# Database Replication and Scalability

Prof. Dr. Peter Braun

17. Oktober 2024

## 1 Titlepage

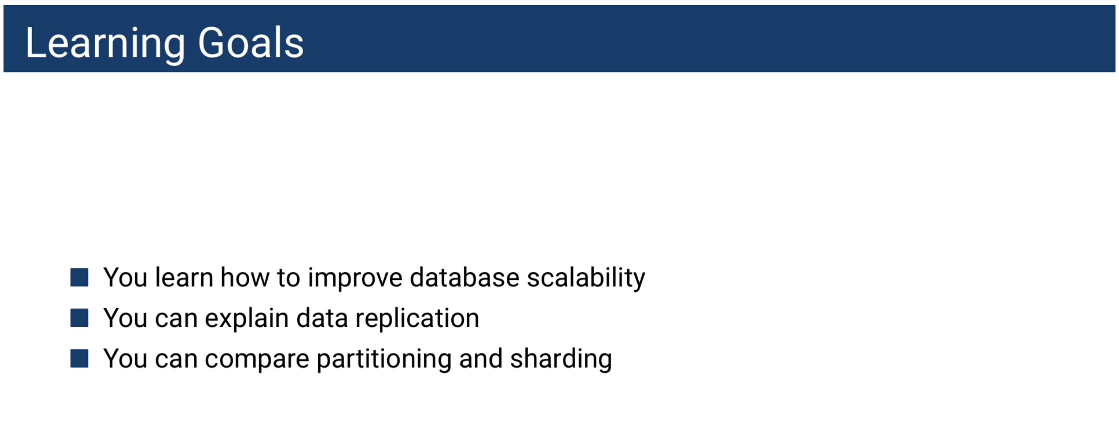
Database Scalability and Replication

Prof. Dr. Peter Braun

thws Technical University of Applied Sciences Würzburg-Schweinfurt

No Text – maybe some music :-)

## 2 Learning Goals



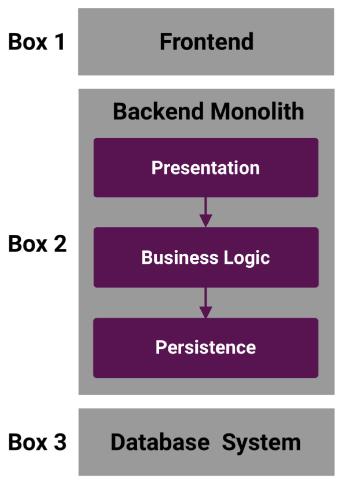
### Learning Goals

- You learn how to improve database scalability
- You can explain data replication
- You can compare partitioning and sharding

Welcome to this unit about database scalability and replication. Today, we will introduce the topic of scalability, which is very important for backend systems. Scalability is related to the problem of increasing the number of requests and how a system responds to this. If a system is not scalable, then there is a maximum number of requests that your backend can handle. More requests will slow down the processing time of requests or even cause the server to crash. Scalability is an issue in all parts of the backend system, and today, we start by discussing some techniques to make the database layer more scalable. Later in this course, we discuss other techniques like load balancing. We will also talk about database replication, which can be a means to improve scalability but is also related to reliability. And when we talk about scalability, we will discuss specific techniques like partitioning and sharding.

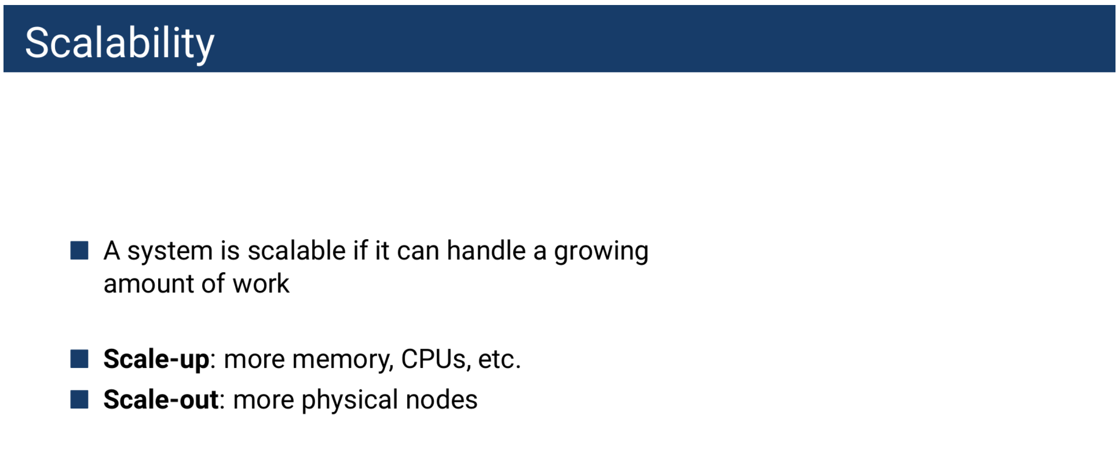
## 3 The Current Set-Up

### The Current Set-Up



Let's summarize the current state of a possible setup. Our system consists of three computers or boxes. Box 1 is the frontend, for example, with a Web browser. This computer belongs to the end-user. Box 2 is the backend that we are developing. We sketch a monolithic architecture here that consists of some presentation layer, for example, an API, the business logic layer with the domain logic, and the persistence layer that you know from the last unit. Finally, box 3 is the computer running the database system, for example, MySQL. Please note the differences between the current implementation we are working on, where we use an embedded database, H2. Let's now define the term "scalability."

## 4 Scalability



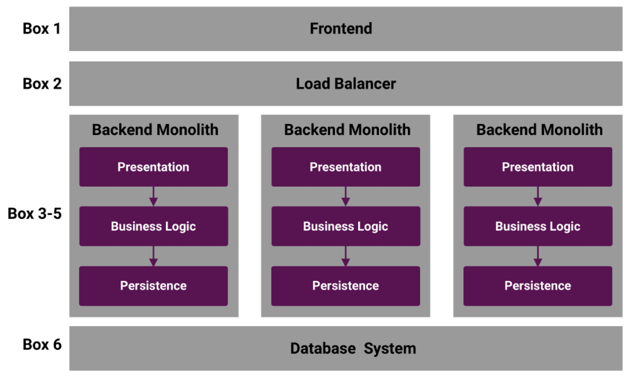
### Scalability

- A system is scalable if it can handle a growing amount of work
- **Scale-up**: more memory, CPUs, etc.
- **Scale-out**: more physical nodes

Regarding scalability, we refer to a system's ability to manage an increasing volume of work. There are two primary ways this scaling is achieved: scale-up and scale-out. When we reference 'scale-up,' we are talking about beefing up our hardware. Specifically, this can be adding more memory or upgrading the CPUs, among other aspects. This action empowers our system to cope with more data directly. So, the computer becomes more powerful, but it is still only a single computer. It's easy to understand that this approach is limited. There certainly is a point where you cannot add more memory or more processors to a single computer. On the other hand, 'scale-out' takes a different approach. Rather than focusing on enhancing existing components, scale-out increases the number of physical nodes. Essentially, we are increasing the number of computers to accommodate more requests. This involves a mechanism to distribute the workload. Both methods have advantages and contexts where they are the optimal choice. However, keep in mind that real-world systems do not scale indefinitely. There's always a limit to how much you can scale up a single machine, just as there's a limit to how effectively you can distribute work in a scale-out scenario.

## 5 Scale-out the Monolith

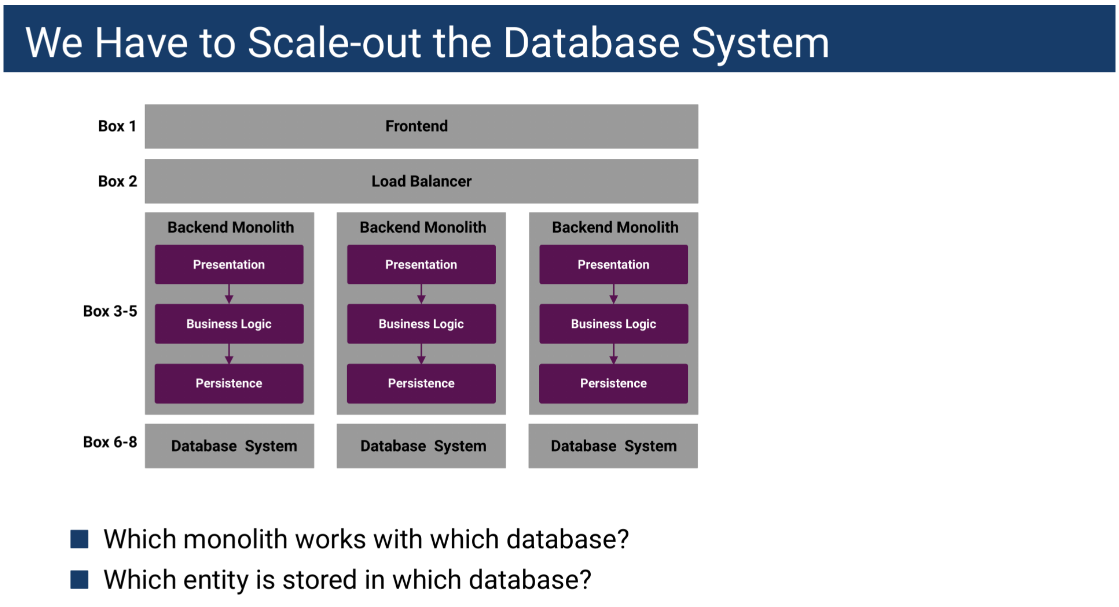
## Scale-out the Monolith



- Backend Monolith must be implemented stateless
- Database is the bottleneck and single point of failure

So, we have a monolithic software architecture. What can we do about scalability? Let's try the scale-out principle for the backend components. We increased the number of boxes; in the picture beside me, there are now three boxes for the backend, all running the same monolith. When we increase the number of boxes, there must be a component to distribute the requests, which is done by the load balancer located in box 2. Every single request that comes in from the frontend is forwarded to one of the three boxes in the middle. This simple approach can work well if the backend monolith is implemented statelessly. Stateless means that each instance of the backend does not store any specific data about the requests other than what is stored in the database. This way, all data is centralized in the database, and it doesn't matter which boxes process a request. In this approach, we still have a single database system on box 6. Beyond being a potential performance bottleneck, the database acts as a single point of failure. If the database server crashes, the whole system does not work anymore, threatening its reliability. Let's try to solve this first.

## 6 We have to Scale-out the Database System

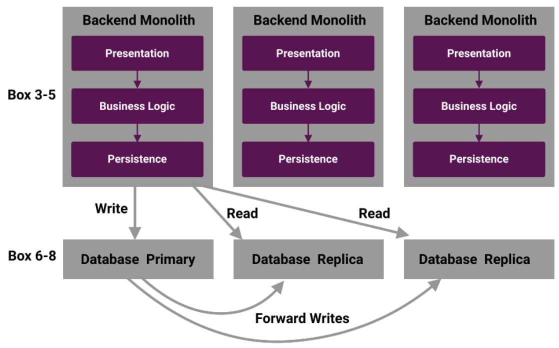

We Have to Scale-out the Database System

- Which monolith works with which database?
- Which entity is stored in which database?

Now, we scale out the database layer, too. The picture shows three boxes, all running a database system, but it could be less or more. The database cannot be implemented statelessly. Therefore, we have a challenge here. How do we distribute the data in the set of all database servers? Which backend server communicates to which database server? Which data is stored in which database?

## 7 Database Primary/Replica – The Basic Idea

### Database Primary/Replica – The Basic Idea



- Assumption: by far more read than write requests
- Split read and write requests to the database
- Only one server accepts write requests; all others only read requests

We now arrive at one of the fundamental concepts of database management: the idea of a Primary/Replica model in the context of Database Replication and Scalability. The prerequisite assumption for this model is that we have far more read queries than write operations, which is a fair assumption in many applications. Our goal is to divide the responsibilities of serving these requests, by splitting read and write requests between different database servers. The core is that only one server deemed the "Primary," accepts write requests. All other servers involved, called "Replicas,äre categorized as taking only read requests. They share the load of read requests, following a round-robin schema for better efficiency.

## 8 Database Primary/Replica – Implementation

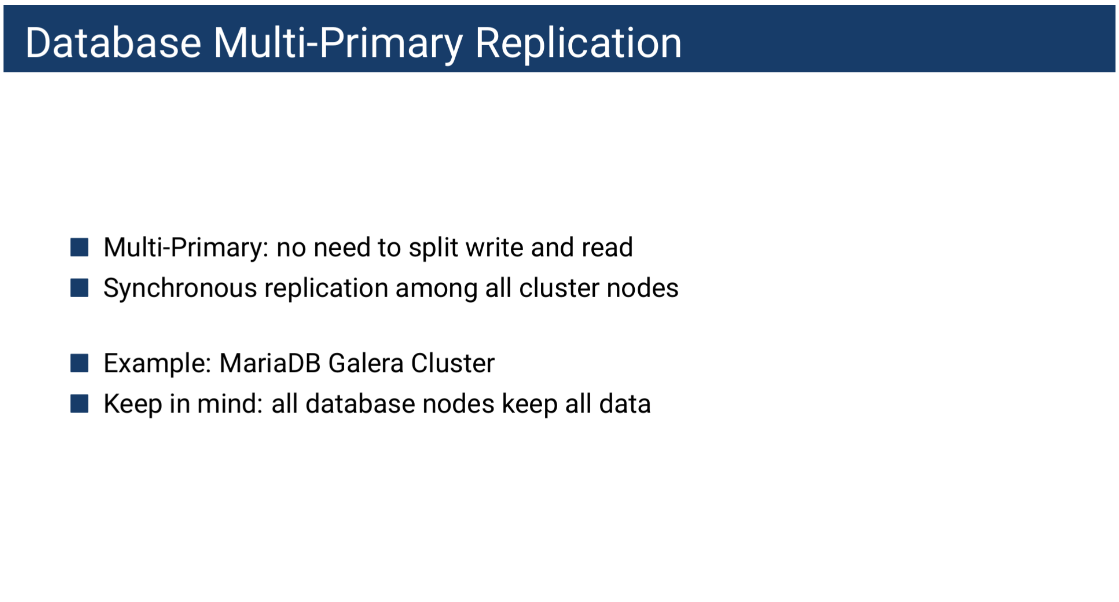
### Database Primary/Replica – Implementation

- All database servers keep all data
- Only the Primary accepts write operations

- Replicas are used in a round-robin schema
- Set up manually by using different entity managers
- Or use third-party frameworks (e.g., Spring)

You might question how differing operations across servers can ensure database consistency. That's where the next key point comes in. Despite the split in responsibilities, every database server ensures that it maintains an identical copy of all data. This must be considered a big disadvantage. If the disk space becomes too small for the database, we cannot solve this problem with this approach. Implementing this model can be undertaken manually by utilizing different entity managers in JPA or Hibernate or by using available third-party frameworks. For instance, Spring simplifies the process by seamlessly coordinating between the Primary and the Replicas. In summary, this architecture ensures scalability by parallelizing read and write operations and enhances our database system's overall performance. But, the approach is limited.

# 9 Database Multi-Primary Replication



## Database Multi-Primary Replication

- Multi-Primary: no need to split write and read
- Synchronous replication among all cluster nodes

- Example: MariaDB Galera Cluster
- Keep in mind: all database nodes keep all data

As an extension of the last approach, we now shortly discuss Multi-Primary Replication. Multi-primary replication holds a significant advantage, eliminating the need to split write and read operations. The implication is that any node in the cluster can handle read and write operations. Taking a closer look, we uncover an essential feature of Multi-Primary Replication—synchronous replication among all nodes in a cluster. This means that any change in data on one node is simultaneously reflected on all other nodes, ensuring high data integrity and consistency across the cluster. To give you a real-world application of this concept, consider MariaDB Galera Cluster, an open-source solution widely used for multi-primary replication. With the MariaDB Galera Cluster, database instances, called nodes, synchronize and keep each transaction in check for an up-to-the-minute reference point, ensuring all cluster nodes maintain the same data. Remember that we still have the challenge of having all data fit into one database.

## 10 Database Partitioning

### Database Partitioning

- A database is partitioned on the level of tables
- Each table is assigned to one specific node
- Each node is responsible for read and write

- SQL Joins are not possible with distributed tables
- Problem: if a table is too large for one node

We will now discuss techniques to scale out the database layer beyond the quite limited replication approach. The first technique is called database partitioning. This means each table in a database is assigned to a singular, specific node. But we keep all data of one table on the same node. Let's say we have a table for the entity "room," which is stored on the first node. Another table for the entity "booking" is stored on the second node. Every node is responsible for all writing and reading operations related to that table. This schema effectively distributes data across different nodes, adding to our database's overall performance and scalability. However, with this partitioning approach, it's crucial to stress that SQL joins become challenging. Why is this? Since the tables are no longer stored centrally but spread across various nodes, carrying out joins becomes complex, if not impossible. Another limitation is still size. A table with millions or billions of rows might become too large for a single node. In this situation, refining our partitioning strategy becomes essential. The next idea is table partitioning.

# 11 Table Partitioning

## Table Partitioning

- Tables are partitioned on the level of columns
- One table is split into many sub-tables
- The key must be re-used in all sub-tables

- Good if requests only use columns from one sub-table

Diving deeper into the concept of scalability within databases, let's discuss Table Partitioning. Table partitioning is based on a fairly straightforward idea: dividing a large table into small sub-tables or partitions based on the columns. Meaning that the segmentation logically categorizes data, focusing on the columns of the data tables. Let's say there is a very big table with information about people. We split this table and store the important data like first name, last name, and date of birth in a sub-table on one node, whereas other not-so-important data like place of birth, gender, and passport number on another node. The importance of data derives from the application and what data are requested together. Table Partitioning can improve performance, particularly when queries only use data from a single partition or sub-table. In this case, the system can focus on the specific sub-table. In contrast, if all data about a person is needed, all nodes must be requested, which will take longer. By the way, we still have the problem of a table or sub-table becoming too big in this case.

# 12 Sharding of Database Tables

## Sharding of Database Tables

- Partition tables by table rows
- Assign ranges of rows to shards
- Keys are used to assign rows to shards

- Write requests must be directed to one shard
- Read requests are sent to all shards in parallel

To effectively scale databases and manage large amounts of data, we can finally use an approach known as sharding. Sharding essentially means partitioning tables by table rows. Not tables or table columns but table rows. This entails dividing a large database into smaller, more manageable parts, which we call shards. One key benefit of this approach is that it can dramatically enhance the speed and performance of your database queries. Let me explain how we assign rows to these shards. In a sharded system, specific ranges of rows are assigned to each shard. This distribution is based on keys, which are used to map each row to its designated shard. These keys establish the framework within which the database tables can be divided and managed on a smaller scale. By carefully selecting these keys, one can create a balancing effect, ensuring that each shard holds roughly the same amount of data. Regarding writing requests, it's important to note that they will always be directed to one shard. This approach ensures data consistency and avoids concurrency issues or potential data corruption. In contrast to write requests, read requests are sent to all shards in parallel. This is because any shard could hold the data being requested. As a result of this design, it's possible to scale database performance linearly as more shards are added. To summarise, sharding database tables involves partitioning tables by table rows, assigning ranges of these rows to individual shards based on keys, directing all write requests to just one shard, and spreading out read requests to all shards simultaneously. This approach is very typical for the so-called NoSQL database systems.

## 13 Summary

### Summary

- Replication: prevent data loss and improve availability
- Partitioning: simple to realize and very useful
- Sharding: complex to implement but extremely fast

- Replication can be combined with partitioning and sharding

That's it for today. We've examined various strategies for enhancing database operation, focusing on replication, partitioning, and sharding. Let's summarize these core points. Replication is a protective strategy improving the availability of our database and acting as a safeguard against data loss. By maintaining identical copy of data on two or more databases, replication ensures continuous data availability even if one database fails. However, it's essential to consider the additional storage space required and the potential lag between updates on each replica. Next, partitioning. This approach to dividing the database into easily manageable segments is relatively simple to implement and highly beneficial. Exercises like backups and optimizations can then be performed on concerned partitions rather than the entire database, accelerating the task and reducing the overall database load. The third concept is sharding. This more complex process involves dividing the database into smaller sections known as shards. The main advantage of sharding lies in its speed and enhanced performance. Despite the complexity of implementation, the increased speed can be especially critical for larger databases. While each strategy has its benefits, it is possible and often beneficial to combine them. For instance, replicating a sharded or partitioned database can provide redundancy and increased accessibility alongside improved performance.