INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

isep

Estratégias de sharding de based de dados - Um guia orientado a domínio para construir sistemas escaláveis

CARLOS MANUEL FONTÃO PINHEIRO ALVES

Setembro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Database sharding strategies

A domain-oriented guide to building scalable systems

Carlos Manuel Fontão Pinheiro Alves

Dissertation to obtain the master’s degree in informatics engineering,

Specialization in Software Engineering

Advisor: Paulo Gandra de Sousa

Porto, 2024

Integrity declaration

Declare that this academic work is being carried out with integrity.

I have not plagiarised or applied any form of misuse of information or falsification of results throughout the process that led to its elaboration.

Therefore, the work presented in this document is original and my own and has not previously been used for any other purpose.

I further declare that I am fully aware of P.PORTO's Code of Ethical Conduct.

ISEP, Porto, 27 de setembro de 2025

Abstract

In the contemporary software landscape, the sheer surge in data generation underscores the necessity for scalable database architectures. Monolithic databases, while known for their robust consistency, often underperform when facing escalating workloads and growing user bases. Sharding—the partitioning of data across multiple nodes—has consequently become an essential architectural approach, facilitating horizontal scalability to meet these demands.

This dissertation offers an exploratory, tutorial-oriented analysis of sharding, with attention to its implementation in both relational and non-relational databases. Rather than focusing on empirical performance testing, the study consolidates insights from existing research and practice to explain how sharding strategies can be designed and applied in different domains. Using PostgreSQL and MongoDB as reference technologies, the work outlines how sharding is implemented, the trade-offs it introduces, and the factors influencing the choice of shard key.

The dissertation presents sharding as a group of design choices rather than just a technical mechanism. In doing so, it offers a structured framework intended for developers, architects, and researchers. It underscores the significance of aligning sharding approaches with the specific data access patterns of a given domain, while also recognizing the abstractions present in contemporary distributed database systems.

Resumo

No panorama contemporâneo do software, o elevado crescimento da geração de dados sublinha a necessidade de arquiteturas de bases de dados escaláveis. As bases de dados monolíticas, embora conhecidas pela sua consistência robusta, frequentemente apresentam desempenho inferior face a cargas de trabalho crescentes e ao aumento do número de utilizadores. O sharding — a partição de dados através de múltiplos nós — tornou-se, consequentemente, uma abordagem arquitetural essencial, facilitando a escalabilidade horizontal para responder a estas exigências.

Esta dissertação oferece uma análise exploratória, com orientação tutorial, sobre o sharding, com atenção à sua implementação em bases de dados relacionais e não relacionais. Em vez de se centrar em testes de desempenho empíricos, o estudo consolida os conhecimentos existentes da investigação e da prática para explicar como as estratégias de sharding podem ser desenhadas e aplicadas em diferentes domínios. Utilizando o PostgreSQL e o MongoDB como tecnologias de referência, o trabalho descreve como o sharding é implementado, os compromissos que introduz e os fatores que influenciam a escolha da shard key.

A dissertação apresenta o sharding como um conjunto de escolhas de design, e não apenas como um mecanismo técnico. Ao fazê-lo, oferece uma estrutura organizada destinada a desenvolvedores, arquitetos e investigadores. Sublinha a importância de alinhar as abordagens de sharding com os padrões de acesso a dados específicos de um determinado domínio, ao mesmo tempo que reconhece as abstrações presentes nos sistemas de bases de dados distribuídas contemporâneos.

Palavras-chave: Sharding de bases de dados; Bases de dados relacionais; Bases de dados não relacionais; Escalabilidade de dados; Bases de dados distribuídas.

Acknowledgements

I want to extend my heartfelt appreciation to my family, particularly my girlfriend, whose unwavering support has played an instrumental role in bringing me to where I am today. I'm also deeply grateful to my cherished pets, whose companionship provided me with much-needed doses of endorphins.

I owe a debt of gratitude to my esteemed teachers who guided me throughout this dissertation. Their wisdom and mentorship have been invaluable.

A special note of thanks goes to my dear friend Pedro Novo, who not only introduced me to the concept of database sharding but also played a pivotal role in helping me choose the topic for this dissertation.

Table of Contents

1 Introduction 2
1.1 Problem 2
1.2 Context 2
1.2.1 Data volume and traditional databases challenges 3
1.2.2 Traditional databases challenges 5
1.2.2.1 Clustering 5
1.2.2.2 Partitioning 5
1.2.2.3 Limitation of traditional database strategies 6
1.2.2.1 The need for sharding 6
1.3 Objectives 7
1.4 Systematic literature review 9
1.4.1 Methodology of SLR 9
1.5 Design science research methodology 10
1.6 Ethical Analysis 11
1.6.1 Societal Impacts 11
1.6.2 Ethical responsibility 11
1.6.3 Legal and Social Responsibility 12
1.6.4 Acknowledgment of AI and writing assistance 12
1.7 Document structure 13
2 State of the art 16
2.1 Relevant papers and research methodology 16
2.2 Database sharding 18
2.3 Sharding strategies 20
2.3.1 Range Sharding 20
2.3.2 Hash Sharding 22
2.3.3 Directory based Sharding 22
2.3.4 Round-Robin Sharding 24
2.4 Alternatives to sharding 24
2.4.1 Vertical scaling 24
2.4.2 Horizontal scaling 24
2.5 When does sharding become necessary 25

2.6 Existing technologies 25
2.6.1 Sharding on MongoDB 25
2.6.2 Sharding on AWS Aurora 26
2.6.3 Sharding on PostgreSQL 27
2.7 Case studies 28
2.7.1 Database Sharding with MongoDB 28
2.7.1.1 Krusche & Company 28
2.7.1.2 Foursquare 29
2.7.2 Database Sharding on Amazon Aurora 30
2.8 Impact of sharding on query complexity 31
2.8.1 Performance comparison: Traditional vs Sharded Database 32
2.8.2 Database partitioning for blockchain systems 33
2.9 Technologies 34
2.9.1 Database: PostgreSQL & MongoDB 34
2.9.2 Testing 35
2.10 Summary 36
3 Domain role in sharding strategy 38
3.1 Domain influences sharding decision 38
3.1.1 Data access patterns are domain specific 38
3.1.2 Business isolation requirements 39
3.1.3 Regulatory and compliance concerns 39
3.1.4 Query Optimizations 39
3.2 Domain does not influence sharding decision 40
3.3 E-commerce application case study 40
3.3.1 Domain overview 40
3.4 Sharding key selection and architectural implications 42
3.4.1 Distribution of data and workload 42
3.4.2 Query routing 42
3.4.3 Cross-shard operations 42
3.4.4 Scaling and rebalancing 43
3.4.5 Fault isolation 43
3.4.6 Conclusion 43
3.5 Challenges of defining shard key 43
3.5.1 Relational databases 44
3.5.2 Non-Relational databases 45
3.5.3 Summary 45

3.5.4 Range sharding and the problem of recent data concentration 46
3.6 Geo sharding 47
3.7 Summary 48
4 Implementation setup: Relational vs non-Relational 50
4.1 Sharding in Relational Databases 50
4.1.1 Manual sharding overview 50
4.1.2 Setup steps (PostgresSQL) 51
4.1.3 Alternative: Middleware 52
4.2 Sharding in Non-Relational Databases 52
4.2.1 Setup steps (MongoDB) 52
4.2.1.1 Range sharding 52
4.2.1.2 Hash sharding 53
4.2.1.3 Operational considerations 53
4.3 Comparative overview 54
5 Achieving scalability through sharding in an e-commerce application 56
5.1.1 Relational Database: PostgreSQL with Citus Extension 56
5.1.1.1 Sharding approach 56
5.1.1.2 Practical example 57
5.1.2 Non-Relational Database: MongoDB 57
5.1.2.1 Sharding Approach 58
5.1.2.2 Practical example 58
5.1.3 Simulation constraints and hypothetical scaling scenario 58
5.1.4 Routing between shards 59
5.1.4.1 Relational databases 59
5.1.4.2 Non-Relational databases 60
6 Demonstration of range-based sharding 62
6.1.1 Create directories 62
6.1.2 Start the config server servers 62
6.1.3 Initialize replica sets 63
6.1.4 Start the router and connect to the cluster 64
6.1.5 Add shards to cluster 64
6.1.6 Enable and setting sharding 65
6.1.7 Insert data and testing 66
7 Conclusion 68
7.1 Next steps 69

List of Figures

Figure 1 - Volume of data created, consumed or stored from 2010-2023 (Statista, 2024) 4
Figure 2 - Number of results per search criteria 18
Figure 3 - Sharded Database Overview (Oracle, 2024) 19
Figure 4 - Range-based sharding (Bailey, 2024) 21
Figure 5 - Directory-based sharding (Guevara, 2024) 23
Figure 6 - Sharding in PostgreSQL (PostgreSQL, 2024) 60
Figure 7 - MongoDB sharding (MongoDB, 2025b) 61
Figure 8 - Create directories for mongo sharding 62
Figure 9 - Starting config server 62
Figure 10 – Initializing config replica set 63
Figure 11 – Initializing replica 1 63
Figure 12 - Initializing replica 2 63
Figure 13 - Start mongos router 64
Figure 14 - Add shard 1 64
Figure 15 - Add shard 2 64
Figure 16 - Enable sharding on ecommerceDB 65
Figure 17 - Creating collection and defining sharding 65
Figure 18 - Create sharding range 66
Figure 19 - Inserting data 66
Figure 20 - Sharding distribution 66
Figure 21 - Database status 67

List of Tables

Table 1 - Conditions for inclusion/exclusion criteria 17
Table 2 - Sharding strategy alternatives 41
Table 3 - Relational vs non-relational characteristics 45
Table 4 - Database systems aspect comparision 54

List of Acronyms

AWS	Amazon Web Services
DRSM	Design Science Research Methodology
GQM	Goal Question Metric
IoT	Internet of things
IS	Information Systems
SLR	Systematic Literature Review
RBDM	Relational Database Management Systems
FDW	Foreign Data Wrapper

1 Introduction

This section aims to present the context, problem, and objectives associated with the development of this dissertation, as well as display the planning and the overall document structure.

1.1 Problem

As datasets increase in size, monolithic databases—whether SQL or NoSQL—often buckle under the strain, sacrificing performance and scalability. To tackle this, engineers have turned to database sharding, splitting data into smaller, manageable chunks spread across multiple servers. However, no single sharding tactic works universally. Success hinges on a trifecta of factors—the quirks of your workload, the type of database and the strategy chosen to split the data.

This research will primarily focus on theoretical exploration, comparing various database sharding mechanisms in both relational and non-relational databases to understand their impact on scalability, performance, and resource usages. To complement the theoretical findings, a practical test will be conducted using a representable dataset in a limited scope, offering controlled insights into the real-world implications of different strategies.

The goal is to emphasize the theoretical underpinnings, analyzing case studies, architectural principles, and existing literature to provide a robust understanding of the strengths and limitations of different sharding strategies.

1.2 Context

This section explores the challenge of database scalability in the face of ever-increasing data volumes and application demands. Traditional databases, particularly relational systems, often struggle with performance bottlenecks, such as high query latency and limited horizontal scalability.

1.2.1 Data volume and traditional databases challenges

Scalability has always been a fundamental aspect of modern business infrastructure, enabling organizations to grow and adapt to evolving market demands. Efficient data scalability ensures that systems can handle increasing volumes of information while maintaining optimal performance, responsiveness, and usability (Bondi, 2000).

The global volume of data generation, replication, and consumption is projected to grow exponentially, with forecasts estimating it will surpass 394 zettabytes by 2028 (Statista, 2024). This dramatic surge reflects the unprecedented growth seen in 2020 when global data creation soared due to the increased adoption of remote work, online learning, and digital entertainment during the COVID-19 pandemic. This acceleration in data production underlines the growing reliance on digital solutions in everyday life.

To provide a clearer understanding, Figure 1 illustrates the exponential growth of global data over the past 14 years, alongside projections through 2028. The data is provided by the aforementioned “Statista” website and is measured in zettabytes.

Zettabyte is a unit that represents 2^70 bytes, equivalent to one sextillion bytes. To contextualize this immense scale, one zettabyte equals 1,000 exabytes or one billion terabytes, making it an apt measure for the vast data volumes generated today (TechTarget, 2022).

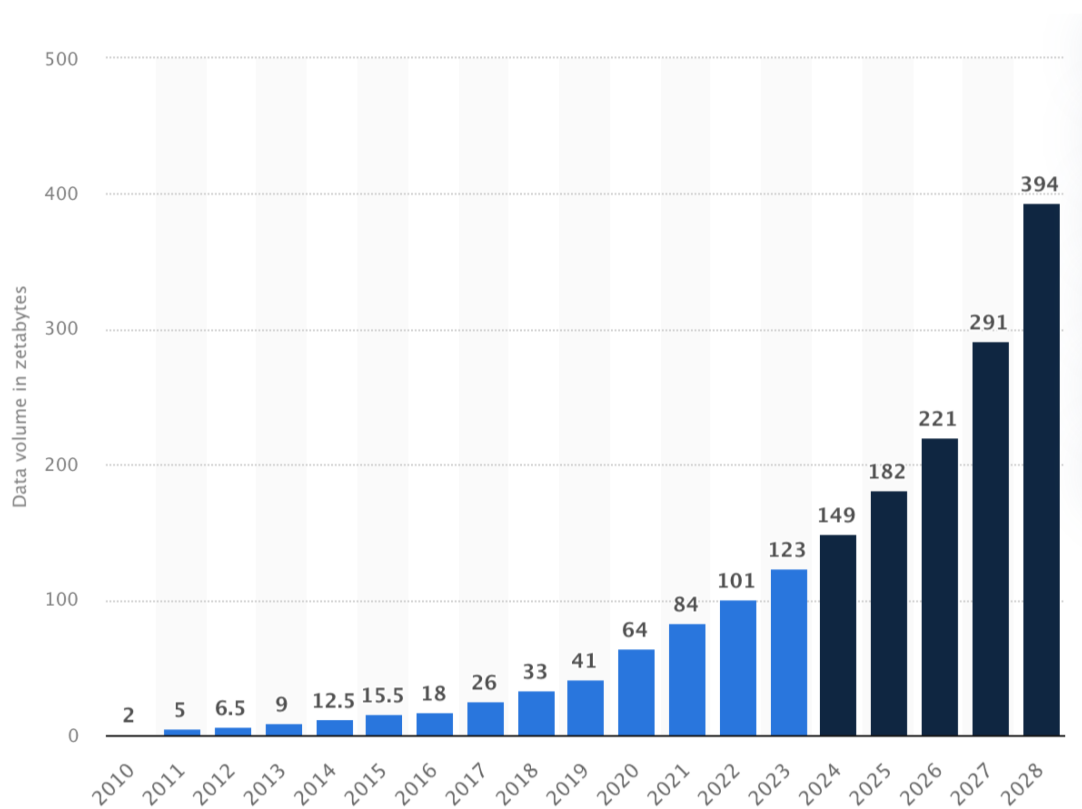

Figure 1 - Volume of data created, consumed or stored from 2010-2023 (Statista, 2024)

Achieving a level of scalability to support this volume of data, however, presents challenges. As datasets expand, it becomes increasingly complex to manage growth without compromising speed or efficiency. To scale effectively, applications must be designed to accommodate larger data volumes seamlessly, ensuring that performance remains consistent, and user experience is not degraded.

According to (Hiren Dhaduk, 2022) the main challenges of database scalability are the following:

- **Complexities with Data**: Managing data across multiple locations and formats creates difficulties in ensuring consistent access and analysis. Variability in storage structures complicates integration and retrieval, often requiring specialized tools or strategies to unify access.
- **Improper Traffic Distribution**: Uneven traffic allocation can overload certain servers or regions while underutilizing others. This imbalance leads to performance bottlenecks and potential system failures in high-demand scenarios.
- **Query Performance Issues**: As data grows, complex queries execute slower, especially in unoptimized databases. High query loads will strain resources, affecting real-time responses and user experiences.

- **Slow Content Loading**: Initial content rendering delays, caused by synchronous requests or resource-heavy operations, result in slower application performance, wasting bandwidth and reducing efficiency during inactive periods.
- **Integration Risks**: Incorporating new datasets into existing systems can lead to inconsistencies in data quality. Without proper alignment, merged datasets may produce inaccurate insights, affecting system reliability.

(RisingWave, 2024) adds to this list the **availability and reliability** challenges, highlighting the difficulties of successfully implementing redundancy and failover mechanisms.

1.2.2 Traditional databases challenges

Traditional databases, often built on monolithic architectures, have employed scaling strategies such as clustering and partitioning to address the challenges of increasing data volumes.
Some of the common strategies to scale are presented in this section.

1.2.2.1 Clustering

Database clustering is a widely known approach in traditional database systems to enhance scalability and fault tolerance. A cluster consists of multiple database servers (nodes) that work together to present a unified interface to the application layer. Clustering provides horizontal scaling by distributing the workload across multiple nodes, thereby improving the system's ability to handle higher query loads and ensuring availability even in the event of a node failure (Dgraph Labs, 2024).

1.2.2.2 Partitioning

In this approach, the database is divided into smaller, independent segments, or partitions, based on predefined criteria, such as ranges of values, hash functions, or list-based keys. Each partition operates as a subset of the larger database, which allows for parallel processing and localized query execution (Timescale, 2025).

Partitioning is particularly effective for improving query performance by narrowing the search space to a specific partition, rather than scanning the entire database.

However, partitioning also has its drawbacks. Managing distributed queries across partitions can introduce complexity, especially when queries span multiple partitions (cross-partition queries).

1.2.2.3 Limitation of traditional database strategies

While clustering and partitioning represent significant advancements in traditional database architectures, they face inherent limitations when scaling to hyperscale workloads (Pavlo et al., 2012):

- **Consistency Challenges**: Maintaining strong consistency across clusters or partitions often results in significant performance trade-offs, especially for write-heavy applications.
- **Single Points of Failure**: Clustering configurations with a single master node for writes can become bottlenecks, undermining fault tolerance.
- **Complexity of Management**: Both clustering and partitioning require complex configuration, monitoring, and maintenance, which increases operational overhead.
- **Data Imbalance**: Uneven data distribution in partitions or unoptimized workload distribution in clusters can lead to suboptimal performance.

1.2.2.1 The need for sharding

These limitations accentuate the need for more sofisticated scaling strategies, such as sharding, which extends the principles of partitioning to an entirely distributed environment. Unlike clustering and traditional partitioning, sharding emphasizes decentralized control and independent scalability, making it a preferred choice for hyperscale applications (Pavlo et al., 2012).

Historically, a single database instance was sufficient to manage the data requirements of most applications, as the volume and velocity of data generation were relatively modest. However, the rapid proliferation of Internet of Things (IoT) devices, social media platforms, and other data-intensive applications has significantly increased the amount of data generated by individual systems. For example, IoT devices alone are projected to produce over 73.1 zettabytes of data annually by 2025 (Statista, 2024).This vast influx of data often exceeds the storage and processing capacity of a single database server, which typically caps out at a few terabytes of data before encountering significant performance bottlenecks.

One illustrative case is social media platforms like Twitter and Facebook. In 2023, Twitter reported handling approximately 500 million tweets per day, while Facebook users generated 4 petabytes of data daily, including posts, photos, and interactions (Statista, 2024) .Such volumes are far beyond the capacity of traditional database solutions to store and query efficiently.

By implementing sharding, these applications can ensure consistent performance even as data volumes grow exponentially. We will expand on this topic later in this document.

1.3 Objectives

The primary objective of this research is to explore and critically evaluate strategic scalability decisions, with a central focus on comparing various database sharding mechanisms to optimize data scaling in both relational and non-relational databases. While the study emphasizes theoretical analysis, it incorporates a controlled practical component to validate key concepts. These strategic choices are essential for enhancing database performance, efficiency, and scalability, particularly in environments with increasing data volumes and complex system requirements.

The goal is to identify scenarios where specific sharding mechanisms deliver optimal results, balancing performance improvements with the complexity and maintainability of implementation.

To ensure these objectives are met with clarity and precision, the Goal-Question-Metric (GQM) approach serves as the guiding framework for this research. Goal Question Metric (GQM) operates on the premise that purposeful measurement in an organization begins with clearly defined goals. These goals must then be linked to specific data that measure their operational success, providing a systematic framework for interpreting the resulting data (Basili et al., 1992).
Through the application of the GQM approach, a tailored measurement system is created, targeting specific challenges and establishing guidelines for analyzing measurement outcomes. This model is structured into three key levels:

- **Conceptual level (GOAL)**: A goal is defined for an object, for a variety of reasons, concerning various models of quality, from various points of view, relative to a particular environment.

- **Operational level (QUESTION)**: A set of questions is used to characterize the way the assessment/achievement of a specific goal is going to be performed based on some characterizing model. Questions try to characterize the object of measurement (product, process, resource) concerning a selected quality issue and to determine its quality from the selected viewpoint.
- **Quantitative level (METRIC)**: A set of data is associated with every question to answer it quantitatively. This data can be objective or subjective.

Following this logic, the three most important questions to be answered during this dissertation are:

1. Goal: Understanding the role of sharding mechanisms on database scalability

- *Question:* How do different database sharding strategies conceptually improve scalability in large-scale systems?
- *Metric:* Analyze findings from prior research, case studies, case reports, and published frameworks, focusing on scalability indicators like data distribution patterns and system adaptability.
- *Reasoning:* This goal establishes a theoretical foundation by consolidating knowledge on how sharding mechanisms address the inherent limitations of monolithic database systems.

2. Goal: Providing a structured overview of sharding strategies and their practical implications

- *Question:* What are the main sharding strategies used in relational and non-relational databases, and how can they be effectively applied in different business domains?
- *Metric:* Examine academic literature, technical documentation, and industry practices to synthesize a tutorial-style overview of common strategies (e.g., range and hash sharding).
- *Reasoning:* This goal aims to create a guide that helps practitioners and researchers understand the strengths and limitations of each strategy without requiring hands-on experimentation.

3. Goal: Assess the trade-offs and challenges of different sharding mechanisms

- *Question:* What are the conceptual trade-offs of adopting sharding in terms of complexity, maintenance, and performance limitations?
- *Metric:* Identify challenges such as cross-shard joins, foreign key constraints, re-sharding, and operational overhead, and discuss how these impact long-term system design.

- *Reasoning:* By mapping out trade-offs, this goal provides guidance for informed architectural decision-making, even in the absence of direct implementation or testing.

The Goal Question Metric approach combines most of the current approaches to measurement and generalizes them to incorporate processes and resources as well as products. This makes it adaptable to different environments, as confirmed by the fact that has been applied in several organizations, such as NASA, Hewlett Packard, Motorola, Coopers & Lybrand etc (Basili et al., 1992).

1.4 Systematic literature review

A Systematic Literature Review (SLR) is a rigorous and structured approach used in research to identify, evaluate, and synthesize all relevant studies on a specific question or topic area (Kitchenham & Charters, 2007). The primary aim of an SLR is to provide a comprehensive and unbiased synthesis of research findings, ensuring that the review accurately reflects the existing ((Petticrew & Roberts, 2006).

1.4.1 Methodology of SLR

The methodology of a Systematic Literature Review (SLR) involves several key steps, including (Gough et al., 2017):

1. **Definition of Research Goals**: The methodology begins with the definition of specific research goals. In this work, we defined them using the Goal-Question-Metric (GQM) methodology. In this work, three distinct research goals were established to guide the systematic literature review.

2. **Development of Review Protocol**: A comprehensive review protocol is formulated to outline the scope and criteria for the literature review. This ensures that the review process remains structured and focused on the desired topic.

3. **Systematic Literature Search**: A rigorous and systematic search of relevant literature is conducted across academic libraries. This is crucial for identifying a wide range of sources related to the research goals.

4. **Selection of Studies**: Studies are selected based on the criteria, which are aligned with the research goals. This selection process helps in including only the most relevant and high-quality sources in the review.

5. **Assessment of Study Quality**: The quality of included studies is validated to determine their validity, ensuring that the review includes studies that meet certain standards of quality and relevance.

6. **Data Extraction and Analysis**: Data relevant to the research goals is extracted from the selected studies. This data is then meticulously analysed to draw meaningful insights and conclusions from the literature.

7. **Reporting Findings**: Finally, the findings of the systematic literature review are reported. This includes summarizing the key insights, synthesizing the information, and presenting it in a structured and transparent manner.

This meticulous and transparent process allows for the reproducibility and verification of the review's findings, ensuring the reliability and rigour of the research methodology (Gough et al., 2017).

1.5 Design science research methodology

The Design Science Research Methodology (DSRM) provides a framework for developing and evaluating innovative solutions to address complex challenges, particularly in Information Systems (Hevner et al., 2004). It emphasizes iterative cycles of designing, building, and evaluating artefacts. (Peffers et al. (2007) further refined DSRM into a structured methodology, outlining steps such as identifying problems, setting objectives, designing artefacts, and validating outcomes.

For this dissertation, DSRM principles were combined with a Systematic Literature Review (SLR) and the Goal-Question-Metric (GQM) framework to address challenges in scaling monolithic database architectures, with a specific focus on database sharding. The research process began by defining the problem scope and objectives, formulating research questions, and identifying metric-driven goals using the GQM approach. A systematic review of literature then provided a foundation for synthesizing theoretical insights and identifying key strategies for sharding.

This blended methodology offers a rigorous and transparent approach, ensuring reproducible findings that contribute both to the theoretical understanding of sharding mechanisms and practical solutions for scalability in modern software architectures.

1.6 Ethical Analysis

In this section, we explore the ethical considerations associated with scaling databases through sharding. The analysis is structured around three perspectives, each highlighting critical aspects of responsible database management. The goal is to identify practices that ensure ethical, sustainable, and compliant database operations while addressing growing data volumes and user demands.

1.6.1 Societal Impacts

Database sharding introduces societal challenges that extend beyond its technical implications. Key concerns include increased energy and resource consumption, which will significantly impact environmental sustainability. For instance, the distribution of data across multiple servers and shards may lead to higher energy demands and an expanded carbon footprint. To mitigate this, it is crucial to prioritize strategies that optimize resource efficiency, hence balancing scalability with environmental responsibility.

1.6.2 Ethical responsibility

Failing to address the scalability issues effectively can result in significant ethical consequences. Systems that are unable to scale adequately may experience degraded performance, higher operational costs, and reduced user satisfaction due to latency or downtime. From an ethical perspective, this can be seen as neglecting the organization’s obligation to provide reliable and efficient services to users. Developers and organizations share the responsibility to design scalable systems that align with user needs, ensuring high performance and reliability. Such efforts will not only foster trust and enhance user experience but also protect the organization’s reputation and competitiveness in the market.

1.6.3 Legal and Social Responsibility

The implementation of sharding mechanisms must comply with legal and ethical standards, particularly regarding data privacy and security. Sharding inherently involves the distribution of data across multiple servers, which can complicate data management practices and introduce new risks. It is essential to ensure compliance with regulations such as the General Data Protection Regulation (GDPR) by maintaining data security, transparency, and adherence to regional data residency laws. Strategies such as encryption, data minimization, and robust access controls are critical to safeguarding sensitive information. By addressing these legal and ethical obligations, organizations can maintain user trust and uphold their responsibilities to both regulators and society.

1.6.4 Acknowledgment of AI and writing assistance

During the development of this dissertation, AI-based language tools such as Grammarly and similar technologies chatbots, were utilized to enhance grammatical accuracy and improve sentence structure. These tools assisted in refining the readability and coherence of the text while preserving the integrity and originality of the research content. The ethical use of such technologies aligns with academic standards by supporting rigorous communication of ideas without compromising intellectual authorship. All critical analysis, arguments, and conclusions remain the independent work of the author.

1.7 Document structure

To ease the reading of this document, below is the list of chapters accompanied by an explanation of their contents:

- **Introduction (Chapter 1)** – Establishes the research problem, contextual background, and objectives. It also presents the methodology used, including the systematic literature review and design science approach, and addresses ethical considerations relevant to the study.
- **State of the Art (Chapter 2)** – Reviews existing research and academic contributions on database scalability and sharding. It introduces fundamental concepts, compares different sharding strategies, discusses alternatives to sharding, and highlights real-world case studies and technological implementations in relational and non-relational systems.
- **Design and Strategy (Chapter 3)** – Describes the conceptual design of the study and outlines the selected technologies (PostgreSQL, MongoDB, and supporting frameworks). It also presents the base e-commerce model and the reference architecture that will be used to illustrate sharding approaches.
- **Domain Influence in Sharding Strategy (Chapter 4)** – Explores the extent to which application domains influence sharding decisions. It discusses scenarios where domain-specific requirements strongly guide the choice of shard key, as well as situations where domain independence allows greater flexibility. This chapter also examines key considerations such as query optimization, business isolation, compliance, and challenges in shard key definition, supported by an e-commerce case study.
- **Implementation Setup: Relational vs. Non-Relational (Chapter 5)** – Provides a tutorial-style walkthrough of how sharding is implemented in both relational (PostgreSQL with Citus) and non-relational (MongoDB) environments. The chapter includes setup procedures, practical examples, and comparative analysis of range and hash-based strategies. It also discusses simulation constraints, scalability projections, routing mechanisms, and an honorable mention of geo-sharding as an alternative strategy.
- **Demonstration (Chapter 6)** - Presents the practical demonstration of the sharding approaches discussed in the previous chapters. It shows real execution of sharded database operations using the e-commerce model in a simplified scenario in a local environment.

- **Conclusion (Chapter 7)** – Summarizes the findings of the dissertation, revisits the research goals and questions, and reflects on the trade-offs, opportunities, and challenges of adopting sharding in modern database systems. It also elaborates on next steps that would follow the work and research of this dissertation.

2 State of the art

This section aims to explore studies and papers that have been published regarding the dissertation's aspects previously contextualized and to analyze and compare their aspects. Then, an overall comparison of these studies is conducted, followed by a conclusion on these, based on the presented information, on how this dissertation positions itself compared to the current investigation. Finally, a description of some existing technologies and commonly applied concepts is presented.

2.1 Relevant papers and research methodology

This section delves into real-world examples in the world of software scalability. These studies provide practical insights into how various companies have addressed the challenges of database scalability using sharding. To learn the evolution of the theme at hand, Science Direct and Institute of Electrical and Electronics Engineers (IEEE) Xplore were surveyed about the number of studies regarding this theme through the IEEE Xplore and Science Direct digital libraries. The research was made following the Design Science Research Methodology (DSRM) described in the previous section of this work.

Grey literature was used in this dissertation as a supplementary source of information, complementing the academic and peer-reviewed sources. While the core foundation of the research rested on academic rigor, official manufacturer websites like Amazon, MongoDB and others' official websites provided essential real-time data and technical documentation, adding depth to the study. Web articles from grey literature sources were utilized sparingly and as a last resort to offer diverse perspectives and insights.

The literature review method applied in this study adopts a hybrid approach that integrates elements from the Systematic Literature Review (SLR) methodology and, subsequently, well-defined research questions are formulated, along with the establishment of metric goals, drawing inspiration from the Goal-Question-Metric (GQM) methodology principles. Section 1.4 of this document comprehensively documents the GQM framework and the chosen research objectives.

Lastly, a set of criteria specifically tailored to the study's context and objectives, is devised. These criteria serve as guiding principles for the systematic retrieval of pertinent literature from reputable sources such as IEEE Xplore and Science Direct. The ensuing process involves thorough data collection and comprehensive analysis.

The conditions used as criteria of inclusion/exclusion are presented in Table 1.

Table 1 - Conditions for inclusion/exclusion criteria

Criteria	
Information Topic	Database scaling; Database sharding; Relational databases; Non-relational databases; Database migration
Type of Information	Academic Journals; Conference Materials; Magazines; Trade Publications; Reports. Review Articles; Research Articles; Web Whitepapers; Web pages/articles
Publication Date	Between 2010 and 2025
Language	English
Type of Review	Peer review
Type of Access	Open access

In Figure 2 we can see the number of results with the defined criteria.

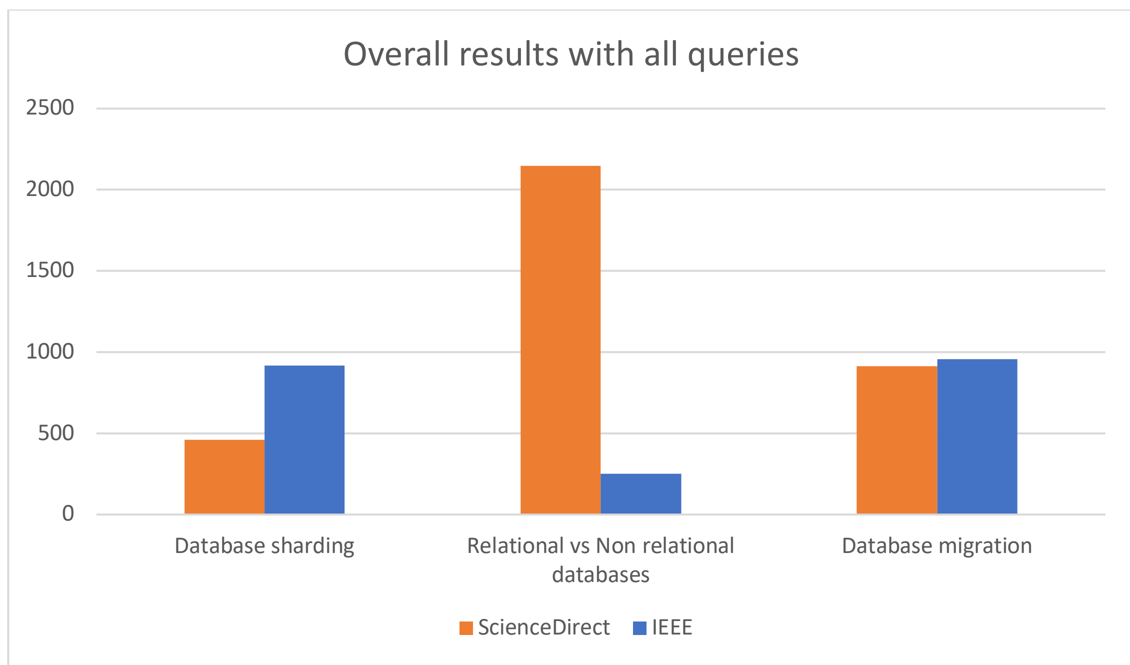

Figure 2 - Number of results per search criteria

Even though there are a few hundred articles containing the defined keywords, roughly only a couple dozen which contained pertinent information were selected and used as a source of information for this dissertation. They were mainly used in sections 2.2, 2.3 and 2.6 of this document, while the information used in section 2.7 was mainly extracted from official product websites, and other web articles containing more practical and up to date information about the topic.

2.2 Database sharding

This section presents the important aspects inherent to this dissertation, such as different database sharding mechanisms, which database management systems support it and the impact of sharding on database performance and scalability.

Database Sharding, as described by (Amazon Web Services, 2023), is a technique used to distribute a large database across multiple servers. Since a single server has limitations on the amount of data it can handle and process, sharding divides the database into smaller, more manageable units called shards. These shards are stored on separate database servers that work collaboratively to manage and process large datasets.

While sharding is often associated with database partitioning, the two concepts differ in scope and implementation. Partitioning refers to dividing a database into smaller subsets of data (partitions) based on specific criteria, such as ranges, lists, or hash values, which can still reside on

a single server or multiple servers, which helps optimize query performance and organization (PlanetScale, 2023).

An example of how (Oracle, 2024) would illustrate database sharding is presented in Figure 3 below.

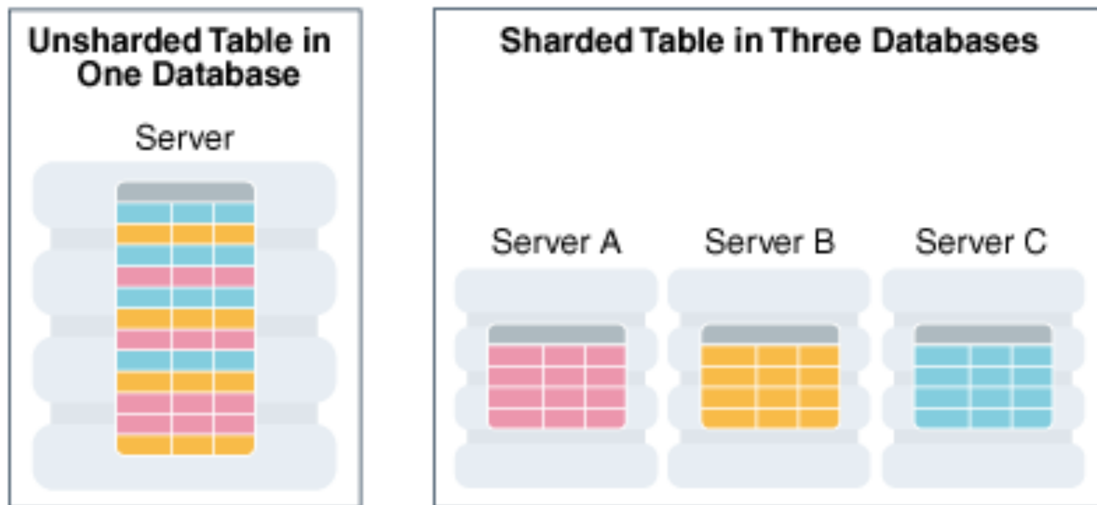

Figure 3 - Sharded Database Overview (Oracle, 2024)

According to (Reselman, 2021), the benefits of employing database sharding include the following:

- **Improve response time**: Data shards have fewer rows than the entire database. Therefore, it takes less time to retrieve specific information, or run a query, from a sharded database, in contrast to a single large database.
- **Avoid total service outage**: Database sharding prevents this by distributing parts of the database to different computers. Failure of one of the computers does not shut down the application because it can operate with other functional shards. Sharding is also often done in combination with data replication across shards. So, if one shard becomes unavailable, the data can be accessed and restored from an alternate shard.
- **Scale efficiently**: A growing database consumes more computing resources and eventually reaches storage capacity. Organizations can use database sharding to add more computing resources to support database scaling.

Like every other of its kind, this architecture has its drawbacks, such as (Reselman, 2021):

- **Requires advanced knowledge**: The sharding pattern requires that database administrators have both specific domain expertise and experience with the best practices of the database technologies in play to manage the sharding segmentation effectively.

- **Geolocation challenges**: Shards distributed over many geolocations can be susceptible to performance degradation due to excessive network traffic.
- **Technology dependent:** Some database technologies are better suited to the sharding pattern than others. Thus, you need to choose wisely.
- **Higher cost:** Added hardware means a higher total cost of ownership of the service.

(Abdelhafiz, 2021) states that the main reasons that justify the adoption of this strategy are:

- When a table grows so big that searching it becomes impractical even with the help of indexes.
- When data management is such that the target data is often the most recently added and/or older data is constantly being purged/archived.
- If you are loading data from different sources and maintaining it as data warehousing for reporting and analytics.
- For a less expensive archiving or purging of massive data that avoids exclusive locks on the entire table.

2.3 Sharding strategies

To implement database sharding effectively, choosing the right strategy is critical. The sharding strategy determines how data is partitioned across shards, directly influencing the system's performance, scalability, and manageability. Each strategy has unique characteristics, benefits, and trade-offs, making it essential to match the method to the application's specific requirements and data patterns.

In this section several sharding strategies will be explored, such as **Range Sharding**, **Hash Sharding**, **Directory Sharding**, and **Round-Robin Sharding**. These approaches vary in the data distribution, addressing challenges such as load balancing and query efficiency.

2.3.1 Range Sharding

Range-based sharding is a method of splitting data across multiple shards by dividing it into contiguous value ranges based on a shard key (Amazon Web Services, 2023). This technique involves selecting a column or field, known as the shard key, and grouping its values into defined ranges. For instance, in a "members" table where the shard key is the "name" column, names

can be divided alphabetically: those starting with A-G are stored in shard 1, H-N in shard 2, and so forth (Guevara, 2024).

Another example would be records with IDs from 1 to 10,000 might go to one shard, while IDs from 10,001 to 20,000 are assigned to another. This approach is effective for data that naturally fits into ranges, such as names, dates or numeric identifiers.

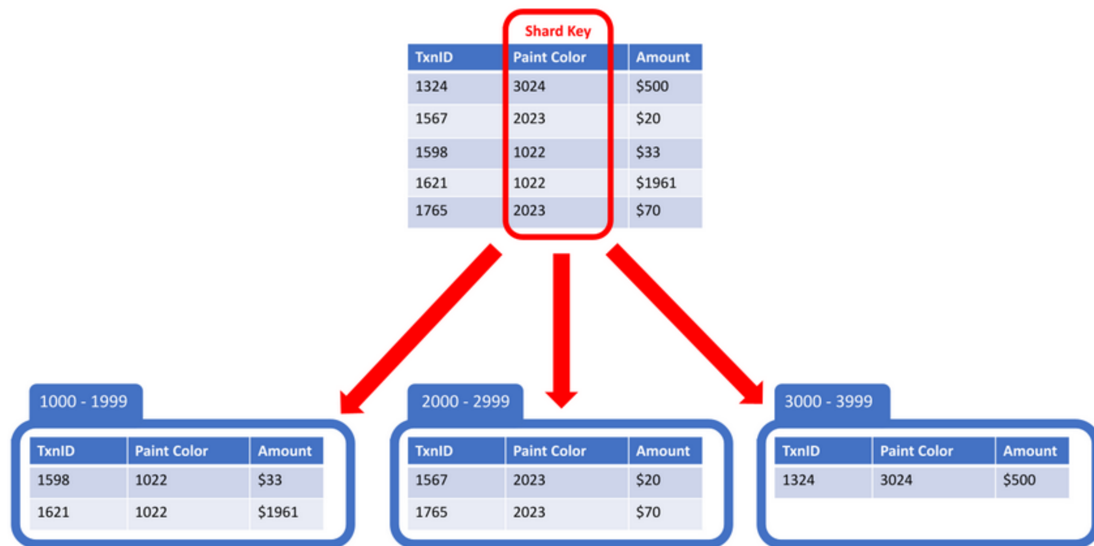

Figure 4 - Range-based sharding (Bailey, 2024)

In Figure 4, range-based sharding is applied to distribute records based on the "PaintColor" column, which serves as the shard key. The data is divided into predefined numeric ranges: Colors from 1000 to 1999 are stored in one shard, colors from 2000 to 2999 are stored in another shard, and colors from 3000 to 3999 are placed in a third shard.

This method is effective when data can be logically grouped into ranges, such as sequential transaction IDs or numeric identifiers, allowing efficient queries and data management

in a logical and predictable manner, making it easy to locate which shard contains specific records.

A possible problem with this strategy is naturally that depending on how the values are distributed, some shards can easily be overloaded. Using the example given before, if for some reason 80% of the "PaintColor" values were in the 2000 to 2999 range, it would cause a significant overload on the shard.

However, this issue can be addressed with strategies like resharding (split said shard into multiple smaller shards) or adjust ranges to account for uneven data distribution—for example, assigning a larger range to shards with less frequent values to balance the load more effectively (Guevara, 2024).

2.3.2 Hash Sharding

In this approach, data is allocated to shards using a hash function applied to a specific attribute, such as an ID or transaction number. Unlike range-based sharding in the previous example, which organizes data by contiguous number-based ranges, this method uses the hashed output of the shard key to evenly distribute data across the available shards.
The hashing process ensures an even distribution of data across shards, which helps balance the load and avoids overburdening any single shard (Aslan, 2023).
One advantage of this approach is that it removes human guesswork from the process, ensuring the data is evenly distributed, provided the shard key has high cardinality. Additionally, this method simplifies resharding if certain shards experience uneven loads. The primary drawback is the extra computational step of hashing, but in most cases, this added step has a negligible impact on performance (Guevara, 2024).

2.3.3 Directory based Sharding

Directory based sharding, also known as lookup-based sharding, uses a lookup table to assign data to specific shards, and instead of directly calculating where data should go, this method relies on a predefined mapping between shard keys and shards (Amazon Web Services, 2023).

For example, if you want to split a “members” table by region, you can create a lookup table that maps each region (or “court” as shown in the figure below) to a specific shard. The “court” column serves as the shard key, and the lookup table determines which shard will store the data for members from each court. This approach allows for flexible and custom data distribution based on the defined mappings.
Figure 5 illustrates how this strategy is implemented.

members

id	court	name
100	night	Feyre
101	summer	Tarquin
102	spring	Tamlin
103	night	Cassian
104	night	Morrigan
105	spring	Lucien
106	night	Rhysand
107	day	Helion
108	night	Azriel
109	night	Amren
110	night	Elaine
111	night	Nesta
..	..	..
..	..	..
..	..	..
12M	dawn	Thesan

Lookup table

court	shard
summer	1
spring	2
night	3
day	4
..	2
..	3
dawn	1

Figure 5 - Directory-based sharding (Guevara, 2024)

The main advantages and disadvantages are (Guevara, 2024):

Advantages:

- **Flexible shard allocation:** This approach is ideal when there are specific criteria for grouping data, such as separating data by regions or categories. It also makes it straightforward to add new shards as the dataset grows. For instance, if new categories are introduced, additional shards can be created, or existing shards can accommodate the new categories by updating the lookup table.
- **Scalability:** Adding new shards or redistributing data is manageable by simply updating the lookup table. This adaptability ensures the system can scale as needed.

Disadvantages:

- **Risk of imbalanced data distribution:** Uneven data or traffic distribution across shards can lead to performance bottlenecks. For instance, if certain groups accumulate a disproportionate amount of data or receive significantly more queries, one shard may become a hotspot, resulting in slower response times for that shard.
- **Increased Query Complexity:** The reliance on a lookup table introduces an extra step for database operations, as each query must first determine the appropriate shard location. While caching mechanisms can mitigate this latency, the need to maintain and update the lookup table adds complexity to the system.

2.3.4 Round-Robin Sharding

Data is assigned to shards sequentially, cycling through all available shards in turn. This straightforward method is easy to implement and is best suited for evenly distributed datasets, as it does not account for specific attributes or patterns in the data (Aslan, 2023).
One obvious advantage is its simplicity and ease to implement and the fact that it doesn't require any complex algorithms or calculations. The drawbacks are the lack of data grouping (related data may end up on different shards because there is no pattern to the distribution, and lack of flexibility for scenarios where complex patterns or data relationships occur (Aslan, 2023).

2.4 Alternatives to sharding

While sharding is a prominent strategy for scaling databases horizontally, several alternative approaches exist that can be employed based on specific system requirements and constraints. In this section we will analyze some of the possible alternatives

2.4.1 Vertical scaling

Vertical scaling involves upgrading the hardware resources of a single database server, such as adding more CPU power, memory, or storage capacity. This approach is straightforward and often serves as an initial scaling strategy. However, it is inherently limited by the maximum capacity of the hardware and may introduce a single point of failure. Vertical scaling is suitable for smaller applications where scaling demands are modest (Hennessy & Patterson, 2020).

2.4.2 Horizontal scaling

Horizontal scaling through replication distributes database workloads across multiple servers without partitioning data. This method typically employs:

1. **Read Replicas**: Secondary servers handle read operations, reducing the load on the primary server.
2. **Write Master with Standby Nodes**: Write operations are managed by a primary node, with secondary nodes available for failover.

While this approach improves scalability for read-heavy workloads, it may not adequately address write-intensive scenarios and could lead to data consistency challenges (Rajaraman et al., 2015).

2.5 When does sharding become necessary

From a data volume perspective, sharding becomes an adequate solution when databases grow to the scale of hundreds of gigabytes or terabytes, when indexes no longer fit in memory and query latency increases significantly. For example, an e-commerce platform processing millions of customer transactions daily may experience index bloat, table scans, and increased I/O contention if confined to a single node (Stonebreaker, 2005).

From a workload throughput perspective, sharding is also necessary when a single server cannot handle the concurrency or transaction rate required by the application. This is common in domains with high read/write patterns such as social media platforms, IoT sensor networks, or financial trading systems (Kraska et al., 2013). In such cases, horizontal scaling across shards distributes both storage and computational load.

It is important to note that premature sharding will likely introduce unnecessary complexity, such as managing cross-shard queries, re-sharding, and balancing. Therefore, sharding is most appropriate once performance degradation due to data volume (hundreds of millions to billions of records) or query throughput (thousands of requests per second) can no longer be mitigated with indexing, caching, or vertical scaling.

2.6 Existing technologies

This section presents some of the technologies that currently exist which support this strategy, and which patterns are used in database sharding to ease the migration process of traditional monolith databases.

2.6.1 Sharding on MongoDB

MongoDB describes itself as “a document-based database that offers both scalability and flexibility, combined with powerful querying and indexing capabilities”(MongoDB, 2023). As a

NoSQL database, it moves away from the traditional table-based relational model, opting instead for JSON-like documents with dynamic schemas. This approach provides a simpler and more flexible way to integrate data(MongoDB, 2023).

Some of its key features include (MongoDB, 2023):

- **High performance**: Support for embedded data models and indexes support faster queries.
- **Rich query language**: Read and write operations and text search and Geospatial Queries
- **High availability**: Its "Replica set" (group of MongoDB servers that maintain the same data set) provides automatic failover and data redundancy.
- **Horizontal Scalability**: Sharding and sharding zones based on a "shard key"

Later in this document some practical benefits from using this database will be analysed.

2.6.2 Sharding on AWS Aurora

Amazon Web Services (AWS) offers several powerful strategies for distributing data across multiple servers, enabling organizations to efficiently manage large datasets as they grow. In the domain of database management, Amazon's Aurora service – a relational database management system - emerges as a notable choice of in the context of database sharding.

Aurora's key characteristic is its 'share-nothing' model, where each shard operates independently. This approach enhances scalability and fault tolerance by eliminating the need for managing communications and contentions among database members (Amazon Web Services, 2024c). However, this model introduces complexities, especially in scenarios requiring data reads or joins across multiple shards.

In terms of monitoring and scaling, Aurora provides comprehensive tools. Amazon CloudWatch, for instance, offers a unified view of metrics across database shards, aiding in performance analysis and capacity planning (Amazon Web Services, 2024b).

One of Aurora's standout features is its support for shard consolidation. Consolidating multiple shards or partitions into a single instance can optimize cross-partition queries, reduce data fragmentation, and lower the total cost of ownership. This feature is particularly advantageous for applications requiring high query throughput while minimizing infrastructure complexity. The ability to merge shards seamlessly provides businesses with flexibility in adapting their database architecture as needs evolve (Zeng, 2019).

Aurora's synergy with other AWS services enhances its utility for modern cloud-native applications. For example, integrating Aurora with Amazon Lambda enables serverless processing of

database events (Amazon Web Services, 2024e), while Amazon S3 can be used to store archival data or large objects that don’t fit well into relational tables (Amazon Web Services, 2024d). These integrations allow organizations to design end-to-end systems that balance performance, cost, and scalability effectively.

2.6.3 Sharding on PostgreSQL

One of the most well-known databases is PostgreSQL, which according to (White, 2024), was the second most used database on the year of 2024.
(PostgreSQL, 2024) describes itself as a “powerful, open-source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads”.

PostgreSQL offers several powerful strategies for distributing data across multiple servers, enabling organizations to efficiently manage large datasets as they grow. These techniques, collectively known as sharding, allow for horizontal scaling of databases while maintaining optimal performance and reliability (Valle, 2024).

Several methods can be employed to distribute data effectively (Valle, 2024):

1. **Column-based Partitioning**: This strategy involves dividing data based on specific columns rather than entire tables. It's particularly useful when certain columns are accessed more frequently or have distinct access patterns.

2. **Hash-based Distribution**: By applying a hash function to a specific column value, data can be evenly distributed across shards. This approach simplifies query routing and ensures relatively uniform data distribution.

3. **Range-based Sharding**: Suitable for scenarios where data follows a natural order (e.g., chronological data or geographic regions). This method partitions data based on predefined ranges of values.

4. **Composite Sharding**: A hybrid approach that combines multiple sharding strategies to achieve optimal data distribution and query performance.

5. **Row-based Sharding**: In this method, data is divided based on rows, with each shard containing a subset of rows from the tables. It's suitable for scenarios where data distribution is relatively uniform across the dataset.

2.7 Case studies

This section explores real-world experiences and past challenges encountered in the pursuit of scalable databases. These examples offer valuable lessons and illuminate the complexities faced when implementing database sharding as remedy for scalability issues. By examining these practical scenarios and the hurdles that have been overcome, we gain a better understanding of the approaches that have proven successful in the field, setting the stage for the subsequent analysis and findings.

2.7.1 Database Sharding with MongoDB

In this section there will be two case studies about sharding implementation using MongoDB.

2.7.1.1 Krusche & Company

(Krusche & Company, 2024) conducted a study on database scaling solutions, demonstrating the effectiveness of sharding in optimizing data management for large-scale IoT applications. Their client, a leading innovator in software and IoT devices for photovoltaic plant monitoring and management, faced substantial data storage and accessibility challenges.

The client’s innovative solutions were utilized by 327,000+ photovoltaic plants across 138 countries, generating vast amounts of data from numerous IoT devices. This data deluge necessitated an efficient, scalable, and reliable storage solution that could handle growing data volumes while ensuring high availability and fault tolerance (Krusche & Company, 2024).

To address these challenges, (Krusche & Company, 2024) opted for a MongoDB sharding approach. This decision resulted in significant cost savings, with estimates suggesting annual savings of €100,000 compared to the enterprise solution.

The case study highlights several key points (Krusche & Company, 2024):

1. Performance optimization: Sharding allowed for efficient distribution of data across cluster nodes, ensuring high availability and rapid access to information.
2. Innovation in IoT data management: The solution demonstrated how sharding can effectively handle the complexities of large-scale IoT data generated by photovoltaic plants worldwide.
3. Balancing innovation and cost-effectiveness: The project showcases how cutting-edge technology can be implemented while maintaining fiscal responsibility.
4. Scalability and flexibility: The open-source nature of MongoDB allowed for easy adaptation to the company's specific needs as they grew and evolved.
5. Reduced operational overhead: By avoiding the need for costly enterprise licensing, the company was able to allocate resources more efficiently towards product development and customer support.
6. Future-proofing: The sharded architecture provides a foundation for future growth, allowing the system to scale horizontally as data volumes continue to increase.

This case study serves as an illustrative example of how innovative scaling strategies can lead to both performance improvements and significant cost savings in large-scale data management projects, particularly in the context of IoT applications (Krusche & Company, 2024).

The solution explored in this case study is not groundbreaking in the sense that nothing that hasn’t been done previously was done. However, the MongoDB sharding approach to clusters both fully answered the requirements of the project and highlighted how database sharding can save financial and temporal resources.

2.7.1.2 Foursquare

A study conducted by (Pathak, 2021) analysed the impact of sharding on a company named “Foursquare”, one of the fifty-two thousand companies using MongoDB (MongoDB, 2024).

According to the study, Foursquare, a popular location-based social networking platform launched in 2009 (uberall, 2024), faced significant challenges in scaling its infrastructure to accommodate rapid user growth. Initially relying on a single relational database, the company soon realized the need for a more robust solution to handle increasing data volumes.

To address this issue, Foursquare implemented a two-node architecture, separating check-in data onto a dedicated node. However, it became apparent that even this approach would eventually reach its limits as check-in data continued to grow exponentially (Pathak, 2021).

MongoDB emerged as a suitable solution for Foursquare’s growing requirements, providing a scalable approach to managing its expanding data needs. Through its auto-sharding capabilities, MongoDB enabled the automatic partitioning of data across multiple nodes, distributing both storage and computational load. This approach facilitated the seamless scaling of “write” operations and allowed Foursquare to incorporate new nodes effortlessly as the application continued to grow (Pathak, 2021).

One of the key advantages MongoDB provided was the ability to simplify Foursquare's data model. For instance, tags such as "has wifi," "great for dates," and "hotspot" were no longer stored in separate tables but embedded directly within venue documents. This approach not only improved runtime efficiency but also made the data structure significantly easier for engineers to understand and manipulate (Pathak, 2021).

By adopting MongoDB, Foursquare was able to focus its engineering resources on developing innovative features rather than building and maintaining complex sharding infrastructure. The solution allowed the company to continue its rapid expansion while maintaining data integrity and performance.

2.7.2 Database Sharding on Amazon Aurora

The analysis presented in the following paragraphs synthesizes the case study of Reddit's migration to Amazon Aurora and the latter’s technical aspects as a sharding solution.
Reddit, a social media company experiencing a 30% year-over-year growth in monthly active users, faced significant operational challenges with its self-managed PostgreSQL database. This growth necessitated a scalable and efficient database management solution or invest more money into development resources. Reddit chose the latter and opted to migrate key workloads to Amazon Aurora, a MySQL- and PostgreSQL-compatible managed relational database service (Amazon Web Services, 2020).
Following the migration, Reddit experienced enhanced database reliability, quicker recovery times, and streamlined failovers, which now occur in approximately 30 seconds. This transition freed the team from time-consuming administrative tasks, saving approximately two business

days per month, enabling engineers to focus on future projects or more valuable tasks (Amazon Web Services, 2020).
Initially, Reddit relied on Amazon EC2 (Amazon Elastic Compute Cloud- service that provides scalable computing capacity in the Amazon Web Services (AWS) Cloud (Amazon Web Services, 2024a) instances and Amazon S3 for its data storage and management needs. While these services provided control and flexibility, the rapid growth in user activity and data presented significant challenges in maintaining infrastructure. Managing PostgreSQL on EC2 required a lot of effort in maintenance, patching, and scaling, leaving little time for other initiatives. Aurora alleviated this burden by automating administrative tasks, allowing the Reddit team to focus on strategic, long-term goals, reducing operational strain on the team to perform repetitive tasks (Amazon Web Services, 2020).
The move to Aurora also improved operational efficiency by simplifying backup restoration and point-in-time recovery. Tasks that previously required manual intervention on EC2 are now automate. Additionally, Aurora’s data cloning capabilities provide other teams within Reddit access to production data for analytics and experimentation without additional infrastructure burdens. This accessibility fosters innovation while maintaining operational efficiency (Amazon Web Services, 2020).

As the social media company continues to grow at a 30% annual rate, Aurora has become a critical component of its strategy for managing relational workloads. This migration underscores the benefits of managed database solutions for organizations undergoing rapid growth and highlights how automation and advanced database technologies can address operational challenges effectively (Amazon Web Services, 2020).

2.8 Impact of sharding on query complexity

Sharding, while beneficial for scalability and fault tolerance, introduces unique challenges in handling queries and updates that span multiple shards. These arise from the inherent distribution of data across separate nodes, which will complicate operations that require access to data stored in different shards.
When data is partitioned into multiple shards, queries (e.g., aggregation queries or joins) become inherently more complex. This requires a coordination mechanism to read and update data accurately (Hazelcast, 2025).

Several strategies can be employed to minimize the impact of sharding on queries and updates (Keep & Ingo, 2020):

- **Strategic Shard Key Design**: Choosing a shard key that minimizes the likelihood of cross-shard operations can significantly enhance performance. For instance, grouping related data within the same shard (e.g., by user ID or region) can reduce the need for cross-shard queries.
- **Data Denormalization**: Storing redundant copies of frequently queried data in each shard can reduce the need for cross-shard joins but comes at the cost of increased storage and update complexity.
- **Query Optimization**: Using caching mechanisms and optimizing query paths can help mitigate the latency introduced by multi-shard queries.

In the practical portion of this dissertation, these challenges will be analyzed, with a particular focus on the trade-offs between performance and consistency in multi-shard operations. Experimental results will shed light on how different sharding strategies affect query and update performance in a controlled environment.
Features like (MongoDB, 2025b)'s distributed transactions or cross-shard writes are very useful when facing these challenges.

2.8.1 Performance comparison: Traditional vs Sharded Database

In a benchmarking test conducted by (Akka, 2025), a simulated IoT workload—consisting of 75% read operations and 25% write operations—was used to evaluate the performance of a single PostgreSQL instance compared to a sharded database system.

The findings revealed significant differences in throughput and query latency:

- **Single Database Instance (Traditional Monolithic Approach):**
  - Achieved a throughput of 108,000 requests per second
  - 99th percentile latency for read operations: 3 milliseconds
  - 99th percentile latency for write operations: 14 milliseconds
- **Sharded Database System (Distributed Approach):**
  - As the number of database shards increased, the system exhibited near-linear scalability
  - The workload was distributed across multiple database instances, preventing bottlenecks
  - Query response times remained consistently low, even under high traffic conditions

For context, a data point that falls at the 99th percentile has a value greater than 99 percent of the data points within the dataset (Britannica, 2025). Meaning that, in this example, the single database instance is in the worst 1 percent of response times.

These results emphasize that traditional databases become performance constrained as data scales, whereas a well-implemented sharding strategy allows for efficient horizontal scaling. By dividing data across multiple nodes, sharding reduces contention on individual database instances, leading to faster query execution, improved system responsiveness, and increased throughput.

2.8.2 Database partitioning for blockchain systems

A study by (Lakshman & Malik, 2010) investigates the challenges and opportunities of implementing sharding in distributed database systems, emphasizing the interplay between scalability, consistency, and query efficiency. The authors argue that traditional centralized databases struggle to handle large-scale workloads, whereas decentralized systems like Apache Cassandra leverage partitioning (sharding) to achieve high availability and scalability. However, they note that sharding can introduce challenges such as increased latency for cross-shard operations (e.g., range queries or multi-key transactions) and the need for careful trade-offs between consistency and performance.

Empirical evaluations in the paper demonstrate that Cassandra's decentralized architecture reduces query latency for write-heavy workloads by up to 40% compared to traditional relational databases. The authors also highlight key trade-offs: smaller partitions improve parallel query processing and fault tolerance but increase the overhead for coordinating cross-shard operations, whereas larger partitions simplify data locality at the cost of reduced scalability. The study also underscores the importance of dynamic partitioning and replication strategies to handle fluctuating workloads, showing that static sharding approaches can lead to suboptimal query performance and uneven load distribution over time.

The result of the study aligns with the theoretical findings discussed in previous sections of this document, particularly how sharding design choices—such as partition granularity, replication factor, and consistency tuning—directly impact query response times and system scalability.

2.9 Technologies

2.9.1 Database: PostgreSQL & MongoDB

The proposed solutions will be implemented in both PostgreSQL and MongoDB.

The first is a powerful, open-source relational database management system known for its reliability, extensibility, and compliance with SQL standards. Its compatibility with the **Citus extension** makes it an excellent choice for implementing sharding, as Citus enables horizontal scaling by distributing data across multiple nodes while maintaining ACID guarantees (CitusData, 2022).

Some benefits of this technology:

- **Sharding with Citus**: It extends PostgreSQL to support distributed tables, enabling seamless sharding without compromising data integrity or SQL (CitusData, 2022).
- **Strong Consistency**: PostgreSQL ensures transactional consistency, even in distributed environments, making it suitable for applications requiring reliable data operations.
- **Integration with Quarkus**: PostgreSQL integrates with Quarkus via JDBC or Hibernate ORM, simplifying database interactions and reducing development overhead.
- **Extensibility**: PostgreSQL supports custom extensions, stored procedures, and advanced indexing, providing flexibility for optimizing sharded database performance (PostgreSQL, n.d.).

The latter, following what we have established earlier in chapter two of this dissertation, is a document-oriented non-relational database designed for horizontal scalability and schema flexibility which natively supports sharding and it is one of the most used frameworks in the non-relational context.

Adding to the fact that is one of the most used databases, some of its key advantages include:

- **Built-in sharding**: Range and hash sharding options are provided, in addition to chuck balancing and scaling.
- **Flexible schema:** Its document model allows simpler data access patterns and reduces the amount of join operations.
- **High availability & load balancing:** Ensures robust availability through replica sets and distributed storage

2.9.2 Testing

Apache JMeter is an open-source performance testing tool designed to measure the behavior and performance of applications under load. It is widely used for stress testing, load testing, and functional testing of databases, web applications, and APIs.

Some of the most important benefits of this technology:

- **Scalability Testing**: JMeter can simulate thousands of concurrent users, making it ideal for evaluating the performance of sharded databases under high load (Almeida & Vieira, 2017).
- **Customizable Test Plans**: JMeter allows the creation of detailed test plans to simulate real-world query patterns, including cross-shard operations and complex transactions.
- **Comprehensive Metrics**: JMeter provides detailed reports on response times, throughput, error rates, and resource utilization, enabling in-depth analysis of sharding performance.
- **Integration with PostgreSQL**: JMeter supports JDBC connections, allowing direct testing of PostgreSQL queries and transactions.

While this study does not include actual performance tests due to limitations in available infrastructure and representative data, JMeter would have been the preferred tool for evaluating sharding strategies under load. Its robust feature set would enable a thorough analysis of performance bottlenecks, scalability trade-offs, and query efficiency across different sharding configurations. Future research with proper testing environments and realistic workloads could leverage JMeter to yield empirical, performance-driven insights.

2.10 Summary

This chapter presented a structured review of the state of the art in database scalability, focusing on the limitations of traditional monolithic systems and the emergence of sharding as a dominant solution. The discussion analyzed range, hash, directory-based, and round-robin strategies, evaluating their mechanisms, strengths, and drawbacks. The review showed that the effectiveness of each approach depends less on a universal rule and more on workload characteristics, data distribution, and query patterns, emphasizing the absence of a one-size-fits-all strategy.

Alongside theoretical perspectives, the chapter examined technological implementations in systems such as PostgreSQL, MongoDB, and Amazon Aurora, highlighting how different architectures operationalize sharding in practice. Case studies further illustrated the trade-offs between scalability, consistency, and complexity, while alternative approaches like vertical and horizontal scaling were acknowledged as limited but complementary solutions. Overall, the

analysis confirms sharding as the most viable technique for addressing modern large-scale data challenges, providing the conceptual basis for implementation strategies explored in the following chapters.

3 Domain role in sharding strategy

When implementing database sharding, one of the most critical questions architects and developers must answer is whether the domain of the application—the business model, data characteristics, and operational requirements—should directly dictate the sharding strategy, or whether any strategy can be universally applied with acceptable trade-offs. This chapter examines this question through theoretical discussion and practical examples, focusing more concretely on the context of the e-commerce application developed in this study.
The shard key and the sharding logic directly impact how data is split across databases, which in turn influences performance aspects such as latency, throughput, and fault isolation.
The domain refers to the core business context—such as an e-commerce platform, social network, or banking system—and defines how data is generated, accessed, and related.
It follows logically that the domain should influence the sharding strategy. However, one could argue that sharding is a technical concern that should be abstracted from business logic, particularly in modern distributed databases that encapsulate sharding as part of their infrastructure layer (Abadi, 2012).
In the following sections we will dive deeper into both arguments and their reasoning.

3.1 Domain influences sharding decision

In this section we will explore the reasoning behind the argument that the business domain impacts the sharding strategy.

3.1.1 Data access patterns are domain specific

The list below enumerates some examples of different business domains and how they would differ in terms of data access patterns (Kleppmann, 2017).

- E-commerce: High read volume for product listings, high write volume during order placement, and multi-tenancy (e.g., many sellers or stores).
- Social media: Highly interconnected data (friends, followers, comments), with unpredictable access patterns.
- Banking: Strong consistency requirements, many small and frequent transactions.

These patterns guide the selection of shard keys. For example:

- In e-commerce, sharding by *store_id* makes sense because data is tenant-bound.
- In social media, sharding by *user_id* works well to ensure all a user's data is in one place.
- In banking, range sharding by *account_number*, for example, could be risky if accounts aren't evenly distributed.

3.1.2 Business isolation requirements

Domains with multi-tenant architectures (like stores or marketplaces) benefit from hash sharding to isolate tenants. If each store is sharded separately, issues in one store don't affect others. Conversely, in a single-tenant domain like a government record system, such isolation is unnecessary and range sharding might be preferred for simplicity.

3.1.3 Regulatory and compliance concerns

In domains like healthcare or finance, certain data may need to reside in specific geographic regions. In this case, geo-sharding (a variant of range or hash sharding by region) would be the most adequate course of action, dictated entirely by domain-level rules.

3.1.4 Query Optimizations

If users frequently filter by date (e.g., "orders in last 30 days"), range sharding by a column like "created_at" helps localize queries to a single shard. If the domain doesn't emphasize time-based queries, then date-based sharding would only add unnecessary complexity.

3.2 Domain does not influence sharding decision

There is a counterargument that with the right abstractions and infrastructure, any sharding strategy can work for most domains. Such as:

- Middleware or database proxies can route queries based on metadata (Vitess.io, 2025).
- Modern distributed SQL databases like CockroachDB and Vitess handle sharding internally (Cockroach Labs, 2025).
- Cloud providers offer autoscaling clusters that abstract away physical sharding (Yun, 2024).

While this is true in principle, the performance and cost trade-offs warrant significant consideration. Ignoring domain-specific patterns can lead to problems such as:

- Uneven shard sizes (hot spots)
- Cross-shard queries (which are expensive and hard to optimize)
- Complex resharding or future rebalancing effort

So, while technically possible, applying a “one-size-fits-all” strategy will often underperform compared to a domain-optimized approach.

3.3 E-commerce application case study

Let’s dive deeper into how domain influences sharding through the lens of the e-commerce application, the base solution implemented in this study.

3.3.1 Domain overview

This platform is designed to support multiple independent sellers—stores—each managing their own inventory, orders, and customers. The platform operates as a multi-tenant system where each store operates semi-independently under the same system architecture.

Some of the key domain characteristics are:

- **Multi-tenancy**: Multiple stores with their own distinct datasets.

- **High data volume**: Orders, products, customer info, and logs can grow quickly over time.
- **Common access patterns**: Store-level queries (e.g., get all products of a store), time-based queries (e.g., list last month's orders), and cross-cutting queries (e.g., total platform revenue).
- **Need for fault isolation**: A store with poor performance shouldn't degrade the whole system.
- **Scalability**: The platform is expected to grow in number of stores and customers.

These business traits could influence the technical design of the sharding approach.

In Table 2 are some possible strategies and sharding keys that could be appropriate to the domain in question.

Table 2 - Sharding strategy alternatives

Entity	Sharding Strategy	Sharding key	Rationale
Store	Hash	store_id	Even data distribution across shards and isolates stores from each other.
Product	Hash	store_id	Products belong to a store, so co-locating store and product data avoids cross-shard joins.
Orders	Range	order_date	Time-based queries are frequent (e.g., monthly reports). Range sharding by date improves efficiency.
Customers	Hash	customer_id	Customers can register across stores; a hash avoids data skew.

Based on the domain overview and Table 2, we can see that , for the domain selected for the case study, we could choose hash sharding in stores and products to ensure even distribution and tenant isolation, range sharding in orders to allow chronological querying/reporting/archiving and other strategies.
Given the analysis from this section, it can be concluded that such strategies would not be appropriate in a social media platform for example, because the amount of interactions between users would often require cross-shard access (Bhat & Chandrasekaran, 2015).

As noted by (Stonebreaker, 2005) in his seminal paper *"One size fits all does not work for database systems."*, the design and architecture of the database system should always reflect the application's domain, its workload characteristics, and the technical constraints at play.

3.4 Sharding key selection and architectural implications

it is a foundational decision that shapes the entire trajectory of a sharded database system. The shard key establishes the core link between the data model and actual application usage, impacting not only system performance but also long-term maintainability.

3.4.1 Distribution of data and workload

The distribution of data—and the resulting workload across nodes—depends directly on the shard key. An effective shard key ensures that data is evenly partitioned, preventing any single node from becoming a bottleneck (often referred to as "hotspots"). Poor shard key choices, such as monotonically increasing values (e.g., timestamps or sequential order IDs), can lead to data skew and overload specific shards (Kleppmann, 2017).

3.4.2 Query routing

Query routing in distributed databases is typically driven by the shard key. When queries include the shard key, requests are routed directly to the relevant shard, minimizing latency and network overhead. Omission of the shard key forces the system to broadcast queries to all shards, substantially increasing response times and undermining scalability (Oracle, 2025). Therefore, aligning the shard key with the application's most common and latency-sensitive access patterns is critical.

3.4.3 Cross-shard operations

The architecture must handle the overhead of cross-shard joins, transactions, and aggregations, which are often unavoidable if the shard key does not reflect the application's relational structure. In relational databases, this becomes especially problematic due to the ACID guarantees and the lack of a native support for distributed transactions. In non-relational systems like MongoDB, developers often resort to denormalization or application-level joins to avoid cross-shard penalties (Kleppmann, 2017).

3.4.4 Scaling and rebalancing

Shard key selection also affects the system’s ability to scale elastically. In systems where shard keys are evenly distributed (e.g., using a hashed ID), adding new shards can be achieved with minimal data migration. In contrast, range-based sharding will require complex rebalancing operations if the value distribution changes over time. This has architectural consequences in terms of how storage, caching, and data movement mechanisms are implemented.

3.4.5 Fault isolation

Some shard keys may provide natural fault isolation — for instance, using “tenantId” in a multi-tenant application allows each tenant’s data to be contained within a single shard. This separation can enhance availability and resilience, since failures or downtime on one shard do es not impact the others. The architecture must support this isolation by ensuring transactional and operational boundaries align with the shard key.

3.4.6 Conclusion

The sharding key is far from a routine decision, it is a cornerstone of both database and system architecture. It influences how data is accessed, distributed, and how the system will scale and evolve over time. Therefore, the choice of shard key should be made with a thorough understanding of the domain, workload, and architectural goals —informed by historical data access patterns and long-term growth projections.

3.5 Challenges of defining shard key

The selection of a shard key is arguably one of the most critical decisions when designing a horizontally sharded database architecture. Defining an effective one is never straightforward, and the ease of doing so varies significantly between relational and non-relational database systems.

3.5.1 Relational databases

In traditional relational databases like PostgreSQL, defining a shard key is often a manual and complex process. Since relational models are inherently normalized and structured around foreign key relationships, choosing a single attribute to partition data can result in fragmented joins, cross-shard transactions, and data duplication if not handled carefully (Kleppmann, 2017). Within an e-commerce schema where orders, products, and users are stored in separate tables with foreign key references, choosing "user_id" as a shard key might simplify queries related to users and their orders, but could complicate queries involving product inventories or cross-user analytics.

In such cases, developers may need to:

- Analyze data access patterns.
- Consider consistency and transactional requirements.
- Implement custom routing and query rewriting logic.
- Accept the overhead of cross-shard operations.

Because of this, defining the shard key in a relational system typically requires expert-level knowledge of both the data model and the application workload.

In a range-based sharding strategy, the shard key is typically a value with a natural ordering—such as timestamps (e.g., order_date), numerical IDs, or even prices. This method is effective for time-series data or when queries are commonly issued over specific value intervals. However, like we have established in earlier sections of this document, it is susceptible to hotspotting, especially when inserts are concentrated at the high end of the range (e.g., new orders all arriving with the latest timestamp).

In contrast, hash-based sharding uses a hash function (e.g. user_id) to uniformly distribute data across shards. The shard key in this case is often an attribute with high cardinality and even distribution, such as a user_id, product_id, or transaction_id. Hash sharding reduces the risk of hotspots, but it complicates range queries and can make debugging or resharding more difficult (Özsu & Valudirez, 2014).

Therefore, selecting a shard key in a relational system is a strategic decision that must consider not only data distribution and query patterns, but also the business logic and growth expectations of the application.

3.5.2 Non-Relational databases

Opposite to their relational counterparts, databases like MongoDB offer more intuitive mechanisms for sharding. These systems are often designed from the ground up to support horizontal scaling and come with built-in support for defining and managing shard keys.

MongoDB, for example, allows developers to specify a shard key using a single command:

```
1. sh.shardCollection("shop.orders", { userId: "hashed" });
```

As analyzed earlier, defining an ineffective shard key can lead to hotspots, inefficient range queries and other future flexibility limitations.
Thus, while the tooling in non-relational databases makes the process easier, the conceptual understanding of workload characteristics remains essential.

3.5.3 Summary

The ease of defining a shard key is not just a technical issue—it depends on the nature of the data, access patterns, and the scalability needs the system will require. While non-relational systems tend to offer simpler APIs and better automation, relational systems typically demand more planning and expertise, particularly due to their normalized structure and strict transactional semantics.

Table 3 illustrates the differences in their features.

Table 3 - Relational vs non-relational characteristics

Feature	Relational	Non-Relational
Native sharding support	Manual / via extensions	Built-in
Shard key selection complexity	High – requires schema awareness	Moderate – tied to document fields
Cross-entity relationships	Common, adds complexity	Less common, often embedded
Automation	Low	High (chunk migration, balancing)
Risk of cross-shard issues	High if not carefully designed	Moderate, depending on key choice

3.5.4 Range sharding and the problem of recent data concentration

One of the main challenges of range-based sharding is the natural tendency for newly inserted records to accumulate in a single shard. Since this strategy partitions data according to an ordered attribute—such as a timestamp, user ID, or price range—the most recent values always fall into the "last" shard in the sequence. Over time, this creates data skew and results in an uneven workload across shards.

In domains such as social media, this effect becomes especially problematic. Platforms like Twitter, Instagram, or TikTok are inherently recency-driven, where most user interactions focus on the most recent content (e.g. the last 48h). If the shard key is based on chronological attributes (e.g., post_date), then all new posts and related queries concentrate on the shard holding the most recent range. This shard would become a hot spot, experience disproportionately higher read and write loads compared to older shards that contain less relevant historical data.

Several techniques have been proposed in both academic and industrial contexts to mitigate this issue:

- Bucketing Timestamps: Instead of sharding by a continuous time range, timestamps are divided into coarser "buckets" (e.g., daily or hourly partitions), distributing load more evenly (Rongali, 2025).
- Pre-Splitting Ranges: Creating multiple "future" shards in advance ensures that insertions can be balanced across more than one shard (MongoDB, 2025a).
- Hybrid Keys: Combining range attributes (e.g., post_date) with a hash function introduces randomness that helps distribute recent records across shards while still supporting partial range queries (Pachot, 2024).

While range sharding offers clear benefits for range-based queries and reporting, in recency-driven domains like social media, it introduces the critical challenge of concentrating the most valuable and frequently accessed data into a single shard. To adopt range sharding effectively in such environments, careful shard key design and additional balancing mechanisms are essential.

3.6 Geo sharding

Although this dissertation primarily emphasizes range and hash-based sharding, it's worth mentioning geo-sharding as another noteworthy approach for certain scenarios. Geo-sharding, in essence, organizes data based on the physical location of users or data sources. Each shard aligns with a specific geographic region—think North America, Europe, Asia, or, for more granularity, individual countries or states.

The key drivers for geo-sharding are reducing latency for users in different regions, ensuring compliance with local regulations, and containing faults so that issues in one area don't spill over and impact the entire system. By grouping data with its most frequent consumers, query response times are minimized, and overall user experience improves (Kossman et al., 2010). Moreover, geo-sharding addresses regulatory requirements such as data residency laws (e.g., GDPR in the European Union), ensuring that user data remains within legally mandated jurisdictions (Veeraragavan & Zhang Kaiwen, 2020).From an operational perspective, isolating failures to a single geographical shard can also enhance availability and resilience, since outages in one region do not necessarily affect global access.

Typical use cases for geo-sharding include:

- Social media platforms, where user interactions are often geographically clustered.
- Global e-commerce platforms, which must adhere to regional compliance while optimizing performance for localized traffic.
- Financial and healthcare systems, where legal frameworks mandate strict geographical control of sensitive data.

Despite these advantages, geo-sharding also introduces challenges. Uneven user distributions can lead to imbalanced shards (e.g., North America having disproportionately more traffic than smaller regions), while cross-region interactions may still require expensive inter-shard communication (Hellerstein et al., 2007). Additionally, routing mechanisms and failover policies become more complex when dealing with geographically distributed infrastructure.

In summary, while geo-sharding is not the primary focus of this work, it represents a critical strategy for globally distributed, compliance-sensitive applications. Its role is complementary to range and hash sharding, and in many real-world systems, a hybrid approach combining multiple sharding strategies may be required.

3.7 Summary

This chapter examined the role of domain considerations in shaping sharding strategies, beginning with an analysis of how data access patterns, business isolation requirements, regulatory concerns, and query optimization differ across application contexts. It was argued that domains such as e-commerce, banking, or social networks present unique workload characteristics that influence the choice of shard key and partitioning strategy. At the same time, the chapter considered the counterargument that advances in distributed databases and middleware can abstract away domain-specific concerns, thereby reducing the extent to which business context directly determines the chosen approach.

The e-commerce case study served as a concrete example, illustrating how sharding decisions can be guided by the need to handle large and diverse datasets, while balancing both performance and operational complexity. This practical scenario highlighted the interplay between theoretical sharding strategies and their implementation challenges, especially when scaling customer and order records across distributed environments.

Subsequent sections focused on shard key selection and its architectural implications. Key issues such as data distribution, query routing, cross-shard operations, and fault isolation were examined in detail, emphasizing the trade-offs inherent in different choices. Particular attention was given to the challenges of defining shard keys in both relational and non-relational databases, where schema design and workload predictability play decisive roles.

Finally, geo-sharding was introduced as an honorable mention, pointing to its relevance in contexts requiring geographic proximity or compliance with data residency regulations. The chapter concludes that while technological abstractions can alleviate some difficulties, domain-specific factors remain central to designing scalable, efficient, and legally compliant sharded architectures.

4 Implementation setup: Relational vs non-Relational

Earlier in this document, it was established that sharding is a widely recognized horizontal partitioning strategy to achieve scalability in modern database systems. However, its implementation varies significantly between relational database management systems (RDBMS) and non-relational (NoSQL) solution. This section serves as a comprehensive tutorial, offering high-level guidance and best practices for setting up sharding in each paradigm. The goal is to educate readers on how one could approach the implementation—not to detail a specific case.

4.1 Sharding in Relational Databases

Relational databases, due to their strong adherence to ACID properties, were historically not designed with horizontal sharding in mind. Nevertheless, the increasing demand for scalable systems has led to the emergence of manual and semi-automated sharding strategies within the relational context.

4.1.1 Manual sharding overview

Manual sharding in RDBMSs like PostgreSQL involves:

- Selecting a shard key (e.g., customer_id, store_id, order_date).
- Creating multiple databases or schemas, each storing a subset of data.
- Implementing application-level routing logic to direct queries to the appropriate shard.

This form of sharding has been studied and applied in various production contexts. Curino et al. (2010) describe this approach in their work *"Schism: a workload-driven approach to database replication and partitioning."* They emphasize workload-aware partitioning as a method to minimize distributed transactions.

4.1.2 Setup steps (PostgresSQL)

The first step is to choose a sharding key that evenly distributes the load and aligns with the most frequent query patterns, like we have explored in the earlier sections of this work.
Next up it would be to create shards, by manually instantiating multiple PostgreSQL databases or schemas. We can create different databases by:

```
1. createdb ecommerce_shard_1
2. createdb ecommerce_shard_2
```

Or different schemas within a single database:

```
1. CREATE SCHEMA shard1;
2. CREATE SCHEMA shard2;
```

Then distribute the tables in each shard

```
1. CREATE TABLE shard1.orders (...) ;
2. CREATE TABLE shard2.orders (...) ;
```

Lastly, it is crucial to implement the routing logic. This is done in the application layer.

```
1. int shardIndex = Math.abs(customerId.hashCode()) % totalShards;
2. DataSource ds = shardIndex == 0 ? shard1DataSource : shard2DataSource;
```

A few important aspects to keep in mind would be (Sikkayan, 2025):

- Foreign keys are typically avoided across shards, since this would require cross-shard /cross-database checks.
- Use of fan-out queries (query all shards) or have some pre-pepared metrics to avoid heavy joins.

4.1.3 Alternative: Middleware

An alternative to implementing the sharding on several databases with PostgreSQL could be Citus. It offers distributed Postgres with sharding at the database level and brings the following advantages (Cubukcu et al., 2021):

- It automates shard placement and query planning.
- Allows combining relational flexibility with NoSQL-like scalability.

4.2 Sharding in Non-Relational Databases

MongoDB sharding relies on the following components:

- Shards: Each is a replica set holding a portion of the dataset.
- Config Servers: Store metadata about the sharded cluster.
- Mongos Router: Query router that directs operations to the appropriate shard.

Academic research, such as, confirms that MongoDB's architecture allows for elastic scaling with relatively little operational overhead compared to traditional databases.

4.2.1 Setup steps (MongoDB)

The first step is enabling sharding on the cluster, as shown in the example below

```
1. sh.enableSharding("ecommerceDB")
```

Once sharding is enabled, select and define the shard key, considering all the information we have consolidated in the previous sections of this document.

MongoDB supports two main strategies: ranged sharding and hash sharding.

4.2.1.1 Range sharding

This strategy can be defined as such:

```
1. db.orders.createIndex({ orderDate: 1 })
2. sh.shardCollection("ecommerceDB.orders", { orderDate: 1 })
```

In previous sections of this document we have already outlined extensively the advantages and drawbacks of this strategy, but there are some mitigation strategies that can also be applied:

- Pre-splitting: Manually creating chunk ranges before inserting data helps distribute writes more evenly.
- Bucketing: Grouping the shard key into larger buckets (e.g., by day instead of exact timestamp) helps prevent excessive splitting and hot spotting.
- Zone sharding: Assigning shards to specific ranges using zones allows fine-grained control over where data is stored.

4.2.1.2 Hash sharding

In the case of hash sharding, the setup could start as the following:

```
1. sh.shardCollection("ecommerce.customers", { customerId: "hashed" })
```

As explored previously, selecting between range and hash sharding in MongoDB depends heavily on the query patterns, write distribution, and business requirements of the application. Developers must analyze these factors early in system design to choose a shard key that aligns with their expected usage, while being prepared to apply mitigation strategies if needed.

From this point, MongoDB handles shard routing, chunk balancing, and range splitting automatically.

4.2.1.3 Operational considerations

- Auto-balancing: MongoDB automatically migrates chunks to balance shards.
- Monitoring: Use mongostat, mongotop, and sh.status() for real-time insight.
- Shard-aware queries: Index shard keys properly to avoid scatter-gather operations.

4.3 Comparative overview

The following table Table 4 presents a formal comparison between relational and non-relational database systems, highlighting their respective sharding capabilities, architectural considerations, and optimal use cases.

Table 4 - Database systems aspect comparision

Feature	Relational (PostgreSQL)	Non-Relational (MongoDB)
Native Sharding	Manual or extension-based (e.g., Citus)	Built-in with automatic handling
Setup Complexity	High (requires orchestration and config)	Moderate (mongos and config servers)
Cross-Shard Joins	Manual, complex to implement	Avoided, prefer denormalized schemas
Scaling Strategy	External logic or extensions (Citus, Vitess)	Internal routing via mongos
Shard Key Constraints	Fully customizable, requires logic	Constrained by shard key selection rules
Best Use Cases	Strong consistency, transactional systems, complex joins	High-scale, document-centric, schema-flexible apps

From a didactic standpoint, the contrast between relational and non-relational sharding paradigms provides a practical appreciation of design trade-offs. While relational systems like PostgreSQL require more manual coordination, they also offer stronger transactional guarantees and structured modeling. In contrast, solutions like MongoDB prioritize ease of scaling, built-in sharding, and flexibility over strict data integrity.

Engineers studying database scaling strategies should focus on:

- Understanding sharding patterns beyond just syntax, to choose shard keys wisely and avoid hot spots and aim for uniform distribution.
- Aligning sharding to access patterns and growth forecasts.
- Appreciating the operational lifecycle of a sharded system—monitoring, failure handling, and rebalance events.

5 Achieving scalability through sharding in an e-commerce application

This section distills the key insights from preceding chapters and translates them into actionable methodologies for implementation. It specifically addresses how scalability via sharding can be implemented, applicable to both relational and non-relational database systems. By bridging theoretical frameworks with practical application, this segment of the dissertation offers a coherent pathway for organizations seeking to leverage sharding to support the demands of large-scale, data-intensive environments.

5.1.1 Relational Database: PostgreSQL with Citus Extension

Relational databases were traditionally designed with ACID guarantees and strong consistency in mind, often favoring vertical scaling. However, extensions like Citus enable horizontal scaling by transforming PostgreSQL into a distributed system.

5.1.1.1 Sharding approach

- **Shard key selection:** For the e-commerce domain, a natural candidate for the shard key is the *customer_id*, since user-specific queries (e.g., orders, shopping carts, payment history) are often isolated per customer. This ensures that most queries are contained within a single shard, minimizing cross-shard joins.
- **Range sharding example:** Product pricing or order dates can serve as range-based shard keys. For instance, order_date enables chronological partitioning, useful for analytics and archiving (e.g., retrieving all orders from a specific time period). However, skewed distributions (e.g., seasonal shopping spikes) can create “hot” shards that require re-balancing.
- **Hash sharding example:** Using customer_id with a hashing function distributes customers evenly across shards, balancing load and avoiding hotspots. This is particularly useful for high volumes of concurrent transactions.

5.1.1.2 Practical example

The following code block illustrates how this concept can be expressed in SQL.

```
1. -- Enable Citus extension
2. CREATE EXTENSION IF NOT EXISTS citus;
3.
4. -- Create a table for orders
5. CREATE TABLE orders (
6.     order_id BIGINT,
7.     customer_id BIGINT,
8.     order_date TIMESTAMP,
9.     total_amount DECIMAL,
10.    PRIMARY KEY (order_id)
11. );
12.
13. ----------------------------------------------------------
14. -- Example 1: Distribute orders using HASH sharding
15. -- Hashing ensures an even distribution of rows by customer_id
16. ----------------------------------------------------------
17. SELECT create_distributed_table('orders', 'customer_id', 'hash');
18.
19. ----------------------------------------------------------
20. -- Example 2: Distribute orders using RANGE sharding
21. -- Range sharding partitions data sequentially by order_date
22. -- Useful for chronological queries, reporting, and archiving
23. ----------------------------------------------------------
24. CREATE TABLE orders_range (
25.     order_id BIGINT,
26.     customer_id BIGINT,
27.     order_date TIMESTAMP,
28.     total_amount DECIMAL,
29.     PRIMARY KEY (order_id)
30. );
31.
32. -- Distribute by range of order_date
33. SELECT create_distributed_table('orders_range', 'order_date', 'range');
34.
```

Like we have established before, while PostgreSQL with Citus maintains strong SQL semantics, some constraints remain: foreign keys across shards are unsupported (requiring application-level enforcement), and cross-shard joins can add latency, best mitigated with denormalization or pre-computed aggregates.

5.1.2 Non-Relational Database: MongoDB

In contrast, MongoDB was designed with horizontal scalability as a first-class concern, offering native sharding without the need for extensions. Its document-oriented model is well-suited to the e-commerce domain, where products, customers, and orders can be represented as flexible, nested documents.

5.1.2.1 Sharding Approach

- **Range Sharding:** Sharding collections by order_date allows efficient execution of range queries (e.g., sales reports for Q4). However, because most inserts are sequential by date, this can overload the most recent shard. MongoDB mitigates this through pre-splitting and chunk migration strategies (MongoDB, 2023).
- **Hashed Sharding:** Sharding by a hashed customerId distributes load evenly across shards, supporting high-volume point queries such as retrieving a user's order history. This avoids hotspots but sacrifices efficiency in range queries.

5.1.2.2 Practical example

The code block below demonstrates how one could implement both strategies in the non-relational context.

```
1. // Enable sharding on the database
2. sh.enableSharding("ecommerceDB");
3.
4. // Range sharding on orderDate
5. sh.shardCollection("ecommerceDB.orders", { orderDate: 1 });
6.
7. // Hashed sharding on customerId
8. sh.shardCollection("ecommerceDB.orders", { customerId: "hashed" });
9.
```

Like we have learned before, MongoDB abstracts most of the sharding complexity through mongo's query router and automatic chunk balancing. Still, cross-shard aggregations remain costly, making careful schema design essential.

5.1.3 Simulation constraints and hypothetical scaling scenario

Although the proposed e-commerce solution has been designed with scalability and sharding strategies in mind, it was not possible to fully replicate the required infrastructure and dataset within the scope of this research. Implementing and evaluating sharding at scale typically demands extensive computational resources, distributed environments, and data volumes in the order of tens of millions of records, which were not available during the development of this academic study.

As (Solat, 2024) highlights in his critical review of distributed database sharding, evaluating sharding mechanisms at scale inherently requires large, heterogeneous datasets and a distributed infrastructure capable of capturing coordination overhead, latency, and data distribution chal-

lenges. Without this scale, smaller experimental setups risk overlooking the systemic complexities—such as cross-shard communication or rebalancing—that only emerge with tens of millions of records.

Nevertheless, it is still possible to estimate and reason about the expected system behavior by projecting how relational and non-relational sharding would perform under a workload of, for example, ten million customer and order records:

- **PostgreSQL with Citus (Relational)**: If sharded by *customer_id* using a hash strategy, queries that target individual customers (e.g., retrieving purchase history) would scale efficiently, as data would be evenly distributed across shards. However, queries that require global aggregation (e.g., calculating platform-wide sales per product) would involve **fan-out queries** across all shards, introducing overhead in coordination and join processing.
- **MongoDB with Range Sharding (Non-Relational)**: Sharding on *orderDate* would allow highly efficient queries over chronological ranges, such as retrieving all sales within a promotional period. Inserting data sequentially, however, could lead to hot spotting on a single shard, unless mitigated with techniques such as pre-splitting or bucketing timestamps. For point lookups (e.g., retrieving a user profile by ID), MongoDB's hashed sharding would perform better, distributing load evenly across shards.

5.1.4 Routing between shards

Routing plays a pivotal role in any sharded database, since it determines how client queries are directed to the correct shard. This process abstracts the physical data distribution from client applications, thereby enabling seamless horizontal scalability without exposing underlying complexity to the domain layer.

5.1.4.1 Relational databases

In relational systems such as PostgreSQL with the Citus extension, routing is handled by a coordinator node. This coordinator uses the shard key to determine which worker node stores the relevant data, and all queries involving this field are intercepted by the coordinator, which then forwards the request to the correct shard. Queries without a shard key may result in "fan-out" operations across multiple shards, introducing higher latency (Kossman et al., 2010).

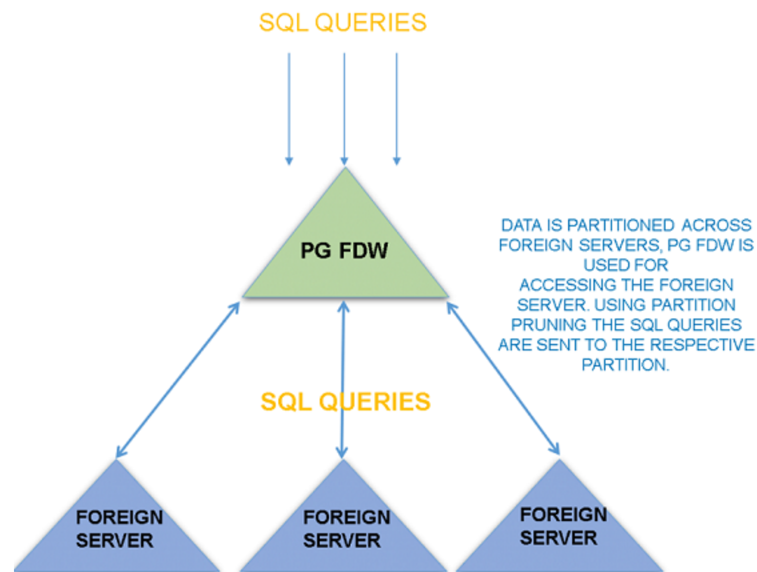

Figure 6 - Sharding in PostgreSQL (PostgreSQL, 2024)

Figure 6 illustrates query routing in a PostgreSQL sharded setup using Foreign Data Wrappers (FDW).

- At the top, SQL queries are issued by the application.
- These queries first reach the PG (Post Gres) FDW (Foreign Data Wrapper), which acts as a coordinator. It is responsible for determining which partition or shard the query should be routed to.
- At the bottom, we see multiple foreign servers, each storing a partition of the dataset. The data is distributed across these servers according to the sharding strategy.
- The partition pruning process ensures that queries are directed only to the relevant foreign server(s) rather than broadcasting to all. For example, if the query specifies a shard key value, PG FDW can route the query directly to the correct partition.

In PostgreSQL sharding, the FDW component serves as the router, coordinating access across distributed partitions. This design allows PostgreSQL to scale horizontally by distributing queries and data across multiple servers while minimizing unnecessary overhead.

5.1.4.2 Non-Relational databases

In non-relational systems such as MongoDB, routing is abstracted by the mongos query router. The mongos instance consults the cluster metadata (maintained by config servers) to locate which shard holds the requested data. If the query includes the shard key (e.g., { customerId: 12345 }), the router can forward it directly to a single shard. Without a shard key, the query

becomes a scatter-gather operation across all shards, increasing resource usage and latency (MongoDB, 2025b).

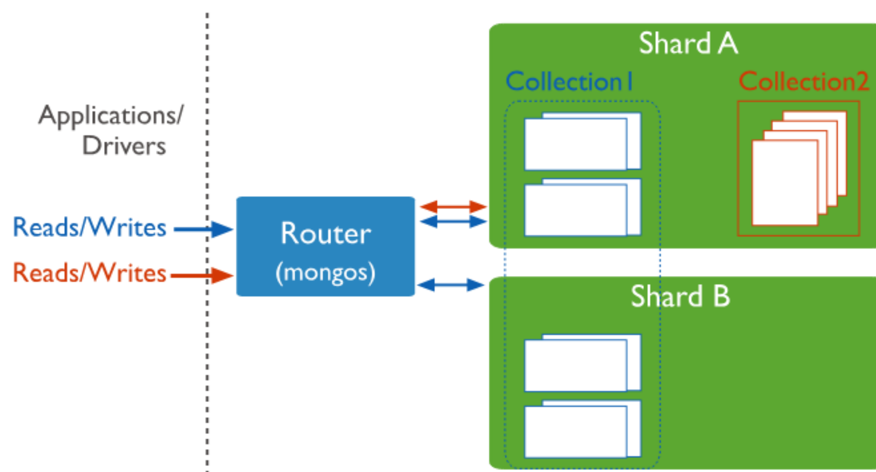

Figure 7 - MongoDB sharding (MongoDB, 2025b)

Figure 7 illustrates query routing in a MongoDB sharded cluster.

- On the left, the applications or drivers send read and write requests.
- These requests are first handled by the Router (mongos), which acts as the query router. "mongos" does not store data itself; instead, it uses metadata from the configuration servers (not shown here) to determine where the requested data resides.
- On the right, we see two shards (Shard A and Shard B). Each shard contains portions of the database collections. For example, Collection 1 is split between Shard A and Shard B, while Collection 2 is entirely stored in Shard A.
- The arrows indicate how reads/writes are routed: if the query includes the shard key, mongos can directly send the request to the correct shard. If the shard key is missing, mongos may need to broadcast the request to multiple shards.

This diagram shows the role of mongos in routing queries to the appropriate shard(s). By using the shard key, operations can be efficiently targeted, minimizing latency and resource usage. Without the shard key, queries can become expensive as they must be broadcast across shards.

Thus, efficient query routing is highly dependent on the selection of shard key. A well-designed key minimizes fan-out queries and ensures that most lookups and updates can be routed deterministically to a single shard, reducing overhead and improving scalability.

6 Demonstration of range-based sharding

This chapter presents a practical demonstration of sharding in a local environment, focusing on the range-based strategy. The aim is not to create a production-ready deployment but to illustrate the core mechanisms of sharding and query routing in a simplified setting. The demonstration is conducted exclusively in MongoDB, as its native sharding framework makes it easier to implement and visualize data distribution. Using a defined shard key, MongoDB automatically partitions and routes documents across shards, providing a clear illustration of how range-based sharding operates without the need for manual routing or additional extensions.

6.1.1 Create directories

For this experiment, a minimal setup will include:

- 2 shards
- 1 config server
- 1 mongos router

```
→  tese mkdir -p ~/mongo-sharding/config
mkdir -p ~/mongo-sharding/shard1
mkdir -p ~/mongo-sharding/shard2
mkdir -p ~/mongo-sharding/mongos
```

Figure 8 - Create directories for mongo sharding

In Figure 8 we can see the creation of the directories for each of the components.

6.1.2 Start the config server servers

To start the servers, we ran the commands in the following image:

```
→  Tese
→  Tese mongod --configsvr --replSet configReplSet --port 27019 --dbpath ~/mongo-sharding/config --bind_ip localhost
```

Figure 9 - Starting config server

And a similar command to shards 1 and 2, like such :

```
1. mongod --shardsvr --replSet shard1ReplSet --port 27018 --dbpath ~/mongo-sharding/shard1
--bind_ip localhost
2. mongod --shardsvr --replSet shard2ReplSet --port 27020 --dbpath ~/mongo-sharding/shard2
--bind_ip localhost
```

6.1.3 Initialize replica sets

Connecting to each shard we performed the following commands:

```
➜  tese mongosh --port 27019
Current Mongosh Log ID: 68d69af86c87b3a606730df7
Connecting to:          mongodb://127.0.0.1:27019/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.8
Using MongoDB:          7.0.24
Using Mongosh:          2.5.8

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-09-26T14:20:14.965+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
   2025-09-26T14:20:14.965+01:00: Soft rlimits for open file descriptors too low
------

test> rs.initiate({
|   _id: "configReplSet",
|   configsvr: true,
|   members: [{ _id: 0, host: "localhost:27019" }]
| })
{ ok: 1 }
configReplSet [direct: other] test>
```

Figure 10 – Initializing config replica set

```
➜  tese mongosh --port 27018

rs.initiate({
  _id: "shard1ReplSet",
  members: [{ _id: 0, host: "localhost:27018" }]
})
Current Mongosh Log ID: 68d69b30ee72e06a85ac2a0f
Connecting to:          mongodb://127.0.0.1:27018/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.8
Using MongoDB:          7.0.24
Using Mongosh:          2.5.8

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-09-26T14:20:53.294+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
   2025-09-26T14:20:53.294+01:00: Soft rlimits for open file descriptors too low
------

test> rs.initiate({
|   _id: "shard1ReplSet",
|   members: [{ _id: 0, host: "localhost:27018" }]
| })
{ ok: 1 }
shard1ReplSet [direct: other] test>
```

Figure 11 – Initializing replica 1

```
➜  tese mongosh --port 27020
Current Mongosh Log ID: 68d69b7fca6b099f2ac9e5f7
Connecting to:          mongodb://127.0.0.1:27020/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.8
Using MongoDB:          7.0.24
Using Mongosh:          2.5.8

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-09-26T14:21:33.631+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
   2025-09-26T14:21:33.631+01:00: Soft rlimits for open file descriptors too low
------

test> rs.initiate({
|   _id: "shard2ReplSet",
|   members: [{ _id: 0, host: "localhost:27020" }]
| })
{ ok: 1 }
```

Figure 12 - Initializing replica 2

At this stage, all servers are initialized and aware of their replica set roles.

6.1.4 Start the router and connect to the cluster

The mongos process is the entry point for client applications. It routes queries to the appropriate shard, based on metadata from the config server.

```
→  tese mongos --configdb configReplSet/localhost:27019 --bind_ip localhost --port 27017
{"t":{"$date":"2025-09-26T13:57:49.963Z"},"s":"W",  "c":"SHARDING", "id":24132,   "ctx":"thread1","msg":"Running a sharded cluster with fewer than 3 config servers should or
{"t":{"$date":"2025-09-26T14:57:49.964+01:00"},"s":"I",  "c":"NETWORK",  "id":4915701, "ctx":"thread1","msg":"Initialized wire specification","attr":{"spec":{"incomingExterr
{"t":{"$date":"2025-09-26T14:57:49.965+01:00"},"s":"I",  "c":"CONTROL",  "id":23285,   "ctx":"thread1","msg":"Automatically disabling TLS 1.0, to force-enable TLS 1.0 specif
{"t":{"$date":"2025-09-26T14:57:49.965+01:00"},"s":"I",  "c":"NETWORK",  "id":4648602, "ctx":"thread1","msg":"Implicit TCP FastOpen in use."}
{"t":{"$date":"2025-09-26T14:57:49.965+01:00"},"s":"I",  "c":"HEALTH",   "id":5936503, "ctx":"thread1","msg":"Fault manager changed state ","attr":{"state":"StartupCheck"}}
{"t":{"$date":"2025-09-26T14:57:49.965+01:00"},"s":"W",  "c":"CONTROL",  "id":22120,   "ctx":"thread1","msg":"Access control is not enabled for the database. Read and write
{"t":{"$date":"2025-09-26T14:57:49.966+01:00"},"s":"I",  "c":"CONTROL",  "id":23403,   "ctx":"mongosMain","msg":"Build Info","attr":{"buildInfo":{"version":"7.0.24","gitVers
{"t":{"$date":"2025-09-26T14:57:49.966+01:00"},"s":"I",  "c":"CONTROL",  "id":51765,   "ctx":"mongosMain","msg":"Operating System","attr":{"os":{"name":"Mac OS X","version":
{"t":{"$date":"2025-09-26T14:57:49.966+01:00"},"s":"I",  "c":"CONTROL",  "id":21951,   "ctx":"mongosMain","msg":"Options set by command line","attr":{"options":{"net":{"binc
{"t":{"$date":"2025-09-26T14:57:49.967+01:00"},"s":"I",  "c":"NETWORK",  "id":5693100, "ctx":"mongosMain","msg":"Asio socket.set_option failed with std::system_error","attr'
{"t":{"$date":"2025-09-26T14:57:49.967+01:00"},"s":"I",  "c":"NETWORK",  "id":4603701, "ctx":"mongosMain","msg":"Starting Replica Set Monitor","attr":{"protocol":"streamable
{"t":{"$date":"2025-09-26T14:57:49.967+01:00"},"s":"I",  "c":"-",        "id":4333223, "ctx":"mongosMain","msg":"RSM now monitoring replica set","attr":{"replicaSet":"config
{"t":{"$date":"2025-09-26T14:57:49.967+01:00"},"s":"I",  "c":"-",        "id":4333226, "ctx":"mongosMain","msg":"RSM host was added to the topology","attr":{"replicaSet":"cc
{"t":{"$date":"2025-09-26T14:57:49.967+01:00"},"s":"I",  "c":"CONNPOOL", "id":22576,   "ctx":"ReplicaSetMonitor-TaskExecutor","msg":"Connecting","attr":{"hostAndPort":"local
{"t":{"$date":"2025-09-26T14:57:49.968+01:00"},"s":"I",  "c":"NETWORK",  "id":23729,   "ctx":"ReplicaSetMonitor-TaskExecutor","msg":"ServerPingMonitor is now monitoring host
{"t":{"$date":"2025-09-26T14:57:49.968+01:00"},"s":"I",  "c":"NETWORK",  "id":4333213, "ctx":"ReplicaSetMonitor-TaskExecutor","msg":"RSM Topology Change","attr":{"replicaSet
9-26T13:57:49.000Z"},"opTime":{"ts":{"$timestamp":{"t":1758895069,"i":1}},"t":1},"type":"RSPrimary","minWireVersion":21,"maxWireVersion":21,"me":"localhost:27019","setName":
,"compatible":true,"maxElectionIdSetVersion":{"electionId":{"$oid":"7fffffff0000000000000001"},"setVersion":1}},"previousTopologyDescription":{"id":"8ae6271e-ffcc-4cac-9753-
{"t":{"$date":"2025-09-26T14:57:49.968+01:00"},"s":"I",  "c":"SHARDING", "id":471693,  "ctx":"ReplicaSetMonitor-TaskExecutor","msg":"Updating the shard registry with confirm
{"t":{"$date":"2025-09-26T14:57:49.968+01:00"},"s":"I",  "c":"SHARDING", "id":22846,   "ctx":"Sharding-Fixed-0","msg":"Updating sharding state with confirmed replica set","a
{"t":{"$date":"2025-09-26T14:57:49.968+01:00"},"s":"I",  "c":"NETWORK",  "id":6006301, "ctx":"ReplicaSetMonitor-TaskExecutor","msg":"Replica set primary server change detect
{"t":{"$date":"2025-09-26T14:57:49.969+01:00"},"s":"I",  "c":"SHARDING", "id":22842,   "ctx":"mongosMain","msg":"Waiting for signing keys, sleeping before checking again","a
{"t":{"$date":"2025-09-26T14:57:50.981+01:00"},"s":"W",  "c":"FTDC",     "id":23911,   "ctx":"mongosMain","msg":"FTDC is disabled because neither '--logpath' nor set paramet
{"t":{"$date":"2025-09-26T14:57:50.982+01:00"},"s":"I",  "c":"FTDC",     "id":20625,   "ctx":"mongosMain","msg":"Initializing full-time diagnostic data capture","attr":{"dat
{"t":{"$date":"2025-09-26T14:57:50.984+01:00"},"s":"I",  "c":"HEALTH",   "id":5936511, "ctx":"mongosMain","msg":"No active health observers are configured."}
{"t":{"$date":"2025-09-26T14:57:50.984+01:00"},"s":"I",  "c":"HEALTH",   "id":5936502, "ctx":"mongosMain","msg":"The fault manager initial health checks have completed","att
{"t":{"$date":"2025-09-26T14:57:50.984+01:00"},"s":"I",  "c":"HEALTH",   "id":5936503, "ctx":"mongosMain","msg":"Fault manager changed state ","attr":{"state":"Ok"}}
{"t":{"$date":"2025-09-26T14:57:50.984+01:00"},"s":"I",  "c":"NETWORK",  "id":23015,   "ctx":"listener","msg":"Listening on","attr":{"address":"/tmp/mongodb-27017.sock"}}
{"t":{"$date":"2025-09-26T14:57:50.985+01:00"},"s":"I",  "c":"NETWORK",  "id":23015,   "ctx":"listener","msg":"Listening on","attr":{"address":"127.0.0.1"}}
{"t":{"$date":"2025-09-26T14:57:50.985+01:00"},"s":"I",  "c":"NETWORK",  "id":23016,   "ctx":"listener","msg":"Waiting for connections","attr":{"port":27017,"ssl":"off"}}
```

Figure 13 - Start mongos router

6.1.5 Add shards to cluster

```
→  ~ mongosh --port 27017
Current Mongosh Log ID: 68d69c570738959246318274
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.8
Using MongoDB:          7.0.24
Using Mongosh:          2.5.8

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-09-26T14:57:49.965+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
------

[direct: mongos] test> sh.addShard("shard1ReplSet/localhost:27018")
{
  shardAdded: 'shard1ReplSet',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1758895205, i: 5 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1758895205, i: 5 })
}
```

Figure 14 - Add shard 1

```
[direct: mongos] test> sh.addShard("shard2ReplSet/localhost:27020")
{
  shardAdded: 'shard2ReplSet',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1758895221, i: 7 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1758895220, i: 4 })
}
[direct: mongos] test> []
```

Figure 15 - Add shard 2

The shard replica sets are now registered in the cluster.

6.1.6 Enable and setting sharding

We now enable sharding for the target database, *ecommerceDB*.

```
[direct: mongos] test> use ecommerceDB
switched to db ecommerceDB
[direct: mongos] ecommerceDB> sh.enableSharding("ecommerceDB")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1758895280, i: 8 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1758895280, i: 2 })
}
[direct: mongos] ecommerceDB>
```

Figure 16 - Enable sharding on ecommerceDB

Then, we create the collection and shard it based on *customerId*.

```
[direct: mongos] ecommerceDB> db.createCollection("customers")
{ ok: 1 }
[direct: mongos] ecommerceDB> sh.shardCollection("ecommerceDB.customers", { customerId: 1 })
{
  collectionsharded: 'ecommerceDB.customers',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1758895432, i: 37 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1758895432, i: 37 })
}
[direct: mongos] ecommerceDB>
```

Figure 17 - Creating collection and defining sharding

To explicitly split the ranges, the boundary was set at id=1000.

```
}
[direct: mongos] ecommerceDB> sh.splitAt("ecommerceDB.customers", { customerId: 1000 })
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1758896110, i: 6 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1758896110, i: 6 })
}
[direct: mongos] ecommerceDB>
```

Figure 18 - Create sharding range

6.1.7 Insert data and testing

To insert a small set of data with five thousand sequential customers we ran a loop shown in Figure 19.

```
[direct: mongos] ecommerceDB> for (let i = 1; i <= 5000; i++) {
|   db.customers.insertOne({ customerId: i, name: "Customer" + i })
| }
```

Figure 19 - Inserting data

Then, we ran “sh.status()” to verify sharding status and distribution

```
---
shardedDataDistribution
[
  {
    ns: 'config.system.sessions',
    shards: [
      {
        shardName: 'shard1ReplSet',
        numOrphanedDocs: 0,
        numOwnedDocuments: 39,
        ownedSizeBytes: 3861,
        orphanedSizeBytes: 0
      }
    ]
  },
  {
    ns: 'ecommerceDB.customers',
    shards: [
      {
        shardName: 'shard2ReplSet',
        numOrphanedDocs: 0,
        numOwnedDocuments: 4001,
        ownedSizeBytes: 244061,
        orphanedSizeBytes: 0
      },
      {
        shardName: 'shard1ReplSet',
        numOrphanedDocs: 0,
        numOwnedDocuments: 999,
        ownedSizeBytes: 58941,
        orphanedSizeBytes: 0
      }
    ]
  }
]
---
```

Figure 20 - Sharding distribution

```
databases
[
  {
    database: { _id: 'config', primary: 'config', partitioned: true },
    collections: {
      'config.system.sessions': {
        shardKey: { _id: 1 },
        unique: false,
        balancing: true,
        chunkMetadata: [ { shard: 'shard1ReplSet', nChunks: 1 } ],
        chunks: [
          { min: { _id: MinKey() }, max: { _id: MaxKey() }, 'on shard': 'shard1ReplSet', 'last modified': Timestamp({ t: 1, i: 0 }) }
        ],
        tags: []
      }
    }
  },
  {
    database: {
      _id: 'ecommerceDB',
      primary: 'shard2ReplSet',
      partitioned: false,
      version: {
        uuid: UUID('23a9c877-0adc-415f-b8e2-8fad8844dbab'),
        timestamp: Timestamp({ t: 1758895279, i: 1 }),
        lastMod: 1
      }
    },
    collections: {
      'ecommerceDB.customers': {
        shardKey: { customerId: 1 },
        unique: false,
        balancing: true,
        chunkMetadata: [
          { shard: 'shard1ReplSet', nChunks: 1 },
          { shard: 'shard2ReplSet', nChunks: 1 }
        ],
        chunks: [
          { min: { customerId: MinKey() }, max: { customerId: 1000 }, 'on shard': 'shard1ReplSet', 'last modified': Timestamp({ t: 2, i: 0 }) },
          { min: { customerId: 1000 }, max: { customerId: MaxKey() }, 'on shard': 'shard2ReplSet', 'last modified': Timestamp({ t: 2, i: 1 }) }
        ],
        tags: []
      }
    }
  }
]
[direct: mongos] ecommerceDB> 
```

Figure 21 - Database status

In Figures 20 and 21 the sharding distribution is shown and it can be proved that the first one thousand records, with id lower than 1000, are on shard1 and the remaining four thousand are on shard.

This demonstration illustrates how range-based sharding distributes records across shards according to ordered intervals of the shard key. By defining a boundary at *customerId* = 1000, the data was effectively divided so that lower identifiers were routed to one shard and higher identifiers to another. This setup provides a clear example of how queries targeting specific ranges can be efficiently directed to the relevant shard, reducing the need to scan the entire dataset.

7 Conclusion

In conclusion, this dissertation has addressed the growing challenges of database scalability in the context of ever-expanding data volumes and increasing application demands. While scalability can be approached through various architectural techniques, this study has focused specifically on database sharding as a primary mechanism for achieving horizontal scalability.

Guided by the research goals, three key contributions emerge:

1. On scalability foundations, this dissertation clarified how sharding strategies conceptually address the limitations of monolithic database systems. By synthesizing insights from literature and practice, it has shown how sharding enables systems to distribute workload and maintain performance under demanding conditions.

2. On sharding strategies and their practical applications, this work provided a structured, tutorial-style overview of relational and non-relational approaches, with PostgreSQL (via Citus) and MongoDB serving as illustrative examples. This fulfills the objective of presenting practitioners with a clear and accessible guide for understanding how sharding can be applied across different database paradigms and domains.

3. On trade-offs and challenges, the dissertation highlighted the complexities of sharding, including the absence of foreign keys across shards, the overhead of cross-shard queries, and the operational difficulties of rebalancing. By framing these as conceptual trade-offs, it offers decision-makers a balanced perspective for evaluating when and how sharding is most appropriate.

Taken together, these findings achieve the intended goals: to explain the role of sharding in scalability, to organize the main strategies in a structured manner, and to evaluate their inherent trade-offs. Rather than providing empirical benchmarks, this dissertation contributes as a guide and reference framework, equipping developers, architects, and researchers with the knowledge necessary to make informed architectural decisions.

Ultimately, the work underscores that there is no “one-size-fits-all” solution. Instead, the effectiveness of sharding depends on aligning the chosen strategy with the domain’s workload patterns, data characteristics, and long-term operational needs. By framing sharding in this way,

the dissertation provides a foundation for building scalable, resilient, and adaptive database systems in both relational and non-relational contexts.

7.1 Next steps

While this dissertation has provided a conceptual and tutorial-oriented exploration of database sharding, including a simple implementation to demonstrate shard creation, data distribution, and query routing, several opportunities remain to expand this work. The following next steps could form the basis for future research or development:

- Scaling the implementation: A natural continuation would be to deploy a larger-scale experimental environment capable of handling datasets on the order of tens of millions of records. This would enable empirical benchmarking of sharding performance, including query response times, throughput, and rebalancing efficiency.
- Advanced sharding strategies: The present study focused primarily on range and hash-based sharding. Subsequent research could examine hybrid approaches (e.g., combining hash and range), directory-based sharding, or geo-sharding in more detail, particularly in compliance-driven environments.
- Workload-specific optimizations: Another avenue involves tailoring shard key design and routing strategies to specific workloads, such as analytical queries, time-series data, or high-write environments. This would test how theoretical principles translate to different real-world application domains.
- Operational concerns: Finally, future efforts could investigate practical aspects of sharding in production, including monitoring, fault tolerance, shard rebalancing, and schema evolution. These are critical for organizations intending to adopt sharding in mission-critical applications.

The suggested next steps highlight how future research and experimentation can move from theoretical guidance and small-scale demonstrations to empirically validated, production-ready sharding solutions.

References

Abadi, D. J. (2012). *Consistency Tradeoffs in Modern Distributed Database System Design*.

Abdelhafiz, B. (2021). *Sharding Database for Fault Tolerance and Scalability of Data*. 2nd International Conference on Computation, Automation and Knowledge Management (ICCAKM). https://ieeexplore.ieee.org/document/9357711

Akka. (2025). *Benchmarking database sharding in Akka*. https://akka.io/blog/benchmarking-database-sharding-in-akka

Amazon Web Services. (2020). *Reddit Migrates to Managed Amazon Aurora to Scale for 30% Year-over-Year Growth*. https://aws.amazon.com/solutions/case-studies/reddit-aurora-case-study/

Amazon Web Services. (2024a). *Compute Service - Amazon EC2*.

Amazon Web Services. (2024b). *Monitoring tools for Amazon Aurora*. https://aws.amazon.com/blogs/database/sharding-with-amazon-relational-database-service/

Amazon Web Services. (2024c). *What is Amazon Aurora*.

Amazon Web Services. (2024d). *What is Amazon S3*. https://aws.amazon.com/s3/

Amazon Web Services. (2024e). *What is AWS Lambda*. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

Amazon Web Services, Inc. (2023). *What is database sharding*. https://aws.amazon.com/pt/what-is/database-sharding/?nc1=h_ls

Aslan, M. (2023, April 3). *Sharding Strategies*. https://medium.com/@murataslan1/sharding-startegies-ba3df1df663d

Bailey, J. (2024, February 14). *What is Database Sharding? An Architecture Pattern for Increased Database Performance*. https://www.pingcap.com/blog/database-sharding-defined/

Basili, V., Caldiera, G., & Rombach, H. D. (1992). *Software Modeling and Measurement: The Goal Question Metric Paradigm*. Institute for Advanced Computer Studies Department of Computer Science University Of Maryland College Park, Maryland.

Bhat, T., & Chandrasekaran. (2015). *Sharding distributed social databases using social network analysis*.

Bondi, A. B. (2000, September). Characteristics of Scalability and Their Impact on Performance. *Proceedings of the 2nd International Workshop on Software and Performance*.

Britannica. (2025). *Percentile*. https://www.britannica.com/topic/percentile

CitusData. (2022). *Citus 11.1 shards your Postgres tables without interruption*. https://www.citusdata.com/blog/2022/09/19/citus-11-1-shards-postgres-tables-without-interruption?utm_source=chatgpt.com

Cockroach Labs. (2025). *Architecture Overview*. https://www.cockroachlabs.com/docs/stable/architecture/overview.html

Cubukcu, U., Erdogan, O., Pathak, S., Sannakkayala, S., & Slot, M. (2021). *Citus: Distributed PostgreSQL for Data-Intensive Applications*. https://paper-notes.zhjwpku.com/assets/pdfs/citus.pdf

Dgraph Labs. (2024, July 18). *What is Database Clustering* . https://dgraph.io/blog/post/clustering-database/

Gough, D., Oliver, S., & Thomas, J. (2017). *An Introduction to Systematic Reviews* (2nd ed.). Sage.

Guevara, H. (2024, July 8). *Sharding strategies: directory-based, range-based, and hash-based*. https://planetscale.com/blog/types-of-sharding

Hazelcast. (2025). *What is database sharding*. https://hazelcast.com/foundations/distributed-computing/sharding/

Hellerstein, J. M., Stonebraker, M., & Hamilton, J. (2007). *Architecture of a Database System*.

Hennessy, J. L., & Patterson, D. A. (2020). *Computer Architecture: A Quantitative Approach* (6th ed.).

Hiren Dhaduk. (2022, November 21). *5 Challenges of Database scalability*. https://medium.com/@HirenDhaduk1/5-challenges-of-data-scalability-be23b74f90d1

Keep, M., & Ingo, H. (2020, February 18). *Performance Best Practices: Sharding*.

Kitchenham, B., & Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering. *EBSE Technical Report*, *Technical Report EBSE 2007-001*.

Kleppmann, M. (2017). *Designing Data-Intensive Applications*.

Kossman, D., Kraska, T., & Loesing, S. (2010). An evaluation of alternative architectures for transaction processing in the cloud. *2010 ACM SIGMOD International Conference on Management of Data*.

Kraska, T., Pang, G., Franklin, M., Madden, S., & Fekete, A. (2013). MDCC: Multi-Data Center Consistency. *8th ACM European Conference on Computer Systems*.

Krusche & Company. (2024). *MongoDB sharding case study: How We Saved Our Client €100k A Year*. https://kruschecompany.com/mongodb-sharding-cluster-configuration/

Lakshman, A., & Malik, P. (2010). *Cassandra - A Decentralized Structured Storage System*. https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf

MongoDB. (2023). *What is Mongo DB*. https://www.mongodb.com/pt-br/what-is-mongodb

MongoDB. (2024). *Customer Success Stories*. https://www.mongodb.com/solutions/customer-case-studies

MongoDB. (2025a). *Create Ranges in a Sharded Cluster*.

MongoDB. (2025b). *Sharding in MongoDB*.

Oracle. (2024). *Oracle Sharding Overview*. https://docs.oracle.com/en/database/oracle/oracle-database/19/shard/sharding-overview.html

Oracle. (2025). *Request Routing in a Sharded Database Environment*. https://docs.oracle.com/en/database/oracle/oracle-database/18/shard/sharding-data-routing.html?utm_source=chatgpt.com

Özsu, M. T., & Valudirez, P. (2014). *Principles of Distributed Database Systems* (3rd ed.). https://vulms.vu.edu.pk/Courses/CS712/Downloads/Principles%20of%20Distributed%20Database%20Systems.pdf

Pachot, F. (2024, June 11). *Scalable range sharding to avoid hotspots on indexes* .

Pathak, S. (2021, July 10). *MongoDB : About & Case-study*. https://www.linkedin.com/pulse/mongodb-case-study-shashwat-pathak/

Pavlo, A., Angulo, G., Arulraj, J., Lin, H., & Ma, L. (2012). *A comparison of approaches to large-scale data management*. https://www.cs.cmu.edu/~pavlo/papers/benchmarks-sigmod09.pdf

Pedamkar, P. (2023, June 23). *SQL Cluster*. https://www.educba.com/sql-cluster/

Petticrew, M., & Roberts, H. (2006). *Systematic Reviews in the Social Sciences: Why and How*. Blackwell Publishing.

PlanetScale. (2023, June 30). *Sharding vs Partitioning: What+s the difference?*

PostgreSQL. (n.d.). *PostgreSQL 64.3 Extensibility*. Retrieved August 7, 2025, from https://www.postgresql.org/docs/11/gist-extensibility.html?utm_source=chatgpt.com

PostgreSQL. (2024). *What is PostgreSQL*. https://www.postgresql.org/about/

Rajaraman, A., Ullman, J. D., & Leskovec, J. (2015). *Mining of Massive Datasets* (2nd ed.).

Reselman, B. (2021). *The pros and cons of the Sharding architecture pattern*. https://www.redhat.com/architect/pros-and-cons-sharding

RisingWave. (2024). *Key Challenges and Solutions for Database Scalability*. https://risingwave.com/blog/key-challenges-and-solutions-for-database-scalability/

Rongali, N. (2025, August 4). *Partitioning, Sharding, Distribution, Hashing, Clustering & Buckets: Clarifying the Confusion*. ACM SIGMOD International Conference on Management of Data.

Sikkayan, B. (2025, March 26). *Avoid Cross-Shard Data Movement in Distributed Databases*.

Silberschatz, A., F. Korth, H., & Sudarshan, S. (2020). *Database System Concepts* (7h ed.). McGraw-Hill.

Solat, S. (2024). *Sharding Distributed Databases: A Critical Review*.

Statista. (2024). *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2023, with forecasts from 2024 to 2028* . https://www.statista.com/statistics/871513/worldwide-data-created/

Stonebreaker, M. (2005). *One size fits all does not work for database systems*.

Tahir, A. (2024, January 15). *Disadvantages of Sharding/ Partitioning*. https://medium.com/@abdullah.tahir_45158/database-management-w-sharding-partitioning-30359797e0f2

TechTarget. (2022, November). *Zettabyte*. https://www.techtarget.com/searchstorage/definition/zettabyte

Timescale. (2025). *Database partitioning : What it is and why it matters*. https://www.timescale.com/learn/data-partitioning-what-it-is-and-why-it-matters

uberall. (2024). *The ultimate guide to Foursquare: Everything you need to know*. https://uberall.com/en-us/directory/foursquare

Valle, G. (2024). *Exploring Effective Sharding Strategies with PostgreSQL*. https://medium.com/@gustavo.vallerp26/exploring-effective-sharding-strategies-with-postgresql-for-scalable-data-management-2c9ae7ef1759

Veeraragavan, N. R., & Zhang Kaiwen. (2020). *A position paper on GDPR compliance in sharded blockchains: rehash of old ideas or new interesting challenges?*

Vitess.io. (2025, January 8). *Global routing*. https://vitess.io/docs/22.0/reference/features/global-routing/

White, E. (2024). *The most popular databases in 2024*. https://www.bairesdev.com/blog/most-popular-databases/

Yun, C. (2024, October 31). *Amazon Aurora PostgreSQL Limitless Database is now generally available*. https://aws.amazon.com/blogs/aws/amazon-aurora-postgresql-limitless-database-is-now-generally-available/?utm_source=chatgpt.com