UNIVERSITY OF TURKU

Relational Database Robustness, Performance and Security

Software Engineering

Master's Degree Programme in Information and Communication Technology

Department of Computing, Faculty of Technology

Master of Science in Technology Thesis

Author:

Jonne Kiukas

Supervisor:

Ville Leppänen (University of Turku)

October 2024

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin Originality Check service.

Master of Science in Technology Thesis
Department of Computing, Faculty of Technology
University of Turku

Subject: Software Engineering
Programme: Master’s Degree Programme in Information and Communication Technology
Author: Jonne Kiukas
Title: Relational Database Robustness, Performance and Security
Number of pages: 57 pages
Date: October 2024

This thesis examines relational databases, which are foundational to modern information systems due to their structured design and ability to store, retrieve, and manage data. The thesis focuses on three important aspects of relational databases, robustness, performance, and security. The goal is to identify the key characteristics that define a robust database, analyze factors affecting performance, and explore potential security threats and prevention measures.

To assess robustness, the thesis identifies transaction management, data availability, ACID properties, concurrency control, deadlock prevention, and other characteristics as crucial factors. Performance is evaluated by analyzing hardware resources, network efficiency, use of indexes, and query optimization techniques. For security, the study highlights common threats such as SQL injection, unauthorized access, and employee misuse, alongside preventive measures like prepared statements, access control, encryption, auditing and penetration testing. The thesis involves tables that summarize findings and provides answers to the research questions.

The findings presented in this thesis are designed to serve as a guide for decision-makers in evaluating and selecting the appropriate database for their organization. With the help of this thesis the company can make the right adjustments that increase the customer and user satisfaction.

Keywords: relational database, database management system, robustness, performance, security.

Table of contents

1 Introduction 1

1.1 Research questions 1

1.2 Structure of the thesis 2

2 Research method 3

3 Database robustness 5

3.1 Defining database and database management system 5

3.2 Basics of transaction management 6

3.3 Concurrency control 10

3.3.1 Concurrency control anomalies 10

3.3.2 Locking-based concurrency control technique 14

3.4 Characteristics of robust database 17

3.5 Summarizing the characteristics of robust database 19

4 Database performance 21

4.1 Hardware 22

4.2 Network 25

4.3 Indexes 26

4.3.1 Using an index to access the data 28

4.3.2 Things to consider before implementing indexes 31

4.4 Query structure 32

4.5 Summarizing database performance factors 34

5 Database security 36

5.1 Basic security principles 36

5.2 Database security threats 38

5.2.1 Internal security threats 39

5.2.2 External security threats 40

5.3 Privileges and access control mechanisms 44

5.4 Encryption 46

5.5 Auditing database 47

5.6 Penetration testing 49

5.7 Summarizing database security 51

6 Discussion 54

7 Conclusion 57

References 58

1 Introduction

In this thesis, the focus is on relational databases. Relational databases form the backbone of modern information systems, allowing organized data storing, retrieval, and management across various industries. Relational databases' structured design, which is based on tables and relationships, has made them a preferred choice, and they are widely used, from small-scale applications to large enterprise systems. Almost every organization relies on data nowadays, and data has become an essential thing that enables business growth.

However, as the amount of data continues to grow, the robustness, performance, and security of relational databases have become aspects that need to be taken into consideration, so that business growth is guaranteed in the future. Robustness is essential to ensure that databases can withstand hardware failures, software errors, and unexpected disruptions without losing or corrupting data. Performance becomes crucial as businesses demand real-time responses to queries, even under high transaction volumes due to concurrent access or when handling large datasets. Security is important in an era where data breaches, cyberattacks, and unauthorized access can have devastating consequences for businesses and individuals. Without addressing these aspects, relational databases could fail to meet the high demands of modern applications, which could lead to companies losing their reputability, reliability, and eventually losing their business completely.

The findings of this thesis can help companies select the right relational database for their needs. The thesis serves as a guide and provides information about database robustness, performance, and security for the reader. After reading this thesis, the reader will have a better understanding of the different characteristics that make a database a robust system capable of handling situations that could prevent it from functioning as expected. The performance section of this thesis provides information about how a database system can handle larger datasets and execute queries faster. In addition, the reader is provided with knowledge about the things related to database security that helps the company to keep their data safe. Database robustness, performance, and security are all aspects that a company should take into consideration when selecting a database for their use case. Neglecting them can have far-reaching negative consequences that may drive customers away for good.

1.1 Research questions

- RQ1: What are the characteristics that define a robust database?
- RQ2: What are the factors that affect database performance?
- RQ3: What security threats can a database face, and how to prevent them?

1.2 Structure of the thesis

Chapter 2 explains how references were found for this thesis. It covers the different digital services and search words used to find relevant material. Chapter 2 also describes the process of selecting references for the thesis. Chapter 3 focuses on database robustness and explains the characteristics considered essential for ensuring that users can trust the database, knowing that any disruptions will not cause harm to the data. Throughout Chapter 3, there has been used an entity called "Robustness Characteristic" to summarize the recognized key characteristics of a robust database. The entity is abbreviated into the form RC, and each entity has been uniquely numbered to identify them. At the end of Chapter 3, there is a table that lists all the RC entities and provides an answer to RQ1.

Chapter 4 focuses on database performance, providing information about the factors that can slow down query execution time and explaining what can be done to reduce it, thereby improving overall performance. In Chapter 4, there has been used an entity called "Performance Factor" to summarize the factors that affect database performance. This entity has been abbreviated as PF and each entity has been uniquely numbered for identification purposes. At the end of Chapter 4, there is a table that lists all the PF entities and provides an answer to RQ2.

Chapter 5 describes what kind of threats a database can face which may eventually lead to unauthorized access and data leakage. Countermeasure actions are suggested to be implemented to keep people away from the data, who are not allowed to access the database. In Chapter 5, there have been used two entities called "Security Threat" and "Security Threat Prevention". The "Security Threat" entity, abbreviated as ST, summarizes various threats that can compromise database security. The "Security Threat Prevention" entity, abbreviated as STP, provides insight into how to prevent these security threats. Both the ST and STP entities have been uniquely numbered. A table at the end of Chapter 5 lists all the ST entities, and another table lists all the STP entities. Together, these tables provide an answer to RQ3.

Chapter 6 discusses how robustness, performance, and security are related to each other. The purpose is to provide an understanding that these three aspects form a basis of a reliable database system, and companies should consider them carefully when choosing a database for their use case. Finally, Chapter 7 contains the conclusion of this thesis, and it summarizes the findings.

2 Research method

Various references from literature and research articles, related to the topic of this thesis, have been used. References were found online from the Volter database, which contains electronic books and scientific papers maintained by the University of Turku, Google Scholar, and Elsevier's ScienceDirect database for technical and scientific research. In addition, O'Reilly Media's electronic books were used to find suitable references.

The process of finding suitable references from the previously mentioned services consisted of three steps. The first step was to use each service's search functionality with search words that are related to the thesis topic. The search words used to find references for RQ1 were: "Robust database", "Designing a robust database", "Implementing a robust database", "Ensuring database robustness", and "Defining a robust database". The search words used to find references for RQ2 were: "Database performance", "Ensuring database performance", "Improving database performance", "Improving query execution", "Database query optimization" and "Optimizing database queries for better performance". The search words used to find references for RQ3 were: "Database security", "Security in database", "Ensuring database security", "Database security threats" and "Preventing database security threats". To narrow down the vast amount of search results, different kinds of filters, which are built into these services, were used. This included filters that filter the results based on the fact is the result a book or a research article, and to which subject area the result belongs to.

The second step was to go through the search results from each service, starting from reading the title of a book or a research article and deciding whether the search result is interesting or not. If the search result was interesting enough for further investigation, it was opened as a new tab, and if it was not considered as an interesting result, then it was left behind. Once the interesting search results were found from one service, they were investigated more. In the new tab, whether it was the book's table of contents, introduction chapter, or research article's abstract, the text was studied to gain a deeper understanding of what the whole book or research article was about. If it seemed that the book or research article is not related to any of the topics that cover this thesis and thus could not provide any valuable information, then the opened tab was closed, and it was proceeded to the next possible reference. Based on the table of contents, the introduction chapter, or the abstract, if it seemed that further reading could result in finding something that could be used in this thesis, then a link to this specific reference was saved, so that the content could have been accessed later. All the saved links were added to a Microsoft Word file.

The third step was to access the saved links one by one and read more content from the books and the research articles. From all the books, the table of content was earlier studied, and it helped to find the

chapters that seemed to be interesting. However, if none of the chapters in the book provided useful information, then a mark was added next to the link to indicate that the reference cannot be used in this thesis. Then again, if the book was used as a reference, necessary information and an access link to this specific book were added in the reference section of this thesis. Similar procedure was used with the research articles to evaluate whether a specific reference can be used in this thesis or not.

This is how different kinds of references for this thesis were found from different services that provide electronic books and research articles. The total number of possible references that were found from all services was 116. After going through all of these possible references and considering whether the content matches with the thesis topics, the final number of used references was 45.

3 Database robustness

In a study [1], software robustness has been explained using the IEEE standard glossary of software engineering terminology, which states that robustness is the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. Researchers also say that robustness is a quality attribute for achieving higher dependability in a system. This means that the system should function in an acceptable way and deliver its originally intended behavior, despite exceptional or unforeseen operating conditions.

In another study [2], software robustness is defined as the ability to detect unexpected inputs or events during execution. These unexpected inputs or events are also known as anomalies [2]. Robustness focuses on the states and events that should not happen in a system, rather than on how the system should function in ordinary cases [1]. A database can encounter states and events, such as various anomalies and deadlocks, which may affect its operation. These are issues that should not occur, and to be considered a robust system, the database environment must handle them properly.

3.1 Defining database and database management system

In its simplicity, a database is a tool that holds data and allows users to create, read, update, and delete data in some manner. A database is a collection of interrelated data, and the data in a database are organized into related tables that can be joined to provide information to users. To be considered a database, the place where the data are stored must contain not only the data, but also information about the relationships between those data. An example of a relationship between data would be that customers of a store are related to the orders they have placed. The database serves as central storage for the data, and the idea is that a user, either a person or an application program, does not need to worry about how the data are stored on a disk. When the user wants to query the database, software called a database management system (DBMS) works between the user's request and the data storage. [3] [4] [5]

It was mentioned that data have relationships, and that data are organized into tables that are related to each other. This is a defining feature in relational databases. This thesis focuses on relational databases, and throughout the thesis, any reference to a database or database management system specifically refers to a relational database and relational database management system. However, it is good to acknowledge that there are other databases available for storing data, than just relational databases. These databases have recognizable differences, and they can be used for different use cases. For example, a NoSQL database is an alternative for relational databases. NoSQL stands for "Not Only SQL", and compared to relational databases, it provides a more scalable solution for Big Data

applications. Another option for relational databases could be a NewSQL database. The NewSQL database tries to make a combination of a relational database and the NoSQL database. The aim is to make a relational database more scalable, while still providing ACID properties, which the NoSQL does not provide. [6] ACID is an acronym that is related to database transactions. ACID stands for atomicity, consistency, isolation, and durability. These characteristics of ACID are explained in more detail later in this thesis.

The DBMS is a tool that locates where the data are stored and performs the actual data storing. The database is just a tool that keeps the data inside of it. DBMSs are a set of programs that manages incoming data, organizes it, and provides ways for the data to be modified. DBMS allows users and other programs to create and access data in a database and run queries on data, so that data can be extracted. Queries are instructions for the DBMS, so that the required data can be returned to a user. These queries are written in a language called Structured Query Language (SQL), and SQL queries are specifically used to extract data from the relational database. When the user sends a query to the DBMS, it performs a functionality called query processing, and the objective is to fetch the queried data from the database. The DBMS has other functionalities than just query processing such as application interface, security control, and consistency control. The application interface level manages interfaces to the user's applications. The security control level is responsible for maintaining data integrity and authorization checking. The consistency control level ensures the correctness of the data despite concurrent data access. [4] [5] [7]

The primary goal of a DBMS is to integrate organizational data into a centralized repository, while ensuring secure and shared access to this data. By centralizing data, a DBMS effectively reduces data redundancy, which in turn maintains data consistency. One significant advantage of using a DBMS is that it offers data independence. This independence is achieved through the system's three-tier architecture. Within the relational data model, the database is presented as a collection of relations. These relations are essentially tables, where each row represents a unique record, and each column represents a specific attribute. Attributes can appear in any order within the table. Users and programs do not have to understand where the data are physically located on a disk or who else may be accessing the data, because the DBMS does all of this when database requests are managed. When the DBMS handles the database requests, it ensures the integrity of the data, meaning that the DBMS makes sure that the data are available and are organized as intended. [4] [5]

3.2 Basics of transaction management

Transaction management in a database system handles the issues of preserving consistent data despite system failures and concurrent data access. A transaction is a set of operations that perform a single

logical unit of work and is a task that a user submits to a database. A transaction can range from a single interactive SQL command to multiple SQL commands. In multiuser databases, transactions are important because they must either be completely successful or entirely fail. These are the only two ways how a transaction can end. If the transaction is successful, then the made changes are committed and stored in the database permanently. If it happens that the transaction fails, all the changes made by the transaction are undone, and the database is restored back to the state where it was before the transaction was started. This process is known as a rollback. [3] [5]

RC1: Transaction management. To handle concurrent data access and system failures, the DBMS needs to manage transactions. Transaction management preserves the data in a consistent state and makes the database a robust system.

The fundamental rule is that every transaction must terminate, and a transaction either succeeds or fails. If a transaction executes successfully, it is committed, and the result will be stored in the database permanently, which cannot be undone. After this, the database moves from one consistent state to another. This consistency must be ensured regardless of whether transactions are executed successfully or if they fail. A transaction can be said to be a unit of consistency and reliability. Reliability means that the DBMS is resilient to various types of failures and can recover from them. Resiliency, on the other hand, makes the DBMS a robust system that continues its normal functioning despite the occurred failure. A robust DBMS always keeps the database in a consistent state, either by rolling back, in the case of a failing transaction, to a previous consistent state, or by ensuring that it is safe to commit the transaction and move forward to a new consistent state. [5]

RC2: Transaction completion. Transactions must end either being successful or failing. When transactions end only in two ways, it ensures the robustness and consistency of the database.

When the DBMS is resilient and recovers from failures, it retains the availability of data. Availability means that the DBMS continues its normal functioning so that users can execute transactions in a database. Failures can occur before the transaction has been executed, and when failures have been repaired, the DBMS must be available to continue its normal operation. [5]

RC3: Data availability. When the data is available for users, it means that the DBMS is operating robustly as supposed, allowing users to run transactions to interact with the data.

A transaction might fail for a variety of reasons. If committing a transaction takes too much time, it can lead to a timeout, causing the execution of a transaction to be interrupted. Hardware malfunctions or software bugs, either in the database or in the operating system, may result in memory loss, forcing

transaction processing to stop. A transaction can also fail if the network connection between the user and the database goes down, or if the server running the database crashes. Other issues that can lead to an undesirable system state and cause transactions to fail include bad input data, processor failures, data not found, overflow, resource limit exceeded, and power failures. [3] [5]

A well-functioning DBMS executes ACID transactions. An ACID transaction has four features that keep the DBMS operating consistently and reliably. These four features are atomicity, consistency, isolation, and durability. An essential feature of ACID is that it leaves the database in such a state that allows a transaction to be executed, and in case of failure, the transaction can be safely retried. If the database is in danger of violation due to a failure, ACID ensures that the transaction is aborted entirely, and no half-finished operations are allowed into the database. [3] [5] [8]

In general, atomic refers to something that cannot be broken down into smaller parts. In ACID, atomicity means that a transaction is a single logical unit of work in a database environment that either completely succeeds or completely fails. This property is known as the all-or-nothing property of a transaction. Transactions are never left partially completed, and a DBMS with atomic transactions will always rollback in case of failure, regardless of what caused it. The DBMS is responsible for maintaining atomicity even when failures occur. While a transaction is in the process of execution and a failure occurs, the DBMS must discard any changes that the transaction has made and perform what needs to be done to recover from the failed state. After this, the user or the application programs can be sure that the transaction did not change anything, and it is safe to retry the transaction. The ability to abort a transaction on error and have all writes from a transaction discarded is the defining feature of atomicity. [3] [5] [8] [9]

Consistency demands that the database must remain in a consistent state after a transaction has been executed. The database remains in a consistent state when every transaction follows the defined constraints. If the operations within a transaction would violate the database's rules, the transaction is rolled back, maintaining consistency before and after the transaction. This means that at the beginning of a new transaction, users or application programs can be confident that the database is in a consistent state. For example, assume that a bank account has a balance of 75 €. A user starts a transaction that would transfer 100 € from his or her bank account to another bank account. This would put the user's bank account balance 25 € below 0 €, violating the database's rule that says a bank account cannot make a payment resulting in a balance less than 0 €. In such a scenario, the transaction is rolled back, and the database remains consistent. [3] [5] [9]

Isolation ensures that the details of a transaction are hidden from everyone except the person making the transaction, maintaining the independence of each transaction. In other words, in a database

environment, transactions are isolated from each other, and they do not depend on each other. The rule with isolation is that the partial effects of incomplete transactions should not be visible to other users. For example, assume that a user transfers 100 € from his or her account to another bank account. A third-party user cannot peek at the database and see a state where neither the user nor the receiving bank account has the 100 € while the transaction is processing. Anyone who is looking at the database will see the 100 € either in the user's account before the transaction or in another bank account after the transaction. Thus, no one can read or modify data that is being modified by another transaction. [5] [9]

Concurrency control has an important role in ensuring that multiple transactions running at the same time are isolated from one another. This means that when two transactions run concurrently, the outcome should be the same as if one transaction had run first in its entirety, followed by another. In particular, two transactions operate in isolation and cannot interfere with each other. Assume one transaction transfers 100 € from the first user's bank account to the second user's bank account, and then another transaction transfers 100 € from the second user's bank account to the third user's bank account. Obviously, one of these transactions must occur first and finish before the other starts. When the second transaction starts, the missing 100 € from the first user's bank account cannot be seen unless it is already in the second user's bank account. [3] [9]

Durability in ACID means that once a transaction is committed, changes are permanent, and they cannot be rolled back. Committed transactions cannot disappear later from the database. This means that in case of power failure or another issue that causes the database to go down, the effects of the transaction will still be there after the database is restarted. The durability property ensures that after the transaction is committed successfully, the changes are durable and cannot be erased from the database. The durability requirement relies on the consistency rule. Consistency ensures that a transaction will not be finalized if it would leave the database in a state that breaks its rules. Durability guarantees that the database will not later decide that the transaction caused such a state that violated database's rule and then retroactively delete the transaction. Once the transaction is committed, it is final. [3] [9]

RC4: ACID transactions. The ACID's four features — atomicity, consistency, isolation, and durability — ensure that a database remains a robust system by committing only valid data from successful transactions, processing every transaction independently, and preserving the committed data in the database, even in the case of database failure.

3.3 Concurrency control

A database can be accessed by thousands of users at one time, meaning that the database is used concurrently. It is called a multiuser database environment whenever two or more users are interacting with the same database at the same time. When the database is accessed by multiple users, and they perform queries on the same data simultaneously, at a frequent pace, it is called high concurrency. In turn, zero concurrency means that users do not read or write the same data at the same time. It is also zero concurrency if users read or write the same data at different times. Zero concurrency does not cause conflicts within the database, but, when it comes to high concurrency, it requires actions. It is the responsibility of DBMS to separate the actions of one user from another and to ensure the integrity of the database while multiple users are accessing and performing operations on the same data. [3] [10]

RC5: Concurrent access. The database is a robust system if multiple users' transactions are handled so that the database's integrity is preserved. This means maintaining the correctness and reliability of the data within the database, so that users can count on the fact that their data is not compromised due to concurrent access.

To coordinate concurrent accesses and to allow users to access a database simultaneously, while preserving the consistency of the data, it requires a mechanism called concurrency control. If two transactions are not accessing the same data, they can be executed concurrently because they are not depending on each other. Concurrency issues appear when one transaction reads data that is modified simultaneously by another transaction, or when two transactions try to modify the same data at the same time. From the technical point of view, the hard part of achieving concurrency control is the necessity to prevent database updates performed by one user from interfering with database retrievals and updates performed by other users. When the database is accessed concurrently by multiple users and the DBMS is lacking a concurrency control mechanism, different kinds of anomalies can arise. [3] [5] [8]

3.3.1 Concurrency control anomalies

Lost update, or dirty write anomaly, can occur when a successfully completed update operation made by a user is overridden by another user. Assume that a user is withdrawing 500 € from his or her bank account that has a balance of 2000 €, while another user is depositing an amount of 600 € into the same bank account. These two transactions are performed at the same time, and because of the concurrent access to the same data, when the first transaction decrements the value of the bank account balance from 2000 € to 1500 €, the second transaction increments the bank account balance from 2000 € to 2600 €. What happened was that the second transaction overrode the first transaction,

resulting in an incorrect balance. To prevent this anomaly from happening, transactions should execute serially, one following another. In the given example, the first transaction should have decremented the balance to 1500 €, and after this, the second transaction could have incremented the balance to 2100 €, resulting in the correct bank account balance. [5]

Figure 3.1 illustrates another example of a lost update or dirty write anomaly. Two transactions from different users, called Alice and Bob, are trying to update the same records simultaneously. Alice starts a transaction that updates the Listings database to set the buyer as “Alice” where id = 1234 and the Invoices database to set the recipient as “Alice” where listing_id = 1234. Concurrently, Bob starts his transaction updating the same listing with his information. However, when both transactions reach the commit point, a conflict occurs. The Listings database has “Bob” as the buyer for id = 1234, and the Invoices database has "Alice" as the recipient for listing_id = 1234. In this scenario, the sale is given to Bob because he performs the winning update to the listings table, but the invoice is sent to Alice because she performs the winning update to the invoices table. [8]

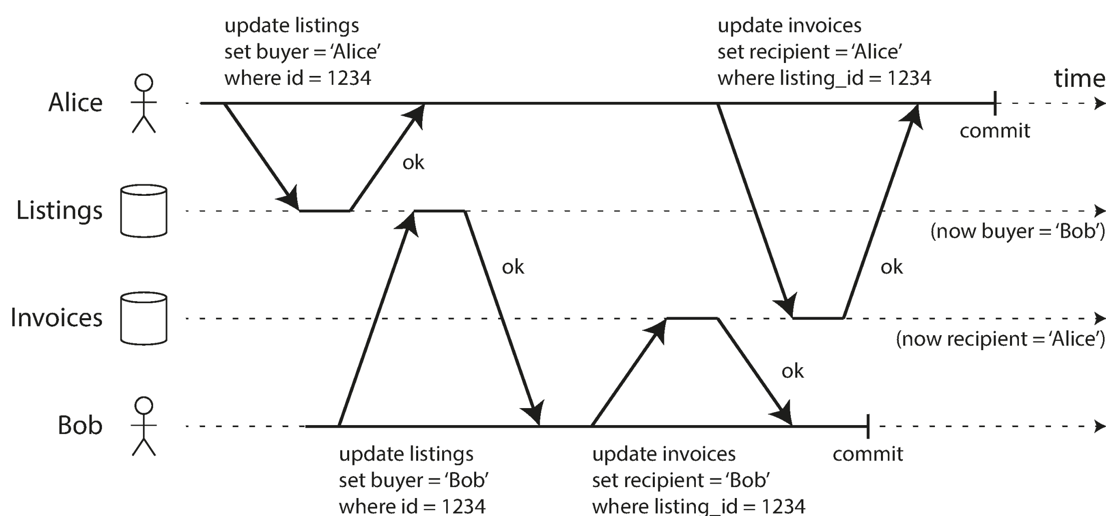


Figure 3.1: Dirty write anomaly causing conflicting writes from different transactions. [8]

Uncommitted dependency, or dirty read anomaly, occurs when other transactions can read data that was modified by one transaction but has not yet been committed. The issue arises if the transaction decides to cancel the changes, but other transactions have already made their own modifications based on the data before the transaction was cancelled and data rolled back to its original state. Assume that two transactions are concurrently accessing a database called Employee. One transaction updates the salary of an employee by 10 percent, and his or her base salary is 4000 €. Another transaction is calculating tax deduction for the same employee depending on his or her base salary. The first transaction updates the employee’s base salary from 4000 € to 4400 €, then allows the second transaction to read this new salary value before the first transaction has been committed, and finally

the first transaction is cancelled. After all this, the situation should end up with the base salary restored to its original value of 4000 €. However, the second transaction has read the new value of the base salary and thinks it is 4400 €. Based on this information, the second transaction calculates the tax deduction with the value of 4400 € instead of 4000 €. [5]

Figure 3.2 demonstrates another issue of uncommitted dependency, or dirty read anomaly. In Figure 3.2, User 1 starts a transaction intending to set the value of x to 3 in the database. While User 1's transaction is still in progress and has not yet been committed, User 2 starts interacting with the database and sends a request to get the value of x from the database. Because User 1's transaction has not been committed yet, the database returns the current committed value of x, which is 2. Finally, User 1's transaction is committed, and the value of x is updated to 3. However, now User 2 has the incorrect value of x because it is not 2 anymore. To receive the correct value of x, User 2 would need to perform the get operation again. [8]

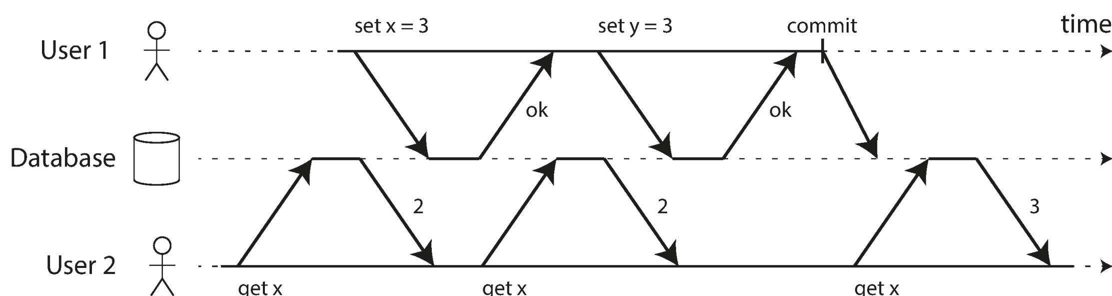


Figure 3.2: Example of dirty read anomaly, where User 2 can read the value of x while User 1's update transaction has not been committed yet. [8]

Inconsistent analysis anomaly occurs when a transaction reads several values from the database, but a second transaction updates some of them while the first transaction is still in the process of execution. Assume a banking system's database, in which two transactions are accessing concurrently. The first transaction is calculating the average balance of all customers, while the second transaction is depositing 1000 € into a customer's bank account at the same time. Due to this, the average balance calculated by the first transaction is incorrect, because the second transaction incremented and updated the balance of a particular customer while the first transaction was being executed. [5]

Fuzzy read or non-repeatable read anomaly occurs when one transaction reads the value of a data, and that same data is later updated or deleted by another transaction. The actual issue will take place if the first transaction attempts to re-read the data. In such a case, either the data may not be found, or it may have a different value. Assume that a transaction reads a salary of an employee from a database while another transaction is updating the salary of the same employee concurrently. If the first transaction

tries to re-read the employee's salary, a different value will be returned. Because another transaction updated the value of the salary, two read operations during the first transaction will return different values. Naturally, if another transaction would have deleted the data about the employee's salary, then in this situation, the first transaction would not have found the data when it attempts to re-read the employee's salary. [5]

Another example of a fuzzy read or non-repeatable read anomaly is presented in Figure 3.3. Alice has savings at a bank totalling 1000 €, divided into two bank accounts with 500 € each. A transaction is made to transfer 100 € from one bank account to the other. Alice queries the balance of her bank accounts at the same time as the transaction is still executing and sees one bank account balance at time before the incoming payment has arrived, and the other bank account after the outgoing transfer has been completed. To Alice, it looks like she only has a total of 900 € in both of her bank accounts, 500 € in one and 400 € in the other, and it seems like 100 € has disappeared. In fact, the 100 € has not disappeared anywhere because the transaction is still being executed while Alice checks the balance of her bank accounts. If Alice would look at her bank accounts after the transaction has been committed, she would notice that one of her bank accounts' balances is 600 €, thus resulting in a total balance of 1000 € in both of her bank accounts. [8]

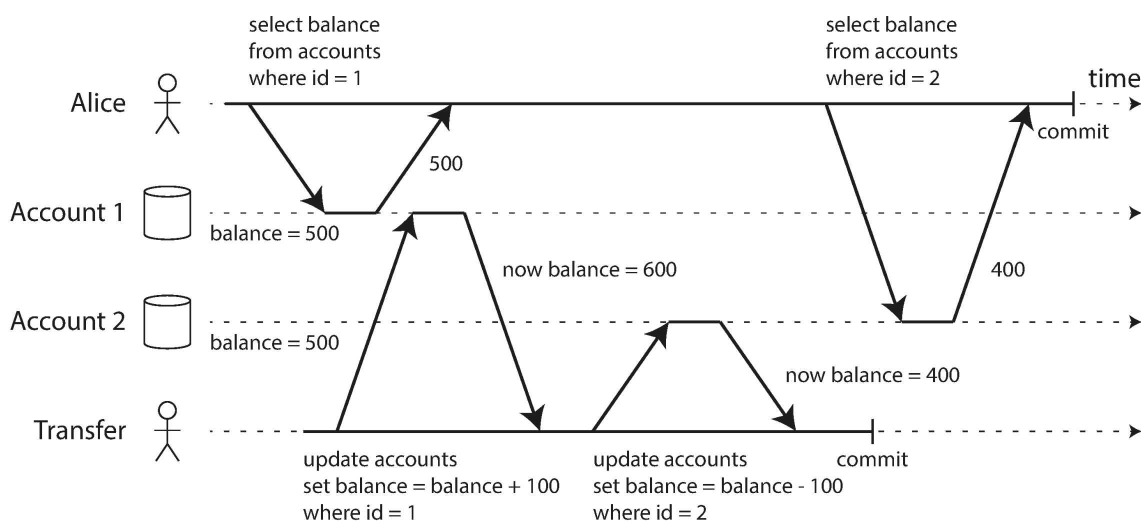


Figure 3.3: Example of non-repeatable read anomaly. [8]

Phantom read anomaly occurs when a transaction performs an operation that reads rows from a database, and then another transaction inserts new rows into the database. This will change the result if the first transaction re-reads the data. If the first transaction runs the same query again, it will see the new rows inserted by the other transaction. These new rows are called phantoms. Assume that in the employee database, one transaction retrieves all employees within a company, while another transaction inserts new employees into that company. If the first transaction re-reads the same data by

executing the previous query, then the retrieved data will involve additional phantom rows that have been inserted by another transaction concurrently. [5]

RC6: Concurrency control. With the help of concurrency control, the DBMS can prevent concurrency control anomalies from occurring within the database. Lost update, uncommitted dependency, inconsistent analysis, fuzzy read, and phantom read are all anomalies that should not happen and can be prevented with concurrency control.

3.3.2 Locking-based concurrency control technique

In a multiuser database environment, the challenge is to maximize concurrent access so that users can access and modify the data consistently at the same time. To address this challenge, a locking-based concurrency control technique is a widely used solution to manage conflicting operations and ensure the serializability of concurrent transactions. This is how a database can be preserved in a consistent state. The fundamental of this approach is that before a transaction can access data, it must acquire a lock on that particular data. Once the lock has been acquired, it prevents other transactions from modifying the data that the original transaction is accessing. In a single-user database, locks are not necessary because only one user is modifying the data. However, in a multiuser database environment, where several users are accessing and modifying data, it is essential to have a mechanism that prevents concurrent data modifications. Locking mechanisms are said to be a key feature of any multiuser database and read lock and write lock are two commonly used locking modes. [5] [11]

If a transaction has acquired a read lock on data, it can read the data but not update it. The primary goal of the locking-based concurrency control technique is to synchronize the conflicting transactions. Two read operations by different transactions are allowed to be performed because read-read operation is non-conflicting, meaning that the original transaction and other transactions can read the same data simultaneously. However, the original transaction is not allowed to modify the data, because doing so would break the consistency of the data, and other transactions might read incorrect data. [5]

If a transaction has acquired a write lock on data, it can read but also update the data. This means that other transactions are not allowed to read or update the data, as read-write and write-write operations are conflicting with each other. Other transactions are not allowed to read the data while it is being modified by the original transaction because it would result in incorrect or inconsistent data being read once the original transaction is committed. Additionally, other transactions are not permitted to modify the data because doing so would create a conflict with the original transaction and potentially lead to loss of consistency. Only one transaction can hold the lock on specific data. If another transaction

wants to read or modify the same data, it must wait until the first transaction is committed, and after that, it can acquire the lock and proceed with its operation. [5] [8]

Read lock and write lock are not the only locking techniques that can be used to handle simultaneous workloads and keep data consistent when multiple users query and modify it. Table locks and row locks are other solutions to ensure a well-functioning database. When a transaction acquires a table lock, it locks the entire table. This postpones other transactions with an intention to read and write the same data until the original transaction releases its lock. On the other hand, if a transaction has acquired a row lock, then a specific data row is locked while the transaction performs its operation. Compared to table locks, the use of row locks offers greater concurrency, allowing multiple transactions to edit different rows at the same time. The row locks reduce the chance that multiple transactions would have to compete for locks on the same resources. However, the downside of using row locks, compared to table locks, is that row locks degrade the performance of the database more. For example, in a situation where a transaction needs to access the table but only row locks have been used, the DBMS has to check the entire table for row locks that are held by other transactions. This will be time-consuming and performance-demanding, especially on larger tables. Therefore, the table lock offers better performance, but it does not offer as good concurrency as the row locks do, because while the entire table is locked, other transactions cannot access it during that time. However, it is noteworthy that every lock operation, including getting a lock, checking if there is a lock available and releasing a lock, consumes computer's resources. When these resources are used to managing locks instead of storing and retrieving data, it affects the performance negatively. [12] [13]

A lock manager in a DBMS is responsible for managing locks for different transactions, and a transaction manager is responsible for passing information to the lock manager that either a read or a write lock is needed. In addition, the lock manager is responsible for checking if the data is currently locked by another transaction or not. If the data is locked by another transaction and the existing locking mode is not compatible with the lock requested by a new transaction, the lock manager does not allow the new transaction to acquire the lock. In this situation, the execution of a new transaction is delayed until the existing lock is released. However, if the existing locking mode is compatible with the lock requested by the new transaction, the lock manager allows the new transaction to acquire the required lock. [5]

The synchronization of transaction executions in the DBMS cannot be ensured by implementing locks only in transactions. Example 3.1 below shows two transactions, T1 and T2.

T1:	T2:
Read(*a*);	Read(*a*);

$a := a + 2$;	$a := a*10$;
Write(a);	Write(a);
Read(b);	Read(b);
$b := b * 2$;	$b := b - 5$;
Write(b);	Write(b);
Commit;	Commit;

Example 3.1: Two transactions modifying the same values. [5]

Assuming that T1, in Example 3.1, acquires a lock on data *a* and releases the lock as soon as the transaction has been executed, however, the transaction T1 may acquire another lock on data item *b* after the lock on data *a* has been released. In this scenario, transaction T1 interferes with transaction T2, resulting in the loss of isolation and atomicity. Therefore, to guarantee that transactions are executed serially, the locking and releasing operations on data need to be synchronized. Two-phase locking (2PL) protocol is a known solution to achieve this. [5]

The 2PL protocol rules that a transaction should not acquire a lock after it has released one of its locks. The life cycle of a transaction is divided into two phases, which are the growing phase and shrinking phase. During the growing phase, a transaction can acquire locks on data and can access data, but locks cannot be released. Once the transaction moves to the shrinking phase, locks can be released, but new locks are not allowed to be acquired. Hence, the ending of the growing phase of a transaction determines when the shrinking phase begins. [5]

The 2PL protocol is also good for preventing a problem called cascading rollback. When a transaction releases a lock, the lock manager allows another transaction to acquire the lock. However, in a situation where the former transaction is cancelled and rolled back, the latter transaction must also be cancelled and rolled back so that the consistency of data is maintained. This leads into a series of rollbacks, where a transaction is cancelled one after another because the operation of other transactions depends on the original transaction. The 2PL protocol is a solution for cascading rollback when it is implemented in such a way that a transaction releases its locks only when the transaction has been committed. [5]

RC7: Two-phase locking. It is responsible for preventing conflicts between concurrent transactions, by ensuring that transactions are executed serially.

Even though the 2PL protocol helps to maintain serializability and prevent cascading rollbacks, its downside is that it can cause deadlocks. The growing phase of 2PL protocol allows transactions to acquire locks on data, and a deadlock occurs in a database when two or more transactions are unable to proceed because each transaction is waiting for the other to release locks on data that they need. For example, transaction A holds a lock on data 1 and needs a lock on data 2, while transaction B holds a lock on data 2 and needs a lock on data 1. In this situation, transactions are waiting endlessly for one another to release their locks, but neither will release their lock. Because of this deadlock situation, transactions cannot be committed, thus preventing the progress of the DBMS. [5]

To resolve deadlock situations, the DBMS should detect deadlocks between transactions and choose one of them to be a victim that is rolled back. When the execution of a victim transaction is cancelled, it releases the locks of the victim transaction, and this allows other transactions to make progress. Once the other transactions are either committed or rolled back, the victim transaction can be restarted. The selection of the victim transaction is done based on the number of data items each transaction holds. The transaction that has the fewest amount of data items in hold is chosen to be the victim. However, the problem with restarting the victim transaction is that it requires doing the work all over again, and this can be an additional performance problem. [5] [8]

RC8: Deadlock prevention. The DBMS must have the ability to prevent deadlocks from happening in the database. The 2PL protocol can cause deadlocks in the database, and when the DBMS is able to prevent this conflict between transactions, it proves to be a robust system.

3.4 Characteristics of robust database

The definition of software robustness in study [1] can be reflected to databases. If software robustness is the system's ability to function correctly in case of invalid inputs or stressful conditions, then this applies to databases. The DBMS handles transactions and does the necessary actions if a transaction fails. The transaction can fail because of several reasons, all of which are stressful conditions. The robustness of a database comes with the ability to recover from these stressful conditions. With the rollback mechanism, the failing transaction can be cancelled, and the database kept in a consistent state, so that accurate and reliable information can be provided to users without any data loss or corruption. On top of the recovery and consistency, availability makes the database dependable. When the data in the database is available to users despite the failure that has happened, and the data is kept in the same consistent state as it has always been, then higher dependability has been achieved, and the database can be characterized as a robust system that continues to deliver its intended behaviour.

Consistency is not the only defining characteristic of a robust database that is part of an ACID transaction. Other properties of ACID transactions are also closely related to what has been said about software robustness in research [1]. Robustness in software ensures that the system can handle unexpected conditions, and the atomicity in ACID transactions is responsible for handling situations that can cause the database to end up in an erroneous state. If a transaction fails due to invalid input or other unforeseen error, then the atomicity ensures that the transaction is entirely rolled back. Despite these stressful conditions, the database is left in such a state where it functions correctly and is able to deliver its originally intended behaviour. This means that even if the transaction happens to fail, a user can be confident that the database is in a state where it is safe to retry the transaction. When users are confident about using the database, they have the feeling of dependability, meaning that users trust that when they are interacting with the database, their performed actions are handled in an appropriate way.

Robust software, as defined in research [1], should handle stressful, exceptional, and unforeseen conditions. Based on this, the isolation in ACID transactions is also a defining characteristic of a robust database. When multiple users are performing concurrent transactions, it can put a significant load on the database, especially because concurrent transactions happen unexpectedly. Users will execute their transactions whenever they need to, and it is nearly impossible to predict when transactions will happen. Certain patterns can be recognized that could be used to estimate when there will be the most traffic towards the database. However, users can always diverge from these patterns and execute their transactions at a completely different time than before. In addition, these patterns do not take into consideration the possible growth of the transactions that could come if the number of users increases. This is why isolation is needed, to ensure that transactions do not interfere with each other. When conflicts between transactions are prevented, and the independence of each transaction is preserved, it is essential for the database's robustness.

In ACID transactions, durability aligns with the concept of software robustness, where the system must continue to operate correctly under stressful conditions. Durability guarantees that once a transaction has been committed, changes made to the data are permanent and cannot disappear, even if the database faces a failure. If the data disappears from the database for any reason other than a second transaction, then the database would not function in the intended way, meaning that it would not be a robust system. When the database is able to keep the data unchanged despite stressful conditions that could lead to data loss, that is when the database is functioning correctly and is a robust system.

When multiple users are performing transactions at the same time, the database's capacity utilization is high, meaning that the database is facing high concurrency. A robust database must be able to manage high concurrency, so that transactions are not conflicting with each other and are not causing

anomalies. Concurrency control anomalies, which can arise when transactions are executed at the same time, are, by definition, unexpected events that can be addressed through a concurrency control mechanism. The concurrency control mechanism is another characteristic of a robust database, as it makes the database function correctly in the presence of anomalies caused by high concurrency. High concurrency is truly a stressful condition, which the robust database needs to handle appropriately, so that conflicting transactions and different anomalies are prevented. The purpose is to ensure the flawless operation of the database in the presence of high concurrency, so that conflicting transactions do not compromise the database's overall functionality. When the robust database manages this, users can rely on the fact that their data is safe and everything is working as intended. So, a concurrency control mechanism ensures that the database maintains its expected behavior despite the high concurrency. This aligns with the definition of robustness in software, where the system is supposed to manage and recover from events that should not happen, ensuring the database still operates correctly under those stressful conditions.

Robustness in a database involves handling scenarios where multiple users interact with the same data concurrently. Interacting with the same data at the same time can potentially lead to different anomalies. It was previously said that these anomalies can be prevented with concurrency control mechanisms, but to be more specific, a locking-based concurrency control technique, which is the actual mechanism, is used to keep the database in a consistent state. The locking-based concurrency control technique can be further broken down into read locks, write locks, table locks, row locks, and two-phase locking, which ensures that transactions are executed serially, one by one, so that transactions do not conflict with each other, and anomalies do not occur. When transactions are executed serially, it maintains the correct order of transactions, and this makes the functioning of the database predictable. Users are expecting a certain outcome when they execute a transaction, and if the result is what was expected to happen, users' dependability towards the database increases. This means that the more the database manages to operate as the users have predicted — usually, users expect the database to successfully do what has been commanded — the more the users trust the database and will use it in the future.

3.5 Summarizing the characteristics of robust database

Throughout Chapter 3, different characteristics of a robust database were recognized. These characteristics were summarized into entities called Robustness Characteristics and uniquely numbered to identify them. Robustness Characteristic entities answer RQ1: What are the characteristics that define a robust database? In Table 3.1, all the Robustness Characteristic entities have been organized into one table to provide a clear understanding of the characteristics that a robust

database should have. The table includes the Robustness Characteristic entity number, the name of the characteristic, and a short description of each characteristic.

RC#	**Robustness Characteristic**	**Description**
RC1	Transaction management	Handles concurrent data access and system failures to preserve data consistency.
RC2	Transaction completion	Ensures that transactions end successfully or fail, maintaining consistency in the database.
RC3	Data availability	Ensures that the data is accessible to users, meaning that the DBMS allows users to interact with the data.
RC4	ACID transactions	Maintains database robustness through atomicity, consistency, isolation, and durability of transactions.
RC5	Concurrent access	Handles multiple users' transactions simultaneously while preserving data integrity and system reliability.
RC6	Concurrency control	Prevents different concurrency anomalies from occurring in the database.
RC7	Two-phase locking	Prevents conflicts between concurrent transactions by enforcing serial execution.
RC8	Deadlock prevention	Ensures that the DBMS can avoid deadlocks from occurring in the database.

Table 3.1: A list of recognized characteristics of the robust database that answer the RQ1.

4 Database performance

Generally, performance means how efficiently a software completes its tasks. Performance is often measured in response time, which refers to the duration it takes for a call to travel from a system to a different part of the system and back. Figure 4.1 not only shows a simplified example of a system where the components crucial for a database system are emphasized, but it also shows the flow of information from the end-user's device to the storage that holds the database. As illustrated in Figure 4.1, the response time in database systems is the time it takes to send a user's query to the application, which passes the query to a DBMS. Then, the DBMS processes that query and returns a data set to the application, which finally presents the data on the end-user's device, resulting in the complete response time. The response time is the only metric anyone truly cares about because it is the only metric that users experience. A fast database is taken for granted, and when a query takes 10 seconds to execute, users experience 10 seconds of impatience. That same query might examine thousands and thousands of rows, but users do not experience these rows being examined. Instead, what they experience is the time, and it is the most precious thing for them. [10] [14]

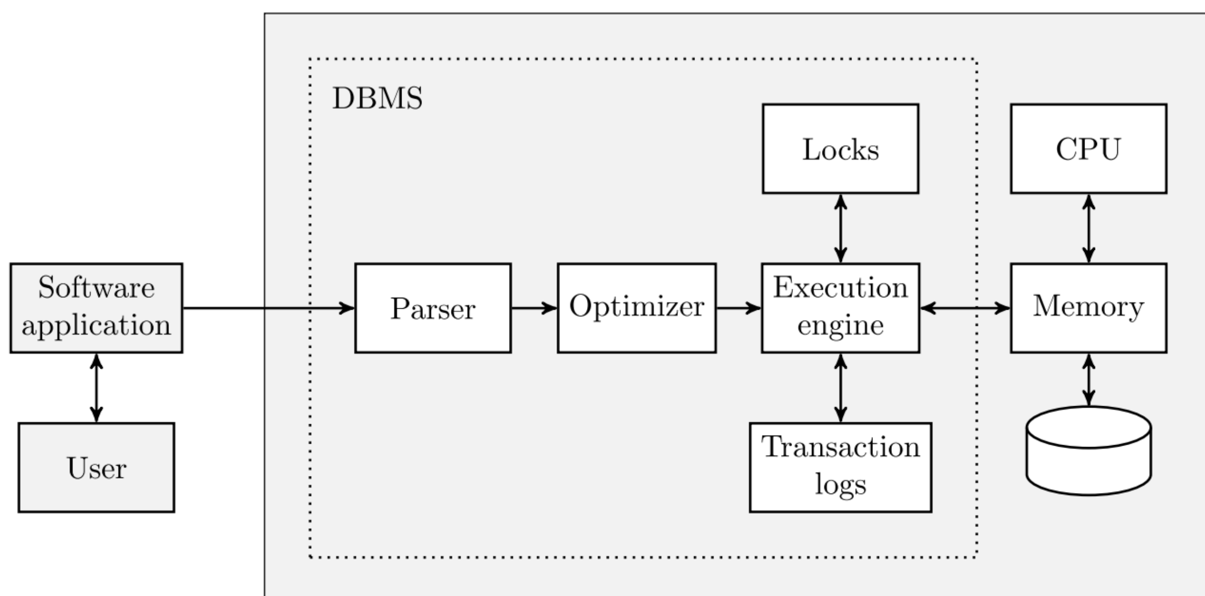


Figure 4.1: A view of the information flow through a database system. [14]

Tuning the database performance involves identifying performance bottlenecks, prioritizing the found issues, troubleshooting their causes, applying different solutions, checking if the performance improved, and then repeating everything again and again until the database functions in the expected way. Hence, the tuning process is an iterative cycle, and after every cycle, it needs to be evaluated whether the changes had any effect. The goal is to make the database run more quickly, which often means lower latency and higher throughput. There is no one right solution to improve the performance, and that is why it is good to be a little creative. However, being creative to find the right

solution can be difficult, and the real challenge is to narrow down the list of possible issues. Database performance tuning can consist of various steps, which will optimize the use of system resources for better efficiency. By fine-tuning hardware, network, indexes, and query structure, the overall database performance can be enhanced significantly. [12] [15] [16]

4.1 Hardware

In general, a DBMS should aim to achieve two main goals, which are low latency and high throughput. Low latency refers to the short time it takes for the DBMS to respond to a query. Time is measured from the moment when the query is made to the point when the result is returned from a database to a user. High throughput means the ability of the DBMS to process a large number of queries simultaneously. In the context of database performance, a fast CPU is required to cover low latency and high throughput. However, having only a fast CPU can cause weak links and bottlenecks. [12] [17]

A database can perform only as well as its weakest link, and the hardware is often a limiting factor. When a database is designed to handle large amounts of data and requests, the primary hardware considerations are CPU (cores), storage or disk, and memory (RAM). The CPU, the size of the storage, and the available memory can limit the capacity of a system. It is essential to carefully choose the hardware, as the wrong components, or the small capacity of a particular hardware resource, can cause bottlenecks and thus reduce the performance of the database. In addition, from time to time, it is important to update the hardware components to prevent bottlenecks from occurring. CPU, storage, and memory need to be in balance, so that a system can work properly. If the system theoretically had 128 CPUs and 1 TB of RAM, but poorly functioning storage, it could still perform slowly. Having extremely fast CPUs, paired with very slow storage, is far from an ideal situation. The aim is to achieve maximum efficiency across all hardware components. As the size of the data increases, so does the usage of CPU, memory, and storage. [12] [17] [18]

The CPU is a compute resource that can affect database's performance and cause a bottleneck. CPU exhaustion can happen when users try to execute too many queries in parallel or when queries run too long on the CPU. Processing hundreds of thousands up to millions of operations per second tends to get very high CPU loads and having too few processors means that less work can be done simultaneously. The more processors are in use, the more workload can be distributed across available resources. An increased number of processors not only provides higher levels of concurrency, but also reduces the query execution time. This means that with more cores it is possible to achieve better efficiency and optimal performance. When choosing the CPU, it is important to pay attention to different models, as each new generation will bring performance improvements over the previous

generations. Before purchasing new CPUs, it is recommended to investigate whether the workload is CPU-bound by checking the CPU utilization. Running the most important queries will tell how heavily CPUs are loaded, and whether they can deliver the needed performance. [12] [13] [17] [19]

PF1: CPU. Upgrading the CPU, or acquiring more CPUs, will provide faster query execution time, and allows more queries to be executed concurrently, thus improving the performance of the database.

The performance of storage systems has advanced significantly since the transition from using hard disk drives (HDD) to solid-state drives (SSD), and because of this transition, latency of query execution has reduced. The latency in HDD forms from the time that arises when the disk platter rotates into the correct position, so that a single item of information can be retrieved from the disk. SSDs do not have moving parts, and the data are stored in semiconductor cells. This reduces latencies for data transfer, as there is no need to wait for the mechanical movement of the disk. SSDs are typically 10 to 20 times faster than HDDs, so even when queries require accessing the storage, they will deliver decent performance. [12] [20]

PF2: Storage. Upgrading the HDD to an SSD will bring significant performance improvement, as it reduces the latency of query execution.

However, despite the fact that storage devices have improved a lot from what they used to be, they are still the slowest component in a computer. When compared to the CPU or memory, storage systems have not evolved as fast. So, when evaluating which component should receive the most resources, it is recommended to invest in storage. If a read or write operation is executed on an unsuitable disk, it is one of the fastest ways to weaken all the other performance optimizations that have been carried out. When storage is going to be acquired, it is wise to select one that will still be suitable in the future. It is necessary to think about the situation where the database is expected to end up. For example, the selection of storage is influenced by the potential growth rate of data and the already existing data. It should be estimated how much data is stored into the database every day and how much it has accumulated after a certain period of time. In addition, it is worth considering whether the data will be kept forever or whether some of it will be deleted later. [12] [17]

Memory exhaustion usually happens when too much memory has been allocated to the database. Memory exhaustion can be addressed by acquiring more memory, which allows caching more data. Caching is a mechanism for optimizing the query execution. Frequently accessed data can be cached into memory, and it reduces the need to constantly retrieve the same information from a disk. Accessing cached data is particularly convenient in situations where queries are similar to each other, and multiple users need to access the same data at the same time. By holding the commonly accessed

data in memory, the caching mechanism improves the query response time and the overall performance. The performance improvement is a result of reduced disk input/output (I/O) activity. Accessing the disk to fetch data is much slower than returning data from memory. Thus, the main reason for having a lot of memory is not just to store a lot of data, but to avoid using the disk. Increasing the amount of memory may be the cheapest and fastest way to address performance issues. Nowadays, memory is inexpensive, so it can be added as much as one can afford. If the database is small and handles only a little data, there is no need to over-allocate memory, but it is good to keep in mind the possible data growth in the future. No one can say what amount of memory is sufficient, and there is no such recommendation. Different databases have different requirements, and it heavily depends on the use case. [12] [13] [17] [19]

PF3: Memory. The more memory the database system can utilize, the more data can be cached. Cached data reduces the need to access the disk when fetching the data, and this improves the performance of the database.

Hardware has improved a lot in the past decades, so upgrading the hardware is the easiest way to increase the performance of a database. It is not possible to entirely avoid a discussion about acquiring new hardware, especially if, after troubleshooting, it is found that the database cannot keep up with the load due to outdated hardware. The fact is that when it comes to performance, hardware is always a limiting factor, and more performing units cannot be squeezed from a system than the underlying components can provide. Before anything else, when it comes to enhancing a database's performance, it is usually recommended to check the components and switch them to newer ones. The new hardware will help significantly to reduce the time it takes from the DBMS to process complex queries. [13] [17] [21]

However, upgrading the hardware really benefits when the newly acquired hardware is significantly better than the old one. Changing the hardware too frequently will not bring the desired outcome. As a matter of fact, it will only repeat what is already known: computers run faster on faster hardware. Switching to new hardware even when it is not necessary is a missed opportunity to learn the real cause of, and solution to, slow performance. In addition, changing the hardware often is not a sustainable approach. Some upgrades can be relatively quick and easy, and some long and complicated. Either way, purchasing hardware can become an expensive process, especially if bought in large quantities. Upgrading the hardware not only requires money, but also time. Therefore, to achieve the highest return on investment, it is crucial to be objective when optimizing performance. When thinking about improving the database performance, it should be considered what is the acceptable performance and is the investment worth the performance gain. [10] [16]

Tuning the database to its theoretical maximum performance is difficult, nearly impossible. Instead, the target should be to tune until the database performance is good enough or acceptable. The performance investment can become very quickly expensive in comparison to the gained performance. The 80:20 rule defines that when 20 percent of resources have been invested, it may result in 80 percent of the possible performance enhancement, but for the remaining 20 percent possible performance gain, it may be required to invest an additional 80 percent of resources. To derive maximum efficiency of the database, it is important to realistically estimate performance objectives. There is no standard definition for a good enough performance that everyone can follow, as it always depends on a use case. Best practices are good guides to improve performance, but in the end, performance requirements are usually set by customers, investors, business owners, application and database developers, and other people who depend on the database. An essential part of database performance tuning will involve talking to these people to get an understanding of what is a good enough and realistic set of requirements. [16]

4.2 Network

No matter how good hardware the system has, executing the queries can still be slow if the network connection is not optimized. Users might execute highly optimized queries but if the network connection used to submit these queries is poor, it adds a significant overhead to the overall performance. It is a fundamental part of the system setup to have an optimal network configuration with appropriate bandwidth. The network can easily become a bottleneck, as it is often forgotten how high the network traffic can get. As with memory, the required network bandwidth depends on the use case, but in general, it can be said that a low throughput workload will consume less traffic than a higher throughput. Possible workload increases are good to keep in mind, and it is wise to prepare for tuning the network connection. [17] [16]

PF4: Network. Even if all the other performance factors have been optimized to deliver the best possible performance for the database, if the network connection is poor, it quickly becomes a bottleneck and negates the other performance optimizations.

A poorly functioning network can greatly slow down performance. One common issue is packet loss, where data packets fail to reach their destination. When packet loss occurs, the network protocols will attempt to fix the problem by waiting and resending the lost packets, which causes delays. The delay time can increase even more because the DBMS has to move data across the network. Whenever users access data from the database, that data have to travel from one place to another, and, in the worst case, the distance can be thousands of kilometers. Network transmissions take time, so minimizing the

number of these transmissions lowers the query execution time. Therefore, network configuration is a key strategy in enhancing query efficiency. [12] [20]

PF5: Data packet loss. When Data packet loss happens, users fail to receive data from the database. Fixing the packet loss issue causes delays, meaning that query execution time is extended, and the performance of the database is degraded.

Network transmissions can be thought of as a rowboat crossing a river. A certain number of people on one side of the river want to go across to the other side in a boat. The more people can get into the boat at each crossing, the fewer trips have to be made, and the sooner they all get across. If the people represent data, and the boat is a single network transmission, then the same logic applies to database network traffic. The goal is to pack as much data as possible into each network transmission so that there is no empty space. This is how the amount of network transmissions is reduced to a minimum, and all the data is sent in the most efficient way possible. [20]

PF6: Network transmission. Each network transmission should contain as much data as possible, so that the fewest number of trips would be needed to keep the query execution time as short as possible.

When the rowboat is fully loaded, the number of trips across the river can be reduced. However, the width of the river is something that cannot be controlled. A fortunate thing is that, in the world of databases, the distance between the user executing queries and the database server can be somehow controlled. The closer the user executes queries to the database server, the less time is elapsed in each network transmission. It is also advised that the application server should be located in the same data center, or even in the same network rack, as the database server. The further the distance between these two, the higher the network latency will be for database requests. [20]

PF7: Database server distance. The closer the user is to the database server, the less time it takes for a query to travel from the user to the database server.

4.3 Indexes

Users run queries on a database to search for one or more rows that satisfy their search criteria. Examining every row within the database to find out which rows satisfy a certain query would be too time-consuming. This would be especially tedious if the table contains thousands and thousands of rows that do not match what the user is looking for. Indexes are the answer to this problem as they provide an alternative way to access data in one or more tables. Indexes are data structures that a DBMS use to improve the performance of SQL queries by finding rows swiftly. An index can be on

one or more columns of a table. The idea is that when only a subset of the rows from a table are needed, the DBMS can utilize an index to identify the matching rows instead of examining each row individually. This helps the DBMS reduce the amount of data that needs to be viewed in order to execute a query. Indexes are essential for good performance, and their use becomes more important as the amount of data grows larger. Small databases with little data often perform well even without indexes, but as the data set grows bigger and bigger, performance can become poor very quickly. Optimizing indexes is one of the most efficient ways to improve the performance of a database, as the capability of indexes increases the speed of data retrieval, saves time, and provides an efficient way to locate data in the database. [4] [12] [22]

PF8: Indexes. The use of indexes in the database helps the DBMS locate queried data from the vast amount of data, meaning that the data is returned to the user much faster.

Indexes in a database can be compared to indexes in a book. The index at the end of a book allows a reader to locate a specific topic in the book. In this way, the reader does not have to search through the book from the beginning page by page to find what he or she is looking for. Instead, the index directs the reader straight to the correct page that contains the desired information. [4]

Similarly, in databases, indexes provide an efficient way to search data from tables. If the database does not have indexes, the DBMS has to search through the entire table to find where the desired data is located. With the help of indexes, however, the DBMS can search through the index structure to quickly find out where the queried data is located and then access those locations directly to retrieve the data. Thus, indexes provide an efficient access path between the user and the data. By providing this access path, the user can query data from the database, and the DBMS will know where the data is located and how to fetch it for the user with minimal effort and in the quickest manner possible. Furthermore, indexes not only help to locate the data, but also inform the DBMS when to stop searching. Without indexes, the DBMS does not know if it has already found all the desired data, or if there are still data remaining in the table that needs to be returned to the user. When indexes have been implemented, the DBMS knows when it has found the data and can stop scanning the remaining rows in the table, thus reducing the query execution time and improving the performance. [4] [18] [23]

Some database developers believe that their databases are different from other databases. These same people will develop and deploy their databases without indexes because they think that they do not need them. From time to time, it is argued that a small proof-of-concept database, that does not have a lot of data in it, does not need indexes because performance is not important in that case. Another argument against adding indexes is that the whole database fits into memory, so adding indexes would require even more memory. However, even databases that are expected to be small, can start growing

quickly once deployed. Any proof-of-concept or unimportant database would not have been created if there was not a need for it, or someone was not interested in spending resources on it. In addition, even if the database could fit in memory, it does not mean it will be fast. As mentioned, indexes provide alternative access paths for data, aiming to decrease the number of rows needed to be examined in order to access the data. Without these alternative routes, data access would require reading every row in a table. It is good to keep in mind that these reasons may not concern every database, as it always depends on the use case, but they will be most likely similar. [23]

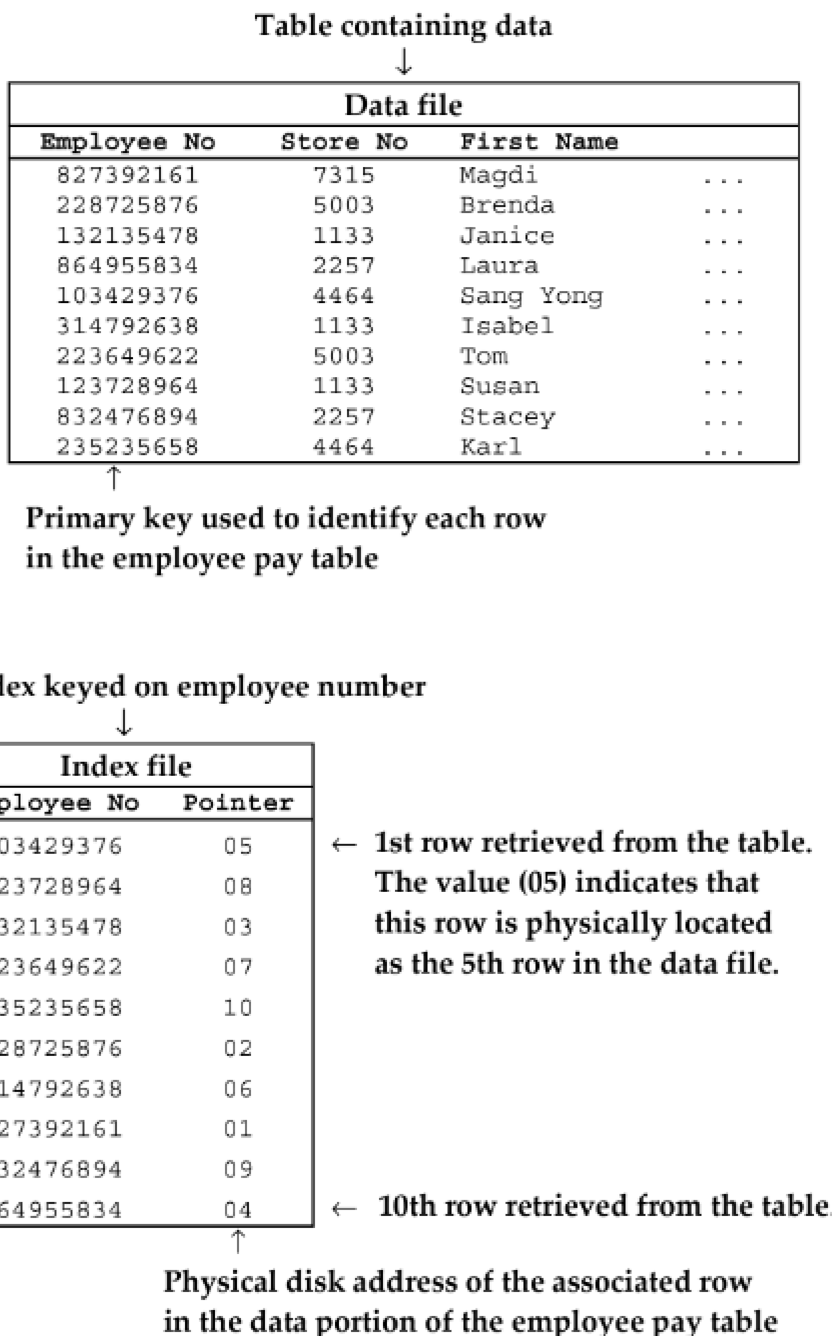
4.3.1 Using an index to access the data

Figure 4.2 illustrates how the rows are stored and organized in an employee pay table. Without an index, the rows would be returned to a user in the same order as they are physically stored in the table. [4]

Record Sequence	Employee Number	Store Number	First Name	Init	Last Name	Dept	Hourly Rate	Hours Worked	Sales
1	827392161	7315	Magdi		Ali	666	015^50	43^5	10400
2	228725876	5003	Brenda	M	Fields	666	009^85	40^0	04555
3	132135478	1133	Janice	A	Porter	333	008^55	10^0	00000
4	864955834	2257	Laura	J	Hansen	444	009^75	15^0	00024-
5	103429376	4464	Sang Yong		Lee	333	009^95	35^0	00563
6	314792638	1133	Isabel	L	Houle	666	023^60	45^0	21245
7	223649622	5003	Tom	P	Simpson	333	017^90	20^0	00099-
8	123728964	1133	Susan	P	Murphy	111	023^21	45^0	10125
9	832476894	2257	Stacey	V	Bond	555	011^25	40^0	00740
10	235235658	4464	Karl	C	Ryckman	333	014^45	42^5	07866

Figure 4.2: 10 first rows of the employee pay table. [4]

Figure 4.3 shows the appearance of the employee pay table when it is organized with an index. Each row in the index contains the key field and a pointer to the related row in the table. The first field in each row of the index holds the key field value corresponding to the related row in the table. In Figure 4.3, the key field is the employee number, and the second column, the pointer, contains the location of the row in the table that is identified by the key field. [4]



Data file

Employee No	Store No	First Name	
827392161	7315	Magdi	...
228725876	5003	Brenda	...
132135478	1133	Janice	...
864955834	2257	Laura	...
103429376	4464	Sang Yong	...
314792638	1133	Isabel	...
223649622	5003	Tom	...
123728964	1133	Susan	...
832476894	2257	Stacey	...
235235658	4464	Karl	...

Index file

Employee No	Pointer
103429376	05
123728964	08
132135478	03
223649622	07
235235658	10
228725876	02
314792638	06
827392161	01
832476894	09
864955834	04

Figure 4.3: Structure of a keyed index table structure. [4]

Indexes enable random access to table rows without needing to access other rows sequentially one by one. The DBMS achieves this by first locating a match on the key field within the index. It then uses the address in the pointer field to directly retrieve the desired row from the physical disk address. For example, to access a specific row in the employee pay table, the user inputs the employee number. The system searches the index for this number, and once a match is found, the DBMS uses the pointer field's address to fetch the physical row from the disk. When the address of the row is retrieved from the pointer field in the index, the disk drive can directly move to the physical address on the disk

where the row is located. This eliminates the need to sequentially read through all the previous rows in the table to find the desired one. [4]

Another example of an index is a balanced tree (B-tree), and it is a hierarchical tree structure. A B-tree starts from the top, where the root node is. This node contains pointers to the appropriate branch node for any given range of key values. Moving down the B-tree, the branch node points to the appropriate leaf node, containing a list of key values and pointers locating the data on a disk. [20]

In Figure 4.4, there is a single-column table with 27 rows in a random order, and 3 rows in one leaf node. Assuming that a user wants to search the value 5, the DBMS has to perform seven read operations to access the correct leaf node when an index has not been implemented. [16]

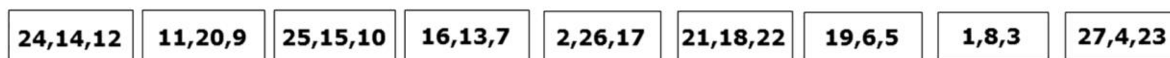


Figure 4.4: Initial layout of 27 rows. [16]

The numbers in Figure 4.4 can be arranged by creating an index on the column, with the result shown in Figure 4.5. With the help of the index, the DBMS has to perform only two read operations to access the value 5. In addition, when the DBMS finds the value 6, it can be sure that there are no more rows with the value 5. [16]

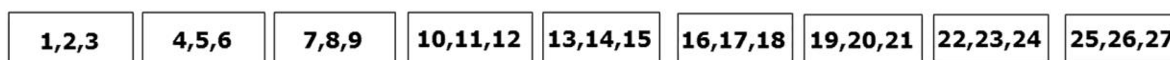


Figure 4.5: Ordered layout of 27 rows. [16]

Even though the value 5 could be searched with a little effort, The solution in Figure 4.5 is not the best one for effective data retrieval. For example, if the user wants to query the value 23, the DBMS has to perform eight read operations if the numbers are in the same order as they are in Figure 4.5. This issue can be solved with the B-tree shown in Figure 4.6. The search for the value 23 starts from the root node at the top of the B-tree. Since the value 23 is bigger than 19, the search process moves to the right branch node. In the right branch node, the value 23 falls between the values 22 and 25, so the search process continues to the middle leaf node on the right that starts with the value 22. Finally, the search process ends as the correct leaf node is found, and the value 23 is retrieved with only two read operations. In the same way, the value 5 can be searched with the same number of read operations, and this is what makes the B-tree so efficient. [16]

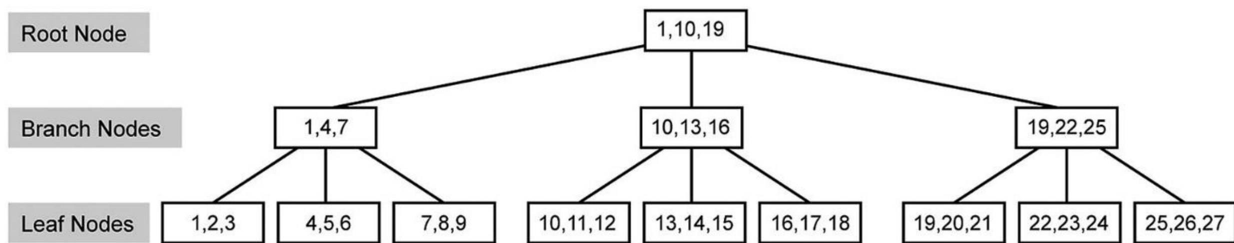


Figure 4.6: B-tree layout of 27 rows. [16]

4.3.2 Things to consider before implementing indexes

Indexes have downsides in both books and databases. In a book, the index list takes space, and if a second edition is ever published, every modification requires a corresponding change to the index. In addition, before creating the index for the book, the author must consider which topics will be frequently looked up. If readers never check specific topics from the index list, then those indexes occupy space unnecessarily. These same issues related to book's indexes, apply to the databases. Indexes consume computer's resources and if users never take the advantage of existing indexes, then the resources are wasted for nothing. Indexes should be added only if it is certain that they will bring the desired performance improvement and users will really benefit from using them. In addition, an index that was once useful may eventually become useless if no queries are utilizing it anymore. Such unused indexes do not bring any value and should be deleted. [24] [25]

PF9: Unused indexes. The use of indexes requires resources from the computer, and if no one takes advantage of the existing indexes, these resources cannot be used for something more useful. A computer's resources are limited, and if they are not used wisely, it will negatively affect the performance of the database.

While indexes can enhance the speed of data retrieval, the performance benefit does come at a cost. Indexes can slow down data modifications because of the additional time required to update the indexes whenever rows are inserted, deleted, or updated. This process demands computational resources to ensure that the indexes remain consistent with the modifications. If there are too many indexes on a table, which are often updated, it can lead to overhead because the DBMS must do extra work to keep indexes up to date while data modification operations are performed. This extra work increases the time it takes to complete the query, thus affecting negatively to the performance. It is essential to eliminate any duplicate, redundant or wrong indexes, because maintaining them will unnecessarily consume CPU, storage, and memory. The DBMSs are generally quite efficient at managing indexes, so having up to five indexes on an active table, or 10 to 20 on a less frequently updated table, is not usually a concern. For tables that are mostly read-only with minimal updates, having numerous indexes typically does not create a major overhead. However, for tables that are

frequently updated, it is advisable to limit the number of indexes to avoid slowing down the database updates. [4] [15] [24]

PF10: Index updates. Whenever data is created, updated, or deleted, indexes need to be updated to ensure they remain consistent with the modifications. This process requires resources from the computer and increases the query execution time, resulting in degraded database performance.

Indexing data in an effective way requires knowledge of how that data will change over time, what kind of queries users will perform, and what is the volume of data that is expected to be dealt with. This is what makes index tuning so challenging. To index effectively, it either needs history or an ability to predict what are the data usage patterns in the future. In any case, when it is going to be decided whether or not to use an index to improve the performance of a query, the effect on overall system performance must be considered. [18]

It is good to remember that an index might not always be the best solution for all the queries, or in every situation. If it is needed to access a large part of the data in the table, it will become faster to read the whole table from the beginning to the end, than use indexes to retrieve the rows. Similarly, when the whole book, or a major part of it, is going to be read, it would be too time-consuming to switch between different book sections, where indexes point to. Instead, it is much more reasonable to read the book from cover to cover. If a specific item needs to be found from both a book and a database, that is when an index is used. [20] [26]

4.4 Query structure

Writing an SQL query is like writing code with a programming language in that sense, it can be done either well or poorly, and poorly written SQL queries often cause issues with database performance. Database administrators are commonly told to fix the database when queries are not performing fast enough. It is not unusual for the database administrators to have to prove that the problem is not the database itself, but instead the SQL queries that are not written efficiently. A good starting point is always to have SQL queries written effectively on the first time, as it is the best way to get good performance from SQL queries. This will lead to a more efficient use of database resources and overall better database performance. [24]

SQL queries, and how they have been structured, affect the effectiveness of indexes. Fetching an overly large number of rows from a table, or using a filter that returns a bigger result set than needed, can make indexes less effective. To enhance performance, it is crucial to write queries that efficiently utilize indexes. Poorly written queries can prevent the DBMS from selecting the appropriate indexes,

leading to longer query execution times. In this situation, the query structure does not give the DBMS the best possible processing strategy to retrieve the data. [16]

Existing queries can be rewritten, and it is a process of turning an original query into a more efficient version without changing what the query does. The goal is to optimize query execution by restructuring the query to take advantage of indexing, so that less computational resources are used to execute the query. Rewriting queries can simplify complex conditions and eliminate redundant operations, resulting in faster execution time and more efficient resource utilization. Even though query rewriting can improve performance, it can be a challenging task to do. When rewriting a query, it needs to be ensured that the query produces an identical result as the original query did. In addition, some queries might not benefit from rewriting, and, in some cases, the rewritten query may perform worse. The modified query needs to be tested to see how it acts on specific data. This can be a time-consuming process which, in a worst-case scenario, does not bring the desired improvement. When tuning the query, it is necessary to think if the query response time is acceptable. If not, then the optimization should be considered. If the time is already acceptable, then it is not wise to waste resources, since nobody is asking questions when the database is fast and works as expected. [10] [27]

One of the best ways to improve performance and reduce query execution time is to limit the amount of data which the query retrieves. This includes both columns and rows. When executing a query, only the necessary columns and rows should be included in the query. For example, it is not wise to use the SELECT * statement to return everything from the table. Queries should be as simple as possible, and checking a query before executing it is essential for reducing the query complexity. Less complex queries return less unnecessary columns and rows, and thus reduces the size of the result set. Selecting more data than is actually needed not only increases computer's resource usage but also data transfer across the network, further degrading performance. In fact, the fastest SQL query is the one that is never sent to the DBMS. In other words, a query should not be sent unless it is absolutely necessary. Even the simplest request involves a network transmission, so interacting with the database should be avoided if possible. [10] [16] [20]

It is good to be aware of the request when someone asks to return thousands and thousands of rows from a database. Just because someone is saying that those are the business requirements does not mean they are right. Returning huge data sets is costly, and few human beings are capable of processing thousands of rows. It is unlikely that thousands, or even more, rows are utilized entirely at once, so one should be ready to push back on these kinds of requests and be able to justify the reasons. [16]

PF11: SQL query structure. When poorly written SQL queries are executed, they do not allow the DBMS to retrieve data from the database in the most efficient way. It is advisable to consider rewriting queries to make them less complex. Queries should return only the data that is needed, and they should utilize indexes to reduce execution time, thus improving database performance.

4.5 Summarizing database performance factors

Throughout Chapter 4, different database performance factors were recognized. These factors were summarized into entities called Performance Factors and uniquely numbered to identify them. Performance Factor entities answer RQ2: What are the factors that affect database performance? In Table 4.1, all the Performance Factor entities have been organized into one table to provide a clear understanding of the factors that affect database performance. The table includes the Performance Factor entity number, the name of the factor, and a short description of each factor.

PF#	Performance Factor	Description
PF1	CPU	Upgrading the CPU or increasing the number of CPUs enhances query execution time and concurrency.
PF2	Storage	Switching from HDD to SSD reduces query execution latency.
PF3	Memory	Increased memory allows more data caching, and it reduces the need to access the disk.
PF4	Network	Poor network connection can bottleneck database performance and negate all the implemented performance optimizations.
PF5	Data packet loss	Fixing the data packet loss extends query execution time, leading to degraded database performance.
PF6	Network transmission	Reducing the number of network transmissions minimizes query execution time.
PF7	Database server distance	Shorter distance between the user and the database server will lead to faster query execution time.
PF8	Indexes	Indexes help the DBMS locate data quickly, speeding up query execution time.
PF9	Unused indexes	Unused indexes consume the computer's resources unnecessarily, meaning that these resources cannot be utilized for something more useful that would improve performance.
PF10	Index updates	Updating indexes during data modifications requires computer's resources and increases query execution time.

PF11	SQL query structure	Poorly written SQL queries do not allow the DBMS to work in the most efficient way possible, which leads to increased query execution time.

Table 4.1: A list of recognized database performance factors that answer the RQ2.

5 Database security

5.1 Basic security principles

Security and privacy are often considered synonyms, but they do not mean the same thing. Privacy is the need to restrict access to data that cannot be disclosed to unauthorized users. Security consists of the actions that are implemented to ensure privacy. When the security is implemented right, it brings protection for invaluable organizational resources against unauthorized reading, updating, and deleting of the data. Especially in database security, the most essential thing is that only authenticated users can perform authorized activities within the database. [3] [28]

Authentication means that users, whether they are end users or employees, verify they are who they say they are. Users authenticate themselves using their user credentials, such as username and password. After a user has been authenticated, it needs to be determined what data that specific user can access and modify. This is called authorization. Just because users are authenticated, it does not mean that they can access all the data and perform whatever action they want to. Instead, users are allowed to access and modify the data within the limits of their authorization. These permissions are also known as privileges. [29]

In authorization, three main entities are involved: users, data, and operations. Operations are the specific actions that can be performed on the data. Therefore, an authorization can be defined as a triple, which indicates that a certain user has the right to perform an operation of a certain type on certain data. Authorization checks whether a given triple can be permitted to process or not. A user logs in to the DBMS using credentials. The username is used to identify each user uniquely, and the password is used to authenticate the user. Credentials are required so that the database can be accessed, thereby preventing unauthorized users from viewing the data within the database. [5]

STP1: Authentication. Not everyone can be allowed to access a database because, otherwise, data would end up in the wrong hands. Before users are allowed to access the database, they need to prove that they have permission to access it by authenticating themselves.

STP2: Authorization. Authentication alone is not enough to keep data safe; it also needs authorization. After users have proved they are who they claim to be, authorization is used to determine what particular operations each user can perform on the data.

In the case of employees, authorization should not give them access to all data. In general, companies follow the principle of least authority when granting access to data. It is safest when employees can

access only the data necessary for them to do their jobs and nothing more. By restricting access to certain data, companies not only protect their most critical information but also help employees find the needed data more quickly. If an employee had access to all the data, searching through it to find what is needed would take much longer. [29]

STP3: Least authority. Employees, and other users, should be granted the least authority to access data. This means restricting access to data, so that employees only have access to data that is absolutely necessary for them to do their jobs. Granting the least authority is a precaution for scenarios where an attacker gains access to an employee's account but is limited to the access rights the employee possesses.

Many organizations have taken DBMSs as their main technology to manage data for day-to-day operations and decision making. As the number of DBMSs has increased, the security of data managed by these systems has become crucial. Data are essential for almost every modern organization nowadays, because it ensures the continuity of business, and therefore, data must be protected. If a company suffers from a database security breach it can result in losing trade secrets or confidential customer and user information ending up in the wrong hands, and this can have devastating consequences for the business. Becoming the target of a security breach is always a negative thing for the company, even if the breacher does not get any sensitive information from the database or does not use the information for any malicious intentions. When the company suffers from a security breach, it can lose its reputation, as customers, investors, and other stakeholders start to think that they cannot rely on the company anymore. Once the trust has been lost, it is hard to gain back, and the stakeholders will most likely take their money and begin to conduct business with someone else who is more reliable. [3] [30]

Security breaches can be categorized as unauthorized data observation, incorrect data modification, and data unavailability. Unauthorized data observation occurs when someone who does not have the right to access a database discloses the data. Incorrect data modification means that someone who is not entitled to access a database modifies the data and leaves it in an inaccurate state. Data unavailability is a situation where the essential information for the proper functioning of the organization is not easily available when needed. Thus, database security is thought to have three goals: confidentiality, integrity, and availability. Confidentiality refers to the protection of data against unauthorized disclosure, so that the data that must be kept private also stays private. Integrity refers to the prevention of unauthorized data modification, and because of integrity, the data remains accurate. Availability means that data can be accessed whenever it is needed. This implies protecting the database from everything that would make data unavailable. Such factors would be, for example,

hardware and software errors and malicious data access denials. Availability also refers to recovering from incidents that made the data unavailable, so that the data can be accessed again. [3] [30]

STP4: CIA. confidentiality, integrity, and availability is a trinity that forms database security. Confidentiality ensures data privacy by preventing unauthorized disclosure, integrity keeps the data accurate by preventing unauthorized modifications, and availability guarantees that data can be accessed when needed by protecting it from potential disruptions.

The DBMS ensures data protection using different kinds of components. An access control mechanism is used for ensuring data confidentiality. When a user tries to access a database, the access control mechanism checks the rights of the user against a set of authorizations. The authorization defines whether the user can perform a particular action on data. It varies between organizations what users are allowed to do with the data, so there is no one right access control policy that everyone can use. Instead, organizations need to carefully consider what different user groups can do with the data. Together with the access control mechanism, the integrity mechanism ensures data integrity. As already mentioned, the access control mechanism verifies that the user has the right to modify the data, but the integrity mechanism is responsible for checking that the data remains accurate and correct after the modification. Any use of incorrect data may result in heavy losses for the organization. Finally, the concurrency control mechanism and the recovery mechanism ensure that data are available and correct despite concurrent accesses, failures, and service denials. [30]

5.2 Database security threats

Database security begins with identifying what kinds of threats are associated with the stored data. Once threats have been identified, solutions can be designed and implemented to mitigate the possible harm from occurring. However, implementing security solutions might be neglected, as development teams are told to focus on the core business objectives, which are almost never related to security. It is common to agree that security is going to be implemented during the next iteration, but when the next iteration is about to start, new business-critical objectives will be prioritized. The reason why this happens is that the business is willing to pay only for the development time for different features, which are considered to bring value to the customers. Because security is typically something which the customer cannot see, the business does not think it is necessary to focus on it. Instead, the business will push new features to be implemented, as they are seen more important to be worked on. There will always be some reason for the business to push back implementing security objectives, and it is easy to argue that if something has been broken and it has not caused trouble, why should it be fixed now? The problem with this kind of thinking is that when the lack of security has become an issue, it

is too late. The attacker has already gained access to a database and exported the data. Trying to fix the security after this is useless. [31]

ST1: Neglecting security. Development teams are often told to focus on implementing new business-critical features that will bring value to the customers. While trying to satisfy the customers with the different functions that they value, security is forgotten and soon neglected completely.

Managing database security was fairly simple before the internet. A username and password were enough to secure access to a database. The most significant security threats were internal, from employees who either accidentally corrupted data or purposely exceeded their authorized access. The time before the internet made it also easier for managing data security because data was centralized in one place. Nowadays, data is decentralized, meaning that it is distributed across many systems and environments. This makes it challenging to control who are accessing the data, and what do they do when they get their hands on it. When the internet started to become a common thing, databases were added to a network. However, adding databases to a network brought new kinds of external security threats, so new security measures needed to be implemented to protect the data. Even though the amount of external security threats has increased, it does not mean that internal threats can be forgotten. It is as important to pay attention to what is happening within the organization as it is to pay attention to external threats. [3] [32]

5.2.1 Internal security threats

Employees often know more about a network and the computers on it than someone who is outside of the company. Databases are especially vulnerable to internal security threats because employees typically have direct and legitimate access to them. This can be extremely dangerous when an employee has malicious intent in mind. An employee can take advantage of legitimate access, for example, to share trade secrets, steal money, or sell customers' information for personal gain. In turn, former employees can be a security threat for a company, especially if they did not leave the organization willingly. These people are often interested in revenge and might try to break into the system, interested in damaging data or sharing it for money. To mitigate the risk of former employees entering the system, it needs to be ensured that all access rights are removed when the employee leaves the company. Of course, this does not remove the risk completely because the former employee can always try to access the system in an illegal way. [3]

ST2: Employees. Data in the database can be at risk of jeopardization if employees decide to abuse their access rights. Employees have direct and legitimate access to the database, and this increases the risk that access rights are used for personal gain.

ST3: Former employees. When an employee leaves the company, all access rights need to be removed from that person. In this way, malicious intentions of former employees are prevented, as they no longer have legitimate access to the database.

As dangerous as the intentional employee security threat may be, employees can also cause damage to the data unintentionally or accidentally. An employee can become a victim of a social engineering attack, unknowingly helping an attacker gain access to the database. Malware can spread to a company's computers if the employee downloads a suspicious program or adds personal equipment on a network to which all other company devices are connected. The data in the database are compromised if the employee performs an action which consequences are not known. In this situation, accidental data deletion or other modification, that cannot be reverted, is very likely. The employee can also lose his or her laptop, and without proper protection, wrong people will possess the data almost certainly. [3]

ST4: Carelessness of employees. If employees are not paying attention to what they are doing or what happens around them, it can put data in great danger. Social engineering, malware, accidental data modifications, and the loss of a laptop can occur if the employee does not think before acting.

5.2.2 External security threats

External database security threats come from sources outside the organization. These can be, for example, hackers, organized crime groups, and government entities. The most distinguishing factor, compared to internal threats, is that external attackers do not have privileges to access a database. When an external attacker accesses the data, it is done in an unauthorized and illegal way. [33]

One of the biggest mistakes that increases external threats is to directly expose a database to the internet. The most common reason why databases are made available on the internet is to access them easily from home. Database administrators and developers often want to connect to the database remotely, so that issues can be solved as quickly as possible. However, it is good to keep in mind that, if employees can connect to the database from anywhere, then so can someone who is not supposed to. The problem with databases connected directly to the internet is that they are open to attacks. Numerous bots are scanning the internet and trying to find unprotected databases that can be broken into. The best practice is to allow the smallest set of network connections possible. It should not be possible to connect to the database from all IP addresses. Instead, the access needs to be restricted so that only the static IP address from the office has access. If the database needs to be managed outside of the office, it is best to connect to a machine in the office over a remote desktop, preferably using a Virtual Private Network (VPN). The VPN is the only secure way to connect from outside a network to

inside the network. Employees' computers outside of the office are assigned an IP address on the office network, allowing them to interact with the office computers through a secure connection rather than accessing them directly over the internet. In this way, the office's computers are not available directly on the internet, and possible attackers cannot exploit the open connection to access the database. [31]

ST5: Exposing a database. Databases should never be exposed directly to the internet because otherwise anyone can access them. Databases should be configured so that access is allowed only from certain IP addresses, such as those from the office. If employees need to access the database outside of the office, it is best to use a VPN connection.

Attackers are trying to find database accounts from the internet that can be used to access the database. Particularly, the system administrator's account is what is desired the most, because it has full rights to everything on the database. It is a good practice to change the password occasionally, but in addition to this, the account should also be renamed. Renaming the system administrator's account, and other accounts as well, greatly increases security, as it takes away the attacker's knowledge of the name of the account to log in with. After the renaming, the attacker would need to discover not only the password, but also the username for the account as well. A secure system will only inform that either the username or the password was wrong when a login fails. This way, the attacker does not know if he or she is on the right track while trying to break into the database. Without knowing the correctness of either credential, it will be very difficult to complete a brute-force attack in a reasonable amount of time. However, certain factors, such as the number of used computers, the length of the username, and if the username is accidentally revealed because of social engineering, affect the time it takes for the attacker to break into the database. In addition, if the application is legacy, third-party, or poorly written, it may require using the system administrator as an account name. In this case, the best way to mitigate the risk of an attack is to disable the system administrator's account whenever it is not in use. [31] [34]

STP5: Renaming account. It is commonly advised that passwords should be changed occasionally, but to increase database security, user accounts should also be renamed. Especially the system administrator's account should be renamed because it has more rights than any other user account. When the account is renamed, it becomes more difficult for an attacker to gain access to the database using a brute-force attack, as both the correct password and username must be discovered.

STP6: Disabling account. Because access to the system administrator's account is what attackers want the most, this account should be disabled whenever it is not in use. If the account cannot be used, attackers cannot extract data from the database or cause other harm to the data.

Another external database security threat is a SQL injection attack, which utilizes harmful SQL query to return or modify data. Returned data can contain any information that is considered as confidential and is not supposed to be revealed without proper authorization. SQL injection usually happens when the web or front-end application creates SQL queries based on user input. The main idea of an SQL injection attack is that a harmful SQL query is sent to the back-end via a form field, where a user input is expected, in the web or front-end application. The intention is to disrupt the original SQL query and make the new, injected command run instead. The core reason why an SQL injection attack can happen is because of poor coding practices. Over the years, developers have learned better development practices to prevent SQL injections. Despite this, SQL injections can still happen in legacy applications if the right protection has not been implemented, or in newer applications if the developers have not taken SQL injection attacks seriously while building the application. [31] [35]

ST6: SQL injection attack. The purpose of an SQL injection attack is either to reveal confidential data without authorization or to modify the data, leaving it in an inconsistent state.

SQL injection attacks are popular among attackers because these attacks can be easily reproduced once it is discovered what works best to exploit the vulnerabilities. It usually takes a long time to fix all the vulnerabilities, and during this time, the application and database are left open for attacks for a long period of time. This is because companies are usually unwilling to shut down their service while the necessary actions are performed to repair security issues. Once the security issues are finally fixed, the attack has already taken place and the data has been compromised. [31]

If a company becomes a victim of SQL injection, it can have far-reaching consequences. For example, the attacker could delete whole tables from the database. Assume that a web application executes the SQL query shown in Example 4.1, when a customer navigates to the sales order history page. This SQL query would be created when the customer goes to the fictional URL: http://www.company.com/orders/orderhistory.aspx?Id=25. [31] [35]

SELECT * FROM Orders WHERE OrderId=25;
Example 4.1: SQL query to retrieve an order with the OrderId value of 25. [31]

A string concatenation can be done when SQL queries are combined. Any value that is put at the end of the query is passed to the DBMS. The attacker could change the SQL query if the URL is changed to http://www.company.com/orders/orderhistory.aspx?Id=25%3Bdelete%20from%20Orders%3B. In this new URL, semicolons are encoded as %3B, and spaces are encoded as %20, so that web servers and browsers can interpret them. The new URL would create a new SQL query, shown in Example 4.2. [31]

SELECT * FROM Orders WHERE OrderId=25; delete from Orders;
Example 4.2: Concatenated SQL query with a delete statement. [31]

The first semicolon in the query in Example 4.2 instructs the DBMS that the statement has ended, and that there is another statement that should be executed. The DBMS then proceeds to the next statement and performs what is commanded. After the whole query in Example 4.2 has been executed, everything from the Orders table has been deleted. [31]

SQL injection attacks can be prevented using prepared statements in an application's back-end code. Prepared statements are precompiled SQL queries with placeholders. When a user gives input through the front-end, the back-end replaces the placeholder with the given input, and sends the SQL query to the DBMS to be executed. In this way, only the expected SQL queries are sent to the DBMS. If the attacker would try SQL injection, the back-end would notice that the query does not match with the prepared statement, and refuses to send the query to the DBMS. In a situation where the malicious SQL query gets to the DBMS to be run, and prepared statements have not been implemented, nothing can be done to stop the DBMS from executing the query. To further reduce the risk of an SQL injection attack, the back-end code should validate the user's input so that the data is in the expected data form. If a number is expected, it should be ensured that there is, in fact, a number within the input. If a string is expected, then it should be checked that the input contains a string. If the user's input is not in the expected data form, then the back-end must refuse to process what has been commanded. [31]

STP7: Prepared statements. When prepared statements are implemented in the back-end code, they prevent SQL injection attacks. The back-end expects certain types of queries, and if the provided query does not match the back-end code, the back-end refuses to send the query to the DBMS for execution.

STP8: User input validation. SQL injection attacks can be prevented when the user's input is validated. SQL injections are formed based on user input, and when the back-end notices that the input is not what was expected, it stops the attacker's command.

SQL injection attacks are not successful against only applications which were built in-house. Similarly, applications which were developed by third parties are prone to SQL injection attacks. When a company purchases third-party applications, it is often assumed that the product is a secure application that cannot be attacked successfully. Unfortunately, that is not always the case, and every time a third-party application is brought into a company, a full code review should be conducted to find possible vulnerabilities. This is how it can be ensured that the application is safe to deploy. When

a company deploys a third-party application from which an attacker can steal data, the company itself is responsible for having an insecure application, compromising their customer data, and dealing with the consequences if a data breach occurs. The company that produced and sold the insecure application cannot be blamed for the attack. [31]

5.3 Privileges and access control mechanisms

Most DBMSs allow database administrators to restrict users' access to specific tables, and even to certain columns and rows within a table. Users can be labelled into different groups, and these groups can be granted permissions to view certain data. When a table is queried, the DBMS checks the user's privileges and returns only the data from tables, columns, and rows that the user is allowed to access. For example, suppose a medical database that contains patient data. General data about the patient, such as name, phone number, and appointment schedule, can be available to almost everyone working in the hospital. More sensitive data, such as medical history, appointment notes, prescriptions, and test results, are available only to nurses, doctors, and other medical staff. Labelling groups to see specific data reduces the risk of data ending up with users who are not allowed to view it. A person belonging to an appointment specialist group can see the patient's phone number so he or she can call a patient, but the DBMS will not reveal the patient's medical history as the person is outside of the group who is allowed to view such data. [9]

STP9: Labelling users. Users' access to data within the database can be restricted by labelling users into groups. These groups have different access rights to various data, so the permission to access the data depends on the label assigned to the user.

It is generally safest to give a user the fewest privileges possible, or privileges that are absolutely necessary for the user. Not only does this prevent the user from performing an irreversible action or something improper, but it also reduces the harm that an attacker can do if the user's credentials end up in the wrong hands. If the user does not have a certain privilege to perform a certain action on data, then the attacker does not have these rights either. In some situations, it can be necessary to make an exception and extend a specific user's privileges. If a user needs more access rights than the group is allowed to view, additional privileges can be granted to this individual user. However, this is something that should be carefully considered, because granting excessive privileges increases the risk of data falling into unauthorized hands. Nevertheless, if it is decided that the user will be granted extra privileges, it is important to revoke those privileges when they are no longer needed. [9] [36]

STP10: Revoking privileges. Users' authorization to access and modify data can be changed, and when a user no longer needs certain privileges, they should be removed.

Granting multiple users only the most essential and needed privileges can be a complex process because it requires determining the correct permissions that each user needs. Users themselves are often too lazy to figure out what access they really need and will instead request what they want to access, which is everything. In the case of databases, accessing everything usually means being a database administrator or a member of the database owner group. Both of these roles' privileges are likely far more than the average user needs. There are no easy technological solutions to determine what privileges users actually need, and the database administrator can make a fatal mistake if excessive privileges are granted. That is why the database administrator must take time and carefully consider what data an individual user can access and what actions certain users are allowed to perform on the specific data. [31]

Discretionary Access Control (DAC) is an access control mechanism to manage users' access to data. In DAC, access is granted or denied based on the identification of the user, and authorizations, which are known as rules. These rules define the specific access rights for each user and what actions they can perform on data. Under this policy, the authority to decide who can access certain data lies with the person who created the data, such as the data owner, or the database administrator. However, DAC has issues that can cause the data ending up for people who are not allowed to access it. Once a user has been granted the privilege to access the data, the DAC cannot prevent the user from sharing the data with someone who is not allowed to see it. This is why the word "discretionary" is in the DAC, as the person who has the authority to grant access to data needs to carefully consider which users are given the right to access the data. The data owner can grant, purposely or accidentally, excessive permissions to users, and this can lead to data breaches. [37] [38] [39]

Mandatory access control (MAC) is an access control mechanism where the system gives users access to data based on data confidentiality and user clearance levels. In MAC, data are labelled into different security levels, which are top secret, secret, confidential, and unclassified. Users are also labelled, and when users want to access data, their security labels are compared with the labelled data. Based on this, the users are granted access to the data level for which their rights are sufficient. In the case of users, the label is called the clearance, and for data, the label is called the security classification. Managing MAC requires a lot of work from administrators. To make the MAC work effectively, it needs to be planned what security classifications are given to all the data, and what data different users can access. This work can be continuous if new users need to access the data, and if new data is added constantly. [37] [38] [40] [41]

Role-Based Access Control (RBAC) is an access control mechanism that has been motivated by the need to simplify authorization administration. As the name suggests, RBAC is based on roles, and these roles are granted to different users. Different roles allow users to perform different functions. In

RBAC, users are not granted privileges. Instead, privileges are given to roles, and roles are given to users. Once a user has been made a member of a role, he or she acquires the authorizations that specific role possesses. The user can then access and modify data within the limits that role allows. The good thing about RBAC is that it simplifies authorization management. Whenever a user needs more privileges, a new role can be given to him or her. In turn, if a user no longer needs to or is not allowed to access data anymore, the role can be removed from the user. This ensures that users do not have more privileges than they actually need, thus reducing the risk of unauthorized data access. The drawback of RBAC is that administrators need to handle user’s requests individually. Especially if it is a large company, granting new roles and revoking old ones can be a long process. [29] [30] [37] [42]

STP11: Access control mechanisms. Access control is a mechanism that regulates who can access data in a database. Different access control mechanisms have different regulations about accessing the data. Discretionary Access Control (DAC) grants access based on user identity and rules set by data owners. Mandatory Access Control (MAC) restricts access based on security labels for both data and users. Role-Based Access Control (RBAC) assigns roles, with specific permissions, to users.

5.4 Encryption

Personally identifiable information (PII), such as social security numbers, credit card numbers, home address, medical data, banking information, and names of family members, possess a great challenge for protection. To protect PII data, and any other types of data, DBMSs offer encryption capabilities. Data encryption is an effective way to protect the data within a database, and the main purpose of encryption is to transform data from the plaintext to an unreadable ciphertext. When the encrypted data is retrieved from a database, it needs to be converted back to plaintext so that the data can be read in an understandable form. This is known as decrypting data. Encrypting data enhances data confidentiality and it can be done either symmetrically or asymmetrically. [43]

In symmetric encryption, data is encrypted and decrypted with the same key. This means everyone dealing with the data must know the key, and these people must keep it secret. This secret key holds all the important information for both encrypting and decrypting the data. There are two worst-case scenarios involved with symmetric encryption. Firstly, if the key is lost, there is no way to get the data back to plaintext, and secondly, if the key is known publicly, everything is exposed, and the data can be accessed freely. It is obvious that trust between parties dealing with the data is essential in symmetric encryption. [43]

Unlike symmetric encryption, asymmetric encryption uses two keys. A private key is used for encrypting the data, and a public key is used for decrypting it. The idea is to increase security so that

the data can only be encrypted with one key and decrypted with the other key. The public keys of all people are known to everyone, but the private keys are kept secret and never disclosed. If one user encrypts data with a private key, anyone who has the user's public key can decrypt it. But if the user does the encryption with a public key, the data can be decrypted only with the private key. Now, if this encrypted data falls into the wrong hands, it has no use because no one else but the user is able to decrypt the data. [43]

Assume two users, user 1 and user 2, are exchanging email messages in a secure way. User 1 and user 2 both hold public and private keys. The private key is secret, and neither user 1 nor user 2 discloses their private keys to anyone. However, both users know each other's public keys because these keys are openly shared, for example, on a database. It is known that in the asymmetric encryption a message is encrypted with one key, and it can only be decrypted with the other key. So, user 1 uses user 2's public key to encrypt the message. Once user 2 receives the message he or she is the only one who can use his or her private key to open the message and read it. In this way, confidentiality of the message is guaranteed, and even if the message accidentally went to someone else's email box than user 2's, they would not be able to open it. But what if user 1 encrypts a message with his or her own public key? To read that message, a person would need user 1's private key. That means nobody but user 1 can read the message. [43]

Even though data encryption can increase data security and enhance confidentiality, it comes with a downside. The process of encrypting and decrypting the data requires computing power from the CPU. The more data there are within a database that needs to be encrypted, the more CPU power will be needed. Every time that same data needs to be decrypted, it increases CPU usage. When CPU usage is high, it negatively affects performance and increases query execution time. So, the key is to carefully consider what is the most important data that needs to be kept encrypted and balance the encryption requirements with the increasing data load. [31]

STP12: Encryption. Data within a database can be protected using encryption. When data is encrypted, it is transformed into ciphertext that cannot be read intelligibly. To convert the encrypted data back into its original form, it needs to be decrypted. Data encryption can be done either symmetrically or asymmetrically.

5.5 Auditing database

The reason why data theft can be a problem is that the thief might be able to steal data without anyone knowing that a theft has occurred. A good thief will be able to enter the database, copy data, and exit without leaving a trace. Because copying data from the database does not affect, in any way, to data,

examining the data will not reveal that copying has occurred. This is why DBMSs need to audit every action that has been made within the database. [3]

Modern DBMSs can audit failed login attempts into a log file. This auditing allows the database administrator to check the log file to see if someone has attempted to log into the database using an account for which they have no password. If it is found that multiple failed login attempts have occurred, then it is a clear sign that someone is attempting to break into the database. The auditing not only records failed login attempts, but also saves the IP address from which the attempt came. Once the IP address is known, it can be blocked so that the attacker's access to the database is prevented, even if he or she manages to find the right credentials. [31]

Successful login attempts can also be audited, and it is as useful as auditing failed login attempts. If the database administrator finds that login attempts have stopped, then he or she knows that whoever tried to break into the database gave up the attempt. But auditing successful login attempts, after the failed ones, reveals that the attacker has managed to break into the database. Then, the database administrator knows immediately that the correct password has been found and it needs to be changed. As useful as the auditing is, it has a downside. Logging all the failed and successful login attempts on a busy day makes the log file very big. Reading this file can be a laborious task, and it requires storage space to save the file. [31]

STP13: Auditing logins. The DBMS should audit both failed and successful login attempts. When multiple failed login attempts for the same account are audited, it reveals that someone is trying to gain access to the database illegally using a brute-force attack. To prevent the attacker from accessing the database, the IP address where the login attempts came from can be blocked. Auditing a successful login attempt, after the failed ones, reveals that the attacker has found the right credentials to log into the database.

DBMSs can audit more than just failed or successful logins. Every action that is performed within the database can be saved and later accessed to see what has been done, who has done it, and when it has been done. For example, if there are any doubts that an employee abuses privileges and compromises data, it is essential to find this wrongdoing. Some employees who are working with the data can have a lot of privileges and valid reasons to view every bit of data in the database. Accessing this data is acceptable if employees are working on a troubleshoot or any other issues that are related to their job. But accessing the data in an improper context is something that employees need to be accountable for, and auditing is the best option to find out if any misuses have occurred. However, auditing does not keep malicious people away from the data, so it is not a full-scale security solution. Auditing is simply

just a tool to watch what users do within the database and to catch if something out of the ordinary happens. [44]

STP14: Auditing database actions. Different actions performed within the database can be audited to track who has performed a certain action and when it was done. In this way, users who have abused their privileges and acted improperly can be held accountable for their actions.

If the database is attacked internally, then the database administrator is the worst option from them all because the database administrator is the one who has the most privileges, and there is not much that can be done to stop the attack. What can be done in this situation is to have a reputability strategy in place. A reputability strategy involves using auditing to ensure that malicious actions are traceable. The purpose is to maintain reputability even though an attack took place. When it is known what happened, appropriate actions can be taken against the malicious database administrator. This will prove that actions have consequences, and wrongdoers will be held accountable for their actions. In addition, when malicious people are made to answer for what they have done, it will act as a deterrent to others not to try anything similar. [34]

STP15: Reputability strategy. Companies can have a reputability strategy to maintain their reputation in case of a data breach. When actions performed within the database can be traced through auditing, malicious people and other wrongdoers who have compromised data, or are planning to do so, are shown that actions have consequences. The purpose is to maintain the company's reputation and to send a message that the company cares about its customers' privacy.

5.6 Penetration testing

Penetration testing is a self-defence tool, and the purpose of testing is to find as many vulnerabilities as possible from a system. In the process of penetration testing, the tester mimics a situation where someone with malicious intent tries to attack the system. If the tester manages to steal data, create a system account, cause delays, stop processes, or shut down the service using a denial of service (DoS) attack, then it proves that a possible attacker can do the same. The attacks can cause great harm to the organization and users, and that is why it is important to find the possible vulnerabilities so they can be fixed. Penetration testing can also be used to check whether an already known and reported vulnerability has been fixed and test again to see if the fixed defect does not exist anymore. This makes penetration testing a repetitive task that is conducted iteratively. [43]

Penetration testing is a great way to find out if the system is safe from SQL injection attacks, and testing should be done regularly to find vulnerabilities. Penetration testing can reveal if an attacker can

inject a malicious query into the database and cause harm to the data. It is guided that, at a minimum, penetration testing should be performed every time there is a major software package release, as well as quarterly, to ensure that none of the minor releases are prone to SQL injection attacks or other attacks. In a perfect world, penetration testing would be performed at every software release. But these penetration tests take time and other resources, and the reality is that most companies do not have the resources available to do penetration testing constantly and consistently. [31]

Penetration testing can be done as black box testing, white box testing, or gray box testing. The choice of methodology impacts how the testing is conducted. In black box testing, the tester simulates an external attack by someone who does not have any prior knowledge about the system. This approach follows the same pattern as how most cybercriminals would attack a system and gives insights into how an outsider might penetrate the system. Based on the publicly available information, the tester tries to uncover vulnerabilities that could be used to gain access to the system. The advantage that comes with black box testing is that it offers the ability to evaluate the system from the perspective of an external person. This means, if the tester is able to break into the system using only the information that everyone can access, then the attacker can do the same. [45]

Unlike black box testing, white box testing provides all the knowledge about the system to the tester. This means, for example, access to source code, architecture diagrams, and other documentation. White box testing allows a complete examination of the system and most likely reveals more vulnerabilities than black box testing. White box testing is an exceptionally detailed testing method and is regarded as the most in-depth form of penetration testing, with the ability to reveal vulnerabilities within the system that no other method can reveal. [45]

The last penetration testing methodology is gray box testing, and it is a middle ground between black box testing and white box testing. In gray box testing, the tester does not have a full understanding of the system, but some knowledge has been given. This partial insight might include details about the architecture or source code, which helps in formulating test strategies. With the available knowledge, the tester tries to get as much information as possible out of the system. [45]

STP16: Penetration testing. To find out if an attacker can perform an SQL injection attack and cause harm to data, penetration testing can be conducted to see if the system has any vulnerabilities that can be exploited. Penetration testing can be done as black box testing, white box testing, or gray box testing, and they all have a different approach.

5.7 Summarizing database security

Throughout Chapter 5, various aspects affecting database security were recognized. These aspects have been summarized into two distinct entities: Security Threat, which addresses the different threats a database can face, and Security Threat Prevention, which outlines the measures to prevent these threats. Both entities answer RQ3: What security threats a database can face, and how to prevent them? To provide a clear understanding, these entities have been organized into two separate tables. Table 5.1 includes the Security Threat entity number, the name of the security threat, and a short description of each threat. Table 5.2 follows the same structure but focuses on the Security Threat Prevention entity.

ST#	**Security Threat**	**Description**
ST1	Neglecting security	Security can be neglected when development teams focus only on delivering new features to keep the customer satisfied.
ST2	Employees	Employees may abuse their legitimate access rights for personal gain, thus putting data at risk.
ST3	Former employees	Removing access rights when employees leave the company prevents malicious actions by former staff.
ST4	Carelessness of employees	Lack of employee attention can lead to data breaches through social engineering, malware, or accidental loss.
ST5	Exposing a database	Directly exposing a database to the internet allows unauthorized access. Access should be allowed only from a certain IP address or through a VPN connection.
ST6	SQL injection attack	SQL injection attacks allow attackers to reveal or modify data without authorization.

Table 5.1: A list of recognized database security threats that answer the RQ3.

STP#	Security Threat Prevention	Description
STP1	Authentication	Ensures that only users with a right permission can access the database by verifying their identity.
STP2	Authorization	Defines what actions authenticated users are allowed to perform within the database.
STP3	Least authority	Restricts user access to only the data that is necessary for their role and reduces the risk if an account is compromised by an attacker.
STP4	CIA	Ensures database security through confidentiality, integrity, and availability of data.
STP5	Renaming account	Renaming accounts, like the system administrator's, increases difficulty for brute-force attacks.
STP6	Disabling account	Disabling unused admin accounts prevents attackers from exploiting them to gain unauthorized access.
STP7	Prepared statements	Prepared statements prevent SQL injection attacks. If the provided query is not what the back-end expected to receive, it refuses to send the query to the DBMS.
STP8	User input validation	Validating user input helps prevent SQL injection attacks. If the back-end notices that the user input is not what was expected, it refuses to send the query to the DBMS.
STP9	Labelling users	Users can be labelled into different groups, which have different data access and modification rights.
STP10	Revoking privileges	To ensure data security, access rights should be removed from users when they no longer need certain permissions.
STP11	Access control mechanisms	Access control mechanisms define how data access is granted based on user identity (DAC), security labels (MAC), or roles (RBAC).
STP12	Encryption	Protects data by converting it into unreadable ciphertext, which can only be decrypted with the correct key.
STP13	Auditing logins	Auditing login attempts helps detect brute-force attacks and it saves the IP address, which can be blocked to prevent the attacker from accessing the database.
STP14	Auditing database actions	Saves user actions within the database to detect and hold accountable those who have abused their privileges.

STP15	Reputability strategy	A reputability strategy guides companies in maintaining their reputation by showing others that actions have consequences. The purpose is to prevent malicious people from compromising data and to send a message that privacy matters.
STP16	Penetration testing	Simulates attacks to identify system vulnerabilities and test defence against SQL injection attacks.

Table 5.2: A list of recognized countermeasures to prevent database security threats that answer the RQ3.

6 Discussion

Even though the research questions presented in this thesis are focusing on their own areas, they collectively contribute to the overall functionality and reliability of a database system. Answers to these research questions form a picture of what constitutes a well-designed database system, one that can be reliably used in business operations. RQ1 focuses on the robustness of a database, ensuring that it operates reliably under different circumstances, and can recover from failures in a way that data is not affected. RQ2 examines the performance factors, which directly impact how efficiently the database can handle concurrent access and large volumes of transactions. Finally, RQ3 addresses security concerns, ensuring that the database is protected against external and internal threats, keeping sensitive information safe from unauthorized access and manipulation.

Robustness and performance are closely related to each other because, in a robust system, performance can be maintained even under heavy transaction loads, concurrent access, or after a system failure. Robustness characteristics, such as transaction management (RC1) and concurrency control (RC6), ensure that the database can handle multiple simultaneous operations without compromising data integrity. Without proper performance optimizations, such as CPU upgrades (PF1), memory caching (PF3), and indexing (PF8), even a robust system might fail to meet user expectations under high loads. These performance factors help execute multiple transactions quickly, ensuring that the system can maintain robustness without slowing down. Similarly, SQL query structure (PF11) can influence how well transactions are handled, and well-structured queries reduce bottlenecks, aligning robustness with performance.

In addition, users need to be able to access data quickly, so data availability (RC3) is essential for robustness. Performance factors, such as network (PF4) and data packet loss (PF5), directly affect the availability of the data. A robust database requires fast, reliable access, and if the network is weak, or users fail to receive data from the database due to data packet loss, it degrades performance and makes the data less available. Because data availability is one of the characteristics of a robust database, robustness can be compromised if performance factors PF4 and PF5 prevent users from fast access to data. In turn, minimizing network transmissions (PF6) and reducing the distance between users and the database server (PF7) are factors that make data more available to users and thus strengthens robustness.

Robustness and security are also related to each other. For example, concurrent access (RC5) ensures that transactions are handled in a way that preserves the database's integrity, and integrity is a security aspect covered by CIA (STP4). Integrity in the CIA means that unauthorized data modifications are prevented, and data is modified through transactions. So, when unauthorized people cannot perform

transactions, it ensures the correctness and accuracy of the data, making the database both a robust and secure system. Furthermore, as already mentioned, availability is related to robustness, but it is also part of the CIA. Availability in the CIA guarantees that data is accessible when needed, meaning that data is protected from disruptions. People who access the database without proper authorization can make the data unavailable and compromise it, which could break the integrity of the data, leading to a situation where the data is no longer reliable. People can access the database without authorization when databases are exposed to the internet (ST5). Making the database open for public access is a security threat that can lead to the unavailability of data, thus putting both database robustness and security at risk. To reduce this risk, each person who tries to access the database needs to be authenticated (STP1), and those with legitimate access rights need to be authorized (STP2) to ensure they have the right to perform certain actions within the database.

Performance and security are related to each other as well. While security mechanisms, such as encryption (STP12) and auditing (STP13 and STP14), enhance security, they can cause additional processing overhead, affecting query execution time and resource usage. Auditing database login attempts and actions, as well as encrypting large datasets, can be resource-intensive and slow down database performance. Performance degradation due to encryption and auditing results from insufficient CPU (PF1), storage (PF2), and memory (PF3). When these computing resources are increased, more data can be encrypted, and more database actions can be audited. In other words, increased computer resources improve database security.

Moreover, authentication and authorization are critical for securing the database, but implementing them can negatively affect performance, especially when the distance between the user and the database server is long (PF7). The farther users are from the server, the more time it takes to authenticate them and check what actions they are allowed to perform within the database. Implementing multiple authentication steps can increase latency for geographically distant users, meaning that query execution is prolonged. This is why it is important to carefully plan the location of database servers, preferably where most users are. This, in turn, would require knowledge or study of the users' locations.

This thesis has provided information about database robustness, performance, and security. These three aspects can be viewed as forming a database system that can be trusted to function in expected way and to keep users' data safe. For companies, this thesis serves as a guide for evaluating different database options, so that decision-makers can make the best possible decision when choosing a database for their company's use. With the help of this thesis, it can be ensured that the chosen database aligns with the company's operational needs and risk tolerance. Additionally, the thesis

guides companies to keep in mind the possible data growth in the future, so that the essential measures can be implemented to maximize the customer and user satisfaction.

7 Conclusion

This thesis has explored the essential aspects of relational databases, focusing on robustness, performance, and security. Three research questions were presented to address these topics. RQ1: What are the characteristics that define a robust database? RQ2: What are the factors that affect database performance? RQ3: What security threats a database can face, and how to prevent them? Answers to these research questions were found in the literature and research articles, and the findings were compiled into entities. Entities were listed in their own tables, which conveniently summarized the findings.

Chapter 3 included the “Robustness Characteristic” entity, which summarized the key characteristics of a robust database. The findings in Chapter 3 were presented in Table 3.1, providing an answer to RQ1. Table 3.1 listed eight recognized key characteristics that are essential for ensuring that the database can be trusted to handle data without causing any situations that could harm it or make it unavailable to users.

Chapter 4 included the “Performance Factor” entity, which summarized the factors that affect database performance. The findings in Chapter 4 were presented in Table 4.1, providing an answer to RQ2. Table 4.1 listed 11 recognized database performance factors, explaining how query execution time can slow down and what can be done to improve performance.

Chapter 5 included “Security Threat” and “Security Threat Prevention” entities. The “Security Threat” entity summarized different threats that can compromise database security, and the findings were presented in Table 5.1, providing a partial answer to RQ3. Table 5.1 listed six database security threats that could lead to unauthorized access and data leakage. The “Security Threat Prevention” entity, in turn, summarized how the threats presented in Table 5.1 can be prevented. These findings were presented in Table 5.2, which, together with Table 5.1, provided the complete answer to RQ3. Table 5.2 listed 16 recognized countermeasures that can be implemented to prevent unauthorized access to the data.

By focusing on robustness, performance, and security, the thesis provided a comprehensive view of what makes a database trustworthy and efficient, offering valuable guidance for companies in selecting and managing databases to support their business operations. If companies neglect any of these aspects, it could lead to a loss of customer trust and, eventually, a loss of business. Therefore, adopting best practices in database management is not only a technical requirement but also a strategic necessity for ensuring future business success.

References

[1] A. Shahrokni and R. Feldt, “A systematic review of software robustness”, *Information and Software Technology*, vol. 55, no. 1, pp. 1-17, 2012, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2012.06.002 [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0950584912001048?via%3Dihub

[2] M. M. Hassan, W. Afzal, M. Blom, B, Lindström, S. F. Andler, and S. Eldh, ”Testability and Software Robustness: A Systematic Literature Review”, in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pp. 341-348, 2015, ISBN: 978-1-4673-7585-6. DOI: 10.1109/SEAA.2015.47 [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7302472

[3] J. L. Harrington, *Relational Database Design and Implementation*, Morgan Kaufmann, 2016, ISBN: 978-0-12-804399-8. [Online]. Available: https://learning.oreilly.com/library/view/relational-database-design/9780128499023/

[4] J. Cooper, *Database Design and SQL for DB2*, MC Press, 2013, ISBN: 978-1-58347-357-3. [Online]. Available: https://ebookcentral.proquest.com/lib/kutu/detail.action?docID=1158870&pq-origsite=primo

[5] C. Ray, *Distributed Database Systems*, Pearson India, 2024, ISBN: 978-1-282-65252-1. [Online]. Available: https://learning.oreilly.com/library/view/distributed-database-systems/9781282652521/

[6] S. Binani, A. Gutti, and S. Upadhyay, “SQL vs. NoSQL vs. NewSQL- A Comparative Study”, vol. 6, no. 1, pp. 43-46, 2016, ISSN: 2394-4714. DOI: 10.5120/cae2016652418 [Online]. Available: https://www.researchgate.net/publication/309467097_SQL_vs_NoSQL_vs_NewSQL-_A_Comparative_Study

[7] C. Wijesiriwardana and M.F.M. Firdhous, “An Innovative Query Tuning Scheme for Large Databases”, in *2019 International Conference on Data Science and Engineering (ICDSE)*, pp. 154-159, 2019, ISBN:978-1-7281-2087-4. DOI: 10.1109/ICDSE47409.2019.8971483 [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8971483

[8] M. Kleppmann, *Designing Data-Intensive Applications*, O’Reilly Media, Inc., 2017, ISBN: 978-1-449-37332-0. [Online]. Available: https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

[9] R. Stephens, *Beginning Database Design Solutions*, Wiley, 2023, ISBN: 978-1-394-15572-9. [Online]. Available: https://learning.oreilly.com/library/view/beginning-database-design/9781394155729/

[10] D. Nichter, *Efficient MySQL Performance*, O’Reilly Media, Inc., 2021, ISBN: 978-1-098-10509-9. [Online]. Available: https://learning.oreilly.com/library/view/efficient-mysql-performance/9781098105082/

[11] D. Kuhn and T. Kyte, *Expert Oracle Database Architecture: Techniques and Solutions for High Performance and Productivity*, Apress, 2021, ISBN: 978-1-4842-7499-6. [Online]. Available: https://learning.oreilly.com/library/view/expert-oracle-database/9781484274996/

[12] S. Botros and J. Tinley, *High Performance MySQL*, O'Reilly Media, Inc., 2021, ISBN: 978-1-492-08051-0. [Online]. Available: https://learning.oreilly.com/library/view/high-performance-mysql/9781492080503/

[13] D. Korotkevitch, *SQL Server Advanced Troubleshooting and Performance Tuning*, O'Reilly Media, Inc., 2022, ISBN: 978-1-098-10192-3. [Online]. Available: https://learning.oreilly.com/library/view/sql-server-advanced/9781098101916/

[14] T. Taipalus, "Database management system performance comparisons: A systematic literature review", *The Journal of Systems & Software*, vol. 208, pp. 1-16, 2023, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111872 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223002674

[15] S. J. Kamatkar, A. Kamble, A. Viloria, L. Hernández-Fernandez, and E. G. Cali, "Database Performance Tuning and Query Optimization", in *Data Mining and Big Data*, Y. Tan, Y. Shi, Q. Tang, Eds., Cham: Springer International Publishing, 2018, pp. 3-11, ISBN: 978-3-319-93803-5. DOI: 10.1007/978-3-319-93803-5 [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-93803-5

[16] G. Fritchey, *SQL Server 2017 Query Performance Tuning: Troubleshoot and Optimize Query Performance*, Apress, 2018, ISBN: 978-1-4842-3888-2. [Online]. Available: https://learning.oreilly.com/library/view/sql-server-2017/9781484238882/

[17] F. C. Mendes, P. Sarna, P. Emelyanov, and C. Dunlop, *Database Performance at Scale : A Practical Guide*, Apress, 2023, ISBN: 978-1-4842-9711-7. [Online]. Available: https://ebookcentral.proquest.com/lib/kutu/detail.action?docID=30882799&pq-origsite=primo

[18] L. Davidson, *Pro SQL Server Relational Database Design and Implementation: Best Practices for Scalability and Performance*, Apress, 2020, ISBN: 978-1-4842-6497-3. [Online]. Available: https://learning.oreilly.com/library/view/pro-sql-server/9781484264973/

[19] V. B. Ramu, "Optimizing Database Performance: Strategies for Efficient Query Execution and Resource Utilization", vol. 71. no. 7, pp. 15-21, 2023, ISSN: 2231–2803. DOI: 10.14445/22312803/IJCTT-V71I7P103 [Online]. Available: https://www.researchgate.net/publication/372683874_Optimizing_Database_Performance_Strategies_for_Efficient_Query_Execution_and_Resource_Utilization

[20] G. Harrison and M. Harrison, *MongoDB Performance Tuning: Optimizing MongoDB Databases and their Applications*, Apress, 2021, ISBN: 978-1-4842-6879-7. [Online]. Available: https://learning.oreilly.com/library/view/mongodb-performance-tuning/9781484268797/

[21] M. J. Hernandez, *Database Design for Mere Mortals*, Addison-Wesley Professional, 2020, ISBN: 978-0-13-678804-1. [Online]. Available: https://learning.oreilly.com/library/view/database-design-for/9780136788133/

[22] E. Pirozzi, *PostgreSQL 10 High Performance : Expert Techniques for Query Optimization, High Availability, and Efficient Database Maintenance*, Packt Publishing, Limited, 2018, ISBN: 978-1-78847-448-1. [Online]. Available: https://ebookcentral.proquest.com/lib/kutu/detail.action?docID=5379701&pq-origsite=primo

[23] J. Strate, *Expert Performance Indexing in SQL Server 2019: Toward Faster Results and Lower Maintenance*, Apress, 2019, ISBN: 978-1-4842-5464-6. [Online]. Available: https://learning.oreilly.com/library/view/expert-performance-indexing/9781484254646/

[24] S. R. Alapati, D. Kuhn, B. Padfield, *Oracle Database 12c Performance Tuning Recipes: A Problem-Solution Approach*, Apress, 2013, ISBN: 978-1-4302-6188-9. [Online]. Available: https://learning.oreilly.com/library/view/oracle-database-12c/9781430261872/

[25] B. Nevarez, *SQL Server Query Tuning and Optimization*, Packt Publishing, 2022, ISBN: 978-1-80324-262-0. [Online]. Available: https://learning.oreilly.com/library/view/sql-server-query/9781803242620/

[26] J. W. Krogh, *MySQL 8 Query Performance Tuning: A Systematic Method for Improving Execution Speeds*, Apress, 2020, ISBN: 978-1-4842-5584-1. [Online]. Available: https://learning.oreilly.com/library/view/mysql-8-query/9781484255841/

[27] M. M. Rahman, S. Islam, M. Kamruzzaman, and Z. H. Joy, "ADVANCED QUERY OPTIMIZATION IN SQL DATABASES FOR REAL-TIME BIG DATA ANALYTICS", vol. 4, no. 3, pp. 1-14, 2024, ISSN: 2997-9552. DOI: 10.69593/ajbais.v4i3.77 [Online]. Available: https://www.allacademicresearch.com/index.php/AJBAIS/article/view/77

[28] A. Hamidi, A. R. Hamraz, and K. Rahmani, "Database Security Mechanisms in MySQL", vol. 1, no. 1, pp. 1-7, 2022, ISSN: 2789-8601. DOI: https://doi.org/10.70648/naturalscience.v4i1.66 [Online]. Available: https://www.arj.af/index.php/arj/article/view/66

[29] L. Schwarz, "What Is Authorization? Definition & Examples", NetSuite, 2022, [Online]. Available: https://www.netsuite.com/portal/resource/articles/erp/authorization.shtml (Accessed 21.8.2024)

[30] E. Bertino and R. Sandhu, "Database Security—Concepts, Approaches, and Challenges", in *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 1, pp. 2-19, 2005, ISSN: 1941-0018. DOI: 10.1109/TDSC.2005.9 [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1416861

[31] D. Cherry, *Securing SQL Server : Protecting Your Database from Attackers*, Elsevier Science & Technology Books, 2015, ISBN: 978-0-12-801275-8. [Online]. Available: https://ebookcentral.proquest.com/lib/kutu/detail.action?docID=2037826&pq-origsite=primo

[32] P. Strengholt, *Data Management at Scale*, O'Reilly Media, Inc., 2023, ISBN: 978-1-098-13886-8. [Online]. Available: https://learning.oreilly.com/library/view/data-management-at/9781098138851/

[33] N. A. Al-Sayid and D. Aldlaeen, "Database Security Threats: A Survey Study", in *2013 5th International Conference on Computer Science and Information Technology*, pp. 60-64, 2013, ISBN: 978-1-4673-5825-5. DOI: 10.1109/CSIT.2013.6588759 [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6588759

[34] P. A. Carter, *Securing SQL Server: DBAs Defending the Database*, Apress, 2018, ISBN: 978-1-4842-4161-5. [Online]. Available: https://learning.oreilly.com/library/view/securing-sql-server/9781484241615/

[35] S. R. Devara and C. Azad, "Improved Database Security Using Cryptography with Genetic Operators", 2023, pp. 1-11. DOI: 10.1007/s42979-023-01990-z [Online]. Available: https://link.springer.com/article/10.1007/s42979-023-01990-z

[36] M. Malik and T. Patel, "Database Security - Attacks and Control Methods", vol.6, no.1, pp. 175-183, 2016, DOI: 10.5121/ijist.2016.6218 [Online]. Available: https://www.researchgate.net/publication/301277002_Database_Security_-_Attacks_and_Control_Methods

[37] A. Patil and B. B. Meshram, "Database Access Control Policies", in *International Journal of Engineering Research and Applications*, vol. 2, no. 3, pp. 3150-3154, ISSN: 2248-9622. [Online]. Available: https://www.academia.edu/1960562/SW2331503154

[38] B. Jayant. D, U. Swapnaja A, A. Sulabha S, and M. Dattatray G, "Analysis of DAC MAC RBAC Access Control based Models for Security", International Journal of Computer Applications, vol. 104, no. 5, pp. 6–13, 2014, ISSN: 0975-8887. DOI: 10.5120/18196-9115. [Online]. Available: https://www.researchgate.net/publication/284367991_Analysis_of_DAC_MAC_RBAC_Access_Control_based_Models_for_Security

[39] C. S. Jordan, "A Guide to Understanding Discretionary Access Control in Trusted Systems", N. C. S. Center, pp. 1-29, [Online]. Available: https://apps.dtic.mil/sti/citations/ADA392813

[40] Ekran System. "Mandatory access control vs discretionary access control: Which to choose?", 2022, [Online]. Available: https://www.linkedin.com/pulse/mandatory-access-control-vs-discretionary-which-choose-ekran-system/ (Accessed 19.8.2024)

[41] S. Osborn, "Mandatory access control and role-based access control revisited", in *RBAC '97 Proceedings of the second ACM workshop on Role-based access control*, ACM Press, 1997, pp. 31–40, ISBN: 978-0-89791-985-2. DOI: 10.1145/266741.266751. [Online]. Available: http://portal.acm.org/citation.cfm?doid=266741.266751

[42] RedHat. "What is role-based access control (RBAC)?", [Online]. Available: https://www.redhat.com/en/topics/security/what-is-role-based-access-control (Accessed 20.8.2024)

[43] R. S. Danturthi, *Database and Application Security: A Practitioner's Guide*, Addison-Wesley Professional, 2024, ISBN: 978-0-13-807373-2. [Online]. Available: https://learning.oreilly.com/library/view/database-and-application/9780138073725/

[44] L. Davidson, *Pro SQL Server Relational Database Design and Implementation: Best Practices for Scalability and Performance*, Apress, 2020, ISBN: 978-1-4842-6497-3. [Online]. Available: https://learning.oreilly.com/library/view/pro-sql-server/9781484264973/

[45] U. T. I. Igwenagu, A. A. Salami, A. S. Arigbabu, C. E. Mesode, T. O. Oladoyinbo, and O. O. Olaniyi, "Securing the Digital Frontier: Strategies for Cloud Computing Security, Database Protection, and Comprehensive Penetration Testing", Journal of Engineering Research and Reports, vol. 26, no. 6, pp. 60-75, 2024, ISSN: 2582-2926. DOI: 10.9734/JERR/2024/v26i61162 [Online]. Available: http://eprints.go4mailburst.com/id/eprint/2262/