Introduction to NoSQL Databases

Chapter 13

Michael Tsiang

Stats 167: Introduction to Databases

UCLA

Introductory Videos

Advantages and Limitations of Relational Databases

Advantages and Limitations of NoSQL Databases

Introductory Videos

Prerequisite: What is a NoSQL Database?

First watch the first ten minutes (up to about 9:44) of the video "What is a NoSQL Database? How is Cloud Firestore structured? | Get to know Cloud Firestore #1" by Firebase:

https://youtu.be/v_hR4K4auoQ

For those who need transcripts: https://bit.ly/3Hkb9VI

This video covers a well-motivated introduction to NoSQL databases (specifically document databases), including the limitations of relational database design, some important considerations of NoSQL databases, some tradeoffs of schema-less databases, and NoSQL database scalability.

Prerequisite: 7 Database Paradigms

In the first video, the NoSQL database discussed was Cloud Firestore, which is an example of a document database. But document databases are just one of many types of NoSQL databases.

Watch the video "7 Database Paradigms" by Fireship:

https://youtu.be/W2Z7fbCLSTw

For those who need transcripts: https://bit.ly/3gaD7aC

This video not only introduces each type of NoSQL (and SQL) database, it builds up the different types of databases based on their complexity, which shows their relationships and use cases.

After watching both videos, we want to summarize the main takeaways and expand on some of the concepts mentioned.

Advantages and Limitations of Relational Databases

Advantages of Relational Databases

Even with the advent and popularization of non-relational databases in the last 20-25 years, it is important to note that there are still many advantages and reasons to using relational databases.

An overarching property of relational databases is *structure*. Through enforcing the relational database design (e.g., multiple tables of rows and columns, enforced datatypes within columns, primary/foreign keys), relational databases have several desirable qualities:

- Normalization
- Data Integrity
- ACID Compliance
- OLTP
- SQL

Normalization

In database design, **normalization** is the process of structuring a relational database according to a set of rules intended to reduce data redundancy (repetition) and reduce issues with mistakes (or anomalies) in insertions, updates, or deletions.

By having strict rules on what data can be stored and in what organizational structure, normalization helps to ensure **data integrity**, which is the completeness, accuracy, and consistency of the data.

Note: Referential integrity is a part of data integrity.

Normal Forms: 1NF and 2NF

Normalization follows a series of **normal forms**, adding stricter rules with each form.

A table (or **relation**) is in the **first normal form** (or **1NF**) if:

- Each column contains **atomic** (single or indivisible) values (rather than a set of values or a nested table).
- Each record is unique and identified by a primary key (i.e., no duplicated records).

A table is in the **second normal form** (or **2NF**) if:

- It satisfies the rules from 1NF.
- There are no **partial dependecies**: All non-key attributes (or columns) must depend on the whole primary key (not just part of it, if the key is based on multiple columns).

Normal Forms: 3NF

A table is in the **third normal form** (or **3NF**) if:

- It satisfies the rules from 2NF.
- There are no **transitive dependencies**: Non-key attributes should not depend on other non-key attributes.

If a table is in 3NF, it is often called **normalized**.

There are at least six normal forms and additional variants, but 3NF is the most common, as it balances between preserving data integrity without introducing too much complexity from additional tables.

<Examples of 1NF, 2NF, and 3NF>

A helpful DataCamp tutorial on normalization:
https://www.datacamp.com/tutorial/normalization-in-sql

ACID Compliance

A database is said to be **ACID compliant** if database transactions satisfy the following properties:

- **Atomicity**: A transaction is viewed as a single operation, so either all parts of a transaction are completed or none are.
- **Consistency**: Data must be in a valid (consistent) state before and after a transaction. If a transaction will take the data to an invalid state, it will not be completed.
- **Isolation**: Each transaction occurs independently of other transactions occurring at the same time.
- **Durability**: Changes to the data after a successful transaction are saved immediately and cannot be undone, even in the event of a system failure.

ACID compliance essentially guarantees that a query to the database should return the most up-to-date data. Most relational DBMSs are ACID compliant.

OLTP

Recall from Chapter 2 (Data Engineering): A standard data processing system for many scenarios is the **OLTP (online transaction processing)** system, which uses a relational database to process a large number of simple database transactions (usually online).

The relational database design is optimized for storing information in tables to be efficient for processing simple read and writes to the database.

SQL

Because relational databases have been used for over 50 years, SQL is a standard language that is largely shared by most relational databases.

SQL is a powerful language that leverages the relations in the database, so RDBMSs are optimized to perform joins and other complex queries efficiently.

Limitations of Relational Databases

While the structure of relational databases has many advantages, the strictness of that same structure is not ideal for more recent applications involving Big Data.

Some limitations of relational databases:

- Inflexible to changes in the schema
- Not suitable for semi-structured and unstructured data
- Limited scalability (can scale up/vertically but not scale out/horizontally)

Advantages and Limitations of NoSQL Databases

Non-Relational Databases

Non-relational databases, also called **NoSQL** databases, are databases that do not store data in relational tables.

Note: NoSQL stands for "not only SQL," not "no SQL."

NoSQL databases cover a wide range of database types. The four most popular types are:

- Key-value databases (also called key-value stores)
- Wide-column (or tabular) databases
- Document databases
- Graph databases

Advantages of NoSQL Databases

The main issues that NoSQL databases try to improve on over traditional relational databases are flexibility, availability, efficiency, and scalability.

- Flexibility

  The design of a non-relational database does not have to follow a strict schema, so there is more flexibility in how data is stored/organized. NoSQL databases can store both semi-structured and unstructured data.

- Availability

  Most NoSQL databases are **distributed**, so data is spread across several computers/servers. Components in a distributed system are replicated (i.e., there is **data redundancy**) so that the whole system will still be available and reliable even if one (or more) nodes is unavailable.

Advantages of NoSQL Databases

- Efficiency

  Relational databases are optimized for an efficient balance between reads and writes to the database, but they can be inefficient if there are many reads to the database at once. NoSQL databases are typically optimized to manage huge volumes of reads simultaneously.

- Scalability

  NoSQL databases scale out/horizontally well. Increasing storage can be done by adding computers or servers, and the distributed system adapts to the added components (usually seamlessly).

ACID Versus BASE

While ACID compliance is guaranteed in most relational databases, requiring every query to always reflect the most recent updates is often impractical in distributed systems that store data across multiple nodes and data centers. Achieving full ACID compliance in such environments can be expensive and difficult to scale.

Instead of strict ACID guarantees, many NoSQL databases follow a more flexible approach known as **BASE**, which emphasizes:

- **Basic Availability**: Data is available most of the time, even during a partial system failure.
- **Soft State**: Data values may change over time, even without new updates (due to eventual convergence).
- **Eventual Consistency**: Updates will propagate to all nodes, but not necessarily immediately.

Essentially, ACID prioritizes data reliability and consistency while BASE prioritizes availability and scalability.

References on ACID Versus BASE

For more information on differences between ACID and BASE:

- https://phoenixnap.com/kb/acid-vs-base
- https://www.scylladb.com/glossary/database-consistency/
- https://www.datacamp.com/blog/acid-transactions

NoSQL Database Design Concepts

To better understand how (distributed) NoSQL databases support BASE principles, we will discuss three fundamental design concepts:

- Replication
- Consistency
- Sharding

Replication

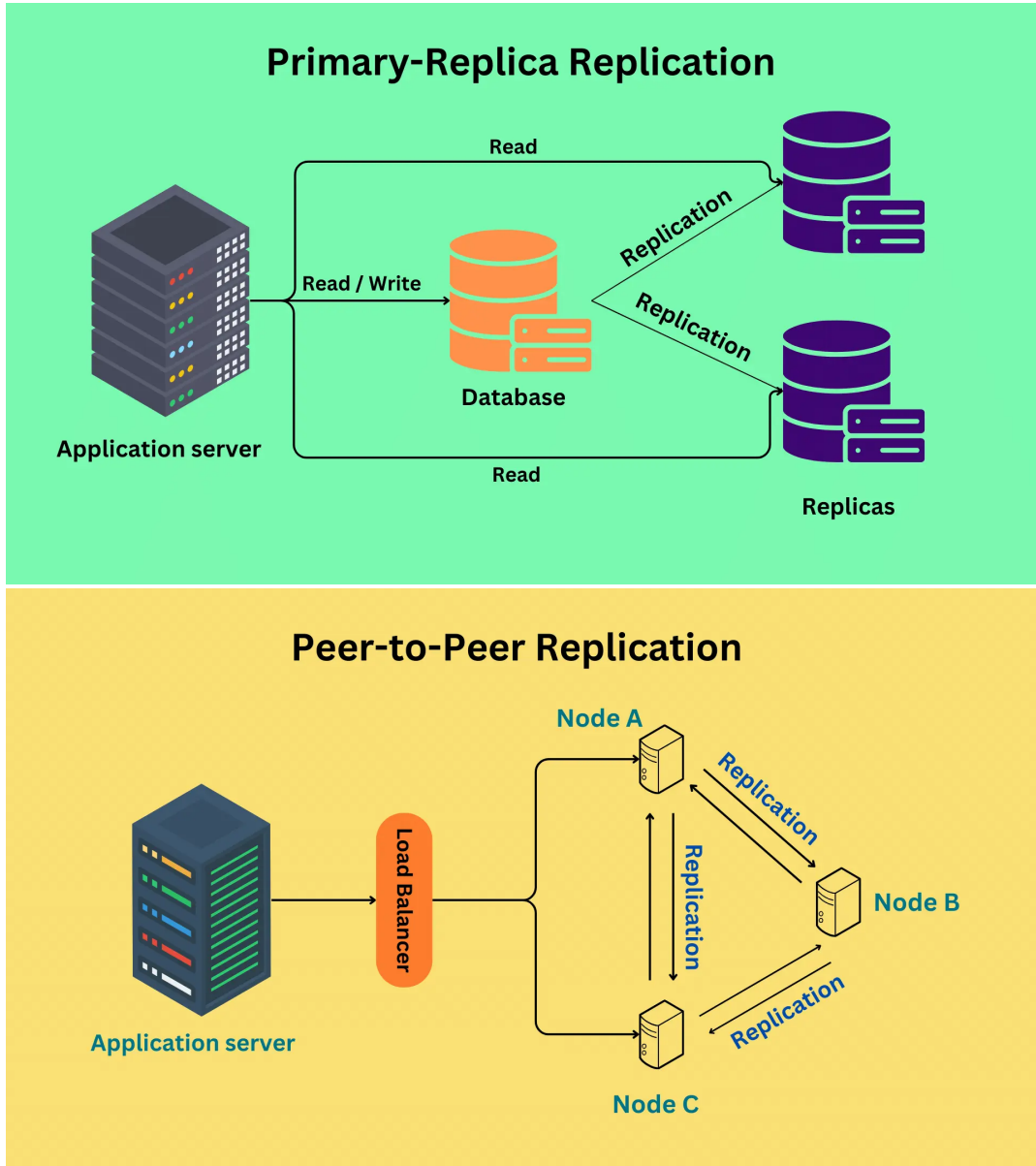
In a distributed system, **replication** is the process of copying and maintaining data in multiple nodes. Replication is an important component in ensuring and improving availability, efficiency, and scalability.

There are two main types of replication:

- **Primary-replica** (or **primary-secondary**) replication (e.g., MongoDB)
  - One node (the primary node) handles all writes.
  - Replica (or secondary) nodes replicate from the primary.
  - If the primary fails, a new primary is elected from replicas.
- **Peer-to-peer** (or **leaderless**) replication (e.g., Cassandra)
  - All nodes are equal (no primary).
  - Any node can handle read/write requests (depending on configuration).

Both types of replication can be done synchronously or asynchronously.

Primary-Replica vs Peer-to-Peer Replication



Source: Design Gurus, "What is Primary-Replica vs Peer-to-Peer Replication?

References on Replication

For more information on replication:

- https://www.ibm.com/think/topics/data-replication
- https://www.qlik.com/us/data-replication/database-replication
- https://www.geeksforgeeks.org/database-replication-and-their-types-in-system-design/

Consistency

Consistency in a distributed system refers to the rules about how and when **writes** (or updates) to one replica become visible to **reads** (or queries) from other replicas.

There are several **consistency models**, the two most common being:

- **Strong consistency**
  - Every read returns the most recent version of the data, no matter which node you read from.
  - When a write occurs on one node, all other nodes in the cluster reflect the latest changes immediately.
- **Eventual consistency**
  - When a write is made, the update will *eventually* be reflected in all nodes that store the data.
  - Early results of eventual consistency data reads may not have the most recent updates because it takes time for updates to reach replicas across a database cluster.

Quorum Consistency

As an example, Cassandra uses a method to achieve a form of strong consistency called **quorum consistency**.

The **replication factor** (**RF**) is the number of copies of the data that is stored in the cluster. For example, if RF $= 3$, then each piece of data is replicated on 3 different nodes.

A **write quorum** (**W**) is the minimum number of replicas that must acknowledge a write before it is considered successful.

A **read quorum** (**R**) is the minimum number of replicas that must respond to a read request.

- If R $+$ W $>$ RF, then a read and write quorum will overlap, ensuring that reads will see the latest successful write. This achieves strong consistency (in the quorum sense).
- If R $+$ W $<$ RF, there may be no overlap, meaning a read could return stale data, resulting in weaker consistency.

Tunable Consistency

Cassandra allows for **tunable consistency**, meaning the user can adjust the values of R and W per operation depending on specific needs (e.g., prioritizing availability and speed over consistency for certain scenarios).

Cassandra's tunable consistency is implemented through **consistency levels** (**CL**), which control how many replicas must respond for an operation to succeed.

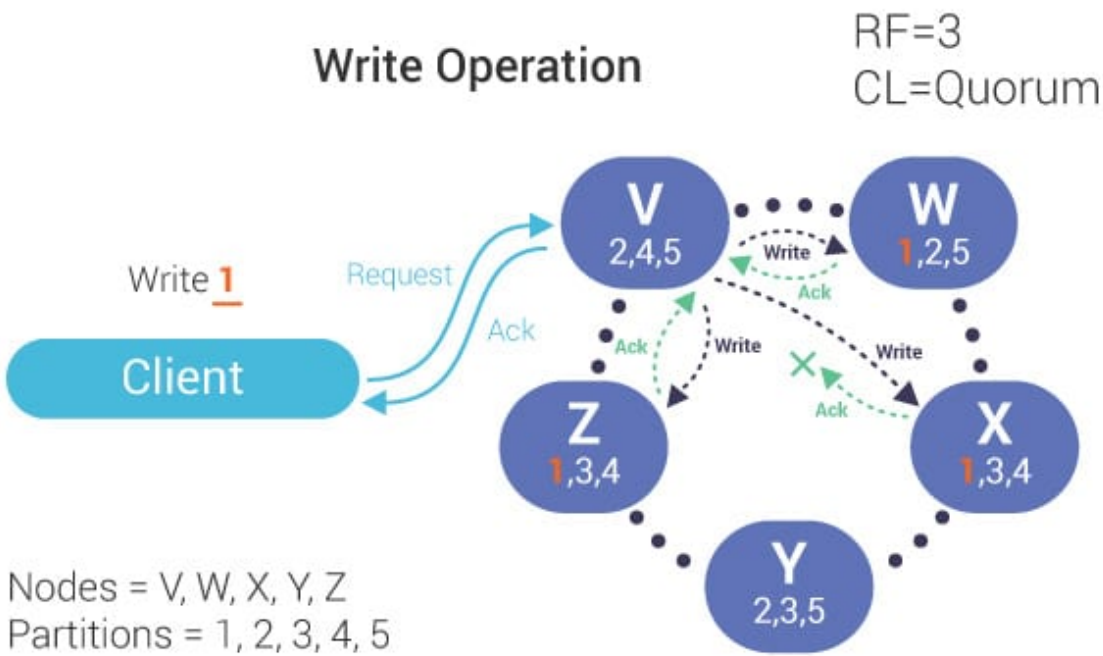
Consistency Levels

Common (tunable) consistency levels:

Consistency Level	Description
`ONE`	Wait for 1 replica to respond (fast, but least consistent)
`QUORUM`	Wait for a majority of replicas (e.g., 2 out of 3 if RF $= 3$)
`ALL`	Wait for all replicas (strongest, slowest, least available)
`TWO`, `THREE`	Wait for 2 or 3 replicas (used when RF ≥ 2 or 3)
`LOCAL_QUORUM`	Like `QUORUM`, but limited to the local data center (for systems with multiple data centers)
`EACH_QUORUM`	Requires a quorum in each data center
`ANY`	A write can be acknowledged by any node (no guarantees of durability)

Example: RF = 3, CL = Quorum

An illustration of a write operation with RF = 3 and CL = Quorum on a 5 node system:



Source: ScyllaDB University article “Consistency Level”

References on Consistency

For more information on consistency and consistency levels:

- https://wiki.databurst.tech/docs/roadmap/distributed-systems-concepts/consistency-vs-eventual-consistency/
- https://www.scylladb.com/glossary/consistency-models/
- https://www.pythian.com/blog/technical-track/cassandra-consistency-level-guide

Sharding

Sharding is a technique used in distributed systems to divide a large dataset into smaller and more manageable pieces called **shards**, which are stored across multiple machines.

Instead of keeping all the data on a single server, sharding distributes it across multiple nodes (or servers), with each node being responsible for a specific subset of the data.

For applications with massive data volumes or high read/write requests, a single machine often cannot scale its storage, memory, or processing power to keep up with demands. Sharding enables **horizontal scaling** (and thus theoretically limitless scalability): The system can grow as needed by adding more nodes and redistributing shards.

Applications of Sharding

Because data storage and access are distributed across shards, sharding enables databases to handle huge datasets with high availability and performance.

Some application areas (from Hazelcast):

- Social media platforms (e.g., posts, photos, videos)
- E-commerce (high website traffic, orders, inventory)
- Gaming (low latency in global massively multiplayer online games)
- Financial services (transaction data, customer records, financial histories)

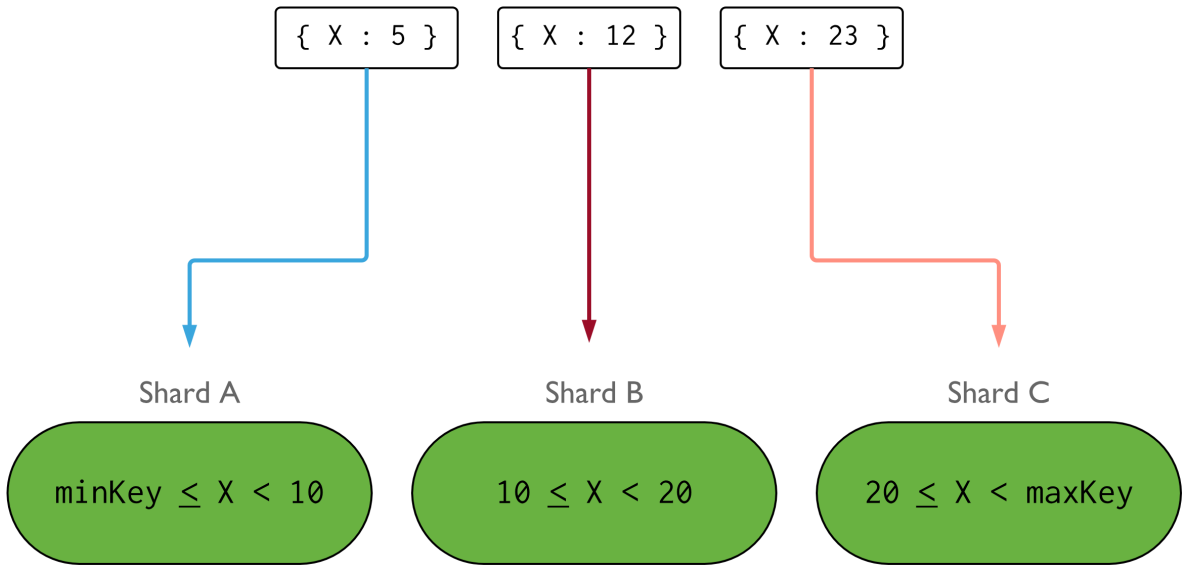
Types of Sharding

There are a two common criteria used to separate data into shards:

- **Range-based** (or **dynamic**) sharding
  - Splits data based on a range of values (e.g., dates, numeric values, alphanumeric identifiers)
  - Efficient for range queries
  - Imbalanced shard sizes if the data distribution is uneven
- **Hash-based** sharding
  - A hash function is applied to a key (e.g., user ID), and the result determines the shard.
  - Evenly distributes data, which can reduce imbalanced workloads across nodes.

Range-Based Sharding

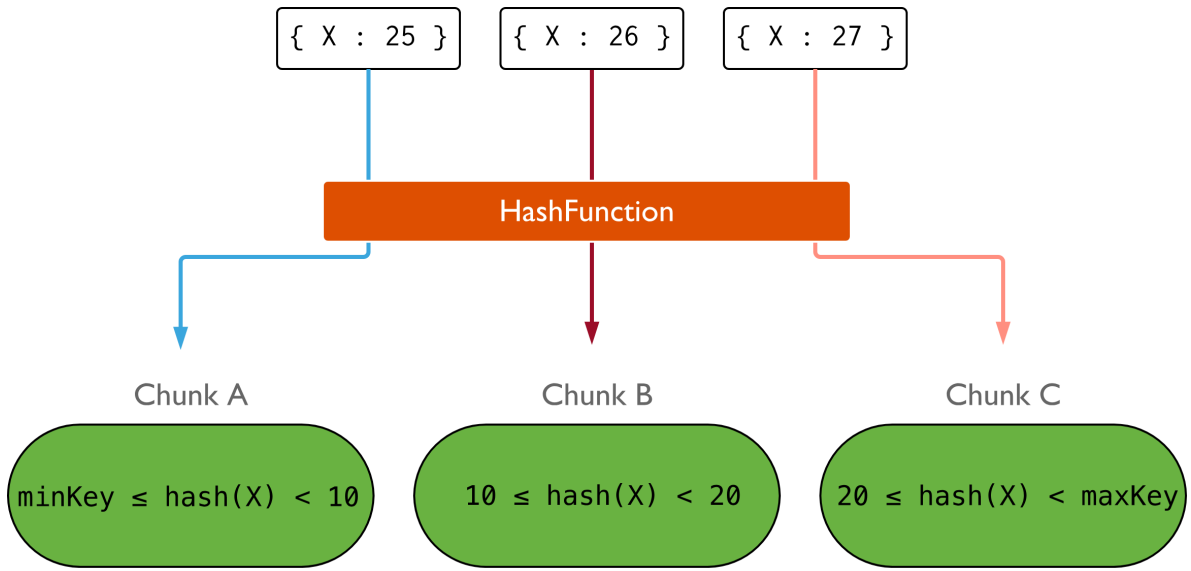
An illustration of range-based sharding:



Source: MongoDB Database Manual article “Ranged Sharding”

Hash-Based Sharding

An illustration of hash-based sharding:



Source: MongoDB Database Manual article "Hashed Sharding"

Choosing the Shard Key

The **shard key** is the field (or combination of fields) that the system uses to determine how to split and distribute data across shards.

Choosing an appropriate shard key is important for designing a balanced and efficient sharded database.

Some key characteristics of a good shard key:

- **High cardinality**

  The shard key should have a large number of unique values.

- **Low frequency**

  Shard key values should not be repeated too often, so data is spread out relatively evenly.

- **Non-monotonically changing**

  Shard key values should not be changing in a predictable or sequential way.

References on Sharding

For more information on sharding:

- https://hazelcast.com/glossary/sharding/
- https://medium.com/@jeeyoungk/how-sharding-works-b4dec46b3f6
- https://docs.mongodb.com/manual/sharding/
- https://searchoracle.techtarget.com/definition/sharding

For more on shard keys:

- https://hypermode.com/blog/shard-key
- https://www.mongodb.com/docs/manual/core/sharding-choose-a-shard-key/

Limitations of NoSQL Databases

The flexibility and lack of structure in the design of NoSQL databases can also be a hindrance for certain scenarios.

Some limitations of NoSQL databases:

- There is no standard query language across different NoSQL databases.

  For example, Cassandra uses CQL, MongoDB uses MongoDB query language, Neo4j uses Cypher

- While simple reads from NoSQL databases are usually very fast, complex queries and analytics may be slower.

- Distributed systems are subject to the restrictions of the CAP Theorem.

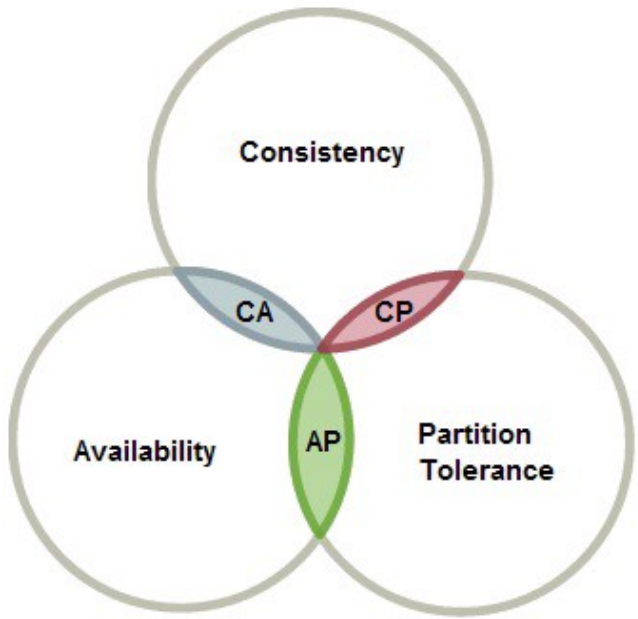
What is CAP?

The "CAP" acronym in the CAP Theorem stand for three desirable properties of distributed systems:

- **Consistency**: All users should see the same (up-to-date) data, regardless of which node they connect to.
- **Availability**: All user requests should get a response from the system.
- **Partition Tolerance**: A **partition** is a communication break between nodes within a distributed system. Partition tolerance means the system will still work even if there is a partition between two nodes.

The CAP Theorem

The **CAP Theorem** states that a distributed database system can only provide two of the three properties: Consistency, Availability, and Partition Tolerance.



Source: https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e

CAP Theorem Restated

In a distributed system, partitions cannot be avoided, so partition tolerance must be guaranteed (this is why sharding and data redundancy are so important).

Thus, the CAP Theorem can be restated to mean that, when a partition occurs, a distributed database system has to make a trade-off between consistency and availability. It cannot guarantee both consistency and availability when a partition occurs.

Capping It Off

Because of the CAP Theorem, NoSQL databases are classified as either CP or AP, depending on which properties they can guarantee.

For example:

- MongoDB is a CP database
- Cassandra is an AP database

For more information about the CAP Theorem:

- https://www.ibm.com/think/topics/cap-theorem
- https://www.analyticsvidhya.com/blog/2020/08/a-beginners-guide-to-cap-theorem-for-data-engineering/
- https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/