EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Increasing awareness of SQL anti-patterns for novices

Mathon, Leonardo E.P.A.

Award date:
2022

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Departement of Mathematics and Computer Science

Increasing awareness of SQL anti-patterns for novices

a **Master Thesis** by

L.E.P.A. Mathon

Supervisors:
prof.dr. George Fletcher
ir. Daphne Miedema
prof.dr. Alexander Serebrenik

Eindhoven, September 2022

Abstract

Structured Query Language (SQL) is a database language for interacting with relational databases. Various barriers to learning SQL have been discovered in previous research [35, 28, 23, 2]. These barriers, combined with incomplete knowledge, often lead to SQL users writing erroneous queries [20, 4, 41, 32]. Research shows us that this can lead to anti-patterns (code smells) that harm software quality [17, 24]. Anti-patterns also influence education, because students may use anti-patterns without realizing it. The result-set of a query containing an anti-pattern is often correct, or appears to be correct. However, anti-patterns are harmful because they make the resulting program or statement inefficient, harder to maintain, and easier to misread.

Anti-patterns for programming languages were first mentioned by Koenig in 1995 [16]. For databases in general, and SQL specifically, anti-patterns were defined by Karwin in 2010 [15]. Early detection of anti-patterns can help developers avoid technical debt, as writing 'smelly' SQL may result in poor performance or erroneous results as shown by [17, 24]. This is why much of the research on SQL anti-patterns has focused on detecting them [25, 18, 5, 11, 30]. However, prevention of anti-patterns is a more effective strategy than detection of patterns in production code. As SQL anti-patterns do not seem to be a core topic in SQL education, students may not be aware of the existence and effect of anti-patterns. Our goal in this study is to increase awareness of SQL anti-patterns for novices, and to provide them with tools on how to resolve them.

Out of all anti-patterns defined in the literature, only a few anti-patterns can be detected in raw SQL queries. For each of these anti-patterns, we developed a novel detection method using a combination of regular expressions and query parsing and made it available through a web application, called QuerySandbox.

Furthermore, we conducted a user study with students to see if QuerySandbox helps novices learn, fix, and avoid writing anti-patterns. In this study, we gave our participants a set of query formulation problems they must answer using our application. These problems were designed in such a way that they should trigger anti-patterns in query formulation for students that are unaware of them. The interface of QuerySandbox provides tools to solve the problem. We first measured their awareness of SQL anti-patterns through a pre-test in questionnaire form. Then, we had a session in which the learners work with QuerySandbox to practice with, and learn more about, anti-patterns. Finally, in a post-test, we checked whether they perform better on the exact same questions as the pre-test.

Out of our findings, we can conclude that our anti-pattern detection toolkit provides a novel way of learning and understanding anti-patterns. We analyzed our participant's edit behavior in the user study and learned that queries containing anti-patterns were eventually fixed, hinting at a successful intervention by QuerySandbox. Furthermore, after the session with QuerySandbox, students could recognize and resolve anti-patterns from a given query. Therefore, we conclude that QuerySandbox helps novices in writing clean SQL queries as it successfully made them aware of a specific subset of anti-patterns.

Acknowledgements

I would like to express my gratitude to my supervisors, prof. dr. George Fletcher and Daphne Miedema for their advice, encouragement, and support during both the seminar in data management as well as my Thesis. Because of our interaction, in the form of weekly meetings, email, and through Microsoft Teams, I never felt that I was working in isolation, and I always felt assured that I was doing meaningful work. I would also like to especially thank Fenia Aivaloglou, an assistant professor at Leiden University, for giving valuable ideas and feedback on my user study design. This resulted in a better, more polished user study which has led to valuable insights. Furthermore, I would like to thank my parents, my girlfriend, and my friends, who even though they might not have understood what I was doing, or why I was doing it, have always encouraged me and helped me to improve.

Contents

Contents vii

1 Introduction 1
1.1 Thesis objectives 3
1.2 Thesis structure 3
1.3 Contributions 3

2 Preliminaries 5
2.1 DB & DBMS 5
2.1.1 Table structure 6
2.1.2 Table relationships 7
2.2 Database languages 9
2.3 SQL 9
2.3.1 History of SQL 10
2.3.2 SQL language elements 10
2.3.3 SQL syntax 11
2.3.4 Increasing SQL performance using database indexes 14
2.3.5 Alternative to SQL: Object Relational Mapping 14
2.4 Regular Expressions 15
2.5 A detailed look at SQL anti-patterns 16
2.5.1 Example schema 17
2.5.2 A list of anti-patterns 18
2.6 Summary 28

3 Related work 29
3.1 SQL learning barriers 29
3.2 Literature on anti-patterns 31
3.3 Literature on anti-pattern detectors 32
3.4 Summary 33

4 Anti-pattern detector 34
4.1 Defining anti-pattern scope 34
4.2 Detector implementation 35
4.2.1 Implicit Columns 35
4.2.2 Poor Man's Search Engine 36
4.2.3 Random Selection 36
4.2.4 Fear of the Unknown 36
4.2.5 Ambiguous Groups 37
4.2.6 Spaghetti Query 39
4.3 Distributing the detector 42
4.3.1 Python package with CLI 43
4.3.2 RESTful API 45

4.3.3 Web application . . . 45
4.4 Summary . . . 45

5 Anti-pattern application **47**
5.1 User interaction and user experience . . . 47
5.2 Application requirements . . . 48
5.3 Application design . . . 50
5.3.1 Low-fidelity mock-ups . . . 50
5.3.2 High-fidelity final design . . . 50
5.3.3 Application architecture & implementation . . . 51
5.4 Conclusion to RQ1 . . . 60
5.5 Summary . . . 60

6 User study **61**
6.1 Participants . . . 62
6.2 Materials . . . 62
6.2.1 Pre-test materials . . . 62
6.2.2 Main test materials . . . 65
6.2.3 Post-test materials . . . 65
6.3 Procedure . . . 67
6.4 Results . . . 68
6.4.1 Results pre-test . . . 68
6.4.2 Results main test . . . 68
6.4.3 Results post-test . . . 71
6.5 Conclusion to RQ2 . . . 76
6.6 Summary . . . 77

7 Discussion **78**
7.1 Discussion on findings . . . 78
7.1.1 Participants were unaware of SQL anti-patterns . . . 78
7.1.2 QuerySandbox raised awareness of SQL anti-patterns . . . 79
7.1.3 Participants found aggregation difficult . . . 79
7.1.4 Participants found QuerySandbox usable . . . 80
7.2 Limitations . . . 80
7.3 Summary . . . 81

8 Conclusion **82**
8.1 Future work . . . 83

Bibliography **85**

Appendix **89**

A Mock-ups of QuerySandbox **89**

Chapter 1

Introduction

We live in the information age where people are bombarded with endless amounts of data, information, and messages. It is the nature of our society to be constantly exposed to this information, be it through social media, newspapers, television, the internet, or in one way or another. Large amounts of data, also known as “big data” is collected from a variety of sources on a daily basis and stored in databases for both use and storage.

The term big data is used to describe the collection, storage, analysis, and dissemination of large volumes of data. As the amount of data generated and processed continues to increase, there is a need to efficiently store this data. This need can be met through the implementation of databases and database management systems (DBMS). A database can be described as a collection of stored data organized in a structured manner. It can be classified into many different kinds, among which the Relational (SQL, Oracle), Object-oriented (ObjectDatabases++, ObjectStore) and NoSQL (MongoDB, Cassandra) databases. A DBMS is a complex software system, which provides the ability to create, read, update, and delete data by storing it in an organized way in a database. Thus, managing the database.

Most of the time, we interact with these databases without even realizing it. With the advent of the internet, the importance of databases has only increased. Massive amounts of data is generated by a variety of different sources, such as devices in the Internet of Things domain, and applications like websites and mobile phone apps. These devices, applications but also humans can retrieve and manipulate the data from these databases. This is useful for tasks such as visualizing data or performing data analysis. The act of retrieving data from databases is commonly referred to as “querying” data from a database and can be done in a variety of ways.

One of the most common ways to query a database is to use a database query language (DBQL). DBQLs are a group of languages that allow users to create queries that have the ability to access, modify or delete data from the database. One of the most common DBQLs is Structured Query Language (SQL), which has been dominant for decades in the database industry, although the adoption of other more modern, standardized languages, like SPARQL, GraphQL, OQL (Object Query Language), or the usage of Object Relational Mappings has increased in recent years.

An Object-Relational Mapping (ORM) is an alternative method of interacting with a database. The use of an ORM is a common practice that allows developers to utilize objects written as structs or classes and utilize the object-oriented programming paradigm to interact with the data in a database. The main benefit of using an ORM is its ability to greatly reduce the amount of work required to develop applications that are capable of interacting with the data in the database. ORMs reduce the development time, because developers do not need to write raw SQL queries from scratch, but instead focus on the design of the application or logic.

Nonetheless, SQL remains one of the most popular data retrieval languages due to its simplicity, performance, and ease of implementation. According to the 2021 Developer Survey by Stack Overflow, SQL is the fourth most popular language out of all programming, scripting, and markup languages, with only Python, HTML/CSS, and JavaScript above it [31].

Another factor emphasizing the importance of SQL is that most of the current popular ORMs and DBQLs on the market are SQL-based. It is, however, important to know that ORMs and DBQLs are not a replacement for SQL. Although ORMs can offer a better development experience by using high level abstractions, they have their own drawbacks. The overhead in code generated by ORMs is a common complaint among developers, as it slows application performance and makes it more difficult to maintain. Secondly, ORMs are sometimes not reliable in the SQL code they generate, impacting the performance. Hence, developers still need to know SQL for cases when the ORM is not able to provide the performance or functionality that is required.

For this reason, it is important that SQL is still taught to novice developers. Learning SQL must be accompanied by learning more about the different concepts that surround it. These concepts include the underlying database theory, but also more practical aspects such as the SQL syntax. Fortunately, the vast majority of developers are familiar with SQL in some shape or form.

Having a good understanding of the SQL syntax is important, as it provides the foundation for all SQL queries, and is not something the majority of people can easily grasp. Various barriers to learning SQL have been discovered in previous research [35, 28, 23, 2]. These barriers, combined with incomplete knowledge, often lead to SQL users writing erroneous queries [20, 4, 41, 32], showing that SQL has become an often-misused language that many people rely on without fully understanding it.

The first common type of error is a syntactical error. A syntactical error indicates that the entered query is not a valid SQL query. In this instance, a DBMS would generate an error message since it is unable to execute the query. As a result, the error is detected at runtime and is usually simple to repair.

The second type of error is a semantic error. These errors are classified as frequent mistakes where a valid SQL query was entered, but the returned result is incorrect for the task. In contrast to syntactical errors, the developer does not need to repair the syntax of the query. However, if they miss a specific detail within the query, it could result in incorrect data.

The last type of error could be considered less of an error, but more of a highly counterproductive and ineffective group of queries that can be closely related to semantic errors. These are so-called SQL code smells or anti-patterns, which are common mistakes in SQL queries that appear to be a good, and sufficiently efficient solution to a particular task. However, these queries tend to cause logical or functional errors, performance issues, or security vulnerabilities. Queries plagued by anti-patterns can also be fragile and prone to bugs making the code hard to maintain. However, we believe anti-patterns also influence education, because students may use anti-patterns without realizing it. The result-set of a query containing an anti-pattern is often correct or appears to be correct. However, anti-patterns are harmful because they make the resulting program or statement inefficient, harder to maintain, or easier to misread.

The first book covering SQL anti-patterns was only published in 2010. Since then, only a handful of other research papers on this topic have been published. The current research on anti-patterns primarily focuses on the examining of prevalence, impact, evolution, and performance issues related to SQL anti-patterns as well as anti-pattern detection in existing codebases. However, we believe the prevention of anti-patterns is a more effective strategy than the detection of patterns in production code. As SQL anti-patterns do not seem to be a core topic in SQL education, students may not be aware of the existence and effect of anti-patterns. Our goal in this study is to increase awareness of SQL anti-patterns for novices and to provide them with tools on how to resolve them.

In this thesis, we will contribute to the education of novices in SQL by taking a deeper look at SQL anti-patterns in novice queries. Our hope is to make SQL anti-patterns into an academic, practical, tangible, and relevant topic. We will first introduce the main concepts surrounding databases and SQL, then review the existing literature on SQL errors, misconceptions, SQL education as well as anti-patterns, and develop an easy-to-use static detection tool to detect anti-patterns in raw queries. Finally, we conduct a user study to verify if our tool can help novice users learn, fix and prevent anti-patterns when writing SQL queries. We hope that this effort can contribute to the field of database education and enable novices to study SQL and develop queries in a safe environment, whilst avoiding common anti-patterns.

1.1 Thesis objectives

In this thesis, we address the following two main research questions:

> RQ1 How can a tool be designed to help novice users understand anti-patterns?
> RQ2 Does the intervention of our tool help novice users learn, fix and prevent writing SQL queries with anti-patterns?

To be able to answer RQ1, we must determine which anti-patterns can occur in raw SQL queries. Therefore, In addition to RQ1, we define the following subquestion:

> RQ1.1 What SQL anti-pattern(s) can occur in raw queries written by novices?

The preceding questions are motivated by the current literature on anti-patterns in SQL queries, which state that there are patterns in SQL queries that impair software quality and maintainability, hence impeding application and database performance. In addition, novice programmers may find it challenging to learn about SQL anti-patterns in general, and how to avoid them. This is because most existing literature and tools are not targeted toward novices. Hence, by helping novice users understand anti-patterns by means of an educational intervention would provide a better path to learning how to write good SQL queries. We aim to make SQL anti-patterns into a tangible and learnable concept.

All in all, the main objectives of this research are to (1) determine which SQL anti-pattern(s) can occur in raw queries written by novices, (2) develop a tool that can educate novice users about SQL anti-patterns, and (3) evaluate the tool by developing and validating a method to help novice programmers understand anti-patterns by means of an educational intervention.

1.2 Thesis structure

This thesis is structured as follows: in chapter 2 we establish some context for the remainder of the thesis. This includes a brief explanation of databases, the relational data model, SQL, ORMs, regular expressions, and finally a list of known anti-patterns from the literature. In the next chapter, chapter 3, we will review the current literature on SQL errors and anti-patterns to gain a better understanding of where we stand in our knowledge of this area. The first research question (RQ1) will be answered using the process described in both chapter 4 and chapter 5. chapter 4 will define the scope of anti-patterns to be implemented in our detection tool by analyzing which SQL anti-pattern can occur in raw SQL queries, as well as the methods we are using to detect them. In the next chapter, chapter 5, we will be going over the development of our web application, QuerySandbox. In chapter 6 will describe our user study as well as our results and answer research question 2 (RQ2). In chapter 7 we will discuss our key findings and present limitations and future work. In the last chapter, chapter 8, we will conclude this thesis with a brief recap.

1.3 Contributions

In this thesis, we will contribute to the field of anti-patterns, as well as SQL, by developing and validating a tool to automatically detect common SQL anti-patterns written by novice programmers. This tool will be based on a novel technique to classify novice SQL queries as anti-patterns or standard queries. This allows us to make SQL anti-patterns more tangible, learnable, and relevant by focusing on the most common anti-patterns that novice programmers make. In addition, we will identify common anti-patterns in raw, novice SQL queries. Furthermore, we will assess the

performance of our tool. We hope that making SQL anti-patterns more tangible, learn-able, and relevant, will contribute to the field of database education.

Chapter 2

Preliminaries

This chapter will describe some concepts that appear frequently throughout this thesis. This ensures that both the writer and reader speak the same language. As well as this, the aim is to help the reader to build a solid grounding in the technical issues that are presented.

We will begin by reviewing basic database concepts and how they can be applied. Following that, we will look at some of the languages that can be used to access and manipulate the database with our main focus on SQL. Next we look at regular expressions and how they can be used to parse strings and preform pattern matching. Finally, we will define a list of SQL anti-patterns from the literature.

2.1 Databases and Database Management Systems

A database, or DB, is an organized collection of structured information or data, usually stored electronically in a computer system [29]. Management of a database is usually left to a separate system called a Database Management System, or DBMS for short, that facilitates interaction with the data to its users using a set of programs that accesses the data [38]. A database system refers to the data, the DBMS, and any applications that are linked to it. This is frequently abbreviated to just database. A database system's primary goal is to give users a high-level overview of the data. This means that the system conceals certain aspects of data storage and maintenance.

Since their inception in the early 1960s, databases have evolved significantly. The first systems used to store and manipulate data were navigational databases such as the hierarchical database (based on a tree-like model that only allowed a one-to-many relationship) and the network database (a more adaptable model that allowed multiple relationships). Despite their simplicity, these early systems were rigid and most of the time not flexible enough for certain tasks. Relational databases became popular in the 1980s, followed by object-oriented databases in the 1990s. NoSQL databases emerged more recently as a response to the growth of the internet and the need for faster speed and processing of unstructured data. Nowadays, cloud databases and self-driving databases are breaking new ground in terms of data collection, storage, management, and utilization. However, despite these new technologies, relational databases are still the most popular choice for all sorts of projects [36]. Therefore, in this study, we will focus on relational databases only.

Databases evolve over time. This happens when data is added or removed, or when the structure of certain tables changes. A collection of data stored in the database at a specific moment in time is called an instance of a database. The structure of a database is commonly referred to as database schema, which defines the overall design of the database.

The data model is a collection of conceptual tools for representing data, data relationships, data semantics, and consistency requirements that lie beneath the structure of a database. There are many different data models. For example, relational databases use the relational model. This model was first described by Codd in 1970 [10] and became the most well-known and the de facto

industry standard for data modeling.

2.1.1 Table structure

Data in the most common types of databases in use today is often described in rows and columns in a series of tables to facilitate data processing and searches. The data can then be used, managed, modified, updated, monitored, and organized with ease. Each row is referred to as a record in database terminology. Sometimes, a record is also known as an object or entity. The records in a table are the objects that you are interested in, such as customers in a store database or products in a product warehouse. A table's name is typically chosen to represent the type of object that it contains. A table of customers, for example, would be named "customer," whereas a table of orders would be named "order." The naming convention for tables may differ between developers. Some developers prefer to use plural names, while others prefer singular names. However, most developers use singular names for tables to indicate that the table contains only one type of record. The ordering of the rows inside a table is immaterial, meaning that the order in which records appear in the rows of a table is often irrelevant. A column in a table is called a field or attribute and represents a single value for a certain record. A field has a certain type, such as an integer, float, or text. In other words, the field type represents the type of data you are storing. A field may occasionally be optional, allowing it to be left blank when it is not required. We call such a field a nullable field. A field must also occasionally be unique. That is, two records of a table with the exact same value for a particular field are not permitted. To summarize, a table is a collection of records which in turn are composed of fields or attributes containing certain values. These values represent various types of information for a certain record that could sometimes be nullable or unique.

Every table stored in a database has a primary key that serves as a unique identifier for that table. The majority of the time, this is a random ID with no meaningful relevance outside of the database. An example of such a table can be found in Table 2.1, showing a customer table. The primary key in this table is the customer ID (cID), which is an automatically incremented number that has no meaning outside this database. However, sometimes the primary key is an intuitively chosen property of the object were are describing in the table. We might for example remove the cID column from Table 2.1 and replace it with the person's social security number. We know this number is unique to every person, so we can use it as a primary key.

customer			
cID	**cName**	**street**	**city**
⋮	⋮	⋮	⋮
855	Bo	Kastelenstraat	Eindhoven
856	Vincent	Zangstraat	Weert
857	George	Bekelaan	Budel
858	Perry	Pannekoekstraat	Rotterdam
⋮	⋮	⋮	⋮

Table 2.1: An example of a customer table in a database with primary key cID.

In some cases, the primary key is made up of multiple columns, this is called a composite primary key. This type of key, for example, is used when there is a many-to-many relationship between two tables that are joined by a third table commonly called join, junction or linking table. This table will then have a composite primary key consisting of the primary keys of the linked tables. Table 2.2 shows an example of a linking table. This table links a customer with a product by using the primary key of the customer table (cID) and the primary key of the product table (pID). The primary key of this purchase table is the composite key pair (cID, pID).

purchase	
cID	pID
⋮	
855	22
855	48
855	46
856	87
⋮	

Table 2.2: An example of a linking table, linking a customer table with a product table. The primary key is the pair (cID, pID).

One can link different tables through the concept of foreign keys that act as the glue between tables. Foreign keys are keys from a table that refer to the primary key of another table. This can be used to enforce a certain type of relationship. For example, if one wanted to ensure that every order comes from a valid customer. Table 2.3 shows an order table that has two foreign keys. The first foreign key is the customer ID (cID) associating an order with a customer from the customer table. The second foreign key is the product ID (pID) associating an order with a product from the product table.

order			
oID	cID	pID	quantity
		⋮	
209	433	12	2
210	128	12	1
211	855	57	1
212	649	10	8
		⋮	

Table 2.3: An example of an order table with primary key oID and foreign keys cID and pID.

2.1.2 Table relationships

Tables in a relational database can be related to one another. These so-called relationships can be established using foreign key constraints. There are three types of relationships that can occur between tables. The first being a one-to-one relationship.

One-to-one relationship

A one-to-one relationship exists when one entity in one table is associated with exactly one entity in another table. Table 2.4 shows a one-to-one relationship between the customer and the address table. This implies that a customer has one address and each address represents one customer. This one-to-one relationship is illustrated in an Entity Relationship Diagram (ERD) shown in Figure 2.1. A one-to-one relationship is not often used, because this type of relationship is rarely a natural way to model a situation. Suppose another customer is added to the database, that lives at the exact same address as the customer with ID of 855, then there would be a duplicate address added to the address table that only has a different address ID. One possible application of this relationship is when a portion of the information of an entity described in some table is optional. This way, instead of having a number of nullable fields in one large table, you can divide it sensibly into a mandatory table and an optional table.

customer	
cID	cName
⋮	
855	Bo
856	Vincent
857	George
858	Perry
⋮	

address				
aID	street	city	postalcode	cID
⋮				
10	Kastelenstraat	Eindhoven	5899CK	855
11	Zangstraat	Weert	4201OK	856
12	Bekelaan	Budel	1337AB	857
13	Pannekoekstraat	Rotterdam	4401EB	858
⋮				

Table 2.4: An example of a one-to-one relation between a customer and an address with foreign key cID.

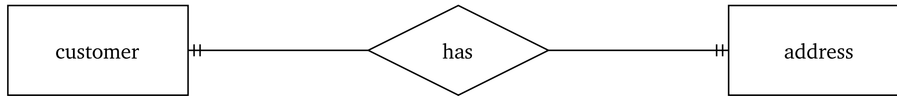


Figure 2.1: An Entity Relationship Diagram (ERD) in crow's foot notation showing a one-to-one relation.

One-to-many relationship

A better way to model the relation shown in Table 2.4, is to use a one-to-many relation. A one-to-many relationship exists when one entity in one table is associated with one or more entities in another table. In our example, a customer would still be associated with one address, but now an address can be associated with multiple customers. An example of this relation is shown in Table 2.5 and in Figure 2.2.

customer		
cID	cName	aID
⋮		
855	Bo	10
856	Vincent	11
857	George	12
858	Perry	13
⋮		

address			
aID	street	city	postalcode
⋮			
10	Kastelenstraat	Eindhoven	5899CK
11	Zangstraat	Weert	4201OK
12	Bekelaan	Budel	1337AB
13	Pannekoekstraat	Rotterdam	4401EB
⋮			

Table 2.5: An example of a one-to-many relation between a customer with foreign key aID and an address.

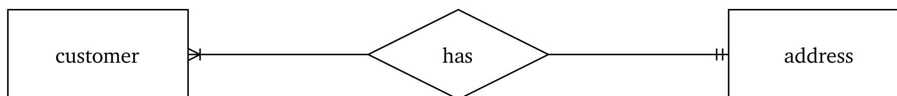


Figure 2.2: An Entity Relationship Diagram (ERD) in crow's foot notation showing a one-to-many relation.

Many-to-many relationship

Lastly, we have the many-to-many relationship which indicates that multiple entities in one table can be associated with more than one entity in another table. We have already seen an example of this relationship, namely in Table 2.2, where we linked customers to products using a link table. Here, a customer can purchase many products and a product can be purchased by many customers. Figure 2.3 shows an ERD of this type of relationship.

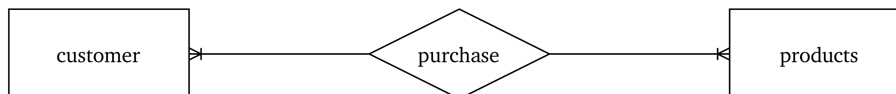


Figure 2.3: An Entity Relationship Diagram (ERD) in crow's foot notation showing a many-to-many relation.

2.2 Database languages

Database languages are a subset of programming languages that allows developers to define and access databases. A database system provides a data-definition language, as well as a data-manipulation language. The former is used to specify the database schema, that is, the underlying organization of the data in the database. The latter is used to express database queries and mutations. A database query is used to locate data based on the database schema, while a database mutation is used to modify data in the database. A well-known and common database language is Structured Query Language (SQL) and is the fourth most popular language out of all programming, scripting, and markup languages [31], meaning that out of all database languages, SQL is the most popular one used today. This emphasizes the importance of SQL in computer science education. Other common database languages include SPARQL [33] and OQL [13].

Database languages can be categorized into four types. The first two types are the *Data Definition Languages (DDL)* and the *Data Manipulation Languages (DML)* and are usually included in the database system. A data definition language is a collection of unique commands that allows us to design and alter the database structure and information, such as metadata. These commands let you create, modify, and delete database structures including schemas, tables, and indexes. A data manipulation language is a collection of unique commands that allows us to access and alter data stored in existing schema objects. These commands are used to perform database operations such as inserting, deleting, updating, and retrieving data. The DML commands that deal with the retrieval of the data from a database are known as a *Data Query language (DQL)*. The third type is a *Data Control Language (DCL)* which is a collection of unique commands used to manage user access in a database system. To perform any operation or query in the database system, we need data access rights. The DCL statements are used to control this user access. These statements are used to give and revoke data or database access to users. The final type of database languages is the *Transaction Control Language (TLC)* which is a set of unique instructions that deal with database transactions. A transaction is a grouping of linked tasks that the DBMS software treats as a single execution unit. As a result, transactions are in charge of carrying out various tasks within a database. TCL commands are used to execute or roll back the adjustments made using DML commands.

2.3 SQL

In this section we provide a quick primer to SQL, including its history, the language elements, and the basic SQL syntax. This section is not intended to be a comprehensive guide to SQL.

2.3.1 History of SQL

SQL [8] was created at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s after they read a series of papers on the relational model of data that was introduced by Codd [10]. Prior to the work of Chamberlin and Boyce, accessing or referencing of data in relation format was done using languages based on first-order predicated calculus. They, however, identified a problem with these languages, namely that they were too hard to use. Since the costs of program creation, maintenance and modification had been rising rapidly and there was an increasing need to bring non-professional computer users into communication with databases, there was also a need for a simpler query language.

Chamberlin and Boyce's initial effort at a relational database language, called SQUARE, was still too difficult to use owing to its subscript and superscript syntax. They began development on a sequel to SQUARE after transferring to the San Jose Research Laboratory of IBM in 1973. The sequel to SQUARE which was originally called SEQUEL. SEQUEL, renamed SQL after a trademark issue, was intended for non-computer specialists that would be willing to learn to interact with a computer at a reasonably high level. The purpose of SQL was to provide the user with a uniform template for expressing simple queries.

IBM began building commercial solutions based on their System R prototype after testing SQL at client test locations to assess the usability and feasibility of the system. This is what initially fueled the language's popularity.

The standard ”Database Language SQL” language specification was officially adopted by ANSI and ISO standard bodies in 1986 with the latest version of the standard released in 2016 [1].

Nowadays, SQL is the most popular query language [31] being an essential component of not only computer science curriculum, but also in other engineering curricula such as data science, applied physics, biomedical sciences, and mechanical engineering.

Although there are different versions and dialects of SQL, most of them are based on a similar standard meaning that switching versions or dialects is relatively easy.

2.3.2 SQL language elements

The SQL language has a relatively straightforward syntax, consisting of several language elements. We will now go over the basic SQL syntax and language elements, providing all necessary information to understand the most common anti-patterns.

The first language element are the *keywords*. These are words that are (pre-)defined in the SQL language. There exists reserved and non-reserved keywords. SQL keywords and other symbols that have special meanings when processed by the Relational Engine are examples of reserved keywords. They should not be used as database, table, column, variable, or other object names. In contrast, non-reserved keywords are SQL keywords that are not currently used by the SQL Server Database Engine. They can be used as database, table, column, or other object names without causing any problems.

The second language element are the *identifiers*. Identifiers are user-defined names for databases, tables, columns, variables, or other SQL objects. For example, a column in a table can be identified by a column name defined by one of its users.

The third language element are *clauses* which are sub-elements of the SQL language. They are typically used to group elements together or to restrict the result of a query. For example, the `WHERE` clause can be used to restrict the result of a query by specifying conditions. There are many clauses available in the SQL language, including the `SELECT`, `FROM`, `WHERE`, `GROUP BY` and `ORDER BY` clauses which are often used.

The fourth language element are *expressions* which are built-in functions, variable names, and operators. For example, a variable can be assigned a value using the assignment operator “=” or the addition operator “+” can be used to add two values. Some common SQL expressions are: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, and `DISTINCT`. These can be used to count the number of rows in a table, sum the values in a column, find the average value in a column, find the minimum or maximum

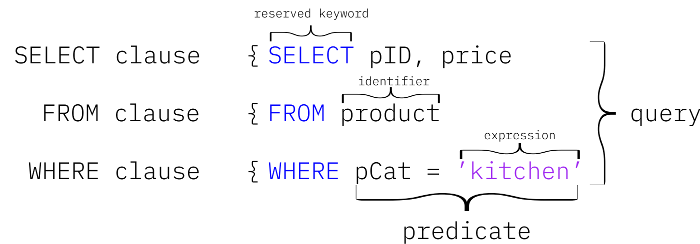


Figure 2.4: A chart showing a SQL query consisting of different language elements.

value in a column, or return only the unique values in a column.

Predicates are the fifth language element in SQL. A predicate is a logical expression that can be evaluated to `true`, `false`, or `unknown`. A predicate can be used in the `WHERE` clause of a query to remove rows for which the result of the predicate is `false`, or `unknown`. For example, the `BETWEEN` predicate can be used to check if a value is in a specified range.

Queries and *statements* are the last two language elements in SQL. Queries are used to retrieve data from the database, whereas statements are used to manipulate the database. For example, the `SELECT` statement is used to retrieve data from the database, whereas the `INSERT` statement is used to insert data into the database.

Figure 2.4 depicts a SQL query made up of various language elements. It is clear that some language elements are the building blocks of others.

2.3.3 SQL syntax

The language elements described above can be used to create SQL queries. For our study, we are only interested in basic `SELECT` queries, that only select data from a database. This section will provide a quick refresher on basic `SELECT` queries.

Retrieving columns

`SELECT` queries are the most basic type of SQL query and are used to retrieve data from a database. The most basic form of a `SELECT` query is: `SELECT column_name FROM table_name`. This query will select the specified column from the table with the name table name. To select specific columns, you can use a comma-separated list of column names after the SELECT keyword as shown in Query 2.1.

```
SELECT column_name1, column_name2
FROM table_name
```

Query 2.1: A simple SELECT query.

Sometimes, we might want to select all columns from a table. In that case, the wildcard character * as shown in Query 2.2 can be used.

```
SELECT *
FROM table_name
```

Query 2.2: A simple SELECT query that selects all columns.

We can also use SQL expressions in the `SELECT` clause. Query 2.3 shows an example where we use the SQL expression `column_name1 + column_name2`.

```
SELECT column_name1, column_name2, column_name1 + column_name2
FROM table_name
```

Query 2.3: A simple SELECT query that uses a SQL expression.

Filtering result-set based on conditions

`SELECT` queries can also be used with a `WHERE` clause to select only rows that match certain criteria. The `WHERE` clause is used to specify conditions that rows must meet in order to be selected. For example, Query 2.4 selects all rows from the table `table_name` where the value of `column_name1` is greater than 10.

```
SELECT *
FROM table_name
WHERE column_name1 > 10
```

Query 2.4: A SELECT query with a WHERE clause.

Retrieving only distinct rows

When writing `SELECT` queries, we might sometimes want to select distinct rows from a table. This can be done by adding the `DISTINCT` keyword after the `SELECT` keyword as shown in Query 2.5. When writing `SELECT` queries that select distinct rows from a table, we must specify all of the columns that we want to select. This is because the DISTINCT keyword only applies to the columns that are listed after it.

```
SELECT DISTINCT column_name1
FROM table_name
```

Query 2.5: A SELECT query with a DISTINCT clause.

Ordering result-set

Adding an `ORDER BY` clause to a `SELECT` query enables us to sort the rows that are returned. The `ORDER BY` clause is used to specify the column or columns that should be used to sort the result-set. For example, Query 2.6 sorts the rows returned by the query in ascending order according to the values in `column_name1`. To sort in descending order, you can replace the `ASC` keyword by `DESC`.

```
SELECT *
FROM table_name
ORDER BY column_name1 ASC
```

Query 2.6: A SELECT query with an ORDER BY clause.

Limiting or offsetting results-set

We may want to limit the number of rows returned in the result-set at times. This is frequently done in order to avoid loading too much data at once. The `LIMIT` clause can be used to limit the number of rows in the result-set. As an argument, the `LIMIT` clause accepts an integer value. The

LIMIT clause can also be combined with the OFFSET clause. As an argument, the OFFSET clause accepts an integer value. The integer number given to the OFFSET clause specifies where in the table we want to begin choosing rows from. Query 2.7, for example, will retrieve the 100 rows starting from row 5 in the table.

```
SELECT *
FROM table_name
LIMIT 100 OFFSET 5
```

Query 2.7: A SELECT query with a LIMIT clause.

Grouping result-set

SELECT queries can also be used in conjunction with GROUP BY and HAVING clauses to group rows together and select only groups that meet certain criteria. The GROUP BY clause is used to specify the column or columns that should be used to group the rows. The HAVING clause is used to specify conditions that groups must meet in order to be selected. The HAVING clause is similar to the WHERE clause in that it is used to filter rows. However, HAVING is used after grouping the rows, and can access information about groups, not just rows. This means that we can use aggregate functions in the HAVING clause to filter groups based on aggregate values. Some use cases of the GROUP BY clause include finding the average value of a column for each group, or counting the number of rows in each group. For example, Query 2.8 groups the rows in the table table_name by the values in column_name1 and selects only groups where the average value of column_name2 after grouping is greater than 10. Furthermore, we count the number of rows in each group using the COUNT function.

```
SELECT column_name1, COUNT(column_name3)
FROM table_name
GROUP BY column_name1
HAVING AVG(column_name2) > 10
```

Query 2.8: A SELECT query with a GROUP BY clause and a HAVING clause.

Joining multiple tables

Sometimes, we would like to query data from two or more tables. In order to do that, we must join the tables together using a JOIN clause. For example, Query 2.9 joins the tables table_name1 and table_name2 together, and selects all columns from the joined table. When joining tables together, we must specify the join conditions to ensure that we only join rows that are related. In the example of Query 2.9, we join the tables on the values stored in table_name1.column_name1 and table_name2.column_name1. We can also specify the type of join we would like to use. There are multiple types of JOIN clauses. For our study, we will only be working with INNER JOINS as shown in the example. Note that the WHERE clause could also be used to perform INNER JOINS.

```
SELECT *
FROM table_name1
JOIN table_name2
ON table_name1.column_name1 = table_name2.column_name1
```

Query 2.9: A SELECT query with a JOIN clause.

Using subqueries

Subqueries are a slightly more advanced topic. Subqueries are like regular queries, but are enclosed in parentheses. They can be used to provide the enclosing query with data. For example, Query 2.10 selects all rows from `table_name1` where the value stored in `column_name1` is greater than the maximum value of `column_name2` in `table_name2`. Subqueries are most commonly used in the `WHERE` or `HAVING` clauses but it is also possible to include a subquery in the `SELECT` or `FROM` clause, in which case the subquery is evaluated to create a temporary table. Subqueries can also be used in conjunction with the `IN`, `NOT IN`, `ANY`, `ALL` or `EXISTS` operator to specify the conditions that the subquery must satisfy.

```sql
SELECT *
FROM table_name1
WHERE column_name1 > (SELECT MAX(column_name2) FROM table_name2)
```

Query 2.10: A SELECT query with a subquery.

2.3.4 Increasing SQL performance using database indexes

Indexes in databases are used to improve the performance of SQL queries. An index is a copy of the data from selected columns from a certain table that is intended to allow for extremely efficient search. The index can be used to rapidly look up the value of a table column. SQL indexes are important because they can significantly improve SQL query performance. In the absence of indexes, the SQL server would have to scan the entire table to find the desired data. This can be very time-consuming, especially for large tables with many rows and columns that are frequently accessed. The SQL server can quickly find the desired data with indexes instead of having to scan the entire table. Most databases will generate indexes for primary and foreign keys automatically. It is, however, the responsibility of the database administrator to create indexes for other columns that are frequently used in SQL queries. The indexes are not visible to the user. Another reason to use indexes is when you know you will order the result-set by a certain column. If the table is large, the `ORDER BY` operation will be very slow. In this case, it is helpful to create an index on this column.

2.3.5 Alternative to SQL: Object Relational Mapping

An Object Relational Mapper (ORM) is a tool that maps objects in an object-oriented programming language to relational database tables. The purpose of an ORM is to provide a layer of abstraction between the object-oriented programming language and the relational database. This layer of abstraction allows the object-oriented programming language to be used with a relational database. An ORM maps objects in an object-oriented programming language to relational database tables allowing us to use the object-oriented language to create, read, update, and delete data from the database.

How ORMs and SQL are related

ORMs use SQL in the background in order to perform database operations. This means that when an ORM is used, the SQL that is generated is usually not visible nor accessible to the user. However, it is possible to see the SQL that is generated by an ORM. To see the SQL query we can use the *logging* functionality that is provided by most ORMs. The logging functionality will print the SQL generated by the ORM to the console.

2.4 Regular Expressions

A regular expression, also known as regex, is a sequence of characters that provides a text search pattern. String-searching algorithms typically employ such patterns for "find" or "find and replace" operations on strings, as well as input validation. We will utilize these regular expressions in our anti-pattern detector. We will now go over the most basic concepts of regular expressions to aid our understanding. We will base this off of the book *Mastering Regular Expressions* [12].

Suppose we have a large piece of text and we want to determine if it contains the word "cat". We can simply use a regular expression for that. A regular expression is simply a string that starts with a ⌈ character and ends with a ⌋ character, sometimes containing special meta characters in between that allow us to make more complex searches than a straight text comparison would provide. Any string is a valid regular expression, so to determine if the word cat occurs in the text, we can simply search with regular expression listed in Code Listing 2.1.

```
⌈ cat ⌋
```

Code Listing 2.1: An example of a regular expression that finds the words matching "cat".

The first two special characters are the ⌈^⌋ (caret) and the ⌈$⌋ (dollar) characters representing the start and end of a line of texts respectively. Code Listing 2.2 shows a regular expression that matches if you have the beginning of the line, followed by a c, an a and t subsequently. A similar regular expression that matches if you have the word "cat" at the end of a line would be shown in Code Listing 2.3.

```
⌈ ^cat ⌋
```

Code Listing 2.2: An example of a regular expression that finds lines that start with "cat".

```
⌈ cat$ ⌋
```

Code Listing 2.3: An example of a regular expression that finds lines that end with "cat".

The meta character represented by the ⌈[]⌋ (open and close square brackets), allows you to match with any character at the position between them. For example, to match with any three letter word beginning with the letter c and ending with t, you could use the regular expression shown in Code Listing 2.4. The $[a - z]$ in this example matches with any lowercase letter from the English alphabet. The $[a - z]$ is equivalent to writing $[abcdefghijklmnopqrstuvwxyz]$. We can perform the same thing with capital letters by using $[A - Z]$, which matches any uppercase letter in the English alphabet. *cyt*, *cxt* and *cct* are all matches of this regular expression.

```
⌈ c[a-z]t ⌋
```

Code Listing 2.4: An example of a regular expression that finds words that start with a "c", end with a "t" and have a length of three letters.

The ⌈.⌋ (dot) is another meta character. This matches with any single character. For example, to match with any three letter word beginning with the letter c and ending with t, you could use the

regular expression shown in Code Listing 2.5. Like the previous regular expression, *cyt*, *cxt* and *cct* are all matches, but this time also *c2t*, *c7t* and *c8t* match since we are not restricted to alphabetical characters.

```
c.t
```

Code Listing 2.5: An example of a regular expression contains a dot meta character.

Another popular meta character is ⌈|⌋ (bar) which allows for the match of multiple different regular expressions. For example, to match with the words "cat" or "hat", you can use the regular expression shown in Code Listing 2.6. The parenthesis in this regular expression are another meta character and are used to group the two regular expressions together so that we can match with any of them.

```
(cat|hat)
```

Code Listing 2.6: An example of a regular expression that matches containing "cat" or "hat".

The ⌈+⌋ and ⌈*⌋ are two other meta characters used to repeat certain characters or character sets. The ⌈+⌋ meta character matches with the character that proceeds it, one or more times. The ⌈*⌋ meta character matches with the character that proceeds it, zero or more times. For example, to match with the string "aaabbcc" and "abbbb", we could use the regular expression shown in Code Listing 2.7.

```
a+b+c*
```

Code Listing 2.7: An example of a regular expression that finds strings with repeating characters.

Suppose we are searching a string that contains a generic whitespace, for example "a cat". We can find such a string using the regular expression shown in Code Listing 2.8. The ⌈\s⌋ meta character matches with any whitespace character including spaces and tabs. The ⌈\t⌋ meta character matches with only tabs.

```
a\scat
```

Code Listing 2.8: An example of a regular expression that finds strings containing "a cat".

2.5 A detailed look at SQL anti-patterns

In programming, design patterns are a best practice way of solving common problems in programming. They are often used to speed up development while also making code more maintainable. An example of a design pattern is the Singleton pattern, which is used to ensure that only one instance of a class is ever created. There are also undesirable patterns, which are best avoided. These so-called anti-patterns are ineffective ways of solving problems and can lead to more problems than they solve [16]. In this section, we will go through and discuss all of the known anti-patterns for SQL. We will first define these anti-patterns and then provide an easy-to-understand example in

which each anti-pattern is demonstrated. This will help the reader fully grasp what each anti-pattern is. We will also look at some of the most common solutions for preventing them. We encourage the reader to refer back to this section when necessary.

2.5.1 Example schema

We will demonstrate each anti-pattern by means of an example. This is done using a predefined schema that will be introduced in this section. The schema is a slightly adapted version as described in [45]. The example schema is given in Figure 2.5.

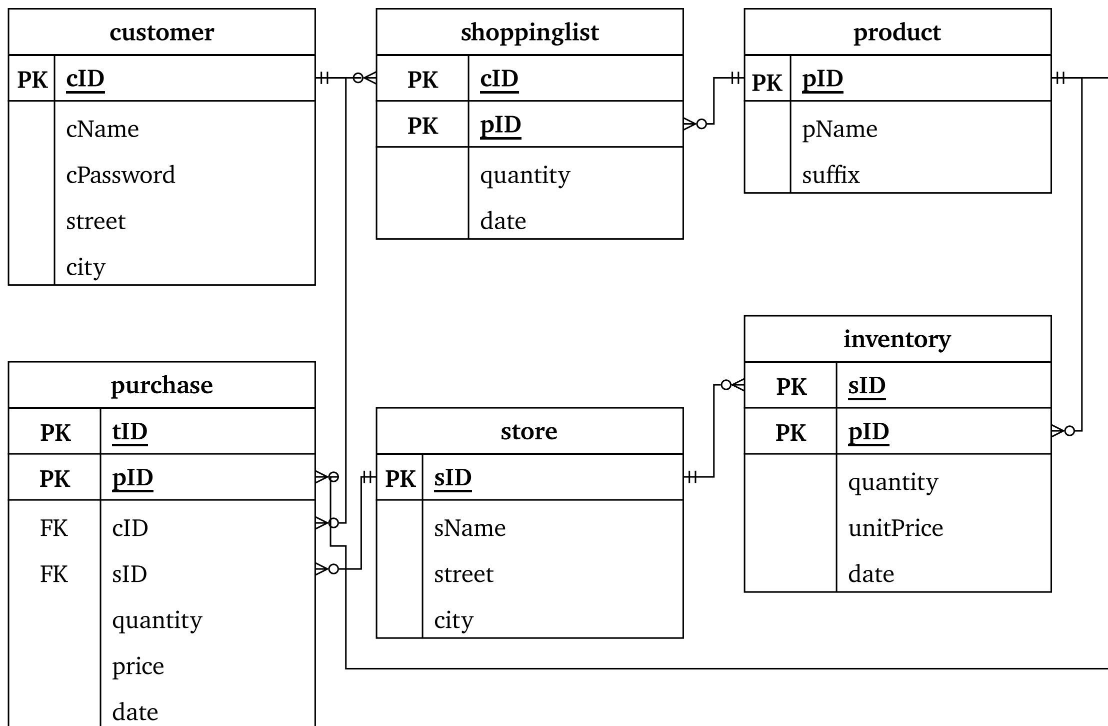


Figure 2.5: Relation diagram of our example schema.

2.5.2 A list of anti-patterns

This section will provide a summary of all known anti-patterns as described in [15] [17] [24]. For each anti-pattern we will provide a brief introduction that explains the anti-pattern, a code example using the schema described in Figure 2.5 and, whenever possible, a solution to fix the anti-pattern. It should be noted that the findings of the previously stated papers are covered in chapter 3 rather than this chapter.

Ambiguous Groups

This anti-pattern emerges when developers misuse the aggregation command GROUP BY. This may cause erroneous results. Every column in a query's SELECT statement must have a single value row per row group, which is also known as the *Single-Value Rule*. Now, for columns in the GROUP BY aggregation this is guaranteed, because it returns exactly one value per group, regardless of how many rows the group matches. For other SQL commands such as MAX(), MIN(), AVG(), it will also result in a single value for each group, so this is also guaranteed. The database server, on the other hand, cannot be so certain about any other field listed in the SELECT statement. It cannot always ensure that the identical value for the other columns appears on every row in a group.

Query 2.11 shows a basic example of this anti-pattern. In this example, because the *shopping-list* table identifies numerous products to a specific customer, there are several distinct values for product ID for a given customer ID. There is no way to express all product ID values in a grouping query that reduces to a single row per customer.

```
SELECT cID, cName, pID, MAX(date)
FROM customer JOIN shoppinglist USING (cID)
GROUP BY cID, cName;
```

Query 2.11: An example of the ambiguous groups anti-pattern.

The example above can easily be fixed. We must ensure that every column is guaranteed a single value. This can be done by including the product ID in the GROUP BY clause. Query 2.12 shows how this is done.

```
SELECT cID, cName, pID, MAX(date)
FROM customer JOIN shoppinglist USING (cID)
GROUP BY cID, cName, pID;
```

Query 2.12: A solution to Query 2.11.

Fear of the Unknown

In SQL, columns can be left empty. This results in an attribute of a certain row having a NULL value. SQL considers NULL to be a special value, distinct from zero, false, true, or an empty string. This is the opposite most developers know from other languages, especially making novices susceptible to this anti-pattern.

The code shown in Query 2.13a is querying the product name and suffix columns from the products table where the suffix is not equal to NULL. One might think that this will result in all rows that have a suffix, however, this is not the case. Any comparison to NULL returns *unknown*, not true or false. Therefore, this query does not return any data.

```
SELECT pName, suffix
FROM product
WHERE suffix <> NULL;
```

(a) An example of a NULL comparison.

```
SELECT UNIX_TIMESTAMP(date) + 86400
FROM shoppinglist;
```

(b) An example of a NULL addition.

Query 2.13: Two examples of the Fear of the Unknown anti-pattern.

There are many other examples of this anti-pattern. Another example is shown in Query 2.13b, where we convert the data to Unix time format and add the number 86400 to it. One would think that when a particular date is NULL, it would simply return 86400. However, this is not the case, since the date is NULL, thus the Unix time is NULL and therefore the addition 86400 is unknown.

A simple fix to Query 2.13a would be to use the IS NOT NULL operator. A fix for Query 2.13b, is to use a special function which takes in an expression and an alternative value and will return the value of the expression if the expression is not NULL or else, it will return the alternative value. Almost every dialect of SQL has such a function available, for example the ISNULL() function in SQL Server or the IFNULL() function in MySQL. In this case, we set the alternative value to 0, meaning that if the data is NULL, we start from 0. The solutions are displayed in Query 2.14a and Query 2.14b.

```
SELECT pName, suffix
FROM product
WHERE suffix IS NOT NULL;
```

(a) A solution to Query 2.13a.

```
SELECT UNIX_TIMESTAMP(ISNULL(date,
    0)) + 86400
FROM shoppinglist;
```

(b) A solution to Query 2.13b.

Query 2.14: Two possible solutions for the Fear of the Unknown anti-pattern.

Implicit Columns

When writing a query that needs a lot of columns, developers often opt to use the SQL wildcard selector (*). This means that every column from the table(s) specified in the FROM clause is returned, meaning that the list of columns is implicit instead of explicit. In many ways, this makes the query more concise. However, this can come at a cost as the result-set can be quite big for large tables. Making the query more specific by explicitly listing the columns can be counter-intuitive, but is often beneficial, for instance to save space and network resources.

Suppose we have the task of finding all customer IDs, store IDs, and product IDs as well as the quantity and price from the purchases table. This would mean that the only columns from the purchases table that are not present in this query would be the purchase ID and data columns. The query shown in Query 2.15 contains the implicit columns anti-pattern as it uses the wildcard (*). Instead of selecting only the columns requested by the task, the developer utilizes a wildcard, returning in a result that includes both the store ID and the date columns.

```sql
SELECT *
FROM purchase;
```

Query 2.15: An example of the Implicit Columns anti-pattern.

```sql
INSERT INTO purchase
VALUES (DEFAULT, 2, 6, 1, 3, 4.99, '2022-2-16 14:44:53');
```

Query 2.16: Another example of the Implicit Columns anti-pattern using an INSERT.

Query 2.16 shows that this anti-pattern does not only occur in SELECT queries, but could also appear in INSERT queries. Instead of providing a subset of columns following the table name, the query applies the values to all columns in the order they are specified in the table. Many developers will prefer this way of writing the query since it is much shorter.

Implicit Rows

This anti-pattern occurs when a query reads more rows than is used by the application. This often happens when the application applies a filter after querying the data.

In Query 2.17, the task was to select all product names and unit prices where the unit price was larger than 2.99. However, the application that utilizes the data returned from this query also applies a filter of its own that calculates the average unit price and discards all the products that are below it.

```sql
SELECT pName, unitPrice
FROM products JOIN inventory USING (PID)
WHERE unitPrice > 2.99
```

Query 2.17: An example of the Implicit Rows anti-pattern.

Instead, it is more efficient to write a single query that applies both the filter from Query 2.17 as well as the internal filter from the application. Query 2.18 shows a possible solution where a subquery is used to calculate the average unit price of unit prices above the threshold of 2.99.

```sql
SELECT pName, unitPrice
FROM product JOIN inventory USING (pID)
WHERE unitPrice > (
  SELECT AVG(unitPrice)
  FROM inventory
  WHERE unitPrice > 2.99
)
```

Query 2.18: A solution to Query 2.17.

Loop to Join

Instead of combining two tables and querying from the joint table, the loop to join anti-pattern arises when we first query from one table and then loop over the data it returns to retrieve values from another table.

The code in Query 2.19 shows this anti-pattern and is written in Python. Note that this anti-pattern can apply to any programming language, not just Python. Assume that we have a function `execute_query()` that takes in a raw SQL query string and returns a list. The first query returns all the product IDs that have a unit price higher than 2.99 and stores it in a variable named `products_ids`. Next, the code creates an empty list named `customer_ids`. This list is filled using a loop structure that loops over all the product IDs that we obtain using the first SQL query. For each product ID, we run a separate query that returns the customer ID of customers that bought the respective product.

```
product_ids = execute_sql(
    """
    SELECT pID
    FROM product
        JOIN inventory USING (pID)
    WHERE unitPrice > 2.99
    """
)

customer_ids = []

for id in product_ids:
    customer_ids.append(
        execute_sql(
            f"""
            SELECT DISTINCT cID
            FROM purchase
            WHERE pID = {id}
            """
        )
    )
```

Query 2.19: Python code that exhibits the Loop to Join anti-pattern.

Instead, the above Python code could be replaced with a single SQL query that uses the `IN` operator, which is the SQL syntax for matching against a set of values. We create this set of product ID values using a subquery.

```
SELECT DISTINCT cID
FROM purchase
WHERE pID IN (
    SELECT pID
    FROM product
      JOIN Inventory USING (pID)
    WHERE unitPrice > 2.99
  )
```

Query 2.20: A solution to Query 2.19.

Not Merging Projection Predicates

Not Merging Project Predicates is an anti-pattern that occurs when we issue multiple queries each of which reading a subset of the columns that are required for a certain task.

```
/* Query 1 */          /* Query 2 */
SELECT cName           SELECT city
FROM customer          FROM customer
```

Query 2.21: An example of the Not Merging Projection Predicates anti-pattern.

Suppose the task was to retrieve all customer names and cities. In Query 2.21, we see two queries both querying a column from the customers table. Instead, these two queries can easily be merged into one, which is shown in Query 2.22.

```
SELECT cName, city
FROM customer
```

Query 2.22: A solution to Query 2.21.

Not Merging Selection Predicates

Much like the previous anti-pattern, the Not Merging Selection Predicates is an anti-pattern that occurs when we issue multiple queries each of which reading a subset of rows that are required for a certain task.

```
/* Query 1 */              /* Query 2 */
SELECT cName, city         SELECT cName, city
FROM customer              FROM customer
WHERE city LIKE "a%"       WHERE street LIKE "%lane%"
```

Query 2.23: An example of the Not Merging Selection Predicates anti-pattern.

Suppose the task is to retrieve all customer names and cities for customers who live in a city that begins with the letter A or on a street that includes the word ”lane”. Query 2.23 shows a query solving this task that has the Not Merging Selection Predicates anti-pattern. Query 1 and 2 both return a subset of the complete data. To obtain the whole set required for the task, we must still take the intersection. A better solution would be displayed in Query 2.24, which merges the selection predicates using the `OR` operator.

```
SELECT cName, city
FROM customer
WHERE city LIKE "a%" OR street LIKE "%lane%"
```

Query 2.24: An solution to Query 2.23.

Not Using Parameterized Query

One of the advantages of SQL is the ability to write a query and utilize parameters to act on the result-set dynamically. A so-called parameterized query (sometimes also called prepared statement) is a method of pre-compiling a raw SQL statement so that only parameters must be entered in order for the query to be performed. These parameters are bound at runtime and can be compared to variables from a function. By parameterizing queries, the DBMS can reuse the query execution plan, making the execution of the query more efficient.

The primary reason for using parameterized queries is to avoid SQL injection attacks, which are queries that are vulnerable to data leakage and corruption. An example of a query vulnerable to SQL injection is also shown in this section, namely within the Vulnerable Query anti-pattern section. The solution used to address this vulnerability is a parameterized queries as shown in Query 2.32.

Not Caching

When multiple syntactically comparable or substantially equivalent queries to retrieve data from the database are issued and all have a common sub-expression, these searches should be cached.

Query 2.25 shows an example of this anti-pattern. One might notice that both queries 1 and 2 issue the same subquery, which returns all product IDs with a unit price higher than 5. Suppose the inventory table had a billion rows, which would make the average computation rather slow especially since this computation is required for both queries. A cache could be used to store the results of these large computations. Then, a query could obtain the results from cache, without having to recompute. Usually, caching queries is done by either the ORM, database, or DBMS, making this anti-pattern less vital to novices.

```sql
/* Query 1 */
SELECT sID
FROM products
  JOIN Inventory USING (pID)
GROUP BY pID
HAVING pID IN (
    SELECT pID
    FROM Inventory
    GROUP BY pID
    HAVING AVG(unitPrice) > 5
  )

/* Query 2 */
SELECT pID, MAX(quantity)
FROM product
  JOIN inventory USING (pID)
GROUP BY pID
HAVING pID IN (
    SELECT pID
    FROM Inventory
    GROUP BY pID
    HAVING AVG(unitPrice) > 5
  )
```

Query 2.25: An example of the Not Caching anti-pattern.

Poor Man's Search Engine

Suppose we want to search for words or sentences in our database. The first thing that comes to mind is using a SQL pattern-matching predicate, such as the LIKE keyword, to which we can specify a pattern. The LIKE predicate can be used with a wildcard (%) that matches zero or more characters. When used before and after a keyword, it matches any string that contains that word. SQL also supports regular expressions to match strings that start with a certain pattern, making both methods seem like a very good option for full searches.

However, one must know that the main problem of pattern-matching predicates is their poor performance. Because they cannot use a traditional index, as described in subsection 2.3.4, they must scan every row of the specified tables. The overall cost of a table scan for this search is very high, since matching a pattern against a column of strings is a costly operation when we compare it to other comparison methods like integer equality.

Another problem with simple pattern-matching using the keyword LIKE or regular expressions is that they can find unintended matches, making the search result not accurate or erroneous.

Query 2.26 shows an example of how to search products that have the word "cat" in their product name. These queries should be avoided if the table size of products is large. Instead, one should opt to use a specialized search engine method some of which even come as standard with certain databases or DBMSs.

```
SELECT *
FROM product
WHERE pName LIKE "%cat%"
```

Query 2.26: A query showing pattern matching.

Random Selection

Random selection is another common anti-pattern that frequently occurs when we want to select a random row from a certain table. This is often done as displayed in Query 2.27, where we use the SQL RAND() function to sort the data randomly.

```
SELECT cID
FROM customer
ORDER BY RAND() LIMIT 1
```

Query 2.27: A query showing the random selection anti-pattern.

However, the performance of this query is rather poor. By using the RAND() inside an ORDER BY clause, the use of an index is not possible, since there is no index containing the values returned by the random function. This is a big concern for the query's performance because using an index is one of the best ways to increase the computation of sorting. As a result of not employing an index, the query result-set must be sorted by the database using a slow table scan, making the performance poor.

Developers should instead choose a random value using other means. Common ways would be to generate a random value between 1 and the greatest primary key, or counting the total number of rows and generating a random number between 0 and the row count. Then we can use the random number inside a WHERE clause.

Readable Password

This anti-pattern is less related to SQL, and more related to application and database development. It is not secure to store passwords and other sensitive information in a plain text field in the database. This anti-pattern is still directly related to SQL, since an attacker can intercept and read SQL statements revealing plain text passwords as shown in Query 2.28. Furthermore, query logs may contain these queries which can be leaked in the event of a data breach.

```
customer_id, password = 123, "secretPassword"

data = execute_sql(
    f"""
    SELECT *
    FROM customer
    WHERE cID = {customer_id} AND cPassword = {password}
    """
)
```

Query 2.28: Python code showing the Readable Password anti-pattern.

Instead, developers should store sensitive information such as passwords using a (strong) cryptographic hash function. These functions offer a one-way encryption, meaning once something is hashed there is no straightforward way of reversing the hash. The only way to determine the value of a hash would be to perform a brute-force search by generating a random word, hashing the word, and checking whether it matches the hash, which is a very slow and cumbersome method. The hash function converts the input string into an unrecognizable string known as the hash. Whenever we need to compare a certain value with a hashed field, we can hash that value in the application code and use it in our query. This is shown in Query 2.29.

```
# Hash password such that it becomes unrecognizable
customer_id, password = 123, hash("secretPassword")

data = execute_sql(
    f"""
    SELECT *
    FROM customer
    WHERE cID = {customer_id} AND cPassword = {password}
    """
)
```

Query 2.29: A solution to Query 2.28.

Spaghetti Query

Queries can be of varying degrees of difficulty. Some are straightforward, such as basic select queries that choose a few columns from a table. Others are significantly more sophisticated, such as a complex join between two databases or a recursive subquery, which runs the subquery on the results recursively. Sometimes, during development of a query for a complex task, the query becomes too complex that the programmer gets stuck. This is most likely because programmers are fixated on solving the task both elegantly and efficiently, thus they try to complete it with a single query. However, the complexity of these single queries can increase exponentially, making both maintainability and correctness more difficult to achieve.

```
SELECT COUNT(p.pID) as numberOfDistinctProducts,
    SUM(i.quantity) as numberOfProducts,
    AVG(i.unitPrice) as averagePrice,
    city
FROM product p
    JOIN inventory i ON (p.pID = i.pID)
    JOIN store s ON (i.sID = s.sID)
GROUP BY s.city
```

Query 2.30: A complex query showing the Spaghetti Query anti-pattern.

The query shown in Query 2.30 can be considered overly complex for what it does, but it demonstrates the type of problem that can occur when a programmer tries to solve a complicated problem in one query. SQL is a sophisticated language that allows you to do a great deal with a single query or statement. However, this does not mean that it is essential to try to solve every problem with a single query or line of code.

A simple way to tackle complex queries is to use the divide and conquer method, where you divide the problem into multiple parts so you can solve them independently. In other words, if you break up a long complex query into several simpler queries, you can then focus on each part individually and do a better job of each of them, since they are less complex. You can then simply run all these parts as a single statement, which can help both in debugging and testing as well as reducing query complexity. While it is not always possible to split a query this way, it is a good general strategy, which is often all that is necessary.

Unbatched Writes

This anti-pattern occurs when a sequence of database writes are issued separately instead of being batched together into a single transaction. If the database repeatedly triggers transactions, the transaction processing overhead of each write request might result in substantial inefficiencies.

Luckily, this anti-pattern does not occur at the query level, but at ORM implementation levels. As described in subsection 2.3.5, an ORM maps standard APIs to SQL statements. This anti-pattern can be fixed by analyzing the dependencies between multiple SQL statements, as developed by Tamayo et al. [43]. If there is no such dependencies between queries, then it is safe to batch these queries together in a single database write.

Unbounded Query

This anti-pattern occurs when a query is executed that may return an unbounded number of records and there is a subsequent computation or action performed on the returned records. This may lead to a couple of problems. The first problem is that it may lead to unresponsive applications. This might happen if the application renders the returned records. Another problem is that malicious users may carry out a denial-of-service attack by tainting the database with a large number of records. If the unbounded query is then issued, CPU exhaustion may happen.

Vulnerable Query

Vulnerable queries are queries containing inputs that are not properly sanitized. An example would be a user input from which the value entered is directly used inside a query. This query would then be vulnerable. A vulnerable query could be used to perform injection or to execute external commands on the database server. This is known as SQL injection.

```
// Retrieve url param e.g. ?id=123
const urlParams = new URLSearchParams(window.location.search);

// Get value of id
const cID = urlParams.get("id")

// Use cID in query
const data = `SELECT * FROM customers WHERE cID = ${cID}`;
```

Query 2.31: JavaScript code showing the Vulnerable Query anti-pattern.

In Query 2.31 we see a snippet of JavaScript client code that queries a customer based on a GET parameter from the current URL. This query is prone to SQL injection. SQL injection is a common attack vector. The attacker's goal is to exploit a weakness in a web application's input validation by using a SQL injection attack. An attacker can gain unauthorized access to data in a database by using this technique. The attacker will create an input that is intended to influence the output of a SQL query. This may cause data leakage or data corruption. For example, if the attacker changes the GET parameter as follows `?id="105OR1=1"`, then the page would display the whole customer database since the query would be parsed as `SELECT * FROM customers WHERE CID = 105 OR 1=1`, making the `WHERE` clause always be true. A parameterized query, as shown in section 2.5.2, could be used to address this vulnerability.

Query 2.32 shows a parameterized query `selectCustomer` that takes as parameter an integer customer ID. Now we can call this procedure using the query `EXEC SelectAllCustomers @CID= 105;`. Note that if an attacker changes the GET parameter string, running the query would result in an error, since the type of `CID` is not an integer.

```
CREATE PROCEDURE selectCustomer @cID int
AS
SELECT * FROM customers WHERE cID = @cID
GO;
```

Query 2.32: A solution to Query 2.31.

2.6 Summary

We began this chapter by discussing some fundamental database concepts, such as table structure and table relations. Following that, we looked at database languages, with a focus on SQL. We discussed SQL's history as well as some of its key language elements. We also looked at how to write SQL syntax for common tasks like filtering, ordering, and table joining. Then, as an alternative to SQL, we introduced ORMs and looked at how they are related to SQL. Finally, we discussed regular expressions in general and how they can be used. For the remainder of the chapter, we examined various anti-patterns presented in various works of literature.

Chapter 3

Related work

In this chapter, we will discuss several topics related to this thesis. We will start by discussing the various research findings regarding the barriers to SQL learning, then talk about the impact and prevalence of SQL anti-patterns, and finally discuss some tools for the detection of these anti-patterns. These topics are relevant to this thesis since they cover various aspects of the topic being researched.

3.1 SQL learning barriers

Various barriers to learning SQL have been discovered in previous research [35, 28, 23, 2]. The common finding of all these studies mentioned is that subjects have difficulties in using query languages. These languages are hard to use, overly verbose, and complicated in general. For SQL in particular, it is shown that users need extensive training and an explicit representation of the database in order to be effective in a query language. Learning to leverage the full potential of SQL may be a challenging task due to its seemingly simple but expressive syntax and the high complexity that results from this expressiveness.

These barriers, combined with incomplete knowledge, often lead to SQL users writing erroneous queries. There are two main types of errors, namely syntactic and semantic errors. The first type can be characterized as simple mistakes that involve syntax errors such as missing parentheses or misspelled keywords, while the second type involves errors in the meaning of a query. These so-called semantic query errors are queries that are syntactically correct but do not produce the intended results for the given task.

There have been numerous previous publications that investigate these two types of errors. These works focus on identifying frequent errors, misconceptions, and their root causes. In 1992, a study by Smelcer revealed that `JOIN` clause omission was a common and troublesome error [39]. This type of error is a semantic error since the query was syntactically correct, but the result-set of the query is wrong. It was shown that the error occurred because either the creator of the query never learned about `JOIN`s, there was no explicit clue to use a `JOIN` in the problem statement, working memory overload made the users forget to include a `JOIN` clause, or users inappropriately used the procedure appropriate for a single table query which is incorrect.

In 2004, Stefan Brass and Christian Goldberg published their paper on semantic errors in SQL queries [6]. In this paper, they give an extensive list of conditions that are strong indications of semantic errors [6, p. 1]. It can be further categorized into two categories. The first category consists of semantic errors where the task must be known in order to detect it is erroneous. The second category consists of semantic errors that could be detected without knowing the task at hand. This paper focuses on the latter variant. The paper then gives a list of over 30 semantic errors collected from exam data.

Ahadi et al. presented a work that looks at syntactic mistakes made by students in writing

different types of SQL queries [4]. Their top, most frequent errors were syntax errors, undefined column errors, and grouping errors. They state that while syntax errors seem to be a lack of practice by the student, the other two errors give us greater insights. They argue that, in some cases, the undefined column error is caused by a typo, in other cases, the complexity of the presented entity-relationship diagram may cause students to choose the incorrect field name. They also suggest that the teaching of SQL needs to place greater emphasis on syntax and syntax errors since they found that syntactic errors are what lead students to abandon attempting a question. Research also shows that students mismanage complexity in SQL by persisting through syntactic and semantic errors, resulting in unnecessarily complex queries [21]. It shows that students stick with their initial attempts when solving query formulation problems and hardly ever start over. The paper urges teachers to integrate SQL problem decomposition in their lectures.

In 2018, Taipalus et al. presented a DBMS independent categorization of SQL errors made by students [42]. The study confirmed prior research findings, shown in [4, 39], and identifies new categories of errors, notably logical errors that repeat in similar ways among different students.

The paper *Explaining Causes Behind SQL Query Formulation Errors*, also written by Taipalus, and published in 2020, presents the most prominent query formulation errors and maps these to cognitive explanations [41]. The paper uses the four root causes for query formulation errors as described by Smelcer [39], namely *working memory overload, absence of retrieval cue, procedural fixedness* and finally *absence of procedural knowledge*. The data used in the paper is collected over a period of four years from four student cohorts. The course used to obtain the data was an introductory database course for second-year university students. The students had no pre-knowledge of SQL. This yielded a total of 12,180 queries after filtering, out of which 3,739 were incorrect. *Missing expression* and *extraneous or omitted grouping column* are the most common type of error, with a combined frequency of over 50%, confirming the findings of [35]. Next, the paper goes over the types of errors and tries to find the causes behind these errors. This is done in a speculative way in the sense that the author does not claim that they identify the causes of the errors, but he presents various different possible cognitive explanations. The author comes to the conclusion that the cause of error does not rest solely on the type of error committed, but also on the query concept [41, p. 8]. Taipalus states that educators should teach students to recognize patterns in the natural language data demand, and map these patterns to corresponding SQL constructs [41, p. 7].

A study by Miedema et al. on *Identifying SQL Misconceptions of Novices* was published in late 2021, with the intent of investigating misconceptions that might be the causes of documented SQL errors made by SQL novices [20]. While the previous paper by Taipalus tried to find the root causes of general SQL errors, it did not utilize qualitative input from the students. This paper has another research method. It uses qualitative, ”think aloud” interviews to not only identify common errors and misconceptions, but also take the students’ perspectives into account. This is then used to see where the mismatch between question and query formulation occurs [20, p. 2]. The authors held online interviews, each interview taking roughly thirty minutes. The sample size was 21 participants, with each participant coming from one category based on education level. The first category were high school students, and the second being university students who had taken one database course before the interview. This shows that the participants can be considered novices. The questions asked in the study were specifically designed to capture all basic concepts of SQL that novices would be familiar with. After the interviews, the data was analyzed and the errors were extracted. The authors then give an overview and explanations of the various errors that were identified. These errors were grouped into error categories. The authors found that SQL misconceptions fall into one of four possible categories: *misconceptions based on previous course knowledge, generalization-based misconceptions, language-based misconceptions* and *misconceptions due to an incomplete or incorrect mental model*. Previous works in computer science had shown that the transfer of students’ pre-existing knowledge from plain language, mathematics, and other programming languages is often the source of programming misunderstandings [34]. The authors can confirm that this is likewise the case with SQL based on their findings. The authors also acknowledge some limitations in their work, the first limitation being that the discovered errors and

misconceptions could be the result of the instructional methodologies and teaching styles used. Secondly, the usage of paper and pencil during the interviews may have facilitated some syntax errors, some of which might not have emerged while working on a computer. Lastly, the authors' error encoding and analysis follows an existing error taxonomy, and hence inherited any limitations of this past work. However, it is still an insightful work that shows that misconceptions can be considered a source of errors.

The works above present common mistakes, errors, and misconceptions when writing SQL queries. A majority of the discussed works have obtained their results by analyzing students or student projects and tests. However, in the real world queries are typically deemed more complex compared to student project queries. We argue that the number of mistakes, errors, and misconceptions that are commonly written when writing real-world queries is considerably higher making it an even bigger issue.

3.2 Literature on anti-patterns

Research shows us that SQL errors can lead to anti-patterns (code smells) that harm software quality [4, 17, 24]. Anti-patterns also influence education, because students may use anti-patterns without realizing it. The result-set of a query containing an anti-pattern is often correct, or appears to be correct. However, anti-patterns are harmful because they make the resulting program or statement inefficient, harder to maintain, and easier to misread. Anti-patterns for programming languages were first mentioned by Koenig in 1995 [16]. For databases in general, and SQL specifically, anti-patterns were defined by Karwin in 2010 [15]. The book written by Karwin and published in 2010, formed the basis of literature on SQL anti-patterns. The book is written for developers who need to use SQL in an efficient way. In part three of the book, Karwin defines a list of query anti-patterns along with some real case scenarios, legitimate uses in which one could argue it is a valid use case, and a solution.

In 2019, Lyu et al. quantified the performance impact of SQL anti-patterns on mobile applications [17]. This research consists of a literature review and a benchmark study to investigate problematic programming practices (anti-patterns) with respect to database usage. For their research, they utilized existing Android apps that make use of a local SQLite database. They collected energy and runtime measurement results from a single device, a Samsung Galaxy S5 running Android 5.0. Throughout their literature review, they identified eleven SQL anti-patterns. In the next phase of their research, whilst performing their experiment, they discovered that eight of these anti-patterns have a significant impact on the runtime and energy consumption of local database operations. Out of these eight, two of them particularly stood out, that even end-users could notice the impact. The anti-pattern they call *unbatched writes* (2.5.2), where a sequence of database writes are issued separately instead of being merged (batched) together into a single transaction, had such a big runtime impact that once fixed, the average runtime reduction exceeded 100 ms, which is a common threshold for a human to perceive delay. The other anti-pattern labeled *loop to join* (2.5.2) occurs when developers use a query to get multiple values from one table and then for each value make a query to another table (using a loop structure). This code smell could be fixed using a `JOIN` that joins the two tables first and would then query from the joined table. This would on average also decrease the runtime by more than 100 ms.

Muse et al. focused their research on prevalence, impact and evolution of SQL anti-patterns in data-intensive systems [24]. They did an empirical study in which they collect 150 open-source projects and examined if there was any occurrence from a list of both traditional and SQL anti-patterns in these projects. During examination, they looked at the prevalence of the anti-patterns from the list, see if there were any co-occurrences with traditional anti-patterns and bugs, and looked at the evolution of each anti-pattern. They considered the book by Karwin as an important work and used a detection tool that was able to find the anti-patterns listed by Karwin [15]. The study focuses on four anti-patterns, namely *ambiguous groups, fear of the unknown, implicit columns* and *random selection*. When the tool is run against the 150 open source projects, it is clear that SQL

anti-patterns are indeed commonly found in data-intensive systems. The *implicit columns* and *fear of the unknown* anti-patterns are prevalent, whilst the other two are not. By further analyzing the 150 projects and categorizing each project as either *business, library, multimedia* or *utility*, the authors found that while the median prevalence for each category is zero, there still exists a significant number of outliers in the library, business and utility categories. This is concerning, since libraries and utilities are used by other developers in other projects as dependencies, making these anti-patterns also prevalent in these dependent projects. The Apriori association algorithm is used to see if there is any associations between traditional and SQL anti-patterns. Analysis shows that while there exists a weak co-occurrence with these traditional smells, the degree of association, measured with a test called Cramer's V Test for Association, is low meaning that they are not strongly related. The same algorithm is used to determine if SQL anti-patterns co-occur with bugs. It is shown that there is no statistical significant association. Lastly, the authors performed survival analysis to determine how long SQL anti-patterns survive in a project. For every project, after 500 commits, a snapshot is created from the most recent commits backwards. This means that the author can look at the evolution of a project. They show that SQL anti-patterns have a higher tendency to survive for a longer period of time compared to traditional anti-patterns [24, p. 9]. Around 80% of anti-patterns persist throughout all available snapshots and hardly get any attention whilst code refactoring by developers. The essence of this paper is that SQL anti-patterns are prevalent among the small project samples the researchers have studied. All though this may not necessarily reflect the true prevalence of SQL anti-patterns, it is evident that more attention is needed for this issue in order to prevent SQL anti-patterns from persisting and potentially affecting projects.

3.3 Literature on anti-pattern detectors

The papers above state that SQL anti-patterns can have an impact on performance and are prevalent in various code bases. To get rid of existing anti-patterns, one must find them in the first place. This is done through detection. There are also various papers on the detection of SQL anti-patterns. One of which is written by Nagy and Cleve [25]. In this paper, a tool is presented for the identification of anti-patterns in SQL queries found in Java source code. The tool implements a static analysis on both the embedded SQL queries as well as the database schema and the data in the database. Out of the six anti-patterns listed in the book by Karwin [15], four of them are implemented by this tool.

Another paper that presents detection methods for various different anti-patterns is written by Lyu et al. [18]. This paper is presented as a follow up paper to [17] which is also written by Lyu et al. Instead of being a tool like the previous paper, the authors opted for a more theoretical approach. SAND can be considered more of an extensible framework for the analysis of anti-patterns using a novel set of abstractions. Missed or future anti-patterns can be added in the future, if the vocabulary of abstractions sufficient. The prototype tool that implements SAND is quick, efficient and, most importantly, very accurate.

The paper written by Arzamasova, Schäler and Böhm [5] was published in 2018. The goal of the paper is to extract and analyze patterns from a query log of a database with a focus on anti-patterns. A common method for detecting anti-patterns, as we have seen in two papers prior in [25], requires access to the software that generates requests. The authors argue that one would then need access to all systems working with the database which is practically impossible. Instead, the paper looks at past queries that are stored inside the query log. It is important to analyze query logs, since knowing how a big database is used is very important to the stakeholders. Finding patterns in this log could help with that. However, anti-patterns might falsify such analyses. Therefore, we would like to detect these from the log and, if possible, fix them. The paper first defines the notion of a pattern and uses the notion of a skeleton tree in this definition. A skeleton tree is obtained from a syntax tree, by replacing all variables in its leaf nodes with placeholders. A query template is then a triple, containing skeleton subtrees and finally, a pattern is defined as a sequence of query templates. The authors then use two anti-patterns as examples and create for each anti-pattern its query template. They state that this framework can be extended to accommodate future anti-patterns.

Now to extract these anti-patterns from the query log, the authors first delete duplicates and remove syntactically incorrect queries by means of a syntax tree. Next, they extract the `SELECT, FROM` and `WHERE` subtrees and their skeleton forms. Now, they can use the formal definitions of the anti-patterns to detect them from the query log.

Another way of detecting anti-patterns in SQL queries is through text classification techniques [30]. A study by Ousmane and Xie demonstrates how machine learning approaches can be leveraged to find anti-patterns in SQL queries by approaching the problem as a text classification problem. First, a large dataset of SQL queries from the SkyServer catalog is created. From these queries, all schema-related terms are removed. The authors then match each query to an anti-pattern, or no anti-pattern when the query does not contain any. They then use word2vec [22], a group of models used to produce word embeddings, to encode the SQL queries. Finally, a convolutional neural network is trained to classify anti-patterns from SQL queries. They state that their model is quite accurate, with an accuracy of 83.2%, and that it can outperform other software.

Lastly, we will discuss a tool named SQLCheck [11], a toolchain for automatically detecting and correcting anti-patterns in database applications. The focus of their technique is to combine code analysis with data analysis. This means that the tool combines the analysis of raw SQL queries, with the underlying data and database models. This means that, in addition to query anti-patterns, this tool may check for logical, physical, and data anti-patterns. The tool begins by extracting context from queries using a query analyzer. Following that, it employs a data analyzer to extract context from the database's tables. The tool then employs a set of rules to identify APs in the given queries based on the application context. These rules are general-purpose functions that take advantage of the application's overall context. The detected anti-patterns will then be reported. They further evaluate their tool and show that their tool has high precision and recall.

One thing these papers on anti-pattern detection have in common is that they are all tools a user needs to have installed in order for them to use them. Research shows that developers often utilize websites such as Stack Overflow as informal crowd-based anti-pattern detectors [40]. The authors hypothesize that because crowd-sourced detection considers contextual factors, it is more trusted by developers than automated detection tools. However, we think this also has to do with the fact that the majority of existing anti-patterns detectors are not accessible and easy to use. Existing tools focus on detecting anti-patterns, without thinking of the user experience.

3.4 Summary

The above-mentioned papers show that there has already been some significant research on the topic of SQL errors, misconceptions, and anti-patterns. Various barriers to learning SQL have been discovered in previous research [35, 28, 23, 2]. These barriers, combined with incomplete knowledge, often lead to SQL users writing erroneous queries [20, 4, 41, 32]. Research shows us that this can lead to anti-patterns (code smells) that harm software quality [17, 24] and the education of SQL. For databases in general, and SQL specifically, anti-patterns were defined by Karwin in 2010 [15]. Early detection of anti-patterns can help developers avoid technical debt, as writing 'smelly' SQL may result in poor performance or erroneous results as shown by [17, 24]. This is why much of the research on SQL anti-patterns has focused on detecting them [25, 18, 5, 30, 11].

Chapter 4

Developing an anti-pattern detector

In this chapter, we will discuss the development of our anti-pattern detector. This chapter, along with chapter 5, will also answer the following research question:

> RQ1 How can a tool be designed to help novice users understand anti-patterns?

We will answer this question by first answering the sub-question (RQ1.1). This can be accomplished defining the anti-pattern scope, where we specify which anti-patterns can be detected in raw queries. Next, we will go over how we can detect those anti-patterns. Finally, we describe how we distributed the detector so that novices could easily use it.

4.1 Defining anti-pattern scope

Out of the list of anti-patterns described in subsection 2.5.2, only a handful of anti-patterns could be detected in raw SQL queries only. In this section, we will give justifications as to why some anti-patterns could not be detected in raw SQL queries. We answer the following research question (RQ.1.1):

> RQ1.1 What SQL anti-pattern(s) can occur in raw queries written by novices?

If we look carefully at the list of anti-patterns, you may notice that some anti-patterns occur when one mixes SQL queries with application logic, meaning that we will not be able to detect these anti-patterns when we are only given a raw query. Following the same order as the list of anti-patterns, the first anti-pattern that uses another language next to SQL is the *Implicit Rows* anti-pattern, which uses application logic to further filter the queried data. Next is the *Loop to Join* anti-pattern, where we use application logic to query data from one table, and then use a loop structure to query another table using the values of the first query. This anti-pattern is followed by the *Readable Password* anti-pattern and the *Vulnerable Query* anti-pattern both using application logic to insert application data in a query. The last anti-pattern that uses application logic is the *Unbounded Query*. This anti-pattern occurs when the result set of a query can return an unbounded number of records and the application that uses the query performs a subsequent computation over that result set. Since our detection tool must be able to analyze raw SQL queries, and there is no way for us to detect these anti-patterns in a raw query without providing application logic, we consider these anti-patterns outside our scope.

Another group of anti-patterns that are not detectable from a single raw SQL query is the group that is only detectable after obtaining multiple queries. This is the case for the *Not Merging Projec-*

tion Predicates, the *Not Merging Selection Predicates* and the *Unbatched Rewrites* anti-patterns. The first two anti-patterns can only be detected if we have seen at least two queries, that either both read a subset of columns, a subset of rows, or both. The latter anti-pattern occurs when we have multiple `INSERT` queries that all write separately to a database, instead of being batched together.

The last group of anti-patterns that cannot be detected by a single raw SQL query is the group containing the *Not Using Parameterized Query* and *Not Caching* anti-patterns. These two anti-patterns occur when certain optimizations are not applied (parameterization and caching of queries). From raw SQL queries, we cannot observe these optimizations, hence they are undetectable.

Now that we have determined which of the anti-patterns are not detectable in raw SQL queries, we can create a list of anti-patterns that can be detected. With this list, we have answered research question (RQ1.1). The following anti-patterns are left and can be implemented in our static raw query detection tool:

- Ambiguous Groups
- Fear of the Unknown
- Implicit Columns
- Poor Man's Search Engine
- Random Selection
- Spaghetti Query

Some of these anti-patterns, like the *Implicit Columns* anti-pattern may occur in statements that do not start with `SELECT`, such as `INSERT` queries. Since our tool will be targeted toward novice users, who will be mostly writing `SELECT` queries, we chose to focus on detecting these anti-patterns in `SELECT` queries only. In the next section, we will go over each of these anti-patterns and discuss our method for detecting them.

4.2 Detection of various anti-patterns

In this section, we will go over the implemented anti-patterns and the way we are detecting them. Note that the code examples provided do not necessarily reflect the implementation in our detection tool. We have chosen a different order than the list described in section 4.1, starting with the easiest anti-pattern to detect and building up from there to anti-patterns that require a more sophisticated detection method. Our programming language of choice for developing a detection tool is Python, since Python code is readable and easy to interpret. It is also one of the most popular programming languages and has numerous amazing libraries that speed up development. On such library being `sqlparse`, which is a non-validating SQL parser for Python [3]. Another reason that we chose Python over other languages is the simplicity of distributing Python code via a custom package. This enables us to distribute the detection tool as a standalone command-line application that one could install using a single command. We will talk more about the distribution of our tool in section 4.3

Python also has a module, called `re`, that provides regular expression matching operations. This is, for most anti-patterns, a perfect way for detecting them, since we are trying to find certain patterns in queries.

4.2.1 Implicit Columns

The *Implicit Columns* anti-patterns is perhaps the easiest anti-pattern to recognize. The anti-pattern occurs when a query contains the wildcard selector (`*`) in a `SELECT` clause. This means that it simply suffices to check if the given query contains this pattern. We can use the Python `re` module to create a regular expression pattern from a certain regular expression. This can be easily done as shown in Code Listing 4.1.

We must take into account that SQL is case insensitive by default. Luckily, the `re` module allows to pass in flags that modify the expression's behavior. For case insensitive matching, we use the `re.IGNORECASE` flag. Next, we can check if the pattern appears in the given query. This is

```
pattern = re.compile("(SELECT\\s+\\*)", re.IGNORECASE)
```

Code Listing 4.1: Compiles a regular expression pattern provided as a string into a regex pattern object used to find `SELECT *`.

again done with the regular expression module provided by Python. The regular expression above matches any string that contains "SELECT" in either uppercase or lowercase (or a combination of both), followed by at least one whitespace character that is eventually followed by a "*".

```
pattern.finditer(query)
```

Code Listing 4.2: Returns an iterator yielding match objects over all non-overlapping matches for the RE pattern in a string.

In the code shown in Code Listing 4.2, pattern must be a regex pattern object. The `finditer()` function will give us a list of matches over which we can iterate. We know that if this list has at least one entry, then the *Implicit Columns* anti-pattern exists in the query. We can then report this anti-pattern.

4.2.2 Poor Man's Search Engine

The *Poor Man's Search Engine* is another pattern that could be avoided since there exist better, more reliable, and more efficient tools for pattern-matching against a database. Much like the previous anti-pattern, we can again use the Python `re` module to create a regular expression pattern from a certain regular expression. However, this time we have two possible patterns for which we must both check their occurrence, namely the `LIKE` and the `REGEXP` keywords. Therefore, instead of creating a single regular expression pattern object, we now create a list of two objects, one for each pattern.

```
patterns = [re.compile("(LIKE)", re.IGNORECASE),
            re.compile("(REGEXP)", re.IGNORECASE)]
```

Code Listing 4.3: Creates a list of two regex pattern objects.

Now, instead of looking for the pattern directly, we use a simple for loop that allows us to check the occurrence of both patterns and act accordingly. This is shown in Code Listing 4.4.

4.2.3 Random Selection

Just like the previous anti-pattern, with the *Random Selection* anti-pattern we must search for more than one pattern. This anti-pattern occurs when the query contains a random `ORDER BY`. As such, we must look for this occurrence in the input query. Our tool tries to be SQL dialect independent as much as possible so that queries with multiple dialects can be inserted by the user. Since the equivalent of the `RAND()` function used for example in MySQL, is `RANDOM()` in for example PostgreSQL and SQLite, we check for both patterns. The regular expressions needed to do this are depicted in Code Listing 4.5.

Checking if a pattern occurs in the input query can now be done in the same way as Code Listing 4.4.

4.2.4 Fear of the Unknown

The *Fear of the Unknown* anti-pattern is the last anti-pattern that we can simply detect by solely using a regular expression. This anti-pattern occurs when the input query contains a wrong `NULL`

```
            for pattern in patterns:
                for match in pattern.finditer(self.query):
                    # Handle detection
                    ...
```

Code Listing 4.4: Detection of this anti-pattern using a double for-loop.

```
patterns = [re.compile("(ORDER\\s+BY\\s+RAND\\s*\\()",
                       re.IGNORECASE),
            re.compile("(ORDER\\s+BY\\s+RANDOM\\s*\\()",
                       re.IGNORECASE)]
```

Code Listing 4.5: Creates a list of two regex pattern objects for finding the *Random Selection* anti-pattern.

comparison. We must check if the input query contains these wrong comparisons, which are shown below in Code Listing 4.6.

```
patterns = [re.compile(r'<>\s*NULL', re.IGNORECASE),
            re.compile(r'!=\s*NULL', re.IGNORECASE),
            re.compile(r'=\s*NULL', re.IGNORECASE)]
```

Code Listing 4.6: Creates a list of three regex pattern objects for finding the *Fear of the Unknown* anti-pattern.

Now, we can again loop over every pattern in the patterns list and see if there exists a match between the pattern and the input query as shown in Code Listing 4.4. If that is the case, we know that this anti-pattern exists.

4.2.5 Ambiguous Groups

This anti-pattern occurs when the single value rule is broken. That is, there exists a column in the `SELECT` statement of a query that does not have a single value row per row group. Aggregate functions, like `MAX(), COUNT(), etc` are guaranteed to have a single value for each group. Columns that are listed in the `GROUP BY` statement are also guaranteed a single value. Hence, we must look at the columns in the `SELECT` clause that do not occur in the `GROUP BY` statement or those that are not aggregated.

For efficiency, we first check if the input query contains a `GROUP BY` statement. This is done using the `re` module to create a regular expression pattern and checking if the input query has any matches to it, shown in Code Listing 4.7.

If no match is found between the current query and the regular expression, we know that this anti-pattern will not occur, so we return. If there is a match, a series of steps have to be performed to determine whether it is a valid query or a query containing this anti-pattern. By using a combination of `sqlparse` and regular expressions, we are able to obtain a list of columns that appear in the `SELECT` statement as well as another list containing the columns from the `GROUP BY` statement. By converting these lists to sets, we will remove all duplicates. Now we use the set difference to obtain all columns appearing in the `SELECT` clause that does not appear in the `GROUP BY` clause. Finally, we check if there exists a column that exceeds the single value rule. For each column, we check if it contains an aggregate function. As soon as a column is found that does not contain an aggregate function, we know that the single value rule has been broken and that the current query contains the *Ambiguous Groups* anti-pattern.

However, important to this anti-pattern is that we must compare the columns in the `SELECT`

```
pattern = re.compile(r'GROUP\s*BY', re.IGNORECASE)
```

Code Listing 4.7: Compiles a regular expression pattern provided as a string into a regex pattern object used to find `GROUP BY` clauses.

```
# Check whether GROUP BY is found in the query
if pattern.search(query):
    # Get columns in SELECT & GROUP BY statement
    select_columns = get_columns_from_select_statement(query)
    group_columns = get_columns_from_group_by_statement(query)

    # Use set difference
    remaining_columns = list(set(select_columns) - set(group_columns))

    # Check if the remaining columns break single-value rule
    single_values = check_single_value_rule(remaining_columns)

    if not_single_values:
        # Handle detection
        ...
```

Code Listing 4.8: Steps needed to check whether this anti-pattern occurs.

and `GROUP BY` clauses only to the current level. This is best illustrated with the example shown in Query 4.1.

```
SELECT cID,
  cName,
  pID
FROM (
    SELECT cID,
      cName,
      MAX(date)
    FROM customer
      JOIN shoppinglist USING (cID)
    GROUP BY cID,
      cName
  )
```

Query 4.1: A valid query without any anti-patterns where the ambiguous groups anti-pattern might be detected as a false positive.

If we simply use a regular expression to grab all the columns from the `SELECT` statement, we would obtain a list containing the columns `cID, cName, pID, cID, cName` and `MAX(date)`. Note that the columns from both inner and outer queries occur in the same list. If we then follow the steps described in Code Listing 4.8 and obtain the set difference, we are left with both `pID` and `MAX(date)` resulting in a false positive detection, since `pID` breaks the single value rule, but the input query is valid. We must therefore parse the query to obtain a list of the outer query and all its subqueries.

We again use `sqlparse` for subquery parsing. Using this library, we can parse the input query using the `parse()` function into a tokenized stream of keywords, statements, and values. Next, we search for any occurrence of the `SELECT` statement and replace subqueries by a constant string "<subquery>" and adding it to a list. We repeat this for all subqueries. The output of this function

is a list of queries, where the first element of this list is the outer query and the subsequent elements are subqueries. Query 4.1 will be parsed in the following way:

```
["""SELECT cID, cName, pID
    FROM <subquery>""",
 """SELECT cID, cName, MAX(date)
    FROM customer JOIN shoppinglist USING (cID)
    GROUP BY cID, cName"""]
```

Code Listing 4.9: An example of a parsed query.

Now, instead of performing the steps to detect this anti-pattern once for the entire query, we perform the steps for each query in this list. If we look back at the example query shown in Query 4.1, and use the parsed list of subqueries from Code Listing 4.9, we will now find that this anti-pattern does not occur. The outer query does not contain any `GROUP BY` statements, making it impossible for this anti-pattern to occur. The subquery however does contain a `GROUP BY` statement, but after performing the steps from Query 2.12, we will not find any column that breaks the single value rule.

4.2.6 Spaghetti Query

The *Spaghetti Query* anti-pattern occurs when a query is considered too complex. This can for example result from using many joins, or nesting subqueries. Measuring code complexity might seem like a very subjective task, however, there are numerous methods available that can be used to objectively determine the complexity of a query.

Methods of calculating query complexity

Internally, query optimizers assess the complexity of queries. This approach, however, concentrates on query runtime and space complexity while completely ignoring the work users must expend in crafting queries. In order to determine the complexity of raw SQL queries, we must fall back to more traditional complexity measures. Our objective is that given a raw SQL query, we need to find a suitable method that can help us determine its complexity.

The simplest method to determine query complexity would be to count the number of lines of code (LoC). This comes with a few problems. The first problem would be that each person has their own way of formatting a query. For example, say you want to fetch a few columns from a certain table. Then you would want to use a simple `SELECT FROM` query. Some like to put every column in a `SELECT` clause on a separate line, whilst others like to put them on the same line. If we would then count the number of lines, the former formatting style would appear to be more complex than the latter, but the queries are considered identical. Formatting the query beforehand using `sqlparse` would solve the problem partially. By doing so, we can ensure that all input queries will be transformed into a certain fixed format so that we can have fair comparisons. However, there still exists a second problem with this method. Suppose we have two queries as shown in Query 4.2a. One query selecting three columns from a certain table, whilst the other query selects four. If we use `sqlparse` to format the query, every column in the `SELECT` clause will be put on a separate line. This implies that the first query is less complex than the second. Which is correct, albeit by a very little margin. Now suppose that we add a `WHERE` clause to the first query with one simple condition, shown in Query 4.2b. This would mean that both queries have the exact same amount of lines of code and thus would be considered even in complexity. One could argue that having an extra `WHERE` clause instead of an extra column would make the query more complex, showcasing that this method of determining complexity is not ideal.

Another way of measuring code complexity is through Cyclomatic Complexity [19]. This software metric is used to determine the number of linearly independent paths through a program's

```
SELECT cID,
  cName,
  street
FROM customer
```

```
SELECT cID,
  cName,
  street,
  city
FROM customer
```

(a) The right query is more complex than the left.

```
SELECT cID,
  cName,
  street
FROM customer
WHERE cID > 5
```

```
SELECT cID,
  cName,
  street,
  city
FROM customer
```

(b) Both queries are of the same complexity.

Query 4.2: Example illustrating a problem using LoC to determine complexity.

source code by obtaining the control-flow graph of the program. This was a graph consisting of nodes, that correspond to indivisible groups of a program's commands, and edges connecting a pair of nodes. If there is an edge $e_{1,2}$ between node v_1 and v_2, then this implies that the group of commands of node v_2 might be executed immediately after the commands from node v_1. Once the control-flow graph is obtained, we can compute the Complexity m using the following formula:

$$M = E - N + 2P \tag{4.1}$$

Where:
E is the number of edges of the control-flow graph.
N is the number of nodes of the control-flow graph.
P is the number of connected components, which is the number of disjoint sets of nodes and edges.

The drawback of this method is that for SQL, obtaining a control-flow graph is not easy as there are simply too many clauses. Another argument made when it comes to using Cyclomatic Complexity is that it is based upon poor theoretical foundations and an inadequate model of software development [37].

A better method for determining query complexity would be to compute the Halstead Complexity presented by Maurice Howard Halstead in 1977 [14]. Halstead stated that many characteristics of a software system can be stated using only the number of operands and operators found in the software. In a program, all variables are considered operands, while all control statements such as `if, while, for` are considered operators. Given η_1, the number of distinct operators, η_2, the number of distinct operands, N_1 the total number of operators and N_2 the total number of operands A variety of measurements may be derived from these variable shown in Equation 4.2 to Equation 4.7.

Equation 4.2 and Equation 4.3 compute the program vocabulary and program length respectively. Equation 4.4 estimates the length with only the number of distinct operators and operands. Equation 4.5 calculates the program volume, which is the bit size of space necessary to store the program. The next equation, Equation 4.6, calculates the program difficulty, which is related to the ability to write or understand the program. Lastly, Equation 4.7 calculates the program effort, which can be used to obtain estimates in the time required to program and the number of errors in the implementation.

A recent work that was published in 2015 provides an analysis of queries from the SQLShare workload, to come up with an empirical formula to measure the query complexity as a function of the cognitive load on users and performance load on the system [44]. They then tested different methods to determine query complexity and compared them with one another and with the ground truth complexity score. The ground truth complexity score was manually determined by rating 117 distinct queries with a number on a scale of 100, with the higher the number indicating greater complexity. The other methods used to determine complexity were linear regression, expert ranking, and Halstead measures. However, the authors use an adapted version of the Halstead

$$\eta = \eta_1 + \eta_2 \tag{4.2}$$

$$N = N_1 + N_2 \tag{4.3}$$

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \tag{4.4}$$

$$V = N \times \log_2 \eta \tag{4.5}$$

$$D = \frac{\eta_1}{2} \times \frac{N2}{\eta_2} \tag{4.6}$$

$$E = D \times V \tag{4.7}$$

measures specifically designed for queries. This version uses the following complexity equation:

$$Query\ Complexity = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2} \times \log_2(\eta_1 + \eta_2) \tag{4.8}$$

Here, similar to the ”standard” Halstead measure, η_1 is the number of distinct operators, η_2 is the number of distinct, N_1 is the total number of operators and lastly, N_2 is the total number of operands. The authors considered the number of columns referenced in a query as operands and the number of operators and expressions in a query as Halstead operators. After a comparison between the different methods, they concluded that their modified Halstead measure version was marginally better than the other methods and was closest to the ground truth. However, they also state that query complexity is inherently difficult to quantify and that there is still room for improvements in this research area.

Determining query complexity in our detector

We decided to use the Halstead Complexity variation described in [44] as this was one of the only papers on the topic of calculating query complexity. The first task is to find the Halstead operators and operands. The paper states that the operators and expressions of a SQL query can be considered Halstead operators. Based on this notion, we defined two lists of all operators and expressions that are possible in SQL. These lists can be found in Code Listing 4.10. Furthermore, it states that every column referenced in the query can be considered as Halstead operands.

To find the Halstead operators, we wrote a helper function that returns a list of all the operators and expressions used inside a query. We do this by using `sqlparse` to first format the query and then use the Python String `split()` function to split the query into a list where each word in the query is a list item. In doing so, a problem arises where we might have split a single SQL expression into two list elements. For example, if the original query contains a `ORDER BY` clause, like the example query shown in 4.9, this will be split into two list elements namely `ORDER` and `BY` as shown in 4.10. This problem occurs with all SQL expressions that have a whitespace between them, i.e. `IS NULL`, `IS NOT NULL`, `ORDER BY` and `GROUP BY`. We must loop over the elements from the obtained list and check whether adjacent elements form one of these expressions. If that is the case, we merge the cells into one cell. The resulting list for the example query from 4.9 is shown in 4.11. Note that we also merge `SELECT` and `*` if they are adjacent, since the `*` character

```
OPERATORS = ["+", "-", "*", "**", "/", "%", "&", "|", "||", "^",
             "=", "!=", ">", "<", ">=", "<=", "!<", "!>", "<>",
             "+=", "-=", "/=", "/=", "%=", "&=", "^-=", "|*=",
             "ALL", "AND", "&&", "ANY", "BETWEEN", "EXISTS",
             "IN", "LIKE", "NOT", "OR", "SOME", "IS NULL",
             "IS NOT NULL", "UNIQUE"]

EXPRESSIONS = ["CASE", "DECODE", "IF", "NULLIF", "COALESCE",
               "GREATEST", "GREATER", "LEAST", "LESSER",
               "CAST", "JOIN", "GROUP BY", "WHERE",
               "HAVING", "ORDER BY", "UNION","EXCEPT"]
```

Code Listing 4.10: A list of all SQL operators and expressions.

is also used as the multiplication operator in SQL. By merging the two, we prevent the wildcard select character from being seen as an operator. Now, we check for each item in the list if they are equal to an operator or an expression from the ones defined in Code Listing 4.10. If this is the case, we have found a Halstead operator which we will remember by storing it in a new list.

$$\texttt{SELECT *, FROM purchase ORDER BY quantity} \tag{4.9}$$

$$\downarrow$$

$$\texttt{["SELECT", "*", "FROM", "purchase", "ORDER", "BY", "quantity"]} \tag{4.10}$$

$$\downarrow$$

$$\texttt{["SELECT *", "FROM", "purchase", "ORDER BY", "quantity"]} \tag{4.11}$$

To obtain all Halstead operands, we use SQL parse to get all columns referenced in the query. In SQL, columns can be referenced in `SELECT, JOIN, ORDER BY`, and `GROUP BY` statements. We also have to take into account that a single query might contain subqueries or consist of a union of queries. For subqueries, we use subquery parsing to get a list of all queries and obtain all Halstead operands for each query. For unions, we again obtain a list of queries by splitting the query at the `UNION` keyword.

We now have a list of columns, which are our Halstead operands, and we have a list of operators and expressions, which are our Halstead operators. We can now obtain variables $\eta_1, \eta_2, N1, N2$ and compute the query complexity. Next, we set three thresholds for the query complexity, which we found using trial and error. If the query complexity is higher or equal to the lowest threshold, we report the *Spaghetti Query* anti-pattern with low certainty. Every time we exceed another threshold, meaning that the query is more and more complex, we increase the certainty that this anti-pattern is present in the current query until we have reached high certainty.

4.3 Distributing the detector

Since our research is directed towards novices, we want to make our anti-pattern detector as accessible as possible. Therefore, we will distribute our detector in three different ways. The first way is via a Python package, which includes a command-line interface (CLI). The second way is via a RESTful API. The last way is via a web application. In this section, we will go over the different ways of distribution.

4.3.1 Python package with CLI

Our anti-pattern detector is called `SQLEyes` and is available on Github[1] and the Python Package Index, also known as Pypi[2]. As mentioned in the previous section, the detector is written in Python. Python is a very accessible language that most novice users will have installed on their local machine. Besides making an accessible tool for novices, another goal was to make the detector as extensible as possible. As a result, when new anti-patterns are discovered, it is simple to add new rules to `SQLEyes`, making it a more universal tool. Another benefit is that `SQLEyes` could also be used to detect more general errors and misconceptions, thus providing a more complete learning system.

Using the `argparse` module, we created an easy-to-use command-line interface (CLI). This CLI program as well as the main detector program is exported as a single package and is uploaded to the Python Package Index. This means that novice users can simply run `pip install sqleyes` to download and install our detector.

After installation, the user can open a terminal and type `sqleyes -h` to open a help guide showing all possible options and arguments, this view is shown in Figure 4.1.

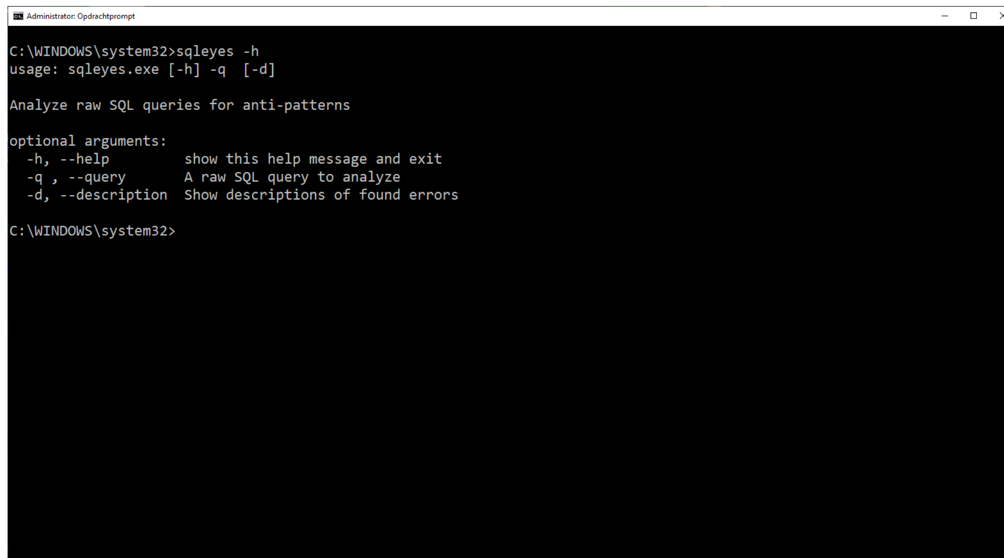


Figure 4.1: The help menu of SQLEyes, showing the possible arguments.

The user can insert a query using the query argument. This is done by utilizing the `-q` or `--query` flag followed by the query to be analyzed. The detector is started when the user presses the enter key. It will now go over all of the previously disclosed anti-patterns and detecting techniques. Finally, it will show a short summary of the anti-patterns that are found by the detector. This summary gives a short title, telling users without the knowledge of anti-patterns what is wrong with the query, the type of anti-pattern, the certainty of the anti-patterns occurrence, and the location where the anti-pattern is detected.

Suppose we pass the query shown in Query 4.3 as input to the CLI. This query uses the wildcard selector, meaning that the *Implicit Columns* anti-pattern occurs. Secondly, it does not handle `NULL` values the right way, meaning that the *Fear of the Unknown* anti-pattern is also present. Furthermore, it uses pattern matching, which is an anti-pattern in itself. Lastly, it uses `ORDER BY random()` to sort the output randomly, this is known as the *Random Selection* anti-pattern. The output is shown in Figure 4.2.

We can also run the same input with the `-d`, also known as the `--description` flag enabled.

[1] https://github.com/leonardomathon/sqleyes
[2] https://pypi.com/projects/sqleyes

```sql
SELECT *
FROM product
WHERE pCat <> NULL
  AND pName LIKE "%ice%"
  AND price <> NULL
  AND pSection LIKE "self%"
GROUP BY name
ORDER BY random()
LIMIT 1
```

Query 4.3: An example query containing various anti-patterns.

```
Selecteren Administrator: Opdrachtprompt

C:\WINDOWS\system32>sqleyes -q "SELECT * FROM product WHERE pCat <> NULL AND pName like "%ice%" AND price <> NULL AND pS
ection LIKE "self%" GROUP BY name ORDER BY random() LIMIT 1"
SQLEyes v0.5.0

Summary of analysis
Found 5 errors in the given query
```

Error	Title	Type	Certainty	Location
anti-pattern	Incorrect NULL usage	Fear of the Unknown	high	(33, 40), (72, 79)
anti-pattern	Avoid usage of wildcard selector	Implicit Columns	high	(0, 8)
anti-pattern	Avoid pattern matching	Poor Man's Search Engine	medium	(51, 55), (93, 97)
anti-pattern	Avoid ORDER BY RAND() usage	Random Selection	high	(118, 134)
anti-pattern	Avoid complex queries	Spaghetti Query	low	

```
C:\WINDOWS\system32>
```

Figure 4.2: The minimal output of SQLEyes when the -d flag is not set.

```
Administrator: Opdrachtprompt

C:\WINDOWS\system32>sqleyes -q "SELECT * FROM product WHERE pCat <> NULL AND pName like "%ice%" AND price <> NULL AND pSection LIKE "self%" GROUP
BY name ORDER BY random() LIMIT 1" -d
SQLEyes v0.5.0

Summary of analysis
Found 5 errors in the given query
```

Error	Title	Type	Certainty	Location
anti-pattern	Incorrect NULL usage	Fear of the Unknown	high	(33, 40), (72, 79)
anti-pattern	Avoid usage of wildcard selector	Implicit Columns	high	(0, 8)
anti-pattern	Avoid pattern matching	Poor Man's Search Engine	medium	(51, 55), (93, 97)
anti-pattern	Avoid ORDER BY RAND() usage	Random Selection	high	(118, 134)
anti-pattern	Avoid complex queries	Spaghetti Query	low	

```
Detailed descriptions of found errors
----------

type: Fear of the Unknown
title: Incorrect NULL usage
location(s): (33, 40), (72, 79)


  Query contains an Anti-Pattern:

  ...E pCat <> NULL AND pN...
          ^------


  Query contains an Anti-Pattern:

  ... price <> NULL AND pS...
          ^------

description:

                    Fear of the Unknown

  In SQL, values in columns can be left empty. This results in an attribute of a certain row having a NULL value. SQL
  considers NULL to be a special value, distinct from zero, false, true, or an empty string. Therefore, it is not
  possible to test for NULL values with standard comparison operators such as =, >=, <>, etc. Instead use IS NULL and
  IS NOT NULL.

                    Example code

   SELECT pName, suffix
   FROM products
   WHERE suffix <> NULL;

  The code shown above is querying the product name and suffix columns from the products table where the suffix is
  not equal to NULL. One might think that this will result in all rows that have a suffix, however this is not the
  case. Any comparison to NULL returns unknown, not true or false. Therefore, this query does not return any data.
```

Figure 4.3: The detailed output of SQLEyes when the -d flag is set.

This will provide error highlighting as well as a more detailed description of each anti-pattern that was detected and a possible solution to fix the anti-pattern in addition to the summary output. This output is shown in Figure 4.3.

4.3.2 RESTful API

The next way of distributing the anti-pattern detector is via a RESTful API that could be self-hosted. The main benefit is that novices do not have to install a program, they just need to run the right request to the API endpoint. Besides being easier to access than the CLI, the API may also be easily integrated into existing applications where raw SQL queries are used.

Our API is built on top of Flask[3] and uses the `SQLEyes` Python package. Flask is a Python web framework that utilizes useful functions and packages that allow for quickly setting up a REST API. It is therefore a very well-established and well-liked framework for building a RESTful API. The user can make a `POST`-request to the API endpoint with the query to be analyzed as payload. This will return a JSON object with a list of the certainties, descriptions, locations, titles, and types of identified anti-patterns. Code Listing 4.11 shows an example HTTP POST request to the API hosted on a server.

```
POST /analyze HTTP/1.1
Host: api.querysandbox.com
Content-Type: application/json
Content-Length: 88

{
    "query": "SELECT pId, supplierId, AVG(price) FROM product WHERE
       price <> NULL"
}
```

Code Listing 4.11: An example API request directed to the API.

4.3.3 Web application

The anti-pattern detector can also be distributed via a web application that can perform and analyze a SQL query. Users simply run the query in their web browser, and the program analyzes it. This is perhaps the most accessible way of using the anti-pattern detector, as it provides the user the ability to execute a query without writing a single piece of code apart from their query or downloading any software.

The web application is called QuerySandbox, as it is a sandbox environment where users can load custom databases and run queries on them. It is heavily inspired by sites like CodeSandbox[4] and CodePen[5]. In the next section, chapter 5, we will go into more depth into the development of QuerySandbox, as well as the various capabilities it provides.

4.4 Summary

In this chapter, we discussed the development process of our anti-pattern detector. This process, along with the process described in the next chapter, answers research question 1 (RQ1). We began by responding to RQ1.1, which asked how we defined the scope of the anti-patterns we wanted to detect. This was accomplished by defining which anti-patterns are detectable from raw

[3]https://flask.palletsprojects.com
[4]https://codesandbox.io/
[5]https://codepen.io/

queries. Following that, we described the specific anti-patterns we were able to detect as well as the methods we used to detect them. Finally, we discussed the three ways the detector was distributed: as a Python package with a CLI, a RESTful API, and a web application.

Chapter 5

Designing a detector application

As mentioned in the previous chapter, this chapter will discuss on the design and implementation of QuerySandbox, our sandbox environment for analyzing SQL queries. In doing so, we will also answer research question (RQ1). We will first go over a couple of important findings in the User Interface/User Experience (UI/UX) domain, which we incorporated into our design. We then establish a set of design requirements using the MoSCoW method [9]. Some of these requirements are linked to UI/UX findings, while others were discussed during weekly meetings. Next, we present the design and development of QuerySandbox in more detail.

5.1 User interaction and user experience

For our application, we did not want to design a complex user interface with too many features, because we did not want to overwhelm the user with too much information. We wanted to design an application that is easy to use and self-explanatory. Users should be able to use QuerySandbox without having to read the documentation. But how does one design such an application? Designing an application seems like a trivial task. Designing an application with a good user interface and user experience, on the other hand, is not always an easy task. The user interface is the first point of interaction between the user and a software application. The user experience goes beyond the user interface to include the user's perceptions and emotions as they interact with the software. Before we formed the main requirements for our application, we looked into some important works of user interface and user experience design. We also looked into some design principles that can help us create a better user interface for our application. There exist various design principles that aim to assist in creating an effective user interface and experience. When forming the requirements for the application, we will keep these principles in mind. Since not all of these design principles are applicable to our application, we will only discuss the ones we considered during the design of QuerySandbox.

One of the most important works on the design of, not just software applications, but things, in general, is The Design of Everyday Things by Don Norman [27]. The author makes a case for Human Centered Design in The Design of Everyday Things, describing how designers should think about users and their problems. He deconstructs common design flaws and provides readers with a framework for user-centered design, which means that designers should design for people and their needs first. The author states that a usable design starts with careful observations of how the tasks being supported are actually performed, followed by a design process that results in a good fit to the actual ways the tasks get performed [27, p. 137]. He also believes it is important to make it easier for people to discover and correct errors that occur since we should never blame a user for an error. In our case, we need to look at how our users will use our application to analyze raw SQL queries. We must also make sure that when an error occurs, for example when the SQL syntax is not correct, the user is guided into fixing this error. Therefore, we must ensure that users will be notified when such errors occur.

In an article written by the Nielsen Norman Group, a consultancy firm on computer user interfaces and user experiences, they state ten important usability heuristics [26]. Three heuristics are particularly important, namely user control and freedom, recognition rather than recall, and flexibility and efficiency of use. The first heuristic states that users frequently make mistakes when performing actions. They require a clearly marked "emergency exit" in order to leave the unwanted action without having to go through a lengthy procedure. The second important heuristic tells us to make elements, actions, and options visible to reduce the user's memory load. The last heuristic tells us that shortcuts, which should be hidden from novice users, may speed up the interaction with the system for the expert users making the user experience more pleasant for them. We have taken these heuristics into account when designing the UI of our application: we help our users get rid of mistakes by pressing a single button, we reduce the user's memory load by having a clean and minimalistic interface with only the core features prominently visible and we implemented a shortcut for every command and action a user can perform.

Furthermore, there are lots of blog posts and articles discussing various design tips and tricks to create a good UI design that benefits the UX. These blog posts and articles discuss various issues such as typography, design inspiration, layout, color theory, dark mode, and so on. These tips and tricks are also used in the UI design of our application.

5.2 Application requirements

Using the insights obtained from the above-mentioned works, we now establish a set of design requirements prioritized using the MoSCoW method [9]. The MoSCoW method of prioritization is a tool used to rank the importance of project deliverables. The acronym stands for Must, Should, Could, and Would. Here is a brief overview of each category:

- **Must**: These are the project requirements that are absolutely essential for the project to be successful. Without these requirements, the project would be considered a failure.
- **Should**: These are important project requirements, but they are not essential for the project's success. The project could still be completed without these requirements, but it would not be as successful.
- **Could**: These are optional project requirements that would be nice to have, but are not essential.
- **Would**: These are project requirements that would be nice to have, but are not essential and are not likely to be completed.

Before creating the requirements, we had to establish the main use case of our application. Our goal was to increase awareness of SQL anti-patterns for novices. This goal can be met by creating an application where novices can enter and analyze a SQL query. By doing so, potential SQL anti-patterns are highlighted which enables novices to learn from them. When then thought of the way the task, namely SQL query analysis, is performed by the end-user. We already have two ways of analyzing queries, namely through the SQLEyes CLI or API. The application is a novel approach for analyzing SQL queries since the users do not have to install anything. They can analyze queries by simply navigating to the website, enter a query in a text box and hit the analyze button. The analysis will then be shown to the user. Based on our goal and the use case, we created requirements. The following requirements are made:

Must have

M1. The system shall contain a section to enter a query and a section to display the query analysis output.

M2. When a query is entered and the "run" button is clicked, the system shall analyze the query for anti-patterns and display the output.

M3. When a query contains an anti-pattern and the query is analyzed by the system, the query analysis output shall display a problem description of

the anti-pattern, an example, a potential fix, and show where the anti-pattern is found within the query (as provided by the SQLEyes API).

Should have

S1. The system shall display entered queries with syntax highlighting.
S2. When a query is entered and the "format" button is clicked, the system shall format the inserted query.
S3. When a query is entered and the "clear" button is clicked, the system shall remove the query, the content of the query output section, and the content of the query analysis output section.
S4. When a valid SQLite database is loaded, a query is entered, and the run button is clicked, the system shall contain a section to display the query output, run the query and display the result-set within this section.
S5. When the SQLEyes API is unavailable or unable to process the query and the ”run” button is pressed, the system should display an error message with the appropriate error description in the query analysis output section.
S6. When a valid SQLite database is loaded, a query with at least one syntax error is entered, and the ”run” button is pressed, the system should display an error message with the appropriate error description in the query output section.

Could have

C1. When the "undo/redo" button is clicked, the system will undo/redo the last action performed by the user.
C2. When the "toggle dark mode" is clicked, the system will undo/redo the last action performed by the user.
C3. The system shall contain keyboard shortcuts (hot keys) to invoke certain actions and provide the user with immediate access to them.
C4. The system can be used as a standalone application, without the need for a web browser.

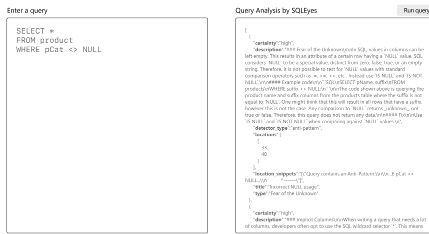
Would have

W1. When a valid SQLite database is loaded and the "view schema" button is clicked, the system shall display the database schema.
W2. When an anti-pattern is found in a query, the system will highlight it in real-time in the query input box.

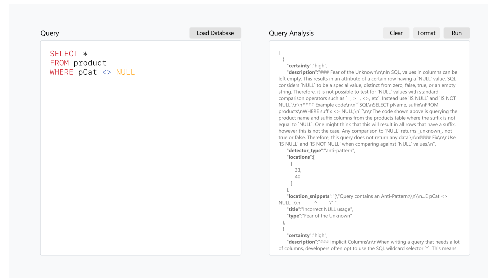
As said before, the "Must have" category are essential features that must be implemented. When these features are not available, we consider this project a failure. The requirements we listed in the "Must have" category are the bare necessities a SQL anti-pattern analysis application should have. For our use case three requirements are a must have: the user can enter text (M1), the user can analyze the query by clicking a button (M2) and the analysis will be shown to the user (M3).

The requirements listed in the "Should have" category are features that make our application worthwhile to other users. With these requirements implemented, the application can be used more generally since it now not only is a SQL anti-pattern analysis application, but also an application that can run actual queries.

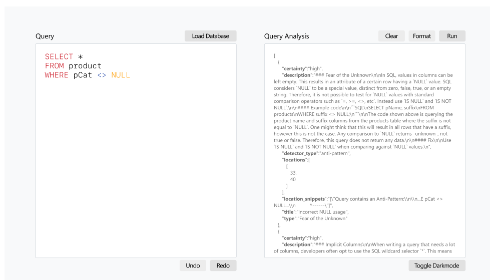
The requirements listed in the "Could have" and "Would have" make the application more user friendly. These requirements are all small requirements that are seen in comparable applications because they make these application more attractive to the end-user and increase productivity.



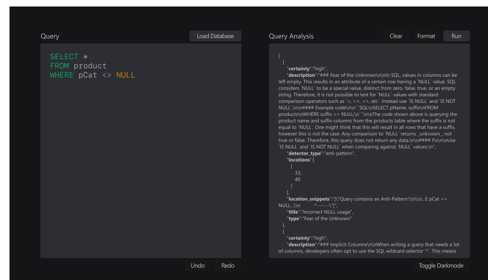
(a) Implements M1 to M3



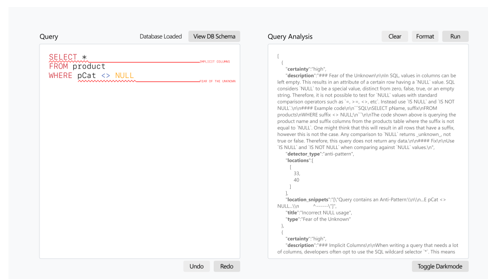
(b) Implements M1 to S6



(c) Implements M1 to C4



(d) Darkmode version of (c)



(e) Implements M1 to W2

Figure 5.1: Mock-ups of QuerySandbox each implementing a subset of the requirements.

5.3 Application design

5.3.1 Low-fidelity mock-ups

We began by designing a user interface after thoroughly understanding the project requirements. We created several (colored) UI mock-ups to assist us in determining the system's structure. After we approved the mock-ups, we began working on the system's actual high-fidelity UI design. We started with a mock-up that only implements the "Must have" requirements. Following that, we used this mock-up as a foundation and added UI elements based on the "Should have" requirements. The third mock-up is based on the second mock-up and includes additional features based on the "Could have" requirements. The final mock-up includes all requirements, including the "Would have "requirements. The mock-ups are created in Figma [1]. All mock-ups are displayed in Figure 5.1. For a bigger version of these mock-ups, we refer to Appendix A.

5.3.2 High-fidelity final design

Figure 5.2 presents a high-fidelity final design of QuerySandbox. All requirements (both "Must have" and "Should have") were incorporated into this high-fidelity design. Due to the time con-

[1] https://figma.com

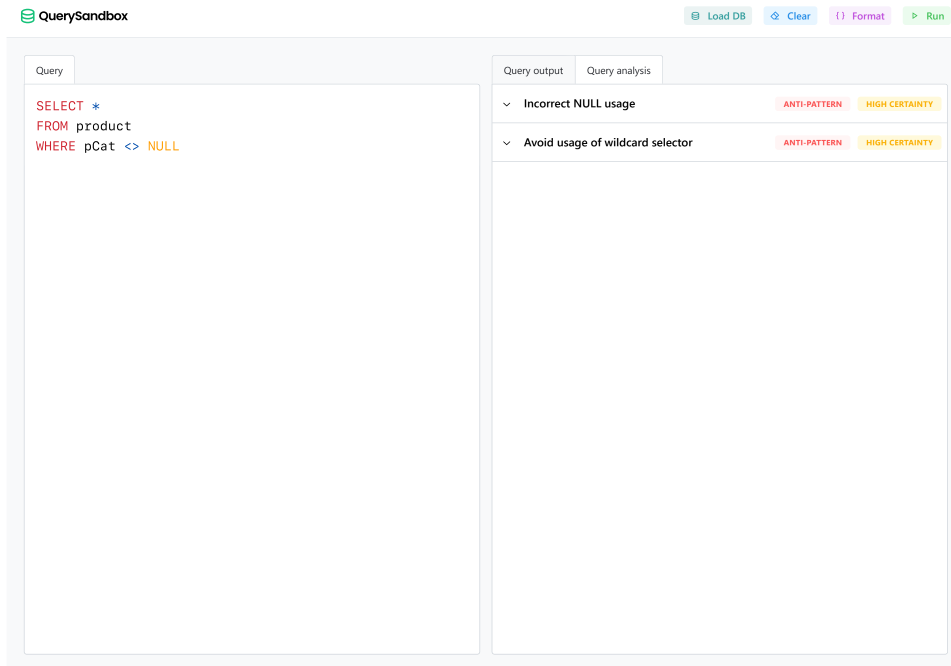


Figure 5.2: High fidelity mock-up of QuerySandbox.

straints of this project, the other requirements were omitted. For the overall look and feel of our application, we used a component library named Mantine [2]. The major advantage of using a component library is that you can focus more on the logic of the system, rather than the aesthetics of the application. Furthermore, it assists us in maintaining consistency in our application design.

The Mantine library provides a full set of components, like buttons, text inputs, and tabs. Mantine also has a Figma design file available. This allowed us to use the Mantine components in both our high-fidelity mock-up created with Figma and our React code whilst maintaining a consistent style. We also used a CSS in JS library named styled-components [3]. In contrast to traditional CSS, styled-components allows you to use JavaScript to style your HTML.

5.3.3 Application architecture & implementation

The final design was made with TypeScript & React [4] and is heavily inspired by the high fidelity mock-up shown in Figure 5.2. We chose React because it is the most popular JavaScript library for building user interfaces, its ability to create reusable components, and its ease of use with third-party Javascript libraries. The final design implements requirements: M1, M2, M3, S1, S2, S3, S4, S5, S6, C2, C3, and C4. We decided to host QuerySandbox publicly, it can be found here.

TypeScript is a JavaScript-based language that adds static type annotations to boost developer productivity and code quality. The syntax is very similar to JavaScript, in fact, its a superset of JavaScript, making it simple to learn for existing JavaScript developers. TypeScript allows the use of TypeScript-specific IDE features like code completion and type checking, which reduces the number of runtime errors. It also enables improved code refactoring support, which aids in the upkeep of large projects.

We used Vite [5] to develop the frontend. Vite is a web development build tool that uses native ES modules and TypeScript to provide a fast development setup. It serves as the main competitor

[2]https://mantine.dev
[3]https://styled-components.com/
[4]https://reactjs.org/
[5]https://github.com/vitejs/vite

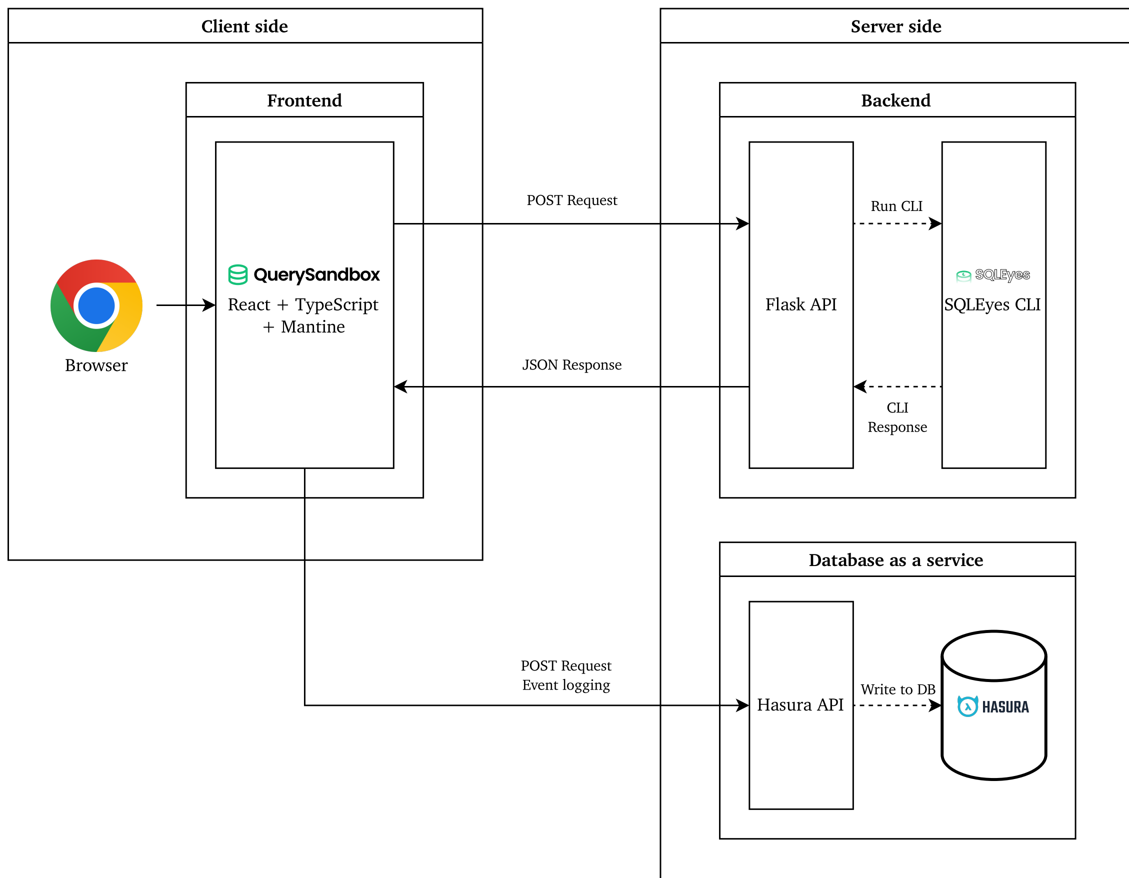


Figure 5.3: A diagram of the architecture of our app.

to Create React App [6]. The main advantage of Vite is its low build time, which is due to its ability to directly serve source files. Other advantages of Vite include its simple config and its support for hot module replacement (HMR).

Both the front and back ends of a traditional web application are implemented on the server. However, for our use case, this was not the most efficient method, since server-side rendering means that the entire application would need to be reloaded each time the user analyzes a query. Instead, we used client-side rendering, which renders the majority of the application on the client side (the user's browser). This means that when a user analyzes a query, the server only needs to respond with the query's analysis data. Client-side rendering is more efficient because it reduces server load while improving user experience by loading data more quickly. This results in a fast-loading single-page application (SPA) that fetches data asynchronously.

Figure 5.3 shows the architecture of our system. With this architecture, the complete frontend is loaded at once on the client-side. Whenever an action is performed that requires a server, in our case when the user requests a query analysis, a POST request containing the entered query is sent to the SQLEyes API running on a server. This will in turn make use of the SQLEyes CLI to analyze the query. The SQLEyes API will then parse the response from the SQLEyes CLI and return the results to the client-side. The client will then render the results and display them to the user.

For further analysis, we also log all user actions to a database. We use a database as a service called Hasura [7]. Hasura includes a PostgreSQL database, which can be used to create tables from within the Hasura environment. Hasura then provides us with a GraphQL API endpoint to which

[6] https://github.com/facebook/create-react-app
[7] https://hasura.io

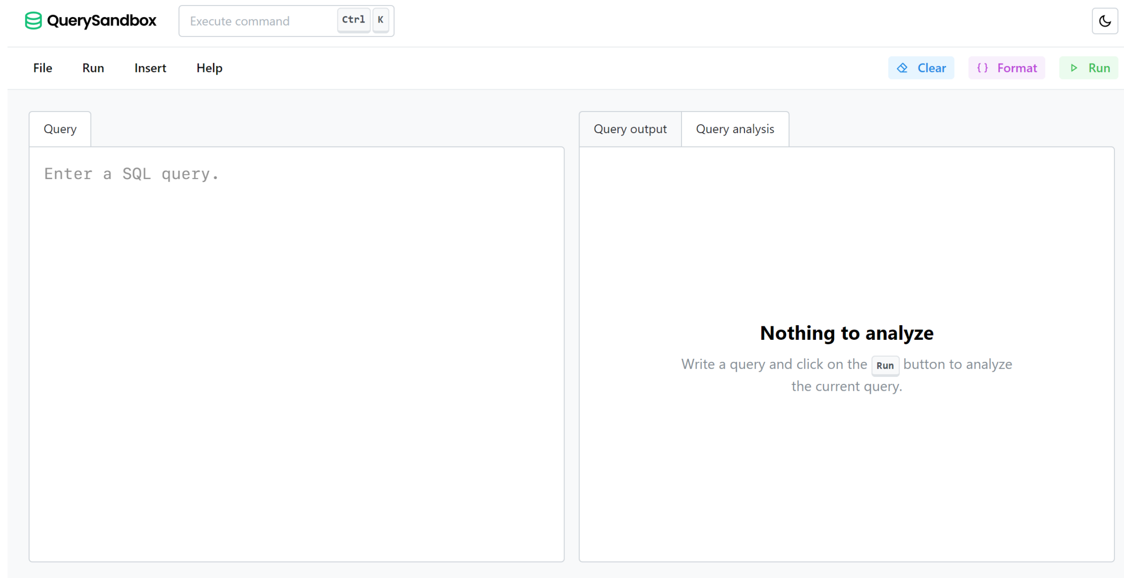


Figure 5.4: The final design of QuerySandbox.

we can make requests, including POST requests to log user actions. The requests are logged asynchronously to avoid interfering with the main application's performance. When the Hasura API receives a POST request, it processes it and creates a new entry in the database automatically. We use this service to log all user actions and track what users are doing within the application. We do not log any user data, only the interactions of the user with the application. This allows us to identify which features are being used, how often, and to improve the application accordingly. The logging mechanism is used in the user study, which will be described in chapter 6.

We also made QuerySandbox available as standalone app, as per requirement C4. For this, we used an open-source JavaScript framework named Electron [8]. Electron allows us to build cross-platform desktop apps with TypeScript, HTML and CSS. Electron works with the Chromium engine to create a web-browser like window, which is the so-called main process. We can then display local HTML files in that window. Since our frontend is a single-page application, we can also export this application to a static single-page application. This will result in a set of static HTML, CSS and JavaScript files. We then serve these files to the Electron framework, which results in a standalone, cross-platform desktop application. Lastly, we build the project, which bundles the code and the files, and creates an installer.

Now that we have taken a look at the underlying architecture of QuerySandbox, we will now go over the implementation of the requirements. We will look at various important elements from the final design, link it to certain requirements if possible, and explain its purpose. Each element will be visually supported by an image showcasing the element where we highlight the element using a green background.

Query tab

Figure 5.5 – Implements M1, S1
The query tab is the place where users can write queries. It supports syntax highlighting to help users write queries more easily. We have used a mono-spaced font for the query tab since that is common for code editing. Since this components uses the native HTML input tag under the hood, it supports the keyboard shortcut `CTRL + Z` to undo and the shortcut `CTRL + Y` to redo out of the box.

[8] `https://electronjs.org`

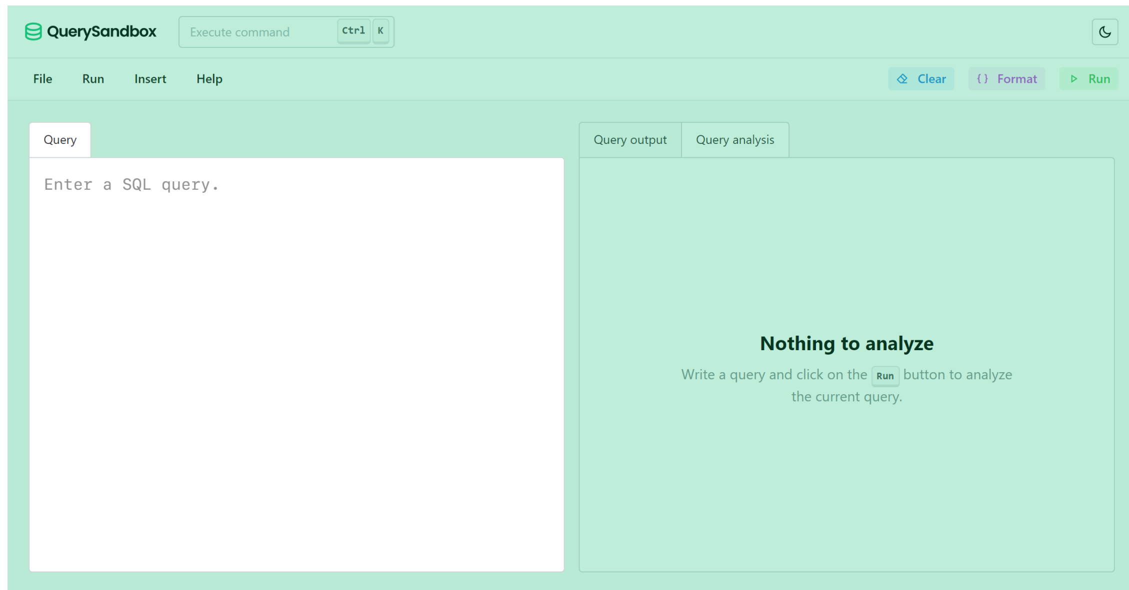


Figure 5.5: The query tab where users can write queries.

Output tabs

Figure 5.6 – Implements M1, M2, M3, S4, S5, S6
The output tabs, consisting of the query output and the query analysis output are places where output is printed. The query output will contain the result-set of the entered query when a SQLite database is loaded and the run query command is executed. If no SQLite database is loaded, the output of this tab will show "No database loaded". The query analysis tab will output the analysis done by SQLEyes. When an anti-pattern is found, it will display it and render the anti-pattern location and description. If no anti-patterns are found, it will show "No anti-patterns found".

Both tabs will display errors accordingly as per requirements S5 and S6 as shown in Figure 5.7. If the entered query contains a syntax error, an error message will be displayed in the query output tab. This error message explains the error and where it occurred. Another error message will be displayed in the query analysis tab if the request to the backend API failed. This might also be caused by a syntax error in the query, or if the backend is down. We believe these error messages are important features because of the work by Norman [27] which state that it should be easy for the user to discover and correct their errors. Without these errors, the user does not get any visual feedback as to why their query or analysis fails.

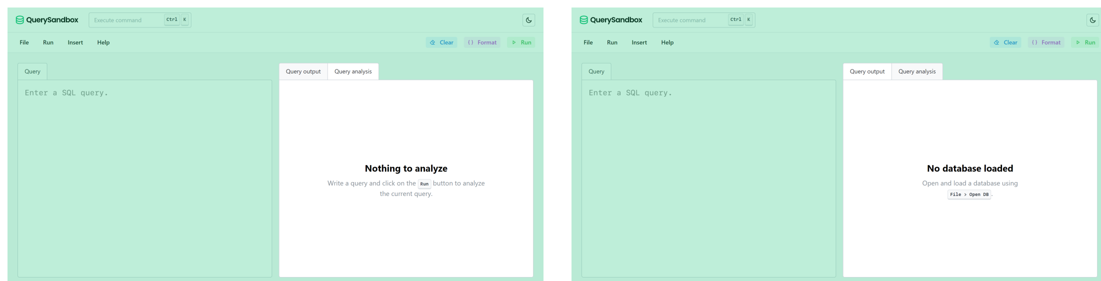


Figure 5.6: The analysis & results tabs where QuerySandbox displays the current result-set and analysis of the entered query.

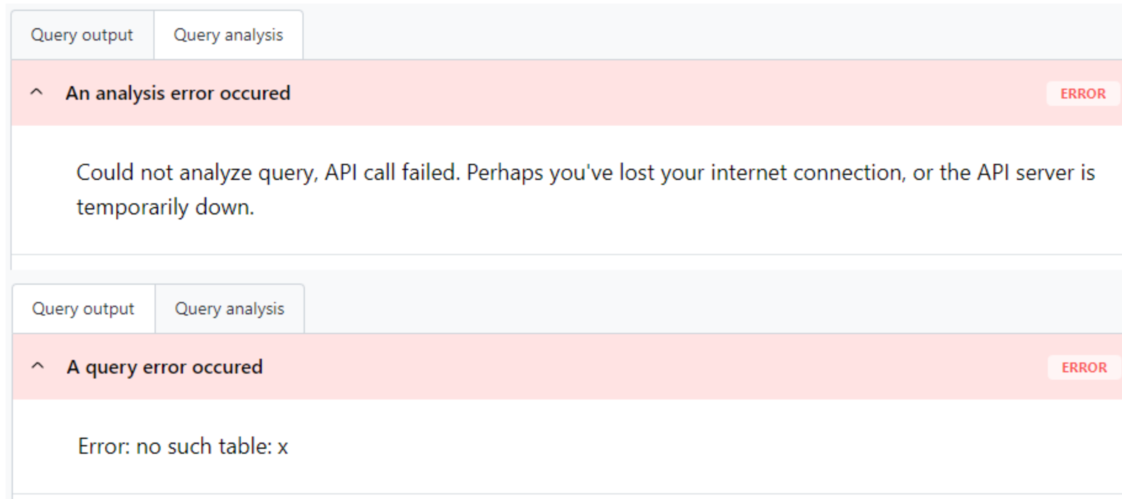


Figure 5.7: Errors shown in the output tabs when something went wrong.

Action buttons

Figure 5.8 – Implements M2, S2, S3, C3
The action buttons provide the main functionality of QuerySandbox. The "clear" button is used to remove all input and output. This means that it will not only remove the query written in the query tab, but also existing output in the query output and query analysis tabs. As stated in the literature, users make frequent mistakes when performing actions. The clear action provides an "emergency exit", or in other words, a way to quickly undo a mistake and start fresh. The "format" button is used to format the current query, which allows the user to view the query in a more readable format. The "run" button executes two subsequent actions. If a database is loaded, it will take the currently entered query, run it against the database and display the result-set in the query output tab. Secondly, it will make a POST request to the SQLEyes API to analyze the query. It will then output the response in the query analysis tab. If there is no database loaded, this action will only analyze the query. The action buttons stand out from the rest of the interface since they are given a distinct notable color. Because of this, we believe the noticeable action buttons will help reduce the user's memory load.

All three actions are mapped to a keyboard shortcut. In the case of the Windows and Linux operating systems, `CTRL + U` is used to run the clear command, `CTRL + B` is used to run the format command and `CTRL + Enter` is used the run the query. On Mac OS, the respective shortcuts are `Command + U`, `Command + B` and `Command + Enter`. Shortcuts are an important usability heuristic, as stated by [26]. They provide a user with a mechanism to quickly and efficiently perform actions, without having to use the pointing device to select an action from a menu.

Toolbar

Figure 5.9 – Additional feature
Many standalone applications include a toolbar. We wanted to provide our users with a sense of familiarity, so we added a toolbar. The toolbar has four dropdown menus. The first menu is *file*, which allows the user to load a SQLite database. If the user selects that option, a modal will appear in which a SQLite database file can be selected. This modal is discussed further in section 5.3.3. The next dropdown menu is *run*, where users can execute the main actions, *clear*, *format* and *run*. The second to last dropdown menu is *insert*, which contains five useful example queries. If the user clicks on one of these five queries, the query will automatically be loaded in the query tab. The last dropdown menu is *help*, where users can open a getting started guide from. This guide contains a quick introduction to the application and its usage.

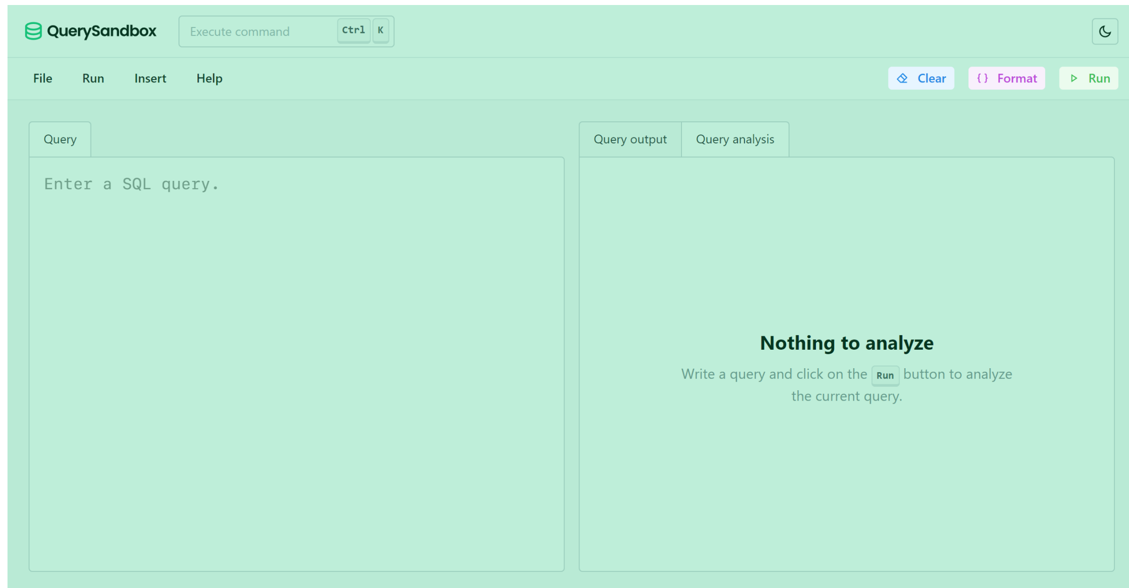


Figure 5.8: The action buttons which execute the *clear*, *format* and *run query* actions respectively.

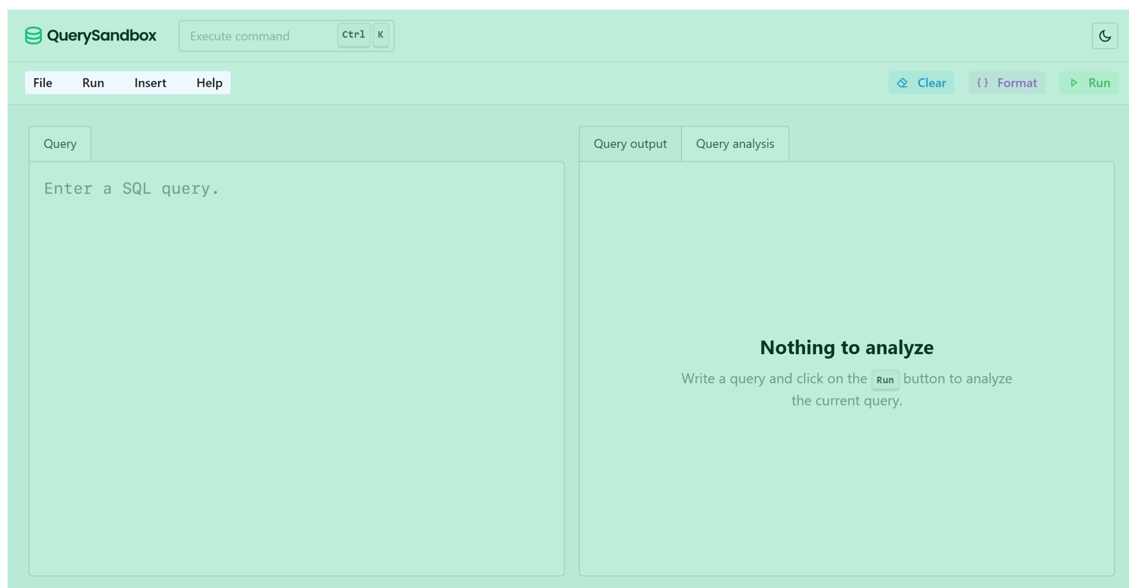


Figure 5.9: A toolbar containing common actions.

Database loading

Figure 5.10 – Implements S4
As introduced in the previous section, the user can load a database using the *file* dropdown menu in the application toolbar. A modal, comparable to a popup dialog displayed in the center of the screen, is used to allow the user to select a database to load. The modal shows the file name and database name, as well as the database size. The user also has the option t clear the current database. In that case, the database is removed from the application and the user is returned to the main view. We chose to let the user perform the loading and clearing of databases in a modal, because the focus is shifted to the modal and the user can only perform actions related to the database. We decided to use modals over popups because modals are usually not dismissed automatically and this way the user can still cancel the action.

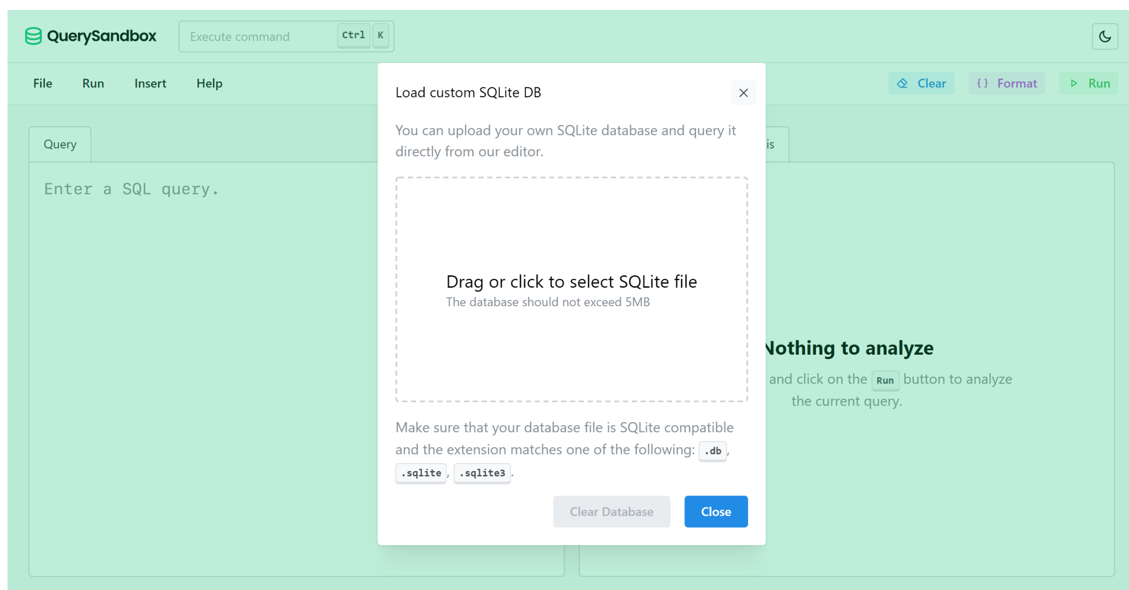


Figure 5.10: A modal allowing the user to load a SQLite database.

Command palette

Figure 5.11 – Additional feature
The command palette is a quick way to launch commands and keyboard shortcuts and is inspired by the Spotlight feature from Mac OS and the Command Palette from Visual Studio Code. The command palette in QuerySandbox is accessible using the `CTRL + K` shortcut, or by clicking the "Execute command" text input in the header next to the QuerySandbox logo. The rationale behind this feature is to have a fast way to access functionalities in QuerySandbox without having to memorize the respective keyboard shortcuts.

Dark mode

Figure 5.12 – C2
We wanted to provide the user with a dark mode version of QuerySandbox. To accomplish this, we added an explicit dark mode toggle button above the action buttons. If the user presses the button, the interface will automatically change to dark mode. If the user presses the button another time, it will turn back to the normal "light" mode.

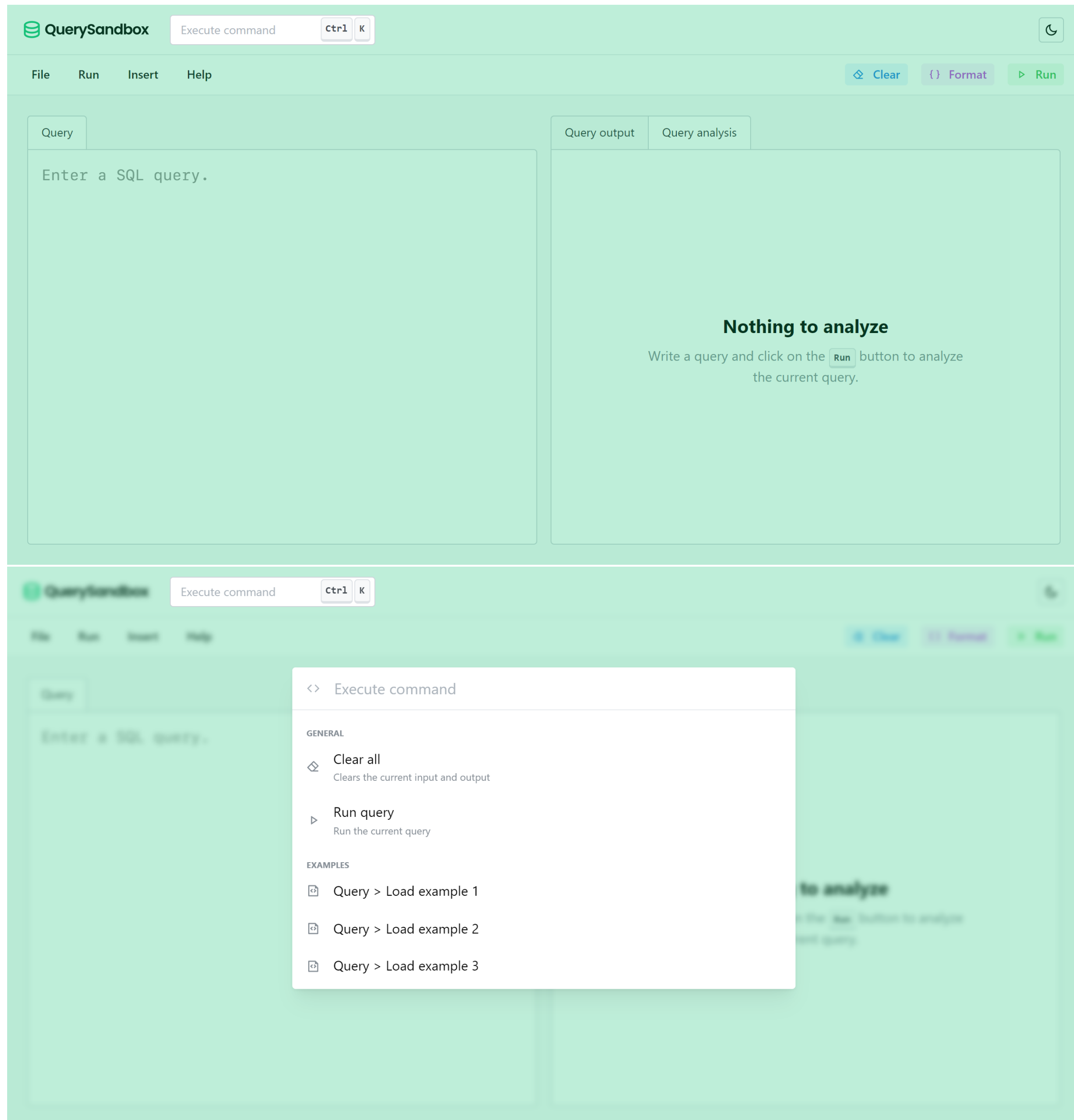


Figure 5.11: A command palette allowing the user to quickly execute command commands without the use of a mouse.

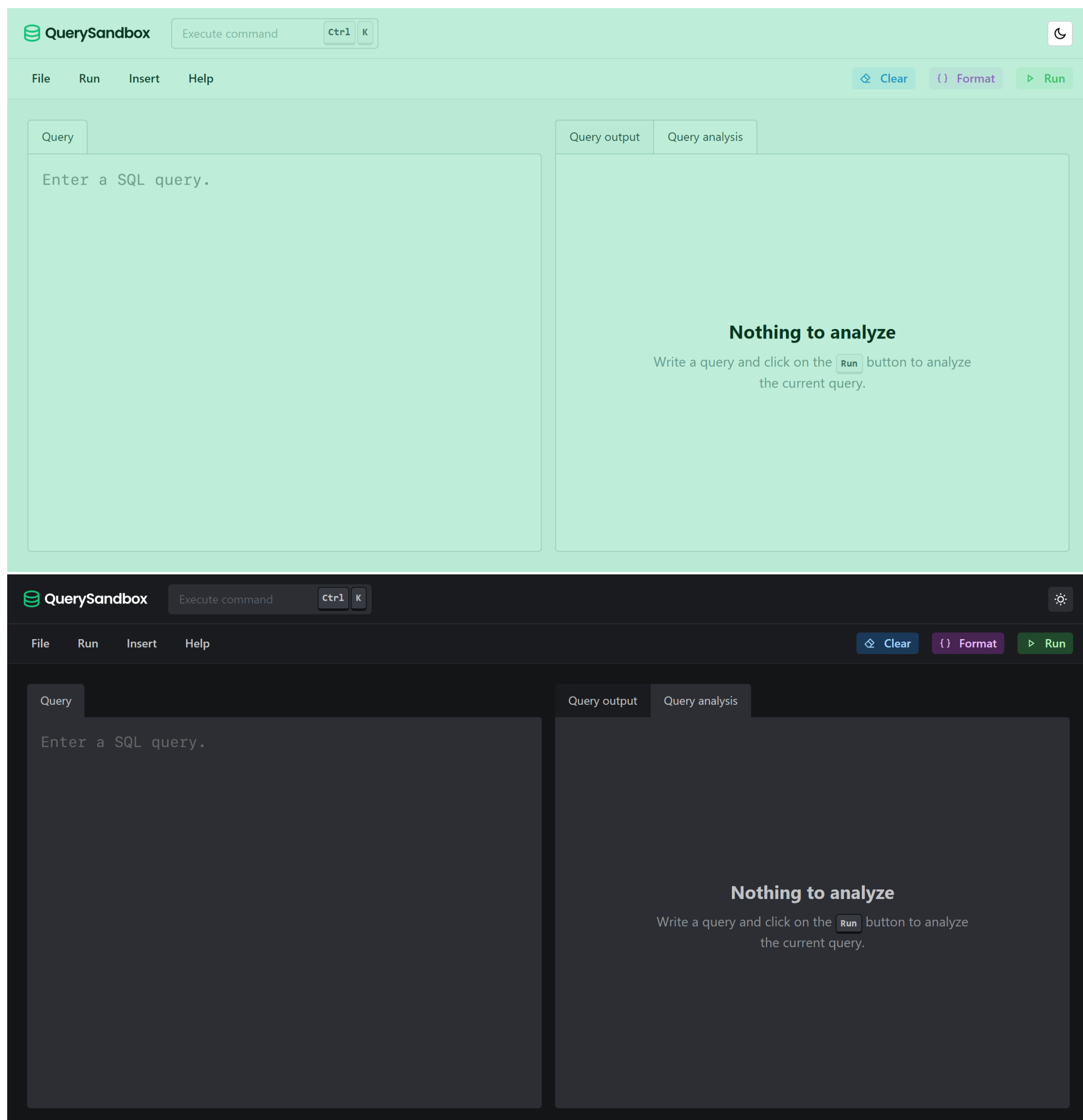


Figure 5.12: The dark mode toggle button and the interface of QuerySandbox in dark mode.

5.4 Conclusion to research question 1 (RQ1)

With the process described in chapter 4 as well as this chapter, we can answer research question 1 (RQ1). In order to design a tool to help novices understand anti-patterns, we first had to define the scope of the anti-pattern detector. We had to obtain a set of anti-patterns that could occur in raw SQL queries. Once this scope was defined and we had this set of anti-patterns, we looked into how to detect these various anti-patterns. The detection was done using regular expressions. Once we were able to detect the anti-patterns we had to think of a way of reaching the intended user, namely novices. We argued that a novice's experience can vary greatly. Therefore, we had to make sure our detector was easily accessible to all kinds of novices. We then explained our ways of distributing the detector, namely through a Python package with CLI, a RESTful API and a web application. We then looked into UI/UX principles, created a list of requirements for the application, created the architecture, designed the UI and implemented it using TypeScript and React.

5.5 Summary

In this chapter, we first gave a quick introduction to UI and UX design followed by some important principles and heuristics. We then introduced our requirements which were partly formed using design principles. Then we introduced the application architecture and the main features of QuerySandbox. We have not only discussed the features, but also the motivation behind the features and the usability aspects that were implemented in QuerySandbox. The development process of QuerySandbox, as well as the process described in chapter 4, was then used to answer research question 1 (RQ1).

Chapter 6

User study

In this chapter we will describe our user study and discuss the findings. We have conducted a user study with 4 students. In this study, we have given our participants a set of query formulation problems which had to be answered using our application. These problems were designed in such a way that they should trigger anti-patterns in query formulation for students that are unaware of them. The interface of QuerySandbox provides tools to solve the problem. The user study is divided into three parts: a pre-, main-, and post-test. We first measured their awareness of SQL anti-patterns through a pre-test in questionnaire form. Then, we had a session in which the learners work with QuerySandbox to practice with, and learn more about anti-patterns using the set of query formulation problems. Finally, in a post-test, we check whether they perform better on the exact same questions as the pre-test. Using the study's findings, we hope to answer the following research question:

> RQ2 Does the intervention of our tool help novice users learn, fix and prevent writing SQL queries with anti-patterns?

The pre-test is a questionnaire with basic demographic questions and a set of queries containing anti-patterns. The demographic questions were solely used to check whether the research population is balanced. The set of queries were used to see whether our participants can improve or "fix" queries without having any background knowledge on anti-patterns. We can use the answer obtained from the pre-test to check whether our participants are aware of anti-patterns. The questionnaire was made using Microsoft Forms which is provided by the TU/e through Office365.

In the main test, we gave our participants a set of query formulation problems. Students had to answer these problems using our developed, web-based application as described in detail in the previous section. The application includes a code editor, where the students can write queries and receive feedback on their queries in terms of query anti-patterns found. When the student opened the web-page their interactions with QuerySandbox were logged for research purposes. This includes interactions like button presses, shortcut usage, but also more general information like the input query, the output of the query when the query is run and the analysis output generated by SQLEyes. These interactions were stored in a database, together with a unique session identifier. This resulted in an event table, where each row in the table is an event record. The table contains a fixed set of attributes, among other the *SessionID*, which maps a participant to a record, the *Action* that occurred and the *Timestamp* it occurred on. Each event record in this table could be independently analyzed. However, to further analyze the interactions with our system, we obtain traces, which are a finite sequences of actions, and use process mining to analyze these traces. This gives us information about common, or frequently occurring interactions with QuerySandbox as well as very uncommon or infrequent interactions.

In the post-test, we asked the user the same questions as in the pre-test. We again asked our

participants to improve or "fix" these queries. This allows us to verify whether the interventions given by QuerySandbox during the main test, had a positive effect on our participants' ability to fix the anti-patterns in the queries. Finally, we asked our participant various questions to determine the system usability as well as general usage of our application. To determine usability, we are using the System Usability Scale. This scale can be used as a reliable, low-cost usability metric that allows for global assessment of a systems usability [7]. It includes ten questions rated on a five-point Likert scale that assess their experience with the system.

In this section we will first describe the participants, materials and procedures used to collect our data. Second, we present the results of the study. Finally, we conclude with a summary of the findings.

6.1 Participants

The study was undertaken with 4 students of various ages in the range of 20 to 23 years old. All our participants were novice SQL users who have had at least 2 years of programming experience with at least 6 months of SQL programming experience. They have a background in Computer Science and Engineering and have followed an introductory database course, meaning they have a basic understanding of SQL.

This user study was a paid study taking roughly 1 hour and 30 minutes for all parts combined. Participants were paid ten euros for participation to compensate for their time.

6.2 Materials

As this experiment involved a working prototype, it was undertaken on a computer (desktop or laptop) of the user's preference. The only requirement was that the Google Chrome browser be used during the study because the system runs in the browser. The interaction between the participants and the system in the main- & post-test was recorded via screen capture.

6.2.1 Pre-test materials

In the pre-test, the participants were asked to answer an online questionnaire containing a set of problem descriptions with each having a corresponding query. For each problem, we asked the participants if the corresponding query is correct and if it could be improved in any way. With the exception of the first problem, the problems were designed to contain anti-patterns. We have listed the questions, as well as possible answers below.

Question 1

A supermarket wants to analyze what kind of products their customers purchase. This information is spread out over three different tables, namely the customer table, the product table and the purchase table.

They use the following query:

```
SELECT cId, city, sId, date, quantity, price, pName
FROM customer
    JOIN purchase USING (cID)
    JOIN product USING (pID)
```

Is the query correct? If no, what is the correct solution? If yes, can you try to improve it?

Answer
The query is correct. Albeit not necessary for this study, it can be improved by

using natural joins.

Question 2

A social media start-up wants to perform sentiment analysis on posts that are placed by users. The post table, where all these posts are stored, contains roughly 30 columns. Luckily, they are only interested in a hand full of columns (text, timestamp, likes, dislikes, numberOfComments). They query all the columns from the posts table and later, during data pre-processing they remove the unused columns.

They use the following query:

```
SELECT *
FROM post
```

Is the query correct? If no, what is the correct solution? If yes, can you try to improve it?

Answer
The query is correct but is not very efficient. The post table has roughly 30 columns, and by using the wildcard operator, we query all 30 columns where most of theses columns will be discarded during data pre-processing. A better solution would be to input the columns needed in the select statement.

Question 3

A supermarket has made a custom app where customers can create an account and add products to their virtual shopping list. A table shoppinglist is used to store the product and the date it was added to the list. The supermarket wants to obtain for each customer in the shoppinglist table the customer ID, the customer's name, the product ID and the date of the first entry the customer added to its shopping list.

They use the following query:

```
SELECT cID, cName, pID,
    MAX(date)
FROM customer
    JOIN shoppinglist USING (cID)
GROUP BY cID, cName
```

Is the query correct? If no, what is the correct solution? If yes, can you try to improve it?

Answer
The query can display incorrect results since the single value rule is broken. For example, for the customer with ID of 2, it shows that the product with ID 4 is last added to the shoppinglist. However, if we inspect the full table of the shoppinglist joined with the customer table, we see that the product with ID 22 is last added.

Question 4

A lottery company has three different prices. They have written a Python script to select three participants from their database. All participants are stored in a table called participant. There are over 100 million participants.

In every iteration, they query below that selects a random participant ID from the participant table. It does this by using the `RAND()` function. It will first order all participants randomly, and then, using the `LIMIT 1`, it will only return a single participant per draw.

They use the following code:

```
for i in range(0,3):
    id = execute_sql("""
            SELECT pID
            FROM participant
            ORDER BY RAND()
            LIMIT 1"""
         )

# Handle notifications
...
```

Is the code correct? If no, what is the correct solution? If yes, can you try to improve it?

Answer
The query is easy to understand, making it a rather popular solution. However, the performance of this query is rather poor. By using the `RAND()` inside an `ORDER BY` clause, the use of an index is not possible, since there is no index containing the values returned by the random function. Developers should instead choose a random value using other means. Common ways would be to generate a random value between 1 and the greatest primary key, or counting the total number of rows and generating a random number between 0 and the row count.

Question 5

A company is sending out emails to their customers which are considered active. Customers are stored in a customer table inside a database. The customer's first name, middle name , last name and a status are stored in this table among others. However, not every customer has a middle name. In this case, the middle name is NULL. There are two email templates used by the company. One template where the customers middle name is specified and one where it is not. Therefore, the email process is done in two steps; first selecting the customers without middle name and sending an email with the second template, and secondly selecting the complement of the first set of customers, namely selecting the customers with middle name and sending an email using the fist template.

They use the following query:

```
SELECT fistName, middleName, lastName, email
FROM customer
WHERE status <> "inactive" AND middleName <> null
```

Is the query correct? If no, what is the correct solution? If yes, can you try to improve it?

Answer
This query is not correct. SQL considers `NULL` to be a special value, distinct from

zero, false, true, or an empty string. Therefore, it is not possible to test for `NULL` *values with standard comparison operators such as =, >=, <>, etc. Instead use* `IS NULL` *and* `IS NOT NULL`

Question 6

A local supermarket is developing a system to quickly find a product on its many, many shelfs. An employee can open the app and search for their product. After submitting their search, the system lists products that match. The products are stored in a product table, which does not only contain products the supermarket currently has in their assortment, but also products they used to have. Therefore, this table contains roughly 500,000 products.

They use the following query:

```sql
SELECT pName, isle, rank, shelfNumber
FROM product
WHERE pName LIKE "%ice%"
```

Is the query correct? If no, what is the correct solution? If yes, can you try to improve it?

Answer
The query above shows an example of how to search products that have the word "ice" in their product name. These queries should be avoided if the table size of products is large. Instead, one should opt to use a specialized search engine method some of which even come as standard with certain databases or DBMS's. The performance issue occurs because they cannot use a traditional index, as described in, they must scan every row of the specified tables.

6.2.2 Main test materials

The main test consisted of six different query formulation problems, where each problem tries to provoke our participants to use an anti-pattern. This was done, so we can see whether they are capable of writing anti-pattern free queries or when they were not they would learn about each anti-pattern. The participants were asked to solve these questions using QuerySandbox, for which we present an example solution in Table 6.1. Each query that the participants executed was automatically logged to a database.

6.2.3 Post-test materials

Finally, in the post-test, the participants were again presented with the same problems as in the pre-test (see subsection 6.2.1). For each problem, we again asked the participants if the corresponding query is correct and if it could be improved in any way. The idea is that after using QuerySandbox, our participants would start recognizing anti-patterns in the queries of these problems meaning they would be able to improve the query.

As part of the post-test, we ask our participants questions on the usability of our system. This includes a list of questions from the System Usability Scale (SUS) [7] which are answered using a 5-Point Likert Scale.. SUS produces a single number that represents a composite measure of the overall usability of the system under consideration. Next to the questions from SUS, we ask some more specific and open questions on QuerySandbox. All questions asked in the post-test are listed below.

[1]The query formulation asks to obtain the sum of all purchase quantities, not only those purchase quantities of purchases in cities other than their own. This is not in line with the suggested answer.

Question / **Possible answer**

We are interested in obtaining all customer IDs of customers that live in Tilburg and where the street name is equal to null. Are there any such customers? if so, enter the customer id(s).

```
SELECT cID
FROM customer
WHERE city = "Tilburg"
  AND street IS NULL
```

We are interested in obtaining all columns of purchases where the quantity is greater than 12. Note that the selected columns are basically the entire purchase table, except for the tID and suffix.

```
SELECT cID, cName, street, city,
    sID, pID, pID, date,
    quantity, price, pName
FROM customer
    JOIN purchase USING (cID)
    JOIN product USING (pID)
WHERE quantity > 12
```

We are interested in obtaining all street names ending with “plein” from customer table.

```
SELECT street
FROM customer
WHERE street LIKE "%plein"
```

We are interested in obtaining for each customer the total quantity of purchases done at each store. For example, John has made 5 purchases at the Coop, 2 Hoogvliet and 4 at the Lidl. Select the customer’s name, sum of the quantities called “total_purchases” and the store name.

```
SELECT cName, sum(quantity) as
   total_purchases, sName
FROM purchase
    JOIN store USING (sID)
    JOIN customer using (cID)
GROUP BY cName, sName
```

We are interested in obtaining the customer IDs and the sum of purchase quantities for customers who have made purchases in stores in cities other than their own. Furthermore, we want to only display those who have made more than 50 purchases in total[1].

```
SELECT cID, sum(quantity)
FROM purchase
JOIN store USING (sID)
JOIN customer using (cID)
WHERE store.city <> customer.city
GROUP BY cID
HAVING sum(quantity) > 50
```

A security audit company is interested in obtaining a random purchase entry from the purchase table to check for fraud. Write a query that selects a random transaction ID (tID) from the purchase table.

```
SELECT tId
FROM purchase
ORDER BY RANDOM() LIMIT 1
```

Table 6.1: The questions for the main test.

System Usability Scale questions

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

Other questions

1. I consider QuerySandbox to be an improvement over existing systems.
2. QuerySandbox is a tool I would use.
3. What do you think about the UI design of QuerySandbox.
4. If you could add additional features to QuerySandbox, that are currently not present, what would you add?
5. If you could remove features from QuerySandbox, that you consider superfluous, what would you remove?
6. Did QuerySandbox help raise your awareness of SQL anti-patterns? If yes, why? If no, why not?
7. Would you agree that QuerySandbox could be used in education to teach students about various errors and anti-patterns? Do you believe that learning using QuerySandbox would be ”better” than traditional methods (i.e. textbooks).

6.3 Procedure

Participants had to finish the pre-test before they were allowed to start with the main & post-test. The link to the pre-test questionnaire was sent to our participants via email. Once a participant completed the pre-test, we scheduled a meeting with them to complete the main and post-test. Important was that the pre-test did not contain our research goal nor any definition of what an anti-pattern is.

Then followed the main part of our study. Upon entering the meeting, the participant was welcomed and introduced to the experiment. We quickly went over their answers from the pre-test. When necessary, participants had to clarify their answer to certain questions. They were then given a questionnaire containing questions of both the main & the pre-test. However, before starting they had to sign a virtual informed consent form. We first introduced the system to the participants, by showing them the general interface as well as various functionalities. Then, the participants started by answering the main test questions. Important was that all their queries need to be written and ran in QuerySandbox so that the logging system would work. The usage of third-

party sources, like Google or StackOverflow, was not allowed. When a participant was unable to solve a question on their own, they could approach us for assistance, and we would point them in the right direction. Because syntax was not part of the study, if we noticed a participant struggling excessively with syntax, we would also assist them.

The post-test was used to determine whether the participants could fix or improve the same queries presented in the pre-test, but this time with knowledge they acquired from QuerySandbox. After the participants had completed the post-test questions, they had to answer some questions for the system usability. We moved on to a brief interview after they completed this. Participants were asked for their overall impression of the system and to provide additional feedback on QuerySandbox. Afterwards, we thanked the participants for their time and effort and ended the meeting.

6.4 Results

In this section we analyze both qualitative and quantitative data gathered during the user study. We start by looking at the results obtained in each part of the user study. We will then compare the answers obtained in the pre-test, with the answers obtained in the post test. We conclude the section with the results from the usability questionnaire and the interview.

6.4.1 Results pre-test

Completing the pre-test took 32 minutes on average. The answers to the set of pre-test problems, described in subsection 6.2.1, is summarized in Table 6.2. If we look at these answers, we can see that there are some anti-patterns our participants recognize without being aware of anti-patterns. For both the Implicit Columns as well as the Fear of the Unknown anti-pattern, most of our participants were able to see that there was a problem with the query and gave the correct reason.

Exercise 6 provided a query with the Poor Man's Search Engine anti-pattern. Only participant 2 knew that pattern matching was a bad practice in SQL, especially with large tables. The other participants were not aware of this.

We can also conclude that our participants are not familiar with the single value rule and using the `GROUP BY` clause. Except for participant 1, all participants stated that exercise 3 was correct, which was not the case. According to Participant 1, the query in exercise 3 was correct but could be improved by removing a column from the `GROUP BY` clause. He states that this would shorten the group by operation and would make the query more readable. This is of course not the right improvement, as now both `cName` and `pID` break the single value rule.

	Participant 1	**Participant 2**	**Participant 3**	**Participant 4**	**Anti-pattern name**
ex.1	0	1	0	1	None
ex.2	2	2	2	2	Implicit Columns
ex.3	1	0	0	0	Ambiguous Groups
ex.4	0	1	1	0	Random Selection
ex.5	2	2	2	0	Fear of the Unknown
ex.6	0	2	1	1	Poor Man's Search Engine

Table 6.2: The answers to the exercises of the participants in the pre-test.
0: "Query is correct and could not be improved"
1: "Query is wrong or could be improved, but given wrong reason"
2: "Query is wrong or could be improved, given good reason"
3: "Query is wrong or could be improved, given good reason and realized it was an anti-pattern"

6.4.2 Results main test

The main test took 37 minutes on average to complete. The system logged a total of 350 actions to our database. From these 350 actions, there are 102 rows where the action is *run query*, meaning

that the system analyzed a total of 102 queries during the test. Out of these 102 queries, 66 of them contained one or more anti-patterns. The total amount of anti-patterns found was 76.

Table 6.3 shows the occurrence of every anti-pattern in the test. The most common anti-pattern is the ambiguous groups anti-pattern with the the second most common anti-pattern being the "implicit columns" anti-pattern.

Anti-pattern name	Count
Ambiguous Groups	30
Fear of the Unknown	3
Implicit Columns	13
Poor Man's Search Engine	8
Random Selection	10
Spaghetti Query	12
Total count	76

Table 6.3: The number of occurrences of every anti-pattern.

Table 6.4 shows the number of queries executed as well as the number of queries containing one or more anti-patterns per participant. Whilst participant 1 has executed the most queries by a large margin, he does not have the most queries with anti-patterns. The most queries containing anti-patterns were executed by participant 4. This participant also executed the second most queries. Participant 2 executed the least amount of queries with also the least amount of queries containing anti-pattern(s). The occurrence of each anti-pattern for each participant is shown in Table 6.5.

Participant	No. Queries executed	No. Queries with anti-pattern(s)
Participant 1	31	17
Participant 2	22	13
Participant 3	23	17
Participant 4	26	19
Total count	102	66

Table 6.4: The number of queries executed along with the number of queries containing anti-patterns for every participant.

Anti-pattern name	Participant 1	Participant 2	Participant 3	Participant 4
Ambiguous Groups	6	3	11	10
Fear of the Unknown	0	2	0	1
Implicit Columns	5	2	4	2
Poor Man's Search Engine	3	2	1	2
Random Selection	2	4	1	3
Spaghetti Query	1	1	6	4
Total count	17	14	23	22

Table 6.5: The number of occurrences of every anti-pattern for every participant.

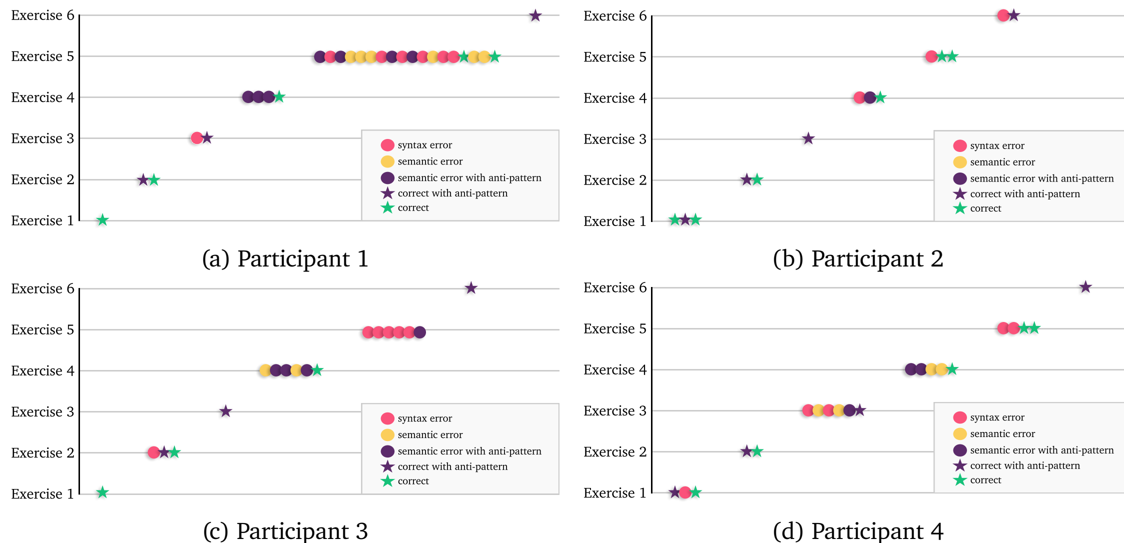
Figure 6.1 shows the execution order plots for all the participants. Interestingly, whilst the participants were explicitly told that they were allowed to do the exercises in any order they want, they all performed the exercises in the order that we designed. For each exercise, the plot shows if the query contained a syntax error, a semantic error, or if it was correct. Furthermore, it shows

when an executed query contained an anti-pattern or not, using the purple color. Sometimes, participants would run extra queries to explore the data or to refresh their SQL syntax knowledge. These extra queries could not be directly tight to a specific exercise so we excluded them from the plots.

From this figure, we can clearly see that exercise 4 and exercise 5 were the most challenging ones, taking a joint total of 49 executed queries. Especially participant 1 was struggling with exercise 5, as it took him 15 queries to reach the correct answer. However, he still was not certain if the query was correct or not, executing another 2 queries before he realized one previous answer was correct.

Another thing we noticed was that whenever an anti-pattern was detected and thus the participant read the anti-pattern description provided by QuerySandbox, the participant was able to get rid of the anti-pattern relatively quickly. It also shows that almost every final query executed by each participant for each query formulation problem was correct and was mostly in line with our own answers in Table 6.1. The only exception was participant 3, who did not execute the final query of exercise 5 but did, however, answer it correctly in the questionnaire.

One may notice that the final solutions to exercises 3 and 6 contain anti-patterns, as indicated by the purple star. This is due to the fact that these query formulation problems could not be solved without the use of an anti-pattern (Random Selection & Poor Man's Search Engine). The results of the post-test revealed that our participants remembered these anti-patterns slightly better than the other anti-patterns.



(a) Participant 1 (b) Participant 2

(c) Participant 3 (d) Participant 4

Figure 6.1: The execution order plots for every participant.

We can also examine which other actions are used among the 350 actions. Both the format and clear all actions were used five times. The toggle theme action was used 7 times. Interestingly, none of our participants liked our app's default light mode, as all participants immediately switched to dark mode. There were no keyboard shortcuts used during the main test. The command palette, on the other hand, was used twice. This is to be expected given that these features were added to improve the user experience for more advanced users. The example queries from the "insert" menu were used a total of six times, with the "Insert query to get columns" query being the most frequently used. Two participants used this query to quickly check which columns belong to a certain table.

6.4.3 Results post-test

The post-test took 15 minutes on average. When the participants saw the post-test question, they immediately knew that they should look for anti-patterns similar to those seen in the main test. The answers to the set of post-test problems is summarized in Table 6.2. We can clearly see that almost every exercise is correctly answered. Participants could identify and name the anti-pattern, as well as explain why the query was incorrect or improve-able. There is, however, one exception and that is exercise 5. Participants 1 and 3 did not notice the Fear of the Unknown anti-pattern, but did know why it was incorrect. This is because participants 1 and 3 did not come across this anti-pattern during the main test. In the main test, they answered exercise 1 correctly on their first try as seen in Figure 6.1, so QuerySandbox never had to intervene making them not aware of the Fear of the Unknown anti-pattern.

	Participant 1	Participant 2	Participant 3	Participant 4	Anti-pattern name
ex.1	0	0	0	0	None
ex.2	3	3	3	3	Implicit Columns
ex.3	3	3	3	3	Ambiguous Groups
ex.4	3	3	3	3	Random Selection
ex.5	2	3	2	3	Fear of the Unknown
ex.6	3	3	3	3	Poor Man's Search Engine

Table 6.6: The answers to the exercises of the participants in the post-test.
0: "Query is correct and could not be improved"
1: "Query is wrong or could be improved, but given wrong reason"
2: "Query is wrong or could be improved, given good reason"
3: "Query is wrong or could be improved, given good reason and realized it was an anti-pattern"

Comparing answers pre- and post-test

In this section we will compare the answers given in the pre-test and the post-test, to see whether the participants learned and improved their query formulation skills. The results are summarized in Table 6.7 which shows the answer to the pre-test as well as the answer to the post-test for every exercise and participant. We can see that the results of the participants are improved after using QuerySandbox. In the pre-test, we can see that our participants were not well aware of anti-patterns. In the post-test, we can see that our participants were well aware of anti-patterns and could name most of them. The only exception is exercise 5, the Fear of the Unknown anti-pattern, for which participants 1 and 3 did not give the correct answer. For both participant 1 and 3, this was because they did not come across this anti-pattern for exercise 1, so QuerySandbox never had to intervene making them not aware of this anti-pattern.

	Participant 1	Participant 2	Participant 3	Participant 4	Anti-pattern name
ex.1	0/0	1/0	0/0	1/0	None
ex.2	2/3	2/3	2/3	2/3	Implicit Columns
ex.3	1/3	0/3	0/3	0/3	Ambiguous Groups
ex.4	0/3	1/3	1/3	0/3	Random Selection
ex.5	2/2	2/3	2/2	0/3	Fear of the Unknown
ex.6	0/3	2/3	1/3	1/3	Poor Man's Search Engine

Table 6.7: The answers to the exercises of the participants from both pre- and post-test (pre-test/post-test).
0: "Query is correct and could not be improved"
1: "Query is wrong or could be improved, but given wrong reason"
2: "Query is wrong or could be improved, given good reason"
3: "Query is wrong or could be improved, given good reason and realized it was an anti-pattern"

In order to show that participants performed better after using QuerySandbox, we will simplify the results shown in Table 6.7 to two dimensions, namely *recognizes problem(s)* and *recognizes anti-pattern(s)*. The former dimension is a binary variable, which indicates whether a participant could recognize that a query contains a problem or not. The latter dimension is also a binary variable, which indicates whether a participant could recognize that a query contains an anti-pattern or not. The results are summarized in Figure 6.2.

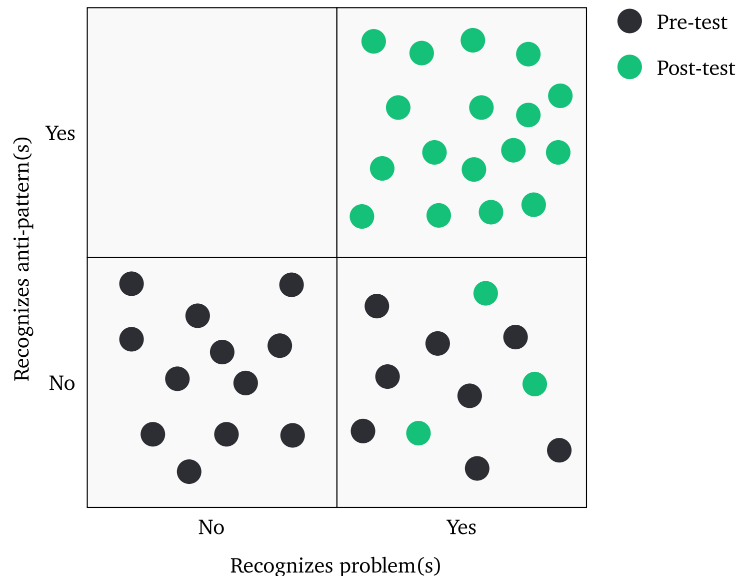


Figure 6.2: Confusion matrix showing the results of the pre-test and post-test.

Based on the confusion matrix, we can conclude that our participants had a general idea of what the problem was for some of the exercises in the pre-test. However, this was not the case for the vast majority of the problems. Following the main test we see that for the majority of problems, our participants could recognize the problem and even identify the anti-pattern to which it belonged. As a result, our preliminary findings suggest that QuerySandbox assists users in becoming aware of, and detecting, anti-patterns.

Results on system usability

Statistically confirming whether the intervention given by QuerySandbox helps novices learn, fix and prevent SQL anti-patterns is hard, especially with a small sample size. We believe it is also important to look at the behavior and thoughts of our participants during the study and perform a qualitative analysis on this as well. We thus had the participating students fill out a form with 10 usability questions and had them answer some open ended questions. Once those were filled in, we had a small conversation on their answers to these questions.

Table 6.8 shows the answers our participants gave to the SUS questionnaire as described in subsection 6.2.3. Participant 2 was the most positive on system usability, whilst participant 4 was most negative. Our system scored an average score of 87.5 which indicates that our participants found the overall usability of QuerySandbox to be good.

	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	SUS score
Participant 1	5	1	4	1	4	2	5	2	4	1	87.5
Participant 2	5	1	5	1	4	1	5	1	5	1	97.5
Participant 3	5	1	5	1	5	1	4	1	5	3	92.5
Participant 4	4	2	4	2	4	2	4	2	3	2	72.5
Average											87.5

Table 6.8: Answers to the System Usability Scale questions.
1: “Strongly disagree”
2: “Disagree”
3: “Neutral”
4: “Agree”
5: “Strongly agree”

Questions 9 and 10 are both two questions that we could improve on in terms of usability. These are the only questions that received a “neutral” rating. Participant 4 gave question 9 a neutral rating, since he did not feel very confident when using the system. This can be confirmed from his interaction with the system during the main test, where he was constantly looking the right buttons to press. He did however like the tool’s UI and considered it an improvement over existing systems like DB browser which he stated to be too complicated for running simple queries. When we asked the participant why he did not feel very confident, he reasoned that it might be because he does not use SQL and SQL/DB applications very often and is very unfamiliar with them in general. Participant 3 gave question 10 a neutral rating, since he felt that he needed to learn quite some things before he could get started with the system. While participant 3 states that the UI of QuerySandbox is minimalistic, clean and intuitive, he still felt that it was still not intuitive enough. Certain actions like loading in a SQLite database, or at first understanding the difference between query output and the query analysis tabs made him slightly confusing. He also found that it would be nice if QuerySandbox implemented a way to view the database’s schema once loaded in. All though QuerySandbox provides use full example queries that one could use to view the different tables, columns of tables and relations, he felt that this might not be enough. We did consider this feature when forming the requirements because it would potentially help novice users with their query formulation. However, we considered this feature to be out of scope because of time constraints. All in all, he felt that QuerySandbox should guide its users a little better.

The responses to the other questions on QuerySandbox were mostly positive. We will now go over the 7 final questions of the post-test and show what our participants answered. We will, however, not discuss their responses in detail. This will be covered in chapter 7.

Question 1
I consider QuerySandbox to be an improvement over existing systems.

Answers:
- **Participant 1**
  Completely agree. First of all, errors a clearly indicated. Moreover, there errors are explained whereas existing systems only indicate that an error has occurred. Finally, the QuerySandbox provides a fix for the error, which is normally only a general suggestion in existing systems.
- **Participant 2**
  I agree with the statement because I believe this tool would be of great help to people who are just starting with SQL.
- **Participant 3**
  The tool is not too intrusive and quite useful in understanding how to fix the code.
- **Participant 4**
  definitely.

Question 2
QuerySandbox is a tool I would use.

Answers:
- **Participant 1**
  Yes I would. The tool is easy to use in cooperation with SQL operations.
- **Participant 2**
  I strongly agree that I would use QuerySandbox because I would be able to quickly check my queries for anti-patters.
- **Participant 3**
  Yes, it would help me unlearn certain antipatterns and fix mistakes in my sql code.
- **Participant 4**
  yes, though I rarely use SQL, if I would use it again I would like to use the tool.

The first two questions test whether our participants would use QuerySandbox and if it can be considered as an improvement over existing systems. All of our participants said that it is an improvement over existing systems and that they would use it. Our participants said that this was mostly because of its clean interface, simplicity and fast loading times. This is consistent with our design goals which stated that it should be simple to use and should load very fast. When we asked what kind of other existing tools compete with QuerySandbox, our participants said DB browser [2], HeidiSQL[3] or other in-browser tools like playsql [4].

Question 3
What do you think about the UI design of QuerySandbox?

Answers:
- **Participant 1**
  Very elegant and professional appearance.
- **Participant 2**
  I enjoy QuerySandbox's UI design because it doesn't overwhelm the user with a lot of options/settings and has a Dark mode..
- **Participant 3**
  I like it - it is minimalistic, clean and intuitive.
- **Participant 4**
  Clean and easy to use.

Questions 3 tested whether our participants were happy with the current UI. They were very satisfied with the look and feel. In the conversations, some of them stated that it was very intuitive and professional looking. This was because of the clean and simple layout of the interface and the colouring scheme. The dark mode is an interesting improvement, since it made the interface less "busy". Another thing a participant pointed out was that QuerySandbox feels very familiar even when they were completely unfamiliar with it. They gave as example the menu bar and the two tabs in the center of the screen. They stated that this layout was often seen in similar tools and applications and that made it seem very intuitive.

Question 4
If you could add additional features to QuerySandbox, that are currently not present, what would you add?

Answers:

[2] https://sqlitebrowser.org/
[3] https://www.heidisql.com/
[4] http://yakovets.ca/

- **Participant 1**
  At the moment, I think the tool contains both all the necessary features and a number of futures the increase the ease of use of the tool.
- **Participant 2**
  I would add error highlighting in the query on the left side whenever there is an anti-pattern detected.
- **Participant 3**
  Intellisense if possible. A way to view the schema of the database. *(Verbally discussed in interview: A way to see query history, and to undo/redo it)*.
- **Participant 4**
  The fix for the Random Selection anti-pattern could state more clearly that the random selection should be done outside of the SQL code. The execute command option does not have the format option included.

Question 5
If you could remove features from QuerySandbox, that you consider superfluous, what would you remove?

Answers:
- **Participant 1**
  None, there are no redundant features.
- **Participant 2**
  I don't believe there are any superfluous features.
- **Participant 3**
  None.
- **Participant 4**
  Nothing.

Questions 4 and 5 tested whether our participants felt there were features that were necessary or unnecessary in QuerySandbox. All of them stated that they felt there were no superfluous features and that they saw purpose in all the currently implemented features. They did, however, identify several features that are currently missing and could potentially enhance the user experience. These features are error highlighting within the query tab, intelliSense (smart autocomplete) within the query tab, a way to view the database schema and a way to store and undo/redo queries from history.

Question 6
Did QuerySandbox help raise your awareness of SQL anti-patterns? If yes, why? If no, why not?

Answers:
- **Participant 1**
  Response not recorded.
- **Participant 2**
  QuerySandbox helped me raise my awareness because it brought to my attention how often I introduce anti-patterns to my query - something I wouldn't have realised without this tool.
- **Participant 3**
  Yes, it explains quite nicely why it is an issue and how to fix it.
- **Participant 4**
  Yes, I did not know about any antipatterns at all so I already learned quite a lot.

Question 7
Would you agree that QuerySandbox could be used in education to teach students about various errors and anti-patterns? Do you believe that learning using QuerySandbox would be "better" than traditional methods (i.e. textbooks).

Answers:

– **Participant 1**
Response not recorded.

– **Participant 2**
I believe that learning using QuerySandbox would be a far better alternative to learning SQL than traditional methods such as textbooks because the tool offers a more hands-on approach to students.

– **Participant 3**
Yes, QuerySandbox will help people who are new to sql and people who know how to use it but forget the syntax after long periods of time. I like the way it displays the query result and show errors on a different pane.

– **Participant 4**
Yes, the SQL lessons that I did have did teach me quite a few bad practices for writing SQL. When learning about antipatterns from the start can also help prevent getting used to faulty methods.

Lastly, question 6 asked participants whether QuerySandbox had helped them raise their awareness of anti-patterns. All participants who answered the question stated that they had indeed done so. This was an expected result as QuerySandbox is developed with the aim of making novices more aware of anti-patterns. Participants were asked how the tool helped them raise their awareness. One of the participants stated that they had been quite unaware of SQL anti-patterns and that SQL lessons taught him quite a few bad practices. QuerySandbox assisted him in becoming more aware of his errors, as well as in solving the error by providing useful information about the error. Question 7 tested whether participants agreed with the idea that QuerySandbox could be used to teach SQL to novice learners and SQL anti-patterns. They agreed that QuerySandbox could be a more hands-on approach to teaching SQL than traditional methods and showed a willingness to learn about SQL anti-patterns. All of them thought the tool would be useful in such a setting because it will be easier for students to learn SQL through QuerySandbox given its query analysis. They felt that learning how to write clean and efficient SQL can be very confusing and that this tool could help students learn SQL more efficiently. Furthermore, some participants suggested that QuerySandbox should also focus on other types of common errors within SQL, like syntax and semantic errors.

From our experience working with the participants during the study, we can see that we managed to positively influence their learning process. For instance, most participants were actively trying to understand the anti-patterns hidden in their queries. When the analysis presented them with an anti-pattern definition, they would go and read the provided definition or use the provided example queries to help find out where the anti-pattern resided in their own query. After a couple of attempts, most of them either understood the query or realized it was not something they did with SQL.

6.5 Conclusion to research question 2 (RQ2)

In terms of the research question at hand (RQ2), which asked whether our system helped participants learn, fix and prevent SQL queries with anti-patterns, we can see that this question was answered with a positive result. Based on the pre- and post-test results, we can see that there was an increase in the number of correctly answered queries, meaning that the participants learned to identify and fix common anti-patterns. Based on the results of the main test, we can see that the

participants were also able to learn about anti-patterns in SQL queries. In addition, the qualitative results indicate that participants were satisfied with the level of assistance provided by QuerySandbox. Overall, these results demonstrate that QuerySandbox helped participants learn, fix and prevent SQL queries with anti-patterns.

6.6 Summary

In this section, we introduced our user study conducted with 4 student participants. The study was setup in 3 parts: a pre-test, main test and post-test. We will now shortly summarize our results and conclude if our research question is answered.

Based on the pre-test results, we obtained 7 correct answers from various participants on why a query is incorrect or could be improved (result from Table 6.2). Following the main test, this number increased to 20 correct answers in the post-test, with almost all participants being able to identify the anti-patterns (result from Table 6.6). According to the main test results, 66 of the 102 executed queries contained at least one anti-pattern. The Ambiguous Groups anti-pattern appeared the most, with a total of 30 occurrences. With only 3 occurrences, the Fear of the Unknown anti-pattern was the least common. When we looked at the execution order plots, we see that the questions that try to provoke the Ambiguous Groups anti-pattern where considered the most challenging ones, taking a joint total of 49 executed queries. From the same plot, we can also see that the question that tries to provoke the Fear of the Unknown anti-pattern was considered very simple, taking a joint total of 9 executed queries. Another thing we noticed was that whenever an anti-pattern was detected and thus the participant read the anti-pattern description provided by QuerySandbox, the participant was able to get rid of the anti-pattern relatively quickly.

Based on qualitative analysis we concluded that the participants were positive about the QuerySandbox's assistance and reported that our system helped them learn about anti-patterns in a fast and efficient way. They stated that QuerySandbox provided a suitable level of assistance and that by using QuerySandbox they were able to understand and fix common anti-patterns within SQL queries which they had previously been unable to understand. QuerySandbox also got rated highly on the System Usability Scale. The responses to the other usability questions were mostly positive as well.

The research question (RQ2) was answered positively. Based on the pre- and post-test findings, we can observe that the number of correctly answered queries increased, indicating that the participants learned to recognize and rectify common anti-patterns. According to the findings of the main test, participants were also able to learn about anti-patterns in SQL queries. Furthermore, the qualitative findings show that participants were pleased with the degree of support offered by QuerySandbox.

Chapter 7

Discussion

In this chapter we will first discuss our key findings as well as their implications. Secondly, we will go over the limitations of our study which could be seen as threats to validity. We will discuss various limitations and their severeness with regard to the findings of our study. Zooming in on the details of our study and the validity, we will provide suggestions for further improvement of our research which we will touch upon in the final section of the proceeding chapter, which is about future work.

7.1 Discussion on findings

During this research project, we have obtained a number of findings. The most interesting findings come from the user study. Out of all our findings, we will list the key findings of the user study and discuss their implications.

7.1.1 Participants were unaware of SQL anti-patterns

The pre-test tested whether participants were aware of SQL anti-patterns. Out of the five anti-pattern exercises, participants were only aware of the drawbacks of the Implicit Columns anti-pattern query, namely that querying too many unused columns has an impact on performance and efficiency. They were unaware of the remaining anti-patterns. This might be due to the fact that some of these anti-patterns, namely the Random Selection and the Poor Man's Search Engine, are considered normal practices. The disadvantages of employing these bad patterns are rarely discussed in SQL literature or tutorials. In fact, the majority of the literature encourages the reader to employ these patterns. Even though the effects of these anti-patterns are most of the time negligible, we still believe that novices must be educated on the potential problems that might arise from their use, and possible alternatives to mitigate these problems. When we discussed the anti-patterns with our participants during the post-test, they stated that they were indeed unaware of the downsides of these anti-patterns.

Though not by these names, the Implicit Columns and Fear of the Unknown anti-patterns are more well-known. It is commonly taught to novice developers that querying too much data is detrimental to performance. Our participants stated that even though they were unaware of the anti-pattern, they were aware that data should only be selected as needed. The Fear of the Unknown anti-pattern was, contrary to the literature [6], a very obvious anti-pattern for our participants. Our participants were well taught how `NULL` comparisons are made in SQL. In the pre-test, three out of the four participants identified the issue with the query from exercise 5 and knew how to improve it. This is a good indication that both these anti-patterns are indeed well-known, even though participants were not aware of the name of the anti-pattern.

7.1.2 QuerySandbox raised awareness of SQL anti-patterns

From the results of the post-test, we can conclude that QuerySandbox helped raise awareness of a certain set of SQL anti-patterns. We can see that the results obtained by our participants are improved after using QuerySandbox. In the post-test, participants were well aware of anti-patterns and could name most of them.

We should, however, soften our conclusion slightly. Although QuerySandbox helped raise awareness of SQL anti-patterns during the user study, we cannot be sure if our participants will remember this information for an extended period of time. This is because the post-test was immediately after the main test which means that the anti-patterns from the main test were still fresh in their minds and there was no significant time for forgetting to take place. This means that there is potential for a forgetting curve to be at play here over the long term. The short period of time between the main test and the post-test could be seen as a limitation to our research, hence in section 7.2 we will discuss this topic further.

We are also unaware if this finding was because of QuerySandbox. Maybe a book on anti-patterns would have worked just as well. When we asked this question to our participants, all of them stated that because QuerySandbox provided a low barrier to writing, running, and analyzing queries it made it much easier for them to write and iterate on queries. This meant that they had a better understanding of how anti-patterns worked because they were able to see their effects of them firsthand. So although we cannot compare the effects QuerySandbox had on raising awareness of anti-patterns to that of a book, we can state that QuerySandbox had a positive effect in general. The latter comparison might be interesting to research in future work, hence we will touch on this topic again in section 8.1.

7.1.3 Participants found aggregation difficult

The Ambiguous Groups anti-pattern occurs when developers use the `GROUP BY` clause incorrectly as explained in section 2.5.2. This could lead to incorrect results. We know that every column in a query's `SELECT` clause must have a single value row per row group. If this requirement is not satisfied, then we say that there is an ambiguous group.

In the user study, we saw that our participants had difficulties with aggregation which introduced the Ambiguous Groups anti-pattern to their queries. In the pre-test, we saw that none of our participants could correct the query containing the Ambiguous Groups anti-pattern. The fix was very simple, the participants only had to add the product ID to the `GROUP BY` clause. Looking at the main test results, we can see that 30 of the total occurrences of anti-patterns (76) were the Ambiguous Groups anti-pattern. In exercises 4 and 5, where we attempted to trigger the Ambiguous Groups anti-pattern during query formulation, we can see that it took our participants a total of 49 query executions to complete the exercises without triggering the anti-pattern and whilst obtaining the right result-set. We can see from inspecting intermediate queries that contain the Ambiguous Groups anti-pattern that our participants had no idea what the single value rule was. We can also see that exercises 4 and 5 were mostly solved by trial and error.

Interestingly, our findings confirm what was found in the literature. Various works describe errors regarding the `GROUP BY` clause, which can be classified as aggregation errors. The GROUP BY clause appears several times in a study by Brass and Goldberg [6]. In another study by Ahadi et al. [4] it was found that one of the top, most frequent errors were grouping errors. In another study by Taipalus, the omitted grouping column was one of the most common types of error. In a study on identifying SQL misconceptions [20], the `GROUP BY` error is also mentioned a lot. In this study, it was, among other things, shown that many students find aggregation a confusing topic. Out of the works mentioned above, we conclude that aggregation in SQL is considered an advanced topic that most students find very difficult. Our findings support the existing body of knowledge, and show that the Ambiguous Groups anti-pattern is a very common error that SQL novices make. We believe that this anti-pattern occurs often because our participants lacked foundational knowledge about what aggregation actually meant and how the `GROUP BY` clause works. We propose that the way aggregation is taught be revised, and that this anti-pattern is explicitly taught in SQL courses.

However, we should avoid jumping to conclusions. We must also consider the difficulty of exercises 4 and 5, and possibly conclude that these exercises were overly difficult. The difficulty of these exercises could perhaps be explained because solving the questions required the use of multiple `JOIN` clauses. The clauses are shown to be error prone [39], and this was also confirmed looking at the screen recording of our participants. Our participants needed a lot of time to figure out how they should join 3 tables. Therefore, our participants might have known the basic of the `GROUP BY` clause but because of the difficulty of the question, which required several `JOIN`s, they might have answered incorrectly due to working memory overload.

7.1.4 Participants found QuerySandbox usable

We discovered that participants found QuerySandbox usable through the system usability questions and the interview. Our participants stated that they found QuerySandbox very intuitive and easy to use. They also stated that QuerySandbox is an improvement over existing features and that it was a tool they would use. However, in the post-test, when asked if there are features still missing from QuerySandbox, some of the participants listed some features that they would have loved to see in QuerySandbox.

The first suggestion was to add error highlighting to the query tab. The participants mentioned that they were not able to know that they had an error until they ran the query. They mentioned it would be beneficial if they could see an indication on the query tab that there is an error and where the error occurs.

The second suggestion was to add IntelliSense. IntelliSense is a code completion feature that suggests a list of valid statements or expressions to the developer as he is typing. They argue that this would help the QuerySandbox users to be more productive as they do not have to remember all the available columns, tables, and functions.

The third suggestion was to add a way to view the database schema if a database is loaded. They would like to see the list of all tables, columns, views, and any other objects that are part of the database. They mentioned that this would help them to know what they can query against and see the available columns. They state that this feature, together with the database schema viewing feature, would help them to understand the data that is available in the database and in turn make QuerySandbox a more complete tool.

The fourth suggestion was to add query history. This would be useful so that users can keep track of the queries that they wrote. This would help users to keep track of the queries that were executed and the ones that were not. It would also help users to be more productive as they would not have to re-type queries that they have already written. They gave as an example that question 5 from the main test was similar to question 4. However, since there was no query history, they had to re-type the query.

All features were discussed during the initial project requirement phase. However, due to this project being a time-sensitive project, these features were considered “Would have” features. We do, however, fully agree that these features will enhance the user experience and that they should be added to future versions of QuerySandbox. Especially the query history feature, with undo/redo buttons is a very important, still missing feature since various works [27, 35] argue that it is important that the user has the ability to reverse its actions. Secondly, a database viewing option would be a good feature. These schemas can give a quick overview of the various relations between tables, which help the user that might be able to help users in creating complex `JOIN`s, like the ones from exercises 4 and 5 of the main test. We hope to add these features to future versions of QuerySandbox and would consider it as future work.

7.2 Limitations

During our study, we did also identify some shortcomings to our research. We will now discuss these limitations briefly.

The user study had a limited sample size due to the difficulty of recruiting users. We see this as a major limitation. We started recruiting during a busy time and were unable to get enough users to join the study. This could be because we started recruiting near the end of a quartile, near students' final exams. Sadly, the decision to only start recruiting students near the exam period was not ours to make. Before we could begin our research, we had to go through a series of procedures to obtain clearance for the study. This meant that we had to rely on various people and organizations within the TU/e to grant us authorization to begin recruiting users. This created a substantial delay in our study, resulting in a small number of participants. We believe that this limitation, the small sample size, affects the study's external validity. That is, the results of the study may not be generalizable to the population of all students. This is a significant limitation and one that we should address in possible future studies. For the results obtained in the user study, we believe we have found the right balance between qualitative and quantitative data. For the quantitative data, we tried to not over-interpret or overgeneralize the data and present the results as they are. This is evident in the results section of chapter 6, where we try to stick to numbers and the facts, and not make any wild claims. We believe that the results of the user study are valid from a qualitative perspective and that we have gained valuable insights into the usability and the effects of the tool.

The fact that we only recruited from one university may have had an impact on the results. Since students from the same university are more likely to have had similar lectures on databases and might have obtained similar habits, we may have recruited users who are too similar to each other. This could have resulted in the study not having enough variation among the participants. However, since most of the participants were still novices, we believe that they are still different enough to obtain valuable insights. Furthermore, we believe that for our research questions, the institutions where the participants studied are irrelevant. The results show that the participants have little experience with SQL, and so we believe that the results of the study are generalizable to any novice SQL user.

Another limitation is the time between the main and the post-test. Because the post-test was immediately after the main test, there is potential that participants were merely reciting anti-patterns they had learned in the main test, rather than remembering and understanding them. However, we believe that since we are not testing the participants' memory, but their understanding and awareness of anti-patterns, the time between the main and the post-test is not a huge issue.

A final limitation of this study is the implementation of determining the query complexity. As discussed in chapter 4, in order to detect the Spaghetti Query anti-pattern, we must determine the query complexity. Our current approach is to compute the Halstead Complexity which uses the number of operands and operators. However, there are many approaches to measuring the complexity of a program. The approach we chose is not necessarily the best or most accurate one. During our testing, we obtained mixed results where the detector would sometimes flag queries as false positives and negatives for the Spaghetti Query anti-pattern. The issue is that query complexity is a relatively unexplored field. There is no clearly defined optimal way to measure query complexity. This is one of the areas where the tool could be improved. Our tests show that the current approach of using the Halstead Complexity is perhaps not the best and most robust one, and we would welcome any feedback on this topic.

7.3 Summary

We started this section by discussing our key findings, namely that participants were unaware of SQL anti-patterns, that QuerySandbox helped raise awareness of SQL anti-patterns, that participants found aggregation difficult, and that participants found QuerySandbox usable. We have also discussed the implications of our findings, and discussed how our findings confirm what is found in the literature on SQL anti-patterns, as well as how our findings on aggregation align with the existing body of knowledge. Next, we went over the limitations of our study. These limitations were mostly related to the small sample size and lack of variation among the participants.

Chapter 8

Conclusion

In this thesis, we addressed the following two main research questions:

> RQ1 How can a tool be designed to help novice users understand anti-patterns?
> RQ2 Does the intervention of our tool help novice users learn, fix and prevent writing SQL queries with anti-patterns?

We started our thesis with chapter 2. We began this chapter by going over some fundamental database concepts like table structure and table relations. Following that, we examined database languages, with a particular emphasis on SQL. We talked about SQL's history and some of its key language elements. We also studied the SQL syntax for common tasks such as filtering, ordering, and table joining. Then, as an alternative to SQL, we discussed ORMs and how they relate to SQL. Finally, we talked about regular expressions in general and how to use them. This was important since our anti-pattern detector was partially based on regex matching. For the rest of the chapter, we looked at various anti-patterns found in various works of literature. We provided a description, an example, and, where possible, a solution for each anti-pattern.

In chapter 3, we looked at related work in three categories: SQL learning barriers, SQL anti-patterns, and SQL anti-pattern detectors. The first category presented us with various barriers to learning SQL that would often be the main cause for users to write erroneous queries. The next category shows that this can often lead to anti-patterns that harm software quality. The final category showed us other works that addressed the problem by creating tools to automatically detect anti-patterns.

We started developing our own anti-pattern detector in chapter 4. We first had to determine the scope of our detector. Since there also exist many anti-patterns that can only be detected when the query context, like code that uses the query, is available, we had to limit our scope to anti-patterns that are easily detectable by studying the query alone. We answered the question *What SQL anti-pattern(s) can occur in raw queries written by novices?* by defining a list of anti-patterns that could be detected in raw queries. We then went over each anti-pattern from this list and explained our detection method. Finally, we discussed the detector's three distribution methods: a Python package with a CLI, a RESTful API, and a web application. The latter of which was discussed in a later section.

In chapter 5 we presented our web application called QuerySandbox. Before we started implementing our application, we first had to do some research on UI and UX design, create requirements, create mock-ups, and think of a useful architecture. Then we explained how we further developed our application and what features it includes.

With the process described in the previous two sections, we have successfully answered research question 1 (RQ1). This process describes how a tool can be designed to help novice users understand anti-patterns that can occur in SQL queries.

In chapter 6, we conducted a user study to examine if our tool was effective in helping novice users learn, fix, and prevent anti-patterns. The results of the study were used to answer the second research question. The user study was conducted with 4 participants and contained both qualitative as well as quantitative questions. The study was setup in 3 parts: a pre-test, main test and post-test. We first measured their awareness of SQL anti-patterns through a pre-test in questionnaire form. Then, we had a session in which the learners work with QuerySandbox to practice with, and learn more about anti-patterns using the set of query formulation problems. Finally, in a post-test, we check whether they perform better on the exact same questions as the pre-test. The results show that our participants were not aware of anti-patterns at the start of the study. Based on the pre-test results, we obtained 7 correct answers from various participants on why a query is incorrect or could be improved. Following the main test, this number increased to 20 correct answers in the post-test, with almost all participants being able to identify the anti-patterns. The main test results show that our participants wrote a large portion of queries that contained at least one anti-pattern, with the most occurring anti-pattern being the Ambiguous Groups anti-pattern which participants found very difficult. However, after the anti-patterns were detected by QuerySandbox and the participant read the anti-pattern description provided in the query analysis tab, the participant was able to get rid of the anti-pattern relatively quickly. When we compared the pre- and post-test results, we saw that there was an increase in the number of correctly answered queries, meaning that the participants learned to identify and fix common anti-patterns giving. In qualitative analysis, participants are positive about QuerySandbox's assistance and reported that the system helped them learn about anti-patterns in a fast and efficient way. Their feedback stated that QuerySandbox provided a suitable level of assistance and that by using QuerySandbox they were able to understand and fix common anti-patterns within SQL queries that they had previously been unable to understand. QuerySandbox also got rated highly on the System Usability Scale. The responses to the other usability questions were mostly positive as well. The answer to the second research question can therefore be seen as affirmative: QuerySandbox is an effective tool for novice users to learn, fix, and prevent writing SQL queries that contain anti-patterns.

In the last chapter, chapter 7 we discussed our key findings. We organized the discussion into two parts. First, we went over our key findings and the implications of these findings. Second, we discussed about the limitations for this paper.

Taking a step back to our main questions, we can conclude that we answered both questions successfully. We also believed we have contributed to the field of anti-patterns as well as SQL and database education. Our tool allows SQL anti-patterns to become more tangible, learnable, and relevant. Though, we must admit that our work is not free of flaws. We had to accept some limitations with our study, that have opened up some directions for future work.

8.1 Future work

During this research project, we have focused on creating a tool to help increase awareness of anti-patterns and to help novices with writing SQL queries. We believe that the tool can help in the educational process by raising awareness of SQL anti-patterns. This can help educate novices on how to avoid anti-patterns and write better SQL queries. However, the tool can also be used for other purposes. Since the tool can be easily extended with custom detectors, we can use the tool for other purposes such as detecting various syntactic as well as semantic errors.

Another possibility for future work would be to perform another user study that measures how much better QuerySandbox would be versus traditional methods, like books. This study would consist of two groups of participants, namely, a group that uses QuerySandbox and a group that uses a book on SQL anti-patterns. The study would need to be carefully designed so that the participants in each group are equally matched. The study would measure the performance of each group in two different tasks: 1) fixing queries containing anti-patterns, and 2) formulating new queries that do not contain any anti-patterns. This study can help verify the effectiveness of QuerySandbox in the context of SQL education.

Lastly, we encourage future work in the field of query complexity. query complexity is a relatively unexplored field. The tool has shown that there are many ways to measure query complexity, and we believe that this is the area where the tool can be improved the most. If query complexity was better understood, we could improve the tool's detectors and make them more accurate and robust.

We do, however, not plan to actively continue developing this tool. As our research project was anchored to a thesis, we outgrew the project. We can no longer add new features to the tool and we only perform minor maintenance tasks on it. Therefore, any future work on the tool will have to be done by a third party. We believe this tool has potential and it could help give some direction to future work on the tool.

Bibliography

[1] ISO - ISO/IEC JTC 1/SC 32 - Data management and interchange. Available at https://www.iso.org/committee/45342/x/catalogue/p/1/u/0/w/0/d/0. 10

[2] SQLator - An online SQL learning workbench. pages 223–227, 2004. iii, 2, 29, 33

[3] Albrecht A. sqlparse. Available at https://pypi.org/project/sqlparse/. 35

[4] Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. Students' semantic mistakes in writing seven different types of sql queries. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, page 272–277, New York, NY, USA, 2016. Association for Computing Machinery. iii, 2, 30, 31, 33, 79

[5] Natalia Arzamasova, Martin Schäler, and Klemens Böhm. Cleaning antipatterns in an sql query log. *IEEE Transactions on Knowledge and Data Engineering*, 30(3):421–434, 2018. iii, 32, 33

[6] Stefan Brass and C. Goldberg. Semantic errors in sql queries: A quite complete list. pages 250–257, 10 2004. 29, 78, 79

[7] John Brooke. *"SUS-A quick and dirty usability scale." Usability evaluation in industry*. CRC Press, June 1996. ISBN: 9780748404605. 62, 65

[8] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, page 249–264, New York, NY, USA, 1974. Association for Computing Machinery. 10

[9] Dai Clegg and Richard Barker. *Case Method Fast-Track: A Rad Approach*. Addison-Wesley Longman Publishing Co., Inc., USA, 1994. 47, 48

[10] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970. 5, 10

[11] Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. SQLCheck: Automated detection and diagnosis of SQL anti-patterns. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, jun 2020. iii, 33

[12] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, Beijing, 3 edition, 2006. 15

[13] Peter M. D. Gray. *OQL*, pages 2003–2004. Springer US, Boston, MA, 2009. 9

[14] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977. 40

[15] Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf, 1st edition, 2010. iii, 18, 31, 32, 33

[16] Andrew Koenig. Patterns and antipatterns. *J. Object Oriented Program.*, 8(1):46–48, 1995. iii, 16, 31

[17] Yingjun Lyu, Ali Alotaibi, and William G. J. Halfond. Quantifying the performance impact of sql antipatterns on mobile applications. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 53–64, 2019. iii, 18, 31, 32, 33

[18] Yingjun Lyu, Sasha Volokh, William G. J. Halfond, and Omer Tripp. Sand: A static analysis approach for detecting sql antipatterns. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 270–282, New York, NY, USA, 2021. Association for Computing Machinery. iii, 32, 33

[19] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976. 39

[20] Daphne Miedema, Efthimia Aivaloglou, and George Fletcher. *Identifying SQL Misconceptions of Novices: Findings from a Think-Aloud Study*, volume 1. Association for Computing Machinery, 2021. iii, 2, 30, 33, 79

[21] Daphne Miedema, George Fletcher, and Efthimia Aivaloglou. So many brackets! an analysis of how sql learners (mis)manage complexity during query formulation. In *2021 IEEE/ACM 29th International Conference on Program Comprehension*, 2022. 30

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. 33

[23] Antonija Mitrovic. Learning SQL with a computerized tutor. In *ACM SIGCSE Bulletin*, pages 307–311, 1998. iii, 2, 29, 33

[24] Biruk Asmare Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. On the prevalence, impact, and evolution of sql code smells in data-intensive systems. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 327–338, New York, NY, USA, 2020. Association for Computing Machinery. iii, 18, 31, 32, 33

[25] Csaba Nagy and Anthony Cleve. A static code smell detector for sql queries embedded in java code. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 147–152, 2017. iii, 32, 33

[26] Jakob Nielsen. 10 usability heuristics for user interface design, Apr 1994. 48, 55

[27] D. Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, 2013. 47, 54, 80

[28] William C. Ogden and Susan R. Brooks. Query languages for the casual user. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems - CHI '83*, pages 161–165, New York, New York, USA, 1983. ACM Press. iii, 2, 29, 33

[29] Oracle. What Is a Database. Available at `https://www.oracle.com/database/what-is-database/`. 5

[30] Abdou Ousmane and Hongwei Xie. Detecting anti-patterns in sql queries using text classification techniques. *International Journal of Advanced Engineering Research and Science*, 6:305–309, 01 2019. iii, 33

[31] Stack Overflow. Stack Overflow developer survey 2021. Available at `https://insights.stackoverflow.com/survey/2021/`. 1, 9, 10

[32] Kai Presler-Marshall, Sarah Heckman, and Kathryn Stolee. Sqlrepair: Identifying and repairing mistakes in student-authored sql queries. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 199–210, 2021. iii, 2, 33

[33] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, January 2008. `http://www.w3.org/TR/rdf-sparql-query/`. 9

[34] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.*, 18(1), oct 2017. 30

[35] Phyllis Reisner. Human Factors Studies of Database Query Languages: A Survey and Assessment. *ACM Computing Surveys*, 13(1):13–31, jan 1981. iii, 2, 29, 30, 33, 80

[36] ScaleGrid. 2019 Database Trends. Available at `https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/` 5

[37] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3:30–36, 04 1988. 40

[38] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts*. McGraw-Hill, New York, 6 edition, 2010. 5

[39] John B. Smelcer. User errors in the use of the structured query language (sql). *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 37(4):382–386, 1993. 29, 30, 80

[40] Amjed Tahir, Aiko Yamashita, Sherlock Licorish, Jens Dietrich, and Steve Counsell. Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, EASE'18, page 68–78, New York, NY, USA, 2018. Association for Computing Machinery. 33

[41] Toni Taipalus. Explaining causes behind sql query formulation errors. In *2020 IEEE Frontiers in Education Conference (FIE)*, pages 1–9, 2020. iii, 2, 30, 33

[42] Toni Taipalus, Mikko Siponen, and Tero Vartiainen. Errors and complications in sql query formulation. *ACM Trans. Comput. Educ.*, 18(3), aug 2018. 30

[43] Juan M. Tamayo, Alex Aiken, Nathan Bronson, and Mooly Sagiv. Understanding the behavior of database operations under program control. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 983–996, New York, NY, USA, 2012. Association for Computing Machinery. 27

[44] Aditya Vashistha. Measuring query complexity in sqlshare workload. 2015. 40, 41

[45] N Yakovets. PlaySQL - SQL practicum. Available at `http://yakovets.ca/playsql/`. 17

Appendix A

Mock-ups of QuerySandbox

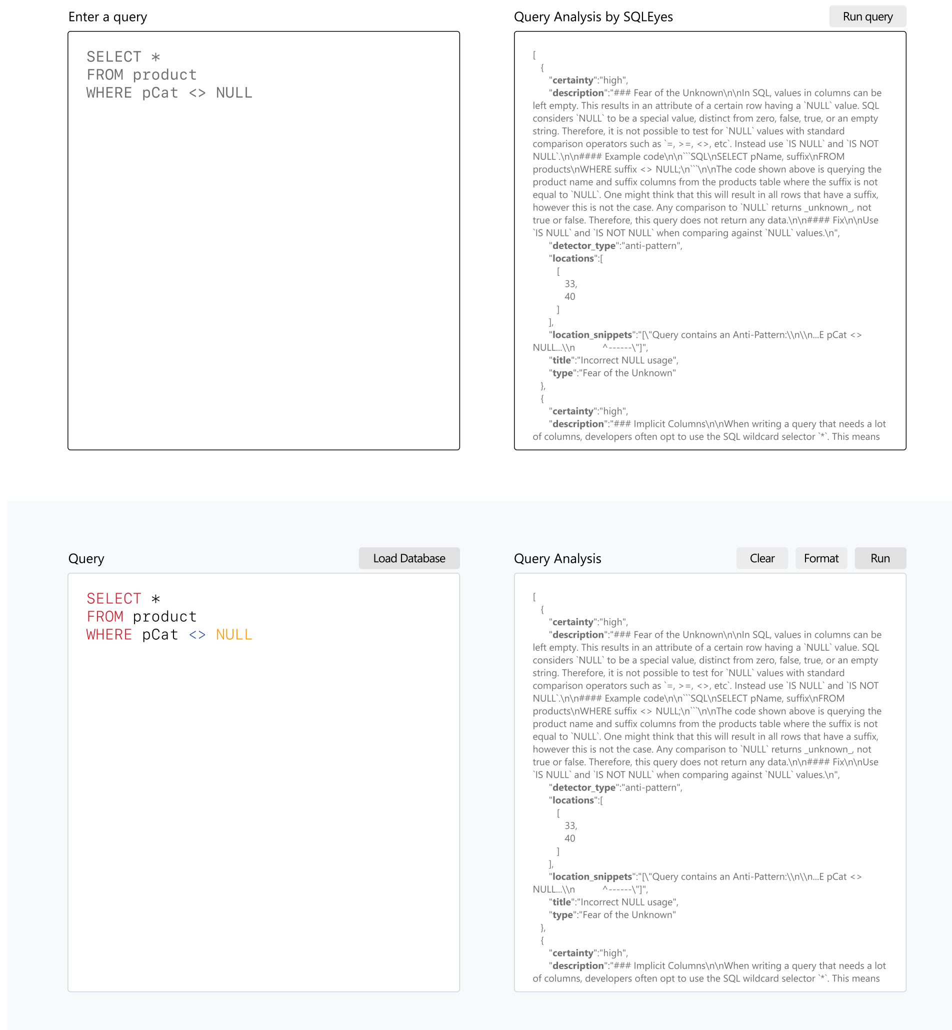


Figure A.1: Mock-ups of QuerySandbox during the UI design phase.

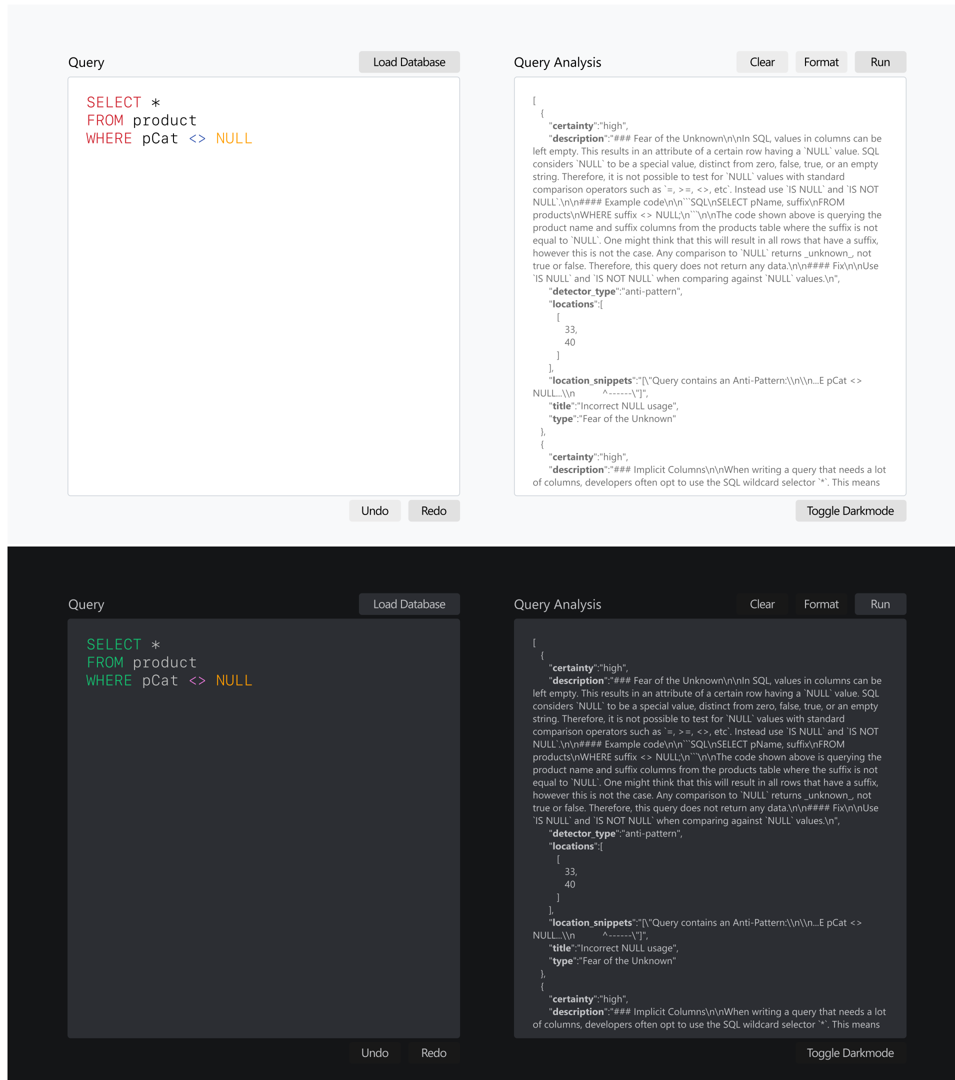


Figure A.1: Mock-ups of QuerySandbox during the UI design phase.

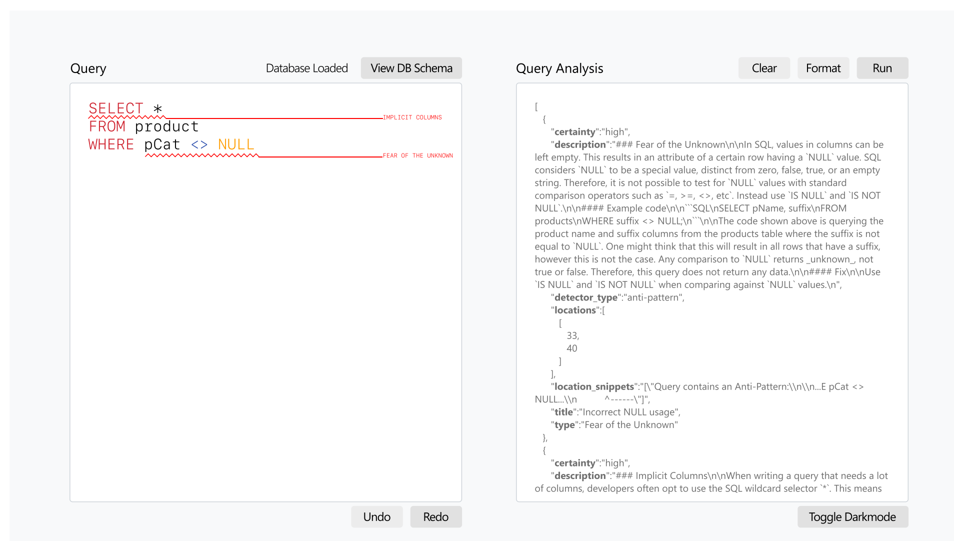


Figure A.1: Mock-ups of QuerySandbox during the UI design phase.