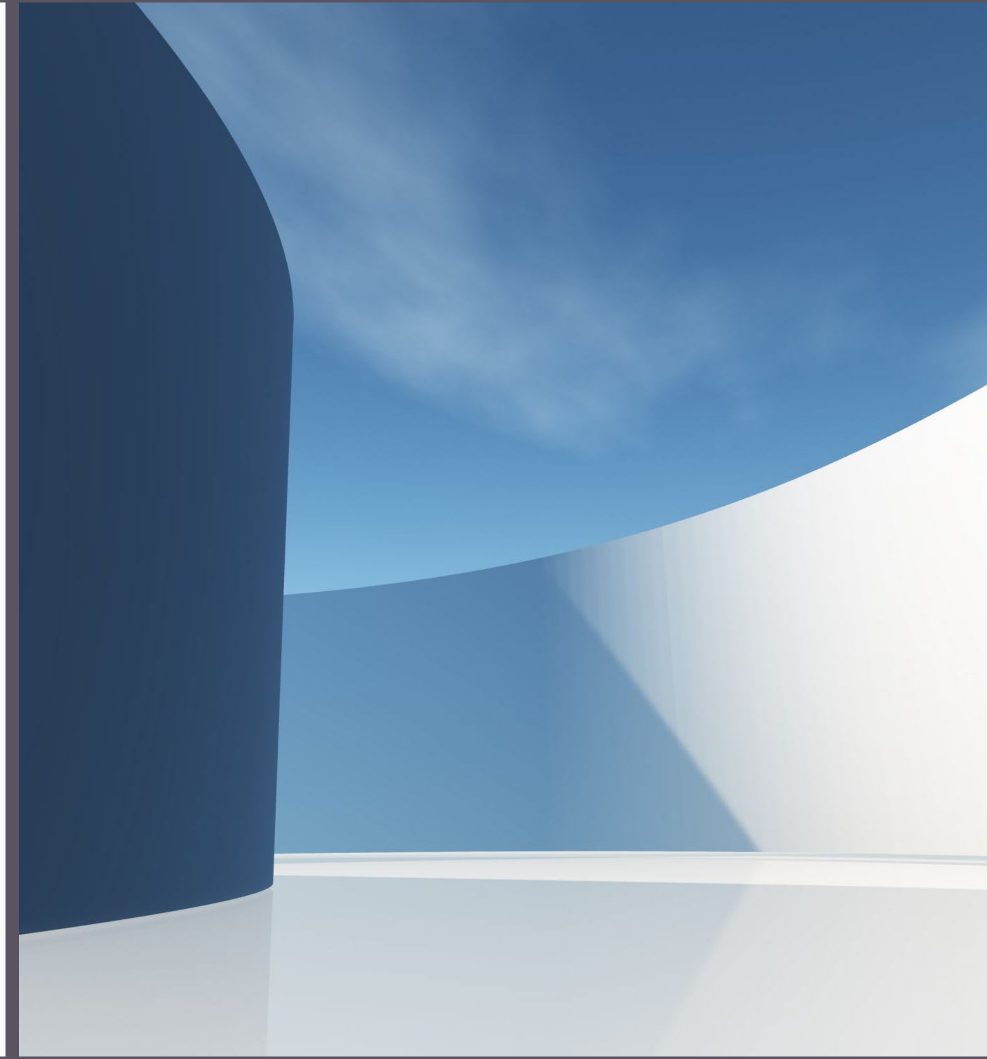
DATA STORAGE

Types, uses, optimization, comparison



Database

Collection of structured data or information.

Easily accessible in a number of ways.

Typically used to support Online Transaction Processing (OLTP).

Database

Typically support transactional systems:

Sales

- Bookings/reservations
- HR - payroll, employee management
- Delivery/Shipping
- Purchasing/stock management

Databases

Typical Use cases:

Automating business processes

Creating reports for financial and other data

Analyzing relatively small datasets

OLTP V OLAP

	OLTP	OLAP
Query types	Simple standardized queries	Complex queries
Operations	Based on INSERT, UPDATE, DELETE commands	Based on SELECT commands to aggregate data for reporting
Response time	Milliseconds	Seconds, minutes, or hours depending on the amount of data to process
Design	Industry-specific, such as retail, manufacturing, or banking	Subject-specific, such as sales, inventory, or marketing
Source	Transactions (typically related to users)	Aggregated data from transactions
Purpose	Control and run essential business operations in real time	Plan, solve problems, support decisions, discover hidden insights

OLTP V OLAP

	OLTP	OLAP
Data updates	Short, fast updates initiated by user/organisational events	Data periodically refreshed with scheduled, long-running batch jobs
Backup and recovery	Regular backups required to ensure business continuity and meet legal and governance requirements	Lost data can be reloaded from OLTP database as needed in lieu of regular backups
Productivity	Increases productivity of customers and frontline organisational workers	Increases productivity of managers, analysts, and executives
Data view	Day-to-day business transactions	Multi-dimensional view of enterprise data
Database design	Normalized databases for efficiency	Denormalized databases for analysis

Data Warehouse

Stores highly structured information Typically store current and historical data from one or more sources

Goal is to combine data from different data sources to support data analysis and business intelligence e.g. look for insights, and create reports and visualisations, support machine learning

Data Warehouse

Stores large amounts of current and historical data from various sources

Typically has a pre-defined and fixed relational schema.

Work well with structured data but some. Some data warehouses also support semi-structured data.

Used to for data-driven decision making.

Can be used with business intelligence tools to explore the data, look for insights, and generate reports and visualisations.

Data Warehouse

Used to for data-driven decision making.

Can be used with business intelligence tools to explore the data, look for insights, and generate reports and visualisations.

Specific, static structures/schemas dictate what data analysis you could perform

Data Warehouse

Contain a range of data, from raw ingested data to highly curated, cleansed, filtered, and aggregated data.

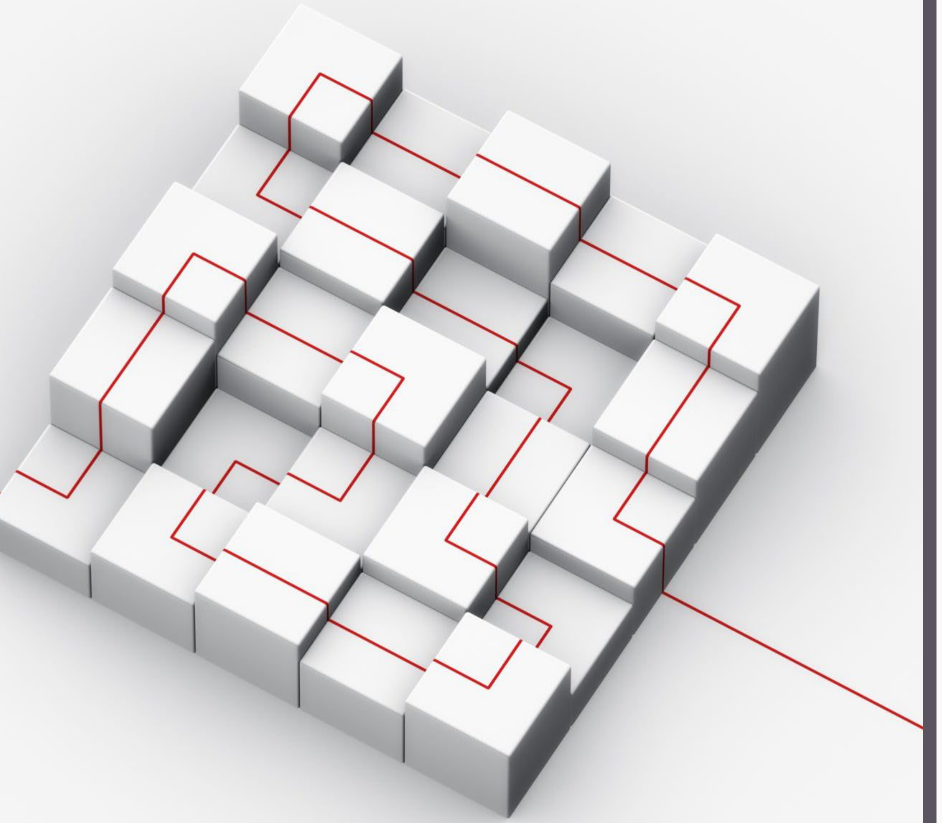
Extract, transform, load (ETL) processes move data from its original source to the data warehouse.

ETL needs to be scheduled so that data is updated but the data in the data warehouse may not reflect the most up-to-date state of the systems from which data is taken.

Data Warehouse

Can be considered a very large database.

Not intended to satisfy the transaction and concurrency needs typical of an application using a database.



Data Warehouse

Before data can be loaded into a data warehouse it must have some shape and structure—in other words, a model.

The process of giving data some shape and structure is called schema-on-write.

A database also uses the schema-on-write approach.

Data Lake

Repository of data from disparate sources

Data is stored in its original, raw format

Store large amounts of current and historical data

Can store structured, semi-structured, and unstructured data.

Can store data in a variety of formats including JSON, BSON, CSV, TSV, Avro, ORC, and Parquet.

Flexibility is key.

Data Lake

Primary purpose of a data lake is to analyze the data to gain insights.

Organizations may use data lakes simply for their cheap storage with the idea that the data may be used for analytics in the future.

Data Lake

Data is loaded in its raw form.

When a user/application needs to use data, a shape and structure has to be provided (by the user/application).

This is called schema-on-read.

Comparison

A database stores the current data required to power an application.

A data warehouse stores current and historical data from one or more systems in a predefined and fixed schema, which allows business analysts and data scientists to easily analyze the data.

A data lake stores current and historical data from one or more systems in its raw form, which allows business analysts and data scientists to easily analyze the data.

Comparison

	Database	Data Warehouse	Data Lake
Workload	OLTP	OLAP	OLAP
Data	Structured and/or semi-structured	Structured and/or semi-structured	Structured, semi-structured, and/or unstructured
Schema Flexibility	Depends on database type (primarily schema-on-write)	Pre-defined and fixed schema definition (schema on write and read)	Schema-on-Read
Data Freshness	Real-time	Dependent on ETL schedule	Dependent on ETL schedule
Users	Anyone	Managers, business analysts, data scientists, application developers	Data Scientists, application developers
Use Cases	Automation, Reporting, Basic Analysis	Complex Analysis, Reporting, Machine Learning	Data Science, Research and Testing
Pros	Fast queries and updates	Makes data easy to work with for analytical processes	Simpler load Application of schema on read provides flexibility for working with data in multiple analytical processes Separate storage and compute phases
Cons	Analytics may be limited	Effort is needed to prepare and organise data for analytics	Effort is needed to use and process the data

Hot v Cold v Warm Data

Levels are determined by how important the data is to an organisation and how frequently it is access

Temperature “cold” and “hot” refer to where the data was located in traditional file based architectures

Hot Data

Data that requires frequent access instantly.

Any piece of information crucial for your business and needs to be retrieved regularly is deemed fit for hot storage.

Most recent translational data and used for current reporting purposes.

Hot Data

Examples:

Data that transforms at a faster pace

Data used for querying customer requests

Data used in the latest real-time projects

Cold Data

Less frequently accessed data that does not require instant access like hot data.

Consists of information that is no longer active and is not relevant.

Cold Data

Examples:

Data for ad-hoc reporting.

Data maintained for regulatory reasons and audit purposes.

Warm Data

A balance between hot and cold

Could be data that is used for reporting or analytics.

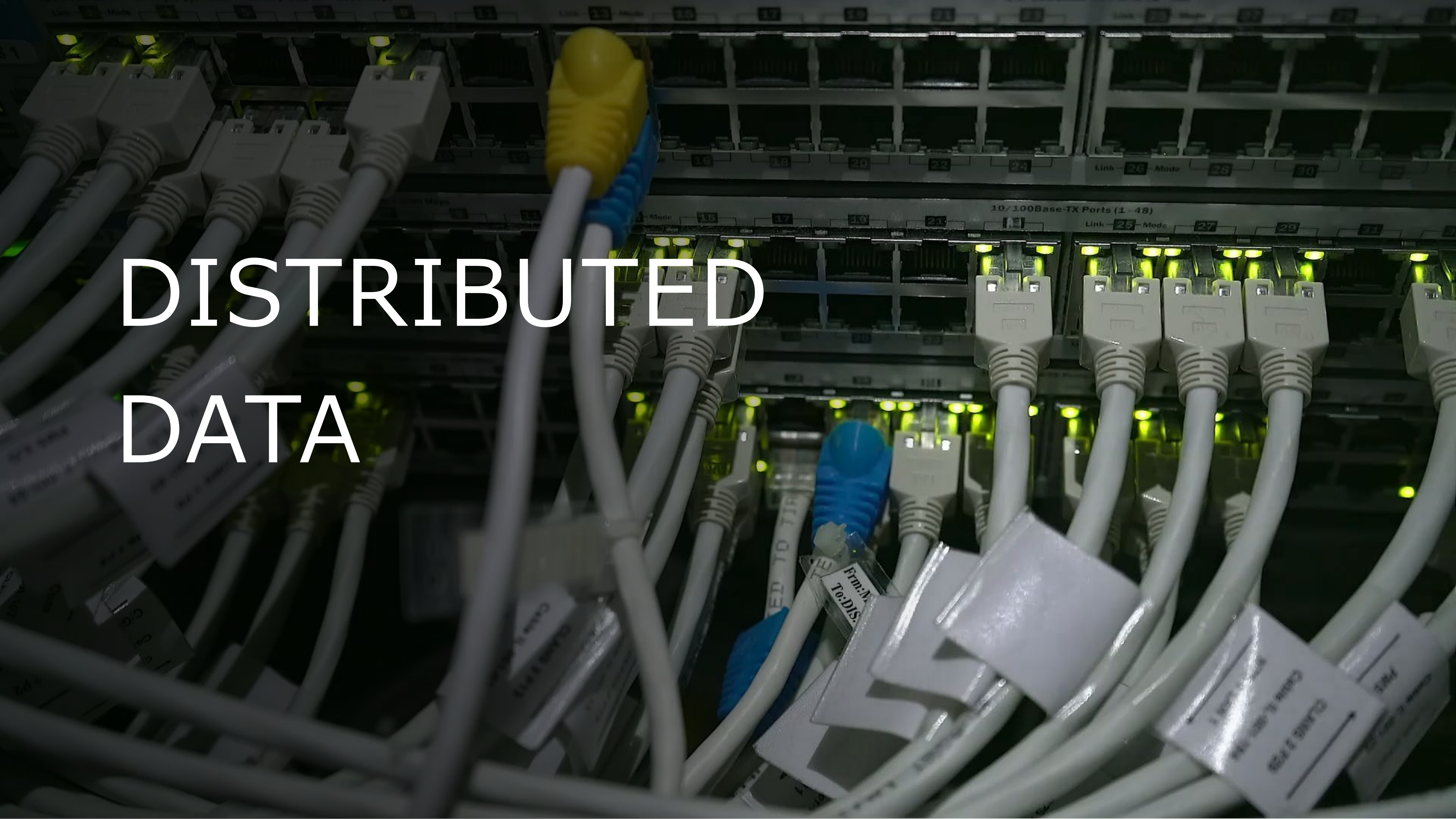
Still needs to be accessed quickly but it doesn’t need to be accessed as quickly as hot data.

Warm Data

Examples

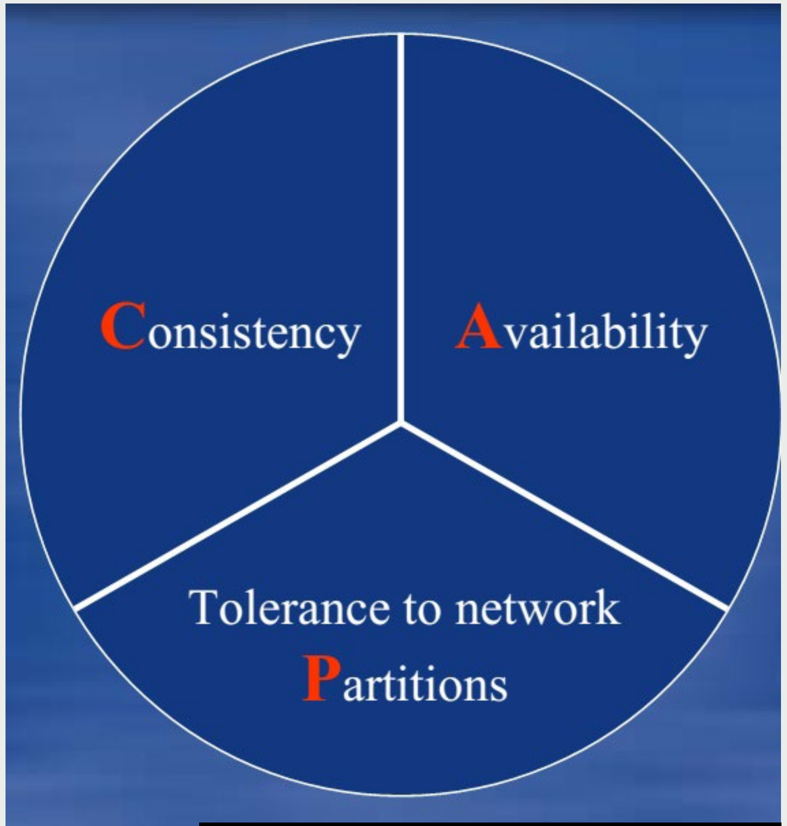
Reference tables data

Last quarters transactional data



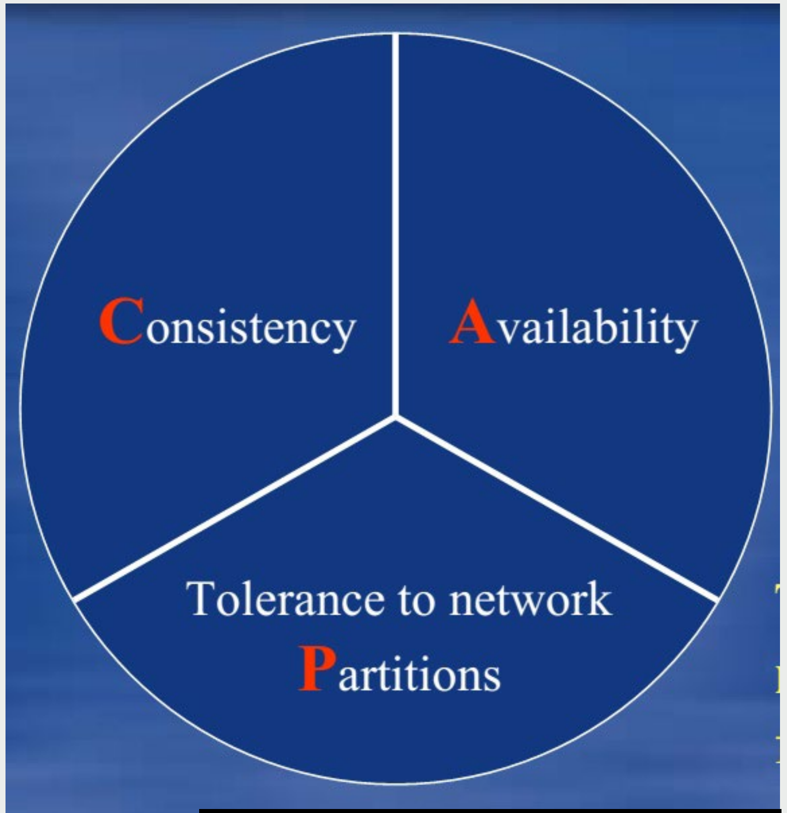
DISTRIBUTED DATA

CAP Theorem



About the tradeoffs inherent in designing a distributed system/distributed solution for storing data.

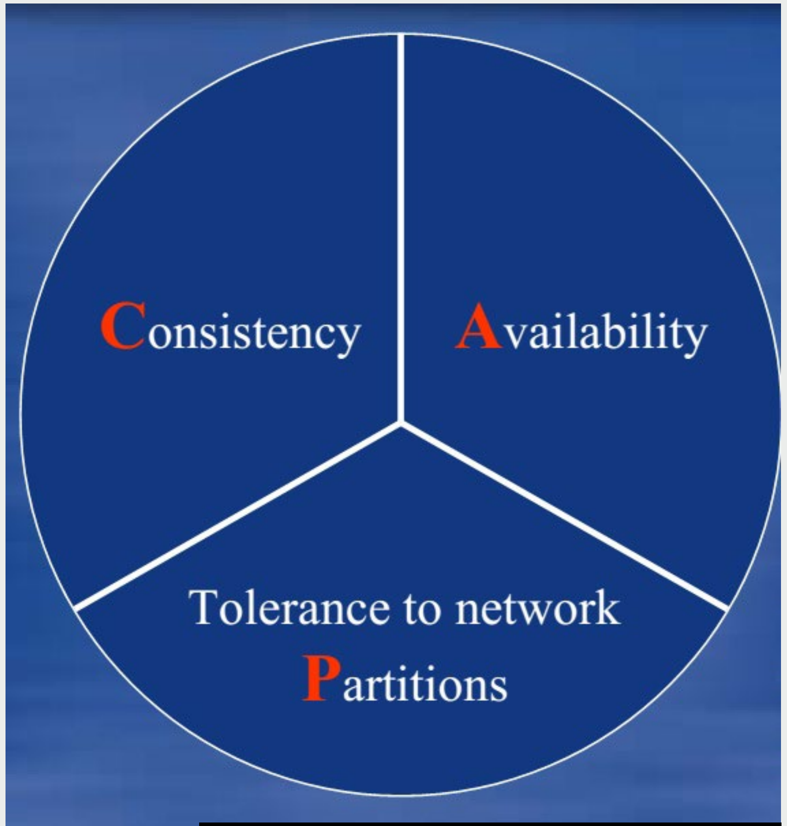
CAP Theorem



Consistency:

Sequential consistency (a data item behaves as if there is one copy) – similar to the A in ACID

All clients connected to a data store see the same data.

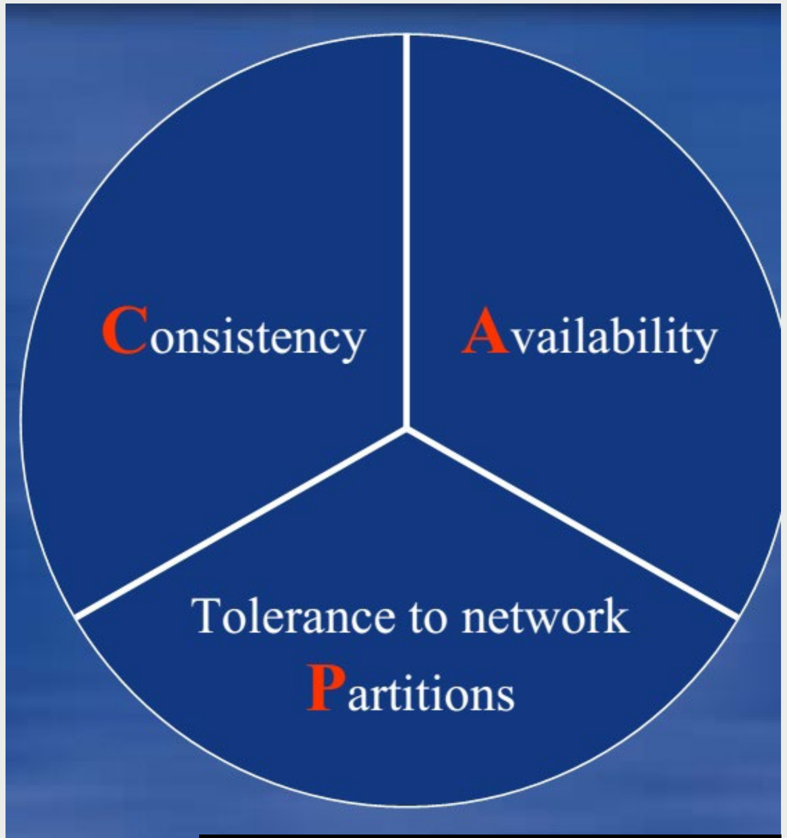


CAP Theorem

Availability:

Node failures do not prevent survivors from continuing to operate.

Clients are able to access and update data rapidly.



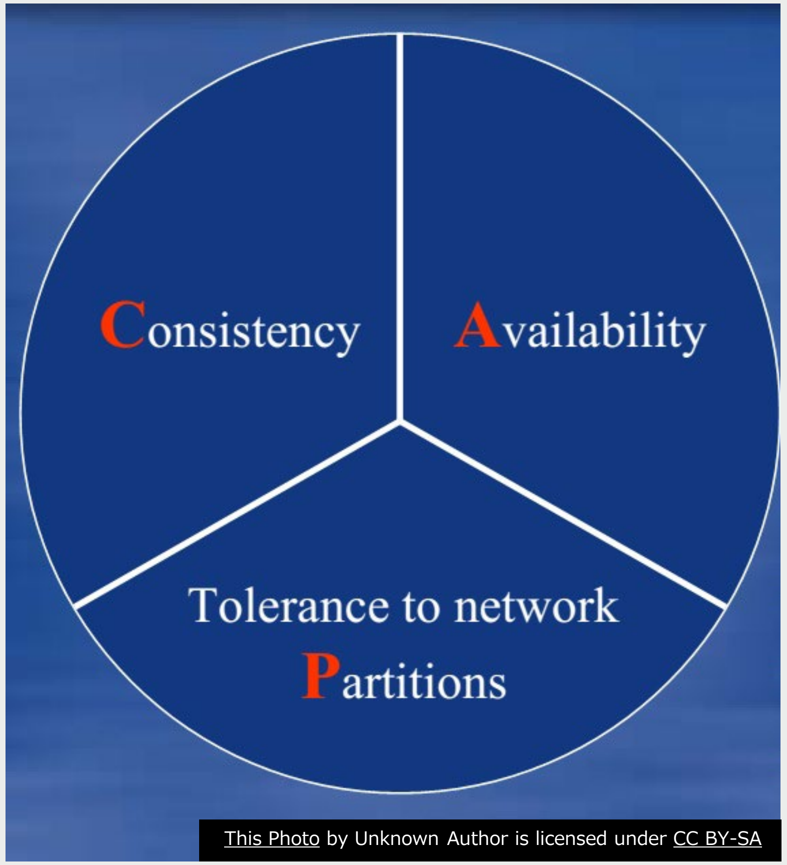
CAP Theorem

Partition-tolerance:

Partition=Break

The system continues to operate even if a network partition causes communication interruption between nodes.

The data store is able to operate even when the network fails in some way.



CAP Theorem

"A distributed system can satisfy any two of these guarantees at the same time but not all three"

CAP Theorem

Misconception:

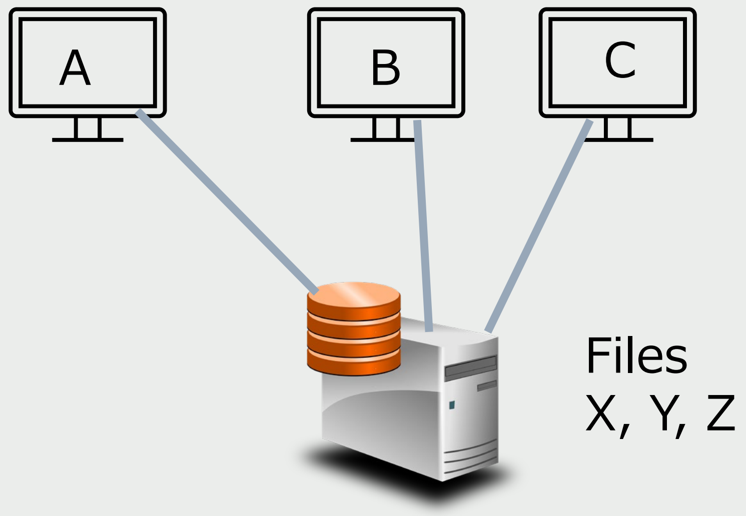
For a distributed application/data store you have to “pick two of the three”

CAP Theorem

Each of these properties exists on a continuum.

Attempting to increase one requires decreasing another to a certain degree.

Example



Suppose we have a server S that contains some files named X, Y, and Z.

The server is accessible via a network to clients A, B, and C.

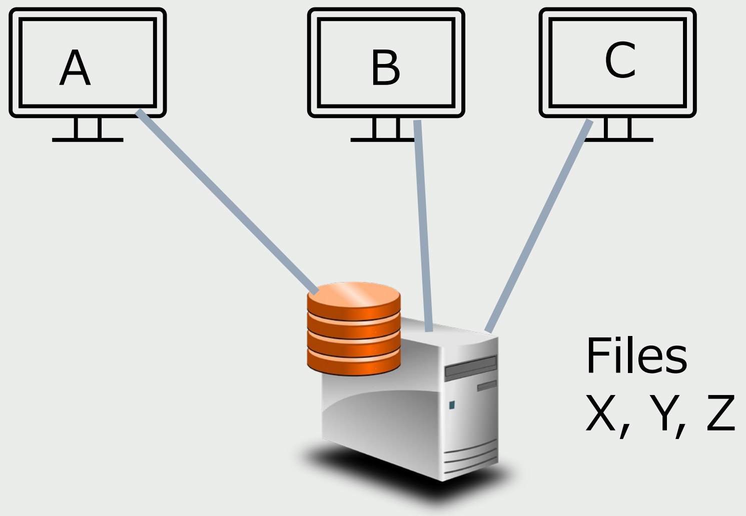
The clients occasionally wish to modify those files, so they can send messages like “read X” or “write Y” to change those files.

As the server receives these requests, it sends as response message back to the client, indicating

that the change is complete.

The client waits for the response to come back before attempting another request.

Example



The network imposes some minimum latency (let’s say 1ms) on each message.

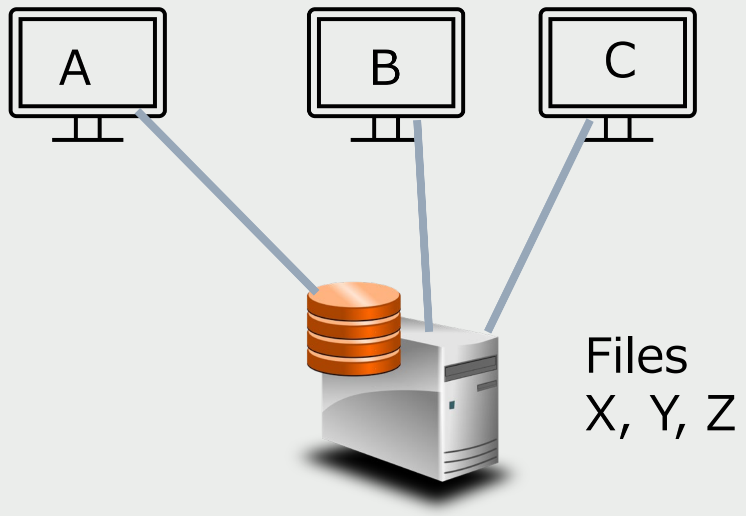
As a result, reading and writing a large amount of data from this file server is going to be very slow – much slower than accessing a local disk.

The network makes message delivery unreliable. A request/response could be delayed arbitrarily, or completely dropped.

If for example a network cable is out of action, there will be no communication along that link

If a client doesn’t receive a response to a message, then it has no choice but to wait and try the request again. It could wait a very long time!

Example

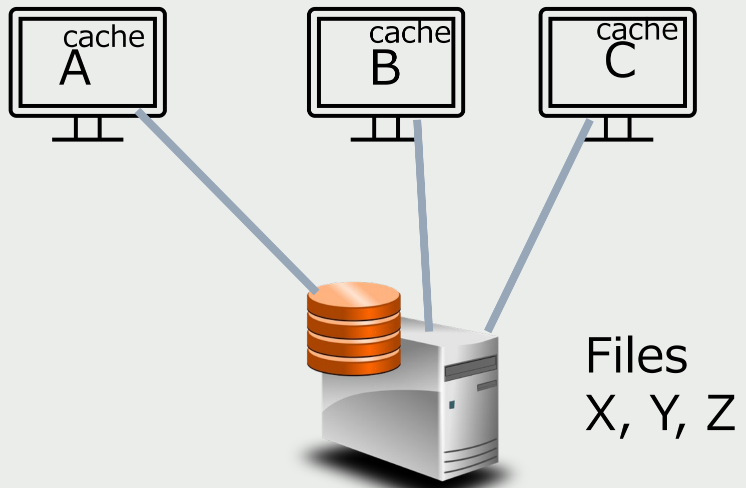


Consistency –every operation is applied in a known order, and all clients have an unfiltered view of the central server. (HIGH)

Availability – Every single read or write requires a network operation, making this system much slower than accessing a local file system. (LOW)

Partitionability –If a single client is partitioned from the file server, it cannot perform any operations. However, all other connected clients are able to continue. (MEDIUM)

Modified Example



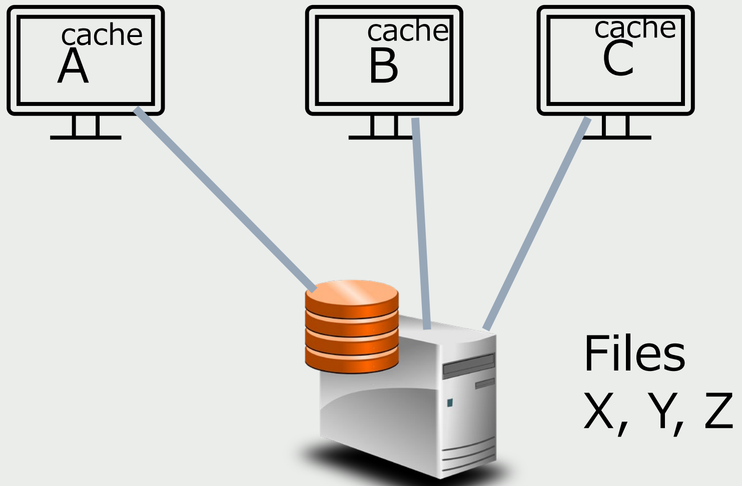
Add a cache to each of the clients (finite capacity)

Simple logic for managing these caches locally at each client:

Read – When a client attempts to read a file, it first looks in its cache to see if that file's data is already present. If it finds it, the read is satisfied from that data. If not, the client issues a read request to the server, waits for the response, and replaces the least recently used (LRU) item in the cache.

Write – When a client attempts to write a file, it first issues a write command to the server, and waits for a response. If an older version of the file exists in the client's cache, it is updated to the new value. If not, the client replaces the LRU item in the cache with the newly written value.

Modified Example

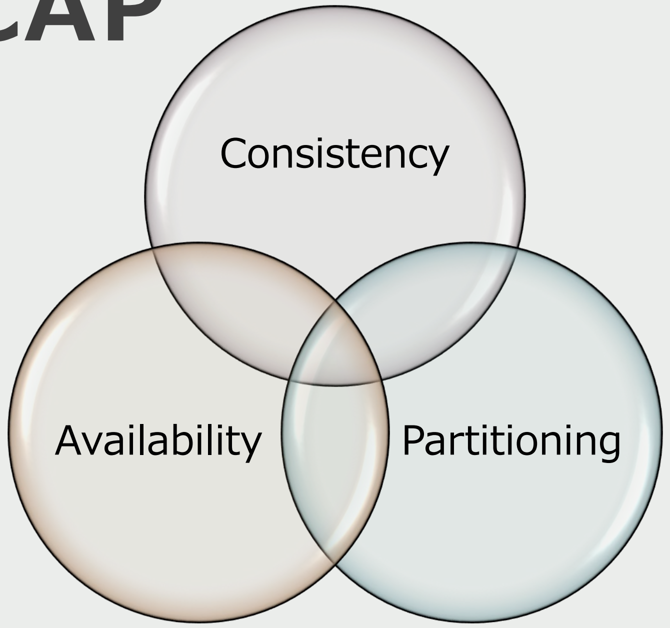


Consistency – (Write Through Cache) is less consistent than the direct access example because a client may fail to see writes made by other clients, when a value is available within its own cache. (MEDIUM)

Availability –will see much better read performance than the direct access example, because reads can be satisfied directly from cache without consulting the central server. However, writes are no faster because they always result in a network operation. (MEDIUM)

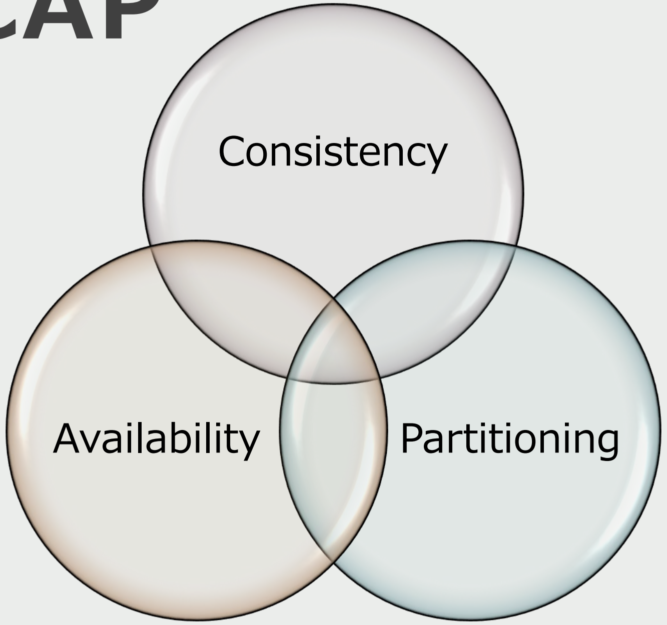
Partitionability – A client might be able to continue operating even when the network is down, if it is only performing reads on cached data. However, any write operation must block until the network results.

CAP



Dealing with the P is central to distributed computing: when we cannot communicate, should we optimistically try to make progress, or pessimistically wait, in order to achieve consistency? Different applications will require different solutions.

CAP



There are a variety of consistency models that can be implemented by adjusting just how and when caches are updated, and whether clients can continue to operate during a partition.

Strong Consistency: Once an update is complete, all clients will see that new value.

Causal Consistency: If process A tells B that it has updated X, then B will see the latest value of X.

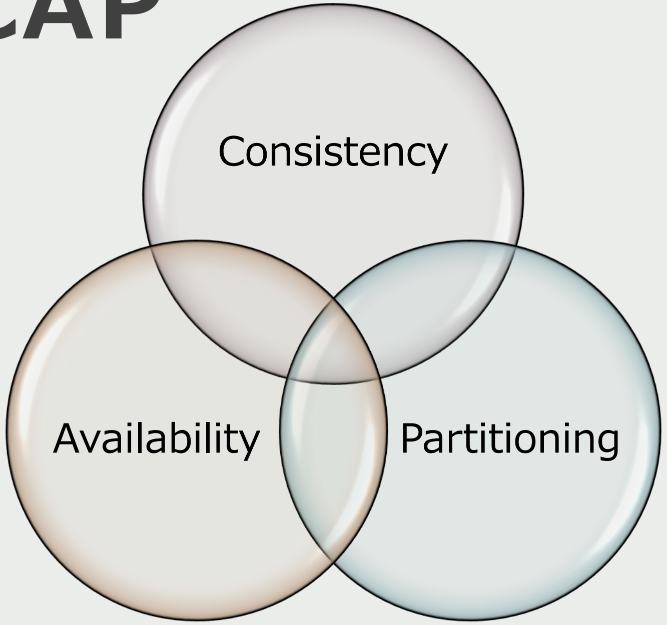
Read-Your-Writes: If process A updates X, then A will never see an older value of X.

Monotonic Reads: If process A reads a value from X, then it will never read back an older value.

Monotonic Writes: All writes by process A are applied in the order they are given.

Eventual Consistency: All updates will become visible to everyone, if you wait long enough.

CAP

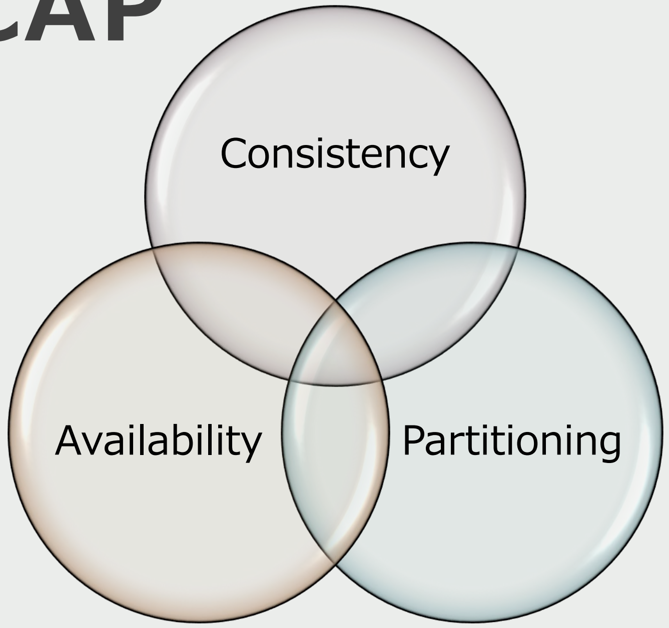


Availability is probably the easiest to work out.

For a given system, one could measure every attempt to read or write a value, and then compute a statistic like the mean, median, or 99th percentile of latency for various operations.

A system that provides a lower mean is providing “more” availability.

CAP

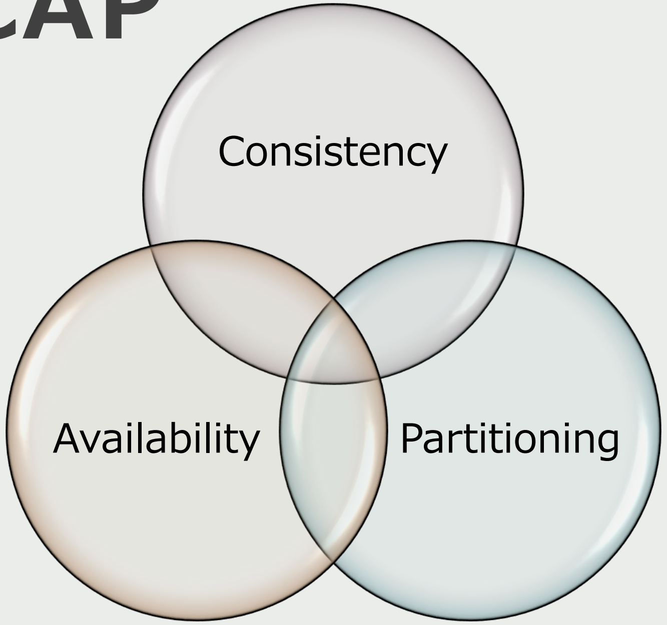


Replication

Provides insurance against storage failures, but also provide a high degree of availability for commonly used data.

How does it impact consistency?

CAP



A variety of consistency models exist:

Strong Consistency: Once an update is complete, all clients will see that new value.

Causal Consistency: If process A tells B that it has updated X, then B will see the latest value of X.

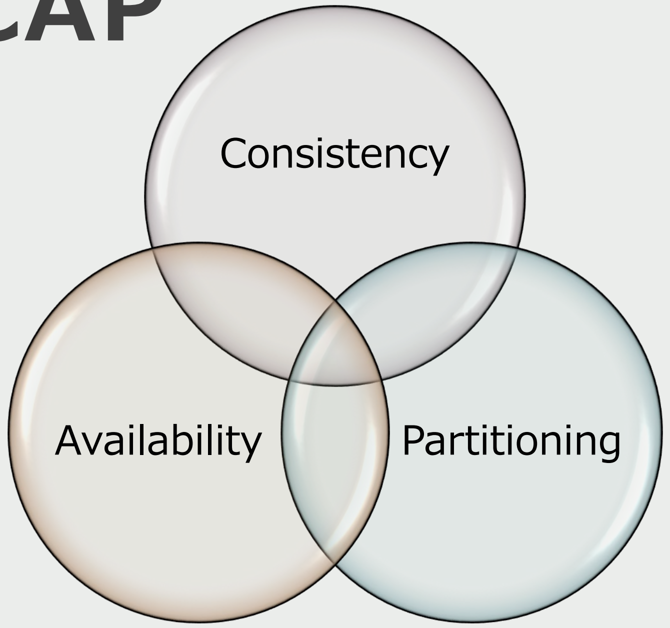
Read-Your-Writes: If process A updates X, then A will never see an older value of X.

Monotonic Reads: If process A reads a value from X, then it will never read back an older value.

Monotonic Writes: All writes by process A are applied in the order they are given.

Eventual Consistency: All updates will become visible to everyone, if you wait long enough.

CAP



Eventual Consistency: All updates will become visible to everyone, if you wait long enough.

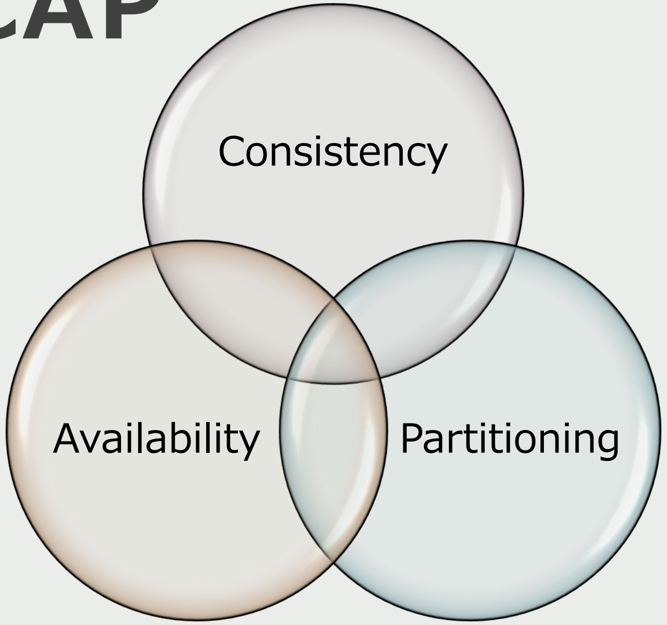
Weakest kind.

No guarantees on order of writes applied to different replicas

● No guarantees on what intermediate states a reader may observe.

● Just guarantees that “eventually” replicas will converge.

CAP



Causal Consistency: If process A tells B that it has updated X, then B will see the latest value of X

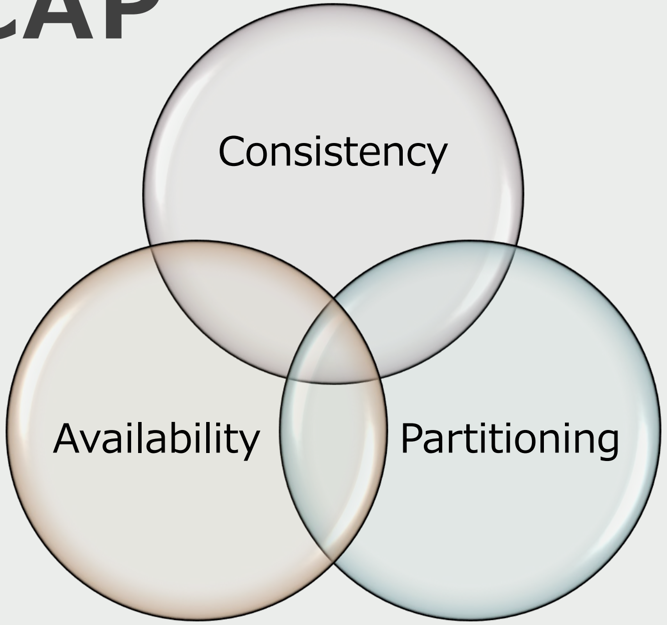
Operations to the datastore applied in causal order.

e.g. A comments on B’s post and then Bob replies to A’s comment. On all replicas, A’s comment written before B’s comment. This also means that it should be impossible to read B’s comment before reading A’s.

No guarantees for concurrent writes.

e.g. if two people comment on two unrelated posts then it is okay to apply the two writes in different orders on different replicas

CAP



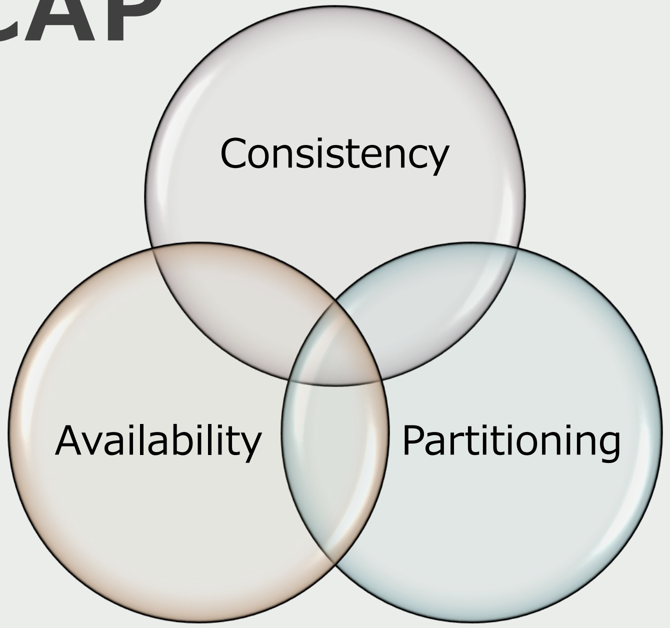
According to Vogel:

Suppose you have a system with N replicated storage units.

To update an item, a client must write W of the replicas upfront. (The remainder will get updated eventually in the background.)

To read an item, a client must read R of the replicas, in order to decide whether the most recent value has been read. (If the values differ, assume you can tell which one is the newest.)

CAP



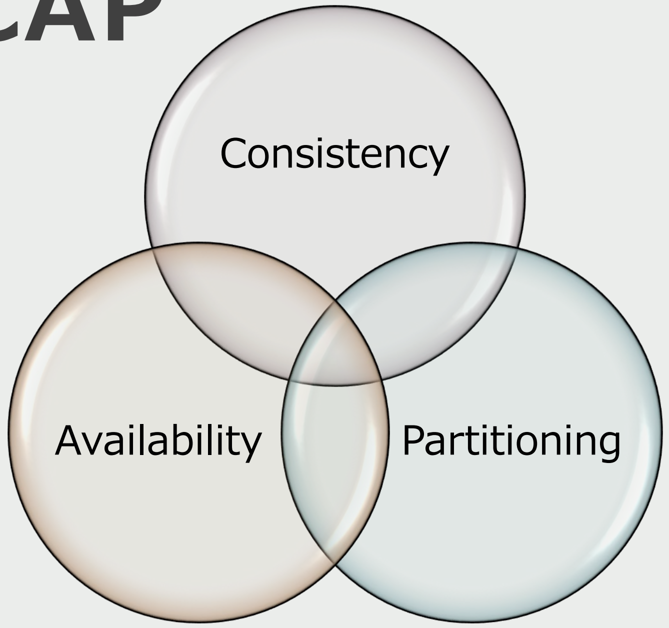
Suppose we have two replicas which are network partitioned (unable to communicate due to a network failure)

If we allow writes on either replica we will end up with inconsistent data

If we force the replicas to wait to synchronize we lose availability

If we never have a network failure then we lose partition tolerance

CAP

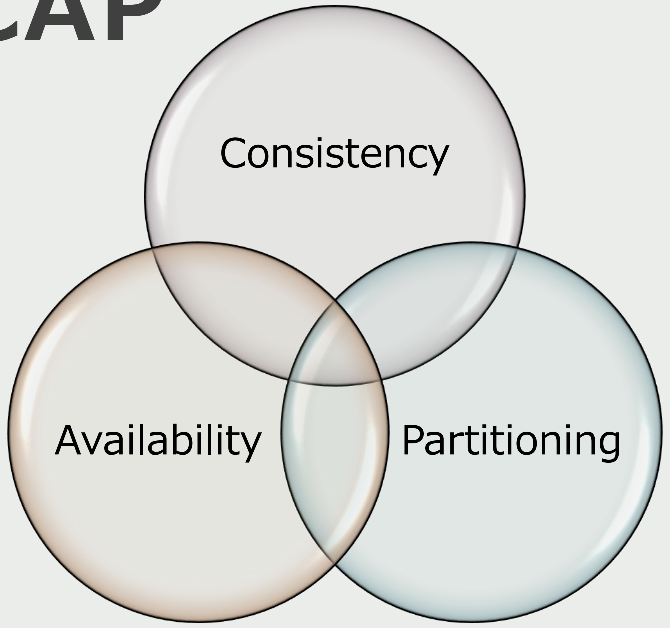


N=2, W=2, R=1 is a strongly consistent system: a writer must update both replicas, and a reader can read either one of them.

N=2, W=1, R=2 is also a strongly consistent system: a writer can update either replica, and a reader must read both to obtain the latest.

N=2, W=1, R=1 is an eventually consistent system: the writer can update either replica, and the reader can read either replica, so you may not see consistent results.

CAP

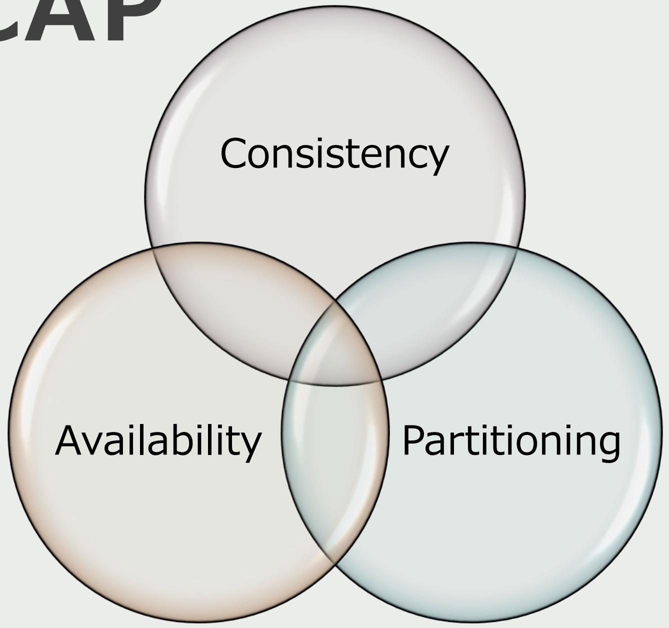


$(W + R) > N$ is strongly consistent

$(W+R) <= N$ is weakly consistent.

$W < (N+1)/2$ means write conflicts can occur

CAP



Modern architectures:

General belief is that for wide-area systems you can’t forfeit P or partitions.

For wide-area systems you can’t choose to not have partitions.

Therefore you have to balance consistency and availability.

This is the case for NoSQL Distributed Data.

ACID V BASE

ACID – highly consistent system

BASE – highly available system

ACID V BASE

ACID good fit for businesses which deal with online transaction processing (e.g., finance institutions) or online analytical processing (e.g., data warehousing).

ACID V BASE

BASE more flexible and fluid. Organisations where growth, expansion, change in data structures is likely should favour BASE.

SQL V NoSQL

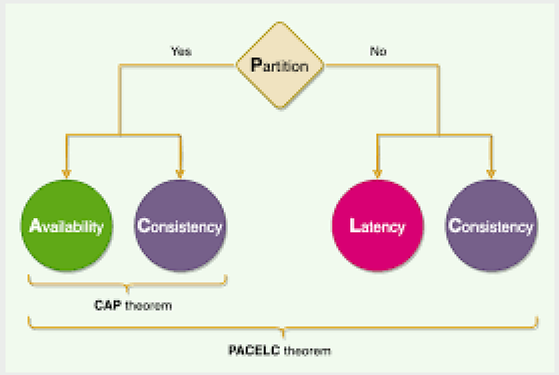
SQL primarily vertically scalable – NoSQL horizontally scalable

SQL table based, schema required – NoSQL column, row, key-value based, flexible schema

SQL better fit for heavy transaction loads and complex queries

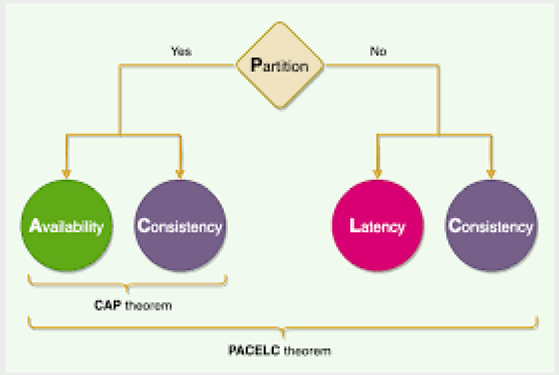
NoSQL preferred for unstructured data where growth and change are envisioned and availability is preferred over consistency

PACELC



What happens when there is no partition?

PACELC



In addition to Consistency, Availability, and Partition Tolerance it also includes Latency as one of the desired properties of a Distributed System.

The acronym PACELC stands for:

Partitioned, Availability, Consistency

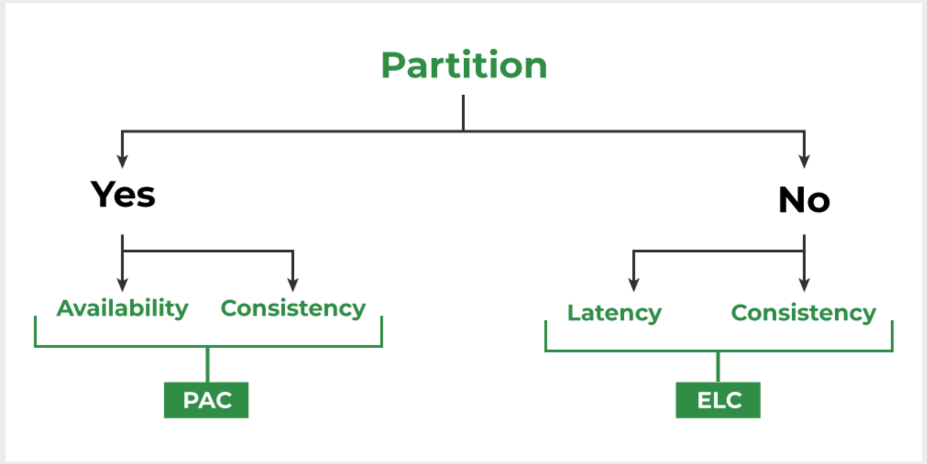
Else Latency, Consistency

PACELC

In the case of Network Partition ‘P’ a distributed system can have tradeoffs between Availability ‘A’ and Consistency ‘C’

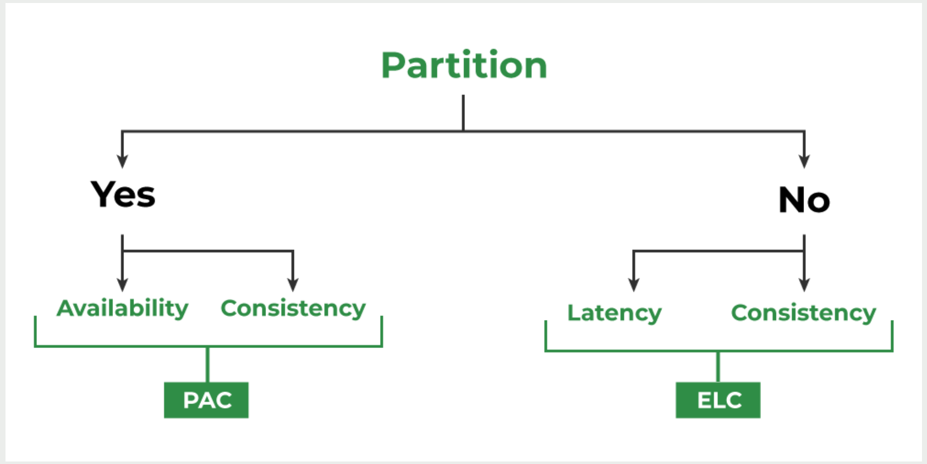
Else ‘E’ if there is no Network Partition then a distributed system can have tradeoffs between Latency ‘L’ and Consistency ‘C’.

Latency - performance



PACELC

Partition: two 2 nodes are not able to communicate with each other.


Partition
Yes
No
Availability
Consistency
Latency
Consistency
PAC
ELC