CAP Theorem 
C: Consistency 
O Value 
Sacrifice 
Use Cases 
Real-world examples 
A: 
Availability 
t 
P: 
blog.bytebytego.com 
Partition tolerance 
o 
O 
o 
O 
o 
o 
O Value @t O Value 1 0 Value @t - 2 
CA systems 
No sacrifice 
Single-node only 
Single node RBMS 
(MySQL, Oracle) 
CP systems 
Availability 
Strong consitency. 
Banks, financial systems 
Zookeeper, 
BigTable 
AP systems 
Consistency 
Low latency. 
Consistency requirement 
is not high 
Cassandra, 
CouchDB 

CAP theorem states that a distributed system can't provide more than two of these three guarantees simultaneously.

**Consistency**: consistency means all clients see the same data at the same time no matter which node they connect to.

**Availability**: availability means any client that requests data gets a response even if some of the nodes are down.

**Partition Tolerance**: a partition indicates a communication break between two nodes. Partition tolerance means the system continues to operate despite network partitions.

The “2 of 3” formulation can be useful, **but this simplification could be misleading**.

1. Picking a database is not easy. Justifying our choice purely based on the CAP theorem is not enough. For example, companies don't choose Cassandra for chat applications simply because it is an AP system. There is a list of good characteristics that make Cassandra a desirable option for storing chat messages. We need to dig deeper.
2. “CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare”. Quoted from the paper: CAP Twelve Years Later: How the “Rules” Have Changed.
3. The theorem is about 100% availability and consistency. A more realistic discussion would be the trade-offs between latency and consistency when there is no network partition. See PACELC theorem for more details.

**Is the CAP theorem actually useful?**

I think it is still useful as it opens our minds to a set of tradeoff discussions, but it is only part of the story. We need to dig deeper when picking the right database.

[CAP Theorem](https://blog.algomaster.io/p/cap-theorem-explained)

1. Consistency 
Consistency ensures that every read receives the most recent Write or an error. This 
means that all working nodes in a distributed system will return the same data at any 
given 
Every read receives the most 
recent write or an error 
In a consistent distributed system, if you write data to node A, a read operation 
from node B Will immediately reflect the write operation on node A. 
Consistency is crucial for applications where having the most up-to-date data is 
critical, such as financial systems, where a balance inquiry must reflect the most up- 
to-date State Of an account. 
2. Availability 
Availability guarantees that every request (read or write) receives a response, without 
ensuring that it contains the most recent write. This means that the system remains 
operational and responsive, even if the response from some of the nodes don't reflect 
most up-to-date data. 
blog." I go master.io 
Val 2 x 
Every request receives a response 
Availability is important for applications that need to remain operational at all times, 
such as online retail Systems. 

3. Partition Tolerance 
Partition Tolerance means that the system continues to function despite network 
partitions where nodes cannot communicate with each other. 
System continues to works despite 
failure in communication 
A network partition occurs when a network failure causes a distributed system to 
split into two Or more groups Of nodes that cannot communicate with each other. 
When there is a network partition, the system must choose between Consistency and 
Availability. 
Partition Tolerance is essential for distributed systems because network failures can 
and do happen. A system that tolerates partitions can maintain operations across 
different network segments. 

**The CAP Trade-Off: Choosing 2 out of 3**

The CAP theorem asserts that in the presence of a network partition, a distributed system must choose between **Consistency** and **Availability**.

Let's explore these scenarios:

**CP (Consistency and Partition Tolerance):**

These systems prioritize consistency and can tolerate network partitions, but at the cost of availability. During a partition, the system may reject some requests to maintain consistency.

Traditional relational databases, such as MySQL and PostgreSQL, when configured for strong consistency, prioritize consistency over availability during network partitions.

**Banking systems** typically prioritize **consistency over availability** since data accuracy is more critical than availability during network issues.

Consider an ATM network for a bank. When you withdraw money, the system must ensure that your balance is updated accurately across all nodes (consistency) to prevent overdrafts or other errors.

**AP (Availability and Partition Tolerance):**

These systems ensure availability and can tolerate network partitions, but at the cost of consistency. During a partition, different nodes may return different values for the same data.

NoSQL databases like Cassandra and DynamoDB are designed to be highly available and partition-tolerant, potentially at the cost of strong consistency.

**Amazon's** shopping cart system is designed to always accept items, prioritizing availability.

When you add items to your Amazon cart, the action almost never fails, even during high traffic periods like Black Friday.

**CA (Consistency and Availability):**

In the absence of partitions, a system can be both consistent and available. However, network partitions are inevitable in distributed systems, making this combination impractical.

**Example Systems:** Single-node databases can provide both consistency and availability but aren't partition-tolerant. In a distributed setting, this combination is theoretically impossible.

**Practical Design Strategies**

Designing distributed systems requires carefully balancing these trade-offs based on application requirements.

Here are some practical strategies:

**1. Eventual Consistency**

For many systems, strict consistency isn't always necessary.

Eventual consistency can provide a good balance where updates are propagated to all nodes eventually, but not immediately.

**Example:** Systems where immediate consistency is not critical, such as DNS and content delivery networks (CDNs).

**2. Strong Consistency**

A model ensuring that once a write is confirmed, any subsequent reads will return that value.

**Example:** Systems requiring high data accuracy, like financial applications and inventory management.

**3. Tunable Consistency**

Tunable consistency allows systems to adjust their consistency levels based on specific needs, providing a balance between strong and eventual consistency.

Systems like Cassandra allow configuring the level of consistency on a per-query basis, providing flexibility.

**Example:** Applications needing different consistency levels for different operations, such as e-commerce platforms where order processing requires strong consistency but product recommendations can tolerate eventual consistency.

**4. Quorum-Based Approaches:**

Quorum-based approaches use voting among a group of nodes to ensure a certain level of consistency and fault tolerance.

**Example:** Systems needing a balance between consistency and availability, often used in consensus algorithms like Paxos and Raft.

**Beyond CAP: PACELC**

While CAP is foundational, it doesn't cover all scenarios.

Daniel Abadi proposed the [**PACELC**](https://en.wikipedia.org/wiki/PACELC_theorem) theorem as an extension by introducing **latency** and **consistency** as additional attributes of distributed systems.

* If there is a partition (P), the trade-off is between availability and consistency (A and C).
* Else (E), the trade-off is between **latency (L)** and **consistency (C)**.

This theorem acknowledges that even when the system is running normally, there's a tradeoff between latency and consistency.

In conclusion, the CAP Theorem is a powerful tool for understanding the inherent trade-offs in distributed system design. It's not about choosing the "best" property, but rather about making informed decisions based on the specific needs of your application.

By carefully evaluating the CAP trade-offs, you can architect robust and resilient systems that deliver the right balance of consistency, availability, and partition tolerance.

2. 
3. 
4. 
5. 
Basic and one of the most important concept in Distributed Databases. 
Useful to know this to design efficient distributed system for your given business logic. 
Let's first breakdown CAP 
2. 
3. 
Consistency: In a consistent system, all nodes see the same data simultaneously. If we perform a read operation on a consistent system, it 
should return the value of the most recent write operation. The read should cause all nodes to return the same data. All users see the same data 
at the same time, regardless of the node they connect to. When data is written to a single node, it is then replicated across the other nodes in 
the system. 
Availability When availability is present in a distributed system, it means that the system remains operational all of the time. Every request 
will get a response regardless of the individual state of the nodes. This means that the system will operate even if there are multiple nodes 
down. Unlike a consistent system, there's no guarantee that the response will be the most recent write operation. 
Partition Tolerance. When a distributed system encounters a partition, it means that there's a break in communication between nodes. If a 
system is partition-tolerant, the system does not fail, regardless of whether messages are dropped or delayed between nodes within the 
system. To have partition tolerance, the system must replicate records across combinations of nodes and networks. 
What does the CAP Theorem says, 
1. The CAP theorem states that a distributed system can only provide two of three properties simultaneously: consistency, availability, and 
partition tolerance. The theorem formalises the tradeoff between consistency and availability when there's a partition. 
CAP Theorem NoSQL Databases: NOSQL databases are great for distributed networks. They allow for horizontal scaling, and they can quickly scale 
across multiple nodes. When deciding which NoSQLdatabase to use, it important to keep the CAP theorem in mind. 
CA Databases: CA databases enable consistency and availability across all nodes. Unfortunately, CA databases can't deliver fault tolerance. In 
any distributed system, partitions are bound to happen, which means this type of database isn't a very practical choice. That being said, you still 
can find a CA database if you need one. Some relational databases, such as MySQL or PostgreSQL, allow for consistency and availability. You can 
deploy them to nodes using replication. 
2. 
3. 
CP Databases: CP databases enable consistency and partition tolerance, but not availability. When a partition occurs, the system has to turn 
off inconsistent nodes until the partition can be fixed. MongoDB is an example of a CP database. It's a NOSQL database management system 
(DBMS) that uses documents for data storage. It's considered schema•less, which means that it doesn't require a defined database schema. It's 
commonly used in big data and applications running in different locations. The CP system is structured so that there's only one primary 
node that receives all of the write requests in a given replica set. Secondary nodes replicate the data in the primary nodes, so if the 
primary node fails, a secondary node can stand-in. In banking system Availability is not as important as consistency, so we can opt it 
(MongoDB). 
AP Databases: AP databases enable availability and partition tolerance, but not consistency. In the event of a partition, all nodes are available, 
but they're not all updated. For example, if a user tries to access data from a bad node, they won't receive the most up-to-date version of the 
data. When the partition is eventually resolved, most AP databases will sync the nodes to ensure consistency across them. Apache Cassandra is 
an example of an AP database. It's a NOSQL database with no primary node, meaning that all of the nodes remain available. Cassandra allows 
for eventual consistency because users can re-sync their data right after a partition is resolved. For apps like Facebook, we value availability 
more than consistency, we'd opt for AP Databases like Cassandra or Amazon DynamoDB. 