**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.

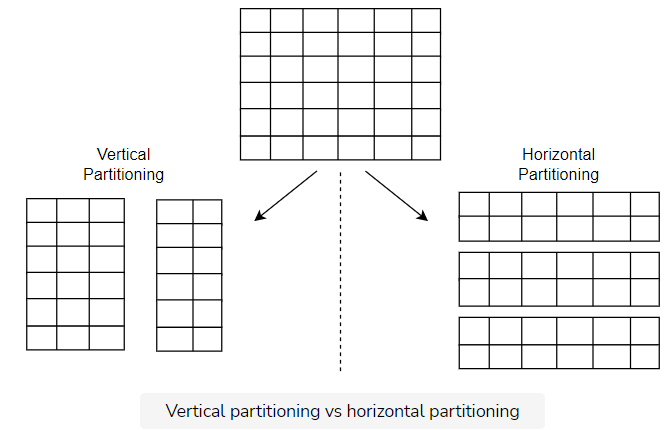
There are two different variations of partitioning:

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.

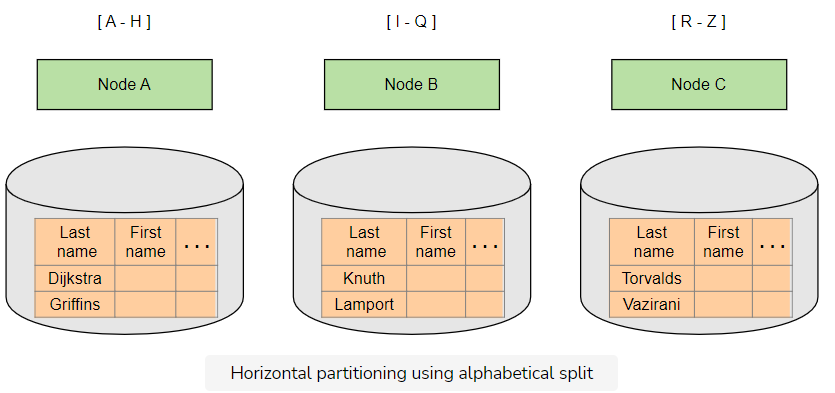
[**Normalization**](https://en.wikipedia.org/wiki/Database_normalization) is one way to perform vertical partitioning. However, general vertical partitioning goes far beyond that: it splits a column, even when they are normalized.

#### 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.

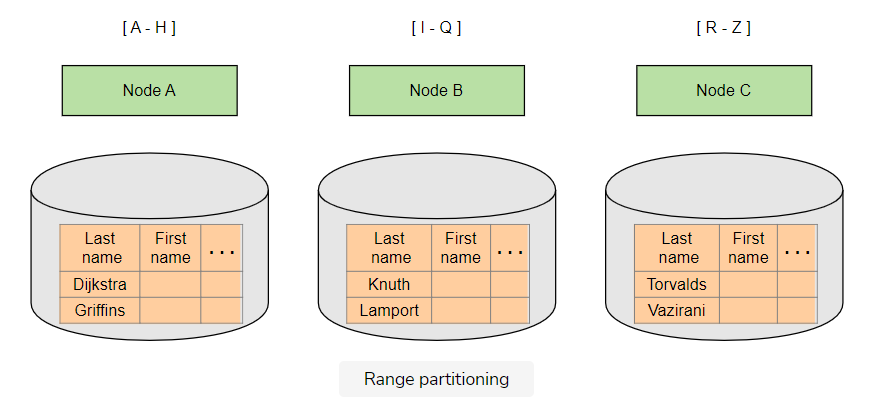
**We'll cover the following**

* [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.

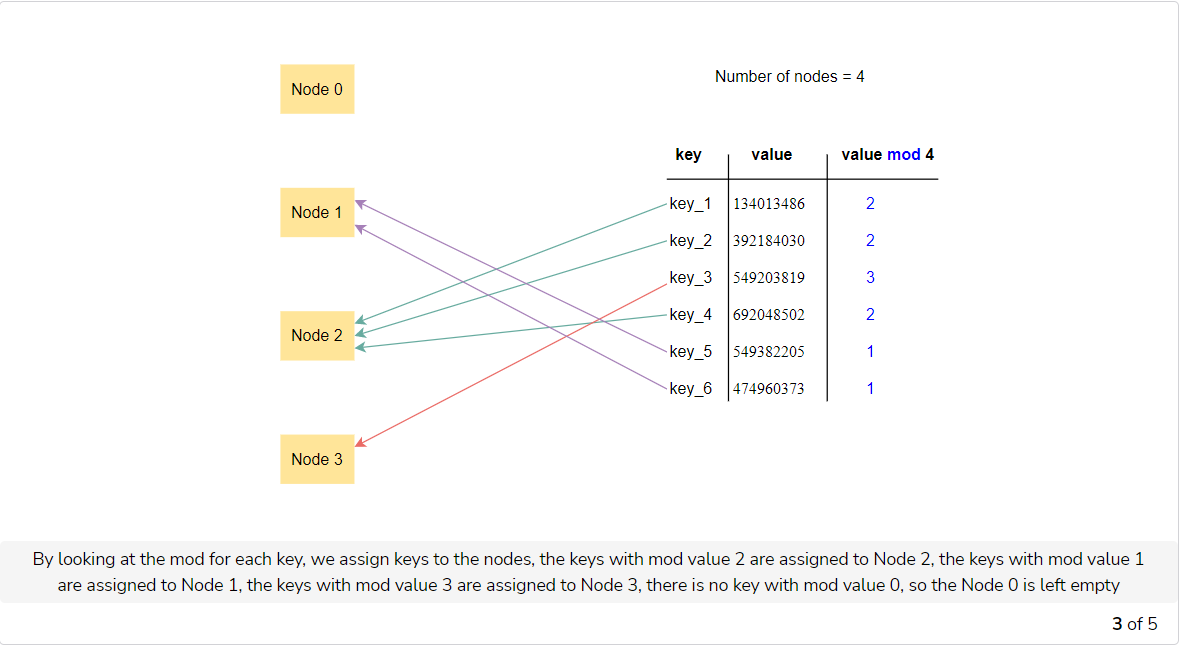
Some systems that leverage a range partitioning technique are Google’s BigTable, and [Apache HBase](https://hbase.apache.org/).

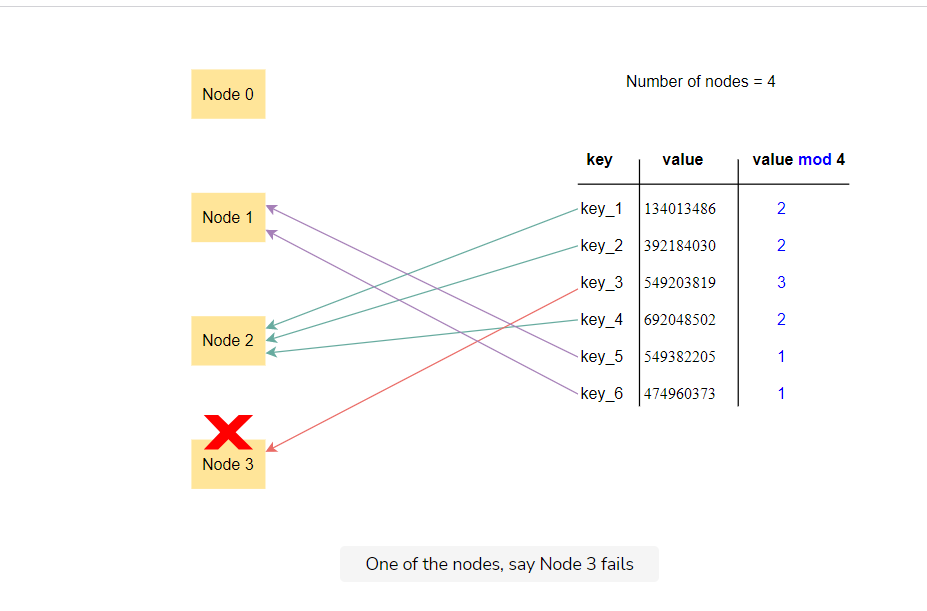
## Hash partitioning

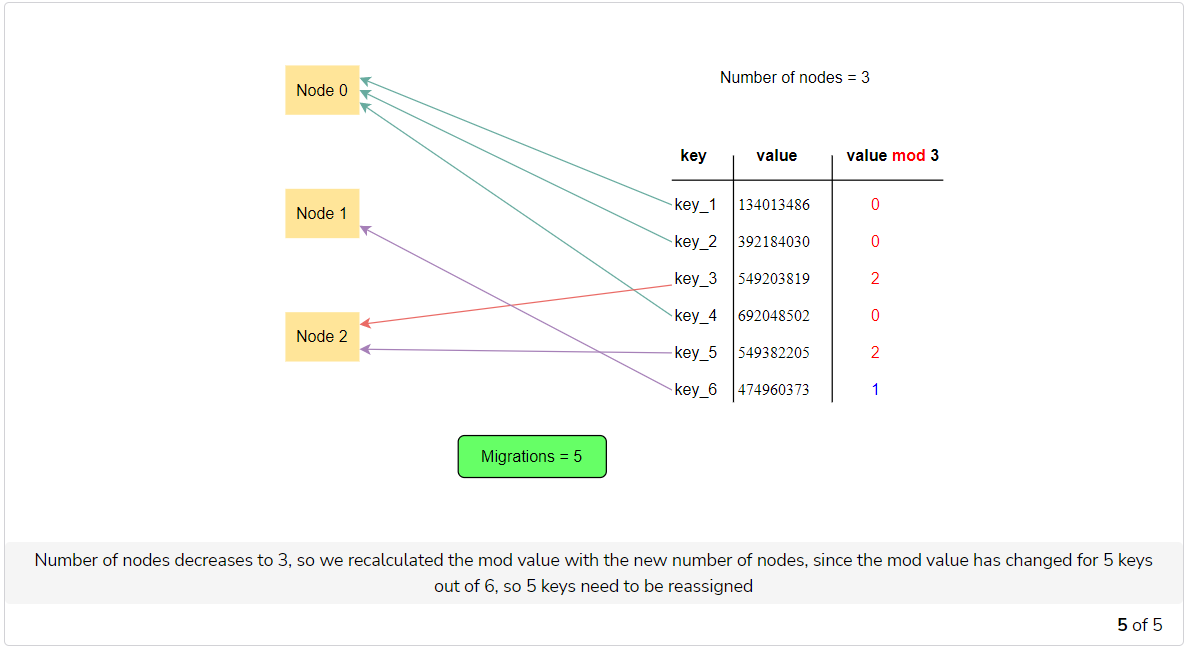
**Hash partitioning** is a technique where we apply a hash function to a specific attribute of each row. This results in a number that determines which partition—and, thus, node—this row belongs to.

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

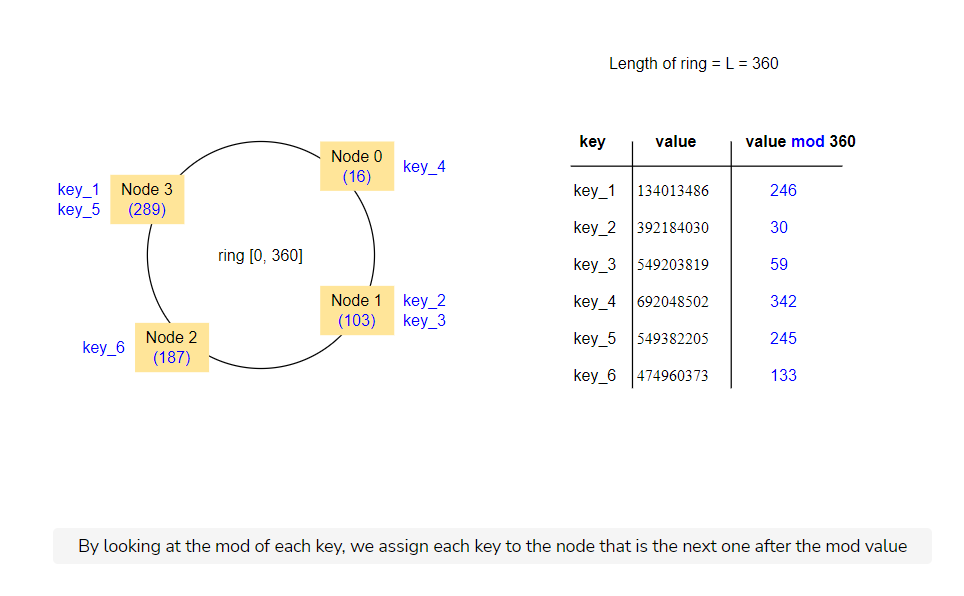
## Consistent hashing

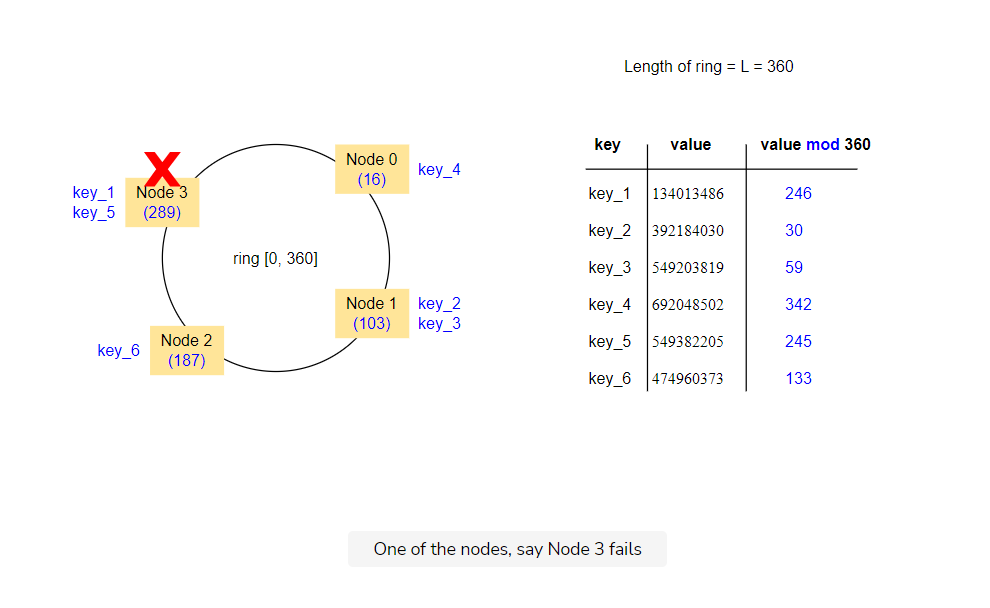
**Consistent hashing** is a partitioning technique that is very similar to hash partitioning, but solves the increased data movement problem caused by hash partitioning.

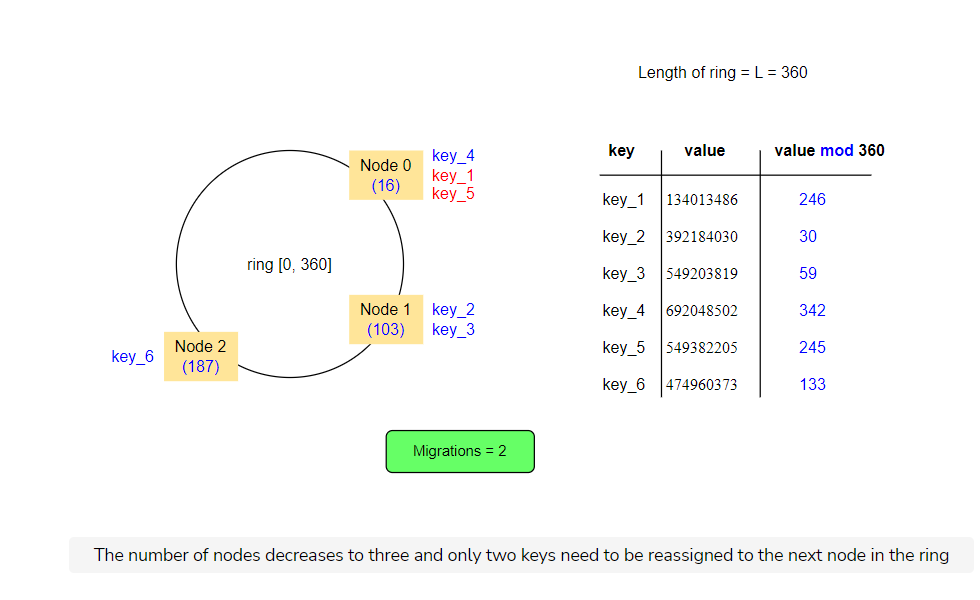
This is how it works: each node in the system is randomly assigned an integer in a range of [0, L]. This range is called ring (for example, [0, 360]). Then, the system uses a record with an attribute value s as a partitioning key to locating the node after the point hash(s) mod L in the ring.

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

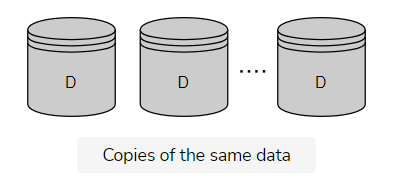
Availability refers to the ability of the system to remain functional despite failures in parts of it.

## 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**

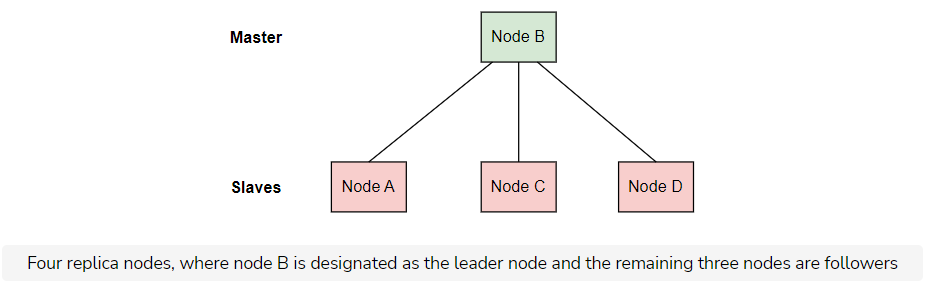
* [Single-master replication](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Single-master-replication)
  + [Techniques for propagating updates](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814832519708672#Techniques-for-propagating-updates)
    - [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

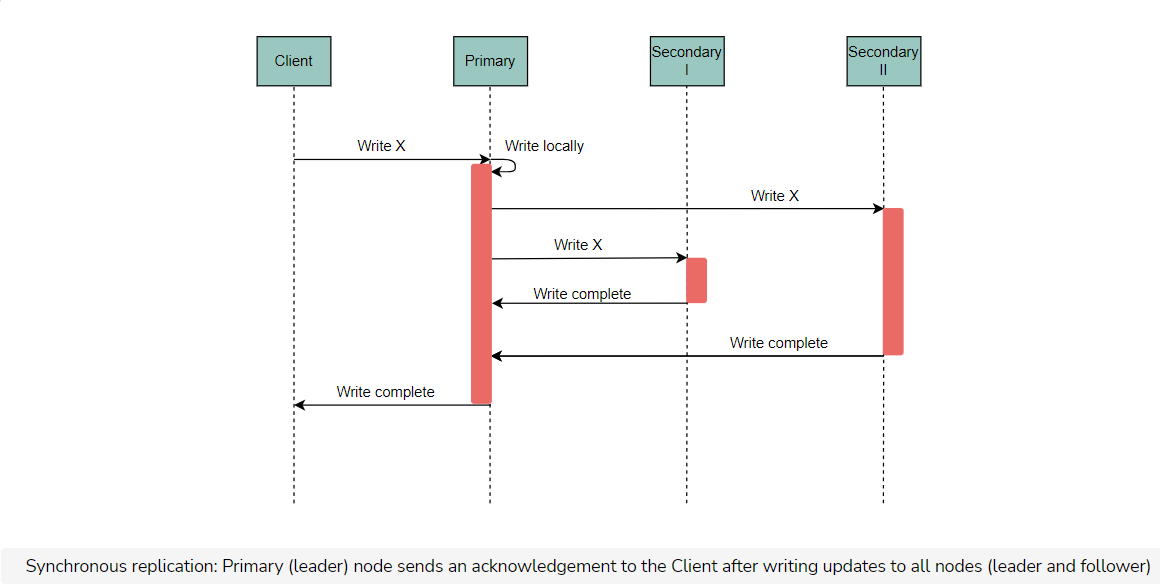
There are two ways to propagate the updates: synchronously and asynchronously.

#### 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.

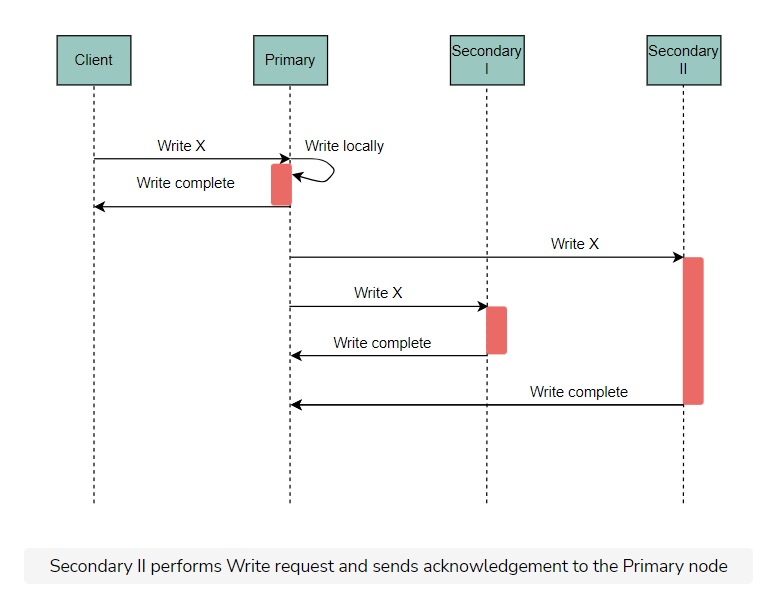


#### Asynchronous replication

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

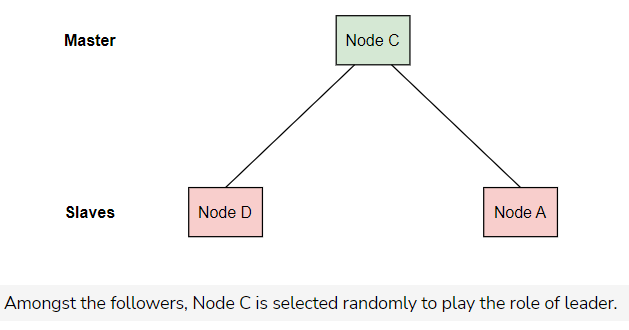
* It is simple to understand and implement
* Concurrent operations serialized in the leader node, remove the need for more complicated, distributed concurrency protocols. In general, this property also makes it easier to support transactional operations
* It is scalable for read-heavy workloads, because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

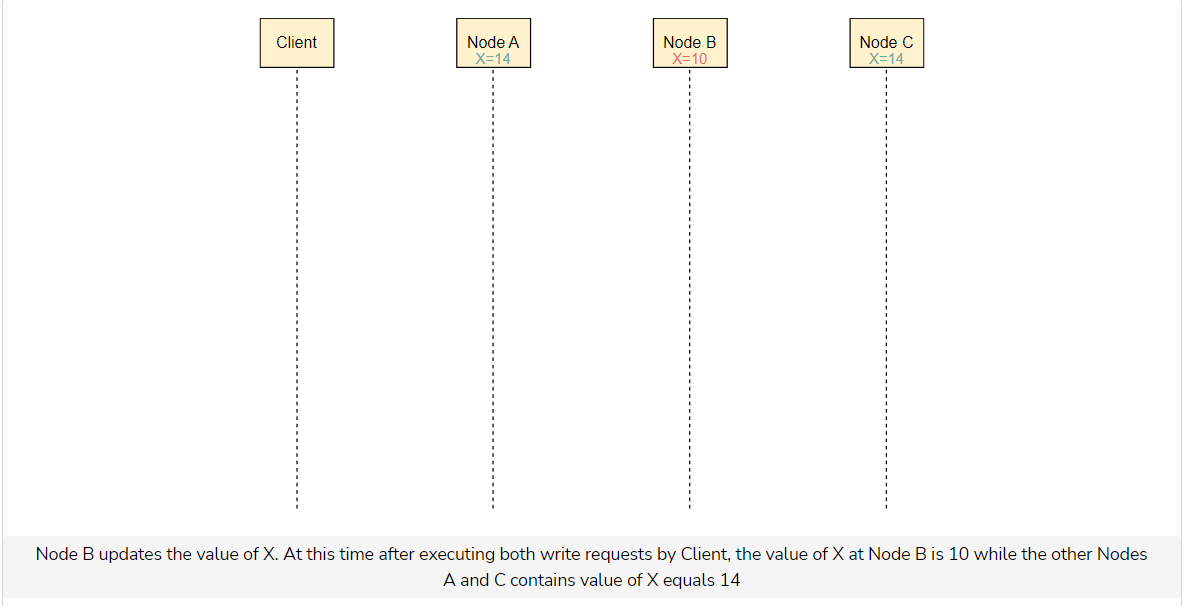
This technique is also known as **multi-primary replication**.

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.