**Partitioning**

See how we can make our system scalable by partitioning.

**We'll cover the following**

* [Scalability](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6142386220826624#Scalability)
* [Mechanism to achieve scalability](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6142386220826624#Mechanism-to-achieve-scalability)
  + [Partitioning](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6142386220826624#Partitioning)
    - [Vertical partitioning](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6142386220826624#Vertical-partitioning)
    - [Horizontal partitioning](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6142386220826624#Horizontal-partitioning)
* [Limitations of partitioning](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6142386220826624#Limitations-of-partitioning)

One of the major benefits of distributed systems is **scalability**.

**Scalability**

Scalability lets us store and process datasets much larger than what we could with a single machine.



## Mechanism to achieve scalability

One of the primary mechanisms of achieving scalability is **partitioning**.

### Partitioning

Partitioning is the process of splitting a dataset into multiple, smaller datasets, and then assigning the responsibility of storing and processing them to different nodes of a distributed system. This allows us to add more nodes to our system and increase the size of the data it can handle.

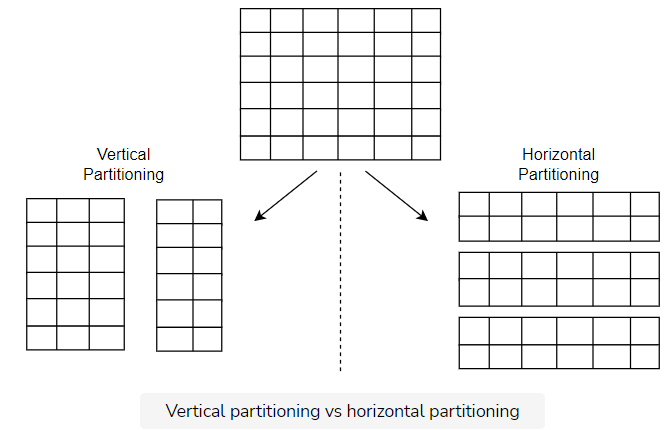
There are two different variations of partitioning:

1. Vertical partitioning
2. Horizontal partitioning (or **sharding**)

The terms “vertical” and “horizontal” originate from the era of relational databases that established the notion of a tabular view of data.

In this view, data consists of rows and columns, where each row is a different entry in the dataset, and each column is a different attribute for every entry.

The following illustration contains a visual depiction of the difference between **vertical partitioning** and **horizontal partitioning**.



#### Vertical partitioning

Vertical partitioning involves splitting a table into multiple tables with fewer columns and using additional tables to store columns that relate rows across tables. We commonly refer to this as a **join operation**. We can then store these different tables in different nodes.

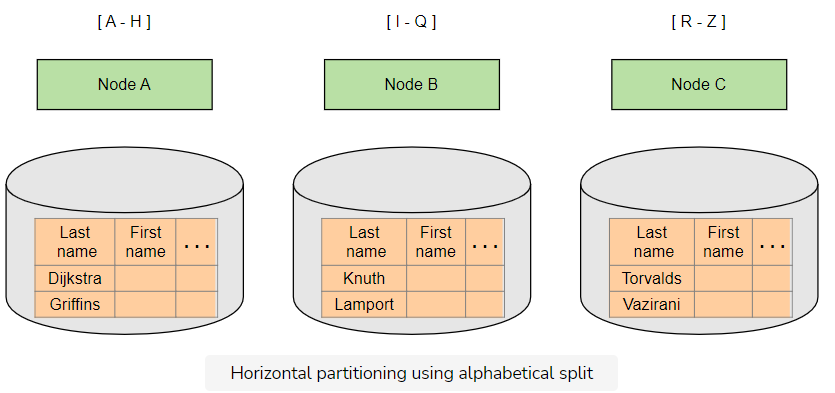
[**Normalization**](https://en.wikipedia.org/wiki/Database_normalization) is one way to perform vertical partitioning. However, general vertical partitioning goes far beyond that: it splits a column, even when they are normalized.

#### Horizontal partitioning

Horizontal partitioning involves splitting a table into multiple, smaller tables, where each table contains a percentage of the initial table’s rows. We can then store these different sub-tables in different nodes.

We can perform this split through multiple strategies.

A simplistic approach for this is an alphabetical split. For instance, we can horizontally partition a table that contains the students of a school by using the students’ surnames. The following illustration shows how.



## Limitations of partitioning

In a **vertically partitioned system**, requests that need to combine data from different tables (i.e., join operations) become less efficient. This is because these requests may now have to access data from multiple nodes.

In a **horizontally partitioned system**, we can usually avoid accessing data from multiple nodes because all the data for each row is located in the same node. However, we may still need to access data from multiple nodes for requests that are searching for a range of rows that belong to multiple nodes.

Another important implication of horizontal partitioning is the potential for loss of transactional semantics.

When we store data in a single machine, we can easily perform multiple operations in an atomic way, where either all or none of them succeed. However, this is much harder to achieve in a distributed system.

As a result, it’s much harder to perform atomic operations—when partitioning data horizontally—over data that resides in different nodes.

This is a common theme in distributed systems; there’s no silver bullet. We have to make trade-offs to achieve the property we desire.

Vertical partitioning is mainly a data modeling practice, which can be performed by the engineers designing a system—sometimes independently of the storage systems used. However, horizontal partitioning is a common feature of distributed databases. So, to use these systems properly, engineers need to know how the system works under the hood. Therefore, we will mostly focus on horizontal partitioning.

# Algorithms for Horizontal Partitioning

Let's look into the algorithms used for horizontal partitioning.

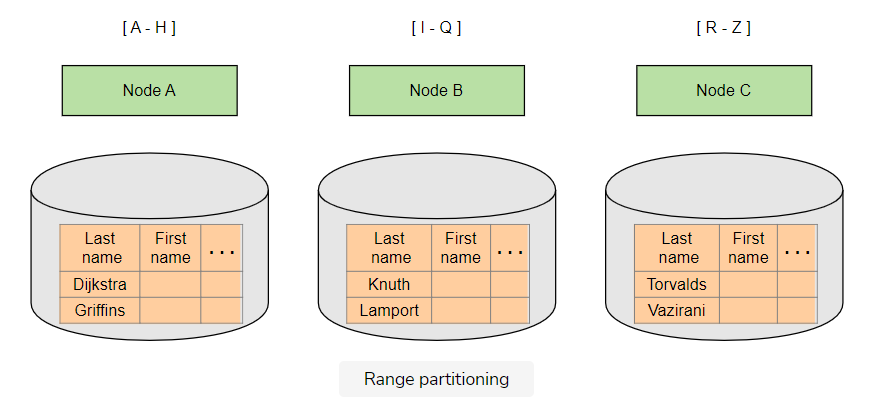
**We'll cover the following**

* [Range partitioning](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6487003491467264#Range-partitioning)
  + [Advantages of range partitioning](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6487003491467264#Advantages-of-range-partitioning)
  + [Disadvantages of range partitioning](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6487003491467264#Disadvantages-of-range-partitioning)
* [Hash partitioning](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6487003491467264#Hash-partitioning)
  + [Advantages of hash partitioning](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6487003491467264#Advantages-of-hash-partitioning)
  + [Disadvantages of hash partitioning](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6487003491467264#Disadvantages-of-hash-partitioning)
* [Consistent hashing](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6487003491467264#Consistent-hashing)
  + [Advantages of consistent hashing](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6487003491467264#Advantages-of-consistent-hashing)
  + [Disadvantages of consistent hashing](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6487003491467264#Disadvantages-of-consistent-hashing)

There are a lot of different algorithms we can use to perform horizontal partitioning. We will study some of these algorithms, and discuss their advantages and drawbacks.

## Range partitioning

**Range partitioning** is a technique where we split a dataset into ranges according to the value of a specific attribute. We then store each range in a separate node. The case we described [earlier](https://www.educative.io/collection/page/10370001/4891237377638400/4736079922462720#horizontal-partitioning)-with the alphabetical split-is an example of range partitioning.



Of course, the system should store and maintain a list of all these ranges and map which node stores a specific range. In this way, the system consults this node map whenever the system receives a request for a specific value (or a range of values) to identify which node (or nodes, respectively) the request should be redirected to.

### Advantages of range partitioning

Some advantages of range partitioning include:

* Simplicity and ease of implementation.
* The ability to perform range queries using the partitioning key value.
* A good performance for range queries that use the partitioning key, when the queried range is small and resides in a single node.
* Makes adjusting ranges (re-partitioning) easier and more efficient. One range can be increased or decreased, which exchanges data only between two nodes.

### Disadvantages of range partitioning

Some disadvantages of range partitioning include:

* The inability to perform range queries using keys other than the partitioning key
* A bad performance for range queries that use the partitioning key when the queried range is big and resides in multiple nodes
* An uneven distribution of the traffic or data, which causes some nodes to overload. For example, while range partitioning through an alphabetical split, we may find that some alphabetical letters appear as the initial letters in surnames more frequently than other letters. This means some nodes may have to store more data and process more requests than others.

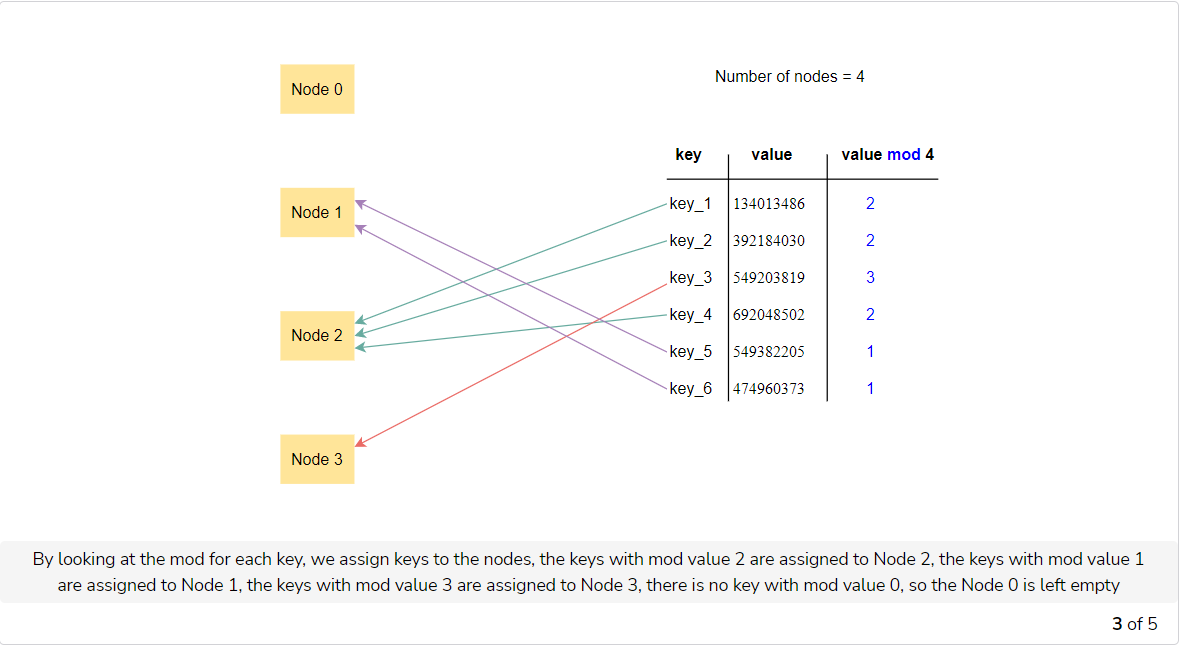
Some systems that leverage a range partitioning technique are Google’s BigTable, and [Apache HBase](https://hbase.apache.org/).

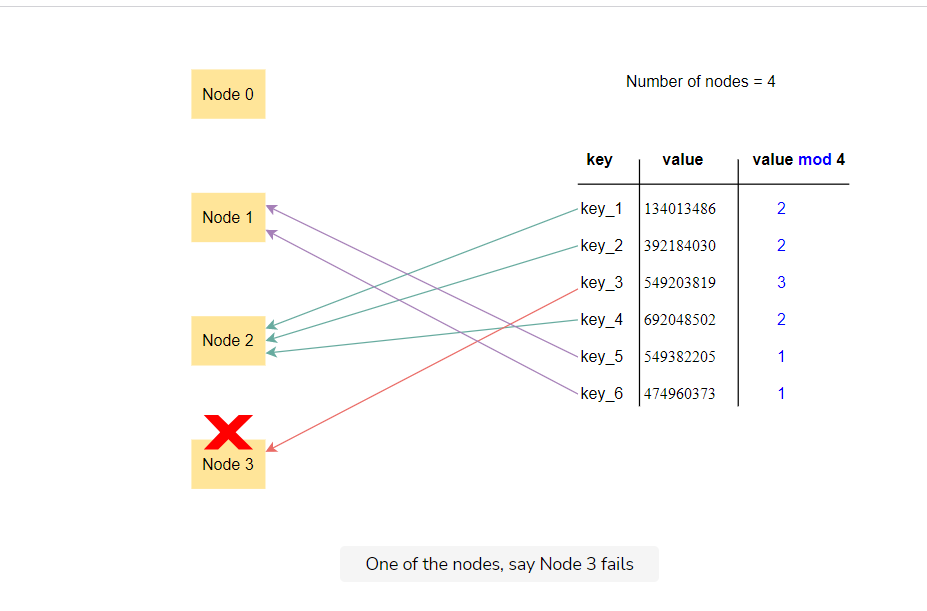
## Hash partitioning

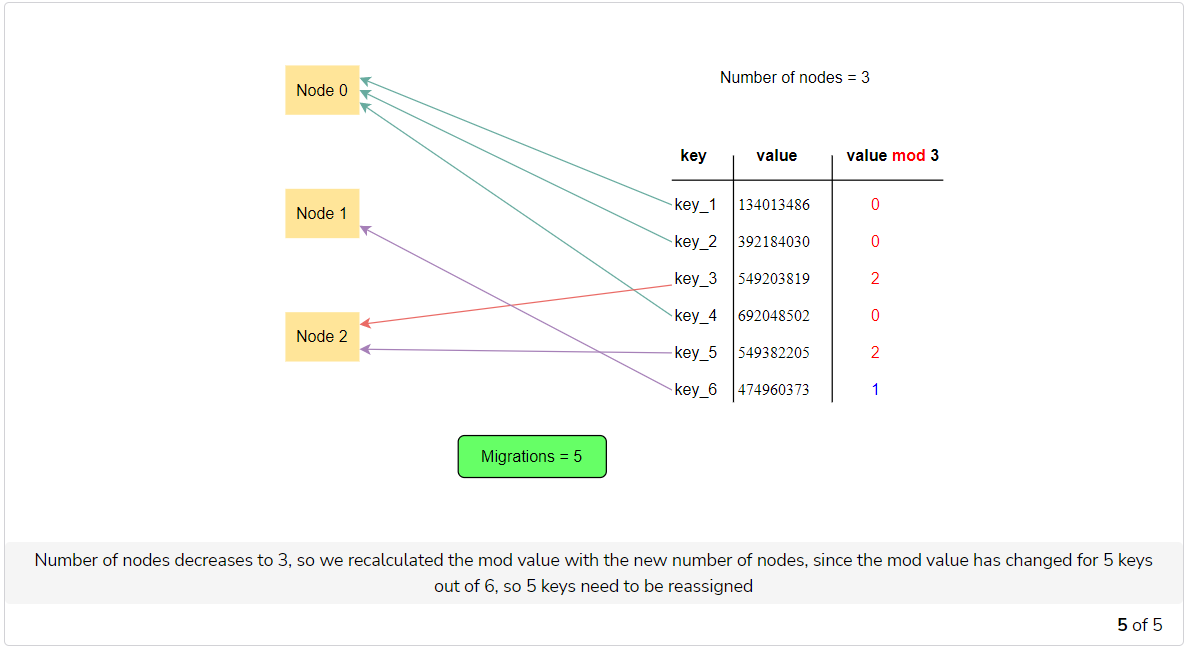
**Hash partitioning** is a technique where we apply a hash function to a specific attribute of each row. This results in a number that determines which partition—and, thus, node—this row belongs to.

For the sake of simplicity, let’s assume we have one partition per node, as in the previous example, and a hash function that returns an integer. If we have n number of nodes in our system and try to identify which node locates a student record with a surname s, we’ll calculate it with the formula hash(s) mod n.

This mapping process will take place both when we write a new record, and when we receive a request to find a record for a specific value of this attribute.







### Advantages of hash partitioning

Some advantages of hash partitioning include:

* The ability to calculate the partitioning mapping at runtime with no need to store and maintain the mapping. This is beneficial both in terms of data storage needs and performance, as we don’t need any additional requests to find the mapping
* A greater chance that the hash function will uniformly distribute the data across our system’s nodes, and prevent some nodes from overloading

### Disadvantages of hash partitioning

Some disadvantages of hash partitioning include:

* The inability to perform range queries at all—even for the attribute we use as a partitioning key—without storing additional data or querying all the nodes
* Adding or removing nodes from the system causes it to re-partition. This results in significant data movement across all nodes of the system

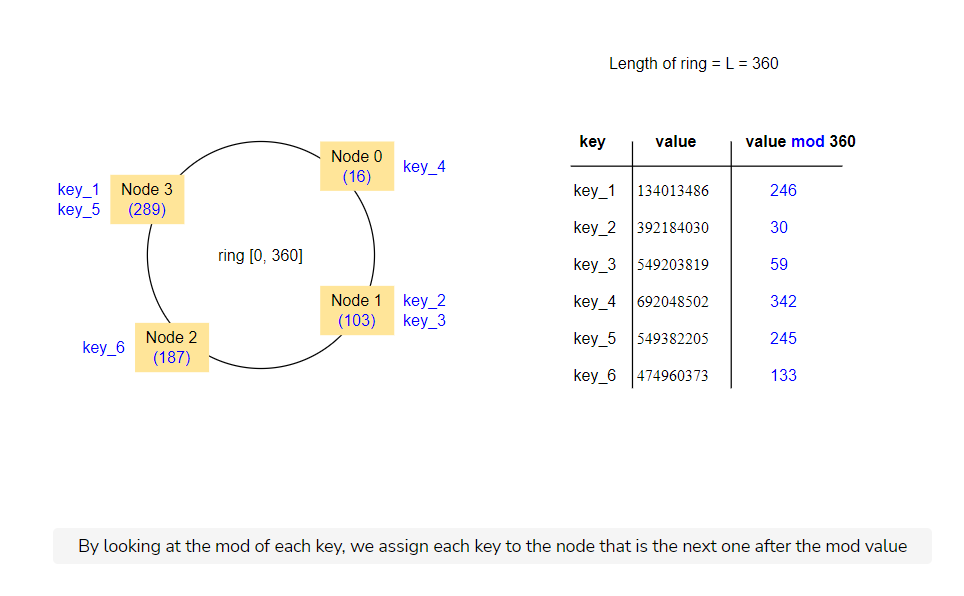
## Consistent hashing

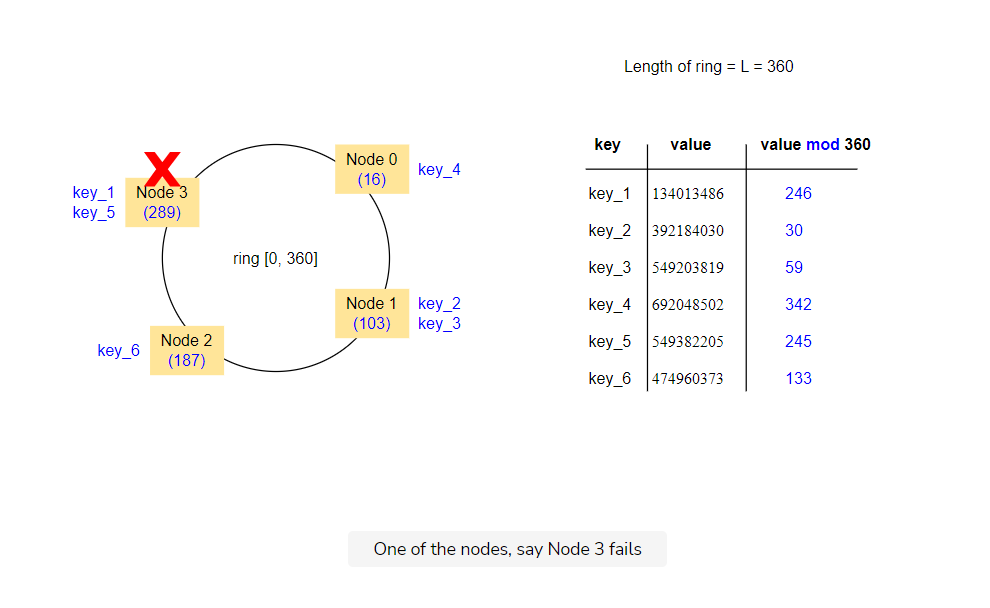
**Consistent hashing** is a partitioning technique that is very similar to hash partitioning, but solves the increased data movement problem caused by hash partitioning.

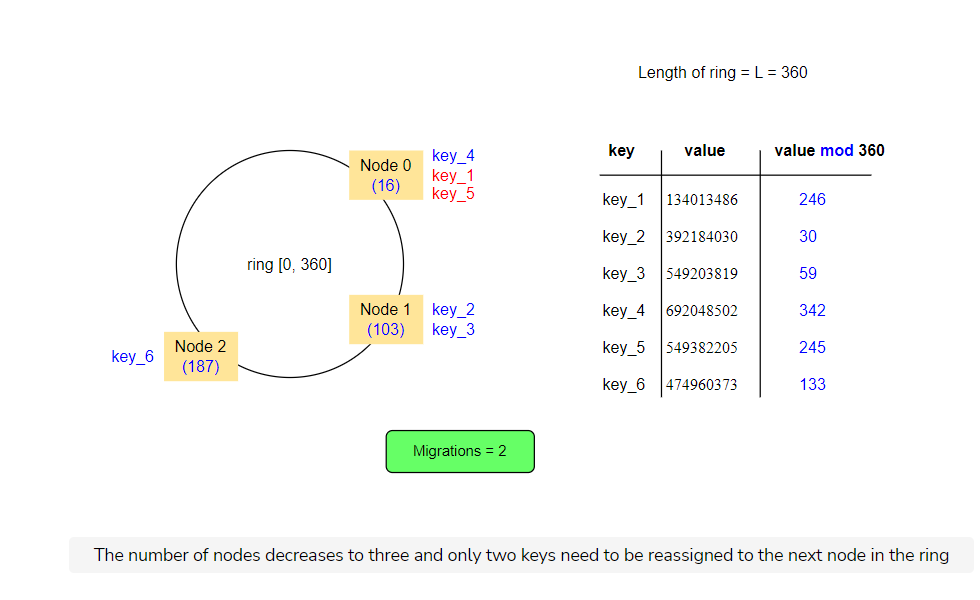
This is how it works: each node in the system is randomly assigned an integer in a range of [0, L]. This range is called ring (for example, [0, 360]). Then, the system uses a record with an attribute value s as a partitioning key to locating the node after the point hash(s) mod L in the ring.

As a result, when a new node enters the ring, it receives data only from the next node in the ring. The other nodes don’t need to exchange any more data. Similarly, when a node leaves the ring, its data transfer to the next node in the ring.

The following illustration depicts this behavior and the difference between these two different algorithms.







### Advantages of consistent hashing

Consistent hashing has one main advantage, when compared to hash partitioning:

* Reduced data movement when nodes are added or removed in the system

### Disadvantages of consistent hashing

Some disadvantages of consistent hashing include:

* The potential for the data’s non-uniform distribution because of the random assignment of nodes in the ring
* The potential for more imbalanced data distribution as nodes are added or removed. E.g., a node’s dataset is not distributed evenly across the system when it is removed but is instead transferred to a single node

We can mitigate these issues through the concept of “virtual nodes,” where we assign each physical node multiple locations in the ring. These locations are known as **virtual nodes**.

For further discussion on this concept, feel free to read the Dynamo paper. (G. DeCandia et al., “Dynamo: Amazon’s Highly Available Key-value Store,” in Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, 2007.)Another widely-used system that uses consistent hashing is Apache Cassandra.( A. Lakshman and P. Malik, “Cassandra — A Decentralized Structured Storage System,” Operating Systems Review, 2010.)

# Replication

Learn what replication is and why it is used in distributed systems.

**We'll cover the following**

* [Availability](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4770154336747520#Availability)
* [Mechanism to achieve availability](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4770154336747520#Mechanism-to-achieve-availability)
  + [Replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4770154336747520#Replication)
    - [Pessimistic replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4770154336747520#Pessimistic-replication)
    - [Optimistic replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4770154336747520#Optimistic-replication)

Partitioning can improve the scalability and performance of a system by distributing data and request load to multiple nodes.

Another dimension that benefits from using a distributed system is known as **availability**.

## Availability

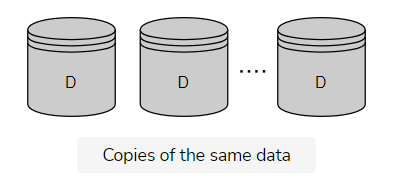
Availability refers to the ability of the system to remain functional despite failures in parts of it.

## Mechanism to achieve availability

The technique we use to achieve availability is **replication**.

### Replication

Replication is the main technique used in distributed systems to increase availability. It consists of storing the same piece of data in multiple nodes (called replicas) so that if one of them crashes, data is not lost, and requests can be served from the other nodes in the meanwhile.



Copies of the same data

However, the benefit of increased availability from replication comes with a set of new complications.

Replication implies that the system now has multiple copies of every piece of data. These copies must be maintained and kept in sync with each other on every update.

Ideally, replication should function transparently to the end-user, or engineer. This is to create the illusion that there’s only one copy of every piece of data. This makes a distributed system look like a simple, centralized system of a single node that is much easier to reason about and develop software around.

Of course, this is not always possible. We may require significant hardware resources or need to give up other desirable properties to achieve this ideal. For instance, engineers sometimes willingly accept a system that provides much higher performance, but occasionally gives a non-consistent view of the data. Hence, they only do this under specific conditions—and in a specific way—they can account for when they design the application.

Therefore, there are two main strategies for replication:

1. Pessimistic replication
2. Optimistic replication

#### Pessimistic replication

**Pessimistic replication** tries to guarantee from the beginning that all the replicas are identical to each other—as if there was only one copy of the data all along.

#### Optimistic replication

**Optimistic replication**, or lazy replication, allows the different replicas to diverge. This guarantees that they will converge again if the system does not receive any updates, or enters a quiesced state, for a period of time.

Replication is a very active field in research, so there are many different algorithms for it.

# Single-Master Replication Algorithm

Learn about single-master replication, and its practical application, advantages, and disadvantages.

**We'll cover the following**

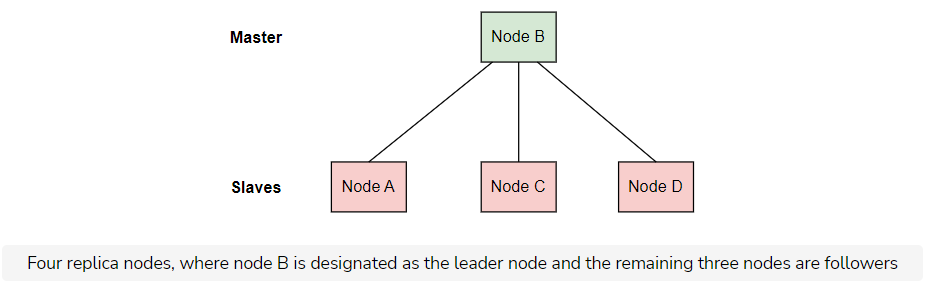
* [Single-master replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Single-master-replication)
  + [Techniques for propagating updates](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Techniques-for-propagating-updates)
    - [Synchronous replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Synchronous-replication)
    - [Asynchronous replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Asynchronous-replication)
* [Advantages of single-master replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Advantages-of-single-master-replication)
* [Disadvantages of single-master replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Disadvantages-of-single-master-replication)
* [Failover](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Failover)
  + [Managing failover](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Managing-failover)
    - [Manual approach](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Manual-approach)
    - [Automated approach](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Automated-approach)

## Single-master replication

Single-master replication is a technique where we designate a single node amongst the replicas as the **leader**, or primary, that receives all the updates.

This technique is also known as **primary-backup replication**.

We commonly refer to the remaining replicas as **followers** or secondaries. These can only handle read requests. Every time the leader receives an update, it executes it locally and also propagates the update to the other nodes. This ensures that all the replicas maintain a consistent view of the data.



### Techniques for propagating updates

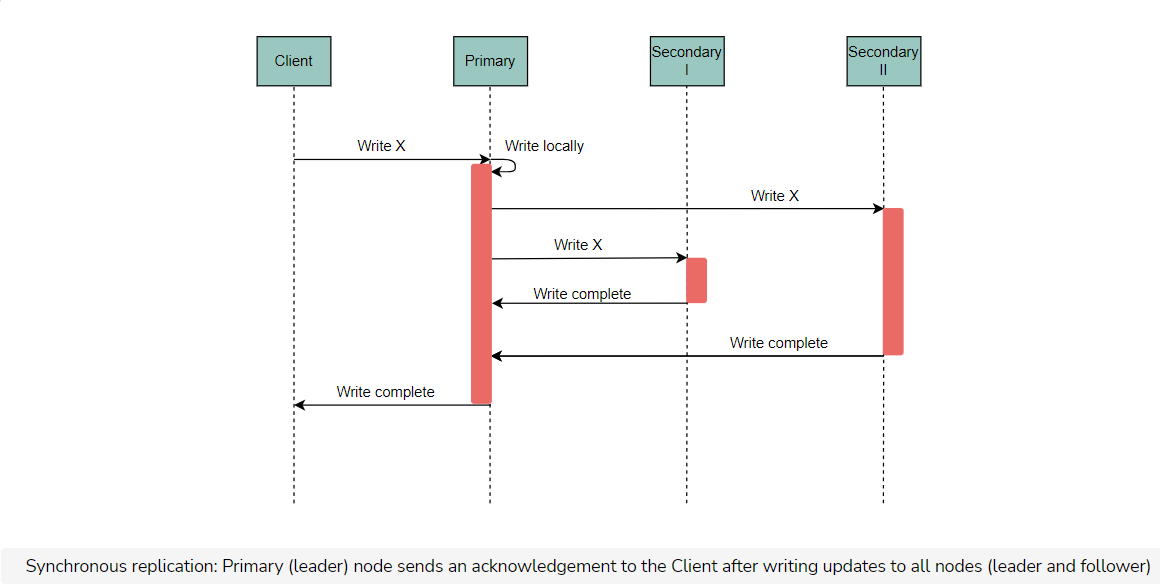
There are two ways to propagate the updates: synchronously and asynchronously.

#### Synchronous replication

In **synchronous replication**, the node replies to the client to indicate the update is complete—only after receiving acknowledgments from the other replicas that they’ve also performed the update on their local storage. This guarantees that the client is able to view the update in a subsequent read after acknowledging it, no matter which replica the client reads from.

Furthermore, synchronous replication provides increased **durability**. This is because the update is not lost even if the leader crashes right after it acknowledges the update.

However, this technique can make writing requests slower. This is because the leader has to wait until it receives responses from all the replicas.

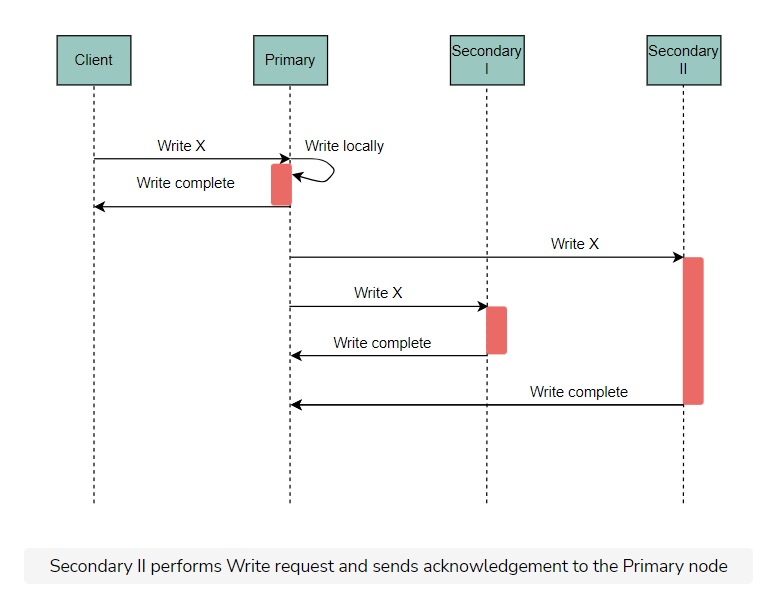


#### Asynchronous replication

In **asynchronous replication**, the node replies to the client as soon as it performs the update in its local storage, without waiting for responses from the other replicas.

This technique increases performance significantly for write requests. This is because the client no longer pays the penalty of the network requests to the other replicas.

However, this comes at the cost of reduced consistency and decreased **durability**. After a client receives a response for an update request, the client might read older (stale) values in a subsequent read. This is only possible if the operation happens in one of the replicas that have not yet performed the update. Moreover, if the leader node crashes right after it acknowledges an update, and the propagation requests to the other replicas are lost, any acknowledged update is eventually lost.



Most widely used databases, such as [PostgreSQL](https://www.postgresql.org/) or [MySQL](https://www.mysql.com/), use a single-master replication technique that supports both asynchronous and synchronous replication.

## Advantages of single-master replication

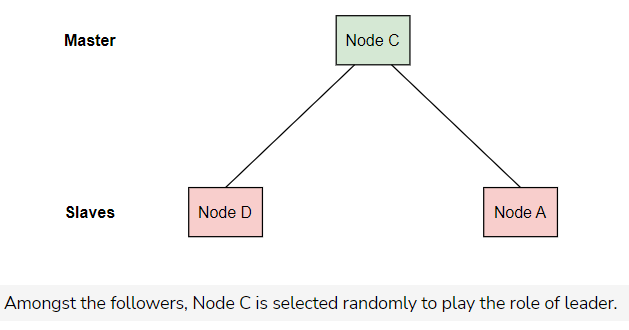
* It is simple to understand and implement
* Concurrent operations serialized in the leader node, remove the need for more complicated, distributed concurrency protocols. In general, this property also makes it easier to support transactional operations
* It is scalable for read-heavy workloads, because the capacity for reading requests can be increased by adding more read replicas

## Disadvantages of single-master replication

* It is not very scalable for write-heavy workloads, because a single node (the leader)’s capacity determines the capacity for writes
* It imposes an obvious trade-off between performance, durability, and consistency
* Scaling the read capacity by adding more follower nodes can create a bottleneck in the network bandwidth of the leader node, if there’s a large number of followers listening for updates
* The process of failing over to a follower node when the leader node crashes, is not instant. This may create some downtime and also introduce the risk of errors

## Failover

**Failover** is when the master node crashes and a follower node takes over.



When the master node crashes, we need to choose another master node. Following are the approaches to perform failover.

### Managing failover

In general, there are two approaches to perform a failover: **manual** and **automated**.

#### Manual approach

In the manual approach, the operator selects the new leader node and instructs all the nodes accordingly. This is the safest approach, but it incurs significant downtime.

#### Automated approach

An alternative is an automated approach, where follower nodes detect that the leader node has crashed (e.g., via periodic heartbeats), and attempt to elect a new leader node. This is faster but is quite risky. This is because there are many different ways in which the nodes can get confused and arrive at an incorrect state.

The chapter about consensus will cover this topic, called [leader election](https://www.educative.io/collection/page/10370001/4891237377638400/4827224044208128#leader-election), in more detail.

# Multi-Master Replication Algorithm

Look at the multi-master algorithm for replication.

**We'll cover the following**

* [Multi-master replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4976779743985664#Multi-master-replication)
  + [Conflict resolution](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4976779743985664#Conflict-resolution)
  + [Approaches to conflict resolution](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4976779743985664#Approaches-to-conflict-resolution)
    - [Exposing conflict resolution to the clients](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4976779743985664#Exposing-conflict-resolution-to-the-clients)
    - [Last-write-wins conflict resolution](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4976779743985664#Last-write-wins-conflict-resolution)
    - [Causality tracking algorithms](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4976779743985664#Causality-tracking-algorithms)

As we saw in the previous lesson, single-master replication is a technique that is easy to implement and operate. It can easily support transactions and hide the distributed nature of the underlying system, i.e., when using synchronous replication.

However, single-master replication has some limitations in terms of performance, scalability, and availability.

As we’ve already discussed, there are many applications where availability and performance are much more important than data consistency or transactional semantics.

A frequently cited example is that of an e-commerce shopping cart, where the most important thing is for customers to be able to access their cart at all times and add items quickly and easily. It is acceptable to compromise consistency to achieve this, as long as there is data reconciliation at some point. For instance, if two replicas diverge because of intermittent failures, the customer can still resolve conflicts during the checkout process.

## Multi-master replication

**Multi-master replication** is an alternative replication technique that favors higher availability and performance over data consistency.

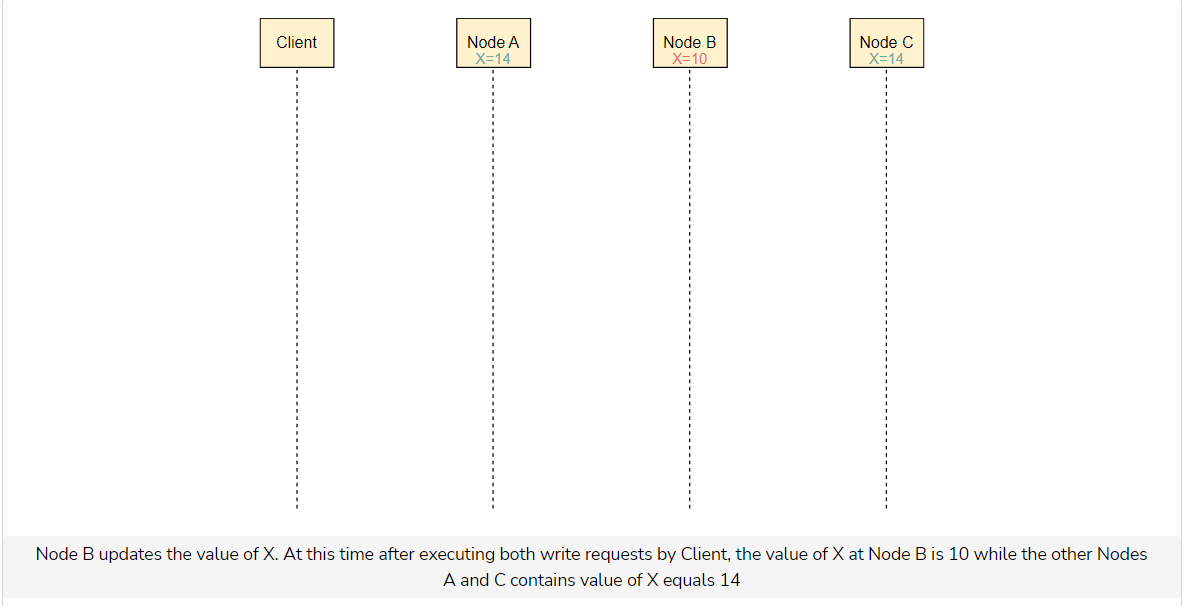
This technique is also known as **multi-primary replication**.

In this technique, all replicas are equal and can accept write requests. They are also responsible for propagating the data modifications to the rest of the group.

Multi-master replication has a significant difference from single-master replication. In multi-master replication, there is no single leader node that serializes the requests and imposes a single order, as write requests are concurrently handled by all the nodes. This means that nodes might disagree on what is the right order for some requests. We usually refer to this as a **conflict**.

For the system to remain operational, the nodes need to resolve this conflict when it occurs by agreeing on a single order from the available ones.

The following illustration depicts an instance where two write requests can potentially result in a conflict, depending on the latency of the propagation requests between the nodes of the system.



In the case of a conflict, a subsequent read request could receive different results depending on the node that handles the request—unless we resolve the conflict so that all the nodes converge again to a single value.

### Conflict resolution

There are many different ways to resolve conflicts, depending on the guarantees the system wants to provide.

An important aspect of different approaches to resolving conflicts is whether they do it eagerly or lazily.

* In the **eagerly** case, the conflict is resolved during the write operation.
* In the **lazily** case, the write operation proceeds to maintain multiple, alternative versions of the data record that are eventually resolved to a single version later on, i.e., during a subsequent read operation.

### Approaches to conflict resolution

Here are some common approaches to conflict resolution:

#### Exposing conflict resolution to the clients

When there is a conflict, the multiple available versions return to the client. The client then selects the right version and returns it to the system. This resolves the conflict.

An example of this is the shopping cart application, where the customer selects the correct version of their cart.

#### Last-write-wins conflict resolution

Each node in the system tags each version with a timestamp, using a local clock. During a conflict, the version with the latest timestamp is selected.

However, this technique can lead to some unexpected behaviors, as there is no global notion of time. For example, write A can override write B, even though B happened “as a result” of A.

#### Causality tracking algorithms

The system uses an algorithm that keeps track of causal relationships between different requests. When there is a conflict between two writes (A, B) and one is determined to be the cause of the other one (suppose A is the cause of B), then the resulting write (B) is retained.

However, there can still be writes that are not causally related, i.e., requests are actually concurrent. In such cases, the system cannot make an easy decision.

We’ll elaborate more on some of these approaches later in the chapters about time and order.

# Quorums in Distributed Systems

Look at the concept of quorums and see how they solve low availability problems in synchronous replication.

**We'll cover the following**

* [The problem in synchronous replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6055804243542016#The-problem-in-synchronous-replication)
  + [Possible solution](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6055804243542016#Possible-solution)
* [Quorums](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6055804243542016#Quorums)

The main pattern we’ve seen so far is this: writes are performed to all the replica nodes, while reads are performed to one of them. When we ensure that writes are performed to all of them synchronously before replying to the client, we guarantee that the subsequent reads see all the previous writes—regardless of the node that processes the read operation.

## The problem in synchronous replication

Availability is quite low for write operations, because the failure of a single node makes the system unable to process writes until the node recovers.

### Possible solution

To solve this problem, we can use the reverse strategy. That is, we write data only to the node that is responsible for processing a write operation, but process read operations by reading from all the nodes and returning the latest value.

This increases the availability of writes significantly but decreases the availability of reads at the same time. So, we have a trade-off that needs a mechanism to achieve a balance. Let’s see that mechanism.

## Quorums

A useful mechanism to achieve a balance in this trade-off is to use **quorums**.

Let’s consider an example. In a system of three replicas, we can say that writes need to complete in two nodes (as a quorum of two), while reads need to retrieve data from two nodes. This way, we can be sure that the reads will read the latest value. This is because at least one of the nodes in the read quorum will also be included in the latest write quorum.

This is based on the fact that in a set of three elements, two subsets of two elements must have at least one common element.

A past paper introduced this technique as a **quorum-based voting protocol** for replica control.

In general, in a system that has a total of *V* replicas, every read operation should obtain a read quorum of *Vr*​ replicas. Meanwhile, a write operation should obtain a write quorum of *Vw*​ replicas. The values of these quorums should obey the following properties:

* *Vr*​+*Vw*​>*V*
* *Vw*​>*V*/2

The first rule ensures that a data item is not read and written by two operations concurrently.

The second rule ensures that at least one node receives both of the two write operations and imposes an order on them. This means that two write operations from two different operations cannot occur concurrently on the same data item.

Both of the rules together guarantee that the associated distributed database behaves as a centralized, one-replica database system.

The concept of a quorum is really useful in distributed systems that have multiple nodes.

The concept of a quorum is used extensively in other areas, like distributed transactions or consensus protocols.

# Safety Guarantees in Distributed Systems

In this lesson, we will explore the properties that guarantee safety in distributed systems, and their relation with difficulties in designing distributed systems.

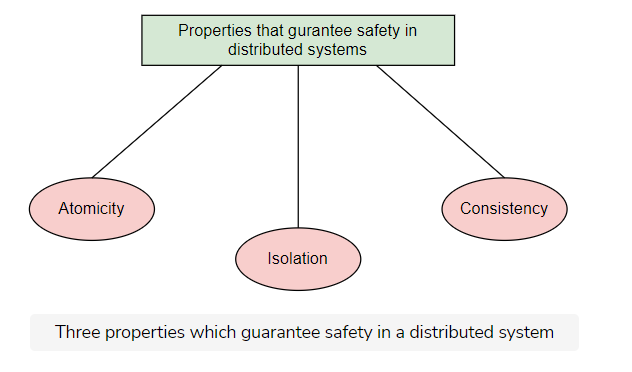
**We'll cover the following**

* [Safety guarantors](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6113163766661120#Safety-guarantors)
  + [Achieving atomicity](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6113163766661120#Achieving-atomicity)
  + [Achieving consistency](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6113163766661120#Achieving-consistency)
  + [Achieving isolation](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6113163766661120#Achieving-isolation)

Since distributed systems involve a lot of complexity, some safety guarantees ensure that the system will behave in specific, predictable ways. This makes it easier for people to reason about a system and any potential anomalies that can occur. This will allow them to build proper safeguards to prevent these anomalies from occurring.

## Safety guarantors

The main safety guarantees that systems provide are around the three properties shown in the illustration.



The concepts of **atomicity** and **isolation** originate from database research and ACID transactions. When we mention **consistency** in this course, we will mostly refer to the notion of consistency made popular by the CAP theorem.

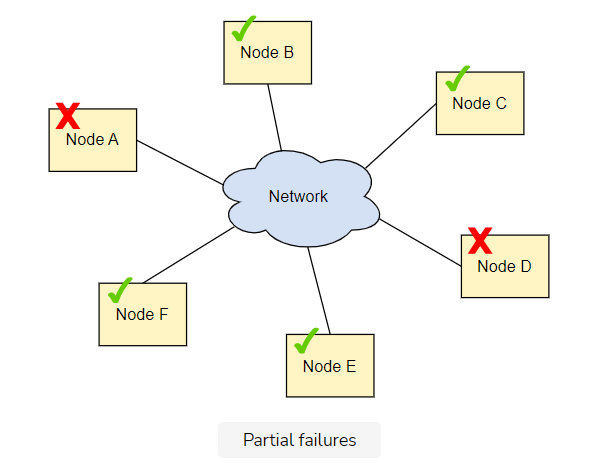
Before going any further, it is useful to look at these topics. We will study these two topics in detail in the next two lessons.

It is interesting to observe that each of these safety guarantees is tightly related to the [aforementioned reasons](https://www.educative.io/collection/page/10370001/4891237377638400/6155524821745664) that make distributed systems hard to design.

### Achieving atomicity

It is challenging to achieve atomicity in a distributed system because of the possibility of **partial failures**.

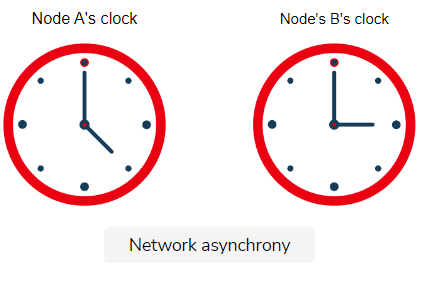
A partial failure occurs when some components in the system fail. The following illustration shows this.



### Achieving consistency

It is challenging to achieve consistency because of the **network asynchrony**.

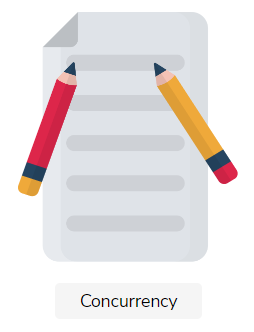
Network asynchrony occurs when different nodes in a network have different values for the current time. The following illustration shows this.



### Achieving isolation

It is challenging to achieve isolation because of the inherent concurrency of distributed systems.

Concurrency occurs when multiple things happen at the same time. The following illustration shows this.



In the above illustration, two pens are trying to write on a single resource at the same time.

# ACID Transactions

Let's see the ACID properties.

**We'll cover the following**

* [Atomicity (A)](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5590893897973760#Atomicity-(A))
* [Consistency (C)](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5590893897973760#Consistency-(C))
* [Isolation (I)](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5590893897973760#Isolation-(I))
* [Durability (D)](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5590893897973760#Durability-(D))

**ACID** is a set of properties of traditional database transactions that provide guarantees around the expected behavior of transactions during errors, power failures, etc. More specifically, these properties are the following.

## Atomicity (A)

**Atomicity** guarantees that a transaction that comprises multiple operations is treated as a single unit. This means that either all operations of the transaction are executed or none of them are.

This concept of atomicity extends to distributed systems, where the system might need to execute the same operation in multiple nodes of the system in an atomic way. So, the operation is either executed to all the nodes or none.

## Consistency (C)

**Consistency** guarantees that a transaction only transitions the database from one valid state to another valid state, while maintaining any database invariants. However, these invariants are application-specific and defined by every application accordingly.

For example, consider an application that has a table A with records that refer to records in table B through a [foreign key relationship](https://en.wikipedia.org/wiki/Foreign_key). The database prevents a transaction from deleting a record from table A, unless any records in table B referenced from this record are already deleted.

Note that this is not the concept of consistency we refer to in the context of distributed systems.

## Isolation (I)

**Isolation** guarantees that even though transactions might run concurrently and have data dependencies, the result is as if one of them was executed at a time and there was no interference between them. This prevents a large number of anomalies.

## Durability (D)

**Durability** guarantees that once a transaction is committed, it remains committed even in the case of failure.

In the context of single-node, centralized systems, this usually means that completed transactions and their effects are recorded in non-volatile storage.

In the context of distributed systems, this means that transactions need to be durably stored in multiple nodes. This way, recovery is possible even in the presence of total failures of a node, alongside its storage facilities.

# The CAP Theorem

In this lesson, we explain the CAP theorem with its proof and its extended PACELC theorem.

**We'll cover the following**

* [Initial statement of the CAP theorem](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#Initial-statement-of-the-CAP-theorem)
  + [Consistency](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#Consistency)
  + [Availability](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#Availability)
  + [Partition Tolerance](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#Partition-Tolerance)
* [Final statement of the CAP theorem](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#Final-statement-of-the-CAP-theorem)
  + [Proof](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#Proof)
* [Importance of the CAP theorem](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#Importance-of-the-CAP-theorem)
* [Categorization of distributed systems based on the CAP theorem](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#Categorization-of-distributed-systems-based-on-the-CAP-theorem)
* [Trade-off between latency and consistency](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#Trade-off-between-latency-and-consistency)
* [PACELC theorem](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#PACELC-theorem)
* [Categorization of distributed systems based on PACELC theorem](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5686579427540992#Categorization-of-distributed-systems-based-on-PACELC-theorem)

The **CAP Theorem** is one of the most fundamental theorems in the field of distributed systems. It outlines an inherent trade-off in the design of distributed systems.

## Initial statement of the CAP theorem

According to the initial statement of the CAP theorem, it is impossible for a distributed data store to provide more than two of the following properties simultaneously: **consistency**, **availability**, and **partition tolerance**.

### Consistency

Consistency means that every successful read request receives the result of the most recent write request.

The concept of consistency in the CAP theorem is completely different from the concept of consistency in ACID transactions. The notion of consistency as presented in the CAP theorem is more important for distributed systems.

### Availability

Availability means that every request receives a non-error response, without any guarantees on whether it reflects the most recent write request.

### Partition Tolerance

Partition tolerance means that the system can continue to operate despite an arbitrary number of messages being dropped by the network between nodes due to a **network partition**.

It is very important to understand that partition tolerance is not a property we can abandon.

In a distributed system, there is always the risk of a network partition. If this happens, the system needs to decide either to continue operating and compromise data consistency, or stop operating and compromise availability.

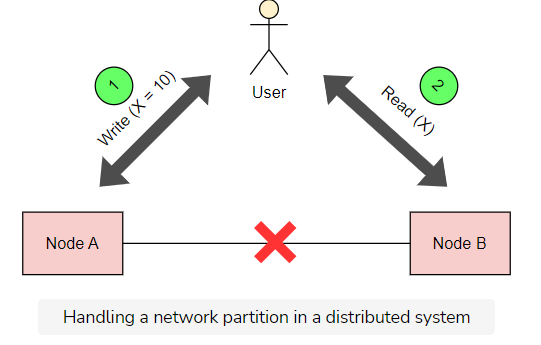
However, there is no such thing as trading off partition tolerance to maintain both consistency and availability. As a result, what this theorem really states is the following.

## Final statement of the CAP theorem

According to the final statement of the CAP theorem, a distributed system can be either consistent or available in the presence of a network partition.

### Proof

Let’s attempt to prove this theorem simplistically and schematically. Let’s imagine a distributed system consisting of two nodes, as shown in the illustration.



This distributed system can act as a plain register with the value of a variable *X*.

Now, let’s assume that there is a network failure that results in a network partition between the two nodes of the system at some point. A user of the system performs a write, and then a read—even two different users may perform these operations.

We will examine the case where a different node of the system processes each operation. In that case, the system has two options:

* It can fail one of the operations, and break the *availability* property.
* It can process both the operations, which will return a stale value from the read and break the *consistency* property.

It cannot process both of the operations successfully, while also ensuring that the read returns the latest value that is written by the write operation. This is because the results of the write operation cannot be propagated from node A to node B due to the network partition.

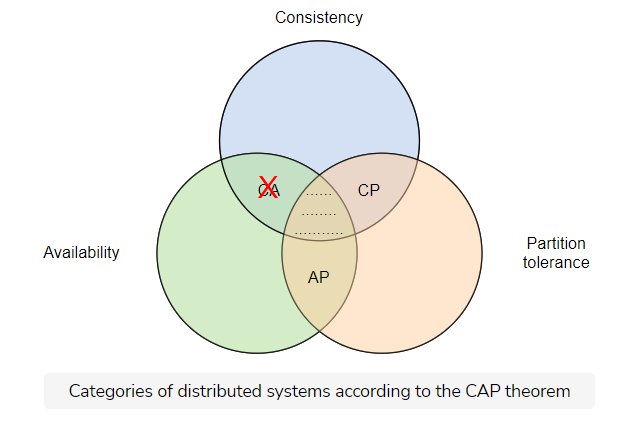
**Importance of the CAP theorem**

The CAP theorem is really important because it helped establish the basic limitations of all distributed systems.

The CAP theorem forces designers of distributed systems to make explicit trade-offs between *availability* and *consistency*. Once the engineers become aware of these properties, they choose the appropriate system.

**Categorization of distributed systems based on the CAP theorem**

When we read the literature and documentation of distributed systems, we notice that systems are usually classified into two basic categories: **CP** and **AP**. This classification depends on which property the system violates during a network partition.



There is another important thing to note about the CAP theorem: the choice between consistency and availability needs to be made only during a network partition.

Both consistency and availability properties can be satisfied when the network partition is not present.

## Trade-off between latency and consistency

When no network partition is present during normal operation, there’s a different trade-off between latency and consistency.

To guarantee data consistency, the system will have to delay write operations until the data has been propagated across the system successfully, thus taking a latency hit.

An example of this trade-off is the single-master replication scheme. In this setting, the synchronous replication approach would favor consistency over latency. Meanwhile, asynchronous replication would reduce latency at the cost of consistency.

## PACELC theorem

The **PACELC theorem** is an extension of the CAP theorem that is captured in a separate article. It states the following:

“In the case of a network partition (P), the system has to choose between availability (A) and consistency (C) but else (E), when the system operates normally in the absence of network partitions, the system has to choose between latency (L) and consistency (C).”

## Categorization of distributed systems based on PACELC theorem

Each branch of the PACELC theorem creates two sub-categories of systems.

The first part of the theorem defines the two categories we have already seen: CP and AP.

The second part defines two new categories: **EL** and **EC**.

These sub-categories are combined to form the following four categories:

* AP/EL
* CP/EL
* AP/EC
* CP/EC

A system from the AP/EL category prioritizes availability during a network partition and latency during a normal operation.

In most cases, systems are designed with an overarching principle in mind: usually either performance and availability, or data consistency. As a result, most of the systems fall into the AP/EL or CP/EC categories.

There are still systems we cannot strictly classify into these categories. This is because they have various levers that can tune the system differently when needed. Still, this theorem serves as a good indicator of the various forces at play in a distributed system.

We can find a table with the categorization of several distributed systems along these dimensions in the associated [Wikipedia page](https://en.wikipedia.org/wiki/PACELC_theorem).

# Consistency Models

In this lesson, we will learn the different forms of consistency.

**We'll cover the following**

* [Consistency model](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#Consistency-model)
  + [A strong consistency model](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#A-strong-consistency-model)
  + [List of consistency models](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#List-of-consistency-models)
    - [Linearizability](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#Linearizability)
      * [Benefits](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#Benefits)
    - [Sequential consistency](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#Sequential-consistency)
      * [Example](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#Example)
    - [Causal consistency](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#Causal-consistency)
      * [Example](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#Example)
    - [Eventual consistency](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#Eventual-consistency)
      * [Example](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5368578505441280#Example)

According to the [CAP Theorem](https://www.educative.io/collection/page/10370001/4891237377638400/6165708642189312), consistency means that every successful read request will return the result of the most recent write. In fact, this is an oversimplification, because there are many different forms of consistency.

In this lesson, we introduce the forms of consistency that are most relevant to us.

To accurately define all these forms really, we need to build a formal model. This is usually the **consistency model**.

## Consistency model

The consistency model defines the set of execution histories that are valid in a system.

In layperson’s terms, a model formally defines the behaviours that are possible in a distributed system.

Consistency models are extremely useful for many reasons:

* They help us formalise the behaviours of systems. Systems can then provide guarantees about their behaviour.
* Software engineers can confidently use a distributed system (i.e. a distributed database) in a way that does not violate any safety properties they care about.

In essence, software engineers can treat a distributed system as a black box that provides a set of properties. Moreover, they can do this without knowing of all the complexity the system internally assumes to provide these properties.

### A strong consistency model

We consider consistency model A stronger than model B when the first allows fewer histories. Alternatively, we say model A makes more assumptions about or poses more restrictions on, the system’s possible behaviors.

Usually, the stronger the consistency model a system satisfies, the easier it is to build an application on top of it. This is because the developer can rely on stricter guarantees.

### List of consistency models

There are many different consistency models used across the modern system design field. We will focus on the most fundamental ones. These are the following:

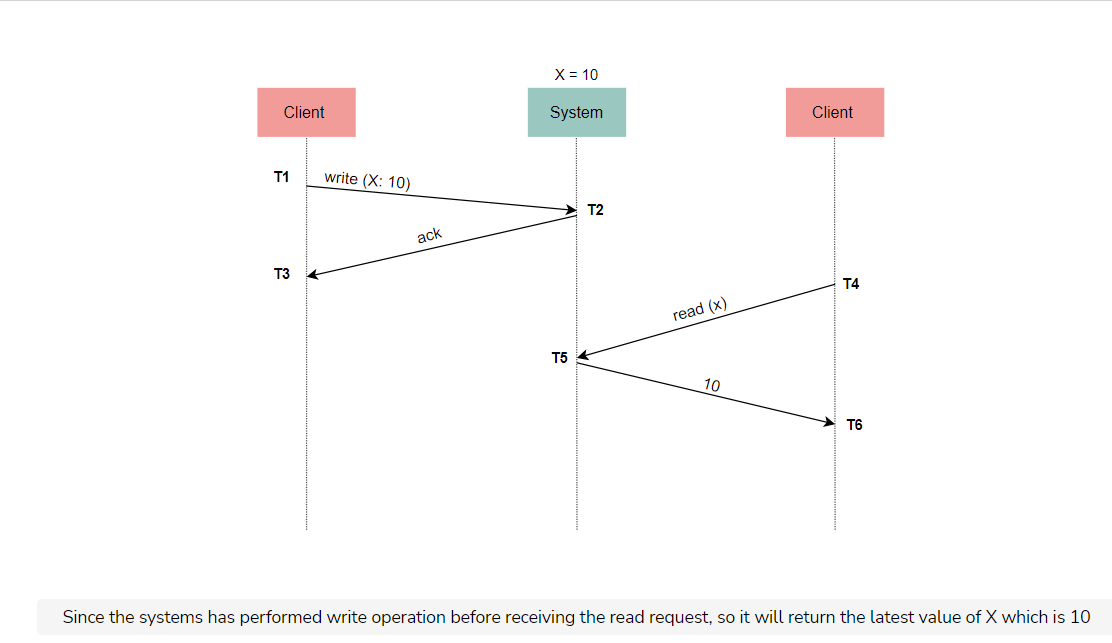
* Linearizability
* Sequential Consistency
* Causal Consistency
* Eventual Consistency

#### Linearizability

A system that supports the consistency model of **linearizability** is one where operations appear to be instantaneous to the external client. This means that they happen at a specific point—from the point the client invokes the operation, to the point the client receives the acknowledgment by the system the operation has been completed.

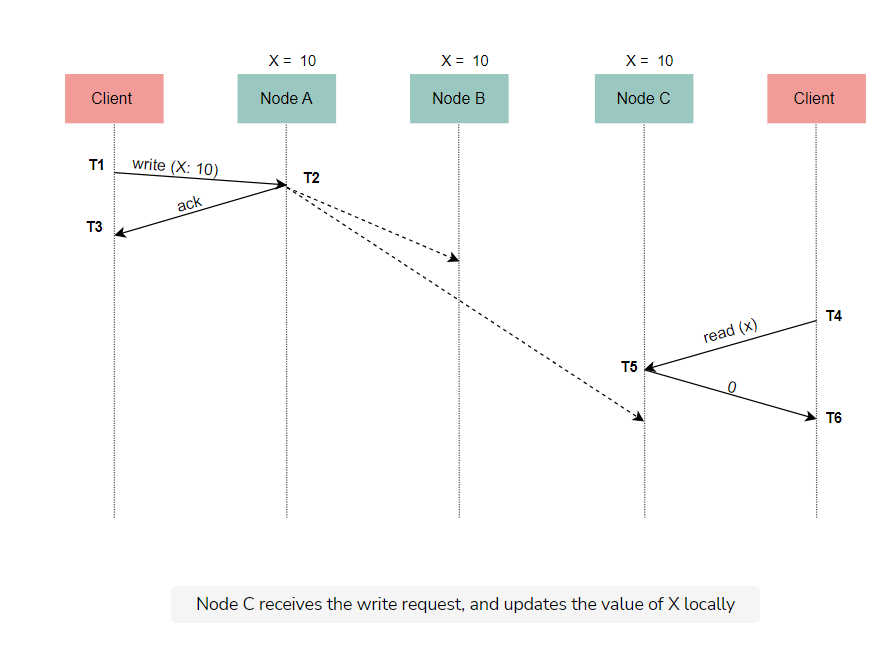
Furthermore, once an operation is complete and the acknowledgment is delivered to the client, it is visible to all other clients. This implies that if a client C2 invokes a read operation after a client C1 receives the completion of its write operation, C2 should see the result of this (or a subsequent) write operation. It may be obvious to some that operations are instantaneous and visible after they’re completed.

In a centralized system, linearizability is obvious. The following illustration shows this.



However, there is no such thing as instantaneity in a distributed system.

The following illustration shows why linearizability is not obvious in a distributed system.



When we think of a distributed system as a single node, it seems obvious that every operation happens at a specific instant of time, and is immediately visible to everyone. However, when we think of a distributed system as a set of cooperating nodes, we realize that we should not take this for granted.

For instance, the system in the above illustration is not linearizable since T4 > T3. However, still, the second client won’t observe the read because it has not yet propagated to the node that processes the read operation. The **non-linearizability** comes from the use of asynchronous replication.

When we use a synchronous replication technique, we make the system linearizable. However, that means that the first write operation takes longer until the new value has propagated to the rest of the nodes. Remember the latency-consistency trade-off from the PACELC theorem.

##### Benefits

As a result of the above discussion, we realize that linearizability is a very powerful consistency model. It helps us treat complex distributed systems as much simpler, single-node datastores, and reason about our applications more efficiently. Moreover, by leveraging atomic instructions provided by hardware (such as [CAS operations](https://en.wikipedia.org/wiki/Compare-and-swap)), we can build more sophisticated logic on top of distributed systems, such as mutexes, semaphores, counters, etc. This is not possible under weaker consistency models.

#### Sequential consistency

**Sequential consistency** is a weaker consistency model, where operations are allowed to take effect before their invocation or after their completion.

As a result, it provides no real-time guarantees. However, operations from different clients have to be seen in the same order by all other clients, and operations of every single client preserve the order specified by its program (in this global order). This allows many more histories than linearizability, but still poses some constraints that can help real-life applications.

##### Example

For example, in a social networking application, we usually do not care what’s the ordering of posts between some of our friends. However, we still expect posts from a single friend to be displayed in the right order (i.e., the one they published them at). Following the same logic, we usually expect our friends’ comments in a post to appear in the order that they submitted them. These are all properties that the sequential consistency model captures.

#### Causal consistency

In some cases, we don’t need to preserve the ordering specified by each client’s program—as long as causally related operations are displayed in the right order. This is the **causal consistency** model, which requires that only operations that are causally related need to be seen in the same order by all the nodes.

##### Example

Consider the same scenario as our previous comments example. We may want to display comments out of chronological order if it means that every comment is displayed after the comment it replies to. This is expected since there is a [cause-and-effect](https://en.wikipedia.org/wiki/Causality) relationship between a comment and the comments that constitute replies to it.

Thus, unlike in sequential consistency, the operations that are not causally related can be seen in different orders in the various clients of the system, without the need to maintain the order of each client’s program. Of course, to achieve that, each operation needs to contain some information that signals whether it depends on other operations or not. This does not need to at all be related to time and can be an application-specific property.

Causal consistency is a weaker consistency model that prevents a common class of unintuitive behaviors.

#### Eventual consistency

There are still even simpler applications that do not have the notion of a cause-and-effect and require an even simpler consistency model. The **eventual consistency** model is beneficial here.

##### Example

For instance, we could accept that the order of operations can be different between the multiple clients of the system, and reads do not need to return the latest write as long as the system eventually arrives at a stable state. In this state, if no more write operations are performed, read operations will return the same result. This is the model of eventual consistency.

It is one of the weakest forms of consistency since it does not really provide any guarantees around the perceived order of operations or the final state the system converges to.

It can still be a useful model for some applications, which do not require stronger assumptions or can detect and resolve inconsistencies at the application level.

Note that there are many more [consistency models](https://en.wikipedia.org/wiki/Consistency_model) besides the ones we explained here.

When we think of a distributed system as a single node, it seems obvious that every operation happens at a specific instant of time, and is immediately visible to everyone. However, when we think of a distributed system as a set of cooperating nodes, we realize that we should not take this for granted.

For instance, the system in the above illustration is not linearizable since T4 > T3. However, still, the second client won’t observe the read because it has not yet propagated to the node that processes the read operation. The **non-linearizability** comes from the use of asynchronous replication.

When we use a synchronous replication technique, we make the system linearizable. However, that means that the first write operation takes longer until the new value has propagated to the rest of the nodes. Remember the latency-consistency trade-off from the PACELC theorem.

##### Benefits

As a result of the above discussion, we realize that linearizability is a very powerful consistency model. It helps us treat complex distributed systems as much simpler, single-node datastores, and reason about our applications more efficiently. Moreover, by leveraging atomic instructions provided by hardware (such as [CAS operations](https://en.wikipedia.org/wiki/Compare-and-swap)), we can build more sophisticated logic on top of distributed systems, such as mutexes, semaphores, counters, etc. This is not possible under weaker consistency models.

#### Sequential consistency

**Sequential consistency** is a weaker consistency model, where operations are allowed to take effect before their invocation or after their completion.

As a result, it provides no real-time guarantees. However, operations from different clients have to be seen in the same order by all other clients, and operations of every single client preserve the order specified by its program (in this global order). This allows many more histories than linearizability, but still poses some constraints that can help real-life applications.

##### Example

For example, in a social networking application, we usually do not care what’s the ordering of posts between some of our friends. However, we still expect posts from a single friend to be displayed in the right order (i.e., the one they published them at). Following the same logic, we usually expect our friends’ comments in a post to appear in the order that they submitted them. These are all properties that the sequential consistency model captures.

#### Causal consistency

In some cases, we don’t need to preserve the ordering specified by each client’s program—as long as causally related operations are displayed in the right order. This is the **causal consistency** model, which requires that only operations that are causally related need to be seen in the same order by all the nodes.

##### Example

Consider the same scenario as our previous comments example. We may want to display comments out of chronological order if it means that every comment is displayed after the comment it replies to. This is expected since there is a [cause-and-effect](https://en.wikipedia.org/wiki/Causality) relationship between a comment and the comments that constitute replies to it.

Thus, unlike in sequential consistency, the operations that are not causally related can be seen in different orders in the various clients of the system, without the need to maintain the order of each client’s program. Of course, to achieve that, each operation needs to contain some information that signals whether it depends on other operations or not. This does not need to at all be related to time and can be an application-specific property.

Causal consistency is a weaker consistency model that prevents a common class of unintuitive behaviors.

#### Eventual consistency

There are still even simpler applications that do not have the notion of a cause-and-effect and require an even simpler consistency model. The **eventual consistency** model is beneficial here.

##### Example

For instance, we could accept that the order of operations can be different between the multiple clients of the system, and reads do not need to return the latest write as long as the system eventually arrives at a stable state. In this state, if no more write operations are performed, read operations will return the same result. This is the model of eventual consistency.

It is one of the weakest forms of consistency since it does not really provide any guarantees around the perceived order of operations or the final state the system converges to.

It can still be a useful model for some applications, which do not require stronger assumptions or can detect and resolve inconsistencies at the application level.

Note that there are many more [consistency models](https://en.wikipedia.org/wiki/Consistency_model) besides the ones we explained here.

# CAP Theorem's Consistency Model

In this lesson, we will explore the consistency model that is used by the CAP theorem.

**We'll cover the following**

* [Consistency model used by the CAP theorem](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5156145765548032#Consistency-model-used-by-the-CAP-theorem)
* [Categorization of consistency models](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5156145765548032#Categorization-of-consistency-models)
  + [Strong consistency models](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5156145765548032#Strong-consistency-models)
  + [Weak consistency models](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5156145765548032#Weak-consistency-models)
* [Two commonly supported consistency models](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5156145765548032#Two-commonly-supported-consistency-models)
  + [Reasons](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5156145765548032#Reasons)

In the [previous](https://www.educative.io/collection/page/10370001/4891237377638400/4728805791367168#list-of-consistency-models) lesson, we saw the four fundamental consistency models. When explaining the CAP theorem, we encountered the term consistency. Let’s explore that now.

## Consistency model used by the CAP theorem

The “C” property in the CAP theorem refers to the linearizability model we previously described. This means it’s impossible to build a system that will be available during a network partition while also being linearizable.

In fact, there is research that shows that even some weaker forms of consistency, such as sequential consistency, cannot be supported in tandem with availability under a network partition.

This vast number of different consistency models creates a significant amount of complexity.

As we explained previously, modeling consistency is supposed to help us reason about these systems. However, the explosion of consistency models can have the opposite effect.

## Categorization of consistency models

The CAP theorem can conceptually draw a line between all these consistency models and separate them into two major categories:

* Strong consistency models
* Weak consistency models

### Strong consistency models

Strong consistency models correspond to the “C” in the CAP theorem and cannot be supported in systems that need to be available during network partitions.

### Weak consistency models

Weak consistency models are the ones that can be supported while also preserving availability during a network partition.

## Two commonly supported consistency models

Considering the guarantees provided by several popular distributed systems nowadays (i.e., Apache Cassandra, DynamoDB, etc.), two models are commonly supported.

* Strong consistency, specifically linearizability
* Weak consistency, specifically eventual consistency

### Reasons

Most probably, the reasons most of the systems converged to the above two models are the following:

* **Linearizability** was selected amongst the available strong consistency models because a system needs to give up availability as part of the CAP theorem to support a strong consistency model. It then seems reasonable to provide the strongest model amongst the available ones, facilitating the software engineers’ work with it.
* **Eventual Consistency** was selected amongst the available weak consistency models thanks to its simplicity and performance. Along the same lines, given the application relinquishes the strict guarantees of strong consistency for increased performance, it might as well accept the weakest guarantees possible to get the biggest performance boost it can. This makes it much easier for people to design and build applications on top of distributed systems, and to decide which side of the CAP theorem they prefer to build their application on.

# Isolation Levels and Anomalies

Let's see a list of Isolation levels and a detailed explanation of anomalies that may occur in distributed systems due to inherent concurrency.

**We'll cover the following**

* [Anomalies](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5872325153259520#Anomalies)
  + [Dirty write](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5872325153259520#Dirty-write)
  + [Dirty read](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5872325153259520#Dirty-read)
  + [Fuzzy or non-repeatable read](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5872325153259520#Fuzzy-or-non-repeatable-read)
  + [Phantom read](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5872325153259520#Phantom-read)
  + [Lost update](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5872325153259520#Lost-update)
  + [Read skew](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5872325153259520#Read-skew)
  + [Write skew](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5872325153259520#Write-skew)

The inherent **concurrency** in distributed systems creates the potential for **anomalies** and unexpected behaviors. Specifically, transactions that comprise multiple operations and run concurrently can lead to different results depending on how their operations are interleaved.

As a result, there is still a need for some formal models that define what is possible and what is not in a system’s behavior. These are called **isolation levels**.

We will study the most common ones here, which are the following:

* **Serializability:** It essentially states that two transactions, when executed concurrently, should give the same result as though executed sequentially.
* **Repeatable read:** It ensures that the data once read by a transaction will not change throughout its course.
* **Snapshot isolation:** It guarantees that all reads made in a transaction see a consistent snapshot of the database from the point it started and till the transaction commits successfully if no other transaction has updated the same data since that snapshot.
* **Read committed:** It does not allow transactions to read data that has not yet been committed by another transaction.
* **Read uncommitted:** It is the lowest isolation level and allows the transaction to read uncommitted data by other transactions.

Unlike the consistency models presented in the [Consistency Models lesson](https://www.educative.io/collection/page/10370001/4891237377638400/4728805791367168), some of these isolation levels do not define what is possible via some formal specification. Instead, they define what is not possible, i.e., which anomalies of the already known ones are prevented.

Of course, stronger isolation levels prevent more anomalies at the cost of performance. Let’s first have a look at the possible anomalies before examining the various levels.

The origin of the isolation levels above and the associated anomalies was essentially the ANSI SQL-92 standard. However, the definitions in this standard were ambiguous and missed some possible anomalies.

Subsequent research examines more anomalies extensively and attempts a stricter definition of these levels. The basic parts will be covered in this lesson, but please refer to this paper for a deeper analysis.

## Anomalies

The anomalies covered here are the following:

* Dirty writes
* Dirty reads
* (Fuzzy) non-repeatable reads
* Phantom reads
* Lost updates
* Read skew
* Write skew

### Dirty write

A **dirty write** occurs when a transaction overwrites a value that was previously written by another transaction that is still in-flight and has not been committed yet.

One reason dirty writes are problematic is they can violate **integrity constraints**. For example, there are two transactions A and B, where transaction A runs the operations [x=1, y=1] and transaction B runs the operations [x=2, y=2]. Then, the serial execution of them would always result in a situation where x and y have the same value. However, this is not necessarily true in a concurrent execution where dirty writes are possible.

**Example**

An example could be the following execution [x=1, x=2, y=2, commit B, y=1, commit A] that would result in x=2 and y=1.

Another problem with dirty writes is they make it impossible for the system to automatically rollback to a previous image of the database. As a result, this is an anomaly we need to prevent in most cases.

### Dirty read

A **dirty read** occurs when a transaction reads a value that has been written by another transaction that has not yet been committed.

This is problematic since the system might make decisions depending on these values, even though the associated transactions might be rolled back subsequently. Even in the case where these transactions eventually commit, though, this can still be a problem.

**Example**

An example is the classic scenario of a bank transfer, where the total amount of money should be observed to be the same at all times. For example, imagine transaction A is able to read the balance of two accounts involved in a transfer during another transaction (B) that performs the transfer from account 1 to account 2. During the transfer, it will look as if some money has been lost from account 1.

However, there are a few cases where allowing dirty reads can be useful if done with care. One such case is to generate a big aggregate report on a full table when we can tolerate some inaccuracies on the numbers of the report.

It can also be useful when we troubleshoot an issue and want to inspect the state of the database in the middle of an ongoing transaction.

### Fuzzy or non-repeatable read

A **fuzzy or non-repeatable read** occurs when a value is retrieved twice during a transaction (without it being updated in the same transaction), and the value is different.

This can lead to problematic situations similar to the example presented above for dirty reads.

Other cases where this can lead to problems are when the first read of the value is used for some conditional logic, and the second is used to update data. In this case, the transaction might act on stale data.

### Phantom read

A **phantom read** occurs when a transaction does a predicate-based read, and another transaction writes or removes a data item matched by that predicate while the first transaction is still in flight. If that happens, then the first transaction might be acting again on stale data or inconsistent data.

**Example**

For example, transaction A runs 2 queries to calculate the maximum and the average age of a specific set of employees. However, between the two queries, transaction B is interleaved and inserts a lot of old employees in this set, thus making transaction A return an average that is larger than the maximum.

Allowing phantom reads can be safe for an application that does not make use of predicate-based reads, i.e., performs only the reads that select records using a primary key.

### Lost update

A **lost update** occurs when two transactions read the same value and then try to update it to two different values. The end result is that one of the two updates survives, but the process executing the other update is not informed that its update did not take effect. Thus it is called a lost update.

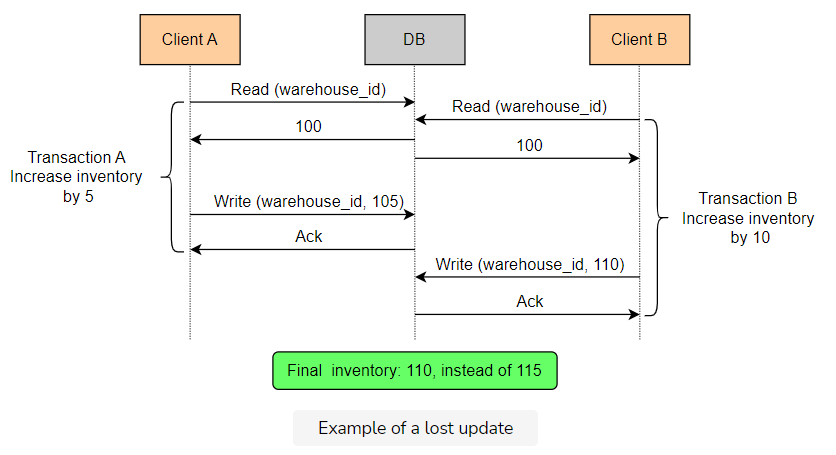
**Example**

Imagine a warehouse with various controllers that update the database when new items arrive. The transactions are rather simple. They involve reading the number of items currently in the warehouse, adding the number of new items to this number, and then storing the result back in the database.

This anomaly could lead to the following problem:

* Transactions A and B read the current inventory size simultaneously (say, 100 items), add the number of new items to this (say, 5 and 10 respectively), and then store this back to the database. Let’s assume that transaction B was the last one to write. This means that the final inventory is 110 instead of 115. Thus, five new items are not recorded!

See the following illustration for a visualization of this example.



Depending on the application, it might be safe to allow lost updates in some cases. For example, consider an application that allows multiple administrators to update specific parts of an internal website used by employees of a company.

### Read skew

A **read skew** occurs when there are integrity constraints between two data items that seem to be violated because a transaction can only see partial results of another transaction.

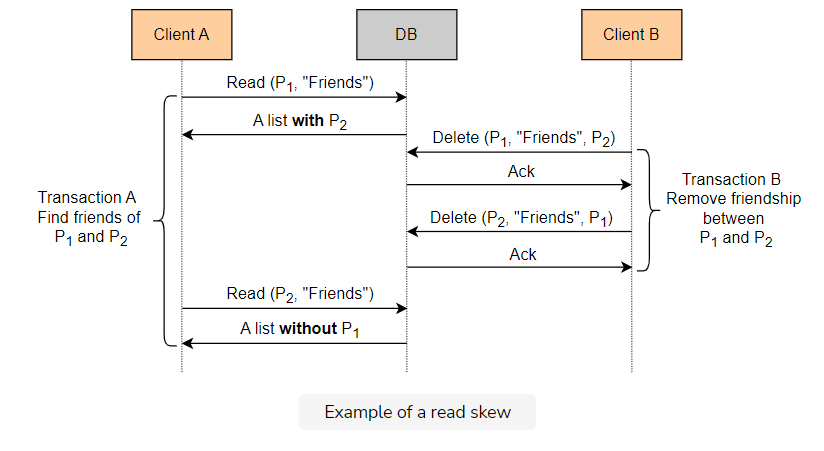
**Example**

Let’s imagine an application that contains a table of persons, where each record represents a person and contains a list of all the friends of this person. The main integrity constraint is that friendships are mutual; if person B is included in person A’s list of friends, then A must also be included in B’s list. Every time someone (say, P1) wants to unfriend someone else (say, P2), a transaction is executed that removes P2 from P1’s list and also removes P1 from P2’s list in a single go.

Now, let’s also assume that some other part of the application allows people to view friends of multiple people simultaneously. This is done by a transaction that reads the friends list of these people. If the second transaction reads the friends list of P1 before the first transaction has started, but reads the friends list of P2 after the second transaction has been committed, it will notice an integrity violation. P2 will be in P1’s list of friends, but P1 will not be in P2’s list of friends.

Note that this case is not a dirty read, because any values written by the first transaction are read-only after it has been committed.

See the following illustration for a visualization of this example.



A strict requirement to prevent read skew is quite rare, as we might have guessed already. For example, a common application of this type might allow a user to view the profile of only one person at a time along with their friends, thus not requiring the integrity constraint described above.

### Write skew

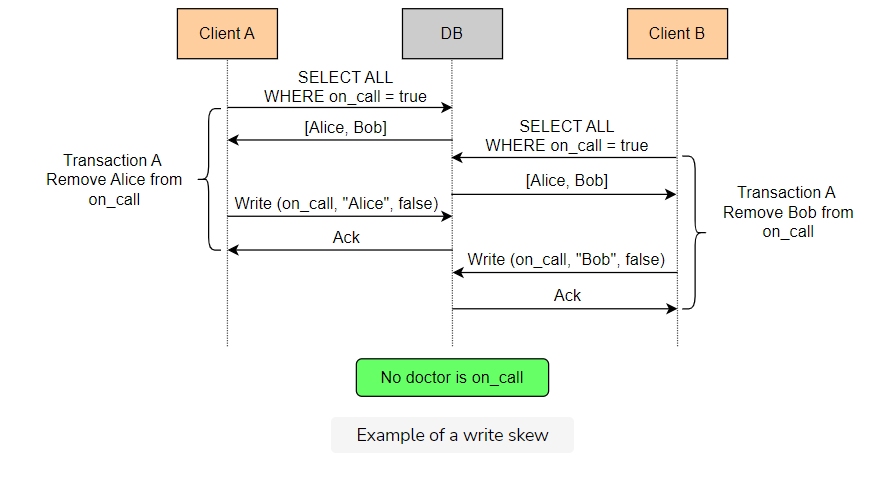
A **write skew** occurs when two transactions read the same data, but then modify disjoint sets of data.

**Example**

Imagine an application that maintains the on-call rotation of doctors in a hospital. A table contains one record for every doctor with a field indicating whether they are on-call. The application allows a doctor to remove themself from the on-call rotation if another doctor is also registered. This is done via a transaction that reads the number of doctors that are on-call from this table. If the number is greater than one, it updates the record corresponding to this doctor to not be on-call.

Now, let’s look at the problems that can arise from the write skew phenomenon. Let’s say two doctors, Alice and Bob, are on-call currently, and they both decide to see if they can remove themselves. Two transactions running concurrently might read the state of the database, see there are two doctors, and remove the associated doctor from being on-call. In the end, the system ends up with no doctors on-call!

See the following illustration for visualization of this example.



It is evident by now that there are many different anomalies for us to consider. On top of that, different applications manipulate data in different ways. So, we have to analyze each case separately to see which anomalies can create problems.

# Prevention of Anomalies in Isolation Levels

In this lesson, we will identify which isolation level prevents which anomalies.

**We'll cover the following**

* [Isolation level that prevents all of the anomalies](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5668174586707968#Isolation-level-that-prevents-all-of-the-anomalies)
* [Other isolation levels](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5668174586707968#Other-isolation-levels)

## Isolation level that prevents all of the anomalies

There is one isolation level that prevents all of these anomalies: the **serializable** one.

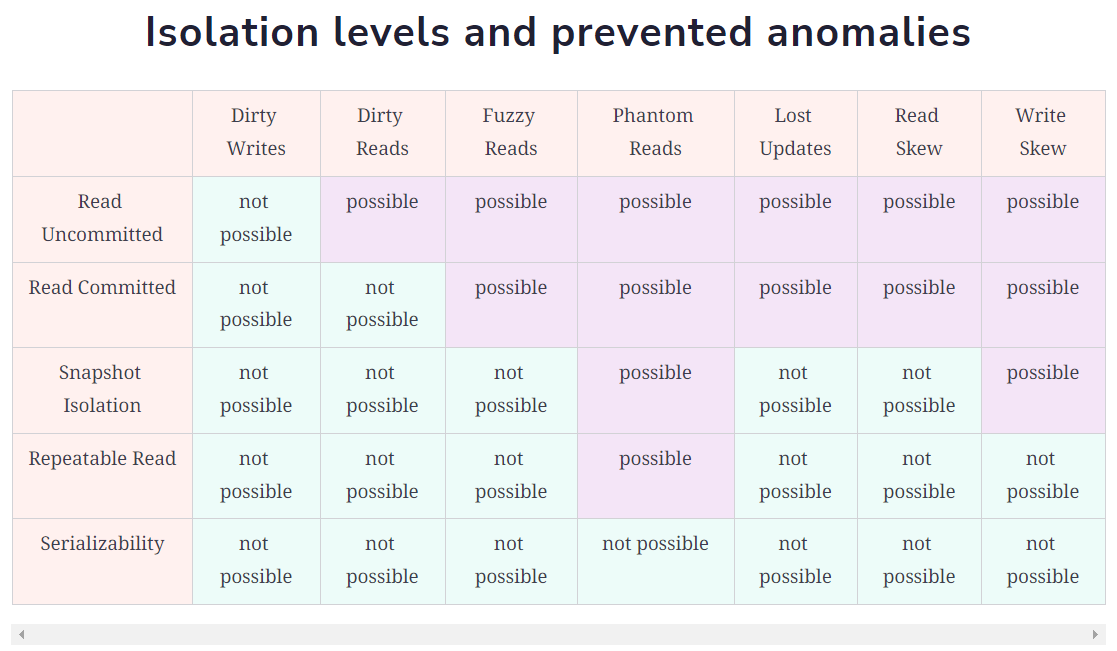
Like the consistency models presented in the [Consistency Models](https://www.educative.io/collection/page/10370001/4891237377638400/4728805791367168) lesson, this level provides a more formal specification of what is possible, e.g., which execution histories are possible. More specifically, it guarantees that the result of the execution of concurrent transactions is the same as that produced by some serial execution of the same transactions. This means that we can only analyze serial executions for defects. If all the possible serial executions are safe, then any concurrent execution by a system at the serializable level will also be safe.

However, serializability has performance costs since it intentionally reduces concurrency to guarantee safety.

## Other isolation levels

Isolation levels other than the serializable ones are less strict and provide better performance via increased concurrency at the cost of decreased safety.

These models allow some of the anomalies we described previously. The following illustration contains a table with the most basic isolation levels, along with the anomalies they prevent.



These isolation levels originated from the early relational database systems that were not distributed. Still, they are applicable in distributed datastores too.

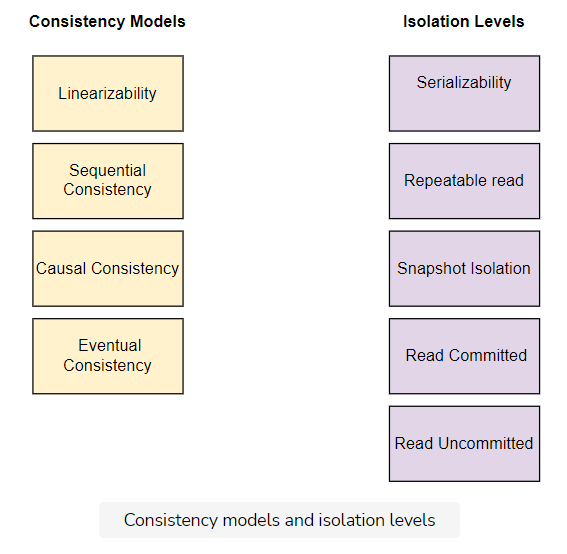
# Consistency and Isolation

Let's examine the differences and similarities between consistency models and isolation levels.

**We'll cover the following**

* [Similarities between consistency models and isolation models](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5958231310729216#Similarities-between-consistency-models-and-isolation-models)
* [Differences between consistency models & isolation levels](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5958231310729216#Differences-between-consistency-models-&-isolation-levels)
  + [Why real-time guarantees are important](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5958231310729216#Why-real-time-guarantees-are-important)
* [Strict serializability](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5958231310729216#Strict-serializability)

The following illustration will help us remember the consistency models and isolation levels.



## Similarities between consistency models and isolation models

It is interesting to observe that isolation levels are not that different from consistency models.

Isolation levels and consistency models are essential constructs that allow us to express:

* Which executions are possible
* Which executions are not possible

In both cases, some of the models are stricter and allow fewer executions, thus providing increased safety at the cost of reduced performance and availability.

For instance, linearizability allows a subset of the executions causal consistency allows, while serializability allows a subset of the executions that snapshot isolation allows.

We can also express this strictness relationship by saying that one model implies another model.

The fact that a system provides linearizability automatically implies that the same system also provides causal consistency.

Note that there are some models that are not directly comparable, which means neither of them is stricter than the other.

## Differences between consistency models & isolation levels

Consistency models and isolation levels have some differences with regards to the characteristics of their allowed and disallowed behaviors.

* Consistency models are applied to single-object operations (e.g. read/write to a single register), while isolation levels are applied to multi-object operations (e.g. read and write from/to multiple rows in a table within a transaction).

Looking at the strictest models in these two groups, linearizability and serializability, there is another important difference.

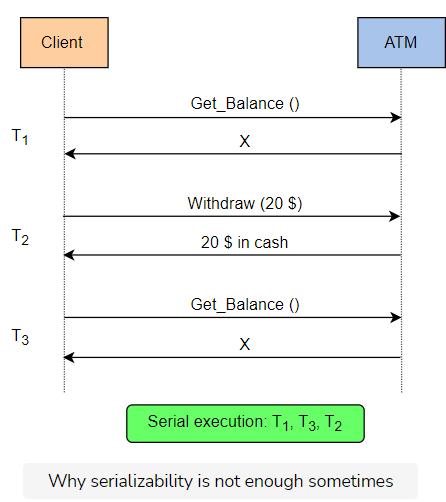
* Linearizability provides real-time guarantees, while serializability does not.

Linearizability guarantees that the effects of an operation took place at some point between when the client invoked the operation, and when the result of the operation was returned to the client.

Serializability only guarantees that the effects of multiple transactions will be the same as if they run in serial order. It does not provide any guarantee on whether that serial order would be compatible with real-time order.

### Why real-time guarantees are important

The following illustration shows why real-time guarantees are important from an application perspective.



Think of an automated teller machine that can support two transactions:

1. GET\_BALANCE()
2. WITHDRAW(amount)

The first transaction performs a single operation to read the balance of an account.

The second operation reads the balance of an account, reduces it by the specified amount, and then returns the client the specified amount in cash.

Let’s also assume this system is serializable.

Now, let’s examine the following scenario:

A customer with an initial balance of x reads their balance and then decides to withdraw $20 by executing a WITHDRAW(20) transaction.

After the transaction has been completed and the money is returned, the customer performs a GET\_BALANCE() operation to check their new balance. However, the machine still returns x as the current balance instead of x-20.

Note that this execution is serializable and the end result is as if the machine executed the GET\_BALANCE() transactions first, and then the WITHDRAW(20) transaction in a completely serial order.

This example shows how serializability is not sufficient in itself in some cases.

**Strict serializability**

**Strict serializability** is a model that is a combination of *linearizability* and *serializability*.

This model guarantees that the result of multiple transactions is equivalent to the result of a serial execution of them, and is also compatible with the real-time ordering of these transactions.

As a result, transactions appear to execute serially, and the effects of each of them take place at some point between their invocation and completion.

As we learned before, *strict* serializability is often a more useful guarantee than *plain* serializability.

In centralized systems, however, providing strict serializability is simple and just as efficient as only providing serializability guarantees. As a result, systems such as relational databases sometimes advertise *serializability* guarantees, while they actually provide *strict serializability*.

This is not necessarily true in a distributed database, where providing *strict serializability* can be more costly because it requires additional coordination.

It is important to understand the difference between these two guarantees in order to determine which one is needed, depending on the application domain.

# Hierarchy of Models

Let's look at the hierarchy of the consistency models and isolation levels based on the strictness and guarantees they provide.

**We'll cover the following**

* [Hierarchy tree](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6450207734890496#Hierarchy-tree)

Models can be organized in a hierarchical tree according to their strictness and the guarantees they provide.

## Hierarchy tree

The following illustration depicts such a tree that contains some of the models.

