

# Chapter 5: Replication

Chapter 5 of our book covers the important topic of replication in the context of database systems. Replication is the process of creating and maintaining multiple copies of the same database on different servers. This is done to ensure data availability, improve performance, and provide fault tolerance.

The chapter starts by introducing the concept of replication and its benefits. It then goes on to explain the different types of replication, including master-slave, master-master, and multi-master replication. Each type is discussed in detail, with its advantages and disadvantages.

The chapter also covers the different replication topologies, such as star, tree, and mesh. It explains how each topology works and when it is appropriate to use them.

Next, the chapter discusses the issues related to replication, such as consistency, conflicts, and latency. It explains how these issues can be addressed using techniques such as locking, timestamping, and conflict resolution.

Finally, the chapter concludes with a discussion on the challenges of replication and how they can be overcome. It also provides some best practices for designing and implementing replication systems.

Overall, Chapter 5 provides a comprehensive overview of the replication process and its importance in database systems. It is a must-read for anyone interested in building scalable and fault-tolerant database systems.

**Date:** @March 13, 2023 9:20 PM

**Topic:** PART II - Distributed Data

## Recall

Why do you need to embrace

## Notes

- Scalability: workload grows bigger than a single machine
- Fault tolerance/high availability

distributed  
databases?

- Latency

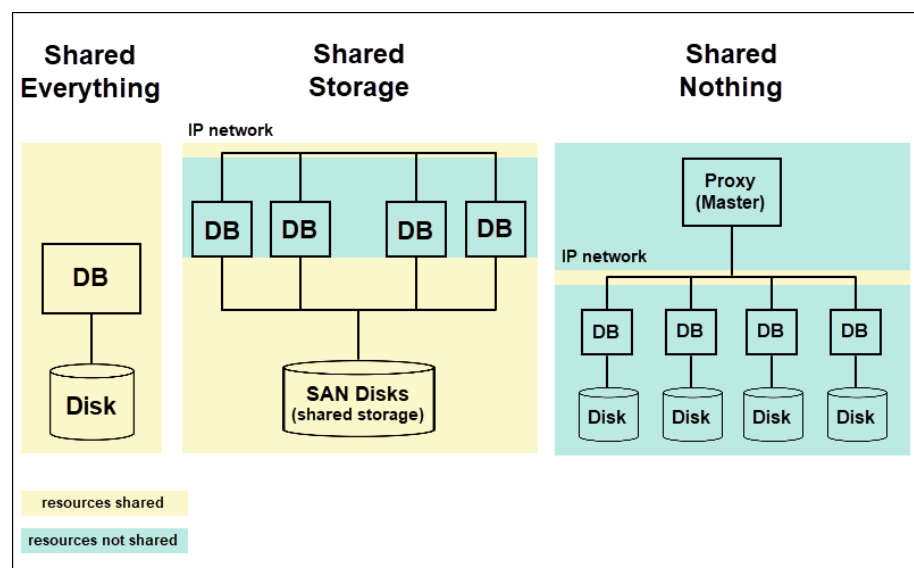
Scaling to a Higher  
load approaches

1. Vertical scaling (shared-memory architecture)
2. shared-disk architecture
3. Horizontal scaling (shared-nothing architecture)

The problems with  
shared-memory  
architecture?

- the cost grows faster than linearly
- limited fault tolerance
- higher latency

What's shared-  
nothing  
architecture? Cons  
& Pros



## #1 Replication

## #2 Partitioning

The two common ways to distribute data across multiple nodes

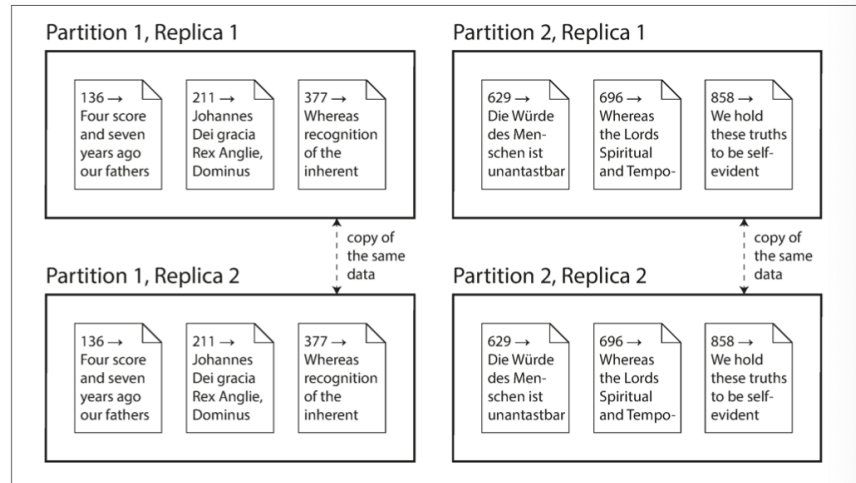


Figure II-1. A database split into two partitions, with two replicas per partition.



**SUMMARY:** To achieve scalability, fault tolerance, and low latency, it is necessary to use distributed databases.

**Date:** @March 13, 2023 9:57 PM

**Topic:** Chapter 5: Replication

### Recall

Why replication?

### Notes

- Scalability: workload grows bigger than a single machine
- Fault tolerance/high availability
- Latency



## SUMMARY:

*Replication* means keeping a copy of the same data on multiple machines that are connected via a network. All of the difficulty in replication lies in handling *changes* to replicated data. This chapter discusses three commonly used approaches: *single-leader*, *multi-leader*, and *leaderless* replication

**Date:** @March 13, 2023 10:05 PM

## Topic: Leaders & Followers

### Recall

How do we ensure that all the data ends up on all the replicas with multiple replicas?

### Notes

- Pick up a leader node. Write have to happen on this node. Other replicas are named followers.
- Whenever the leader writes new data, it generates replication log or change stream and then sends them to followers.
- When a client wants to read from the database, it can query either from the leader or from any of the followers.

Synchronous and Asynchronous replication

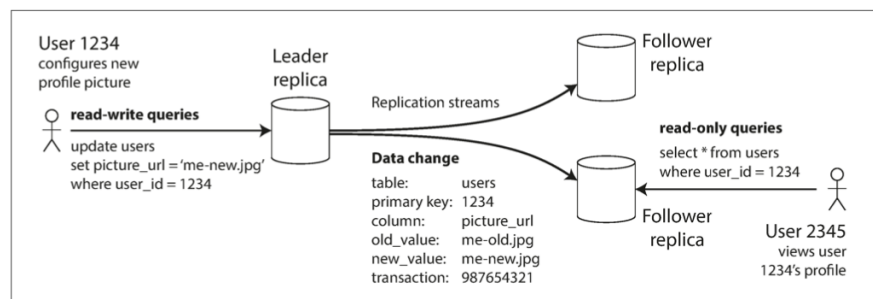


Figure 5-1. Leader-based (master-slave) replication.

## Cons & Pros of Synchronous replication

## Cons & Pros of Asynchronous

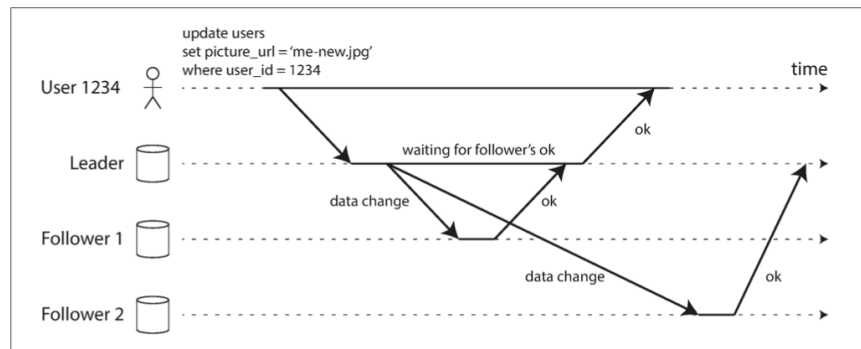


Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.

cons: up-to-date copy

pros: write depends on followers

Why completely asynchronous is widely used nevertheless it has weakening durability guarantees.

pros: write is not guaranteed to be durable. incomplete replication possibly lost in the case of the leader outage.

cons: leader can continue processing writes even if all of its followers have fallen behind.

**SUMMARY:**

Each node that stores a copy of the database is called a *replica*. It is impractical for all followers to be synchronous: any one node outage would cause the whole system to grind to a halt. In practice, if you enable synchronous replication on a database, it usually means that *one* of the followers is synchronous, and the others are asynchronous(*semi-synchronous*).

**Date:** @March 14, 2023 8:55 PM

## **Topic: Setting Up New Followers & Handling Node Outages**

### **Recall**

How do we ensure the newly added follower has an intact copy of the leader's data without downtime?

### **Notes**

1. take a consistent snapshot
2. copy the snapshot to the new follower
3. `binlog` coordinates: get replication log of delta changes
4. caught up.

How do we handle follower nodes outage?

### **Catch-up recovery**

- keeps durable change logs
- when crashed or network issue recovery from its log

### **Failover**

How do we handle leader node outages?

1. *Determining that the leader has failed (timeout)*
2. *Choosing a new leader (run election process, consensus problem)*
3. *Reconfiguring the system to use the new leader. Clients send writes to the new leader.*

Probably problems with Failover?

1. inconsistent data, the new leader may not have received all the writes from the old leader with an asynchronous replication approach. → semi-synchronous replication
2. discarding writes is bad. Caused primary key conflicts in the incident at GitHub.
3. split brain: two nodes both believe they are the leader. consensus issue
4. `timeout` to declare the leader dead is not deterministic. Too long or too short.



#### SUMMARY:

Our goal is to keep the system as a whole running despite individual node failures and to keep the impact of a node outage as small as possible.

When the leader is failed, one of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader. This process is called **failover**.

These issues—node failures; unreliable networks; and trade-offs around replica consistency, durability, availability, and latency—are in fact fundamental problems in distributed systems. There are no easy solutions to these problems. We have to discuss them in greater depth later.

**Date:** @March 14, 2023 9:40 PM

## **Topic: Implementation of Replication logs**

### **Recall**

How does leader-based replication work under the hood?

### **Notes**

#1 Statement-based replication.

#2 Write-ahead log (WAL) shipping: storage versions must be identical

#3 Logical(row-based) log replication: transaction log, CDC(change data capter) MySQL version greater than 5.1 binlog

#4 Trigger-based replication: greater overheads.

What's statement-based replication?  
And what are the problems?

Definition: the leader sends write statements to its followers.  
The followers execute the statement as if they had been received from a client.

Problems: NOW(), RAND() functions; execution order;



**SUMMARY:**  
**TODO**

**Date:** @March 15, 2023 9:01 PM

## **Topic: Problems with Replication lag**

### **Recall**

### **Notes**



Eventual consistency	The leader and followers may be inconsistent at a moment, however, as long as stop writing and wait a while, the followers will eventually catch up and become consistent with the leader. This effect is known as eventual consistency.
Replication lag	the delay between a write happening on the leader and being reflected on a follower
The approaches to solving inconsistencies?	<ol style="list-style-type: none"> <li>1. Reading your own writes</li> <li>2. Monotonic Reads</li> <li>3. Consistent Prefix Reads</li> </ol>



#### **SUMMARY:**

**In the read-scaling architecture, by improving the read throughput, we just create many followers and distribute the read requests across those followers. However, this approach only works with asynchronous replication as a single node failure would make the entire system unavailable for writing. Unfortunately, asynchronous replication probably leads to apparent inconsistencies in the database.**

**Date:** @March 15, 2023 9:12 PM

**Topic:** Reading Your Own Writes

#### **Recall**

Asynchronous replication inconsistency issue

#### **Notes**

Read-after-write consistency is not guaranteed!

How can we  
implement read-  
after-write  
consistency?

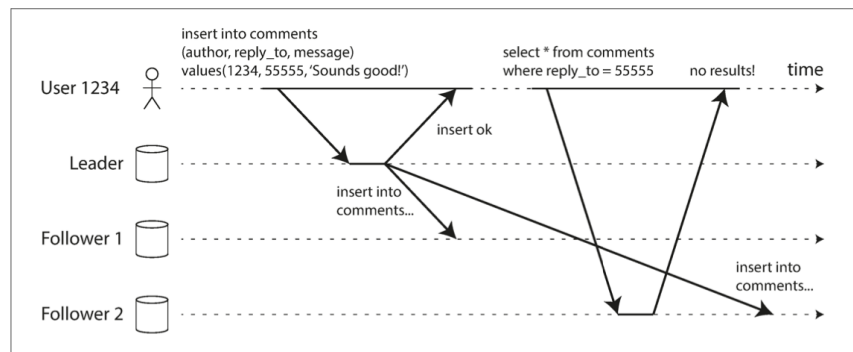


Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

### Relevant technicals

1. read it from the leader if the data possibly have been modified
2. Read from leader or followers based on the last write time.
3. Clients remember the most recent write timestamp



#### SUMMARY:

**read-after-write consistency, also known as read-your-own-writes consistency is a guarantee that the user will always see any updates they submitted if they reload the page.**  
*cross-device read-after-write consistency: the user should see identical data.*

**Date:** @March 15, 2023 9:36 PM

**Topic:** Monotonic Reads

**Recall**

**Notes**

## *moving backward in time* anomaly

How can  
implement  
Monotonic Reads  
consistency?

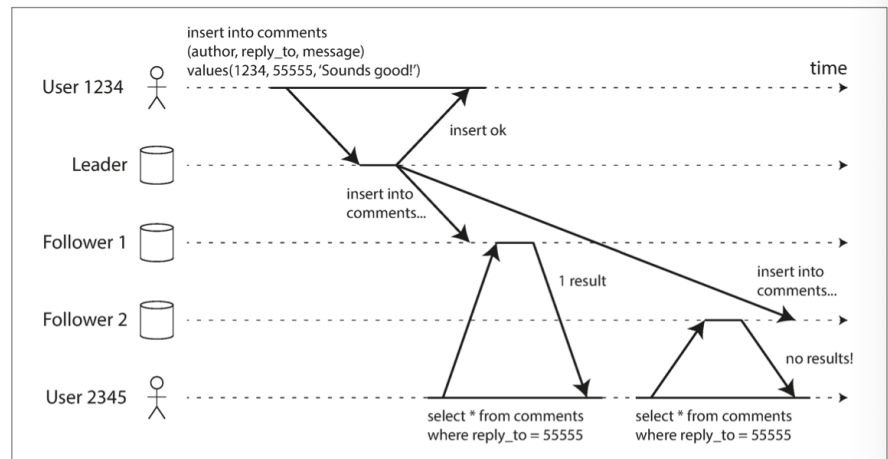


Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.

- #1. always make reads from the same replica
- #2 ... (not mentioned in the book)



### SUMMARY:

If the user makes several reads from different replicas, the user can see the response change randomly. This is known as *moving backward in time* anomaly.

By solving the moving backward in time anomaly, we have to guarantee Monotonic reads consistency. When you read data, you may see an old value; monotonic reads only means that if one user makes several reads in sequence, they will not see time go backward— i.e., they will not read older data after having previously read newer data

Date: @March 15, 2023 9:44 PM

Topic: Consistent Prefix Reads

Recall

violation of  
causality

## Notes

Expected:

*Mr. Poons*

How far into the future can you see, Mrs. Cake?

*Mrs. Cake*

About ten seconds usually, Mr. Poons.

**Got:** (Answer comes first violation to the causality)

*Mrs. Cake*

About ten seconds usually, Mr. Poons.

*Mr. Poons*

How far into the future can you see, Mrs. Cake?

The root  
cause of  
causality  
violation.

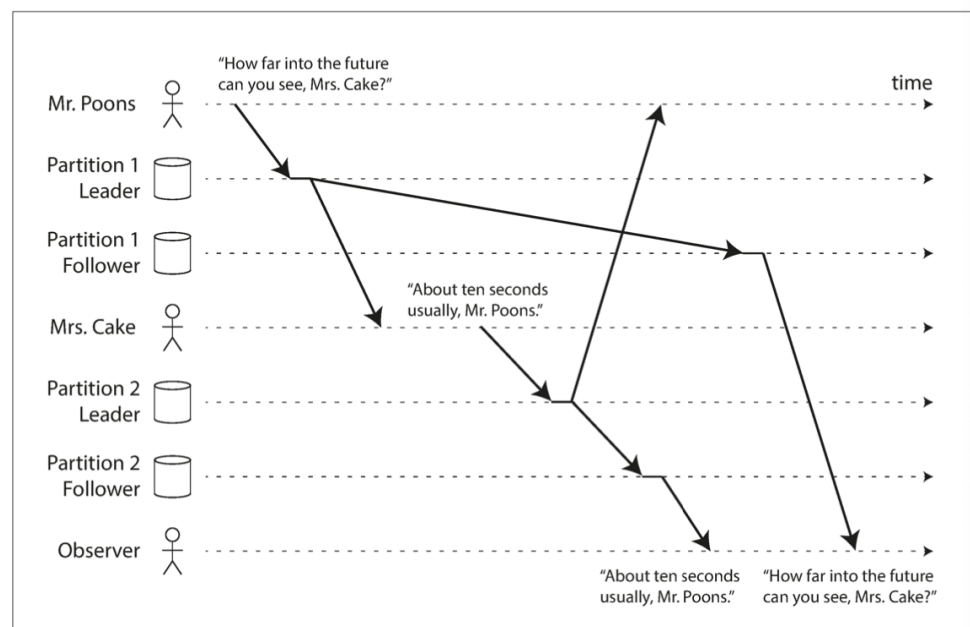


Figure 5-5. If some partitions are replicated slower than others, an observer may see the answer before they see the question.

**SUMMARY:**

With replication lag, the causality violation could occur. E.g. In a dialog, the answer comes first than question.

To avoid such violation, consistent prefix reads guarantee should be implemented.

This guarantee says that if a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order. (Reads in the same order with writes happened order)

---

**Date:** @November 5, 2019

**Topic:** Solutions for Replication Lag

**SUMMARY:**

When working with an eventually consistent system, it is worth thinking about how the application behaves if the replication lag increases to several minutes or even hours. If the answer is “no problem,” that’s great. However, if the result is a bad experience for users, it’s important to design the system to provide a stronger guarantee, such as read-after-write.

---

**Date:** @March 15, 2023 10:11 PM

**Topic:** Multi-Leader Replication

## Recall

Downsides of  
single leader

## Notes

- write is unavailable if network between client and leader is broken
- low efficiency with multiple data centers

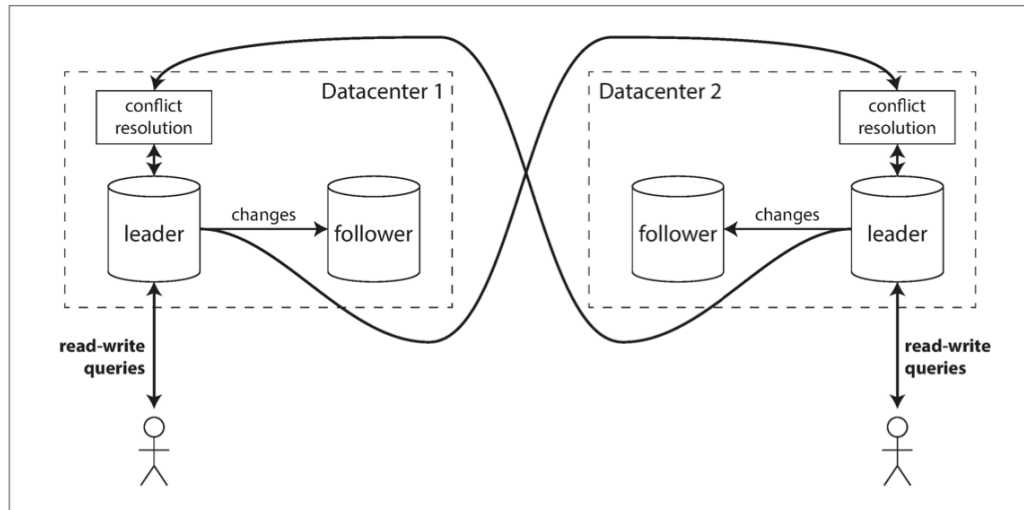
Comparison  
between single-  
leader and multi-  
leader  
configurations

1. performance:  $M > S$
2. tolerance of datacenter outages
3. tolerance of network problems:  $M > S$



## SUMMARY:

**Multi-leader replication across multiple datacenters.**



*Figure 5-6. Multi-leader replication across multiple datacenters.*

big downside of multi-leader replication is: write conflicts. The same data may be concurrently modified in two different datacenters, and those write conflicts must be resolved.