Replication

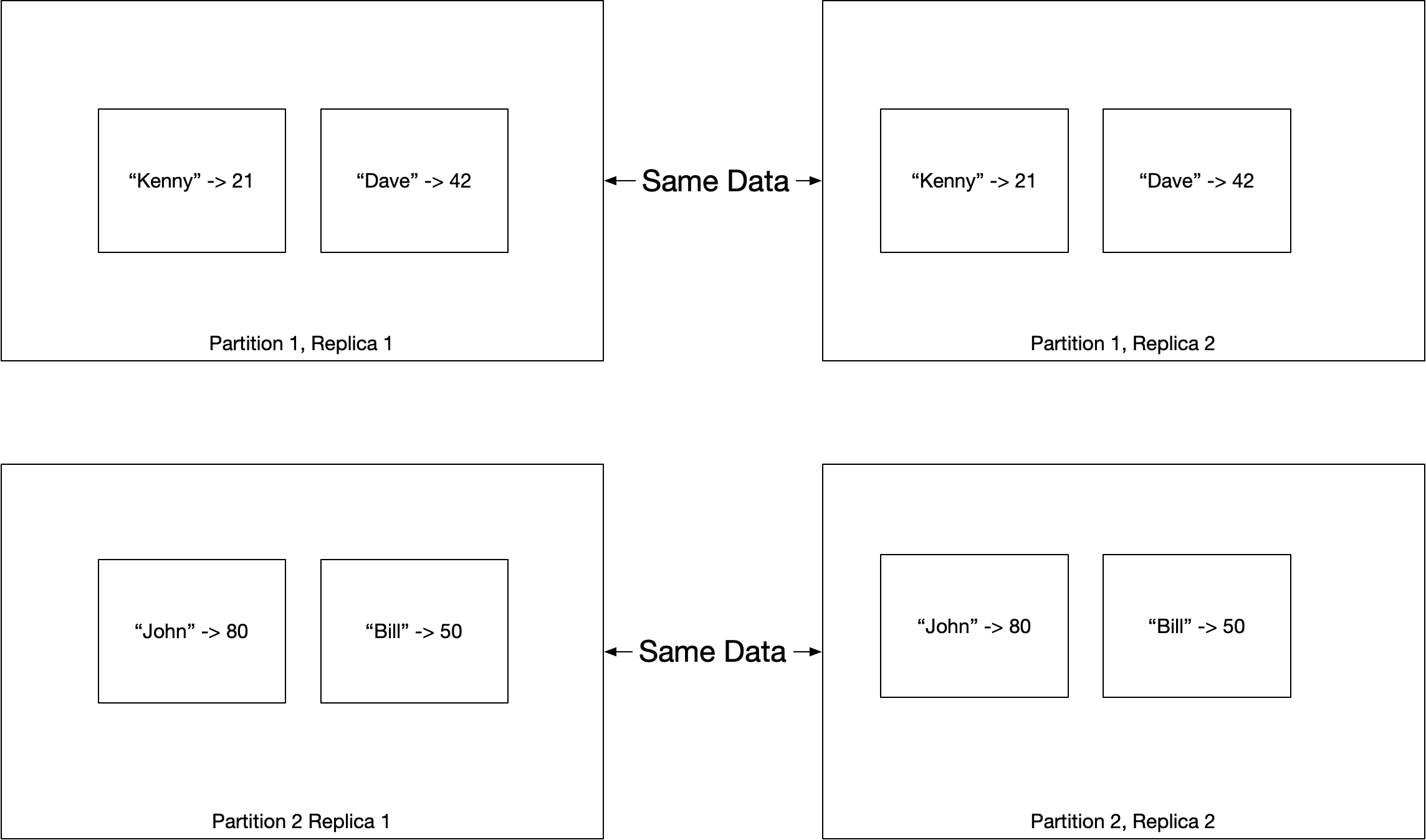
## Overview

We scale out in one of two ways.

* Replication – storing the same data on multiple nodes.
* Partitioning/Sharding – Splitting the data over multiple nodes.

The two approaches are not mutually exclusive and indeed often a system will combine both approaches. In this document we look at replication.

Table Partitioning and Replication



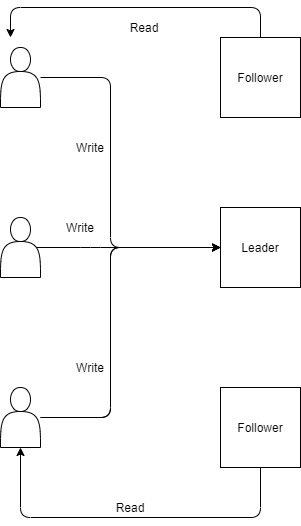
## Why Replicate?

* Increase fault tolerance.
* Increase availability.
* Increase read throughput.

The biggest problem with replication is keeping the data on each replica in sync. We consider three different strategies, single leader, multi leader, and leaderless.

## Single Leader Replication

A single leader system is also known as leader-based replication, active/passive, or master/slave. This methodology is supported by PostgreSQL, MySQL, Oracle, SQL Server, MongoDB, Kafka and RabbitMQ. All writes go through the leader; however, clients can read from any follower thus increasing concurrent read throughput.



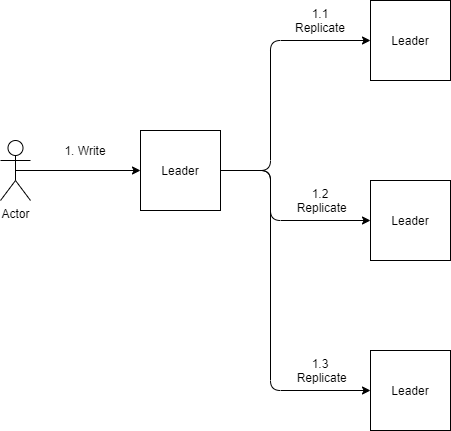
### When to use

Works well when a high percentage of requests are reads and a small percentage of requests are writes. Much of the web fits this use case.

### Updates

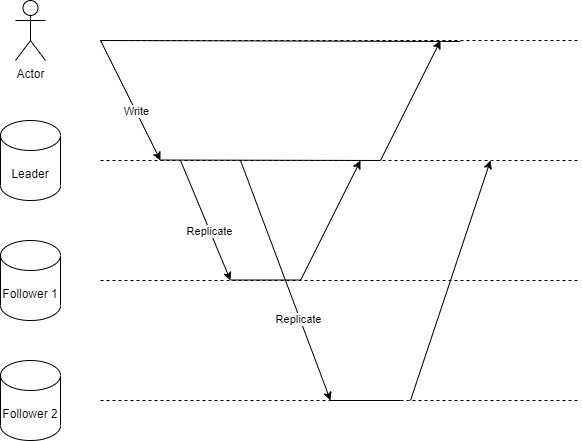
The algorithm works as follows.

1. All clients write to the leader (AKA master or primary)
2. The leader writes data to its local storage
3. The leader then updates the followers (AKA slaves or secondaries) by sending a replication log (AKA change stream)



#### Synchronous/Asynchronous Replication

When replicating from a leader to followers we need to decide whether to use synchronous or asynchronous replication. If we replicate to a follower synchronously, we can be sure that the follower has a fully consistent replica of the data when we return to the caller. The downside is that if the follower does not respond the system grinds to a halt. One common scenario is to use synchronous replication to a single follower and asynchronous replication to the other follower. In this way we are guaranteed to have at least one fully consistent copy of the data for fault tolerance purposes. The diagram below shows two followers. The leader updates the first follower synchronously and the second asynchronously.



#### Other Considerations

##### New Followers

When a new follower is added it takes a snapshot of the leader’s data and makes a note of the log sequence number of the most recent record in the snapshot. Once it is up and running it asks the leader for all updates that are more recent than its log sequence number.

##### Downtime of Follower

If a node is down, either due to failure or planned maintenance, when it comes back up it needs to update its state to reflect any updates that occurred when it was down. It does this by asking the leader to send all updates that have occurred since the log sequence number of the last update it received before going down.

##### Failover

When a leader goes down, we need to failover to a new leader. Typically, the new leader is chosen amongst the followers as the node with the most up to date data.

#### Replication Lag

In a read-scaling architecture we can increase read capacity without bounds by adding nodes however this is only practical when replication occurs asynchronously. Asynchronous replication can lead to temporal inconsistencies where a client reading from a leader and follower at the same time sees different results. The inconsistencies are only temporary because if we wait for the asynchronous replication to complete the system will be eventually consistent. When they is high load or network problems replication lag can run to minutes or even longer.

#### Read After Write Consistency

If we guarantee that we read our own writes and get consistent results we have a number of options.

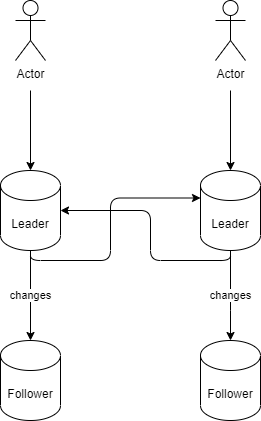
1. If we know something could have been modifies read it from the leader
2. Read from leader if the elapsed time since last read is under some threshold
3. Client can ensure it only reads from replica whose last write was more recent than clients last write.

#### Monotonic Reads

We can ensure if a client makes sever reads it will not see old results after fetching newer results. One way to achieve this is to make sure one client always reads from the same replica.

## Multi Leader Replication

With multi leader replication more than one node accepts writes. This organisation is also known as active/active or master/master Typically we would have one cluster per data center each with one leader.



Because we have writes to multiple nodes, we have the problem of write conflicts. For this reason, one should only consider multi-reader configurations for a really good reason.

## Leaderless Replication

With leaderless replication clients can write to any replica. Consider a topology that contains a total of n nodes. Typically reads and writes are sent to all n node in the topology. Thresholds r and w determine how many nodes must respond on a read and write respectively for the relevant operation to be considered successful. Let us consider the implications of various values of n,r,w

|  |  |
| --- | --- |
| Property | Implication |
|  | we are guaranteed that a read has the most up to date data |
|  | We can write when we have failed nodes |
|  | We can read when we have failed nodes |

### Updating Stale Nodes

When a client reads from multiple nodes it can update the nodes which have stale data at the same time. In addition, a background thread can run that also updates stale nodes.

## Partitioning

Sharding splits data across multiple data stores in such a way that we can work out which information is on which host. For an in-depth discussion of partitioning see

[Azure Best Practices For Data Partitioning](https://docs.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning)

Table Benefits of Partitioning

|  |  |
| --- | --- |
| Scalability | Dividing data across multiple data stores prevents us being limited by the physical limits of a single store |
| Performance | Splitting data cross multiple data stores can lead to better performance as we need to search through a smaller amount of data on each partitioned store. |
| Flexibility | We can allocate different types of data to different types of data store. In this way the data store used is the one most appropriate for the type of data. |
| Availability |  |

There are several different ways of partitioning the data.

### Horizontal Partitioning / Sharding

Each partition is separate data store. All data stores have the same schema and holds a subset of the data. The following sections describe some strategies for allocating subsets to partitions

#### Range Strategy

### Vertical

Each partition holds a subset of the fields. Fields are divided according to how they are used e.g. frequently accessed fields might go into one partition.

#### Functional

The partitions are determined by the bounded contexts of the architectural solution. We might put orders in one partition and product definitions in another partition.

#### Key/Hash Based

Given N servers put the data on mod(key,n). As we add servers we need to repartition all the data which is expensive.

#### Directory based

Use a lookup table to prevent repartitioning as we add servers. The drawback of this approach is that the lookup can become a single source of failure and the extra level of performance can impact performance.