Big Data and Cloud Computing

Dr. Walid Miloud Dahmane khemis miliana University SMI Department Email : w.miloud-dahmane@univ-dbkm.dz

1.0 March 2025

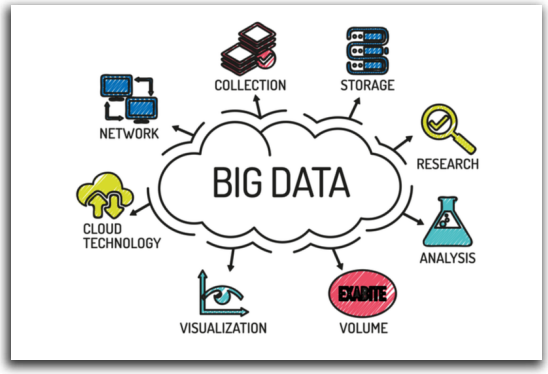
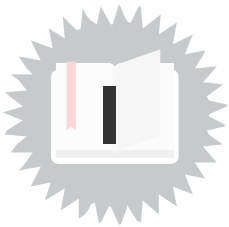


Table of contents

Chapter objectives



By the end of this chapter, the student will be able to:

- Understand the concepts of sharding and replication.
- Distinguish between file systems and distributed file systems.
- Understand fundamental concepts that enhance the robustness of big data.

Chapter 2: Big Data Storage Concepts

1. Introduction

Data from outside sources often comes in messy formats that can't be used right away. To fix this, data wrangling is used. This means cleaning, organizing, and preparing the data so it can be analyzed properly.

Here's how it works:

- First, the raw data is stored as it was received.
- Then, after cleaning and organizing, the prepared data is stored again for use.

Typically, storage is required whenever the following occurs:

- We get new data from outside or plan to use it in big data systems.
- The data is changed to make it easier to analyze.
- We process data through ETL (Extract, Transform, Load) or get results from an analysis.

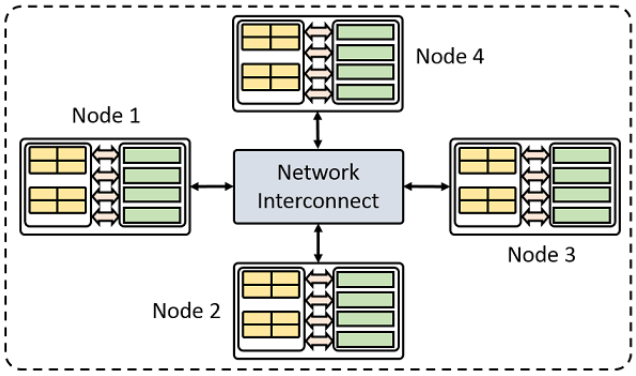
2. Clusters

A cluster is a tightly coupled collection of servers, or nodes.

They servers usually have the same hardware specifications and are connected together via a network to work as a single unit.

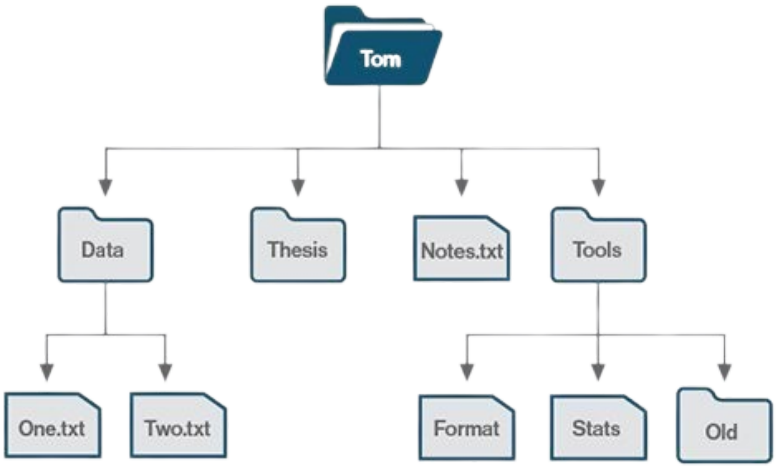
Each node in the cluster has its own dedicated resources, such as memory, a processor, and a hard drive.

A cluster can execute a task by splitting it into small pieces and distributing their execution onto different computers that belong to the cluster.

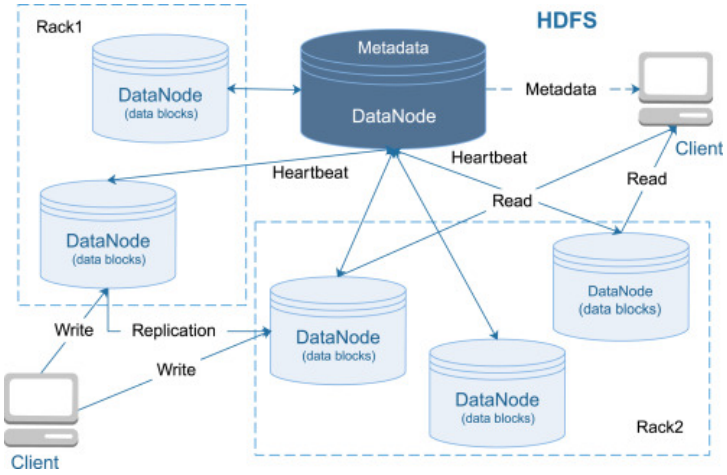


Cluster

3. File Systems and Distributed File Systems



File Systems (FS)



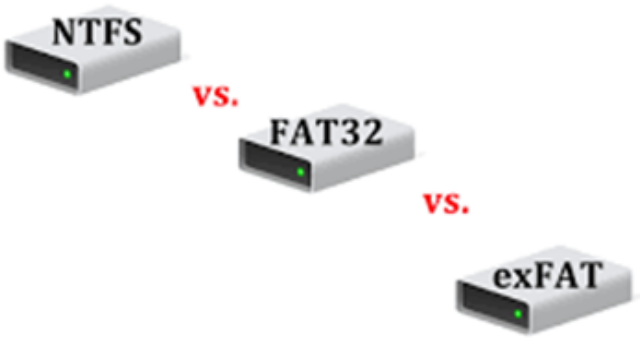
Distributed File Systems (DFS)

File Systems

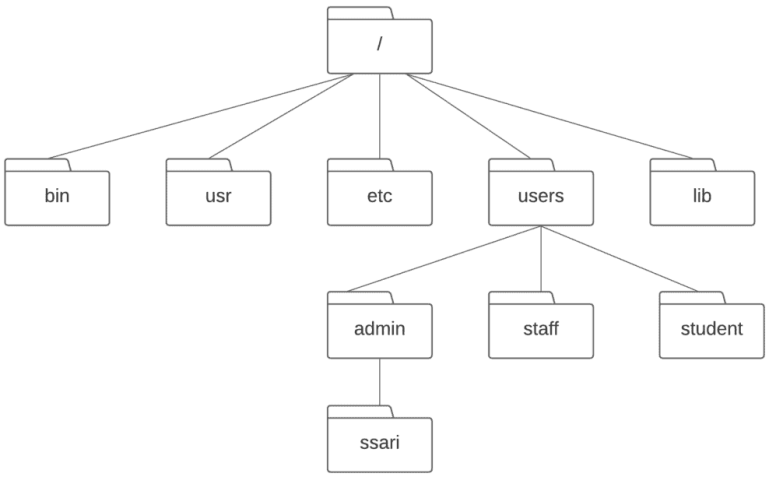
Definition

- A file system is the method of storing and organizing data on a storage device, such as flash drives, DVDs and hard drives.
- A file is an atomic unit of storage used by the file system to store data.

Example



Examples of File Systems

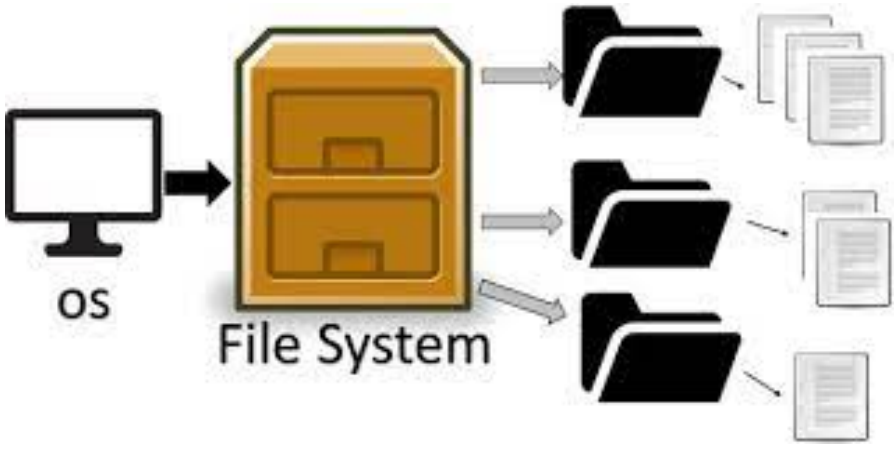


Linux File System Hierarchy Diagram

Extra

A file system provides a logical view of the data stored on the storage device and presents it as a tree structure of directories and files.

Operating systems employ file systems to store and retrieve data on behalf of applications.



OS and FS

Distributed File Systems *Definition*

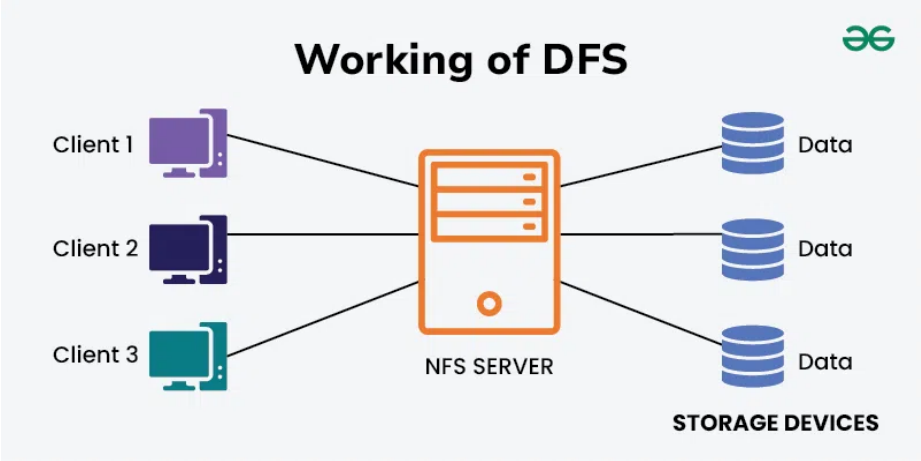
A distributed file system is a file system that can store large files spread across the nodes of a cluster.

To the client, files appear to be local; however, this is only a logical view.

Physically, the files are distributed throughout the cluster.

This local view is presented via the distributed file system and it enables the files to be accessed from multiple locations.

Examples include the Google File System (GFS) and Hadoop Distributed File System (HDFS).



Working of DFS

4. NoSQL Database

Definition

A **Not-only SQL (NoSQL)** database is a non-relational database

It is **highly scalable, fault-tolerant and specifically designed** to house semi-structured and unstructured data.

They often provides an API-based query interface that can be called from within an application.

They also support query languages other than Structured Query Language (SQL) as SQL was designed to query structured data stored within a relational database.

An examples:

- a NoSQL database that is optimized to store XML files will often use XQuery as the query language.
- a NoSQL database designed to store RDF data will use SPARQL to query the relationships it contains.

5. Sharding

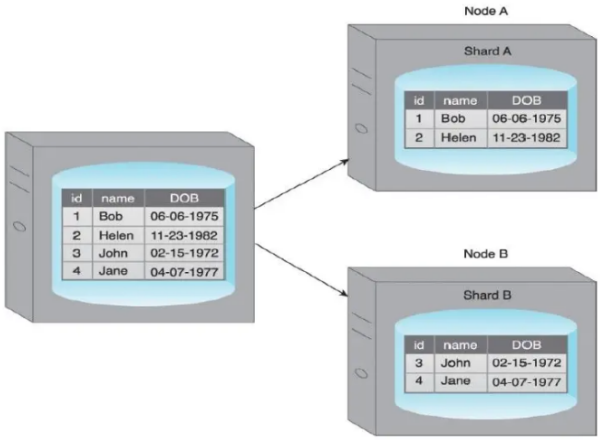
Definition

Sharding is the process of horizontally partitioning a large dataset into a collection of smaller, more manageable datasets called shards.

The shards are distributed across multiple nodes, where a node is a server or a machine.

Each shard

- It is stored on a separate node and each node 1s responsible for only the data stored on it.
- It shares the same schema, and all shards collectively represent the complete dataset.



An example of sharding where a dataset is spread across Node A and Node B, resulting in Shard A and Shard B, respectively

Extra

- Sharding allows the distribution of processing loads across multiple nodes to achieve horizontal scalability.
- Horizontal scaling is a method for increasing a system's capacity by adding similar or higher capacity resources alongside existing resources.
- Since each node is responsible for only a part of the whole dataset, read/write times are greatly improved.
- How sharding works in practice:
    1. Each shard can independently service reads and writes for the specific subset of data that it is responsible for.
    2. Depending on the query, data may need to be fetched from both shards.
- A benefit of sharding is that it provides partial tolerance toward failures.
- Incase of a node failure, only data stored on that node is affected.

6. Replication

Definition

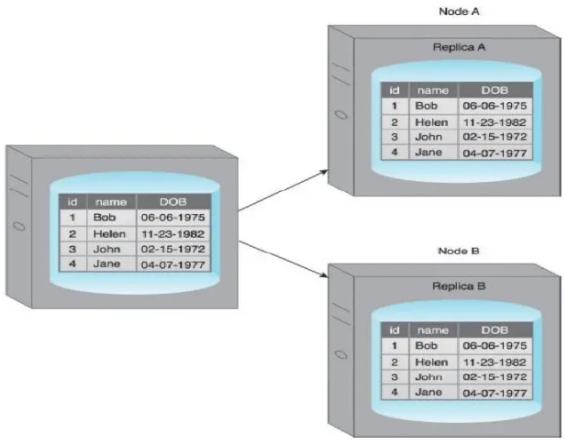
Replication stores multiple copies of a dataset, known as replicas, on multiple nodes.

Replication provides scalability and availability due to the fact that the same data is replicated on various nodes.

Fault tolerance is also achieved since data redundancy ensures that data is not lost when an individual node fails.

There are two different methods that are used to implement replication:

1. master-slave
2. peer-to-peer



An example of replication where a dataset is replicated to Node A and Node B, resulting in Replica A and Replica B.

Master-Slave Replication

Method

Nodes are arranged in a master-slave configuration, and all data is written to a master node.

Once saved, the data 1s replicated over to multiple slave nodes.

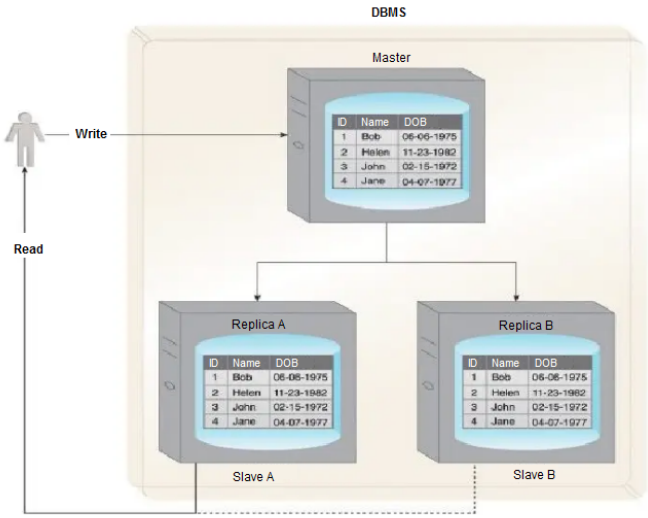
All external write requests, including insert, update and delete, occur on the master node, whereas read requests can be fulfilled by any slave node.

It is ideal for read intensive loads rather than write intensive loads since growing read demands can be managed by horizontal scaling to add more slave nodes.

Writes are consistent, as all writes are coordinated by the master node.

Write performance will suffer as the amount of writes increases.

If the master node fails, reads are still possible via any of the slave nodes.



An example of master-slave replication where Master A is the single point of contact for all writes, and data can be read from Slave A and Slave B.

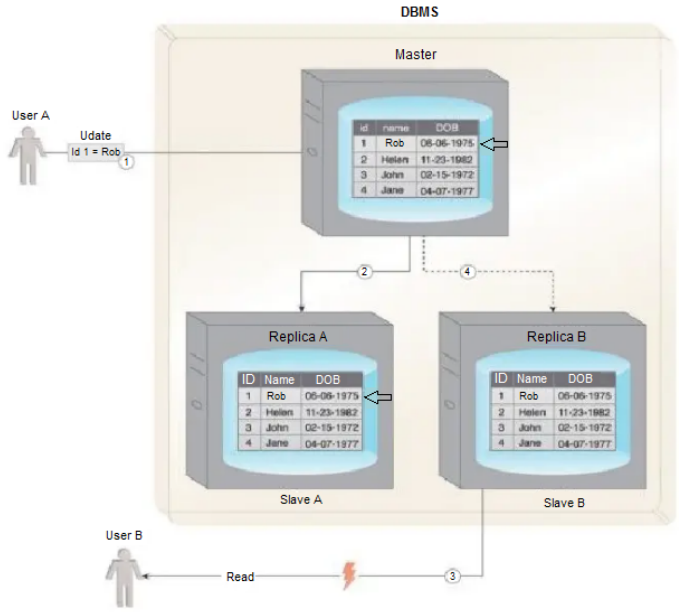
A **slave node** can be configured as a backup node for the master node.

Read **inconsistency**, which can be an issue if a slave node is read prior to an update to the master being copied to it.

To ensure read consistency, a voting system can be implemented where a read is declared consistent if the majority of the slaves contain the same version of the record.

Implementation of such a voting system requires a reliable and fast communication mechanism between the slaves.

1. User A updates data.
2. The data is copied over to Slave A by the Master.
3. Before the data is copied over to Slave B, User B tries to read the data from Slave B, which results in an inconsistent read.
4. The data will eventually become consistent when Slave B is updated by the Master.



An example of master-slave replication where read inconsistency occurs.

Peer-to-Peer Replication

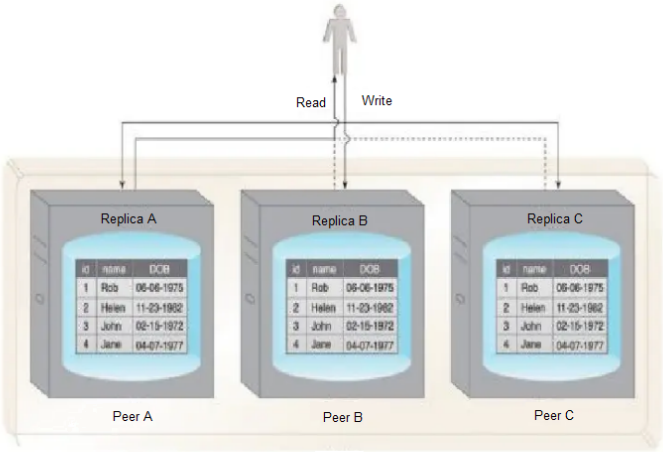
Method

With peer-to-peer replication, all nodes operate at the same level.

In other words, there is not a master-slave relationship between the nodes.

Each node, known as a peer, is equally capable of handling reads and writes.

Each write is copied to all peers.



Writes are copied to Peers A, B and C simultaneously. Data is read from Peer A, but it can also be read from Peers B or C.

Peer-to-peer replication is prone to write inconsistencies that occur as a result of a simultaneous update of the same data across multiple peers.

This can be addressed by implementing either a **pessimistic** or **optimistic** concurrency strategy.

- **Pessimistic** concurrency is a proactive strategy that prevents inconsistency.
    - It uses locking to ensure that only one update to a record can occur at a time. However, this is detrimental to availability since the database record being updated remains unavailable until all locks are released.
- **Optimistic** concurrency is a reactive strategy that does not use locking.

Instead, it allows inconsistency to occur with knowledge that eventually consistency will be achieved after all updates have propagated.

With optimistic concurrency, peers may remain inconsistent for some period of time before attaining consistency. However, the database remains available as no locking is involved.

Reads can be inconsistent during the time period when some of the peers have completed their updates while others perform their updates.

However, reads eventually become consistent when the updates have been executed on all peers.

To ensure read **consistency**, a voting system can be implemented where a read 1s declared consistent if the majority of the peers contain the same version of the record.

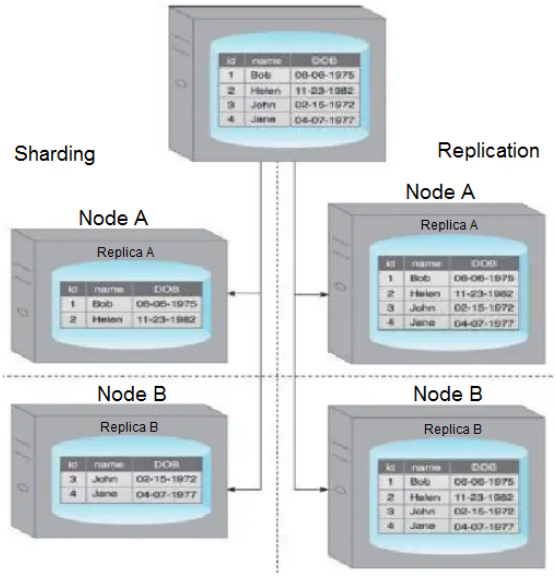
As previously indicated, implementation of such a voting system requires a reliable and fast communication mechanism between the peers.

Demonstrates a scenario where an inconsistent read occurs.

1. User A updates data.
2. 1. The data is copied over to Peer A.
    2. The data is copied over to Peer B.
3. Before the data is copied over to Peer C, User B tries to read the data from Peer C, resulting in an inconsistent read.
4. The data will eventually be updated on Peer C, and the database will once again become consistent.

Sharding and Replication *Note*

To improve on the limited **fault tolerance** offered by sharding, while additionally benefiting from the increased **availability and scalability** of replication, both sharding and replication can be combined

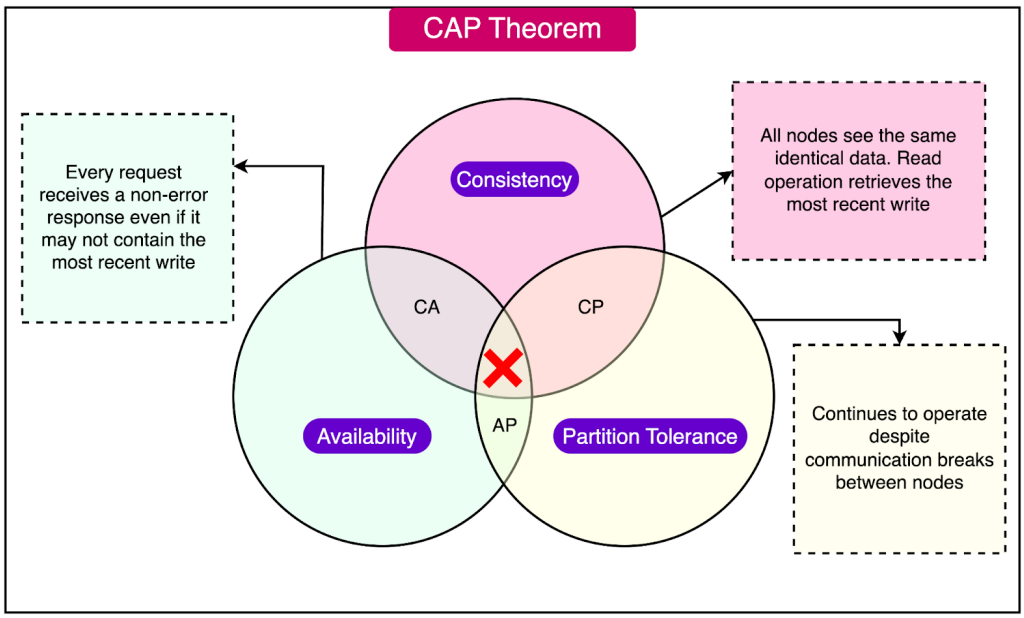


comparison of sharding and replication that shows how a dataset is distributed between two nodes with the different approaches

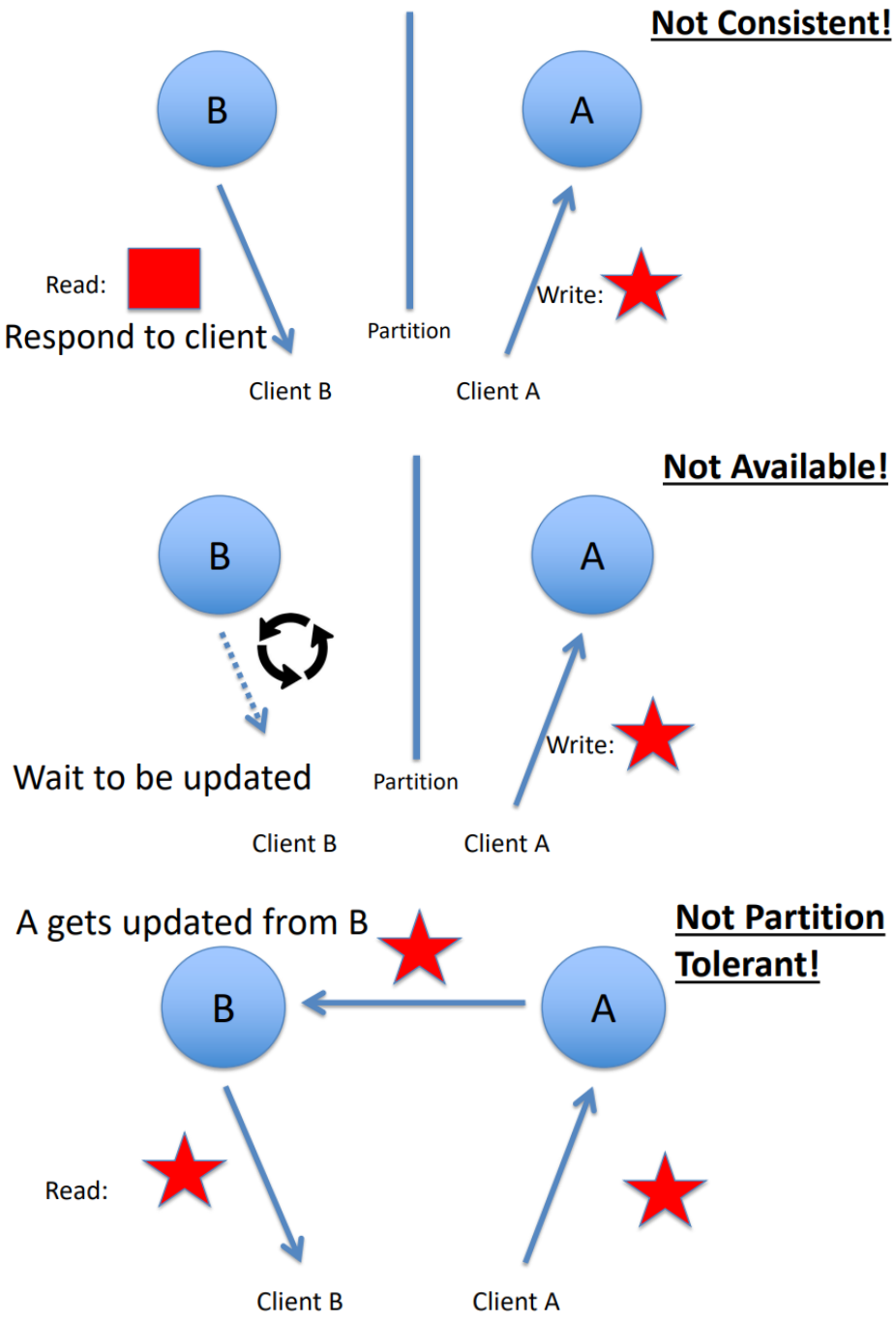
7. CAP and ACID Theorems

7.1. CAP Theorem

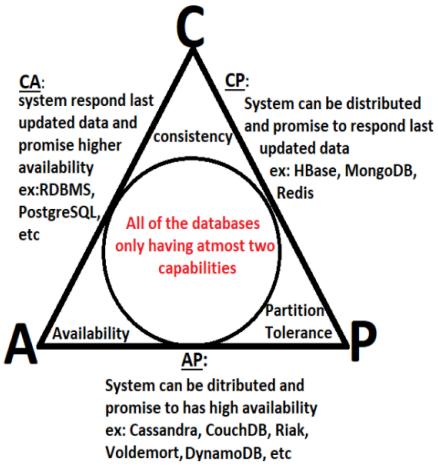
CAP Theorem *Definition*

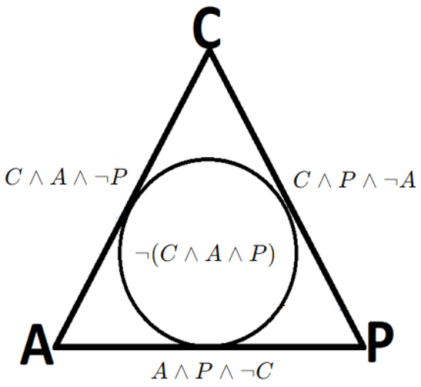


Example


Not Consistent!
B
A
Read:
Respond to client
Partition
Write:
Client B
Client A
Not Available!
B
A
Wait to be updated
Partition
Write:
Client B
Client A
A gets updated from B
Not Partition Tolerant!
B
A
Read:
Client B
Client A

Note

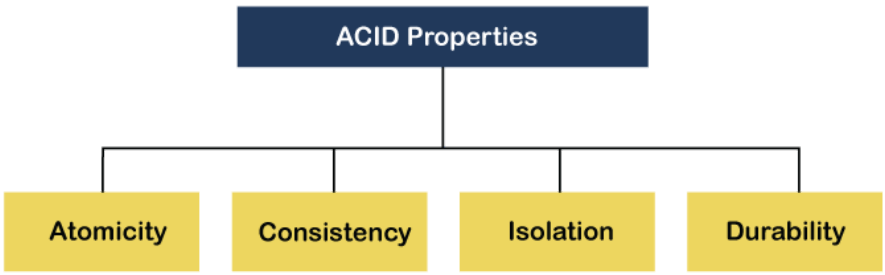




7.2. ACID Theorem

Definition

The **ACID theorem** refers to a set of properties that ensure reliable transaction processing in databases. These properties guarantee that database transactions are processed accurately, even in the presence of errors, power failures, or other disruptions. ACID stands for:



Atomicity

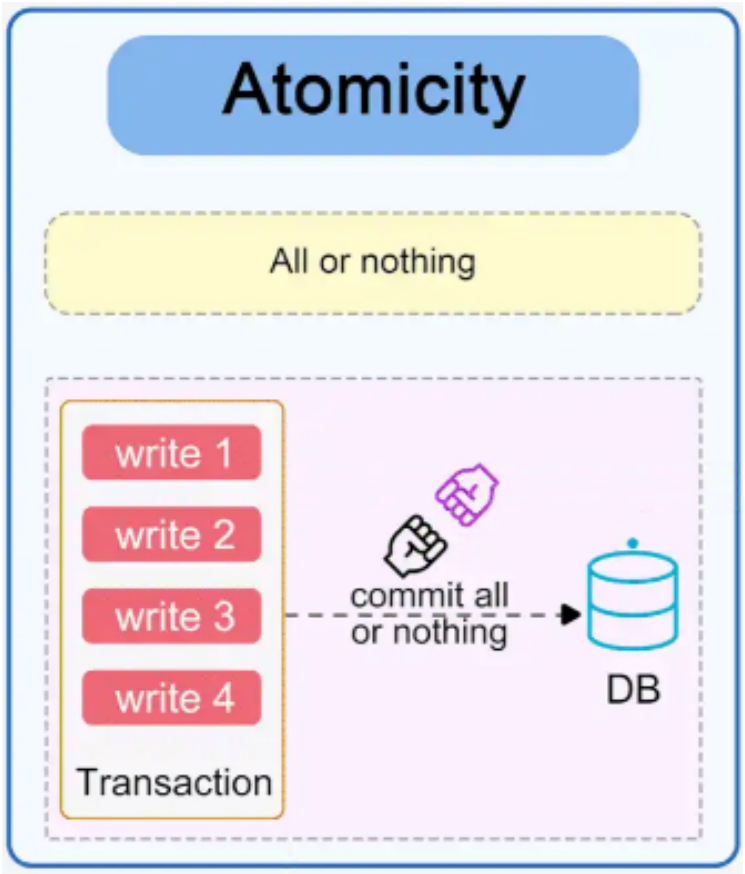
Extra

A transaction is treated as a single, indivisible unit. Either all its operations are completed successfully, or none are applied.

If any part of a transaction fails, the database remains unchanged, as if the transaction never occurred.

Example:

In a bank transfer, if money is debited from one account but cannot be credited to another due to an error, the entire transaction is rolled back.



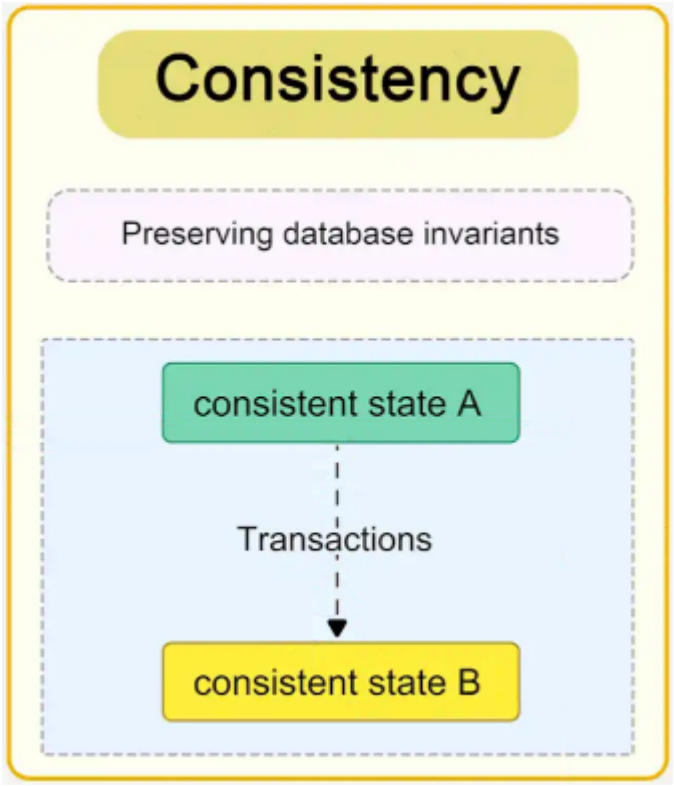
Consistency *Extra*

A transaction ensures that the database transitions from one valid state to another, maintaining all defined rules and constraints.

After a transaction completes, the database remains in a consistent state (e.g., constraints like unique keys, foreign keys, and checks are upheld).

Example:

In an inventory system, a transaction that decreases product stock cannot leave the stock in a negative state if the system enforces non-negative quantities.



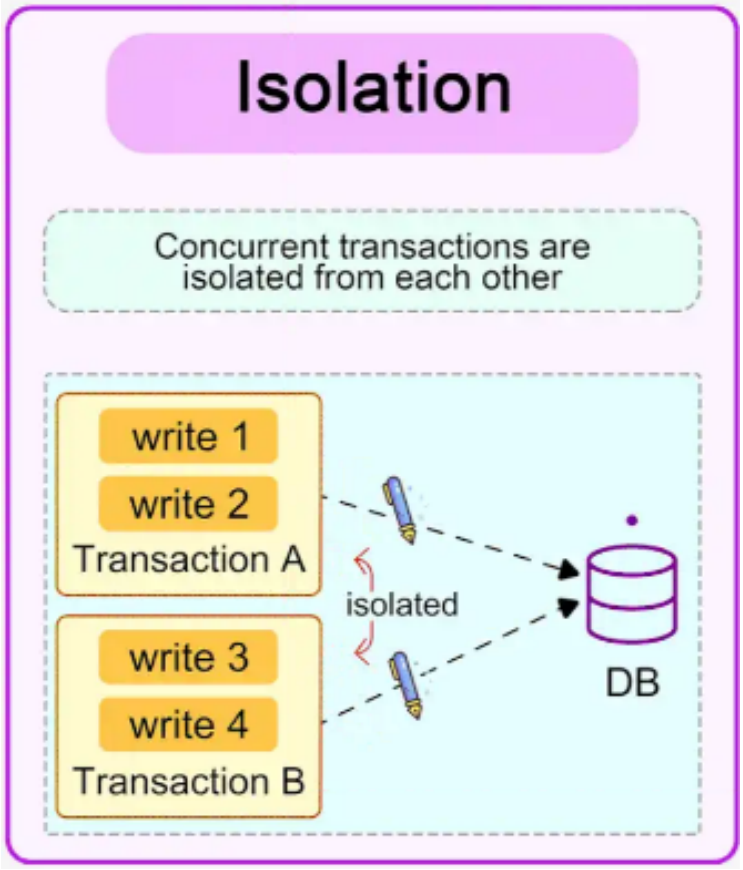
Isolation *Extra*

Transactions are executed independently and transparently. The operations of one transaction do not interfere with another.

Intermediate results of a transaction are not visible to other transactions until the transaction is committed.

Example:

Two customers cannot simultaneously purchase the last item in stock; the database ensures that one transaction completes before the other starts.



Durability

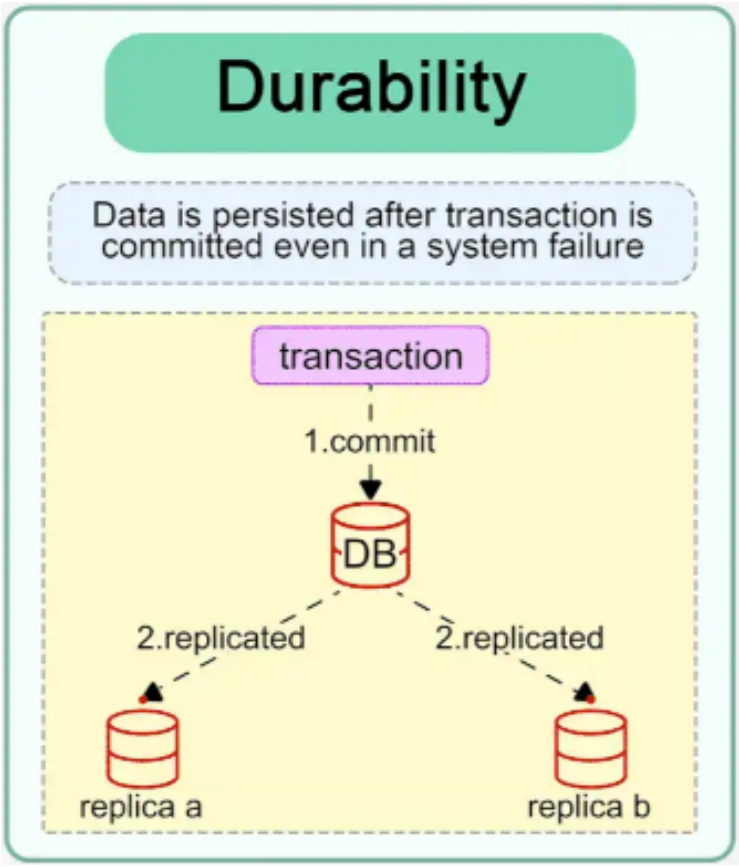
Extra

Once a transaction is committed, its results are permanent, even in the event of a system failure.

Changes made by a committed transaction are stored reliably (e.g., written to disk or replicated).

Example:

After confirming a flight booking, the reservation remains intact even if the server crashes immediately afterward.


Durability
Data is persisted after transaction is committed even in a system failure
transaction
1.commit
DB
2.replicated
2.replicated
replica a
replica b