Benefits of Relational Model:

- (Mostly) Standard Data Model and Query Language
    - The relational model is widely adopted across industries, providing a structured way to store and manage data in tables with well-defined relationships.
    - SQL (Structured Query Language) is the standard query language for relational databases, making it easier for developers, analysts, and data scientists to work with data across different database systems.
    - Most relational databases follow common conventions, allowing for easier migration between different database management systems (e.g., MySQL, PostgreSQL, SQL Server).

- ACID Compliance (Atomicity, Consistency, Isolation, Durability) - Ensures reliable transactions by enforcing four key properties:
    - Atomicity: A transaction is either fully completed or fully rolled back—no partial updates (aka transaction is treated as an atomic unit - it is fully executed or no parts of it are executed)
    - Consistency: The database remains in a valid state before and after a transaction (aka a transaction takes a database from one consistent state to another consistent state consistent state - all data meets integrity constraints)
    - Isolation: Transactions execute independently without interfering with each other
        - Two transactions T1 and T2 are being executed at the same time but cannot affect each other
        - If both T1 and T2 are reading the data - no problem
        - If T1 is reading the same data that T2 may be writing, can result in
            - Dirty Read: a transaction T1 is able to read a row that has been modified by another transaction T2 that hasn't yet executed a COMMIT
            - Non-repeatable Read: two queries in a single transaction T1 execute a SELECT but get different values because another transaction T2 has changed data and COMMITTED
            - Phantom Reads: when a transaction T1 is running and another transaction T2 adds or deletes rows from the set T1 is using
    - Durability: Once a transaction is committed, its changes are permanently stored, even in the case of a system failure.
        - Once a transaction is completed and committed successfully, its changes are permanent.
        - Even in the event of a system failure, committed transactions are preserved

- Works Well with Highly Structured Data:
    - Best suited for scenarios where data can be organized into predefined schemas with fixed relationships, such as financial records, inventory management, and customer databases.

    - Ensures data integrity through constraints like primary keys, foreign keys, and unique constraints.
    - Enables efficient joins and aggregations, which are crucial for analytical and transactional workloads.

- Can Handle Large Amounts of Data
    - Optimized indexing, partitioning, and query optimization techniques allow relational databases to scale efficiently.
    - Many enterprise-grade relational databases support horizontal and vertical scaling, handling millions or even billions of rows.
    - Used in large-scale applications like banking systems, airline reservation systems, and enterprise resource planning (ERP) software.

- Well Understood, Lots of Tooling, Lots of Experience
    - Relational databases have been around for decades, leading to a wealth of knowledge, best practices, and mature ecosystem support.
    - A large number of tools exist for database administration, performance tuning, data migration, and backup management.
    - Many database professionals, from developers to DBAs, have deep experience in relational databases, making hiring and knowledge transfer easier for organizations.

Relational Database Performance

- Many techniques help relational database management systems (RDBMS) optimize efficiency, ensuring fast queries, reduced latency, and scalable performance.

1. *Indexing*
    - Indexes improve query performance by allowing the database to locate data quickly rather than scanning entire tables.
    - Common types of indexes:
        - B-Tree Indexes: Used for range and equality searches.
        - Hash Indexes: Ideal for exact-match lookups.
        - Bitmap Indexes: Efficient for low-cardinality columns (e.g., gender, Boolean values).
        - Proper indexing speeds up read operations but can slightly slow down writes due to index maintenance.
2. *Directly Controlling Storage*
    - RDBMSs manage how data is physically stored on disk, optimizing storage structures for performance.
    - Techniques include:
        - Data compression to reduce disk I/O.
        - Page-level optimizations for fast retrieval.
        - Efficient allocation of memory buffers and disk space.

3. *Column-Oriented Storage vs. Row-Oriented Storage*
    - Row-Oriented Storage: Stores entire rows together; efficient for transactional (OLTP) workloads where frequent inserts, updates, and deletes occur.
    - Column-Oriented Storage: Stores data by columns instead of rows, improving performance for analytical (OLAP) queries that aggregate large amounts of data. Reduces disk I/O by only retrieving necessary columns.
4. *Query Optimization*
    - RDBMSs analyze SQL queries to find the most efficient execution plan.
    - Techniques include: Using cost-based optimization to determine the lowest-cost query plan; Reordering joins to minimize processing time; Pushdown filtering to reduce the amount of processed data.
5. *Caching/Prefetching*
    - Caching: Frequently accessed query results are stored in memory to reduce redundant computations.
    - Prefetching: Anticipates future queries and loads relevant data into memory ahead of time.
    - Reduces disk I/O bottlenecks and speeds up query execution.
6. *Materialized Views*
    - A materialized view is a precomputed result of a query stored as a table.
    - Unlike regular views, which recompute results each time they are queried, materialized views store results persistently.
    - Commonly used in reporting and analytics to speed up expensive aggregations and joins.
7. *Precompiled Stored Procedures*
    - Stored procedures are precompiled SQL queries that execute faster because they don't need to be re-parsed every time.
    - Benefits: Reduces query execution time, Improves security by limiting direct access to raw tables, Allows complex logic to be executed at the database level.
8. *Data Replication and Partitioning*
    - Replication: Copies data across multiple database instances to improve availability and fault tolerance
        - Common types: Master-slave replication (one primary, multiple read replicas); Multi-master replication (multiple writable nodes).
    - Partitioning: Splits large tables into smaller, more manageable chunks to improve query performance.
        - **Horizontal partitioning**: Divides rows across different tables (e.g., by region, time range).
        - **Vertical partitioning**: Stores specific columns in separate tables to optimize read performance.

Transaction Processing

- Transaction: a sequence of one or more of the CRUD operations performed as a single, logical unit of work
    - Either the entire sequence succeeds (COMMIT)

    - OR the entire sequence fails (ROLLBACK or ABORT)
- Help ensure: Data Integrity, Error Recovery, Concurrency Control, Reliable Data Storage, Simplified Error Handling

Transaction Processing

What is a Transaction?

- A transaction is a sequence of one or more CRUD operations (Create, Read, Update, Delete) performed as a single, logical unit of work. Transactions ensure data consistency and reliability in database systems.
    - All operations must either complete successfully together (COMMIT) or fail together (ROLLBACK or ABORT).
    - If any part of the transaction encounters an error, the entire sequence is reversed to maintain data integrity.

How Transactions Work

1. Begin Transaction – The system starts a new transaction.
2. Perform CRUD Operations – Multiple database operations occur (e.g., inserting a record, updating a balance).
3. Validation & Checks – The system ensures constraints and consistency rules are met.
4. Commit or Rollback
    - COMMIT: If all operations succeed, changes are saved permanently.
    - ROLLBACK (ABORT): If any operation fails, all changes are undone, ensuring the database remains in a consistent state.

Why Are Transactions Important?

Transactions play a crucial role in maintaining a reliable and robust database system by ensuring:

1. **Data Integrity**: Transactions maintain database correctness by ensuring that only valid and complete changes are saved.
    a. Prevents issues such as half-completed updates or lost data during failures.
2. **Error Recovery**: If a system crashes or a query fails, the database can roll back to a stable state.
    a. Ensures that no partial or inconsistent data is left behind.
3. **Concurrency Control**: Multiple users can safely perform operations on the database at the same time.
    a. Prevents issues like lost updates, dirty reads, and inconsistent data retrieval.
4. **Reliable Data Storage:** Transactions ensure that once data is committed, it remains stored even in the event of a power failure or system crash.
    a. Guarantees Durability, one of the key ACID properties.
5. **Simplified Error Handling:** Developers can focus on application logic without worrying about data corruption.
    a. If an error occurs, rolling back ensures a clean database state.

Example Transaction - Transfer $$

```
DELIMITER //

CREATE PROCEDURE transfer(
   IN sender_id INT,
   IN receiver_id INT,
   IN amount DECIMAL(10,2)
)
BEGIN
   DECLARE rollback_message VARCHAR(255)
         DEFAULT 'Transaction rolled back: Insufficient funds';
   DECLARE commit_message VARCHAR(255)
DEFAULT 'Transaction committed successfully';

   -- Start the transaction
   START TRANSACTION;

   -- Attempt to debit money from account 1
   UPDATE accounts SET balance = balance - amount WHERE account_id = sender_id;

   -- Attempt to credit money to account 2
   UPDATE accounts SET balance = balance + amount WHERE account_id = receiver_id;

   -- Check if there are sufficient funds in account 1
   -- Simulate a condition where there are insufficient funds
   IF (SELECT balance FROM accounts WHERE account_id = sender_id) < 0 THEN
      -- Roll back the transaction if there are insufficient funds
      ROLLBACK;
      SIGNAL SQLSTATE '45000'          -- 45000 is unhandled, user-defined error
         SET MESSAGE_TEXT = rollback_message;
   ELSE
      -- Log the transactions if there are sufficient funds
      INSERT INTO transactions (account_id, amount, transaction_type)
                  VALUES (sender_id, -amount, 'WITHDRAWAL');
      INSERT INTO transactions (account_id, amount, transaction_type)
VALUES (receiver_id, amount, 'DEPOSIT');

      -- Commit the transaction
      COMMIT;
      SELECT commit_message AS 'Result';
   END IF;
END //

DELIMITER ;
```

Relational Database Problems

While relational databases are powerful and widely used, they may not be the best choice for every use case. Some challenges and limitations include:

1. Schemas Evolve Over Time
    - Rigid Schema Structure:
        - Relational databases require predefined schemas, meaning tables and relationships must be designed before data is stored.
        - As applications grow and change, modifying the schema (e.g., adding new columns, changing relationships) can be cumbersome and may require schema migrations, which can be slow and risky in large databases.
    - Alternative Solutions:
        - NoSQL databases (e.g., MongoDB, Cassandra) provide schema flexibility, allowing fields to be added dynamically without disrupting existing data.
2. Not All Applications Need Full ACID Compliance
    - ACID compliance ensures strong consistency, but at a cost:
        - Transactions in relational databases follow Atomicity, Consistency, Isolation, and Durability, which can introduce performance overhead, especially in distributed systems.
        - Some applications prioritize availability and speed over strong consistency.
    - Alternative Solutions:
        - Eventual consistency models in NoSQL databases (e.g., Amazon DynamoDB, Apache Cassandra) are often preferred for applications where absolute consistency is not critical (e.g., social media feeds, recommendation engines).
3. Joins Can Be Expensive
    - Performance Bottlenecks:
        - SQL queries involving multiple joins can become computationally expensive, especially on large datasets.
        - Each join operation requires scanning and matching rows from multiple tables, which can lead to high CPU and memory usage.
    - Alternative Solutions:
        - Denormalization (storing related data together) is sometimes preferred in NoSQL databases to avoid expensive join operations.
        - Columnar databases (e.g., Apache Parquet, ClickHouse) optimize analytical workloads by reducing the need for joins.
4. A Lot of Data Is Semi-Structured or Unstructured (JSON, XML, etc.)
    - Relational databases are designed for structured data:
        - They work best when data fits neatly into rows and columns.
        - However, modern applications generate semi-structured (JSON, XML) or unstructured (text, images, videos) data that doesn't always fit a tabular format.
    - Alternative Solutions:
        - NoSQL document stores (e.g., MongoDB, Firebase) allow for flexible JSON-like storage.
        - Search engines like Elasticsearch and OpenSearch are optimized for handling unstructured text and logs.
5. Horizontal Scaling Presents Challenges
    - Scaling relational databases across multiple servers is complex:
        - Relational databases traditionally scale vertically (by adding more CPU, RAM, or storage to a single server).

        - Scaling horizontally (distributing data across multiple servers) requires techniques like sharding, replication, and distributed transactions, which add complexity.
    - Alternative Solutions:
        - NoSQL databases like Cassandra and DynamoDB are natively designed for horizontal scaling, handling massive amounts of data across distributed nodes with ease.
6. Some Applications Require More Performance (Real-Time, Low Latency Systems)
    - Relational databases can be too slow for real-time applications:
        - High-throughput, low-latency applications (e.g., stock trading platforms, real-time analytics, gaming leaderboards) need sub-millisecond response times.
        - RDBMS architectures, with their strict consistency and transaction management, can introduce delays.
    - Alternative Solutions:
        - In-memory databases (e.g., Redis, Memcached) provide ultra-fast key-value lookups.
        - Time-series databases (e.g., InfluxDB, TimescaleDB) are optimized for time-based data analysis.

Scalability – Up or Out?

Conventional Wisdom: Scale Vertically (Up) Until High Availability Demands Horizontal Scaling (Out)

- Vertical Scaling (Scaling Up):
    - Definition: Scaling up refers to adding more resources (CPU, RAM, storage) to a single server to handle increased workload.
    - Why it's common:
        - It's simple and relatively cost-effective in the short term.
        - No architecture changes are needed; you just upgrade the existing server hardware.
        - Often sufficient for many small to medium-sized applications.
    - Limitations:
        - Physical limits: At some point, hardware upgrades become prohibitively expensive or ineffective. You can't keep adding more CPUs, RAM, or storage indefinitely.
        - Single point of failure: With a vertically scaled system, if the server goes down, the entire application can become unavailable. This poses availability and redundancy challenges, especially in mission-critical systems.
        - Cost efficiency: The larger the system, the more expensive it is to upgrade. There's a diminishing return on investment with each incremental upgrade.

- Horizontal Scaling (Scaling Out):
    - Definition: Scaling out means adding more servers or nodes to distribute the load, effectively splitting the workload across multiple machines.
    - Why it's challenging:
        - Requires distributed computing models (e.g., sharding, replication), which can introduce complexity in managing consistency, coordination, and fault tolerance.
        - Managing a large number of machines introduces complexity in system design (network communication, load balancing, etc.), and data consistency must be carefully handled.

    - Why it's necessary:
        - For high-traffic applications or applications requiring high availability and fault tolerance, scaling vertically alone eventually becomes insufficient.
        - High-demand apps (e.g., social media platforms, e-commerce, cloud services) often need to serve millions or billions of users, requiring distributed systems that can horizontally scale.

- Why Scaling Up (Vertical Scaling) is Often Easier
    - Simple Upgrades:
        - It's often easier to just increase the resources of an existing machine than to architect a distributed system.
        - No need to redesign the application: Vertical scaling doesn't require significant changes in the system architecture or application code.
    - Less Complex Infrastructure:
        - No need to worry about network communication, data sharding, or distributed transactions.
    - Fewer Management Overheads:
        - Fewer machines mean fewer maintenance tasks, backups, and monitoring systems.

- Practical and Financial Limits to Scaling Up
    - Physical Hardware Constraints:
        - Eventually, you'll hit hardware limits. For example, there's a limit to the number of CPUs, RAM, and storage you can add to a single server.
        - As machines grow more powerful, the cost of upgrading them grows exponentially. High-end servers can be very expensive compared to commodity servers.
    - Single Point of Failure:
        - If your vertically scaled system crashes, everything crashes. There's no inherent redundancy unless you add complex failover mechanisms.
        - High availability and fault tolerance are limited unless you implement replication or clustering, which pushes the system toward horizontal scaling.

Modern Systems That Make Horizontal Scaling Less Problematic
- Distributed Databases:
    - Modern distributed databases (e.g., Cassandra, Google Spanner) are designed to scale horizontally with built-in mechanisms for data partitioning (sharding), replication, and eventual consistency.
    - They handle much of the complexity of scaling, making horizontal scaling much more accessible for modern systems.
- Cloud Computing and Serverless Architectures:
    - Cloud platforms (e.g., AWS, Google Cloud, Azure) allow dynamic horizontal scaling, where resources (such as virtual machines or containers) can be added automatically based on demand.
    - Serverless frameworks (e.g., AWS Lambda, Azure Functions) enable scaling on-demand without worrying about managing infrastructure.
    - These platforms provide auto-scaling capabilities that automatically spin up new resources as traffic increases and scale them down when demand drops.

- Microservices:
    - Microservices architectures decompose applications into smaller, more manageable components, which can be scaled independently.
    - This allows individual services to be scaled out independently, making it easier to handle varying levels of load across different parts of an application.

- Containerization (Docker, Kubernetes):
    - Containers allow for easier deployment and scaling of applications across distributed environments.
    - Kubernetes, for example, simplifies the process of managing containers and automatically scaling services as needed.

So What? Distributed Data when Scaling Out

What is a Distributed System?

- A distributed system is a network of independent computers that work together to provide a unified service, appearing to users as a single system. As Andrew Tannenbaum puts it:
    - *"A distributed system is a collection of independent computers that appear to its users as one computer."*
- When scaling out in a distributed system, multiple machines (or nodes) handle different pieces of the workload, allowing the system to scale horizontally. This enables systems to handle larger amounts of data and more requests concurrently, but it also introduces a unique set of challenges.

Characteristics of Distributed Systems

Distributed systems have several key characteristics that differentiate them from traditional, monolithic systems:

1. **Computers Operate Concurrently**:
    - Concurrency means that multiple processes or threads can be executed in parallel across different machines.
        - Each machine in a distributed system may be working on different tasks at the same time, contributing to the overall processing power of the system.
    - This concurrent processing allows distributed systems to handle a higher volume of requests and tasks efficiently. For example, one node may be processing user requests, another may be updating a database, while a third node handles background jobs.
    - Challenges:
        - Synchronization between machines can be tricky, especially when they must share or access common data.
        - This requires effective communication mechanisms and often a distributed consensus to ensure that all parts of the system are working in harmony.
2. **Computers Fail Independently**
    - Independent Failures: In a distributed system, each machine can fail independently of the others. This means that even if one node goes down, the others can continue to function.
        - This fault tolerance is one of the primary benefits of distributed systems, as failures do not bring down the entire system.
    - Challenges:
        - Fault detection and recovery become complex when systems are distributed. How do we know a machine has failed, and how do we ensure it recovers or that its tasks are redistributed to healthy machines?

- Data consistency and availability can be impacted by node failures, which must be handled carefully through techniques like replication and partitioning.
- Example: If a server in a database cluster fails, replicas of the data stored on other machines can still serve requests, but the system must know which copies are most up-to-date and reliable.

3. **No Shared Global Clock**
    - No Global Clock means there is no single time source or synchronized clock across all machines in a distributed system. Each machine operates based on its local time, leading to discrepancies across the system.
        - Without a shared clock, it's impossible to guarantee that all events in the system happen in a perfectly synchronized manner.
        - This has important implications for tasks like ordering operations (e.g., in transaction processing or event logs) and event coordination across nodes.
    - Challenges:
        - Clock skew: Differences in time across nodes can cause issues with event ordering. For example, determining the exact sequence of operations in a distributed database might be difficult.
        - Distributed systems often use techniques like logical clocks (e.g., Lamport timestamps) to maintain an order of events across machines without relying on synchronized physical clocks.

Challenges in Scaling Out with Distributed Data

While distributed systems offer scalability and fault tolerance, they come with inherent challenges:

1. **Data Partitioning (Sharding)**
    - Sharding is the process of dividing a large dataset into smaller, more manageable parts (shards), which can be distributed across multiple machines.
    - Challenges:
        - Deciding on the best strategy for partitioning data. For example, which data should go on which machine, and how do we ensure balanced distribution of the load?
        - Handling cross-shard queries can be complex and inefficient. If a query needs data from multiple shards, it must be coordinated and consolidated efficiently.
2. **Data Replication**
    - Replication involves creating multiple copies of data on different machines to ensure high availability and fault tolerance.
    - Challenges:
        - Managing consistency across replicas: How do we make sure all replicas are up-to-date after a change is made?
        - Handling replication lag: When updates are made to one replica, it might take time for the changes to propagate to others, potentially leading to inconsistent reads.
        - Distributed databases often employ eventual consistency or use techniques like Quorum-based replication to strike a balance between consistency and availability.
3. **Communication and Latency**
    - In a distributed system, nodes must communicate over a network, which can introduce latency and potential bottlenecks.
    - Challenges:

- Ensuring low-latency communication between nodes while scaling out to a large number of machines.
- Handling network partitions, where communication between certain nodes is temporarily broken. This requires sophisticated strategies for maintaining availability and consistency during these partitions (e.g., CAP theorem considerations).

4. **Consistency and Consensus**
    - Achieving consistency in a distributed system can be difficult, especially when multiple nodes are involved.
    - Challenges:
        - Distributed consensus protocols (e.g., Paxos, Raft) are often needed to agree on the state of the system across nodes. These protocols can be complex and costly in terms of performance but are crucial for maintaining consistency.
        - CAP theorem: It's impossible to guarantee consistency, availability, and partition tolerance all at once in a distributed system, leading to trade-offs.

Distributed Data Stores

What Are Distributed Data Stores?

- A distributed data store is a system where data is stored across multiple machines or nodes rather than on a single server. The goal is to achieve high availability, scalability, and fault tolerance by spreading data across different locations.
    - **Replication**: A key feature of distributed data stores is data replication, where each block of data is typically replicated across multiple nodes to ensure redundancy and fault tolerance. For example, a database might store the same data on N nodes, ensuring that even if one or more nodes fail, the data can still be accessed from the other nodes.
    - **Sharding**: Data is partitioned into smaller, more manageable pieces (called shards) that are distributed across nodes. Each shard contains a subset of the overall data, helping to improve performance and scalability.

Types of Distributed Databases

Distributed databases can either be relational or non-relational (NoSQL), with both types supporting distribution, replication, and sharding in different ways.

1. **Relational Distributed Databases**: Some traditional relational databases have evolved to support distributed configurations:
    - MySQL and PostgreSQL are examples of relational databases that offer replication (the process of maintaining copies of the same data on different nodes) and sharding (splitting data into subsets that reside on different machines).
    - These systems typically require additional tools or setups (like MySQL Cluster, PgSQL's logical replication, or PostgreSQL partitioning) to scale horizontally and achieve distribution across multiple nodes.
        - Replication ensures data availability and durability in case of failures.
        - Sharding helps distribute the load across multiple nodes to improve scalability, though managing distributed transactions and maintaining consistency becomes more complex in these cases.
2. **NoSQL Distributed Databases:** Many NoSQL databases were designed with distributed systems in mind, providing more inherent support for sharding and replication:
    - Cassandra, MongoDB, and Couchbase are some examples of distributed NoSQL systems that automatically handle sharding and replication.

- NoSQL databases often favor eventual consistency over strict ACID compliance, meaning that while data is eventually consistent across all nodes, it may not be immediately consistent after updates. This allows for more flexibility in distributed configurations but requires careful attention to potential inconsistencies.
- Cassandra uses a peer-to-peer model and supports replication and partitioning, with the ability to scale horizontally across many nodes.
- MongoDB offers replication and sharding out of the box, making it suitable for horizontally scalable applications.

3. **Newer Players**
    - CockroachDB:
        - A newer distributed relational database, CockroachDB was designed to provide horizontal scalability and high availability with strong consistency (supporting the ACID properties of transactions). It automatically handles sharding and replication, and its architecture is similar to that of Google Spanner, focusing on global distribution.
        - It's a distributed SQL database that offers strong consistency, allowing users to scale applications globally while maintaining the guarantees typically associated with relational databases.

<u>Key Characteristics of Distributed Data Stores</u>

Replication: Replication ensures that data is available even when some nodes or machines fail. Typically, each data block (or row) is replicated across multiple nodes (often N nodes, with N being a configurable number).

- Synchronous replication ensures that data is written to all nodes before acknowledging the write.
- Asynchronous replication writes data to one node first and later synchronizes it to other replicas. This is often faster but can lead to eventual consistency.
- Challenges: Managing consistency between replicas can be complex. There is always a trade-off between performance and consistency, particularly when nodes are located in different regions (latency issues can arise).

Sharding: Sharding involves splitting data into smaller chunks called shards, each of which resides on a separate node. This partitioning is usually done based on a shard key (e.g., user ID or region) to ensure that data is distributed evenly across nodes.

- Advantages: Sharding allows for horizontal scalability, meaning you can add more nodes to the system to distribute the load as demand increases.
- Challenges:
    - Managing cross-shard queries can be complex and potentially slower. If data is spread across different shards, queries that need data from multiple shards may incur additional overhead.
    - Rebalancing shards across nodes can be difficult, especially when data grows unevenly, requiring manual intervention or complex algorithms to ensure even distribution.

Network Partitioning is Inevitable

One of the most important principles in distributed systems is that network partitioning is inevitable, meaning that at some point, the network may experience temporary failures, leading to split-brain situations where some nodes can no longer communicate with others.

- What is Network Partitioning?
    - Network partitioning occurs when there's a disruption in the communication between parts of the system, resulting in some nodes being cut off from others.
    - For example, if a failure occurs in the network between two data centers, the systems on each side of the partition may be isolated and unable to communicate with each other.

- System Needs to Be Partition Tolerant
    - Partition Tolerance refers to the ability of the system to continue functioning even when parts of the network are unavailable. This is crucial in distributed systems because network failures are inevitable and cannot always be prevented.
    - According to the CAP Theorem (states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees), a distributed system can only guarantee two of the following three properties:
        - **Consistency**: Every read operation sees the most recent write.
            - Every user of the DB has an identical view of the data at any given instant
                - *Consistency + Availability: System always responds with the latest data and every request gets a response, but may not be able to deal with network issues*
        - **Availability**: Every request to the system gets a response (either success or failure).
            - In the event of a failure, the database remains operational
            - *Availability + Partition Tolerance: System always sends are responds based on distributed store, but may not be the absolute latest data.*
        - **Partition Tolerance**: The system continues to operate even if there's a network partition.
            - The database can maintain operations in the event of the network's failing between two segments of the distributed system
                - *Consistency + Partition Tolerance: If system responds with data from a distributed store, it is always the latest, else data request is dropped.*
        - *What CAP is really saying: If you cannot limit the number of faults, requests can be directed to any server, and you insist on serving every request, then you cannot possibly be consistent (But it is interpreted as: You must always give up something: consistency, availability, or tolerance to failure)*
    - In practice, most distributed systems opt for Partition Tolerance and Availability, as network failures are common. This often leads to eventual consistency (e.g., Cassandra, MongoDB).