Using antipatterns to improve database code fragments, and utilizing knowledge graphs and NLP patterns to extract standardized data element names

by

BADER ALSHEMAIMRI

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2022

Using antipatterns to improve database code fragments, and utilizing knowledge graphs and NLP patterns to extract standardized data element names

The members of the Committee approve the doctoral dissertation of Bader Alshemaimri

Ramez Elamsri
Supervising Professor _____

Christoph Csallner _____

Leonidas Fegaras _____

David Levine _____

Dean of the Graduate School _____

To my mother, my father, my wife and my sons Khalid and Saud

ACKNOWLEDGEMENTS

In the name of Allah, the Most Gracious, the Most Merciful. Thanks to Allah above all for giving me the strength and time to complete this work. Thanks to my greatest teacher, prophet Mohammad, peace be upon him, who encouraged us to obtain knowledge from birth until death.

I would like to thank my supervisor, Dr. Ramez Elmasri, who has been a great mentor to me since I started my program at UTA. I am honored to be one of his Ph.D. students. I really appreciated his continuous guidance, support, and motivation during my doctoral program. I would also like to thank my academic committee members: Dr. Christoph Csallner, Dr. Leonidas Fegaras, and Mr.David Levine, for being my teachers during Ph.D. programs and for their insightful comments.

I would like to thank my parents, Khalid Alshemaimri (may he rest in peace) and Sherifah Alwetaid, for all they have provided me in life and are still providing, I would not have been able to complete this work without your continuous love and support. I lost my father during my fourth year of study. It was a great lost and had an impact on me. May he rest on peace. I would also like to thank my wife, Mona Alhusayni, our sons, Khalid and Saud, for their support, patience, and encouragement during my program.

My sincere gratitude goes to King Saud University and the government of Saudi Arabia for funding and supporting me during my ESL, and higher education.

Last but not least, I would like to thank my friend Dr. Tariq Alsahfi for his time and effort to evaluate and review my work. I would like also to thank Michael Ellis, a data engineer at BNSF, for his great feedback and comments. I would also to

thank my friends Dr. Mousa Almoutairi, Dr. Bhanu Jain, Mary Koone, Mohammed Shito, and all members of the MAST lab.

March, 2022

ABSTRACT

Using antipatterns to improve database code fragments, and utilizing knowledge graphs and NLP patterns to extract standardized data element names

Bader Alshemaimri, Ph.D.

The University of Texas at Arlington, 2022

Supervising Professor: Ramez Elamsri

Database code fragments exist in software systems by using SQL as the standard language for relational databases. Traditionally, developers bind databases as backends to software systems for supporting user applications. However, these bindings are low-level code and implemented to persist user data, so Object Relational Mapping (ORM) frameworks take place to abstract database access details. These approaches are prone to problematic database code fragments that negatively impact the quality of software systems. In the first part of the dissertation, we survey problematic database code fragments in the literature and examine antipatterns that occur in low-level database access code using SQL and high-level counterparts in ORM frameworks. We also study problematic database code fragments in different popular software architectures such as Service Oriented Architecture (SOA), Microservice Architecture (MA), and Model View Controller (MVC). We create a novel categorization of both SQL schema and query antipatterns in terms of performance, maintainability, portability, and data integrity.

In the second part of this dissertation, we create NLP patterns that support data architects when modeling and naming data element definitions. We design and develop rule-based natural language processing (NLP) techniques to automatically extract standardized data element names from data element definitions written in American English. The goal is to study how using NLP techniques can improve the accuracy of extracting standardized data element names in a domain-independent context. It is a challenge to come up with NLP patterns in natural language definitions as opposed to unambiguous code.

To achieve automated data element naming, we first identify heuristic patterns that mine noun phrases and relationships from data element definitions. Then, we use these noun phrases and relationships as input to determine components of data element names. The output of the patterns is reviewed by a domain expert. We apply our method to extract the five standard components of a data element name in the Railway and Transportation domains. We first achieved 80% accuracy, then by improving the rules and adding a similarity function using knowledge graphs, we improved the accuracy to 95% in our final experiments.

We also introduce our tool entitled as Data Element Naming Automation (DENA) tool. The tool consists of four components: DENA NLP, DENA assembly, preprocessing, and duplicate checker. In the last part of the dissertation, we propose how we preprocess data element definitions and evaluate the deduplication detection.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

This dissertation addresses two problems that are highly relevant for software development that depends on database access. The first problem is how to identify and deal with bad database code fragments in software, commonly referred to as database antipatterns. The second problem is how to extract and standardize database element names in large organizations with many diverse databases. In this chapter, we briefly introduce these two problems, discuss our contributions, and then give an outline for the chapters of the dissertation.

These two problems might appear different, but they share the same goal of creating useful, correct, efficient and standardized software to access databases. These two problems are related because they make use of better database practices. On one hand, implementing tools that can spot database antipatterns helps achieve the goal of correct and efficient software. This problem addresses applying standardized database designs and programming constructs. Without a comprehensive catalog of antipatterns, these tools will not achieve the goal. On the other hand, implementing an automation tool for the standardized naming of data elements cannot be done without a standardized knowledge graph model and representation. This helps achieve the goal of useful and standardized database software.

In section 1.1, we start this chapter with definitions and motivation for utilizing patterns and antipatterns to improve database code fragments, and using NLP patterns and knowledge graphs to extract standardized data element names from natural language definitions. Then we summarize our research contributions in section 1.2.

After that, Section 1.3, gives an outline of the remaining chapters of the dissertation and how it is organized.

1.1 Definition and Motivation of Patterns and Antipatterns for Database Code Fragments and Data Element Names from Natural Language

Generally, when a software system uses SQL as means of communicating with a database, database code fragment is written as a set of Data Definition Language (DDL) or Data Manipulation Language (DML) statements. A problematic database code can be a bug, SQL smell, or SQL antipattern. Bugs manifest errors that cause a software system to crash. SQL antipatterns allow executing programs, but they have quality problems, as we shall discuss. Some research papers use SQL antipatterns term, and others coin them as SQL code smells even though they are slightly different.

A data element is the smallest level of granularity of data that has a precise meaning and is both unique and of interest and use to the business is a data element. Data modelers write data element definitions to describe a data element by stating its complete and precise meaning in a unique English statement.

Hence, a database is an integral part of any software system. Having high-quality databases lead to high-quality software systems that have a fewer number of problematic database code fragments. Developers should know which parts of the source code degrade database quality. Several studies compile database-related categorization of problematic database code [ref: 14,15,16]. Developers can use software tools that implement these categorizations to automatically perform database analysis to determine flaws affecting the quality of software systems. However, these software tools require comprehensive categorization of problematic database code fragments in various application domains. As yet, there has been no systematic investigation of database problematic code fragments in disparate software systems

1.2 Dissertation Contributions

The contributions of this dissertation can be divided into two parts. In the first part, we provide two categorizations of antipatterns that represent bad practices of database programming and design. To achieve these categorizations, we surveyed 42 database-related antipatterns in various application domains and higher-level frameworks. These surveyed 42 database related antipatterns can have a bad effect on four non-functional requirements of software systems: performance, maintainability, portability, and data integrity. These anti-patterns degrade the quality of software systems and creating these categorizations allow researchers to develop software tools that can identify such problematic database code fragments. We analyze the impact of each categorized database-related antipattern to allow researchers and developers to identify and correct any given antipattern.

In the second part, we address and propose a solution to a problem that affects many large organizations: namely to understand, standardize, and avoid duplication of data element definitions and names in the many diverse databases within the organization. Large organizations migrate data in databases from one generation to another. They also might acquire other organizations that use different naming conventions for data. Data modelers often use the same data elements but may write different definitions and use different naming conventions. In some cases, similar data element names may have different meanings in diverse databases. Thus, it is important for such an organization to have a standardized way of naming these data elements, with standard terminology for all databases. At a minimum, different names for the same data element must be identified as duplicates of the same concept to avoid confusion and misunderstanding. In this part of the dissertation, we extract standardized data element names from definitions written by data modelers. These standardized data element names help resolve inconsistent data element names within

an organization. Our contribution led to developing a prototype system that includes the following: preprocessing NLP definitions to identify noun phrases and relationship using NLP tools; developing heuristic rules to create the data element names by assembling its components from the noun phrases and relationships; creating a knowledge graph that models these terms and relationships of data element components; and utilizing the knowledge graph to improve accuracy and identify duplicates.

1.3 Dissertation Organization

In Chapter 2, we review existing studies on SQL antipatterns, and we define these antipatterns and provide a classification to group antipatterns into schema antipatterns and query antipatterns. We further classify these antipatterns based on the four nonfunctional requirements of software systems: performance, maintinability, portability and data integrity. We also provide the support for SQL antipatterns by reviewing the state of the art tools that use these antipatterns to improve the quality of database within software systems.

In Chapter 3, we survey framework-specific antipatterns and we classify them the same way as we did in Chapter 2. Framework-specific antipatterns include antipatterns relating to different software architectures: Model View Controller (MVC), Service Oriented Architecture (SOA), and Microservice Architecture (MA). Framework-specific antipatterns also contain ORM antipatterns and these antipatterns might show issues in different ORM vendors such as Hibernate and Ruby on Rails.

In Chapter 4, we introduce the problem of extracting components of data element names from natural language definitions. We discuss the related work and explain the data element naming standard that is vital for a large organization. We also discuss the five component of a standardized data element name. We use patterns

to create the standardized name and assemble its components using NLP techniques in Spacy.

In Chapter 5, we define and describe a Data Element Naming Automation (DENA) tool. We also go through the inner workings of DENA's framework. We discuss how DENA preprocesses a definition so that we have a normalized definition after replacing terms such as common misspellings, undesired synonyms, American English in the definition. Then, we explain a deduplication checker functionality that DENA calls to search for duplicats in the knowledge graph. DENA can rank all possible duplicates and sort them based on name rank, definition rank and a combination of both.

In Chapter 6, the future proposed work is discussed.

1.4 Published Papers

As a result of my research, some articles were published during my Ph.D. study. The following are the published papers:

- **B. Alshemaimri**, R. Elmasri, T. Alsahfi, and M. Almotairi, "A survey on problematic database code fragments in software systems," Engineering Reports, vol. 3, no. 10, pp. 53–66, Jul 2021. Available: https://doi.org/10.1002/eng2.12441
- Tariq Alsahfi, Mousa Almotairi, Ramez Elmasri, and **Bader Alshemaimri**. 2019. Road Map Generation and Feature Extraction from GPS Trajectories Data. In Proceedings of the 12th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'19). ACM, New York, NY, USA, Article 2, 1–10.
- **Bader Alshemaimri**,Michael Elli, Ramez Elmasri, and Abdullah Almoqbil. 2021. Extracting Data Element Names from Natural Language Definitions.

In Proceedings of The 19th International Conference on Scientific Computing (CSC 2021). Springer International Publishing.

- **Bader Alshemaimri**, Michael Elli, Ramez Elmasri, and Tariq Alsahfi. "DENA: Data Element Naming Automation." The International FLAIRS Conference Proceedings. Vol. 35. 2022. (submitted)

CHAPTER 2

A Survey of SQL Antipatterns (Drafted from [1])

2.1 Introduction and Motivation

The Structured Query Language (SQL) is the standard language for relational databases. SQL is the most popular language to communicate with a database. As stated by the 2019 survey of StackOverflow[4], SQL is the third most popular programming language gaining only fewer votes than JavaScript and HTML/CSS and more votes than popular programming languages such as C/C++, C, and Python. Generally, when a software system uses SQL as means of communicating with a database, SQL is written as a set of Data Definition Language (DDL) or Data Manipulation Language (DML) statements [5]. These statements comprise the concept of low-level database access code. Database access code written using SQL is a data sublanguage in the host programming language. Alternatively, programming languages have Object-relational Mapping (ORM) frameworks, which mitigate writing database access code and rely on SQL to be the chosen standard language for querying databases in the industry.

Binding databases to software systems. One way to write database access code is to implement low level database-related binding styles. There are three traditional binding styles between the host programming language and the Domain Specific Language (DSL i.e. SQL): Module Language, Embedded SQL, and SQL Call Level Interface SQL/CLI[6]. Module Language binding style (also referred to as a database programming language) encourages writing all SQL statements in one or more modules, also known as Stored Procedures. Stored procedures are vendor-

specific SQL statements written outside of the host programming language programs, and these programs can call stored procedures. The stored procedure defines programming language constructs such as looping, branching, and merging with the DBMS at the server-side; thus, it reduces the overhead that results from network round trips between client and server. For instance, Procedural Language/Structured Query Language (PL/SQL) is an example of stored procedures for the Oracle DBMS.

In the embedded SQL approach, database access code can be written by statically or dynamically embedding SQL into the application source code layer. Statically embedded SQL is a way to intertwine SQL statements with programming language source code. For example, SQLJ, an extension to Java, integrates static SQL statements with Java source code. In SQLJ, the precompiler can statically extract embedded SQL code fragments from the application and send them to the database management system (DBMS) to process and execute the query. SQLJ is not commonly used in the industry because it lacks IDE support, type safety, and dynamic SQL binding. A successor of SQLJ, termed Java Object-Oriented Query JOOQ [7], was developed to overcome these issues. An equivalent component, called LINQ to SQL, exists for the .NET framework as well. Dynamically embedded SQL statements are not known at compilation time since they are interactive and require user input beforehand. SQLJ is not able to dynamically bind SQL, while JOOQ and LINQ to SQL have dynamic features. However, the latest versions of SQLJ support dynamic SQL embedding.

SQL/CLI binding style, also known as Function call libraries, allows programmers to communicate with the DBMS by providing a library of database functions in which developers can dynamically pass queries as strings and bind parameters at runtime. Examples of SQL Call Level Interface that access the library of database API

calls from the host languages are Open Database Connectivity (ODBC), which allows source programs to use operations from various DBMS vendors, and Java Database Connectivity (JDBC). JDBC, for instance, enables one java program to link to multiple vendor-specific DBMSs. JDBC does not abstract database schema and, unlike ORM, enables developers to write complex queries. The three binding styles between the host language and the data sublanguage, SQL, form the low-level database access details.

ORM frameworks. As an alternative to using traditional binding styles, developers can use an ORM framework to implement database access code without dealing with the low-level database access details. The ORM abstracts SQL queries by adding a high-level vendor-specific query language to ease developers' development time. For instance, Hibernate, an ORM framework for Java, implements Java Persistence API (JPA) and uses JDBC connections, adds Hibernate Query Language (HQL), and maps the database accesses written using the host language to the SQL flavor of the relational DBMS (RDBMS). It lets developers deal with persistent objects rather than table rows. Goeminne et al [8] conduct a survival analysis of 3,707 GitHub projects implemented in Java. They state that JDBC, Hibernate, and Java Persistence API (JPA) are the top popular technologies to interact with a DBMS. Hibernate tends to be used along with other database frameworks such as JDBC and JPA, while 56.3% of the GitHub projects use JDBC alone to facilitate database access [8].

Regardless of which way Object-Oriented Programming Language (OOPL) is used to interact with the RDBMS, the object-relational impedance mismatch problem, which is the differences of converting database model to the object-oriented paradigm, exists [9]. Instead of conversion, using only one language representing either an application program or the database eliminates the impedance mismatch

program. This application program implemented by a host language can either directly embed/integrate SQL into the host language constructs. Another way to solve the impedance mismatch problem is by using an abstraction that maps OOPL objects into relational constructs and abstracts low-level database access code using an ORM framework, such as Hibernate. Using DBMS integrated languages, such as DDL and DML, in either low-level database access code or ORM can include problematic database codes.

Problematic database access. The focus of our paper is to survey the types of problematic database code fragments. We categorize them based on those concerning schema or query and the impact on various system factors that include performance, maintainability, portability, and data integrity. A problematic database code can be a bug, SQL smell, or SQL antipattern. Bugs manifest errors that cause a software system to crash. SQL antipatterns allow executing programs, but they have quality problems, as we shall discuss. Some research papers use SQL antipatterns term, and others coin them as SQL code smells even though they are slightly different. Sharma et al. [10] discuss whether SQL smells and SQL antipatterns are synonyms. They treat them differently because poorly chosen design decisions lead to SQL antipatterns, while SQL smells occur unintentionally due to developers' oblivion. SQL code smells are hints that might indicate there is something wrong with your code, but it is not for granted. Kent Beck introduced the term code smells when he presents "Once and only once" in his blog[11]. Unlike SQL smells, SQL antipatterns surely lead to less maintainable, slow-performing, corrupted data, and platform-dependent source code eventually. Koenig [12] introduces the notion of antipattern as "a common solution to a recurring problem that is usually ineffective and highly counterproductive". Linares-Vásquez et al. [13] illustrate that software anti-patterns are framework-specific and found in different application domains. SQL

antipatterns follow the definition of antipatterns. SQL, as a DSL, has its code smells and antipatterns in its domain. Various application domains use SQL as a data sublanguage and SQL antipatterns can manifest differently in these domains. For this reason, we study problematic database code fragments in various and popular software architectures.

Other software architectures for problematic database code. We review problematic database code fragments in different architectural application domains such as Service Oriented Architecture (SOA) [14], Microservice Architecture (MA) [15], and Model View Controller (MVC) [16]:

- SOA is an architectural style that decomposes monolithic software systems into standalone, maintainable, and platform-neutral software components, called services. These services can be used with web services and communicate with protocols such as Remote Procedure Call (RPC), Single Object Access Protocol (SOAP), JavaScript Object Notation (JSON), and Extensible Markup language (XML) by sending and receiving messages. A successor of SOA, known as Microservice Architecture (MA), has emerged and embraced some concepts of SOA, but they differ in service characteristics.
- MA has fine-grained services, while SOA has coarse-grained services. SOA has component sharing as one of its basic tenets, but MA embraces "share as little as possible". Problematic database code can vary between the two approaches, but both need to be analyzed when developers request for services using either SOA or MA.
- MVC is another architectural design that partitions application development into model, view, and controller to separate concerns. Examples of MVC web frameworks are Ruby on Rails and Spring MVC, a popular Java project. Transactions in Spring MVC differ from transactions in Hibernate because Spring

manages configuration and dependencies across objects while Hibernate abstracts database components to class objects. Generally, web applications follow MVC pattern[16] and problematic database code can occur in either one of these three layers.

A database is an integral part of any software system. Having high-quality databases lead to high-quality software systems that have a fewer number of problematic database code fragments. Developers should know which parts of the source code degrade database quality. Several studies compile database-related categorization of problematic database code [2], [3], [17]. Developers can use software tools that implement these categorizations to automatically perform database analysis to determine flaws affecting the quality of software systems. However, these software tools require comprehensive categorization of problematic database code fragments in various application domains. As yet, there has been no systematic investigation of database problematic code fragments in disparate software systems. Furthermore, as antipatterns can appear in any framework or application domain [13], it is crucial to study its characteristics and impacts to foster the development of software tools that capture such antipatterns. To encourage more research in this direction, we review the literature about database problematic code across legacy information system applications, service-based applications, microservice-based applications, and web applications. We also categorize problematic database code in terms of the impact they leave on software systems such as performance, maintainability, portability, and data integrity.

Our contributions can be summarized as follows:

- We introduce two categorizations of the studied 42 database-related antipatterns in various application domains and higher-level frameworks.

- We classify database-related antipatterns based on the impact on non-functional requirements such as performance, maintainability, portability and, data integrity.
- We analyze the impact of each categorized database-related antipattern.

2.1.1 Terminology and Concepts Used

Problematic Database Code: A poorly written database access code that is a result of either inefficiently connecting to the database, querying the database, making a database transaction, or returning the result from the DB causes problems in terms of performance, maintainability, portability, and data integrity.

- **Performance Antipatterns:** A poorly written database access code that causes performance degradation of overall software systems. Reasons vary from improper data retrieval and excessive data processing in memory or data transmission. Performance antipatterns slow down the overall system.
- **Maintainbility Antipatterns:** Maintainability antipatterns make the system harder to maintain, leading to higher maintenance time and costs.
- **Portability Antipatterns:** Portability antipatterns prevent the system from migrating to a different system architecture or database engine.
- **Data Integrity Antipatterns:** Data integrity antipatterns make the overall system unreliable since it corrupts or conceals data; thus, it forces the system state to be inconsistent.

We organize the rest of the paper as follows. Section 2.2 reviews the literature of SQL problematic code fragments, including generic SQL antipatterns and framework-specific antipatterns because it is evident that antipatterns vary across application domains. As a result, we discuss a set of frameworks and their related antipatterns. In section 2.3, we study the support for SQL antipatterns and others' work at detecting

such antipatterns so that it is possible to extend these catalogs and categorize them. We discuss our findings and categorizations of SQL antipatterns in section **??**. We conclude about our findings in section **??**.

2.2 Review of SQL Problematic Code Fragments

In this section, we review the literature on SQL problematic code fragments. By problematic code fragments, we mean antipatterns in software artifacts that are symptoms resulting in eventual errors or quality problems. We introduce the studied SQL antipatterns in Section 2.2.1. In section 3, we review the literature about SQL antipatterns that are specific to frameworks such as architectural application frameworks and object-relational mapping frameworks.

2.2.1 SQL Antipatterns

SQL antipatterns found in database code can eventually affect software systems with problems in performance, maintainability, portability, and data integrity. Karwin [2] outlines SQL antipatterns from developers' point of view that lead to problems with SQL. As he stated in his book, "SQL Antipatterns describe the most frequently made missteps I've seen people naively make while using SQL.". He classifies SQL antipatterns into three categories: database design antipatterns, including physical and logical database design antipatterns, query antipatterns, and application development antipatterns.

In addition, other books exist identifying SQL code smells [3] and SQL antipatterns [17]. Red Gate [3] documents 119 SQL code smells, concerning database design issues, table design, data types, expressions, naming, routines, query syntax, and security loopholes.

We introduce SQL antipatterns based on schema and SQL antipatterns based on query in sections 2.2.1.1 and 2.2.1.2.

2.2.1.1 SQL Antipatterns Based on Schema

In this section, we introduce and review low-level database problematic code fragments in terms of SQL schema antipatterns. Karwin [2] classifies SQL schema antipatterns into physical and logical database design antipatterns. He lists 8 antipatterns for the logical database design antipatterns and 4 antipattens for the physical database design antipatterns. The table 2.1 shows the SQL Antipatterns based on Schema that are classified into logical and physical database design antipatterns by Karwin [2]. We will not categorize Rounding errors and Phantom Files because these are subjective and are not implemented in tools that support these antipatterns.

Table 2.1: Classification of SQL Anitpatterns by Karwin [2].

Schema		**Query**
Logical Database Design	**Physical Database Design**	
Compound Attribute	Rounding Errors	Buried NULL
Adjacency List	Values in Attribute Definition	Reference Nongrouped Column
Superfluous Key	Phantom Files	Random Selection
Missing Constraints	Index Shotgun	Pattern Matching Predicates
Metadata as Data		Spaghetti Query
Polymorphic Associations		Implicit Columns
Multicolumn Attribute		Poor Man's Search Engine
Clone Tables		

Sharma et al [10] enhances the catalog of Karwin [2] in terms of SQL schema antipatterns by introducing 4 more antipatterns gathered from the literature, industry and online discussions [[2], [3], [18]]:

- Mutlicolumn attribute.
- God table.
- Meaningless name.
- Overloaded attribute names.

Redgate [3] categorizes SQL smells into many dimensions, but we only focus on problems with database design, table design, and query syntax. Table 2.2 shows the list of antipatterns for each dimension. Most of these antipatterns exist in Karwin's catalog [2], but they have different name and description. We will include these antipatterns in our categorization except contrived interfaces and using command-line and OLE automation antipatterns from database design antipatterns. For problems with table design, we will also categorize these antipatterns except creating a table without specifying a schema.

The following maps the two lists of items in Table 2.1, 2.2 as well as antipatterns found in Sharma's catalog [10]:

- **Compound Attribute:** It is also known as packing lists, complex data, or other multivariate attributes into a table column.
- **Adjacency List**: It is also known as storing the hierarchy structure in the same table as the entities that make up the hierarchy.
- **Missing Constraints:** It includes not using referential integrity constraints.
- **Polymorphic Association**: It is also known as using a polymorphic association.
- **Clone Tables:** It is also known as creating dated copies of the same table to manage table sizes.

- **God Table** [10]: It is also known as creating tables as 'God' Objects.
- **Values in Attribute Definition:** It is also known as using constraints to restrict values in a column.
- **Index Shotgun:** It is also known as using too many or too few indexes.
- **Overloaded Attribute Names:** It is also known as using the same column name in different tables but with different data types.

The antipattern Buried NULL from Table 2.1 that was classified as query antipattern has a similar definition to the definition of misusing Null values antipattern that was classified as problems with Table Design in Table 2.2. Since we classify problems related to database design and table design into schema antipatterns, we found a conflict between these categories. We shall explain and classify most of the above SQL schema antipatterns from Tables 2.1 and 2.2 in more detail in section 2.4

2.2.1.2 SQL Antipatterns Based on Query

We describe and analyze low-level database problematic code fragments in terms of SQL query antipatterns. Table 2.1 shows a column of query antipatterns where all query antipatterns are addressed by Karwin [2]. We will not categorize Spaghetti Query and Implicit Columns because these are subjective and are not implemented in tools that support these antipatterns.

Redgate [3] also classifies some SQL smells into query syntax. Table 2.2 shows antipatterns that have problems with query syntax. The following groups similar query antipatterns:

- **Buried Null:** It includes not handling NULL in nullable columns and using '= NULL' or '<> NULL' to filter a nullable column.
- **Pattern Matching Predicates:** It is also known as using LIKE in a WHERE clause with an initial wildcard character.

- **Spaghetti Query:** It is also known as creating UberQueries (God-like Queries).
- **Row-by-row or one-by-one [19]:** It is also known as using correlated subqueries instead of a JOIN. The row-by-row antipattern will be discussed in section 3.2.
- **Implicit Columns**: It is also known as using INSERT INTO without specifying the columns and their order.
- **Using Union or Distinct to Remove Join Duplicates [20]:** It is also known as using SELECT DISTINCT to mask a joining problem.

We shall define and classify these SQL query antipatterns in more detail in section 2.5.

2.2.1.3 SQL Antipatterns Based on Security Breaches

In this section, we introduce and review the literature concerning SQL antipatterns that can impact the security of software systems. Karwin [2] categorizes this group of antipattern into application development antipatterns. This group of antipatterns contains six antipatterns in total. Two of the six antipatterns are SQL antipatterns related to security breaches. Table 2.1 shows schema and query antipatterns as two groups of the classification.

Another study surveys SQL antipatterns in local databases that mobile applications rely on heavily [21]. They identified eleven SQL antipatterns through their literature review. They classified eight of them into the runtime and energy consumption antipatterns and the rest as security antipatterns. The following shows the list of the antipatterns:

- **Vulnerable Query:** It is also known as SQL Injection.
- **Unbounded-Query:** It is also known as Unnecessary Computation [22].
- **Readable Password:**. Not encrypting user password poses a security threat.

Table 2.2: Classification of SQL Anitpatterns by Red Gate [3].

Schema		**Query**
Problems with Database Design	**Problems with Table Design**	**Problems with Query Syntax**
Packing lists, complex data, or other multivariate attributes into a table column	Using constraints to restrict values in a column	Creating UberQueries (God-like Queries)
Storing the hierarchy structure in the same table as the entities that make up the hierarchy	Not using referential integrity constraints	Using correlated subqueries instead of a JOIN
Using an Entity Attribute Value (EAV) model	Using too many or too few indexes	Using INSERT INTO without specifying the columns and their order
Using a polymorphic association	Misusing NULL values	Not handling NULL values in nullable columns
Creating tables as 'God Objects'	Creating a table without specifying a schema	Using LIKE in a WHERE clause with an initial wildcard character
Contrived interfaces	Using the same column name in different tables but with different data types	Using '= NULL' or '<> NULL' to filter a nullable column
Using command-line and OLE automation to access server-based resources	Creating dated copies of the same table to manage table sizes	Using SELECT DISTINCT to mask a join problem

2.3 Support for Identifying SQL Antipattern

In the following, we discuss recent studies that identify SQL antipatterns from different points of view: querying system catalogs in section 2.3.1; parsing database schema (DDL) in section 2.3.2; parsing SQL statements (DML) extracted from SQL parser in section 2.3.3; and mining software repositories to identify SQL schema antipatterns in section 2.3.4.

2.3.1 Identifying SQL (Schema) Antipatterns by Querying the Schema Stored in RDBMS Catalog

Based on [2], many research studies analyze system catalogs and schema to identify SQL antipatterns. Eessar [23] analyzes 11 SQL antipatterns including 8 logical and 3 physical design anti-patterns presented by [2]. Then, he creates a list of 14 SQL detection queries to search for these 11 SQL antipatterns and conduct semiautomate analysis of the quality of databases.

In a later work, Eessar [24] assesses the anti-pattern-detection queries by using them to evaluate 41 databases designed by students, introduce more design questions about conceptual data modeling of SQL databases and study potential application uses of the queries. As a result of executing detection queries, he found 81 cases that refer to antipattern occurrences in the student database designs. Out of 81, 23 real cases are results of manually checking the database specifications. Eight cases out of 23 are subject to false-positives.

Querying database schema to search for potential database design anti-patterns is also discussed by other researchers. Khumnin et al. [25] propose research about how they can help database administrators detecting SQL anti-patterns and recommend refactoring techniques similar to the way developers perform it with code antipatterns. They aim to automate the process of identifying SQL anti-patterns by implementing a tool that employs Transact-SQL language to query and examine the database schema. Khumnin et al. [25] experiment on the five logical database design anti-patterns out of 8 in total. The tool can document prospective anti-patterns by following a query-based detection/heuristic approach to find them.

Khummin et al [25] compare their findings with earlier work of [23]. Khummin et al [25] implement generic query detections, whereas in [23] detection queries are bound by written SQL queries as examples in [2]. They mention that the work of [23]

excludes any query evaluation, but the authors can compare their query performance with their later work [24]. In this work, the authors consider different cases of missing constraints antipatterns, while in Eessar's work, missing foreign key constraints is the only analyzed case.

2.3.2 Identifying SQL Antipatterns by Statically Analyzing Database Schema Using the Schema Dump

One study parses the internal structures of database schema to find SQL antipatterns. Delplanque et al. [26] examine the database schema to evaluate the quality of the database code they embed. They believe that only querying the system catalog is not enough to locate SQL antipatterns in triggers and stored procedures. Consequently, they treat the database as source code when applying software quality and tools. They implement DBCritics, which is a tool to analyze PostgreSQL schema dumps. The tool parses the schema dump conforming to PL/SQL as an input, and it generates a model. The three usage scenarios that the authors use Dbcritics for are SQL antipattern detection, DBMS migration, and database schema consistency [26]. The first usage scenario applies to the scope of our paper.

2.3.3 Identifying SQL Query Antipatterns Using the Query Parser

Another study aims to identify SQL antipatterns in embedded queries using SQL query parser. Nagy et al. [27] propose a prototype tool developed to inspect SQL queries written within Java source code without running the code and recognize SQL query antipatterns described in [2]. The prototype tool consists of two main components: SQL extractor and SQL code smell detector. The input of the SQL extractor is Java source code. Then, by using a string resolution technique confined within a procedural along with Abstract Syntax Tree (AST), the SQL extractor gen-

erates several SQL statements with call stack information that shows tracing of these SQL statements in a call chain. They design their tool to identify SQL query antipatterns such as buried NULL, ambiguous group, random selection, and implicit columns. However, they excluded Spaghetti Query and Poor Man's Search Engine antipatterns because there is no objective definition for Spaghetti Query that makes the query Spaghetti (too large), and the solution for the Poor Man's Search Engine using pattern matching predicates is dependent on DBMS. This publicly available tool can distill SQL queries from the source code before it runs, then analyze embedded SQL statements in the program, the database model, and the data to determine antipatterns in the queries.

2.3.4 Identifying SQL Schema Antipatterns Using the Query Parser from Software Repositories

A study aims to explore the presence of SQL antipatterns [2] in source code and analyze their correlations between other software artifacts by mining embedded SQL in software systems. Sharma et al. [10] used regular expressions to extract embedded SQL statements from the repositories and then find SQL antipatterns. They categorize SQL antipatterns into SQL schema antipatterns, SQL query antipatterns, and SQL data antipatterns. Sharma et al. [10] examine SQL statements in 357 industrial and 2568 open-source projects to evaluate how schema quality of relational database affects the performance and maintainability of software systems. They specifically investigate how frequent SQL schema antipatterns occur in software systems and to what degree SQL schema antipatterns correlate with each other. They also explore the aspects that impact the density of SQL antipatterns. The aspects are the size of the project, database, and source code.

They survey developers' perspectives about SQL schema antipatterns to understand their views. Sharma et al. [10] implement a DbDeo tool that is a smell detection to distill embedded SQL statements from source code programs. The DbDeo tool relies on a third-party tool called SQLParse to analyze SQL statements and generate metadata. The DbDeo smell detection tool exercises the meta-model to identify SQL schema antipatterns. The survey results indicate that being aware of SQL antipatterns is essential to design and develop high-quality software systems. However, it depends on the developer's perception of whether the problematic code is indeed a SQL antipattern or if it serves a particular purpose in source code [10]. Their tool detection results show that index abuse is the most common SQL schema antipattern, and some antipatterns, such as adjacency list, are more likely to exist in industrial projects than open-source projects by a measurable difference. Sharma et al. [10] find that using an ORM framework does not hinder SQL antipatterns.

2.4 Categorization of SQL Antipatterns based on Schema

In this section, we define and classify 11 *SQL schema antipatterns* based on the underlying four system factors of performance, maintainability, portability, and data integrity. Table 2.3 shows the result of the classification. We identify 9 schema antipatterns as performance antipatterns. Out of the 15 studied antipatterns, we categorize 8 schema antipatterns into maintenance antipatterns. We assign almost half of these schema antipatterns into portability and data integrity classification groups. We define all the schema antipatterns in a separate section for each and then discuss their effects on performance, maintainability, portability, and data integrity.

Table 2.3: Categorization of SQL Schema Antipatterns.

Reference	Antipattern	Performance	Maintainability	Portability	Data Integrity
[2, 18, 3, 10]	Compound Attribute	✓	✓	✓	✓
[2, 3, 10]	Adjacency List	✓	✓	✓	
[2, 10]	Superfluous Key				✓
[2, 3, 10]	Missing Constraints		✓		✓
[2, 3, 10]	Metadata as Data	✓			✓
[2, 3, 10]	Polymorphic Association	✓	✓		
[2, 18]	Multicolumn Repeating Attribute	✓	✓		
[2, 10]	Clone Tables	✓	✓	✓	✓
[2, 10]	Values in Attribute Definition	✓	✓	✓	
[2, 3, 10]	Index Shotgun	✓		✓	
[18, 3, 10]	God Table	✓	✓		✓

2.4.0.1 Compound Attributes Anti-pattern:

Definition: This antipattern emerges when developers utilize a column to store a comma-separated value for a compound attribute. For instance, when they want to store license plates for cars in the database, they might use a non-atomic attribute, i.e., (state, number) to save cars' license plates. The following impacts discuss a case where a compound attribute store comma-separated lists as values. This antipattern is bad because it complicates querying the DBMS. It also does not allow developers to easily use aggregate queries that include functions such as SUM() and Average().

Impact on performance: It affects performance if not used properly. If programmers have foreign keys organized in comma-separated lists, they cannot benefit from pattern matching and indexes. Thus, they have to write slower, complex queries, and they need to parse them. It is better to separate the component into individual attributes. Then, programmers can create a compound index on the combination of attributes if needed, or individual indexes on the individual attributes.

Impact on maintainability: Developers traditionally use Aggregate functions such as SUM(), COUNT(), etc over rows with atomic attributes, so by using these comma-separated lists, developers need to increase their effort at debugging and maintaining their database schema.

Impact on portability: As discussed in the impact on performance, each vendor has its syntax, when it comes to pattern matching, so the SQL statement is tied to a portability constraint.

Impact on data integrity: This antipattern affects data integrity for two reasons. With comma-separated lists, developers cannot prevent users from entering invalid entries like a word in a language that is different from English. It will not be an error, but the integrity of data is at stake. Choosing a separator character

is another reason why this antipattern degrades data integrity. If developers chose comma to separate values in a list, how would they assure that a comma will not appear in the entry? Using the comma as a separator character between entries might be ambiguous. Denormalization and storing a comma-separated value does not help the database to preserve data integrity, but it might increase performance.

Solution: Developers should create an intersection table of license plate states and numbers. They can name it as StateNumber table. This intersection table has a many-to-many relationship between State table and Licence Number table. Developers can eliminate this antipattern by creating the intersection table.

2.4.0.2 Adjacency List Anti-pattern:

Definition: This antipattern arises when a column in a table references another record in the same table. Specifically, it is an antipattern when developers query hierarchical data such as organizational structures, product categorization, and thread discussions, and they only depend on one node's parent to query the given node's descendants or ancestors at various levels. It is also referred to as a recursive relationship [5]. The adjacency list becomes an antipattern when developers want to query all descendants of a tree. The reason why it is bad is that it enforces developers to overuse joins for each decedent of the tree.

Impact on performance: This antipattern is categorized into performance antipatterns because developers need to write additional queries for each deeper level in the hierarchy when querying tree data structures. It shows up in cases such as querying a hierarchical structure and deleting a node in the structure. Developers sometimes depend on only one parent of a given query when they query a tree data structure at various depth levels. In this case, Adjacency List should not be used [2], [3], [10] because it is expensive to query all descendants of a given node in the tree

data structure using join strategies. The authors [28] recommends avoiding the join strategies if programmers can transform an SQL statement to a simpler equivalent. This antipattern is categorized into performance antipatterns because various tree data models have different uses, and choosing the right data model for the needed task helps to generate the best performance. We discuss these alternative solutions next.

Impact on maintainability: Using a traditional adjacency list without recursion and using only the immediate parent of a given node results in maintaining many numbers of SQL queries. Therefore, we also categorize this antipattern into one of the maintenance antipatterns.

Impact on portability: This antipattern affects portability because some DBMSs implement their own tree data model, and developers need extra efforts to migrate between these proprietary data models. An online discussion requests an alternative solution to replace Hierarchyid, a data type implemented by SQL server, with its equivalent MySQL data type [29]. The developer wants to migrate its DB from MS SQL to MySQL to support a specific tree data model.

Solution: Developers can also use Recursive Common Table Expression (CTE) with the adjacency list pattern to overcome generating queries for each hierarchy level. CTE is known as a recursive query in SQL3 or later versions. In SQL3, developers can write it as a single query, but the performance is dependent on how developers implement their systems. Alternatives to the Adjacency List, tree models are proposed such as path enumeration, nested sets, and closure table. Some of these are easier than others when querying a tree, querying children, inserting or deleting a node, and handling referential integrity to mitigate such performance overhead.

2.4.0.3 Superfluous Key Anti-pattern:

Definition: It occurs when you define a surrogate key, also known as an artificial key, over a natural key from other attributes in a table that fits the database schema. Developers learning SQL might comprehend the idea that any column named ID is always a primary key. For instance, they might define an ID column as the only primary key in addition to Student ID in a Student table. Developers may also fall into a situation of having an intersection of two tables, each of which has a primary key. They might define a surrogate key (ID column) and leave these two candidates to become a compound key; thus, this primary key, in turn, creates duplicate rows.

Impact on data integrity: We classify this antipattern into one of data integrity antipatterns since using the additional surrogate key creates duplicate rows, and a normalized table should have distinct rows. However, they can define a unique constraint over the compound key to solve the problem of duplicate rows.

Solution: Developers should select meaningful names for their primary key. For example, the primary key of the Employees table ought to be emp_id. Developers should also use compound keys when they are suitable. Listing **??** solves the example of Superfluous key antipattern illustrated in Listing **??**.Listing **??** removes the misunderstood ID column as the primary key and chooses the primary key to be the compound key (emp_id, depart_id), instead.

2.4.0.4 Missing Constraints Anti-pattern:

Definition: Developers sometimes choose not to include referential integrity constraints. When this occurs, they run into a missing constraint anti-pattern, forcing them to apply manually referential integrity at the application level.

Impact on maintainability: Missing referential integrity constraints will increase the time to develop the code because developers have to check referenced and referencing tables by writing SQL select statements. As a result, this escalates maintenance issues, and they have to lock tables before sending any DML queries because some other queries might go in the middle. Deleting a row or querying all children of a given row might introduce errors to the database [10]. Queries need to include joins for each level of the tree, so developers have to be careful and aware of what they issue to their database schema. They have to maintain and debug nested SQL statements of joins; thus, it minimizes the maintainability of such DBMS in a software system and increases the likelihood of introducing errors in the database.

Impact on data integrity: In a concurrent scenario and unlike single-threaded applications, when developers want to insert a new row into a database, they need to search a table and then insert the row if not found. Other developers might insert the same row while developers search for it. Developers will end up having duplicate rows. Preventing such a race condition requires locking the table because developers are inserting a new row, and there is no row to lock. Without referential integrity constraints, developers have to run select queries to check before they insert a child row and delete or update a parent row. In a concurrent scenario, it is not enough to check with select queries. They might also need to lock the table to prevent multiple changes from occurring. Developers do not need to run periodic quality scripts to remedy orphaned rows because referential integrity ensures it. The referenced and referencing tables can have mismatched rows when developers choose not to enforce referential integrity constraints. As a result, this antipattern affects the performance, maintainability, and data integrity factors of the database.

Solution: Enforcing referential integrity constraints minimizes the time it takes for developers to correct and check for data integrity mistakes. Developers can avoid

them from bypassing the database in the first place. Referential integrity constraints also have a cascading updates feature which cannot be replicated at the application level. This feature lets developers to update or delete the parent row and allows the database to carry out the same task on any child rows that reference the parent row. Even though referential integrity constraint brings some overhead, it proved to have better performance compared to the alternative. Developers do not need to write SELECT queries to verify before they insert, delete or update. Developers also do not need to lock tables to prevent associated table changes.

2.4.0.5 Metadata as Data Anti-pattern:

Definition: This antipattern occurs when developers want to support storing variable attributes. They can do it by creating a second table, storing attributes as rows in the form of EAV (entity-attribute-value). They use a generic attribute table that consists of an entity or child table which is a foreign key referencing a parent table, an attribute name defined in each row, and the value belonging to the entity for that attribute. The reason why this antipattern is bad is that using the EAV design sacrifices many advantages that a traditional database design would have given developers.

Impact on performance: Using this EAV design destroys all benefits of the conventional database design such as using SQL data types, enforcing referential integrity, and checking for consistent attribute names. Using EAV design, fetching a single row is complex and error-prone since it involves writing many mandatory outer joins as the number of defined attributes in the EAV design is increased. Retrieving the mixture of metadata and data results in performance degradation because alternative solutions such as single table inheritance, concrete table inheritance, and class table inheritance allow developers to reconstruct a single row using one or two SQL

statements and the query is not attribute-centered and is decoupled from how many attributes are in the database schema. Chen et al. [30] investigated performance issues of using EAV in a clinical database and found that performance using EAV was three to five times slower than using the conventional database design. One industry white paper lists EAV as a SQL code smell. Red Gate [3] proposes using XML in case developers are unaware of the nature of data being stored so that they can utilize XML Schema Definition (XSD) to impose XML constraints, generate indexes on the data, and utilize XML Path Language (XPath) for retrieving specific elements within the XML.

Impact on data integrity: When using EAV design, developers lose many advantages that are offered by a traditional database design. This antipattern is categorized into data integrity antipatterns because of the following reasons. Each attribute and its value are stored as a pair in each row, so developers cannot enforce mandatory columns because, in a traditional database, they can define a NOT NULL constraint for a column. Instead, developers need to define a constraint for each row that exists for a particular attribute and its value. Nevertheless, they have to develop this constraint since this feature is not offered by SQL. In EAV design, users might enter any value with different data types and formats, but this design cannot deter them from inserting into the database. However, they can add a column storing a data type as a value, but this will complicate querying the database. Unlike conventional database design, EAV design does not support referential integrity because it is applied on a column not a row. Database tables might result in rows with inconsistent attribute names in EAV design. Both intend to give the same information. Developers need a way to enforce data integrity by checking for inconsistent data.

Solution: Developers should use other modeling techniques to represent their data as EAV. They can model their data using single table inheritance or concrete

table inheritance [2]. With single table inheritance, the table contains a column that determines which type this row belongs two. For instance, modeling animal as EAV, developers might use a single table animal with a column species that tells whether this row is a cat or dog. For concrete table inheritance, developers may create one table for each subtype. For example, when developers model animals, they need to create a table for cats and another for dogs with some columns exclusive for each subtype.

2.4.0.6 Polymorphic association:

Definition: This occurs when developers use polymorphic association. One example is when people and companies can own a car. How can you tell who owns the car? Developers introduce polymorphic anti-pattern by adding an extra attribute in addition to a foreign key in a table that has two parent tables. This additional column tells what is the parent of the specified row. This is also sometimes referred to as a union type [5]. This reason why this antipattern is bad is because it does not enforce this association in meta data when referential integrity constraint is not in place. The reason why this antipattern is bad is because solving the complexity of having a large table by partitioning it creates another hurdle of having to manage many number of table partitions.

Impact on performance: This anti-pattern is classified as one of the performance anti-patterns even though foreign constraints have no direct effects on performance. However, having foreign key constraints allows developers to manually create indexes that increase performance gains and are targets for search optimizations. The query optimizer hardly can provide good plans for joins that are used to query polymorphic anti-pattern because its join predicate can only be met by one of the parent tables. Not only the optimizer might face difficulties dealing with this join, but also

developers have to discern the complicated logic behind it. ORM frameworks, for instance Hibernate, allow developers to use polymorphic associations, while RDBMS does not permit them to use polymorphic association. The benefit developers can get from using ORMs is that the complexity behind the polymorphic association is abstracted away by these frameworks.

Impact on maintainability: As discussed in the impact on performance, a foreign key constraint cannot be used because a foreign key must state exactly one table. Introducing the extra attribute causes the query to be complicated and undermines the ability to read and understand SQL code.

Solution: This antipattern has two solution as suggested by [2]. One solution is to simply reverse the reference of the polymorphic relationship. Instead of one child referencing more than one parent, it can be modeled the way around. Another solution is to create interesection tables. For instance, developers can create two intersection tables for people and companies owning cars. These intertsection tabels can be names as personalCars and businessCars.

2.4.0.7 Multicolumn Repeating Attribute:

Definition: It emerges when developers create multiple columns for an attribute. For instance, cars can have multiple colors, and developers might define three columns as colors for a car table. Using this anti-pattern complicates tasks such as searching for values, adding and removing values, ensuring uniqueness, and handling growing sets of values for an attribute in the conventional database design. Developers address growing sets of values by altering a table to include a given number of columns for a specific attribute. This is also sometimes referred to as a multivalued attribute, or repeating group [5].

Impact on performance: Creating multiple columns for an attribute incurs a cost in many ways because changing the structure of a table that holds data might need locking. Locking affects performance because a user process holds a lock on the old table and causes lock contention until this process releases the lock to the other user process trying to access the table.

Impact on maintainability: Developers might define a new table matching the structure of the old table and copy data from the old table to the new table, but this transfer can take time because the old table might have large data. When developers alter the table to have new columns, they need to revisit source programs that interact with the database to apply changes to SQL statements so that they can have the newly created columns. In addition, when searching for values for this attribute, developers need to examine all columns created for that particular attribute; thus, making the ability to read such SQL statements worse compared to using joins as a result of creating a dependent table storing those values as rows instead of columns. This antipattern also violates the Don't Repeat Yourself (DRY) principle [31] and enforces the query to be constrained to a fixed number of column values. Since using this smell affects the readability of SQL code, we classify the antipattern as one of the maintainability antipatterns. When adding and removing values, developers become uncertain about which column is NULL or not, so they have to retrieve this row and check by themselves at the application level. The database also cannot help ensuring uniqueness when developers use a multicolumn attribute.

?? Creating a dependent table with one column for the multivalue property is the best way. Numerous values should be stored in multiple rows rather than multiple columns. For instance, a dependent table can be named as colors. It contains two attributes such as car_id and color. The primary key is the combination of the two attributes. You can insert colors for a specific car by executing this query: INSERT

INTO Colors (car_id, color) VALUES (1010, 'gray'), (1011, 'black'), (1011, 'blue'); searching for cars with multiple colors becomes easier using this dependent table.

2.4.0.8 Clone Tables:

Definition: It is also called sharding or horizontal partitioning. Developers create clone tables when they manually split one table horizontally to support scalability without using the capabilities of DBMSs. This antipattern is bad because the number of tables or columns continue to grow, since new data values can make developers create new schema objects . For instance, partitioning can be done based on a date attribute, so data is inserted into tables based on the value year.

Impact on performance: Those clone tables might contain large data, and developers might need to query all clone tables at once; thus, performance becomes worse for any query as the volume of data increases. DBMSs cannot prevent developers from inserting into wrong clone tables, but they can define check constraints to verify whether data is added in concert with a date. However, some DBMSs, such as MySQL, do not support check constraints, so developers have to define them in their DDL query.

Impact on maintainability: For synchronizing data, developers might change the value of the date attribute, making it an invalid row for this table, so they have to move the abnormal row from one table to another by updating the row and inserting it into its applicable table. In order to ensure uniqueness, developers must create a table that has a super serial key that is unique across clone tables. Querying across tables is cumbersome as time goes on because developers need to create new tables as the year, which is the partitioned attribute, progresses, so they need to revisit their application to update written queries to have the newly created table. This results in an added level of the chore of maintaining the co-evolution between source programs and the

database. Splitting a table into many identical tables complicates maintenance tasks [32] because creating many identical tables is a maintenance nightmare. This is the reason why we categorize this antipattern into maintenance antipatterns.

Impact on portability: As discussed in the impact on maintainability, To query across tables, developers union all clone tables and query that as a derived table. Relying on Union to query across all clone tables is not always a good solution because not all DBMSs, such as MySQL, support Union. We categorize this antipattern into portability antipatterns.

Impact on data integrity: Developers cannot obtain referential integrity when clone tables have a dependent table because of the aforementioned reason that the foreign key cannot reference more than one parent. The absence of referential integrity enforces developers to enforce it manually by adding checking constraints, so when a check constraint is missing, data integrity is at stake.

Solution: An alternative to dividing the table into clone tables by developers, horizontal partition, vertical partition, and dependent tables are solutions to clone tables antipattern. Horizontal partitioning (also known as sharding) is done by creating a logical table with specifying how to form a partition, i.e., using hash partitioning, range and etc. and how many partitions a table is going to be horizontally distributed across disk volumes. Sharding is recommended when there are write scaling problems. Sharding address specific use cases to improve performance. One use case is cleansing old data by partitioning them into separate tables to improve performance [32]. Vertical partitioning splits the table based on columns when some columns are large such as TEXT or BLOB or rarely required such as variable-length strings. Developers can access some columns efficiently if they execute a query without projecting these columns. They might sometimes use select * wildcard to query all columns without recognizing that one column can have large a file that is seldom needed at the time

of the query which is intentionally affecting performance. The solution is to store seldom needed columns outside their current table. For instance, MySQL's MyISAM storage engine queries a table more efficiently when the records contain fixed size, so columns with VARCHAR data type should be stored in a separate table because these columns affect performance.

2.4.0.9 Values in Attribute Definition:

Definition: This antipattern arises when developers restrict a column to specific values. For instance, a student table might have a status column that indicates whether a student is FULL-TIME, PART-TIME, ON-LINE, and so on. It is not a good practice to mix meta-data with data. Developers define such columns using a check constraint.

Impact on performance: To get all possible values of the status column, developers need to query its definition from the metadata. When using the ENUM data type, developers have to parse the result set that is returned by the information schema. When changing the list of possible values of the status column, they have to write an ALTER statement to redefine the column definition. This results in a large performance penalty since it is required to lock and rebuild the table.

Impact on maintainability: As discussed in the impact on performance and portability, because ALTER statements are not supported by all DBMSs, developers need to drop the table, recreate the table with an updated definition, and insert the data again. If a table is referred by other entities, developers need to drop them too before they drop the table. This complicates the task of maintaining an attribute definition. As a result, this is why we classify this antipattern under maintainability anti-patterns.

Impact on portability: As discussed in the impact on performance, the ALTER statement is used to modify the attribute definition. However, the ALTER statement is not supported by all DBMS. As a result, This anti-pattern affects portability. To resolve the issue of being importable, developers have to drop the table, redefine its definition and import its data [33].

Solution: Some DBMSs support a nonstandard data type called ENUM. Other solutions exist, such as domains, user-defined types (UDT), and using triggers.

2.4.0.10 Index shotgun:

Definition: This antipattern occurs when developers use the indexes inefficiently. Index misinterpretations can happen in three ways: 1) using no indexes or insufficient indexes 2) using too many indexes or indexes that do not help 3) executing SQL statements that no index can help.

Impact on performance: Even though using the indexes brings overhead, they have their benefits. Once developers create the B-tree, a data structure that implements the indexes, the DB retains the index automatically. When developers execute INSERT, UPDATE, or DELETE on a given table, the DBMS must keep track of the index data structures for it to be consistent. As a result, their subsequent searches utilize the index structure to get the requested number of records efficiently and keep the tree in balance. Adding indexes to the database gives no guarantees that the performance of the database is going to increase, so it is a tradeoff that needs to be considered [34]. Being a victim of index shotgun forces developers to have queries that last longer times than usual because they might not have indexes or they have slow indexes. Thus, it affects the performance of the DB.

Impact on portability: As discussed in the impact on performance, developers can use query execution plan (QEP) to monitor their queries to check to see

how the DB optimizer chooses to use indexes. This antipattern also affects the portability factor because QEP and database profilers that help optimize the queries are vendor-specific.

Solution: The solution to this antipattern is having good judgement about whether to have indexes or remove them. Karwin outlines a checklist for applying one's discretion which includes measure, explain, nominate, test, optimize and rebuild. Measuring your database operations is important because it lets identify one query that takes most time to execute. After developers identify such query, they need to explain why it causes overhead by using optimizer for any database vendor. The optimizer gives query execution plan (QEP) for the selected query. Then, the developer needs to analyze the report to come up with a strategy. Nominating which cases of the query access the table without indexes is the next step. After developers decide to use index on any query, they need to test and evaluate changes. Optimizing and rebuilding are used to enhance the usage of computing resoruces.

2.4.0.11 Denormalized/god Table:

Definition: This antipattern occurs when developers mistakenly put all attributes in one table and avoid normalizing the table to minimize redundancy. Software developers coin this as a god table or universal table since it includes all attributes even though this well-known concept is called denormalization in the database community.

Impact on performance: This denormalization process might result in an update, insert, or delete anomalies, which in turn bring unnecessary updates to the table that affect the performance of database systems. Denormalization can improve database performance when developers apply it as an additional step between physical and logical designs of the database and use in tandem with application requirements

[35, 36]. As a result, denormalization takes effect after developers normalize their database designs. Developers need to balance between normalization and denormalization because write-intensive operations require normalized data across tables in different locations, while read-intensive operations need de-normalized data to avoid joins that minimize the performance of the database.

Impact on maintainability: As discussed in the impact on data integrity, even though denormalization increases performance, it makes the database vulnerable to data inconsistencies. Developers sometimes write application code that prevents violations of data integrity. When it occurs, they need to manually fix these inconsistencies by changing data values that belong to different tables. This process affects the maintainability factor of the database system. Generally, a normalized database is more maintainable and stable than a denormalized one.

Impact on data integrity: As discussed in the impact on performance, the denormalization process might result in an update, insert and delete anomalies which in turn bring inconsistencies to database tables. Non-normalized tables have duplicate data that developers store in different locations. Developers usually prevent such inconsistencies by implementing logic in the high-level application. Developers may likely update one value without updating the rest of the copies in other tables. When it occurs, developers introduce inconsistency to one part of their database [37].

Solution: Views or table-valued functions can usually take their place [3]. Although indexed views have a higher maintenance cost, they are far preferable to denormalization.

2.5 Categorization of SQL Antipatterns based on Query

In this section, we define all 6 *SQL query antipatterns* in a separate section for each, and then we analyze and categorize their effects on performance, maintainability,

portability, and data integrity. Table 2.4 shows the result of the classification. We find that 4 SQL query antipatterns are performance antipatterns. Out of 6, we categorize only one SQL query antipattern into maintainability antipatterns. Two antipatterns affect the portability factor of software systems. We classify one SQL query antipattern as data integrity antipatterns.

2.5.0.1 Reference Nongrouped Columns

Definition: This antipattern appears when developers' goal is executing a query that includes an aggregate function such as MAX(), MIN(), and AVG() to get an aggregate value in a group and other columns of the record where that value is returned. Listing 2.1 shows an example of an incorrect SQL query that results in an error or unreliable answer in MySQL or SQLite DBMSs. The antipattern occurs when a non-aggregated column is not used in the group by clause. Two students can have the same name, so it is a nonaggregated column, and it violates the single value rule, whereas student IDs are unique for each student.

We recommend developers to follow the single-value rule that all columns within a select clause of the query need to have a unique value per record group. The aggregate functions are guaranteed to end in a single value for each group. However, developers fall into the misconception that SQL will pick a value for attributes that are not followed by a group by clause. They, for instance, might think that the DBMS will have a single value per group for courseId in as in listing 2.1. The DBMS cannot guarantee that any column mentioned in the select clause will have the same value stored in every row for one group.

Portability Impact: This antipattern has an impact on portability because each database vendor deals differently with this antipattern. Database vendors send error messages when developers write database queries that violate the single-value

rule. If a column is neither listed in an aggregate function or group by clause, it technically violates the rule. Some database brands figure out functionally dependent columns when developers group them by a primary key column. For instance, SQLite and MySQL allow such ambiguous groups when developers group by a primary key because these database brands can determine dependencies on the fly.

Solution: Querying functionally dependent columns and using a correlated subquery, derived table, and join are ways to solve this antipattern. Developers need to measure the performance of some solutions. Querying functionally dependent columns which is the easiest form of solution is done by removing ambiguous columns from the query. For listing 2.1, developers should remove courseId from the query. An associated inner query includes a reference to the higher-level query, so it produces various outcomes for each record of the higher-level query [2]. However, this solution may not be best for performance because they are just hidden cursors run once for each record of the high-level query. Developers can utilize an inner query as an extracted table, which creates an interim result. Then, they rely on this result for joining against tables, but the database must save the temporary result set in a short-term table, so this solution is still not good for performance. Developers should use window functions instead as they usually carry out the same operations much faster, or they should replace them with JOIN queries because they are simpler and faster.

```
import java.sql.*;
public class SimpleDemoSQLJ
{
  public Void NonGroupedColumn()
    throws SQLException
  {
```

```
    #SQL Iterator StudentIter (Integer, Float, Integer);

    StudentIter sti;
    #SQL sti = {SELECT studentId, avg(GPA) as MyGPA, courseId
                FROM StudentGrades
                GROUP BY studentId;}

    While(StudentIter.hasNext()){
    System.out.println(sti.studentID() + " " + sti.name());
    }
  }
}
```

Listing 2.1: Example of Reference Nongrouped Columns illustrated in SQLJ

2.5.0.2 Joining Data in Memory

Definition: DBMSs offer JOIN as a SQL query to easily and efficiently combine two sets of data, but developers might not choose it to carry out their tasks for many reasons and create a solution that fetches the first and second sets of data separately to the database client and builds an algorithm using a programming language that joins those two data sets into one within a PL's main memory. Performance, scalability, memory utilization, and maintainability are reasons why this approach is problematic [38].

Performance Impact: It is slow because network bandwidth is an issue when developers transfer a large amount of data twice over a network to perform the join. Also, the data mass is another reason that adds a burden to the network. The

larger data is being transferred, the more time the network needs to handle it. The main memory also would not handle such large data. The implementation of this antipattern that needs to be fixed is scattered across the system; thus, it hurdles the refactoring process of this antipattern.

Solution: When this antipattern occurs and results in problems as discussed in the impact on performance, solving this antipattern causes the system to be unusable; thus, it leads to loss of customers. The solution of this antipattern is not straightforward and requires multiple testing and fixing cycles. This approach does not only affect performance but also maintainability and complexity because developers build everything from scratch and they have to maintain it.

2.5.0.3 Buried Null

Definition: Forgetting or misusing Null in any DBMS is considered as a smell. DBMSs respond and behave differently when it comes to Null. Unlike programming languages, Null is a special value that is different from zero, false, or an empty string. Listing 2.2 shows an example of arithmetic operation with a major attribute that is Null. The DBMS should return NULL. Another example of this smell is when developers equate any variable to Null using an equal sign '='. They should use the phrase: is Null, instead.

Maintainability Impact: Many developers choose not to adhere to this guideline. Treating Null as a special value solves many unseen problems, but if they desire to choose another way to represent the unknown, they have to make their way to settle changes. For instance, let us say they choose 1 to represent an unknown value. They restrict themselves to this numeric number. They cannot use aggregate functions anymore, such as AVG() and SUM(), if one record contains 1. The value 1 may be important in another column, so they select a different value on a case-by-case

basis for each column; thus, they have to document and do unnecessary transactions that affect the DBMS maintainability.

```
SELECT major + 10 from Students;
```

Listing 2.2: an SQL query

Solution: Developers should think of NULL as unique values. Developers are used to traditional two way logic in programming, whereas in SQL, they should think using three way logic including NULL as an additional value. NULL is neither TRUE or FALSE. Database vendors define predicates for searching for NULL because neither equality or inequality with NULL returns expected values.

2.5.0.4 Using UNION or DISTINCT to Remove JOIN Duplicates

Definition: Developers can witness JOIN duplicates for two reasons. They may fall into the misconception that it is guaranteed to have the same number of rows in a left table when they use a left join. It is guaranteed to have at least as many rows as the left table contains. Developers also might forget to add JOIN or ON predicates when multi-column foreign key relationships are included in the SQL statement. For both reasons, they might use UNION or DISTINCT to remove duplicate rows. In addition to overstating results, this antipattern will not solve the problem, but the symptoms. This antipattern is a mistake that developers can make when using Java with JDBC or JOOQ.

Performance Impact: Using UNION or DISTINCT with many columns and hundreds of records is slow. Using DISTINCT requires using ORDER BY that re-orders all result sets to remove duplicates. This antipattern is also categorized as a performance antipattern. Using UNION or DISTINCT might solve the symptoms

but not the issue. It might not solve the symptoms in edge-cases. Specifically, some duplicates might not be deleted when developers use UNION or DISTINCT.

Data Integrity Impact: As discussed in the impact on performance, developers misunderstand that using UNION or DISTINCT is guaranteed to have the same number of rows in a left table when they use a left join. However, they might get the same number of rows, but they are not identical rows. This antipattern affects data integrity. Using UNION or DISTINCT might solve the symptoms but not the issue. It might not solve the symptoms in edge-cases. Specifically, some duplicates might not be deleted when developers use UNION or DISTINCT.

Solution: It is better if developers determine why the join query generates duplicate rows without using UNION or DISTINCT to delete them. Then, they should be able to fix the problem.

2.5.0.5 Pattern Matching Predicates

Definition: When any application stores documents or texts, it might offer a search feature for its users. Developers use SQL to store a large amount of data, but they may find difficulties when they try to offer the data at greater speeds via the search function. This antipattern occurs when developers choose pattern matching predicates, a feature, offered by SQL to search for texts. The Like operator is the most widely supported among pattern matching predicates which accepts a wildcard (%) that equals zero or more characters. Listing 2.3 shows an example of how to search for a text using pattern matching. The search keyword is SQL and it comes after the LIKE construct. Then, it is surrounded by wildcard signs.

Performance Impact: Pattern matching operators incur poor performance because they cannot benefit from a traditional index. Even so, they may not find the intended search results. However, if queries that are built for full-text search are a

few, sometimes developers do not need to optimize for performance since maintaining an index is costly.

```
SELECT *
FROM databases
WHERE name LIKE '%SQL%';
```

Listing 2.3: an example of using Like predicate in SQL

Portability Impact: Pattern matching predicates is well suited for simple cases, and it does not yield good results with hard ones. We recommend to avoid SQL and use a specialized text search engine technology that is supported by different vendors or a third party. If developers do not want to use those options, they can build an inverted index, which is a list of all words that one might search for and save search keywords in a table. The index associates these words with the text entries that have the respective word in a many-to-many relationship. For instance, a keyword such as SQL can exist in many database names, and every database name may have other tokens.

Solution: Developers should use vendor extensions for full text search capability or implement their own search functionality. These vendor extensions are not equivalent and offer different features. On the other hand, implementing search functionality can be done using inverted index [2].

2.5.0.6 Random Selection

Definition: This antipattern occurs when developers want to fetch a sample row from the database. The most popular SQL stratagem to choose an arbitrary record from the result of a SQL query is to put the query in order randomly and choose the first sorted row. Listing 2.4 shows an example of such a query.

Performance Impact: Listing 2.4 has a vulnerability because it uses RAND(), as a nondeterministic expression that sorts without respect to the index. This is a problem because the query does not use an index, and the DBMS does a table scan, instead which manually sorts all rows by saving them into a temporary table and swapping them physically. An index-assisted sort is much faster than a table scan sort, and the performance increases proportionally to the size of the data set. We categorize this antipattern as one of the performance antipatterns.

```
SELECT *
FROM Students
ORDER BY RAND() LIMIT 1;
```

Listing 2.4: an example of fetching random row in SQL

Solution: Developers should rely on the chosen primary key and choose its value randomly using a variety of techniques as suggested by **??**. They can also count number of the rows and choose a random row as an offset between zero and count.

2.6 Categorization of SQL Antipatterns based on Security Breaches

In this section, we define and classify 3 SQL security breaches based on the four nonfunctional requirements: performance, maintainability, portability and data integrity. Table 2.5 shows the result of the classification. We classify all of them as performance antipatterns. We also identify 3 SQL antpatterns as antipatterns affecting data integrity. None of these antipatterns have an impact on maintainability and portability of the system factors.

Table 2.4: Categorization of SQL Query Antipatterns.

Reference	Antipattern	Performance	Maintainability	Portability	Data Integrity
[2],[27]	Reference Nongrouped Column			✓	
[38],[20]	Joining Data in Memory	✓			
[2], [3], [27]	Buried Null		✓		
[20]	Using Union or Distinct to Remove Join Duplicates	✓			✓
[2]	Pattern Matching Predicates	✓		✓	
[2]	Random Selection	✓			

2.6.0.1 Vulnerable Query

Definition: This antipattern becomes apparent when unsanitized input is passed to a query at run time. For instance, when an input string comes from an unreliable user and is appended to a database query, this final query would be categorized as vulnerable query. This antipattern is bad because it affects the confidentiality, integrity and authorization of the database system [39].

Performance Impact: The findings of Lyn's work [21] revealed that cleaning the sensitive input can increase runtime and energy usage by 18% and 16%. However, their analysis indicated that the cost was quite low. For the most part, the improvements in performance did not reach statistical significance. Calling the sanitiazation API along incurs performance overhead alone when authors want to secure the concatenated query. For these reasons, we categorize this antipattern as one of performance antipatterns.

Data Integrity: This query has an impact on data in general. if one attacker is able to gain access using such query, the stakes will be high because data might be lost or altered. One reason is enough to place this antipattern into antipatterns affecting data integrity.

Solution: This antipattern can be solved using sanitization or parameterized query. The authors sanitized the input query using encoding schemes from the OWASP Enterprise Security API (ESAPI) Toolkits [40]. The encoding routines clean the provided input and use the database's suitable escape technique. The DBMS will not mix up the sanitized input with the developer's SQL code, preventing SQL injection vulnerabilities. Karwin [2] explained how the antipattern can be solved using parameterized queries although Lyn et al [21] prefers sanitizing the input because parameterized queries incur performance overhead.

2.6.0.2 Readable Passwords

Definition: Some applications give users the option to reset their password by requesting an email that shows their password in clear text. This is a serious database design issue that causes a number of security risks, including the possibility of unauthorized users gaining privileged access to the application. This antipattern occurs when a password is stored as a string literal in an insert or update statement. For instance, "SELECT * FROM Users WHERE user id = '123'AND password = 'mypassword'".

Performance Impact: Lyn et al [21] showed that this antipattern has a slight impact on performance. The runtime and energy were increased 0.5% and 0.1% on average. This is the reason why we categorize this antipattern as performance antipattern.

Data Integrity: Exposing passwords and storing them in a database allows attackers to search for readable password in logs and steal them. Then, attackers can have access to a target system by utilizing this readable password and gain access to data which makes data integrity at risk.

Solution: To fix this antipattern, developers should use hash and salt to hide the readable passwords before keeping it in the dataset as suggested by Karwin [2].

Table 2.5: Categorization of SQL Antipatterns based on Security Breaches.

Reference	Antipattern	Performance	Maintainability	Portability	Data Integrity
[21, 2]	Vulnerable Query	✓			✓
[22]	Unbounded Query	✓			✓
[41]	Readable Passwords	✓			✓

CHAPTER 3

Framework-Specific Antipatterns (Drafted from [1])

In this section, we review the literature on antipatterns that occurred in various application domains. We first discuss problems of misusing Active Record pattern, Model View Controller (MVC), and Service-Oriented Architecture (SOA) antipatterns in section 3.1. These antipatterns are architecture-specific and cause different types of problems. We find a couple of antipatterns and the target domain in which the antipattern appears. Linares-Vásquez et al. [13] shows that it is possible to discover new antipatterns when developers test a different framework. We then review ORM related performance antipatterns in section 3.2. We find antipatterns, relating to data manipulation language (DML), main memory-related, database access (such as Hibernate), and associations between classes.

3.1 Architecture-Specific Antipatterns

Several research studies on design antipattern characterization and detection are domain-specific. In this section, we list architectural software antipatterns. One study addresses antipatterns that are specific to **a web MVC architecture** [41]. They conducted a layer-focused survey, role-focused survey, and unstructured interviews with industrial developers to find the following system architecture antipatterns for web MVC architecture:

- Brain repository antipattern, AKA "Complex Logic in the Repository": This specialized object is allocated to help persist other objects in the database using SQL or JPQL (Java's JPA query language). However, when a repository class

has complicated business logic or even complex queries, some survey participants confirm this class as a smelly class.

- Fat repository antipattern, AKA "a Repository managing too many entities": An Entity and a Repository have a common one-to-one relationship, so developers should have a separate repository per class domain. If a repository deals with more than one entity at once, this might lead to low cohesion and maintenance problems.
- Laborious repository method antipattern, AKA "a Repository method having multiple database actions": if a single method sends more than one database request, it would be an antipattern since it affects the method readability.
- Meddling service antipattern, AKA "Services that directly query the database": Developers should place in repositories instead of services because services contain code that implements business rules and relationships among domain classes.

We believe these antipatterns relate to Repositories or Data Access Object (DAO) classes within MVC pattern [42] because these classes are responsible for dealing with the communication towards the databases in MVC applications. We find that other antipatterns identified in [41] such as promiscuous controller and brain controller are not related to problematic database code, so we omit them from the list. The authors confirmed those antipatterns with an industry expert in Spring MVC as problematic. They also propose detection strategies and implement a tool that can find these antipatterns using those detection strategies.

Another study provides two catalogs of service-based antipatterns for two architectural styles, including service component architecture (SCA) and web services of service-based systems (SBSs) based on **Service Oriented Architecture (SOA)**.

Palma et al. [43] classify service antipatterns that are prevalent in SCA or web services. SCA abstracts away different communication standards that service consumers can call via APIs such as JSON or SOAP. They classify antipatterns based on whether they are intra-service or inter-service. They also classify antipatterns by whether the antipatterns are static, dynamic, or both. The catalog in SOA includes the following database-related web service antipatterns:

- Chatty Web Service.
- CRUDy Interface.
- Data Web Service.
- Maybe It's Not RPC.

We believe the link between Chatty Web Service antipattern and the database is the fact that Chatty Web Service is an anemic object or model that contains numerous attribute values, also known as setters and getters, and lacks logic. It is the class in a database. CRUDy Interface is derived from Chatty Web Service. It is related to database because the antipattern involves the basic database operations such as create, read, delete and update. In a later section, we categorize and explain the CRUDy Interface, which is a mapping between database entries and objects, and also Chatty Web Service, and Maybe It's Not RPC antipatterns. The catalog also lists the following database-related SCA antipatterns:

- Sand Pile.
- Chatty Service.
- Data service.

We believe the link between sand pile and database is that when Sand Pile occurs, its service comprises of more than one service components sharing common data. Since these service components access the same data, we consider this antipattern as problematic database code. Chatty service refers to the fact that services exchange a

lot of small data. We also include this antipattern to database related antipatterns because this antipattern sends and receives data.

Bogner et al. [44] create a comprehensive repository of service-based antipatterns for both **SOA** and **Microservice Architecture** systems. They review 14 research papers in the literature and distilled 36 SBS related antipatterns from them. Bogner et al. [44] classify antipatterns into two dimensions: the abstraction level of their applicability and influence, and the domain in which antipatterns have an impact such as architecture, application, and business. Their categorization differentiates antipatterns that belong to either SOA, MA, or both. They generate a JSON schema and utilize it to record these antipatterns. Antipatterns found in the previous study are included in the work of [44]. Bogner et al. [44] study additional antipatterns related to persistence mechanisms. These antipatterns are data-driven migration, on-line only, shared persistency, and transactional integration.

Another similar study creates a catalog of smells on cloud-native applications based on microservices. Taibi et al. [45] analyzed 265 bad practices conducted by 72 experienced developers to produce a catalog of 11 antipatterns. Inappropriate service intimacy and shared persistency are database-related antipatterns of the catalog. They apply an open and selective coding procedure [46] to obtain the antipattern catalog from the developers' opinions. Taibi et al. [45] consider practice as an antipattern if it degrades characteristics of software quality such as maintainability, extensibility, understandability, reusability, and testability of the developed system. They create a description for each identified antipattern, propose a detection strategy, and explain the problem that it may cause along with its adopted solutions.

We shall explain and classify the architecture-specific antipatterns in more detail in sections 2.4 and 2.5.

3.2 ORM Antipatterns

Many authors have investigated object-relational mapping concerning software design. From the point of the pragmatic programmer's view, Karwin [2] provides an overview of SQL design anti-patterns, but he explicitly spots one antipattern that is related to ORM. He only studies the active record antipattern. The active record antipattern mirrors an object to a given row in a relational table. Developers usually deal with the active record antipattern when they develop a web application using a web application framework based on the model-view-controller (MVC). Being a popular architectural design pattern for web applications, MVC separates concerns for better reuse of components, but when developers treat the M of the MVC as the active record pattern, it becomes an antipattern, and three issues occur as a result of adopting this approach:

- Models and database schemas are tightly coupled. Whenever a schema evolves, the model reaches an inconsistent state.
- A class with CRUD operations (CREATE, READ, UPDATE, DELETE) reveals those operations to any children classes inheriting from it, permitting direct database access, minimizing cohesion.
- Active record encourages anemic data model, an antipattern identified by Martin Folwer [47]. Active record classes exclude business logic implementation and include only data access objects without behavior. The active record pattern can be an antipattern if developers misuse it.

Karwin [2] excludes performance issues because the scope of his book is to classify missteps people make while working with SQL.

Some work concentrates on how developers implement ORM code. Chen et al. [19] show that ORM anti-patterns involve repetitive structures such as loops, which are considered as memory-efficient in the source code but not for the database access

part. Being able to be optimized by SQL engines, set-based queries usually promote productivity and communication. Chen concentrates on two ORM performance anti-patterns: the eager fetching problem and the row-by-row problem [19]. We will further analyze and categorize these antipatterns in 3.4.1 and 3.4.2 .

In a later work, Chen et al [48] document 5 framework-specific antipatterns of database access code. They implement these 5 antipatterns to their developed static bug detection tool, called DBChecker [19]. These antipatterns are the source of both functional and non-functional problems of the studied industrial systems. They analyze a bug using its impact on the system, which is functional or non-functional, its description, an example, and the developer's awareness towards it. With the help of developers' feedback, they found the following database access antipatterns in Hibernate:

- Nested transaction.
- Unexpected transaction behavior.
- Inconsistent transaction read-write level.
- Sequence name mismatch.
- Incorrect SQL orders.

Yang et al. [22] made a comprehensive study of 12 ORM applications which led them to generate 9 generic ORM patterns that address the performance and scalability of web applications. We omit the Application Functionality Trade-Off (FT) and Content display trade-offs (DT) from the categorization because we only focus on database-related antipatterns. The following is a list of 7 database-related antipatterns that they identified:

- Inefficient computation (IC).
- Unnecessary computation (UC).

- Unnecessary data retrieval (UD): Four types of redundant data problems are analyzed in Chen's study [49] as follows:
  - *Select all.* ORM query language, i.e., Hibernate Query Language (HQL), select all columns of a database table even though developers select only some attributes from an entity class.
  - *Update all.* Once developers update some object attributes of a class, the ORM updates all associated columns of an object attribute.
  - *Excessive data.* Unlike select all, excessive data is concerned with querying associated entities (more than one entity that participates in an association such as @ManyToOne, and other types).
  - *Per-transaction cache.* ORM queries are repeatedly generated for each transaction to get a result even though the result is not changed. Such queries are redundant across transactions and, their results can be found in a cache.
- Inefficient rendering (IR).

  When a view file renders a set of objects, it indicates a balance that must be taken into consideration between performance and readability .
- Missing fields (MF):
- Missing Indexes (MI):
- Inefficient data accessing (ID).

The causes of inefficiencies of these antipatterns are database design problems, application design tradeoffs, and ORM API misuses. Instead of analyzing the client-side, Yang et al [22] analyze the performance issues on the backend because they suppose it is a solution to having efficient ORM applications.

We shall explain and categorize these ORM antipatterns of database access code in section 2.5

3.3 Categorization of Framework-Specific Antipatterns based on Schema

In this section, we describe and categorize 4 *framework-specific schema antipatterns* based on the underlying four system factors of performance, maintainability, portability, and data integrity. Table 3.1 shows the result of the classification. We identify 3 schema antipatterns as performance antipatterns. Two of these antipatterns are ORM antipatterns, while the remaining antipattern is architectural. Out of the 4 studied antipatterns, we categorize 3 schema antipatterns into maintenance antipatterns. Two of these antipatterns belong to ORM antipatterns, and the remaining antipattern is an architecture-specific antipattern. We categorize one schema antipattern into portability and data integrity antipatterns. We define all the schema antipatterns in a separate section for each and then discuss their effects on performance, maintainability, portability, and data integrity.

3.3.1 Shared Persistency/data Ownership:

Definition: Various microservices communicate with the same database or even worse, they access the same entities of the same database without any concurrency control [45]. This is also proposed as a data ownership antipattern. Various services should not directly share data unless the database has strong atomicity, consistency, isolation, and durability (ACID) properties, and they should keep their data private and accessed via APIs [50]. Richardson term this antipattern as a shared database antipattern.

Impact on performance: All services accessing the same database may impede with one another [51]. For instance, one service might execute a long transaction that places a lock on any table of the shared database. It degrades the overall performance of the DBMS, so we classify it as one of the performance antipatterns.

Impact on maintainability: As discussed in the impact on performance, services access the same relational database, or even in the worst scenario the same entities of the same relational tables. This antipattern increases the coupling between services and lessens team and service independence. Working on different services that share the same database results in schema changes that need to be maintained and coordinated between developers. Maintaining schema changes for a shared database between services is more troublesome than maintaining a database per service. This antipattern complicates maintenance, and that is why we categorize the antipattern into maintenance antipatterns.

Solution: Database per service is another approach that eliminates coupling between services. Unless developers prefer ACID guarantees that are offered by the DBMS, they should choose the other approach.

3.3.2 Active Record Anti-pattern:

Definition: The active record design pattern as a persistence layer that reflects an object to a specific record in RDBMS. Being a popular architectural design pattern for web applications, model-view-controller (MVC) separates concerns for better reuse of components, but when developers treat the M of the MVC as the active record pattern, it becomes an antipattern because Active Record couples models to the schema, exposes CRUD functions and encourages anemic domain model [47], treating models as simple access objects.

Impact on performance: Because rows in a database and objects in an OOPL are not loosely coupled, the database needs to be statically connected most of the time. This results in performance bottlenecks when thousands of users establish many databases hits from any domain business model [52].

Impact on maintainability: Using the active record antipattern increases the coupling between objects and database entities. Developers have to maintain the database and objects at the same time; it complicates the maintainability of the system.

Impact on portability: Since the active record antipattern increases the coupling between objects and database entities as we discussed in the impact on maintainability, the DBMS becomes harder to decouple itself from the application program. Thus, developers cannot switch their DBMS vendors i.e., from MySQL to Postgres, affecting the portability of the DBMS.

Impact on data integrity: Domain entities know about their own persistence in software systems implemented using such a pattern. Objects of domain entities tend to be more like SQL objects [52]. Using the active record antipattern reveals CRUD operations to other programmers in the team because they can have access to the database (table) from any entities. They might utilize the model class to bypass some intended usage that violates business requirements. It affects the data integrity of the underlying DBMS.

Solution: The solution is to avoid misusing Active Record pattern. As suggested by [2], understanding the model by decoupling domain model classes from DAO classes, consulting with information expert, utilizing controllers and views to use domain classes and disconnect their interactions from databases and putting the model into action help solve the Active Record antipattern.

3.3.3 Fat Repository/Generic Repository:

Definition: Fowler [42] defines a repository as the intermediary that separates the domain and data mapping layers and is represented as an in-memory domain object collection. In other words, a repository encapsulates persistence logic. This

antipattern occurs as a result of relating a repository to many entities. In other words, developers end up having a generic repository interface that consists of methods, such as get, update and save. They think that it applies to all entities of their business domain. They implement the interface by creating a generic repository class for all entities. An entity should be related to an equivalent repository class that implements its specific repository interface which is unique and part of the domain.

Impact on maintainability: Listing 3.1 shows an example of a generic repository. T can be any type of object. Using such a repository causes low cohesion and makes maintenance harder because it does not adhere to a contract between domain objects and data stores. Instead, a generic repository passed as a dependency to entities excluding specific database queries and tend to have CRUD operations imitating the job of ORM [53]. It makes the generic repository a useless abstraction. The ORM is responsible for the generic query mechanism. The repository pattern addresses specific queries for the business domain; thus, a repository class is a part of the domain model. It is not easy to work with composite keys with a generic repository.

```
public interface IRepository<T>
{
    IEnumerable<T> GetAll();
    IEnumerable<T> Find(Expression<Func<T, bool>> query);
    T GetByID(int id);
    void Add(T item);
    void Update(T item);
    void Delete(T item);
}
```

Listing 3.1: an example of using generic repository in Java

Solution: The proposed solution is that developers should use one repository per an entity, so they should create one repository for each entity. This would actually cause an increase in cohesion and solve maintenance problems.

3.3.4 Missing Fields (MF):

Definition: This antipattern concerns the decision on what object fields are persisted in the database. Missing fields are attributes that are selected to be transient where they should be persisted by the application. If developers choose some transient object fields or attributes, they can decide that they do not these attributes after executing the application. Developers can derive these transient attributes from other persisted attributes using some way of calculation, which can be expensive if repeated many times.

Impact on performance: Yang et al [22] show that transient attributes incur computation costs when authors derive them. The location_name of a string diary in Openstreetmap [54] can be derived from using latitude and longitude attributes. Developers can cache the result of the computation, but using a cache is not recommended, which is confirmed by SO developer, using core data framework in Swift because it cannot lead to performance gains [55]. Instead, we can persist in transient attributes because we avoid the need to recompute the result every time developers request access to such transient attributes. Yang et al [22] explain why developers need to eliminate unnecessary computation overhead by storing the result of location_name of the diary in the database. This antipattern affects the performance of a software system.

Solution: Profiling software systems is critical when execution time takes longer than expected. Developers might create a correlated column for a database table that is as a result of using other columns in the table. Removing such columns

enhances the time it takes to execute any operation related to the modeled table. For instance, removing location string from Openstreetmap [54], which is created as a result of using longitude and latitude columns in the same table, reduces execution time for the overall software system.

3.4 Categorization of Framework-Specific Antipatterns based on Query

In this section, we define all 21 *framework-specific query antipatterns* in a separate section for each, and then we analyze and categorize their effects on performance, maintainability, portability, and data integrity. Table 3.2 shows the result of the classification. We found that 13 query antipatterns could be classified as performance antipatterns. Two third of these performance antipatterns are ORM antipatterns, and the remaining are architecture-specific antipatterns. Out of 21, 9 query antipatterns are categorized into maintainability antipatterns. More than half of the maintainability antipatterns are architecture-specific antipatterns. A few antipatterns affect the portability factor of software systems, and all of them are architecture-specific antipatterns. Some antipatterns are found to have an impact on data integrity. Only one of these antipatterns belongs to ORM antipatterns, and the rest are architecture-specific antipatterns.

3.4.1 The Eager Fetching Problem

Definition: The Eager Fetching Problem, also known as Excessive Data, brings unnecessary associated data with the queried object, and when the anti-pattern is alleviated, the system under study exhibits 71% increase in performance [19]. Listing 3.2 shows that a university class is associated with a student class (access to the university class will lead to access to the student class) and includes students as a collection of instances that are eagerly loaded. Listing 3.3 shows if we run Excessive.java, using

Table 3.1: Categorization of Framework-Specific Schema Antipatterns.

Reference	Antipattern	Type	Performance	Maintainability	Portability	Data Integrity
[45, 50]	Shared Persistency/Data Ownership	*Architecture*	✓	✓		
[2]	Active Record	*ORM*	✓	✓	✓	✓
[41]	Fat Repository/Generic Repository	*Architecture*		✓		
[22]	Missing Fields (MF)	*ORM*	✓			

ORM configuration that is set in Listing 3.2, the execution of this program would create database access code to extract university objects and students objects from the database because the fetch type is set to the EAGER setting on the student instance variable in University.java, although we do not use Student objects in Excessive.java. We categorize this antipattern into SQL query smells since an ORM generates SQL queries from OOPL code and using either annotations, declarations, or a separate XML file for the persistence mechanism.

Performance Impact: The author [19] confirms the performance impact by having unnecessary data retrieval in Excessive.java as illustrated by listing 3.3. They insert 300 rows into one table and 10 rows for each one of those 300 rows in the associated table. The response time before repairing Execssive.java is 1.68 seconds, while repairing the performance antipattern brings down the response time to 0.48 seconds, leading to a 71% performance improvement. As a result, this antipattern is categorized as one of the performance antipatterns because it drastically affects the system performance running such code patterns. The N+1 selections problem has received quite attention in the industry. Munhoz [56] explains why loading initially the full collection is not advised. Loading the full collection would waste a lot of resources unnecessarily and would maximize lock time in the database which can have a direct influence on the concurrency of the application.

Solution: Changing the fetch type from EAGER to LAZY (the original setting in University.java) is one method to remedy this performance anti-pattern as suggested by Chen et al [19]. The author conducts the experiment of having 300 rows to analyze the proposed solution by decreasing the response time from 1.68 seconds to 0.48 seconds.

```
@Entity
```

```
@Table ( name = "university")
public class University{
  @Id
  @Column(name="university_id")
  private long universityId;

  @Column(name="university_name")
  private String universityName;

  @OneToMany(mappedBy="university",fetch = FetchType.EAGER)
  private List<Student> student;

  Void setUniversityName(String name){
    this.universityName = name;
  }

  ... other setters and getters functions.

}
```

Listing 3.2: University.java file that shows how to associate entities using eager loading

```
for(University u:universityList){
    u.getUniversityName();
```

```
}
```

Listing 3.3: An application program, Excessive.java, that generates unnecessary data retrieval

3.4.2 Row-by-row

Definition: The row-by-row, or one-by-one, anti-pattern occurs when developers perform a task that requires a large number of requests. It is derived from the *Empty Semi Trucks* anti-pattern [57], which happens when performing a task that requires a large number of requests. This antipattern can occur in the context of ORM and it is also known as joining data in memory or N+1 query problem because developers mistakenly try to join data from two different queries in a main memory of applications.

```
for(University u:universityList){
    for(Student s:u.getStudent(){
        s.getStudentName();
    }
}
```

Listing 3.4: An application program, RowByRow.java, that generates a huge number of select statements

Performance Impact: RowByRow.java, as in listing 3.4, illustrates a source program that loops through all the universities (universityList) and gets the student attribute, which is a name, for all students in each university. If we run RowByRow.java using ORM configuration of listing 3.2 except that we change extract type

to LAZY loading instead of EAGER, it would create one *select student* statement for each university object. Too many SQL select statements are generated and sent to the database which impacts the overall performance. The author [19] experiment the performance by filling the database with 300 records in one table and 10 records for each one of those 300 records in the associated table. They confirm the performance impact by creating 10 * 300 *select student* statements for each university object as illustrated by RowByRow.java in the same listing. The response time before repairing RowByRow.java is 1.68 seconds, while repairing the performance antipattern brings down the response time to 1.39 seconds, leading to a 17% performance improvement.

The reason why this antipattern is coined as the N+1 query problem is that it generates one select statement for the outer query and then N additional selects. It is slow because it can generate hundreds or thousands of queries for N. The nested select problem is still an issue and a subject for online blogs [58]. For every query that is sent to the database, it incurs database execution time, network response time, and bandwidth, which are multiplied by the number of queries. The bottleneck is the network response time, while the previous antipattern's limiting factor is bandwidth. The antipattern belongs to the category of performance antipatterns. Listing 3.5 shows an example of N+1 query problem.

```
for each row from outer query:
    for each row from inner query:
        ## join results to new data structure from query 1 row and query 2
    end loop
end loop
```

Listing 3.5: A pseudocode that shows the join operation

Solution: The proposed solution is to use batching with a specific number of queries to limit the number of queries in each batch.

3.4.3 Inappropriate Service Intimacy

Definition: This antipattern is identified when a microservice incessantly tries to query the private data owned by other services instead of using its data. A microservice's data should be private and accessed via an API gateway or using its API. **Maintainability Impact:** The coupling can be increased if the microservice connects to the private data of other microservices. With tight coupling, when developers make changes to one service, other services need to be changed as well; thus, it is slow for developers to build, test, and deploy tightly coupled services. Developers have to maintain services as they would with a monolithic application because the core benefit of microservice architecture is wasted.

Data Integrity Impact: By using a database per service pattern, developers might experience catastrophic issues caused by tight coupling such as data inconsistencies and/or data loss. For instance, a service trying to connect other services' databases can change their entries without changing their own. This antipattern is certainly classified into a group of antipatterns that affects data integrity and maintainability.

Solution: Developers should consider combining microservices as suggested by [59].

3.4.4 Brain Repository

Definition: A repository is simply an in-memory collection of objects. This antipattern is classified under query antipattern since it supports the querying capabilities of in-memory objects.

Portability Impact: Repositories loosely decouple business domain and persistence mechanisms. When developers mix business logic and database related code, they create a brain repository antipattern that adds a pointless abstraction on top of other abstractions such as ORM. As a result, it affects the portability factor of such a system. For instance, because of how business domain and persistence code is tightly coupled, developers cannot change their underlying persistence framework, i.e., from the Entity framework to Hibernate.

Solution: Developers should place the sophisticated logic in one method and the SQL query in another. They can move the sophisticated logic to a COMPONENT if it is used by other repositories. This solution is proposed by [60].

3.4.5 Laborious Repository Method

Definition: Developers call more than one method within a repository class because they might join data to satisfy a business rule/logic. Developers should apply at a higher level.

Performance Impact: In general, calling a database more than one time within the same method repository might cause performance issues. Since this antipattern occurs in an MVC web architecture, resources such as network, bandwidth, and possibly a database are shared. However, sometimes it is required to add a call to a database within a repository method for validation [61]. For instance, developers might check user credentials before they proceed and fetch database records.

Solution: They should separate database methods that reside in one repository class into independent methods as suggested by Aniche et al [60]. The newly formed methods may or may not be private to that REPOSITORY; for example, the new method may be public if a persistence action may be utilized independently.

3.4.6 Meddling Service

Definition: This antipattern is classified under query smells because a service queries a database directly. In MVC architecture, the service layer should not directly expose database internals to the application and should delegate another layer,.i.e, a repository, or data access object (DAO) to query the database.

Maintainability Impact: Three reasons why a service layer exists are the separation of concerns, security, and loose coupling [62]. By calling the database through the service layer, developers overlap the service and persistence layer, causing tight coupling, possible code duplication, scattered bugs, and code inflexibility [63].

Portability Impact: As discussed in maintainability impact, by mixing business logic with data access details, developers couple the business and persistence layers. For instance, let us suppose that developers have 50 methods in a controller class, which in turn calls 20 methods in a Data Access Object (DAO) layer. If they decided to change the DAO layer, which the controller relies on to support different persistence strategies, they would also change the code, which resides in the controller class. Thus, it is harder to maintain such code and ported to another DBMS. The antipattern affects the data integrity, maintainability, and portability of the DBMS.

Data Integrity Impact: As discussed in maintainability impact, having database queries inside a service layer degrades security and exposes the database to the attacker [62]. Developers should abstract away the service layer from any persistence details; thus, it adds an extra layer that is isolated from the database and enhances the security of the overall system by hiding persistence details from the attacker when using the application. As a result, the data integrity of such a service is at stake.

Solution: Developers should place SQL queries in repository classes [60].

3.4.7 Chatty (Web) Service

Definition: Developers complete one abstraction by invoking many fine-grained operations. When methods calls or data web services are abundant to complete one abstraction, this antipattern is called chatty web service. Such methods are typically attribute-level setters or getters. This antipattern is classified as one of the query antipatterns because it relies on data web service as an access point to query. For instance, instead of returning all attributes of user status in a user's profile such as wins, losses, and emotions in a game app using setUserProfile() and getUserProfile(), developers can call setters and getters of each attribute of a user profile. They end up having six setters and six getters, depending on the number of attributes. Listing 3.6 shows an example of setting and getting the emotion attribute.

Performance Impact: This antipattern results in poor performance because it is a variant of the chatty interface where developers have to communicate with its operations multiple times through service to fulfill one overall process. One process may require some fine-grained operations to be called, and it depends on how granular operations are within such an interface. Each fine-grained operation has a penalty in terms of performance [64]. Because this chatty interface usually contains a high number of fine-grained operations, it becomes confusing. Developers are unaware of the order in which they can invoke these operations. It is harder to change the order of method calls, leading to an overall higher response time [43].

Maintainability Impact: Because this chatty interface usually contains a high number of fine-grained operations, it becomes confusing. Developers are unaware of the order in which they can invoke these operations. If developers combine these operations into fewer coarse-grained operations, these operations would be easier to be invoked; thus, it makes the interface more readable and understandable [64]. This

antipattern complicates the system maintainability. As a result, it is harder to change the order of method calls.

Solution: Developers should specify parameters of the web service or API call to return only intended data pertaining to selected parameters. This would solve the performance overhead. The API documentation normally explains what kind of parameters it supports so that developers can choose needed parameters.

```
function getEmoticon() {
    if ($this->emoticon) {
        return $this->emoticon;
    }

    $stmt = $this->conn->prepare('SELECT emoticon FROM profile WHERE id=:u
    $stmt->execute(array('userid' => $this->userid));
    $row = $stmt->fetch();

    return $this->emoticon = $row['emoticon'];
}

function setEmoticon($emoticon) {
    $stmt = $this->conn->prepare('UPDATE profile SET emoticon=:emoticon WH
    $stmt->execute(array('id' => $this->userid, 'emoticon' => $emoticon));
    return $this->emoticon = $emoticon;
}
```

Listing 3.6: Example of Chatty Web Service Interface antipattern

3.4.8 CRUDy Interface

Definition: This antipattern occurs when SOA and distributed system architecture developers reveal business functionality wrapping data access and providing a complete abstraction that involves more than one operation (an aggregate of more than one operation). Developers design service with RPC-like behavior that implements CRUD operations instead of sending a well-defined message that dictates the action to be taken using REST. Listing 3.7 shows an example of CRUDy Interface that does not encapsulate knowledge about its processes.

Performance Impact: Calling these CRUD operations can entail chatty responses that affect system performance. This antipattern can cause API chatty service antipattern to occur if multiple CRUD operations are inevitable to finish one abstraction since multiple calls are required to attain one objective [43]. It can appear in .NET (i.e., using VB.net).

Data Integrity Impact: As discussed in performance impact, services implementing CRUDy interfaces reveal business functionality. Developers should not expose services to consumers in this manner because they allow mixing internal behavior and data with public interfaces. Developers also need to know the order in which they can call these operations. If not, the service can make the data to be inconsistent. For instance, when developers call to create an operation to create an instance of any entity, the system will not instantiate this instance. Since developers forget to call the commit function within the interface, the service leaves the data in an inconsistent state.

Solution: Developers should design composed business-based interfaces instead of CRUDy interfaces. CreateCustomer is not good candidate for a business usecase. Instead, they should create granular interfacers and represent the whole business proces such as ServeCustomer.

```
service.CreateCustomer(c);

foreach(Group group in c.Groups)

  service.AddCustomerToGroup(c.CustomerId, group.GroupId);

foreach(Person person in c.Contacts)

  service.AddCustomerContact(c.CustomerId, person);
```

Listing 3.7: Example of Service Interface Using CRUDy antipattern

3.4.9 Maybe It Is Not RPC

Definition: This antipattern arises when a remote procedure call (RPC) is used to intentionally exchange documents that are inherently suited for document-style interactions (electronic document interchange). Document-style interactions interchange instances of significant business entities. The motive behind such an exchange is to apply CRUD operations on such entities. The RPC approach is not suitable for implementing CRUD operations on entities or business documents because it generates operations that have a lot of parameters with custom types, SOAP document fragments, and so on [64].

Performance Impact: it manifests within the Web service that mostly issues CRUD operations. These CRUD operations come with a large number of parameters causing poor system performance [43]. The reason why it impacts performance is that client threads are stuck waiting for synchronous responses [64]. Clients may experience the system freezing because the service uses RPC-style interactions with

synchronous requests. When synchronous requests are used altogether, it can be disastrous, so we classiy this antipattern into one of the performance antipatterns. This antipattern can occur using Java (specifically J2EE) [64].

Maintainability Impact: As discussed in the impact on performance, the web service require a large number of parameters to communicate via RPC-style. However, these parameters might be user-defined types or document fragments, confirming that they are parts of documents [43]. These document parts are communicated individually as separate elements using RPC-style exchange format instead of document-style interactions as a whole. As a result, Clients may have to create instances of document fragments that would likely be better constructed as a document. It complicates the maintenance factor of the underlying web service.

Solution: Developers should change RPC-based communication to document-based communication to solve this antipattern. There are proposed steps to faciliate the conversion including defining an XML schema for the document, making Web Services Description Language (WSDL) changes, making service endpoint changes, and making service client changes for client [64].

3.4.10 Data (Web) Service

Definition: Data service antipattern, known as data class [65], resembles a service that has only setters and getters of a given class, acting as an access point to a distributed database in SOA.

Maintainability Impact: Developers simply use this service to execute simple data retrieval information and includes only basic read operations. This data service might consist of many low cohesive operations. Palma et al. [43] suppose that it can hurt maintainability. We categorize this antipattern as one of the maintainability antipatterns.

Solution: As suggested by [66], developers should remove setters by setting attribute in the constructor. They also can remove getters by implementing a visitor pattern.

3.4.11 Sand Pile

Definition: This antipattern appears when developers implement an SOA with one basic service per software component. It leads to many smaller services sharing common data. This abundant number of services per software component might cause inefficiencies and maintenance problems [43]. We classify this antipattern as one of the maintainability antipatterns.

Maintainability Impact: A wrong grouping of atomic capabilities cause Sand Pile. Even though this antipattern is applicable for SOA based systems, it is not an antipattern for Microservice Architecture-based systems. When the developer replied to a question about having one service per multiple contracts or many services, he recommended to create fine-grained services such as GetUser(userObject user) and avoid finer services such as GetUserByName(string Name) [67]. The developer contradicted his reply by confirming that API chatty web service and Sand Pile are not antipatterns for MA-based systems [67].

Solution: Developers should recombine related basic services into a single software component with a common interface.

3.4.12 Nested Transaction

Definition: This antipattern appears when developers use Spring framework transaction management along with Hibernate. Specifically, developers can annotate methods or classes with **@Transactional** wrapping them in a transaction, and then within them, they can call other methods. Properties of a transaction can be specified

such as *REQUIRES_NEW or NOT_SUPPORTED*. When developers annotate the called method with transactional property and specify it as *REQUIRES_NEW*, the transaction manager will create a nested transaction that suspends the transaction from which the call originates.

Listing 3.8 shows an example of a nested transaction managed by the Spring transaction management framework. When the framework executes the Foo1 method, the framework creates a transaction with a timeout of 300ms. Then, this method will call another method, Foo2, which creates a nested transaction. The nested transaction suspends the outer transaction and continues executing Foo2. The reason why a nested transaction is bad is that the outer transaction might timeout. Also, Using incorrect properties causes database deadlocks [48].

Performance Impact: This antipattern results in smelly code that may introduce problems such as timeout, and deadlock [48]. The authors comment that it has a nonfunctional impact without elaborating in many details in terms of performance.

The execution of a transaction can be bound by configuring a timeout, for instance, 300ms. When the timeout is elapsed while waiting for the subsequent method to be executed, it causes transaction overhead by rolling back the outer transaction as illustrated in Listing 3.8. The outer transaction will throw *TransactionTimedOutException* since the inner transaction takes longer than the specified time to execute. Any RuntimeException promotes rollback, and any checked Exception does not. Such overhead affects the performance of the overall system. The lock only occurs when the suspended transaction acquires a lock on the same object that the nested transaction tries to grant the lock. We found a developer on a StackOverflow providing problems that can happen when using transactions [68]. Locking on the DB level, which can be a row, table, or a cell value, is among the problems and developers should avoid it [68]. Thus, we categorize this antipattern into performance antipatterns.

Solution: Depending on the nature of the situation, there are a variety of options as suggested by [48]. Developers can, for example, remove the transaction property if the transaction isn't required, or perform the second transaction asynchronously if the two transactions aren't interdependent. It's also possible to fix the code to include the annotation in other methods or create new APIs with alternative transactional behaviors.

```
Class A{
@Transactional(timeout=300ms)
Public void Foo1(){
        Foo2();
}
}

class B{
@Transactional(REQUIRES_NEW)
Public void foo2(){
      ...
}
}
```

Listing 3.8: Nested transaction in Spring

3.4.13 Unexpected Transactional Behavior

Definition: This antipattern appears when a calling method and called method to reside within the same class. Spring creates proxy when developers annotate methods or classes with *@Transactional*. Spring will not create a proxy if developers

annotate two methods with *@Transactional* and reside within the same class. Spring creates proxies using either standard Java proxies or CGLIB, a code generation library. A proxy object intercepts incoming external calls that originate from a class that is different from the class of the proxy because Spring wraps transactional methods in a try-catch block and rolls back if an exception is triggered. Developers are oblivious of the way Spring implements transactions, so they write code that shows unexpected transactional behavior. They should annotate functions carefully and avoid nested transnational behavior. Listing 3.9 shows an example of such behavior. When a Foo1 method is annotated with @Transactional and calls another method Foo2 that is also tagged with @Transactional inside the same class, the outer transaction created by Foo1 will not create a nested transaction. A self invocation will not start a new transaction even though it is annotated as illustrated in Listing 3.9. Even if developers remove @Transactional from the Foo1 method, it will not create a transaction in the Foo2 method.

Maintainability Impact: As an application gets larger, it is harder to spot unexpected transactions behavior in Spring transaction management; thus, developers might find it difficult to find such problems in the code. Since developers might not be aware of how proxies handle transactions, it takes time to manually debug their code finding bugs that occur as a result of *@transactional* annotation. There are many posts on StackOverflow about how to maintain self-invocation transactions. One developer explained why he advises developers to be cautious when using self invocation [69]. It is implicit and buried within the framework, so it is not good practice to maintainable code. Some developers are unaware that *@transactional* should be applied only to public methods when using proxies [70]. Applying the notation to private methods will not lead to any effects. Chen er al [48] confirm that when developers deal with complicated systems, it is difficult and time-consuming to look

for bugs caused by using the notations. This antipattern impacts the maintainability factor of software systems.

Solution: Refactoring the code so that methods annotated with @Transactional(REQUIRES NEW) are in a different class than the caller is one solution for Unexpected Transaction Behaviour [48].

```
Class A{
@Transactional
Public void Foo1(){
        Foo2();
}

@Transactional
Public void Foo2(){
       ...
}
}
}
```

Listing 3.9: Unexpected transaction behavior in Spring

3.4.14 Inconsistent Transaction Read-write Level

Definition: This antipattern occurs when developers use the read-write level property as a default for Spring transactions with read operations. Developers should annotate read transactions with read-only as a hint for the Hibernate engine even if DBMSs do not support read-only transactions because they incur lower performance

overhead. Listing 3.10 shows an example of a transaction with a default read-write level, but the function only reads data from the database.

Transactions might not change the status of the database. Specifically, they only fetch database tables or rows to read the content, instead of adding, updating, or deleting database records. If so, developers can specify the property of a transaction with read-only when its method or subsequent methods do not change data in the database. Listing 3.10 illustrates a simple retrieval case from the database. The default property is set to be read-write level. It should be changed to be read-only level. Modifying the property level to read-only might enhance the performance of the system because fewer lock conflicts can occur.

Performance Impact: Developers can make use of Hibernate performance boosts by specifying read-only level with *@Transactional* annotation. Developers are aware of reasons why they should mark transactions as read-only [71]. Specific DBMS such as MySQL can optimize read-only transactions in the InnoDB engine starting from the 5.6.4 version. InnoDB can evade the overhead by creating transactions ID for read-only transactions. A transaction ID is required when transactions include write operations or locking read such as SELECT ... FOR UPDATE. Extraneous transaction IDs are removed minimizing the size of the internal data structure required. These internal data structures are referenced whenever a query or data change statement constructs a data view. The same developer [71] also illustrates another optimization that prevents transactions from dirty checks because Spring sets the FlushMode to MANUAL in case of read-only transactions while developers use Hibernate. This antipattern clearly shows that it affects performance because these reasons can convince developers to not use the default read-write level property.

Solution: For functions that do not modify data in the DBMS, the solution would necessitate changing the annotated transaction to a read-only transaction [48].

```
@Transactional
Public Employee readEmployeeByID(int id){
        return session.find(Employee.class, id);
}
```

Listing 3.10: Unspecified transaction readonly level property

3.4.15 Sequence Name Mismatch

Definition: Hibernate allows developers to choose a sequence object in the DBMS by annotating an instance variable with @SequenceGenerator to identify the name of the sequence object that the variable utilizes. Sequence objects produce numbers sequentially when new sequence objects are constructed. Nevertheless, developers might mismatch the name of the sequence object in the source code with the one in the schema file, so duplicated sequence numbers can occur and might be the basis of redundant primary key errors. Listing 3.11 shows an example of a mismatch between the DB and source code.

```
class Employee{
@Id
@SequenceGenerator(sequenceName="employee_seq")
@Column(name="employee_id")
Private int id;
}


employee_schema.sql
```

```
employee_id BIGINT NOT NULL DEFAULT nextval ('employee_id_seq')
```

Listing 3.11: Names mismatch between an annotation of instance variable and schema file

As illustrated by listing 3.11, the Employee class represents the employee table in the database. The employee identifier attribute of the employee class links the employee_id column, which is a primary key of the employee table in the database management system (DBMS), but the sequence name object "employee_seq" in the annotation does not match the one in the schema file 'employee_id_seq'. Chen et al [48] claim that copy-and-pasted code can cause such inconsistent naming.

Maintainability Impact: As brought by [48], we confirm there are no discussions online commenting on the impact of such mismatch. However, Chen et al. [48] believe such mismatch might be related to common copy-paste problems in practice. We think copy-paste problems have a significant impact on maintainability. StackOverflow question [72] phrased as, what hurts maintainability? and developers answered by writing duplicated code. Copying and pasting code results in duplicated code, so it drastically affects maintainability. A developer illustrates an answer in SO with an example of frequent maintenance of scattered and duplicated code. Every time the developer fixes an issue of a given code fragment, the same issues will show up with the same fix [72]. Thus, it hinders developers' productivity and development time.

3.4.16 Incorrect SQL Orders

Definition: This antipattern appears when the order of the database access code is different from the order of generated SQL queries by Hibernate. Writing database access code using a framework such as Hibernate generates SQL statements

that are dissimilar to the developer's intended order in the database access code. For instance, deleting an employee using a unique key and then reinserting the employee with updated key results in an update followed by a delete which is not similar to the order of logic in the database access code. This incorrect SQL order is caused by Hibernate because it performs optimizations on SQL statements by reordering them. Listing 3.12 shows an example of such mismatch.

```
group.getEmployeeList().clear();
group.addEmployee(Employee);
update(group);

// resulting SQL queries
Insert into Employee Values ...
Delete from Employee Where ...
```

Listing 3.12: A mismatch of order between database access code and generated SQL statementes in Hibernate

Initially, Chen et al [48] clear all employees in the group. Then, they insert a new Employee into the group. Nevertheless, Hibernate generated SQL queries that are inserting a new Employee followed by deleting Employees from the group. The order does not match and it might be a source of duplicate key problems and or produce unanticipated results.

Data Integrity Impact: When Hibernate reorders SQL statements, it may lead to constraint violations. An online post inquires about modeling bidirectional parent-child relationships with ordered children [73]. The generated SQL statements by Hibernate violate unique constraints because updating a sibling is being executed before deleting one child (target) from the same parent. When developers ignore

unique constraints, data becomes exposed to unauthorized edits. Thus, this is why this antipattern has an impact on data integrity.

Solution: One solution of this antipattern is to call the Hibernate method flush() to force the reorder and send SQL statements. a JPA provider,i.e., ORM, implement the flush() method, and it varies from one provider to another. Thorben Janssen [120] considered using the flush() method without a reason as a mistake. When developers use the flush() method after updating a persisted entity, they enforce Hibernate to execute a dirty check on all managed entities and to create and execute SQL statements for all pending INSERT, UPDATE or DELETE operations. Using the flush() method without a reason slows applications because it deters Hibernate from applying optimizations on the generated SQL statements. Hibernate keeps track of all managed entities in the persistence context and attempts to postpone the execution of write operations as long as possible. This way of handling the antipattern degrades performance. Thus, we classify this antipattern as one of the performance antipatterns.

3.4.17 Inefficient Computation (IC)

Definition: This antipattern appears when the badly running code conducts useful computation but inefficiently. Inefficient computation can be a result of having inefficient queries or having computation in the incorrect component of the web application, such as the DBMS instead of the server or vice versa. Inefficient queries, for instance, can happen when developers use *any?* instead of *exists?* [22]. Different ORM APIs can implement the same functionality on the persisted data with different performance overheads. Listing 3.13 illustrates how a shopping system checks whether the inventory of the product variants is not tracked. The only difference between the two Ruby statements is in the call of *any?* versus *exists?*.

Performance Impact: Useful computation but inefficient computation consists of more than %10 of performance problems in the bug reports [22]. The generated queries by different ORM APIs that implement the same functionality can incur extremely various performance overheads [22]. They claim that the research community has not touched this area for ORM applications.

Inefficient queries can happen when developers use *any?* instead of *exists?* [22]. Using *any?* keyword generates a query that iterates through table rows to calculate occurrences of missing indices, while using exists? requires scanning and locating the first row if the predicate is equal to true. This is the case if developers use Rails 5.0 or below, but the implementation of any? has changed since Rails 5.1. It is similar to exists? except that it is not memorized unless loaded. Another typical issue is when developers use API calls that create queries' results in a useless sequence. For instance, some developers in one of the studied projects call Object.where(c).first for retrieving an object that meets predicate c instead of Object.find_by(c), becoming oblivious of the fact that using the API call with where clause arranges Objects by primary key after evaluating predicate c [22]. This is different from Incorrect SQL antipattern where we have a mismatch between the database access code and the generated SQL by Hibernate. The unnecessary order here reflects the ordered records caused by the generated SQL statement.

Generated queries sometimes incur unnecessary network Round Trip Time (RTT), or consuming computation on the server-side instead of the DBMS, leading to performance degradation. For instance, one project report uses pluck(:total) instead of sum(:total) which generates a query to load the total column of all corresponding records and then calculates the sum in memory, while calling sum(:total) issues a query that directly carries out the sum in the DBMS without returning actual records to the server. On the other hand, developers should move other computation cases

to the server from the DBMS. For instance, developers substitutes Object.count with Object.size in 17 locations [22]. We classify this antipattern as a query antipattern that affects the performance of a given system.

Solution: As illustrated in Listing 3.13, solving this kind of antipattersn require ORM expertise. Developers should use the keyword exists? instead of any? to eliminate performance overhead caused by inefficient query.

```
# One way to check (inefficient)
variants.where(track_inventory: false).any?
# The generated SQL query is inefficient because of table scan
SELECT COUNT(*) FROM variants WHERE track_inventory = 0?

# Another way (efficient)
variants.where(track_inventory: false).exists?
# The generated SQL query is efficient because of limiting the result to o
SELECT 1 AS ONE FROM variants WHERE track_inventory = 0? LIMIT 1
```

Listing 3.13: Ways of checking the product inventory by the online shopping system using any? and exists?

3.4.18 Unnecessary Computation (UC)

Definition: This antipattern occurs when unnecessary queries are generated and executed. These generated queries can be loop-invariant queries, dead-store queries, and queries with known results. Loop-invariant queries are generated when an ORM API is called within a loop. Dead-store queries are repeatedly issued similar to loop-invariant queries. However, they fetch different database contents into the same main-memory object, and developers do not use the results of database con-

tents between the iterations in the same object. Developers issue Queries with known results even though their results are already fetched.

Performance Impact: Developers are oblivious of the fact that these repeatedly generated queries retrieve the same database contents which slow down the overall system. We recommend developers to hit the database a fewer number of times. Two reasons as follows are discussed in Online discussion [74]. Developers can obtain the best optimization from the database because it is best at retrieving all things at once. By doing so, developers also eliminate the overhead associated with calling the database multiple times.

Dead-store queries are repeatedly issued similar to loop-invariant queries. However, they fetch different database contents into the same main-memory object, and developers do not use the results of database contents between the iterations in the same object. For instance, in a shopping cart application, the order table has one to many relationship with line-items. Order is not complete until users check out their shopping carts, so there is no need to call reload in the order table whenever users update their line-items. Reload calls should be placed inside the payment table when users already check out their shopping carts. The reason behind doing this is because of the underlying implementation of the Rails framework. The ORM of this framework, ActiveRecord, has an associated overhead, and developers can prevent such overhead by using bulk updates [75]. Reloading line items of order generates bulk updates to the database. The best way to optimize the Rails framework is to utilize its memory. Developers need to be aware of how much ActiveRecord takes a large amount of main memory.

Queries with known results are issued even though their results are already fetched. Consequently, such queries cause additional query processing time and network round trips. Thus, the solution would be simply to use a cache and execute

only the minimum number of queries. We classify the unnecessary computation antipattern into performance antipatterns.

Solution: This can be broken down into the three typs of unnecessary commputation. For loop-invariant queries, developers should remove any loop that repeateadly issues the same query on the same set of attributes. For dead-store queries, developers should make sure that the query is issues differently on different reload session in main memory. This way ensures that each data fetched in different reload session is used efficiently. For queries with known results, developers should set thresholds for queries that retrieve the same known results, repeatedly; it ensures that these queries do not incur unnecessary communication overhead between a systems and database.

3.4.19 Unnecessary Data Retrieval (UD)

Definition: This antipattern appears when either select all, update all, excessive data, or per-transaction cache is detected in the generated queries. It occurs when the application gets data and this data is already stored in the database, but it is not utilized later. Unnecessary data retrieval is carefully studied in [49]. They check unnecessary data retrieval by comparing the requested database accesses (those are the generated SQL statements during program execution) and the needed database accesses (defined as how functions in these accesses are called during execution).

Performance Impact: Fixing update all by eliminating unnecessary columns improves the performance of the studied systems by 4-7% [49]. The more unused columns the ORM sends to the DBMS, the more likely performance is going to be affected. When the generated SQL statements are updating columns that are not used by developers and non-cluster indexed, it can slow down performance. work in [49] shows that studied systems such as Pet Clinic, ES, and Broad Leaf have suffered

from a performance impact by using select all, one kind of redundant data problem. There is a need in practice to fix select all kind of data retrieval antipattern because it degrades performance [76]. With Hibernate, developers can use projections from Hibernate criteria query API to eliminate such unnecessary columns to enhance the performance of the data retrieval. Hibernate Criteria Query API is more complex, harder to maintain, and it is suited for dynamic queries. Excessive data is discussed in detail when we classified paper [19]. Per-transaction cache is the fourth type of unnecessary data retrieval. Authors in [49] witnessed cases of using per-transaction in the large-scale industry systems in which redundant SQL queries are generated many times to fetch a result that is already found in the cache. For instance, Hibernate expert recommends utilizing QueryCache to stay away from firing additional SQL queries to get the same result [77]. We classify this antipattern as one of the 13 performance antipatterns.

Solution: For update all, developers should use projection by selecting which columns they need in their applications. For select all, they should delete the eagerly fetched table in SQLs where the fetched data is not utilized in the code. For exessive data, developers should change the fetch type of transactions. For per-transaction cache, developers should adjust cache configurations pertaining to queries that fetch the same data sequentially without changing the data.

3.4.20 Inefficient Rendering (IR)

Definition: When developers use Object Relational Mapping as an abstraction layer on top of the DBMS, they need to worry about the time the ORM takes to interact with the application, for the database to execute queries and return the result and also render the result via application views. When a view file renders a set

of objects, it indicates a balance that developers must take into consideration between performance and readability.

Performance Impact: It is true that Inefficient rendering indeed causes performance overhead, but it is negligible when compared to the performance impact caused by inefficient queries. Partial rendering slows down the performance of the system if developers iterate through each partial and Rails evaluate each one in turn [78].

Maintainability Impact: Regarding performance impact caused by inefficient queries, IR antipattern enforces developers to consider the trade-off between readability and performance. Looping through each partial is better in terms of code readability [74]. Developers can use one part to create a view and use string substitution to change views without additional rendering time. Although using this alternative method is better for performance, it has an impact on code readability which affects how developers maintain the system.

Solution: Shao et al [79] suggested a solution for fixing this antipattern by Using more performant APIs for view rendering.

3.4.21 Inefficient Data Accessing (ID)

Definition: This antipattern appears when inappropriately developers use batch processing or real-time processing by sending individual queries to the database to be executed. Inefficient lazy loading case, N+1 query problem, is discussed in *The Row-by-row* antipattern in section 3.4.2. Another case of Inefficient data accessing is inefficient eager loading. We also discuss in *Excessive data* antipattern in the same section. Inefficient updating is a third case that generates N queries separately without combining them into one query (as a batch) to modify N database rows. Listing 3.14 shows an example of a Rails code that issues N queries individually. This antipat-

tern can occur in another framework (i.e., Hibernate) [80]. However, with Hibernate, developers cannot create one SQL statement that updates N records. They need to use native SQL or JPA to implement bulk updates. We categorize this antipattern into performance antipatterns.

Performance Impact: This antipattern causes data transfer overhead, including not batching data transfers (e.g., the well-known "N+1" problem) or batching too much data into one transfer [22]. This is related to inefficient lazy loading, inefficient eager loading and inefficient updating.

Solution: Developers should batch the N update queries into a single query.

```
#Iterating through an array of objects to generate N queries
objects.each |o| o.update end)
# One Rails statement that issues N query as a batch to update N database
objects.update_all
```

Listing 3.14: Inefficient updating before and after the fix

Table 3.2: Categorization of Framework-Specific Query Antipatterns.

Reference	Antipattern	Type	Performance	Maintainability	Portability	Data Integrity
[19],[74]	Eager Fetching Problem	*ORM*		✓		
[19],[38],[20],[74]	Row-by-row	*ORM*	✓			
[45]	Inappropriate Service Intimacy	*Architecture*		✓		✓
[41]	Brain Repository	*Architecture*			✓	
[41]	Laborious Repository method	*Architecture*	✓			
[41]	Meddling Service	*Architecture*		✓	✓	✓
[43],[44]	Chatty Service	*Architecture*	✓	✓		
[43]	CRUDy Interface	*Architecture*	✓			✓
[43],[64]	Maybe Its not RPC	*Architecture*	✓	✓		
[65], [43],[44]	Data Service	*Architecture*		✓		
[43],[44],[67]	Sand Pile	*Architecture*		✓		
[48]	Nested Transaction	*ORM*	✓			
[48],[69],[70]	Unexpected Transactional Behavior	*ORM*		✓		
[48]	Inconsistent Transaction Read-write Level	*ORM*	✓			
[48]	Sequence Name Mismatch	*ORM*		✓		
[48],[81]	Incorrect SQL Orders	*ORM*				✓
[22]	Inefficient Computation (IC)	*ORM*	✓			
[22]	Unnecessary Computation (UC)	*ORM*	✓			
[19], [49],[74], [76], [77]	Unnecessary Data Retrieval	*ORM*	✓			
[74], [78]	Inefficient Rendering (IR)	*ORM*	✓	✓		
[74], [80]	Inefficient Data Accessing (ID)	*ORM*	✓			

CHAPTER 4

Extracting Standardized Data Element Names from Natural Language Definitions

In this chapter, we design and develop rule-based natural language processing (NLP) techniques to automatically extract data element names from data element definitions written in American English. The goal is to study how using NLP techniques can improve the accuracy of extracting standardized data element names in a domain-independent context. To achieve this, we first identify heuristic patterns that mine noun phrases and relationships from data element definitions. Then, we use these noun phrases and relationships as input to determine components of data element names. The output of the patterns is reviewed by a domain expert. We apply our method to extract the 5 standard components of a data element name in the Railway and Transportation domains. We first achieved 80% accuracy, then by improving the rules and adding a similarity function, we improved the accuracy to 95% in our final experiments.

4.1 Introduction

Enterprises need specific data and information to run their operations. The smallest level of granularity of data that has a precise meaning and is both unique and of interest and use to the business is a data element. Data modelers write data element definitions to describe a data element by stating its complete and precise meaning in a unique English statement. They manually create unique data element names by which data elements can be identified within a business. They are the basis for all other technical data names. Enterprises aim to manage their data in a

centralized manner. However, they tend to have data silos separated across different data centers. These data silos meet the need for the common business meaning.

ISO framework for Data Element Naming. Data element names stored in different data silos need one language to communicate. There exist Data Element Framework's naming and identification principles, ISO/IEC 11179-5 [82], that facilities such communication. Among organizations, railroads require a standardized way of communicating data element names, and they are nominated as Class I, II, or II in the United States. We develop a Class I Railroad's naming standard from the ISO Framework. Their Framework only had 4 components of a data element name: an object class, property qualifier, property, and class word qualifier. We added the fifth one, the Class Word component, and many additional rules for creating the data element names. Not adhering to a data element naming standard causes organizations, namely Class I railroad, to have erroneous operations.

NASA's Mars Climate Orbiter, for instance, burned up in the Martian atmosphere in 1999 because engineers failed to convert units from English to metric [83]. The problem was in the software controlling the orbiter's thrusters. The software calculated the force the thrusters needed to exert in pounds. A separate piece of software took in the "Thruster Force" data assuming it was in the metric unit: Newtons. This discrepancy between the two measures, a factor of 4.45 Newtons per pound, caused the orbiter to approach Mars at too low an altitude. The result was the loss of a $125[1] million spacecraft and a significant setback in NASA's ability to explore Mars. The solution is that NASA should have used a name for the data like "Thruster Force Pounds" that clearly specified the unit of measure for thruster force. Class I railroads can develop naming standards to avoid the problem of inconsistent data.

[1] An amount adjusted for inflation, since that was 1999.

Class I Railroad Naming Standards. Class I Railroad's Data Element Naming Standards are designed to allow any Data Element to be used in any context (operational or analytical) anywhere in the organization – alone or in combination with any other Data Element(s). To achieve this and to support both current and future design methods and technologies, all Data Elements must be uniquely named. However, the definition of Data Elements exists in the form of text, and there is a need for text extraction techniques to support naming standards.

Open Information Extraction. Data Element Definitions contain data that can be extracted using open information extraction techniques to generate data element names. Open information techniques take the definition as input and extract relationships as triples (subject, predicate, object) in a domain-independent manner [84]. We identify patterns on the extracted triples to mine data element names from data element definitions. Open information extraction techniques suffer from high precision and low recall.Some relationships are verbal and other are non-verbal such as prepositional predicates. These prepositional predicates are polysemous; they minimize the accuracy of extracting a data element name from a definition.

The experiments show that our patterns may play supportive roles in the extraction of data element names from natural language. This is one of the first works in which NLP techniques are used on a random set of industrial data element definitions written by domain experts. We contribute by comparing the manual techniques used in industry for naming, and our method based on NLP for extracting data element names.

The remainder of this paper is organized as follows: Section 4.2 describes related work. In Section 4.3, we introduce the Data Element Naming standard and describe its components. Then, we describe Data Element Naming Automation (DENA) tool

that we implement for the Data Element Naming standard in section 4.4. Finally, Section 4.6 concludes our work and discusses future work.

4.2 Related Work

To the best of our knowledge, we have not found work related to extracting data element names from English statements. However, we find three research directions that are partially related to our research.

Extracting Patterns. Definition/inclusion or facts/case relations are ways to determine the semantic relationships between data element components. Definitions/inclusion (Dallas is a city) denote hyponymy relations that are addressed by Hearst patterns [85]. We use heuristic patterns based on linguistics to extract data element names from unstructured text. These patterns rely on a dependency parser and trained model. We believe having a good number of these patterns serves as a basis for solving the problem of extracting data element names. These patterns target one component, the object class, which sets the domain for a data element name. However, the literature is lacking empirical studies on the application of these patterns in industrial settings.

Attribute based Access Control (ABAC) standardized policy documents. **(author?)** [86] propose a framework to extract authorization attributes from Natural Language Access Control Policy (NLACP). Their attribute extraction module detects values of attributes in sentences defining policies. We focus on what is related partially. They use a deep learning approach, but we utilize NLP techniques and heuristics patterns for extracting information. Due to our small number of available test data input and output samples, we decided to use the above-mentioned approach to tackle the problem.

Figure 4.1: A definition and an extracted Data Element Name

Categorization of Semantic roles in dictionary definitions. Building a knowledge graph from natural language definitions has been studied, and many methods depend on dependency parsers for finding patterns that show relationships between words [87]; [88]; [89]; [90]. One of the early studies shows the creation of LKB, a Lexical Knowledge Base based on a set of attributes for a given concept such as the concept of drink and its attributes "origin", "color", "smell", "taste" and "temperature" [91]. The authors parse definitions from machine-readable dictionary definitions to mine the definiendums' genus and differentiae. They only consider the entities with their relevant attributes in a restricted domain. However, in our study, we consider them in a domain-independent manner.

4.3 Data Element Naming Standard

Data element naming standard is the set of rules that guide the construction of consistent Data Element Names. These rules identify the various "parts" of a name – what words/terms can and should be used and in what order they should appear.

4.3.1 Data Element

We define what a Data Element is and the characteristics required to be considered a Data Element. A Data Element is the smallest level of granularity of data that has a precise meaning and is both unique and of interest and use to the business. For instance, many enterprises create a single Data Element for 10-digit telephone number data (without including international code), while telecommunications enter-

prises use three separate and distinct Data Elements: area code (3 digits), a central office code (3 digits), and a line number (4 digits).

Any Data Element has to include metadata about the data it represents in terms of classifying the data, representing the data as a specific feature or quality, and attributing the feature or quality to entities or concepts. The data's classification tells us what kind of general data it is. It, for instance, can be a Date, Image, Number, Code, Rate and etc. The data can represent specific qualities or features such as Telephone Number, Last Name, Temperature, Weight, etc. These qualities or features belong to many entities or concepts such as a Company, Employee, Person, etc.

The Data Element can also include additional components that represent the data in terms of its context and domain values. The Data Element can be used in various contexts including specific industries, locations, and/or professions. We can use the Data Element in Railroad Industry, Finance, a Hospital, Pest Exterminator. In addition to values imposed by the data's classification, the Data Element might also contain valid values about the data it represents. For example, these data values are normally a range or enumerated list of values such as a day of the week: Sunday, or Monday.

4.3.2 Data Element Naming Standards

Once a clear, concise, accurate, complete data element definition has been obtained, a new Data Element can be named by following the Naming Standard, which is defined by a set of lexical and semantic rules that govern a Data Element Name's construction. These rules identify the various components of a Data Element Name, define the composition of each component, specify the order in which those compo-

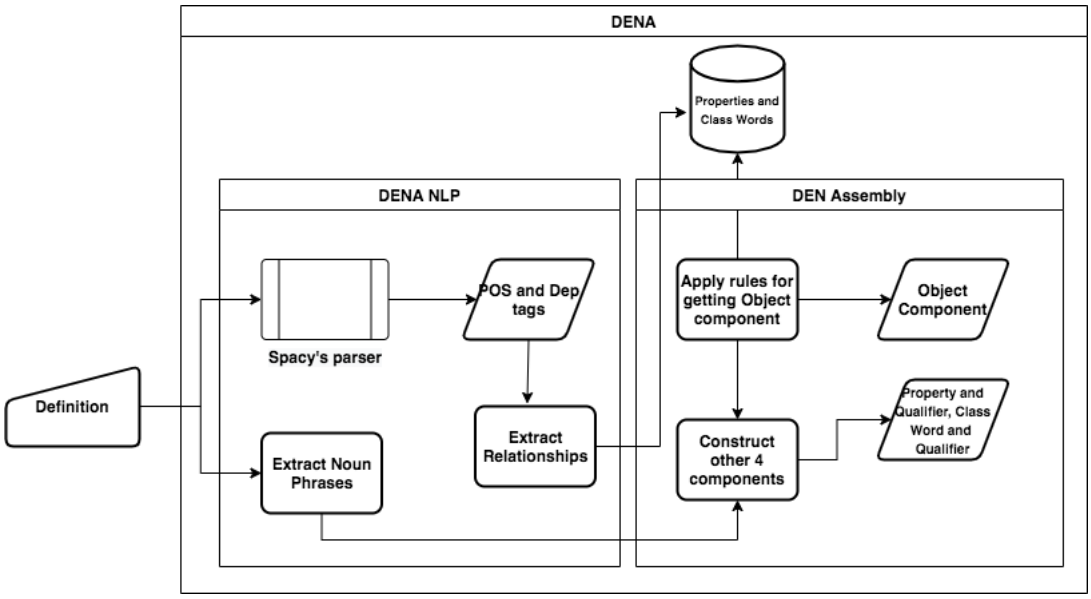


Figure 4.2: Data Element Naming Automation (DENA) Tool

nents appear in the name, resolve conflicts arising from combining components, and cover special formatting situations.

Class I Railroad's goal is to solve the following naming problems of the past:

- Duplicate names with more than one meaning.
- Different names with the same meaning.
- Ambiguous names that can be interpreted as having more than one meaning.
- Misleading names that appear mean one thing but mean something else.

4.3.3 Data Element Name Components

A Data Element Name (DEN) is a single noun phrase that consists of one or more words that represent the meaning of data it entails. The Class I Railroad Data Element Naming standard has recognized 5 separate components that can be part of a DEN. The Object Class, Property, and Class Word components are required for all Data Elements, while Property Qualifier and Class Word Qualifier are only required

for some Data Elements. The following are the components and a description for each component:

- **Object Class:** is a word or phrase that represents a person, place, or thing in the real world that is recognized by the business with specific boundaries and meaning.
- **Property Qualifier:** is a word or phrase that, if needed, differentiates, limits, and/or modifies the meaning of a Property.
- **Property:** is a word or phrase that represents the main idea in a Data Element Name. Each Property is a noun phrase that represents a discrete (separate and distinct) quality or feature that is common to all members of an Object Class, for which the business wants to retain data. The property component can be further classified into:
  - Strong Property: A property whose meaning is clear, and whose Data Classification is intuitively obvious as belonging under only one of the existing approved Class Words. For example, width is a Strong Property under a Class Word measure.
  - Weak Property: A property whose meaning may not be clear, and whose Data Classification is not intuitively obvious as belonging under only one of the existing approved Class Words. I.e., the Property could be validly classified under two or more existing approved Class Words. For instance, speech is a Weak Property under two Class Words: video and audio.
- **Class Word Qualifier:** is a word or phrase that modifies the meaning of a Class Word by more accurately describing the type of data it represents.
- **Class Word:** is a word used for distinctly classifying data within organizations. It describes the content and type of information such as date, name, amount, etc., or its function such as Code or ID.

Figure 5.1 shows an example of a Data Element Definition and corresponding Data Element Name. As an input to our tool, a Data Element Definition should be preprocessed by making grammar correction, punctuation correction, and word usage correction. The data element name is composed of 5 data element components. It starts with the company as the Object Class in red. The second component is annual, which is underlined and its color is blue. Annual as Property Qualifier could be used to constrain the Property's period. The third and fourth components, Property and Class Word Qualifier, are gross revenue and Mexican Peso, and we color them with green and purple, respectively. Classifying Data Elements makes it easier to apply proper governance, which assists in maintaining good data quality.

Table 4.1: Patterns for extracting relationships from definitions

Pattern	**Pattern Rule**
1	A verb follows a noun subject
2	A verb with open clausal complement follows a noun subject
3	A preposition does not follow a verb
4	A preposition follows a verb, and the verb should not be proceeded by an auxiliary verb
5	An adverb modifier follows noun or pronoun, and an auxiliary verb should not follow the adverb modifier
6	A definition starts with a verb and then a noun follows the verb
7	An adposition follows a verb, and a noun or determiner comes after the adposition
8	A verb which is not part of a relative clause follows a particle such as to, the dependency of the particle is auxiliary and auxiliary verbs cannot follow the particle.

4.4 Data Element Naming Automation Tool

In this section, we introduce and describe the components of the Data Element Naming Automation (DENA) tool for extracting Data Element Name components. The DENA tool consists of a DENA Natural Language Processing (NLP) in section 4.4.1 and DENA Assembly in section 4.4.2 .

4.4.1 DENA NLP

4.4.1.1 Spacy

Spacy is an open-source Natural Language Processing (NLP) framework used to heuristically extract relationships and Data Element Name components. We use Spacy's dependency parser to come up with rules for extracting relationships and components. The dependency parser can parse and tag the words in a definition that is fed into the Spacy framework. The Spacy framework also generates base noun phrases that act as arguments between the extracted relationships. Figure 4.2 illustrates the components of the Data Element Naming Automation tool that is constructed using the NLP component. We parse definitions using the English language and load large vector models designed for different tasks in NLP.

4.4.1.2 Heuristic Rules for Extracting Noun Phrases and Relationships

Noun Phrases. As starters, we want to rely on Spacy's base noun phrases, also known as chunks. These chunks have a noun as their head. They exclude coordination, prepositional phrases, or relative clauses. Later we implement utility functions that create these complicated phrases.

Relationships. We utilize the output of Spacy's dependency parser to curate a list of heuristic rules for extracting relationships from definitions. In the context

of relationship extraction (RE), we use Spacy's POS tagger to generate candidate instances by matching against predefined POS patterns. POS is also beneficial in providing lexical information needed to design RE models. Definitions can be incomplete sentences and some open information extraction systems cannot handle such sentences. Figure 4.2 shows a relationship extraction process of the tool. Table 4.1 shows a list of heuristic patterns that are used to extract such relationships. The extracted relationships from sentences are represented using two arguments and a predicate (arg1, predicate, arg2).

4.4.2 DENA Assembly

This part of the tool is responsible for finding the components of the data element name, namely the Object Class, Property Qualifier, Property, Class Word Qualifier, and Class Word. We use the output of the DENA NLP part in the pipeline to derive the heuristics. We introduce the heuristic rules that are used for extracting the Object Class component in section 4.4.2.1. Then, we describe and explain our method for extracting the other four components in section 4.4.2.2

4.4.2.1 Heuristic Rules for Extracting the Object Class Component of a DEN

We start by finding the object class. Then, we find the class word. If a class word qualifier exists, it will be extracted as the third component. The fourth component that we later extract is property. Lastly, we extract property qualifier if it exists.

We iterate through relationships that are extracted from definitions in section 4.4.1.2 to find the object class. The object class component can span either:

- **A single-word noun phrase (NP):** It occurs within at most one relationship. Figure 5.1 shows an example of a single-word NP object class such as a company

- **multiple-word noun phrases (NP):** It occurs within at least one relationship. [Make figure 2 with an example of customer contact]

A single-word NP Object Class. We apply rules to find candidate relationships where the object class can be found. We use the following rules to find the single word candidate object class.

- Object Classes tend to exist in the first sentence. They appear in later sentences with pronouns.
- Object Classes do not appear in subordinate clauses. They can appear in main clauses and relative clauses.
- Object Classes appear in the right entity of a prepositional relationship. Similar to class word qualifiers, they also exist as the right entity of verb + preposition pattern.
- Object Classes also appear in the right entity of active voice verbs and left entity of passive voice verbs.

We cannot use the Class I Railroad Common and Business Terms catalog to distinguish between Object Classes and Properties. Some terms can be both Object Classes and Properties.

We read relationships from the DENA NLP component to determine whether an Object Class is a single-word or multiple-word NP.

multiple-word NP Object class. We iterate through relationships that are extracted in section 4.4.1.2 to find the multiple-word NP object class. We specifically use the following patterns to create the multi-NP object class:

- We use the found single-word NP object class as a basis of finding the multiple-word NP object class.

- We use dictionaries of strong and weak properties and the Class I Railroad's Class Word List to create the multiple-word NP Object Class for one or more relationships.
- We check whether the other NP in a relationship where the object class is found is a property. If so, we combine the NP with the single world object class to become the multiple-word NP Object Class.

We believe that there are no generic tools that can identify the best appropriate noun phrase that consists of multiple words. We think that this is an active and ongoing research area, and the complications of natural language multi-word expressions make it currently impossible to identify the most desirable multi-word noun phrase without the help of a domain expert. In our tool, the domain expertise is represented by the dictionaries that hold the previously-identified noun phrases and their classifications.

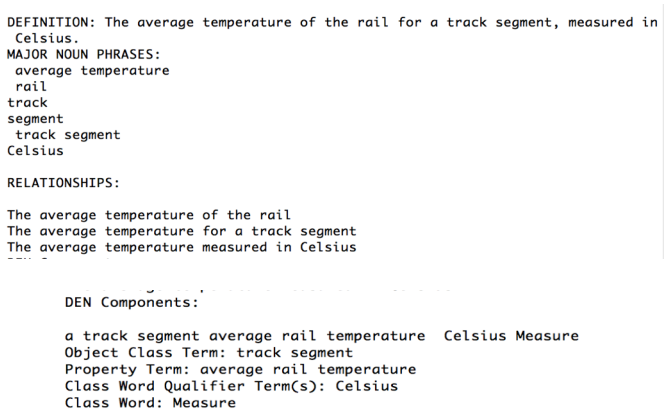


Figure 4.3: A Definition and an extracted Data Element Name from our tool

4.4.2.2 Heuristic Rules for Extracting Other Components of a DEN

Class word. We iterate through noun phrases to find the class word. We only equate one noun phrase with the class word component. We use the following logic when extracting the class word component from a data element definition:

1. We check whether a NP is a Strong property or not.
2. If not, we check whether a NP is a Class Word Qualifier.
3. If not, we check whether a NP is a Class Word from the Class I Railroad's list of Class Words.
4. If not, we check whether a NP is a Weak Property.

Property. We iterate through relationships to find the property. We investigate whether one or more NPs is tagged with Object Class because it sets the context and identifies the location where the property can appear in a sentence. If we find an object class in one entity of a relationship (more likely to be a left entity) and the other entity (more likely to be the right entity) is neither a class word nor object, it becomes the property.

As a result, we assign noun phrases to the DEN components. However, we cannot rely on only noun phrases to extract object and property components. We also use relationships to extract part of object or property components within predicates. Once we find the object class component, we assign other components to each remaining and untagged noun phrase.

4.4.3 Storing the Extracted DEN in the Knowledge Graph using Neo4j.

We store 121 data element definitions and their equivalent data element names in the Neo4j graph database. We propose this approach because our method of extracting an Object Class component rely on checking whether an Object Class is a

single-word NP or multiple-word NP. An extracted relationship can have the following possibilities:

- Property phrase, relationship and object phrase. → (single-word NP Object Class)
- Object phrase, relationship and another object phrase. → (multiple-word NP Object Class)

The knowledge graph stores data element definitions and data element components as nodes with labels. The knowledge graphs store nodes with different labels such as ObjectClass, Property_Qualifier, Property, ClassWord_Qualifier and ClassWord. These labels will help answer the question about whether an entity is an object class or property. We want to compare the similarity at the definition and component level. IF DENA recognizes the data element definition which is stored in the knowledge graph using a similarity function, we have a perfect match, and DENA only return its equivalent data element name components from the knowledge graph. DENA can also compare the input to the stored data element name components in the knowledge graph. DENA can evaluate to the following:

- Known Object class and Known Property (Perfect match at the component level).
- Known Object class and unknown Property.
- Unknown Object class and known Property.
- Unknown Object class and unknown property.

Coding will utilize this new Knowledge Graph to help determine whether Noun Phrases in Definitions are Object Classes or Properties.

Due to the change in our approach of utilizing Knowledge Graphs, a new estimate has been proposed for the completion of the third test. We propose this change to allow DENA to accurately extract data element names based on prior knowledge.

4.5 Results and Discussions

We find that applying patterns to extract data element names from natural definitions is possible with high accuracy and in short time. Manually extracting data element names from data element definitions by data modelers is subject to errors, and it is also a time-intensive process.

An Example of A Data Element Name. Figure 4.3 shows the resut of our tool when it extracts data element names from data element definitions using patterns. We start by finding the object class using the following relationships:

- The average temperature of rail.→ (The right entity rail, as the object class, is overwritten because we have remaining relationships need to be visited)
- The average temperature for a track segment.→(last seen object class)
- The average temperature measured in Celsius.→ (The right entity is class word qualifier, so do not use it and back track to get track segment, instead)

We iterate through the following noun phrases to find the class word:

- Average temperature. → (Because temperature is a Strong Property and classified under measure in the Class I Railroad's list of Class Words, measure becomes the Class Word. Average Temperature becomes part of the Property)
- Track segment → (Skipped NP)
- Celsius → (Skipped NP)

Table 4.2: Accuracy results for extracting Data Element Names from definitions

Test Number	Number of Data Element Definitions	Passed	Failed	Accuracy
1	20	17	3	85%
2	20	16	4	80%
3	20	19	1	95%

Q1 - What is the accuracy of the NLP patterns for extracting data element names from definitions?

We evaluate our tool on three different experiments each of which contain 20 test examples. The domain expert chooses these 20 test examples, and he makes sure that each test trial contains a different set of test examples. Table 4.2 shows the accuracy results of the three test. We evaluate 20 Test Data Examples in the first test, and 17 of these 20 test examples pass the criteria set by the domain expert. The most important criterion is that the tool should extract correct noun phrases and relationships from a data element definition and, these extracted noun phrases and relationships should match the manually extracted noun phrases and relationships determined by the domain expert. We find that our tool fails to generate the correct noun phrases and relationships for the remaining 4 test examples. On the other hand, in the second and third tests, the tool determines the exact phrases to assign to each component of a Data Element Name, based on its definition, with an accuracy of 80%. 16 and 19 of the second and third test data examples pass the criterion with an accuracy 80% and 95%, respectively. The only passing criterion is that the tool should generate correct components of a data element name and, these extracted components should match the manually extracted components by the domain expert.

We create different test criteria for the third test, which will include:

- Five Data Elements that are already incorporated into the new KG.
- Five New Data Elements that are partially incorporated into the new KG, sharing known Object Classes.
- Five New Data Elements that are partially incorporated into the new KG, sharing known Properties.
- Five New Data Elements that are NOT incorporated into the new KG

On average, an experienced modeler can name a data element in 5-10 minutes. The question is how good is it? And did they duplicate something while naming the data element definition? They might have to go back and rename it. In some cases, it might be less than that 5 minutes. However, it can take longer than 5 minutes. On the other hand, our tool takes one minute to mine a data element name.

Q2 - Which are the cases of inaccuracy of the NLP patterns for extracting data element names?

Definition normalization. We find that it is harder to extract data element names from some data element definitions because these data element definitions need to be normalized so that it better fits the heuristics. To improve the coverage of these patterns, we have to deal with complicated phrases that do not fit into any one of created patterns. Normalizing a data element definition removes any extra phrase that hinders extraction of a specific component for a data element name.

Word inflections. We observe that the first and second tests have data element definitions that require word inflection, such as nounification, within the extracted verb phrases. In the third test, we improve the accuracy by implementing a function that uses NLTK library for extracting derivationally related forms.

For instance, we had to inflect the verb arrived to arrival to denote train arrival, which is an Object Class component of the data element name.

4.6 Conclusion

Data management is a key for many organizations, including Class I railroad. These organizations require a common and standardized meaning to communicate using their data element names, because they have many separate but overlapping databases. Data modelers describe data elements in plain text, and data element definitions contain domain-independent knowledge that require both natural language

processing and information extraction techniques. We develop a Data Element Naming Automation (DENA) tool that uses information extraction techniques to identify heuristic patterns and extract data element names from data element definitions. We achieve 95% accuracy using a knowledge graph and 80% accuracy with just using heuristic patterns. In our future work, we intend to include a preprocessing pipeline that extracts common misspelling, undesired synonyms, unknown and dictionary terms from data element definitions using our knowledge graph and one of the popular dictionary APIs.

CHAPTER 5

DENA: Data Element Naming Automation Tool

Semantic interoperability of data terminology has pressed obstacles to unify data elements from different data sources. Because different data modelers can define the same data elements in different ways, this might result in communication failures and erroneous operations. In this paper, we design and develop an automation tool that can intelligently preprocess a data element definition written by data modelers, extract a standardized data element name, and check for possible duplicates of the extracted data element name. The Data Element Naming Automation (DENA) tool uses a ranking algorithm that can sort similar data element names based on multiple features, including data element name, name rank, data element definition rank, and best rank. Our experiments are evaluated in two different ways. We compare our results with other semantic tools and evaluate them with a domain expert. The experiments show that DENA produces highly accurate results and correct identification of duplicate names. DENA introduces a novel technique for preprocessing data element definitions using NLP and additional rules that are incorporated into DENA's knowledge graph.

5.1 Introduction

Railroad industries use sophisticated technologies such as smart sensors, internet of trains, Positive Train Control (PTC), and Global Positioning System (GPS) tracking during the lifecycle of railroad equipment. These technologies create data at a faster speed, and they store them in various storage mediums. Because the railroad

business is so fragmented, railroad business providers create lifecycle data on an individual basis and store them in their repositories [92]. Sharing and integrating data within a railroad industry promotes data reusability, which translates to less data duplication, better decision support systems, and more production.

The smallest level of granularity of data that has a precise meaning and is both unique and of interest and use to the business is a data element. Data modelers write data element definitions to describe a data element by stating its complete and precise meaning in a unique English statement. They manually create unique data element names by which data elements can be identified within a business. They are the basis for all other technical data names. Enterprises aim to manage their data in a centralized manner by meeting the need for the common business meaning. The railroad industry, on the other hand, has yet to achieve a high level of interoperability [93].

Semantic interoperability concerns the problem of having two repositories that might not agree on a unique business description of the same data element name [94]. Data element names may differ among repositories in the fragmented railroad domain. Polysemy and synonymy are two main linguistic roadblocks to semantic integration of a diverse set of repositories [95]. Polysemy describes situations in which a single data element name has many meanings depending on the context. The disparity in meaning is due to the diversity and transient nature of data element definitions, as well as differences in writing styles curated by data modelers [96]. Synonymy, on the other hand, is related to the disparity of data element names for the same data element in different systems. Polysemy can result in a mismatch between two semantically distinct data element names and definitions, while synonymy can fail to aggregate comparable data elements.

Most similar tools use machine learning (ML) which requires training and parameter tuning to extract the semantic meaning of terms and their relationships [97, 98]. Our focus is on a semantic-based tool that uses NLP processors and a Neo4j database [99] to store the knowledge graph.

We use this approach to show that it can achieve very high accuracy without having to go through the trial and learning process of machine learning. compared with other semantic tools [100, 101], our tool gives slightly better results. Our focus is to use newer technologies such as Neo4j and Spacy [102] to develop a tool that uses a data element definition to derive a standardized data element name and then uses both the definition and name to find any preexisting duplicates in the railroad domain.

To fill this gap, this paper implements a data element naming automation (DENA) tool that preprocesses data element definitions, generates data element names, and checks for duplicate data element names and definitions. DENA validates the automatically generated data element name by searching for duplicates in the knowledge graph. This tool does not require human involvement in gathering design documents. Instead, DENA preprocesses a data element definition by identifying unknown terms, common misspellings, undesired synonyms, and American English terms. DENA uses a knowledge graph and 3rd-party APIs that incrementally build appropriate terminology to create consistent data element names.

Figure 5.1: A definition and an extracted data element name

5.2 Related Work

Some research papers have used ontologies, libraries, and taxonomies [103, 104]. A semantic resource focuses on the description of terms based on their lexical relationships. They lack comprehensive vocabularies in the railroad domain regardless of their years of effort because they require manual efforts from developers and they consume their time. Another line of research uses machine learning to identify the semantic relation between a new concept and the existing data element names [98, 97]. Based on the specified lexical relations supplied in WordNet, the algorithm detects possibly related concepts. This method would not scale well on matching terms in a railroad domain because WordNet is a broad lexicon that lacks concepts in several transportation sectors [98]. Another work uses NLP in addition to machine learning, but their tool is not fully automated and it expects human involvement for validating the automatically generated data sets [97].

Knowledge graphs need a preprocessing stage to give users accurate results. There are a couple of tools that conduct duplication detection in knowledge graphs [105, 100]. Duke and LIMES tools can preprocess text before they find duplicate matches in the knowledge graphs.

5.3 Data Element

We define what a data element means and how it is used in DENA. A data element is the lowest level of granularity of data that has a precise meaning and is both unique and of interest and use to the business. It roughly corresponds to an attribute of a relational database table. For example, many organizations generate a single data element for 10-digit telephone number data (without including international code),

while telecommunications enterprises use three separate and distinct data elements: area code (3 digits), a central office code (3 digits), and a line number (4 digits).

Organizations create metadata about the data it represents to attain governance. They can reach the goal when metadata is able to classify the data, represent the data as a given feature or quality, and attribute the feature or quality to entities. DENA helps data engineers achieve such a goal by assisting them to create unique data element names.

5.3.1 Data Element Naming

Once a clear (unambiguous and easy to understand), concise (short but still clear), accurate (correct), complete (containing all relevant information) business definition has been obtained, a new data element can be named by following the Naming Standard, which is defined by a set of rules that govern a data element name's construction. Figure 5.1 shows an example of a data element definition and data element name, that is extracted by DENA.

A Data Element Name (DEN) is a single noun phrase that consists of one or more words that represent the meaning of the data it entails. The following are the five components of a DEN and a description for each component:

- **Object Class:** is a word or phrase that represents a person, place, or thing in the real world that is recognized by the business with specific boundaries and meaning.
- **Property Qualifier:** is a word or phrase that, if needed, differentiates, limits, and/or modifies the meaning of a Property.
- **Property:** is a word or phrase that represents the main idea in a data element name. Each Property is a noun phrase that represents a discrete (separate and distinct) quality or feature that is common to all members of an Object Class,

for which the business wants to retain data. The property component can be further classified into:

  - Strong Property: A property whose meaning is clear, and whose Data Classification is intuitively obvious as belonging under only one of the existing approved Class Words. For example, width is a Strong Property under the Class Word measure.
  - Weak Property: A property whose meaning may not be clear, and whose Data Classification is not intuitively obvious as belonging under only one of the existing approved Class Words. I.e., the Property could be validly classified under two or more existing approved Class Words. For instance, speech is a Weak Property under two Class Words: video and audio.

- **Class Word Qualifier:** is a word or phrase that modifies the meaning of a Class Word by more accurately describing the type of data it represents.
- **Class Word:** is a word used for distinctly classifying data within organizations. It describes the content and type of information such as date, name, amount, etc., or its function such as Code or ID.

5.4 Framework and Algorithm

This section introduces the four stages of DENA. We will explain the first stage of the tool, which is the preprocessing stage of data element definitions. Then, when the data element definition is clear, and DENA is aware of all the terms that are used in the definition, DENA can generate a data element name from the preprocessed definition. DENA finalizes the pipeline by giving its users the ability to check for duplicates of the extracted data element name. Figure 5.2 shows the prepossessing stage of the entered definition, DENA NLP, which is the information extraction stage, data element naming assembly, and searching for duplicate components.

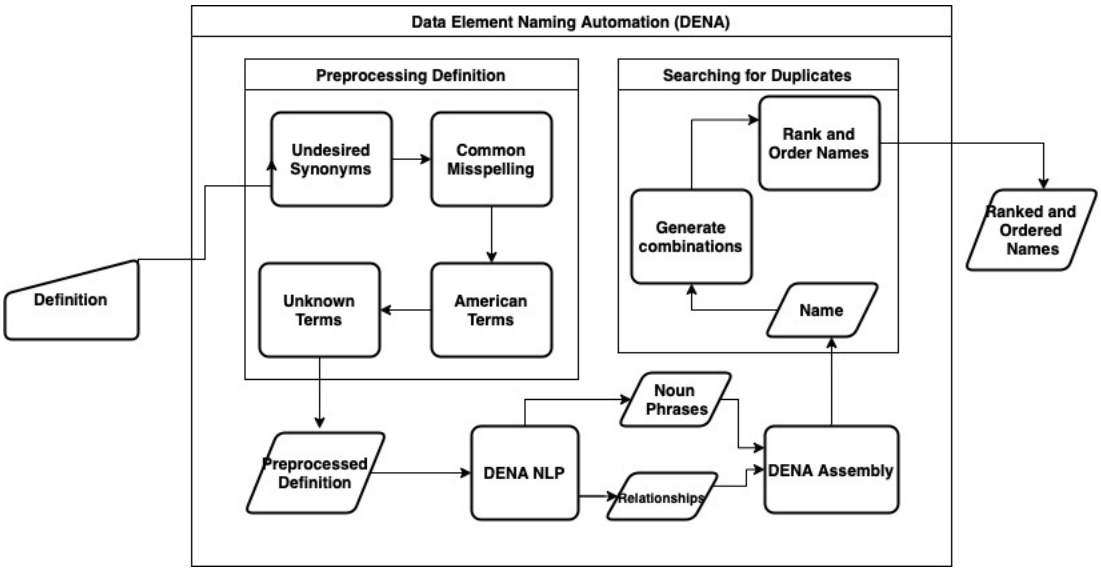


Figure 5.2: Four stages of DENA tool: Preprocessing the definition, natural language processing (DENA NLP), assembling the data element name, and searching for duplicate components.

5.4.1 Preprocessing Data Element Definitions

DENA can address the research question of how to incrementally build internal knowledge for data element names and externally gather terms that can improve the accuracy of DENA. First, we define different types of preprocessing phrases and give an example of each. Then, we explain the inner workings of one of the preprocessing phrases: undesired synonyms. Due to space constraints, we omit the explanation of how DENA finds common misspellings, American English terms, and unknown terms in definitions.

DENA can preprocess a data element definition when it finds a word or phrase that falls under one of four classifications: undesired synonyms, common misspellings, unknown terms, and American English terms.

- **Undesired Synonym** - A valid, correctly spelled term, that for a specific context shares the same definition with one or more company's terms, but is not the term that should be used in data naming. For instance, U.S. Dollar is

an undesired synonym for USD. DENA can highlight in yellow and annotate the phrase U.S. Dollars as an undesired synonym in the sentence. Figure 5.3 shows a definition entered by a user and a preprocessed definition where the undesired synonym is annotated and highlighted in yellow.

- **Common Misspelling** - A spelling that is often used for the intended term and that is either an actual misspelling or is another correct spelling for the same term that we have declared as misspelled so as not to allow multiple valid spellings of the same term. For example, "UUID ID" is a common misspelling for "UUID (Universally Unique Identifier)". Some data modelers use incorrect acronyms by adding an extra ID (Identifier). DENA can highlight in red and annotate the phrase "UUID ID" as a common misspelling in the sentence.
- **American English Dictionary Term** - A valid, correctly spelled American English word or phrase that can be found in an American English Dictionary, and that might or might not currently be Approved by an organization through DENA for use in data naming.
- **Unknown Term** - A string that cannot be found in any current Term references. For instance, "tie plate" is an unknown term to the internal knowledge of the organization. Namely, the company does not have the term stored in the knowledge graph. DENA annotates and highlights the unknown term in gray when it finds the term. Figure 5.3 shows the preprocessed definition containing the unknown term "tie plate" in the definition.

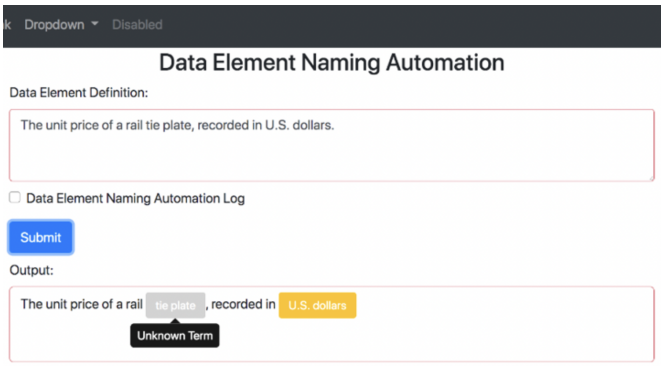


Figure 5.3: DENA preprocess a data element definition that contains two classifications of preprocessing errors: undesired synonym and unknown term.

Undesired Synonyms. We step through how DENA preprocesses definitions that contain undesired synonyms. Figure 5.4 illustrates how DENA approaches the preprocessing stage when a data element definition contains undesired synonyms. When DENA annotates and highlights undesired synonyms in yellow, a user can click on any undesired synonym. As a result, DENA interacts with the user by suggesting undesired synonyms corrections and similar terms and matching user term definitions.

Suggesting undesired synonym corrections and similar terms- The first part of the algorithm is auto-correction. DENA suggests undesired synonyms corrections and similar terms to the user when he clicks on any highlighted undesired synonyms. DENA provides the user with a list of the following corrections:

- One or more company's terms which includes preferred Abbreviations or Acronyms that have the User's Term as an Undesired Synonym or as a Common Misspelling.
- company's terms that are a near lexical match to the User's Term.
- American English Terms that are a near lexical match to the User's Term.

DENA also displays definitions for each of the terms above. The company's terms come from the company's internal knowledge, whereas DENA integrates Amer-

ican English terms by using unofficial Google Dictionary API [1] and DataMuse API [2]. DENA gives the user the option to re-enter the term or to declare that his Term is correct as entered.

The auto-correction part contains three functions that DENA uses to normalize definitions using approved terms from the knowledge graph including verifying undesired synonyms, confirming the replacement of undesired synonyms, sending the company's term change request for approval, and selecting user term definition.

1. **Verify Undesired Synonym**: DENA verifies the chosen undesired synonym with the user by confirming that the term is correct. When the user acknowledges that his term is correct, he ignores the list of suggested terms and wants to find only the right definition for his highlighted undesired synonym. As a result, DENA provides the user with a list of one or more company's terms which includes Preferred Abbreviations/Acronyms and their definitions, where the company's term has the User's Term as an Undesired Synonym. Then, DENA prompts the User to select the company's term Definition that semantically matches the User's intended Definition for their Term or indicate that there is no semantic match on the list provided.
2. **Confirm Replacement of Undesired Synonym** : DENA confirms with the user to replace his Term (the Undesired Synonym) with the company's term (which may be a Preferred Abbreviation/Acronym) having the User's chosen Term Definition, or change the company's term to the user's Term (the Undesired Synonym in yellow). environment.
3. **Send the Company's Term Change Request for Approval**: For the User Term and User Term Definition, DENA allows users to submit an Abbrevia-

[1] https://github.com/meetDeveloper/freeDictionaryAPI

[2] https://www.datamuse.com/api/

tions and Acronyms Request to Abbreviation Support to change the current company's term to the user's Term.

4. **Select User Term Definition**: DENA gives users the ability to select a Definition for their entered term from the list of one or more Definitions found in the source reference through dictionary APIs or manually input their definition, if no definitions on the list are a semantic match. For example, if the term is from the American English Dictionary, the list would be of all definitions for that Term in that Dictionary.

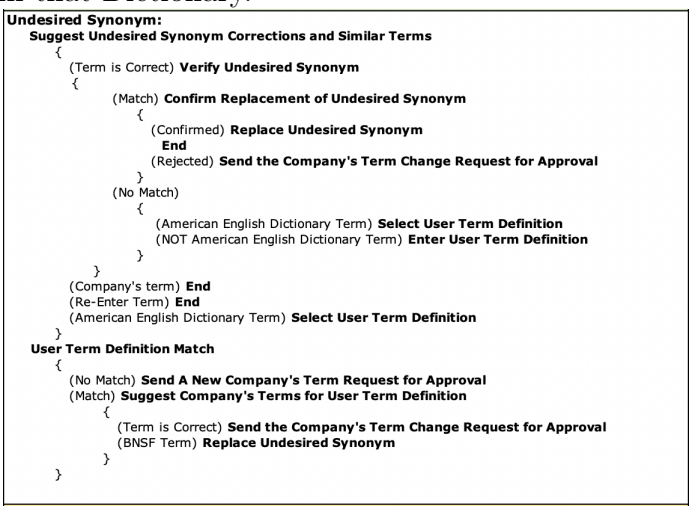


Figure 5.4: Undesired Synonym Algorithm

Matching user term definition - DENA determines if the user's term definition is a semantic match or near semantic match to any existing company's term definition. Sending a new company's term request for approval, suggesting the company's terms for user term definitions, and sending the company's term change request are three steps that contribute to incrementally building the company's knowledge graph.

1. **Send A New Company's Term Request for Approval**: For the user term and user term definition, DENA keeps track of submitted company's abbreviations and acronyms requests and sends them to the knowledge graph for a new company's term.

2. **Suggest the Company's Terms for User Term Definition**: DENA provides users with a list of the company's terms (which includes Preferred Abbreviations/Acronyms) and Definitions where users' definition is a semantic match or near semantic match to any company's term definition, and give users the option to declare that their Term is correct as entered.
3. **Send the Company's Term Change Request for Approval**: For the user term and user term definition, DENA submits an Abbreviations and Acronyms Request to the knowledge graph to change the current company's term to the user's Term.

5.4.2 Data Element Name Extraction

In this section, we explain how DENA extracts the data element name from the preprocessed definition. We published the findings of the naming extraction component in previous work (Anonymous 2022). We use Spacy [102], an NLP framework, to work with text as it provides pretrained models of dependency tags for each word in a sentence. Figure 5.2 shows DENA NLP and DENA assembly as steps of data element name extraction.

DENA NLP. We devise heuristics that use dependency tags as patterns to extract the five components of the data element name. DENA utilizes Spacy's base noun phrases. Space omits the implementation of prepositional phrases, coordination, or relative clauses in their models. We support DENA with component phrases that generate these complicated phrases. We utilize the output of Spacy's dependency parser to generate a list of heuristic rules for distilling relationships from definitions. In the context of relationship extraction (RE), we use Spacy's Part of Speech (POS) tagger to create candidate phrases by matching them against heuristic patterns cre-

ated using dependency tags. POS is also beneficial in providing lexical information needed to create relationship models.

DENA Assembly. We also use heuristics to assemble the five components of the name. DENA assembly accepts noun phrases and relationships as input to facilitate assembling the components of the name.

5.4.3 Searching for Duplicates of the Extracted Data Element Name

We introduce and explain the last stage of the tool in this section. DENA searches all existing data element names and definitions and determines the probability that the new data element has already been stored in the knowledge graph and is in fact redundant. DENA performs the search based on which words it assigns to each Data Element Name Component. If likely to be redundant, DENA allows the user to review the names and descriptions of the existing data element(s) that are possible matches.

DENA identifies all existing data element names, definitions, and the combinations thereof that might constitute a duplicate match to the new data element, providing a stated match confidence percentage. Table 4.1 shows the list of the 13 patterns that DENA uses to match against generated combinations of the name. As a result, DENA generates DEN search name combinations, assigns a pattern ranking from the list, and calculates a ranking factor based on the matching words in the search name combination and its total word count.

We use a ranking factor that ranks all matches of generated combinations concerning extracted data element name by DENA. The ranking factor is described by the following equation

$$R = (M/len(N))(M \cdot S)$$

Table 5.1: Patterns for searching combinations from names

Pattern Ranking	**DEN Search Component Pattern**
1	Object Class + Property Qualifier + Property + Class Word Qualifier + (Class Word)
2	Object Class + Property Qualifier + Property + (Class Word)
3	Object Class + Property + Class Word Qualifier + (Class Word)
4	Object Class + Property + (Class Word)
5	Property Qualifier + Property + Class Word Qualifier + (Class Word)
6	Property Qualifier + Property + (Class Word)
7	Property + Class Word Qualifier + (Class Word)
8	Property + (Class Word)
9	Object Class + Property Qualifier + Class Word Qualifier + (Class Word)
10	Object Class + Property Qualifier + (Class Word)
11	Object Class + Class Word Qualifier + (Class Word)
12	Object Class + Class Word
12	Class Word

where R is a ranking factor, N is the extracted data element name from DENA, M and S are the matching words and total words in the search name combination, respectively.

DENA outputs name rank and definition rank separately. The name rank uses ranking factor results of the matched search name combinations that are found in the knowledge graph. Then, DENA calculates the best rank between the two ranks and displays it to the user. A complete example of the data element name and search name combinations can be found online.[3]

[3] https://docs.google.com/spreadsheets/d/1meQeaHUnSLoSfW4Zm9XtANGj6KfSiWwyWFh9LQlD8cY/edit?usp=sharing

5.5 Results

We tested the tool twice with the domain expert. The first test used a preliminary version of DENA and yielded only about a 60% accuracy. Based on these results, we created a more sophisticated version of DENA. We conducted the second test on twenty data element definitions using the new version of DENA. This test yielded about a 95% success, with 18 out of 20 names being exact matches with expected results and an additional 5% contributed to partial matches. This increase occurred as a result of using a knowledge graph along with information extraction heuristics to extract the name.

We refer to the survey conducted on duplication detection tools to evaluate our duplicate check functionality [101]. The survey claims that Duke has the best execution time and accuracy at finding duplicates in one dataset, so we evaluate DENA against Duke [100]. Duke has some preprocessing techniques that include basic string and parsing cleaners. On the other hand, DENA cleans and normalizes text by choosing terms that exist in the knowledge graph. If they do not exist, DENA incrementally builds terminology with the user and enhances the accuracy of extracting the name and finding duplicates.

We tested our dataset using the Duke tool. We used the record linkage mode since we compared one record against a set of records to find duplicates. We inject the dataset with 20 duplicates. We run DENA and it is able to detect the 20 duplicates along with its name rank and definition rank. Table **??** shows F-score for all different metrics that can be tuned in Duke. DENA outperforms Duke for all metrics except the QGramComparator metric where it gives a comparable performance. The QGramComparator gives the best precision and recall in combination. QGramComparator only misses one record when it has a poor definition. Duke is not able to find such records, while DENA finds duplicates in names and definitions individually. As

Table 5.2: Comparison table between Duke and DENA

User Metric	TP	FN	TN	FP	F-score
JaroWinkler	17	3	62	58	%35.7
Levenstein	6	14	120	0	%46.1
QGramComparator	19	1	120	0	%97.4
ExactComparator	2	18	120	0	%18.2
DifferentComparator	19	1	118	2	%92.4
DiceCoefficientComparator	11	9	118	2	%66.5
DENA's RankingFactor	20	0	120	0	%100

an improvement over Duke, we implement some suggested improvements mentioned in the survey. DENA shows status information by browsing through logs. These logs show what terms and phrases are normalized and swapped to desired synonyms in the knowledge graph. DENA also shows statistical results; it displays sorted results of ranked definitions and names along with the accuracy of the ranking factor.

5.6 Conclusion

We present a software tool, DENA, that cleans data element definitions by normalizing terms with approved terms in the knowledge graph, extracts standardized data element names from definitions and finds duplicates in the knowledge graph, and alerts users with record linkage capability that avoids duplication. We compared our results with a pioneer tool in the literature. DENA also introduces a novel technique for preprocessing data element definitions using NLP and additional rules that are incorporated into DENA's knowledge graph.

CHAPTER 6

CONCLUSION

6.1 Summary of Contributions

The first part of this dissertation made use of the concepts of antipatterns such as SQL antipatterns and framework-specific antipatterns in order to create useful and efficient software for database access. The second part of the dissertation analyzes DENA extraction patterns, DENA assemply patterns and DENA search patterns to standardize database naming in large organizations. We proposed categorizations and applications to work with both antipatterns and patterns. In chapter 2, we introduced a categorization entitled as A Survey of SQL Antipatterns in which we reviewed the literature on SQL schema antipatterns, SQL query antipatterns, SQL security breaches and tools for identifying SQL antipatterns. After that, in chapter 3, we introduced a categorization of architecture-specific antipatterns and ORM antipatterns based on both schema and query. In chapter 4, we proposed a novel method to extract standardized data element names from natural language definitions. Then, in the last chapter 5 we proposed our Data Element Naming Automation (DENA) with techniques for preprocessing and finding duplicates of data element definitions and names, respectively.

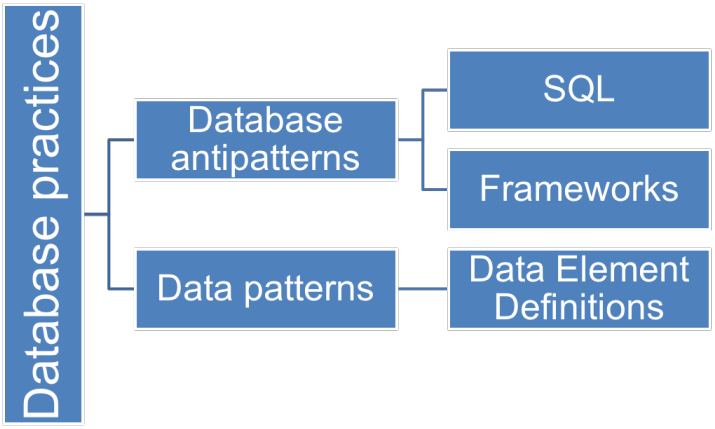
The main contributions of the first part of the dissertation are the following:

- Two categorizations of antipatterns representing bad practices of database programming and design.
- Analysis of the impact of each categorized database-related antipatterns.

The main contributions of the second part of the dissertation are the following:

- A prototype system for the naming problem that many organizations face involving standardization and duplication detection.
- A preprocessing of data element definitions.
- Heusirics rules for extracting data element names from data element definitions and assembly patterns for the final deliverable data element names.
- A duplication detection stage of both data element definitions and data element names.

Figure 6.1: Summary of contribution



6.2 Future work

In our future work related to antipatterns, we intend to:

- Analyze the impact of antipatterns on other nonfunctional requirements of software systems.
- Study the interaction of antipatterns and find casual relationships.
- Build a tool that can identify the antipatterns categorizaed in this dissertation.

In our future work related to domain name standardization, we plan to do the following:

- Add more data element definitions and names from other domains.
- Apply one of graph neural networks algorithms such as GraphSage that can study the structure of graphs and input a few labeled data.
- Conduct use case studies with potential users of DENA.

Bibliography

[1] B. Alshemaimri, R. Elmasri, T. Alsahfi, and M. Almotairi, "A survey of problematic database code fragments in software systems," Engineering Reports, vol. 3, no. 10, 2021.

[2] B. Karwin, SQL antipatterns avoiding the pitfalls of database programming. The Pragmatic Bookshelf, 2014.

[3] "119 sql code smells," https://www.red-gate.com/library/119-sql-code-smells, Last accessed on 2019-02-20.

[4] "Stack overflow developer survey 2019," https://insights.stackoverflow.com/survey/2019, Last accessed on 2018-11-30.

[5] R. Elmasri and S. Navathe, Fundamentals of database systems. Pearson Education, 2017.

[6] M. Venkatrao and M. Pizzo, "Sql/cli; a new binding style for sql," SIGMOD Rec., vol. 24, no. 4, pp. 72–77, Dec. 1995. [Online]. Available: http://doi.acm.org/10.1145/219713.219763

[7] "The easiest way to write sql in java," https://www.jooq.org/, Last accessed on 2019-01-14.

[8] M. Goeminne and T. Mens, "Towards a survival analysis of database framework usage in java projects," 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015.

[9] C. Ireland, D. Bowers, M. Newton, and K. Waugh, "A classification of object-relational impedance mismatch," 2009 First International Conference on Advances in Databases, Knowledge, and Data Applications, 2009.

[10] T. Sharma, M. Fragkoulis, S. Rizou, M. Bruntink, and D. Spinellis, "Smelly relations," Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP 18, 2018.

[11] K. Beck, "Once and only once," http://wiki.c2.com/?OnceAndOnlyOnce, Last accessed on 2019-01-25.

[12] A. Koenig, Patterns and Antipatterns. USA: Cambridge University Press, 1998, p. 383–389.

[13] M. Linares-Vásquez, S. Klock, C. Mcmillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc, "Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps," Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014, 2014.

[14] T. Erl, Service-oriented architecture: concepts, technology, and design. Prentice Hall Professional Technical Reference, 2016.

[15] M. Fowler, "Microservices," https://www.martinfowler.com/articles/microservices.html, Last accessed on 2019-02-6.

[16] G. Krasner and S. Pope, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," J Object-Orient. Prog., vol. 1(3), 01 2000.

[17] P. Robson and S. Faroult, The art of SQL. Oreilly, 2006.

[18] "what are the most common sql anti-patterns?" 2009, http://stackoverflow.com/questions/346659/, Last accessed on 2018-3-02.

[19] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," Proceedings of the 36th International Conference on Software Engineering - ICSE 2014, 2014.

[20] L. Eder, "10 common mistakes java developers make when writing sql," 2013, https://blog.jooq.org/2013/07/30/10-common-mistakes-java-developers-make-when-writing-sql/ , Last accessed on 2019-07-17.

[21] Y. Lyu, A. Alotaibi, and W. G. Halfond, "Quantifying the performance impact of sql antipatterns on mobile applications," 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019.

[22] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "How not to structure your database-backed web applications," Proceedings of the 40th International Conference on Software Engineering, 2018.

[23] E. Eessaar, "On query-based search of possible design flaws of sql databases," in Innovations and Advances in Computing, Informatics, Systems Sciences, Networking and Engineering, T. Sobh and K. Elleithy, Eds. Cham: Springer International Publishing, 2015, pp. 53–60.

[24] E. Eessaar and J. Voronova, "Using sql queries to evaluate the design of sql databases," Lecture Notes in Electrical Engineering New Trends in Networking, Computing, E-learning, Systems Sciences, and Engineering, p. 179–186, Aug 2014.

[25] P. Khumnin and T. Senivongse, "Sql antipatterns detection and database refactoring process," 2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2017.

[26] J. Delplanque, A. Etien, O. Auverlot, T. Mens, N. Anquetil, and S. Ducasse, "Codecritics applied to database schema: Challenges and first results," 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017.

[27] C. Nagy and A. Cleve, "A static code smell detector for sql queries embedded in java code," 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2017.

[28] P. Gulutzan and T. Pelzer, SQL performance tuning:. Addison-Wesley, 2006.

[29] "Mysql: Alternate solution of sql server's hierarchyid datatype," 2014, https://stackoverflow.com/questions/24525634/mysql-alternate-solution-of-sql-servers-hierarchyid-datatype/37000707, Last accessed on 2019-7-2.

[30] R. S. Chen, P. Nadkarni, L. Marenco, F. Levin, J. Erdos, and P. L. Miller, "Exploring performance issues for a clinical database organized using an entity-attribute-value representation," Journal of the American Medical Informatics Association, vol. 7, no. 5, p. 475–487, Jan 2000.

[31] "Normalization: What does "repeating groups" mean?" 2014, https://stackoverflow.com/questions/23194292/normalization-what-does-repeating-groups-mean, Last accessed on 2019-6-16.

[32] "Mysql large table sharding to smaller table based on unique id," 2019, https://stackoverflow.com/questions/54283154/mysql-large-table-sharding-to-smaller-table-based-on-unique-id, Last accessed on 2019-4-05.

[33] "Mysql enum performance advantage," 2009, https://stackoverflow.com/questions/766299/mysql-enum-performance-advantage, Last accessed on 2019-7-7.

[34] "Will more indexes on a table affect performance?" 2013, https://dba.stackexchange.com/questions/17830/will-more-indexes-on-a-table-affect-performance, Last accessed on 2019-4-11.

[35] G. Sanders and S. Shin, "Denormalization effects on performance of rdbms," Proceedings of the 34th Annual Hawaii International Conference on System Sciences.

[36] "How far should we go with normalization?" 2001, https://dba.stackexchange.com/questions/505/how-far-should-you-go-with-normalization, Last accessed on 2019-4-14.

[37] "What will happen if a database of 30 tables is not normalized?" 2017, https://www.quora.com/What-will-happen-if-a-database-of-30-tables-is-not-normalized, Last accessed on 2019-5-16.

[38] V. Bilopavlović, "Antipatterns in data access, part 1-memory joins," 2019, https://www.linkedin.com/pulse/

antipatterns-data-access-part-1-vedran-bilopavlović/ , Last accessed on 2019-7-17.

[39] R. Johari and P. Sharma, "A survey on web application vulnerabilities (sqlia, xss) exploitation and security engine for sql injection," 2012 International Conference on Communication Systems and Network Technologies, pp. 453–458, 2012.

[40] "the owasp enterprise security api," https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API, accessed: 2022-03-10.

[41] M. Aniche, G. Bavota, C. Treude, A. V. Deursen, and M. A. Gerosa, "A validated set of smells in model-view-controller architectures," 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016.

[42] M. Fowler, "P of eaa: Repository," https://martinfowler.com/eaaCatalog/repository.html, Last accessed on 2019-3-22.

[43] F. Palma and N. Mohay, "A study on the taxonomy of service antipatterns," 2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP), 2015.

[44] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, and A. Zimmermann, "Towards a collaborative repository for the documentation of service-based antipatterns and bad smells," 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), 2019.

[45] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," IEEE Software, vol. 35, no. 3, p. 56–62, 2018.

[46] J. Corbin and A. L. Strauss, Basics of qualitative research: techniques and procedures for developing grounded theory. SAGE, 2015.

[47] "Anemic domain model," 2003, https://www.martinfowler.com/bliki/AnemicDomainModel.html, Last accessed on 2018-3-16.

[48] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting problems in the database access code of large scale systems," Proceedings of the 38th International Conference on Software Engineering Companion - ICSE 16, 2016.

[49] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," IEEE Transactions on Software Engineering, vol. 42, no. 12, p. 1148–1161, Jan 2016.

[50] "Avoiding microservice megadisasters," presentation at 2017 NDC London Conf, Ed., 2017, https://www.youtube.com/watch?v=gfh-VCTwMw8, Last accessed on 2019-5-16.

[51] M. Fowler, "Shared database," https://microservices.io/patterns/data/shared-database.html, Last accessed on 2019-4-22.

[52] "What are the drawbacks to the activerecord pattern?" 2011, https://softwareengineering.stackexchange.com/questions/70291/what-are-the-drawbacks-to-the-activerecord-pattern, Last accessed on 2019-4-28.

[53] "How should i manage generic repository pattern when the works of different entities are pretty much different?" 2018, https://stackoverflow.com/questions/49974181/how-should-i-manage-generic-repository-pattern-when-the-works-of-different-entit.htm , Last accessed on 2019-7-15.

[54] "Openstreetmap," https://www.openstreetmap.org, Last accessed on 2019-5-08.

[55] "Why would i use a transient attribute to represent a derived read only property?" 2018, https://stackoverflow.com/questions/7957130/why-would-i-use-a-transient-attribute-to-represent-a-derived-read-only-property, Last accessed on 2019-5-12.

[56] J. Munhoz, "Hibernate and the n+1 selections problem," 2019, https://medium.com/quintoandar-tech-blog/hibernate-and-the-n-1-selections-problem-c497710fa3fe , Last accessed on 2019-8-21.

[57] C. U. Smith and L. G. Williams, Performance solutions: a practical guide to creating responsive, scalable software, 2002.

[58] V. Bilopavlović, "Antipatterns in data access, part 2- nested select," 2019, https://www.linkedin.com/pulse/antipatterns-data-access-part-2-nested-selects-vedran-bilopavlovi%C4%87/ , Last accessed on 2019-7-17.

[59] D. Taibi, V. Lenarduzzi, and C. Pahl, "Microservices anti-patterns: A taxonomy," Microservices, p. 111–128, 2019.

[60] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen, "Code smells for model-view-controller architectures," Empirical Software Engineering, vol. 23, no. 4, p. 2121–2157, 2017.

[61] "Are multiple database calls really significant with a network call for a web api?" 2015, https://softwareengineering.stackexchange.com/questions/255275/are-mutliple-database-calls-really-significant-with-a-network-call-for-a-web-api , Last accessed on 2019-7-26.

[62] K. Alam, "Why to use service layer in spring mvc," 2018, https://medium.com/stackavenue/why-to-use-service-layer-in-spring-mvc-5f4fc52643c0 , Last accessed on 2019-8-12.

[63] "Java persistence layer in regards to service/repository layers," 2015, https://stackoverflow.com/questions/25313231/java-persistence-layer-in-regards-to-service-repository-layer , Last accessed on 2019-8-12.

[64] B. Dudney, S. Asbury, J. K. Krozak, and Wittkopf, J2EE AntiPatterns. John Wiley Sons, 2003.

[65] M. Fowler, Refactoring: improving the design of existing code. Addison-Wesley, 2019.

[66] B. Aaronson, "How do you avoid getters and setters?" Mar 1963. [Online]. Available: https://softwareengineering.stackexchange.com/questions/284215/how-do-you-avoid-getters-and-setters

[67] "Wcf decision: One service multiple contracts or many services," 2014, https://stackoverflow.com/questions/17424392/

wcf-decision-one-service-multiple-contracts-or-many-services , Last accessed on 2019-8-18.

[68] "Spring transnational slows down complete process," 2018, https://stackoverflow.com/questions/46214484/spring-transactional-slows-down-complete-process , Last accessed on 2019-8-27.

[69] "Spring @transactional annotation: Self invocations," 2015, https://stackoverflow.com/questions/23931698/spring-transactional-annotation-self-invocation , Last accessed on 2019-9-18.

[70] "@transactional spring chaining self invoking," 2019, https://stackoverflow.com/questions/54698150/transactional-spring-chaining-and-self-invoking , Last accessed on 2019-9-28.

[71] "Why do i need transaction for read-only operations?" 2013, https://stackoverflow.com/questions/13539213/why-do-i-need-transaction-in-hibernate-for-read-only-operations , Last accessed on 2019-9-08.

[72] "What hurts maintainability?" 2012, https://softwareengineering.stackexchange.com/questions/123293/what-hurts-maintainability/123296 , Last accessed on 2019-10-05.

[73] "How to change the ordering of sql execution in hibernate," 2014, https://stackoverflow.com/questions/20395543/how-to-change-the-ordering-of-sql-execution-in-hibernate , Last accessed on 2020-10-14.

[74] "Which is more expensive for loop or database call?" 2010, https://stackoverflow.com/questions/1390757/which-is-more-expensive-for-loop-or-database-call. , Last accessed on 2019-9-10.

[75] "Rails performance - what do you need to know," https://www.airpair.com/ruby-on-rails/performance#1-introduction , Last accessed on 2019-9-13.

[76] "Hibernate criteria query to get specific columns," 2013, https://stackoverflow.com/questions/11626761/hibernate-criteria-query-to-get-specific-columns , Last accessed on 2019-9-12.

[77] T. Janssen, "Hibernate tips: Use the querycache to avoid additional queries," 2013, https://thoughts-on-java.org/hibernate-tips-use-querycache-avoid-additional-queries/ , Last accessed on 2019-9-16.

[78] A. Rodi, "Partial rendering performance in rails," 2017, https://medium.com/@coorasse/partial-rendering-performance-in-rails-101fdfb6ffb9 , Last accessed on 2019-9-16.

[79] S. Shao, Z. Qiu, X. Yu, W. Yang, G. Jin, T. Xie, and X. Wu, "Database-access performance antipatterns in database-backed web applications," 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020.

[80] T. Janssen, "5 common hibernate mistakes that cause dozens of unexpected queries," https://thoughts-on-java.org/5-common-hibernate-mistakes-that-cause-dozens-of-unexpected-queries/ , Last accessed on 2019-9-16.

[81] ——, "10 common hibernate mistakes that cripple your performance," 2019, https://thoughts-on-java.org/common-hibernate-mistakes-cripple-performance/. , Last accessed on 2019-9-05.

[82] "Information technology — Specification and standardization of data elements — Part 5: Naming and identification principles for data elements," International Organization for Standardization, Geneva, CH, Standard, 1995.

[83] R. Lloyd and C. I. S. Writer, "Metric mishap caused loss of nasa orbiter," CNN Interactive, 1999.

[84] M. Banko, M. J. Cafarella, S. Soderland, M. A. Broadhead, and O. Etzioni, "Open information extraction from the web," in CACM, 2008.

[85] M. A. Hearst, "Automatic acquisition of hyponyms from large text corpora," in Coling 1992 volume 2: The 15th international conference on computational linguistics, 1992.

[86] M. Alohaly, H. Takabi, and E. Blanco, "Automated extraction of attributes from natural language attribute-based access control (abac) policies," Cybersecurity, vol. 2, no. 1, p. 2, 2019.

[87] N. Calzolari, "Acquiring and representing semantic information in a lexical knowledge base," in Workshop of SIGLEX (Special Interest Group within ACL on the Lexicon). Springer, 1991, pp. 235–243.

[88] P. Vossen, "Converting data from a lexical database to a knowledge base (technical report esprit bra-3030 acquilex wp 027). amsterdam, the netherlands: University of amsterdam," English Department, 1991.

[89] ——, "The automatic construction of a knowledge base from dictionaries: a combination of techniques," Euralex'92 Proceedings I-II, Tampere, Finland, 1992.

[90] P. Vossen and A. Copestake, "Untangling definition structure into knowledge representation," in Inheritance, defaults and the lexicon. Cambridge University Press, 1994, pp. 246–274.

[91] A. Copestake, "The lkb: a system for representing lexical information extracted from machine-readable dictionaries," in Proceedings of the ACQUILEX Workshop on Default Inheritance in the Lexicon, Cambridge, 1991.

[92] F. Harrison, M. Gordon, and G. R. Allen, "Leadership guide for strategic information management for state departments of transportation," NCHRP Report, 2016.

[93] N. Lefler, "Roadway safety data interoperability between local and state agencies," NCHRP Synthesis of Highway Practice, 2014.

[94] S. Heiler, "Semantic interoperability," ACM Comput. Surv., vol. 27, pp. 271–273, 1995.

[95] N. Noy, "Semantic integration: a survey of ontology-based approaches," SIGMOD Rec., vol. 33, pp. 65–70, 2004.

[96] C. M. Walton, D. P. K. Seedah, C. Choubassi, H. Wu, A. K. Ehlert, R. Harrison, L. Loftus-Otway, J. T. Harvey, J. Meyer, J. Calhoun, L. Maloney, S. Cropley, and F. Annett, "Implementing the freight transportation data architecture: Data element dictionary," 2015.

[97] T. Le and H. D. Jeong, "Nlp-based approach to semantic classification of heterogeneous transportation asset data terminology," Journal of Computing in Civil Engineering, vol. 31, p. 04017057, 2017.

[98] J. Zhang and N. El-Gohary, "Automated information transformation for automated regulatory compliance checking in construction," Journal of Computing in Civil Engineering, vol. 29, 2015.

[99] I. S. Robinson, J. Webber, and E. Eifrém, "Graph databases," 2013.

[100] L. M. Garshol and A. Borge, "Hafslund sesam - an archive on semantics," in ESWC, 2013.

[101] E. Huaman, E. Kärle, and D. A. Fensel, "Duplication detection in knowledge graphs: Literature and tools," ArXiv, vol. abs/2004.08257, 2020.

[102] M. Honnibal and M. Johnson, "An improved non-monotonic transition system for dependency parsing," in Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. Lisbon, Portugal: Association for Computational Linguistics, September 2015, pp. 1373–1378. [Online]. Available: https://aclweb.org/anthology/D/D15/D15-1162

[103] J. Tutcher, J. M. Easton, and C. Roberts, "Enabling data integration in the rail industry using rdf and owl: The racoon ontology," ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part A: Civil Engineering, vol. 3, 2017.

[104] S. Bischof and G. Schenner, "Rail topology ontology: A rail infrastructure base ontology," in SEMWEB, 2021.

[105] A.-C. N. Ngomo, M. A. Sherif, K. Georgala, M. M. Hassan, K. Dreßler, K. Lyko, D. Obraczka, and T. Soru, "Limes: A framework for link discovery on the semantic web," KI - Künstliche Intelligenz, pp. 1–11, 2021.

BIOGRAPHICAL STATEMENT

Bader Alshemaimri is a PhD candidate in the Department of Computer Science and Engineering, at the University of Texas at Arlington and a lecturer at the Software Engineering Department at King Saud University. He completed his Bachelor in 2012 and his Master's in 2015 in Computer Science. His research interest is the artificial intelligence, natural language processing, software engineering for intelligent systems.