UNIVERSIDADE FEDERAL FLUMINENSE

RAQUEL MAXIMINO DE BARROS SANTOS

ANALYZING THE ADOPTION OF DATABASE ENGINES THROUGHOUT THE LIFE CYCLE OF JAVA OPEN SOURCE PROJECTS

NITERÓI

2023

RAQUEL MAXIMINO DE BARROS SANTOS

ANALYZING THE ADOPTION OF DATABASE ENGINES THROUGHOUT THE LIFE CYCLE OF JAVA OPEN SOURCE PROJECTS

Master's dissertation presented to the Graduate Program in Computing at the Universidade Federal Fluminense, as a partial requirement for obtaining the Master's Degree in Computing. Concentration area: Computer Science.

Advisor:
LEONARDO GRESTA PAULINO MURTA

Co-advisor:
VANESSA BRAGANHOLO MURTA

NITERÓI

2023

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

S237a Santos, Raquel Maximino de Barros
Analyzing the adoption of Database Engines throughout the life cycle of Java open source projects / Raquel Maximino de Barros Santos. - 2023.
77 f.: il.

Orientador: Leonardo Gresta Paulino Murta.
Coorientador: Vanessa Braganholo Murta.
Dissertação (mestrado)-Universidade Federal Fluminense, Instituto de Computação, Niterói, 2023.

1. Database Engines Adoption.. 2. DBMS Adoption.. 3. Projects Life Cycle. 4. Java. 5. Produção intelectual. I. Murta, Leonardo Gresta Paulino, orientador. II. Murta, Vanessa Braganholo, coorientadora. III. Universidade Federal Fluminense. Instituto de Computação.IV. Título.

CDD - XXX

Bibliotecário responsável: Debora do Nascimento - CRB7/6368

RAQUEL MAXIMINO DE BARROS SANTOS

ANALYZING THE ADOPTION OF DATABASE ENGINES THROUGHOUT THE LIFE CYCLE OF JAVA OPEN SOURCE PROJECTS

Master's dissertation presented to the Graduate Program in Computing at the Universidade Federal Fluminense, as a partial requirement for obtaining the Master's Degree in Computing. Concentration area: Computer Science.

Approved in November 2023.

EXAMINATION BOARD

Prof. Leonardo Gresta Paulino Murta - Advisor, UFF

Vanessa Braganholo Murta

Prof. Vanessa Braganholo Murta - Advisor, UFF

Prof. Daniel Cardoso Moraes de Oliveira, UFF

Igor Scaliante Wiese

Prof. Igor Scaliante Wiese, UTFPR

Niterói

2023

*Dedication: For **God**, my **family** and to my **mother-in-law**, **Regina Célia de Barros** (In Memoriam).*

Acknowledgments

First and foremost, I offer my gratitude to God, whose unshakable support has guided me throughout this journey. It is through His boundless grace that I stand at this milestone today.

To my husband and son, Eduardo and Enzo, my sincere appreciations for their unconditionally support, understanding during my prolonged absences, and selfless sacrifices that allowed me to immerse myself in my pursuits.

To my parents, who have consistently incentivized me to go further, I am deeply thankful. Their belief in me has been a constant source of inspiration.

To my mother-in-law Regina Célia (in memoriam), an example of resilience and a great supporter. The nine months she was hospitalized allowed me to carry out this work with tears and uncertainty, but the faith that she rests in the Heavenly Father's arms encouraged me to continue.

I am profoundly grateful to the Brazilian Navy for granting me the invaluable opportunity to dedicate myself fully to my master's degree, without which I would never have ventured into these unknown waters.

I also wish to express very thankful to my advisors, Leonardo Murta and Vanessa Braganholo, for their encouragement to transcend my own boundaries and precious learning. My UFF (Fluminense Federal University) colleagues for the tips and help, especially my dear friend Camila, whose partnership encouraged me to move forward.

To my companions within the Brazilian Navy, who shared my aspiration for this course and offered encouragement to go ahead, I offer my heartfelt appreciation. I reserve special gratitude for my CASNAVIANOS friends, whose guidance, feedback, and shared experiences have been fundamental.

To my fellow members in Christ, for their empathetic listening and uplifting words, which have been a source of consolation during moments of emotional exhaustion.

Lastly, to all those who have provided their support, even through the smallest gestures, I extend my profound appreciation. Your encouragement has been a driving force throughout my journey.

Agradecimentos

Em primeiro lugar e acima de tudo, agradeço a Deus, cujo apoio inabalável me guiou ao longo desta jornada. É mediante à Sua graça sem limites que hoje me encontro neste marco.

Ao meu marido e ao meu filho, Eduardo e Enzo, os meus sinceros agradecimentos pelo apoio incondicional, pela compreensão durante as minhas ausências prolongadas e pelos sacrifícios altruístas que me permitiram mergulhar nos meus objetivos.

Aos meus pais, que sempre me incentivaram a ir mais longe, estou profundamente grata. A sua crença em mim tem sido uma fonte constante de inspiração.

À minha sogra Regina Célia (em memória), exemplo de resiliência e grande apoiadora. Os nove meses que ela esteve hospitalizada permitiram-me realizar este trabalho com lágrimas e incertezas. Mas a fé de que ela repousa nos braços do Pai Celestial me incentivou a prosseguir.

Agradeço profundamente à Marinha do Brasil por ter me concedido a inestimável oportunidade de me dedicar integralmente ao meu mestrado, sem a qual eu jamais teria me aventurado por essas águas desconhecidas.

Também quero expressar meus sinceros agradecimentos aos meus orientadores, Leonardo Murta e Vanessa Braganholo, pelo incentivo a transcender meus próprios limites e pelo precioso aprendizado. Aos meus colegas da UFF (Universidade Federal Fluminense) pelas dicas e ajudas, em especial, à minha querida amiga Camila, cuja parceria me incentivou a seguir em frente.

Aos meus companheiros da Marinha do Brasil, que compartilharam da minha aspiração por este curso e me encorajaram a prosseguir, o meu mais sincero agradecimento. Reservo especial gratidão aos meus amigos CASNAVIANOS, cuja orientação, feedback e experiências compartilhadas foram fundamentais.

Aos meus companheiros em Cristo, pela sua escuta empática e palavras edificantes, que foram uma fonte de consolo nos momentos de desgaste emocional.

Por último, a todos aqueles que me apoiaram, mesmo através dos mais pequenos gestos, os meus profundos agradecimentos. O vosso encorajamento tem sido a força motriz ao longo do meu percurso.

Those who sow in tears will reap in songs of joy. (Holy Bible, Psalms 126.5)
Aqueles que semeiam com lágrimas, com cantos de alegria colherão.
(Bíblia Sagrada, Salmos 126.5)

Resumo

Sistemas de Gerência de Banco de Dados (SGBDs) são amplamente utilizados para armazenar, recuperar e gerenciar as vastas quantidades de dados que os aplicativos modernos manipulam. Existem vários SGBDs disponíveis na indústria. Embora alguns estudos tenham examinado a coevolução de SGBDs e do código-fonte das aplicações, ainda há uma lacuna de pesquisa no exame da adoção de SGBDs em sistemas reais. Conhecer os SGBDs mais comumente usados, com que frequência eles são usados juntos e seus padrões de substituição pode ajudar os gerentes de projeto a tomar decisões mais assertivas sobre a adoção de SGBDs. Desta forma, conduziu-se uma investigação histórica em 317 projetos populares de código aberto, voltados para usuários finais, desenvolvidos em Java e hospedados no GitHub, com o intuito de determinar se esses projetos empregaram, em algum momento, algum dos 50 SGBDs mais populares de acordo com o ranking da DB-Engines. Foi observado que o MySQL é o SGBD relacional mais utilizado, sucedido pelo PostgreSQL e H2. Considerando apenas SGBDs não relacionais, o Redis surge como a escolha predominante, com Cassandra logo atrás. SGBDs multimodelos são os mais bem classificados em projetos de gerenciamento de infraestrutura. Além disso, foram encontradas diferentes combinações de subconjuntos de 11 SGBDs sendo usados juntos no início do ciclo de vida dos projetos (por exemplo, PostgreSQL e MySQL). Na metade do ciclo de vida dos projetos, foram encontradas combinações de 25 SGBDs sendo usados juntos (por exemplo, MS SQL Server e Oracle). Finalmente, ao final do ciclo de vida, esse número aumenta para 29 SGBDs (por exemplo, Redis e H2). Também foram investigadas as substituições de SGBDs. Ao minerar padrões sequenciais, descobriu-se 20 situações em que projetos substituíram SGBDs. Por exemplo, observou-se 11 substituições do PostgreSQL em 8 projetos do corpus, sendo o MySQL uma escolha de substituição dominante, já que substituiu o PostgreSQL em quatro instâncias. Por outro lado, nenhum projeto mudou do MySQL para o PostgreSQL. Em resumo, este trabalho oferece insights sobre os padrões de adoção, co-uso e tendências de substituição de SGBD.

Palavras-chave: Adoção de Mecanismos de Bancos de Dados, Adoção de SGBD, Ciclo de Vida dos Projetos, Java.

Abstract

Database Management Systems (DBMSs) are largely used to store, retrieve, and manage the vast amounts of data that modern applications handle. There are various DBMSs available in the industry. While a few studies have examined the coevolution of DBMSs and application source code, there is a research gap in examining the adoption of DBMSs in real systems. Knowing the most commonly used DBMSs, how frequently they are used together, and their patterns of replacement can assist project managers in making informed decisions about DBMS adoption. Therefore, we conducted a historical investigation of 317 popular open-source end-user applications developed in Java and hosted on GitHub. We determined if these projects had, at any point, employed any of the top 50 DBMSs as ranked by DB-Engines. We observed that MySQL is the most utilized relational DBMS, succeeded by PostgreSQL and H2. Considering only non-relational DBMSs, Redis emerges as the predominant choice, with Cassandra trailing behind. Multi-model DBMSs are top-ranked in Infrastructure Management projects. Furthermore, we found different combinations of subsets of 11 DBMSs being used together at the beginning of the project life cycle (e.g., PostgreSQL and MySQL). Halfway through the project life cycle, we found combinations of 25 DBMSs being used together (e.g., MS SQL Server and Oracle). Finally, at the end of the life cycle, this number increases to 29 DBMSs (e.g., Redis and H2). We also investigated the replacements of DBMSs. We mined sequential patterns and discovered 20 situations where projects replaced DBMSs. For example, we could observe 11 replacements of PostgreSQL in 8 projects in our corpus, with MySQL being a dominant replacement choice, having superseded PostgreSQL in four instances. Conversely, no project switched from MySQL to PostgreSQL. In summary, this work offers insights into the patterns of DBMS adoption, co-use, and replacement tendencies.

Keywords: Database Engines Adoption, DBMS Adoption, Projects Life Cycle, Java.

List of Figures

List of Tables

List of Abbreviations and Acronyms

AR Architecture Refactoring

DBA Database Administrator

DBMSs Database Management Systems

DML Data Definition Language

DML Data Manipulation Language

DQR Data Quality Refactoring

ERP Enterprise Resource Planning

HPC High-Performance Computing

IDS Integrated Database System

JDBC Java Database Connectivity

JPA Java Persistence API

MLxtend Machine Learning Extensions

MR Method Refactoring

NoSQL Non Structure Query Language

RDBMS Relational Database Management Systems

RDF Resource Description Framework

RIR Referential Integrity Refactoring

SCM Supply Chain Management

SPMF Sequential Pattern Mining Framework

SQL Structure Query Language

SR Structure Refactoring

Trans Transformations

SUMMARY

1 INTRODUCTION

The exponential growth and increasing complexity of data generated by applications have driven the development of a variety of solutions for data storage and management. However, the diverse and heterogeneous nature of existing Database Management Systems (DBMSs) complicates the selection of an appropriate DBMS for an application (GESSERT et al., 2017).

As there is no DBMS that is the best choice for all needs (CATTELL, 2011), there is a multitude of DBMSs available. According to Gessert et al. (2017), the reason for this variety is the impossibility of a system achieving all the desirable properties simultaneously. Thus, distinct DBMSs may be used concomitantly in a single application. For example, relational DBMSs have reached unmatched reliability, stability, and support through decades of development and are widely used by applications. They are often complemented with non-relational DBMSs (i.e., NoSQL) (SAHATQIJA et al., 2018), which have emerged as an alternative for storing information that does not fit well on the relational data model—such as key-value, column-oriented, document-based, and graph-based information—in environments with high demand for scalability (CATTELL, 2011).

1.1 Motivation

Selecting a DBMS for a new project, or re-evaluating a prior choice, can benefit from past project experiences. For example, managers can base their decisions on data showing which DBMSs are commonly used. They might also reconsider their choice if they learn certain DBMSs are often replaced. Finally, they could investigate the adoption of additional DBMSs when they know that some DBMSs are frequently used together.

In the literature, some studies analyze the correlation between changes in the database code and application code. Among these studies, Goeminne et al. (2014) stands out, analyzing how new functionality added to an application can affect the structure of the database and, conversely, how changes in the structure of the database can affect the source code associated with it. Qiu et al. (2013) investigate the coevolution between database schemas and the source code of the applications. These studies focus on the structural level of granularity, considering changes made to schemas or database files and how they affect the source code. Thus, understanding the trends in adopting DBMSs over time is still underexplored in the literature.

Additionally, with the significant evolution of the database management ecosystem, relational DBMSs are no longer the only option. Non-relational DBMSs have emerged, but their adoption patterns in real-world projects remain a subject of interest. Thus, investigating how various DBMSs are adopted, replaced, and used simultaneously is extremely necessary to expand knowledge about data storage and management solutions.

1.2 Goals and Research Questions

In this research, we investigate the use of DBMSs during the life cycle of 317 popular open-source end-user projects developed in Java and hosted on GitHub. We looked for the 50 most used DBMSs ranked on the DB-Engines website (DB-ENGINES, 2022), regardless of the data model they implement—relational or non-relational—and analyzed their adoption, considering a DBMS's inclusion, replacement, or removal. In addition, we checked for a tendency to support different DBMSs concomitantly. Note that the DB-Engines ranking includes some database solutions that are not exactly a DBMS, such as search engines. They are included in the DB-Engines ranking since they somehow index, search, or store data. Some of them are positioned in a high ranking, and since we chose the most popular database engines from that website, we did not discard them from our analysis. Although the DB-Engines call them *database engines*, in our work, we simplify and call them DBMS. Thus, from this point on, we refer to any database engine as DBMS as well.

The popularity of a DBMS in the DB-Engines ranking is measured by several parameters, including the number of mentions on websites, general interest in the DBMS, the frequency of technical discussions about the DBMS, the number of job offers, etc. in which that particular DBMS is mentioned. Additionally, the number of profiles in professional networks mentioning the DBMSs and the DBMSs relevance in social networks are considered. In essence, the criteria to build the DB-Engines ranking is based on the number of DBMS citations from different perspectives. Although we use the DB-Engines classification to guide our search, our work goes much further than analyzing the citations of specific DBMSs, as we delve into the historical projects to investigate the adoption of DBMSs in Java Open Source projects. To guide our efforts, we answer the following research questions:

RQ1: What are the most commonly used DBMSs during the projects' life cycle? In this question, we identify which DBMSs are most commonly adopted during the history of the projects in our *corpus*. We conceived a set of heuristics (see Section 3.3) to identify each of the selected DBMS. Then, we counted their occurrences in the projects and ranked the most used DBMSs.

RQ2: Which DBMSs are often used together? In this question, we identify which DBMSs are frequently adopted together during the projects' history. We used association rules, a data mining technique that detects correlations among frequent item sets. According to Agarwal (2013), this technique generates frequent item sets, from which strong association rules in the form of A -> B are defined. The analysis of associations enables the discovery of correlation rules, presenting statistical correlations between sets A and B. With this technique, we could detect DBMSs that are frequently adopted together.

RQ3: Which DBMSs are frequently replaced by others? In this question, we identify the most frequent replacements of DBMSs during the projects' history. We used sequential patterns mining, a data mining technique that finds patterns of sequential events over time. According to Agarwal (2013), this technique allows mining the set of frequent subsequences in a sequence or in a set of sequences. Using this technique, we identified which DBMSs are commonly replaced by others, either simultaneously or in sequential time intervals.

Our research questions are based on the need to demystify the dynamics of DBMS use in real-world software projects, providing a comprehensive view of how these systems are integrated into the projects life cycles and shed light on trends and practices.

1.3 Contributions

To answer our questions, we sliced each project history, looking for evidence of the selected DBMS. In addition, we analyzed how the application domains and the data models affect the observed results. We found that MySQL is the most used relational DBMS, followed by PostgreSQL and H2. As for the non-relational models, Redis comes ahead, followed by Cassandra. We also found that multi-model DBMSs are the most adopted in the Infrastructure Management domain. Furthermore, we found combinations of the concomitant use among 11 DBMSs early in the projects' life cycle (e.g., PostgreSQL and MySQL). By the middle of the life cycle, the number of DBMSs involved in combinations has more than doubled, reaching 25 DBMSs (e.g., MS SQL Server and Oracle). This number increased to 29 DBMSs (e.g., Redis and H2) at the end of the project life cycle. We also discovered 20 situations that indicate replacements of DBMSs. For example, we found that PostgreSQL was replaced eleven times in eight projects in our corpus. In those projects, PostgreSQL was replaced by four distinct DBMSs. Among them, MySQL was the predominant choice as it replaced PostgreSQL four times in four projects. We also noted the opposite is not true: MySQL was not replaced by PostgreSQL in our corpus. Thus, in addition to discovering the most common DBMSs during the projects' life cycle, we observed that the concurrent use of different DBMSs has grown over time and that substitutions between them occur.

Our analysis aims to bring valuable knowledge through the experiences of real and relevant projects. Identifying the DBMSs most commonly used together and their patterns of coexistence or substitution can enable project managers, developers, and DBAs to make better decisions about the DBMSs they choose. We believe these insights are precious as modern applications evolve in complexity and scale, requiring adaptable and efficient data management strategies. In short, this research represents a significant step towards improving the alignment of DBMS technology choices with the needs of real-world software projects.

1.4 Organization

The remainder of this work is organized as follows:

- Chapter 2 describes a brief history of the evolution of DBMS and data mining, in addition to the main concepts that are considered in this work. It also discusses related work.

- Chapter 3 describes the selected corpus, the proposed heuristics, the research methodology, and discusses the threats to validity.

- Chapter 4 presents the research questions results.
- Chapter 5 concludes our work discusses the main contributions and opportunities for future work.

2 BACKGROUND AND RELATED WORK

This Chapter presents a brief history of the evolution of DBMSs and data mining concepts considered in this thesis in order to prepare the reader for the discussion of the coevolution of DBMSs and applications source code.

2.1 Background

Charles W. Bachman designed the Integrated Database System (IDS) in 1960 (PATNI et al., 2022). IDS [1] was responsible for maintaining a single set of shared files on disk with tools capable of structuring and maintaining them. IDS is considered the precursor of DBMSs and inaugurated a vast field with numerous influential researchers such as Peter Chen (CHEN, 1976; WINSLETT, 2004a), Michael Stonebraker (STONEBRAKER et al., 1976; WINSLETT, 2003b), Philip A. Bernstein (BERNSTEIN et al., 1987; WINSLETT, 2004b), Jim Gray (GRAY, 1981; WINSLETT, 2003a), C. J. Date (DATE, 2006), David DeWitt (DEWITT; GRAY, 1992; WINSLETT, 2002b), and Abraham Silberschatz (SILBERSCHATZ et al., 2019; WINSLETT, 2002a), who contributed to its concepts and development. Each contribution represents a different perspective on what a DBMS is, reflecting various aspects of database management. In fact, there is unanimity in the understanding that DBMS is a crucial software system for efficient data storage, retrieval, integrity, and management. Patni et al. (2022) summarize the definition of a DBMS: "The general-purpose software package that manages the data stored in a database is called a database management system (DBMS)."

Faced with the evolution of storage mechanisms and capabilities, new types of DBMSs have emerged, which brought the need for standardization (PATNI et al., 2022). New ideas arose, including the Relational Model concept proposed by Edgar F. Codd in the 1970s (CODD, 1970), suggesting the data organization into relations (SQL: tables) and each

[1] https://amturing.acm.org/award_winners/bachman_9385610.cfm

relation representing an unordered tuple collection (rows). This model constituted the basis of modern Relational Database Management Systems (RDBMS), and since then, relational DBMSs have remained the most used type of DBMS. They thrived for offering various benefits, including the ACID properties (atomicity, consistency, isolation, and durability), guaranteeing more reliability and data integrity (SAHATQIJA et al., 2018), its efficiency in storing large volumes of data (KLEPPMANN, 2017), and the using of Structure Query Language (SQL) for maintaining and querying such databases.

Despite RDBMS being efficient at managing structured data (PATNI et al., 2022), they are not designed to handle unstructured, schema-less data, which has become increasingly prevalent with the growth of the Internet and social media. In the 2010s, Non-Relational DBMSs (NoSQL) arose to fill in these gaps. These DBMSs store and manage unstructured data without schema in (possibly) multiple nodes that do not require a fixed table structure (PATNI et al., 2022). They have been widely used due to their flexibility, scalability, cost-effectiveness, and ability to manage different data types. However, they do not guarantee the consistency and integrity that relational databases are capable of providing (SAHATQIJA et al., 2018). According to Sadalage and Fowler (2013), the non-relational concept (NoSQL) is applied to an ill-defined set of generally open-source DBMS, mostly developed in the early 21st century, and mainly not using SQL. There are several categories of non-relational DBMS, and the most popular ones are key-value, document, column-family, and graph. In this work, we considered both models, relational and non-relational, regardless of whether that model is implemented or emulated in a given DBMS. In addition, we have inherited the classification of models contained in the DB-Engines ranking (DB-ENGINES, 2022), where a DBMS that supports more than one storage model is considered to be *multimodel*.

The emergence of non-relational database management systems (DBMSs) has not intimidated relational DBMSs. Instead, they have implemented new features to meet market demands. Currently, there are approximately 416 DBMSs of various models available (DB-ENGINES, 2022). This shows that different DBMSs are being developed to solve different problems (SADALAGE; FOWLER, 2013). The initial idea that non-relational DBMSs would replace relational ones has been discarded, giving rise to the concept of Polyglot Persistence, where different DBMSs are used to handle different circumstances. To achieve this, it is crucial to understand the nature of the stored data and how it will be manipulated (SADALAGE; FOWLER, 2013). One technology alone may not be sufficient to achieve the desired results. Consequently, it is becoming more common to combine different DBMS technologies to meet the diverse needs of organizations. This research aims to investigate this concept further (see Section 1.2).

The increase in data generation and availability, as well as advances in database technology, has promoted data exploration to extract knowledge, patterns, and insights with applications in various sectors, from business and finance to science and technology (TAN et al., 2016). Thus, despite having emerged in the 1960s, the concept of Data Mining gained popularity in the 1990s (HAN et al., 2022) and, since then, has been a valuable complement to DBMSs, as it has helped them to discover patterns and relationships in their data. Tan et al. (2016) define it as the automatic discovering process of helpful information in large databases that, in addition to enabling the discovery of new patterns, provides resources for predicting the outcome of a future observation.

Data mining tasks can be categorized into two main types: predictive and descriptive tasks. Predictive tasks involve predicting the value of an attribute based on other attributes. In contrast, descriptive tasks focus on deriving patterns from the relationships between data, such as correlations, trends, clusters, trajectories, and anomalies (TAN et al., 2016). In this particular work, the focus was on mining frequent patterns, associations, and correlations. *Frequent patterns* are sets of items that frequently occur in the data, while *sequential patterns* are subsequences of frequent items(HAN et al., 2022). *Association rules* are used to find rules that strongly associate attributes with different values (WITTEN et al., 2017), and measures like *confidence* and *support* are used to assess the certainty and usefulness of these rules. *Support* is the quantity of correctly predicted results, whereas *confidence* is the quantity of correctly predicted results among all cases in which a given rule is used (WITTEN et al., 2017). Another important measure is *lift*, as it assesses the extent to which the occurrence of one set of items increases the occurrence of another (HAN et al., 2022). For correlation rules, the *lift* measure is used in addition to filtering out misleading associations between sets of items (HAN et al., 2022).

2.2 Coevolution of DBMSs and application source code

To the best of our knowledge, no study in the literature analyzes the evolution of the usage of database systems in a set of projects. Thus, we investigated work that deals with the coevolution of source code changes and DBMS changes (GOEMINNE et al., 2014; QIU et al., 2013; SCHERZINGER; SIDORTSCHUCK, 2020; VASSILIADIS, 2021; DIMOLIKAS et al., 2020). Each of them brings a different contribution regarding both the methods applied and the findings about the influence of source code changes in DBMS changes (mostly schema changes) and vice versa. However, none analyzes how DBMS changes occur throughout the projects history.

Qiu et al. (2013) conducted an empirical study on ten applications using popular DBMSs to understand the coevolution between database schemas and the source code of these applications. For this, they analyzed the entire history of these changes over time, starting with statistics related to the number of database revisions and defining which of them would be considered valid revisions. According to the authors, a valid revision is one that makes changes to a database schema, excluding those related to syntax, comments, formatting, data content, DBMS versions, application-specific implementations, and reversions. By quantifying the number of valid revisions in each project, they found that the ratio of valid revisions is high (about 50%-90% of the total revisions). An exception is the Joomla! project, which, by keeping the DMLs (Data Manipulation Language) and DDLs (Data Definition Language) changes in the same schema file, resulted in database revisions that only changed the data content, a type of change that was not considered a valid revision. In addition, they analyzed the frequency and extent of schema changes. They found schema evolution is common as each project version had around 5 to 25 valid database revisions and around 15 to 180 atomic schema changes. They also identified significant growth in the number of columns and tables in most projects, demonstrating that schemas tend to expand over time.

Another essential aspect they analyzed (QIU et al., 2013) was how database schemas evolve. To do this, they categorized atomic schema changes into six distinct categories. Transformations (Trans), Structure Refactoring (SR), and Data Quality Refactoring (DQR) are the most prevalent categories, collectively representing more than 80% of all schema changes in the ten projects and more than 95% in seven projects. Architecture Refactoring (AR), which occurred in eight projects, and Referential Integrity Refactoring (RIR), which occurred in only three, are both less common. Finally, Method Refactoring (MR) did not occur in any of the ten projects. In view of this, it was observed that additions (as add table, add column) and changes in columns datatypes were the more prevalent atomic changes in schema evolution.

They also investigated (QIU et al., 2013) schema co-changes to estimate its influence on applications code. To do this, they used two main questions: 1) How many valid database revisions contain schema and code co-change information? and 2) How much code-level change is truly caused by schema changes? While the first question intended to understand how often schema and code changes are committed together and whether co-change history information is valid, the second question was designed to estimate to what degree schema changes lead to code-level changes. They manually examined a 10% random sample out of 1464 valid database revisions, as well as their co-changes to answer these questions. The results indicated that more than 70% of the valid revisions contain effective co-changes information, with more than 70% having accuracy greater than 80%. Furthermore, they observed that each atomic schema change generates modifications in 10 to 100 source code lines. Regarding a valid database revision, the number of changed source code lines grows exponentially from about 100 to 1,000, reaffirming the influence of evolving database schemas on the source code of applications. In terms of schema change categories, Transformations (Trans) and Structure Refactoring (SR) had a more significant impact on project code than the other categories.

Goeminne et al. (2014) conducted a study to analyze the coevolution between source code and related activities in a large data-intensive open source system, the OSCAR application repository[2]. First, they investigate coevolution evidence among changes in source code and the database. They examined Oscar's growth over time and found a strong correlation between the number of active pure source code files (without database-related content) and code files containing embedded SQL queries. This suggests a pattern of coevolution as the project evolves, as the number of both types of files increases over time. They then analyzed the monthly proportion of pure source code files versus source code files related to different database technologies (raw SQL, Hibernate, JPA). This allowed them to understand the influence of changes in the use of these technologies on the evolution of source code. Although they observed that these changes occurred over time, such as the introduction of Hibernate and later the JPA's adoption as an alternative for raw SQL queries, the authors concluded they did not significantly impact the evolution of the source code. Finally, they investigated how contributors divide their work among database-related and non-database-related activities and how this division evolves over time. Therefore, they identified contributors' activities based on file types and found that, for the most part, contributors carry out a combination of both activity types, indicating that responsibilities were not clearly separated among the different contributors, except for Hibernate mapping files.

Scherzinger and Sidortschuck (2020) analyzed the coevolution of NoSQL DBMS schemas by investigating entity class declarations in the commit history of ten projects. These projects were selected from a pool of 1,200 open-source Java projects hosted on GitHub, specifically focusing on those with the largest DBMS schemas. The first point they in-

[2] https: //github.com/scoophealth/oscar

vestigated was whether the database schemas (Google Cloud Datastore and MongoDB) of these projects were denormalized, a technique used to improve query performance in NoSQL databases. During the source code analysis, they discovered that each project contained at least one denormalized entity class, characterizing the existence of this technique in all schemas. Although this finding suggests that the developers actively used denormalization, it was unclear whether this was a conscious choice or influenced by the limitations of the NoSQL database. The second point they investigated was the growth in complexity of NoSQL schemas during the life cycle of these projects. For this, the number of entity classes and code lines (Schema-LoC) of these classes were used as metrics to measure the complexity of the schema. They observed that NoSQL schemas tend to grow over time, similar to studies on the evolution of relational schemas. However, there were drops in the growth curves due to occasional refactoring phases. Even so, the growth of these schemas tends to make them increase in complexity. Lastly, they investigated how NoSQL schemas evolve by comparing the frequency in which entity classes undergo schema changes with other classes within the same project, as well as the size of these changes using the lines of code metric (Schema-LoC). Therefore, they observed that some classes undergo more schema changes than others and divided the schema changes by type: additions, removals, and annotation changes. Among these, additions of both entity classes and relevant attributes dominate the evolution of the schema within the projects. Removals, on the other hand, are less frequent and highly project-specific. In addition, they observed that turnover in these schemas is higher than in studies dealing with the evolution of relational schemas, suggesting that NoSQL developers evolve their schemas more frequently.

Vassiliadis (2021) conducted a historical analysis of relational DBMS schemas in 195 open-source projects to understand how these schemas evolve. They analyzed the frequency of changes in the projects' commit history and identified schema families with similar evolution characteristics. With this, they introduced the concept of schema evolution rates, distinct classes of behavior observed in schema evolution over time. This taxonomy allowed them to categorize and understand different patterns of schema evolution: "History-less" projects, "Frozen" projects with no changes, "Almost Frozen" projects with minimal activity, "Focused Shot and Frozen" or "Focused Shot and Low" projects with sporadic changes, "Moderate" projects with a moderate volume of activity and "Active" projects with continuous schema maintenance. They discovered a low frequency of changes and few active commits in most projects. They also found few changes to the number of tables, except for the most recent projects. As for the evolution of the schema, they found that a significant percentage of projects (40%) did not undergo any schema evolution, 10% of the projects presented different versions but did not undergo any logical schema changes, and 20% of the projects were almost frozen. Overall, 70% of the projects showed a total absence or presence of minor change, except for those with active schema maintenance profiles, refuting the belief that schema evolution is extensive.

Dimolikas et al. (2020) conducted a study on the evolution of DMBS schema tables, focusing on the foreign key structures of related tables. They performed historical analysis on six relational schemas to extract information about table births, table deaths, intra-table updates, and their foreign key relationships. The schema versions were represented as graphs, the tables as nodes, and the foreign keys as directed edges, allowing the evolution of the schema to be examined as a diachronic graph covering all versions. In addition, they introduced a concise taxonomy of topological graph patterns to characterize table relationships, classifying them into four categories: isolated (no connections), source (no input connections), lookup (no output connections), and internal (with input and output connections). The study revealed that complex internal tables tend to have high intra-table schema update activity and are usually born in the initial schema version. On the other hand, isolated tables, the simplest in terms of topology, undergo minimal changes and are more likely to be added in later schema versions. The study revealed that complex internal tables tend to have high intra-table schema update activity and are usually born in the initial schema version. The relationship between the topological category of a table and its update activity was another aspect explored in this study. To this end, tables were categorized as rigid (no updates), quiet (few updates), or active (many updates). The results indicated that isolated and source tables are more likely to be scarce or silent, with minimal updates. In contrast, search and internal tables are more likely to be active, undergoing a significant number of updates. These results suggest that the topological complexity of the hierarchy significantly influences the tables' behavior during evolution. Therefore, evolutionary behavior depends on this hierarchy.

Table 1 shows a comparison of the analysis performed by related work. All of them perform a historical investigation to study the coevolution of DBMS changes. Also, they consider only a single data models (either relational or non-relational). Regarding the number of selected projects, only one work (DIMOLIKAS et al., 2020) analyzes a corpus of more than 100 projects. In addition, the mentioned studies look for more granular changes related to DBMS attributes, tables, and schemas, our study differs by identifying changes in adopting different DBMSs. We discover which DBMSs were or still are used during the project's life cycle by inspecting source code and investigating evidence of DBMS

Table 1: Comparative analysis between related work and this study. We abbreviate the Related Work names due to space constraints: (Q)iu et al. (2013), (G)oeminne et al. (2014); (S)cherzinger and Sidortschuck (2020); (D)imolikas et al. (2020) and (V)assiliadis (2021).

Analysis	Q2013	G2014	S2020	D2020	V2021
Coevolution DBMS	✓	✓	✓	✓	✓
Historical analysis	✓	✓	✓	✓	✓
Size of Corpus	10	1	10	6	195
DBMS attributes, tables, schemas	✓	✓	✓	✓	✓
Relational DBMS	✓	✓	X	✓	✓
Non-relational DBMS	X	X	✓	X	X
DBMS adoption	X	X	X	X	X

usage through heuristics based on DBMS connections. Using data mining techniques, such as association rules and sequential patterns, we find the DBMS most commonly used together and the most frequent DBMS substitutions. Thus, our study complements the existing literature and represents a relevant contribution to understanding trends in DBMS adoption.

2.3 Final Remarks

This Chapter has presented an account of the history and concepts of DBMS and data mining. It has also discussed studies on the coevolution of changes between DBMS and the source code of real-world applications. Although those investigations are not exactly on target when compared to the purpose of this work, they show that there is still much to be explored in this area. Chapter 3 describes all the stages in the process of investigating and analyzing DBMS adoption over time.

3 MATERIALS AND METHODS

This Chapter describes the process and the data we used to answer the research questions. We analyzed 317 Java project repositories from GitHub (see Section 3.1) and identified which DBMSs, among the 50 most popular DBMSs, were used during the life cycle of the projects. Then, we used data mining techniques to process the dataset to generate the results.

3.1 Project Corpus

We aimed to create a corpus that balanced representativeness with manageability, as we manually classified each project. Our goal was to select popular open-source end-user applications written in Java. We focused on a single programming language because our method for determining a project's DBMS usage depended on searching for language-specific database-related constructs within the project's source code.

We used the GitHub GraphQL API (v4) to search for all public repositories that were not forks of other repositories, had at least 1,000 stars, were not archived, and received at least one push in the last three months. According to Kalliamvakou et al. (2014), avoiding forks is important to guarantee that the corpus contains only one repository per project. Since we are aware of forks that are much more successful than the original forked projects, we checked our initial sample for forked projects that met our selection criteria and found none. Besides that, restricting the number of stars to at least 1,000 signals that our corpus contains relevant and popular repositories (BORGES; TULIO VALENTE, 2018). Finally, avoiding archived repositories or repositories that did not receive pushes in the last three months ensures a certain degree of activity in all repositories of our corpus. We did not filter out mirror repositories as our analyses do not focus on GitHub-specific features; thus, replicas of external repositories stored in GitHub are welcome. This search was performed on March 27, 2021, and returned 21,149 repositories.

Afterward, we analyzed the metadata of these 21,149 repositories to perform addi-

Table 2: Characteristics of the selected projects for our corpus

	contributors	**stargazers**	**forks**	**commits**
mean	133.10	3,228.07	1,066.56	8,372.85
std	163.80	3,981.99	1,563.43	25,191.38
min	10	1,000	81	1,007
25%	46	1,285	334	1,799
50%	86	1,824	598	3,208
75%	149	3,364	1,114	6,980
max	1,000	33,981	18,871	330,851

tional filters on the number of contributors (10 or more) and the number of commits (1,000 or more) in the default branch. Filtering out repositories with less than ten contributors aims at avoiding personal or coursework projects in our corpus (KALLIAMVAKOU et al., 2014). Moreover, restricting the number of commits in the default branch to 1,000 or more is an attempt to remove immature or short-term projects from our corpus. After applying these filters, our corpus was reduced to 6,708 repositories.

GitHub classifies these 6,708 repositories as using ten different primary programming languages. We used this information to filter projects written in Java, which is the focus of this work. This resulted in a corpus with 633 repositories.

We then manually inspected the 633 repositories. For each project, four researchers examined the GitHub repository and the project web page (when available) to answer four questions: (i) Is this repository documented in English? (ii) Does this repository contain a software project? (iii) Does this repository contain an end-user application? (iv) What is the project domain? The answers to these questions were discussed between two researchers and inspected and revised by the remaining two researchers. The first question aimed at guaranteeing that the researchers could understand the documentation of the projects. The second question aimed to refine the primary programming language's automatic filter. The third question aimed at keeping only applications, removing projects that are not focused on end users, such as frameworks, libraries, programming languages, compilers, and interpreters. The fourth question aimed at identifying the general application domain of the project. The projects in our corpus belong to 21 established domains: Application Container, Automation, Collaboration, Communication, Cryptocurrency, E-commerce, ERP, File Management, Gaming, HPC, Infrastructure Management, Machine Learning (end-user applications in the machine learning domain), Media, Monitoring, Network, Operating System, Personal Management, Program Analysis, SCM, Security, and Software Development. After answering these four initial questions and dismissing those projects that are not documented in English, not software projects, and not end-user applications, our corpus was reduced to 317 repositories.

Table 2 presents descriptive statistics for the characteristics of the projects selected for our corpus. For example, the "commits" column provides an overview of the number of commits of the analyzed projects. The project with the fewest commits has 1,007, while the one with the most has 330,851 commits. On average, projects have 8,372.85 commits, demonstrating significant development activity in the projects. Table 2 also shows statistics for the number of contributors, stargazers, and forks.

3.2 Research Method

To answer our research questions, we first conceived heuristics based on regular expressions for detecting which DBMSs are adopted by each project (see Section 3.3). We also built an infrastructure that automatically clones the projects, runs the heuristics over each project, and populates a database with the obtained outputs.

Since some of our research questions require an analysis of the history of the projects, we split each project into ten equal-sized slices (in terms of commits). We then analyzed each of the ten slices in each project history, looking for changes on added, maintained, and removed DBMSs. In practice, we determined the total number of commits for each project and divided them into segments, with each segment representing 10% of the project's commit history. Therefore, the first slice does not correspond to the initial commit of the project but to the last commit of the first 10% of the project history, and so on for the remaining parts until completing 100%. Note that the last slice does not necessarily correspond to the project's last commit on the collection date. This may occur when the number of commits is not divisible by 10. In this case, we ignore some of the last commits to split the project history into slices of the same size. Also, the last slice may not reflect the end of the project since the life cycle of the projects continued after we collected our data in March 2021.

Figure1 illustrates the Skywalking repository, with 7,127 commits sliced into 10 parts, where each slice corresponds to 10% of the total number of commits in this repository. To do so, we first ignore the last seven commits. Then, we distributed the reminder 7,120 commits in each of the 10 slices, so each slice has 712 commits. Each slice is represented by its last commit, which is a snapshot of the repository at 10%, 20%, ..., 100% of the project history.

After defining the method for slicing the history of the projects, we selected five projects to validate the heuristics we used to automatically detect the DBMSs used by

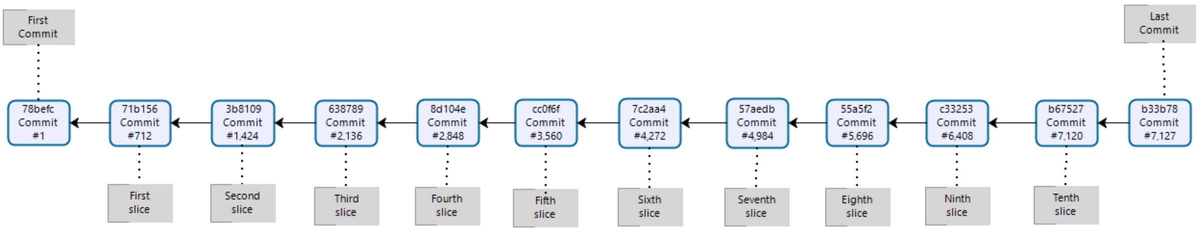

Figure 1: Skywalking repository commits' history slicing

each project. Thus, in each of the ten slices defined for each project, we used the heuristics to automatically detect the DBMSs and stored the results in the database we created for our analysis. Once we confirmed that the detection occurred properly in our five projects sample (see Section 3.3), we ran the heuristics in all ten slices of all the projects in our corpus.

Using these results, we then generated a dataset suitable for the data mining tools used to answer the research questions through association rules and sequential patterns. We first used the MLxtend library from Raschka (2018) to generate the association rules and the SPMF library from Fournier-Viger et al. (2016) to generate the sequential patterns. MLxtend has about 4.3k stars on GitHub, demonstrating popularity in the community. As for the SPMF repository, since it is not public, we cannot consider it in terms of stars, but we found around 1,000 research papers that cited or adopted the SPMF library [1]. Also, both libraries' authors are active researchers in the scientific community, having many papers published in the data mining and machine learning fields.

Aiming to validate the observed patterns and run complementary analyses, we used a Pattern Counter tool [2]. Besides counting patterns from sequence lists, the tool generates support, confidence, and lift measures. We used this tool extensively to filter the results obtained with association rules and sequential patterns and to validate them (see Section 3.4).

Figure 2 illustrates our research methodology. It consists of examining the adoption of 50 DBMS, selected from the DB-Engines ranking, over the life cycle of 317 open-source Java projects. The complete experimental package, containing the data and scripts used in this research as well as the reproducibility instructions, is available at the DB-Mining repository [3].

[1] https://www.philippe-fournier-viger.com/spmf/index.php?link=citations.php
[2] https://patterncounter.readthedocs.io/en/latest/
[3] https://gems-uff.github.io/db-mining

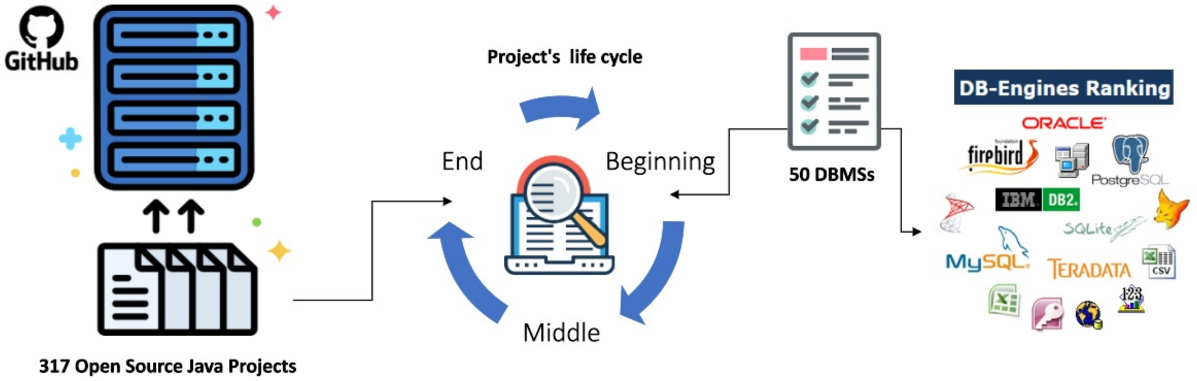

Figure 2: Research Methodology Summary

3.3 Database Heuristics

We selected the top 50 DBMSs listed in the February 2022 DB-Engines ranking (DB-ENGINES, 2022), which ranks DBMSs according to their popularity. To ensure a balanced analysis, we searched for the most popular DBMSs among relational and non-relational models. We selected the top 26 relational DBMSs and the top 27 non-relational DBMSs. Although this totals 53, we have exactly 50 DBMS since three of them (Ignite, Virtuoso, and MarkLogic) appear in both categories (relational and non-relational). Although we found some DBMSs that were a fork of another, such as MariaDB, which is a fork of MySQL, we considered them separate DBMSs since they are considered as such by the ranking.

We analyzed the official documentation of the top 50 DBMSs looking for information about how they are accessed by Java programs to conceive our heuristics. We considered four aspects: (i) What imports are required to use these DBMSs? (ii) Which drivers are needed? (iii) How is the connection established? (iv) Which libraries are needed?

The first aspect aims to identify the required **imports** for using these DBMSs. Each DBMS has specific classes that must be imported to enable their usage in Java projects. This aspect allowed us to identify most non-relational DBMSs. For instance, Cassandra requires the statement "`import.com.datastax`," so we use this as a search string in our heuristic for Cassandra. However, relational DBMSs utilize Java Database Connectivity (JDBC), which means a generic import is used for most DBMSs of this type. For instance, Oracle, MySQL, and MS SQL Server are relational DBMSs that utilize JDBC. Therefore, we disregarded the import strings heuristic for relational DBMSs that used generic imports to avoid false positives. The second aspect aims to identify how the project loads the **driver** to access each DBMS. For example, for the Oracle

Table 3: Import's heuristics that indicate DBMS adoption.

DataBases	Import
Oracle	-
MySQL	-
MS SQL Server	-
PostgreSQL	-
MongoDB	import\s{1,}com\.mongodb
IBM DB2	-
Redis	redis\.clients\.jedis\.Jedis com\.lambdaworks\.redis
ElasticSearch	-
SQLite	-
MS Access	-
Maria DB	-
Cassandra	import\s{1,}com\.datastax
Ehcache	import\s{1,}org\.ehcache

Table 4: Driver's heuristics that indicate DBMS adoption.

<table>
<tr><th>DataBases</th><th>Driver</th></tr>
<tr><td><b>Oracle</b></td><td>oracle\.jdbc\.driver\.OracleDriver
oracle\.jdbc\.OracleDriver</td></tr>
<tr><td><b>MySQL</b></td><td>com\.mysql\.cj\.jdbc\.Driver
com\.mysql\.jdbc\.Driver</td></tr>
<tr><td><b>MS SQL Server</b></td><td>com\.microsoft\.sqlserver\.jdbc\.SQLServerDriver</td></tr>
<tr><td><b>PostgreSQL</b></td><td>org\.postgresql\.Driver</td></tr>
<tr><td><b>MongoDB</b></td><td>-</td></tr>
<tr><td><b>IBM DB2</b></td><td>com\.ibm\.db2\.jcc\.DB2Driver</td></tr>
<tr><td><b>Redis</b></td><td>[^a-zA-Z|^\/"_#|0-9]RedisClient[^a-zA-Z|^\/"_#|0-9]
[^a-zA-Z|^\/"_#|0-9]Jedis[^a-zA-Z|^\/"_#|0-9]</td></tr>
<tr><td><b>ElasticSearch</b></td><td>org\.elasticsearch\.client</td></tr>
<tr><td><b>SQLite</b></td><td>SQLite\.JDBCDriver</td></tr>
<tr><td><b>MS Access</b></td><td>sun\.jdbc\.odbc\.JdbcOdbcDriver</td></tr>
<tr><td><b>Maria DB</b></td><td>org\.mariadb\.jdbc\.Driver</td></tr>
<tr><td><b>Cassandra</b></td><td>[^a-zA-Z|^\/"_#|0-9]CassandraConnector[â-zA-Z|^\/"_#|0-9]</td></tr>
<tr><td><b>Ehcache</b></td><td>-</td></tr>
</table>

DBMS, we use the following search strings "`oracle.jdbc.driver.OracleDriver`" and "`oracle.jdbc.OracleDriver`." The third aspect aims to identify how the **connection string** that points to the DBMS the project wants to connect to is created. Each DBMS has its connection string, but in Java, relational DBMSs have certain features in common. For example, for the Oracle DBMS, we use the following search string "`jdbc:oracle`". The fourth aspect aims to identify a DBMS usage by looking after the **libraries** declared in the Maven descriptor. For example, we use the following search strings for the Oracle DBMS in our heuristic to find Oracle database usage: "`ojdbc`" and "`com.oracle`."

We also discovered that some heuristics were common to multi-model DBMSs; for example, Virtuoso relational and Virtuoso non-relational DBMSs use the same **libraries** heuristics. Since we found 3 multi-model DBMSs—Ignite, Virtuoso, and Marklogic—we separated the heuristics common to the two models. We combined them with the model-

Table 5: Connection's heuristics that indicate DBMS adoption.

DataBases	Connection
Oracle	jdbc:oracle
MySQL	jdbc:mysql
MS SQL Server	jdbc:sqlserver
PostgreSQL	jdbc:postgresql
MongoDB	[ˆa-zA-Z|ˆ\/"_#|0-9]MongoClient[ˆa-zA-Z|ˆ\/"_#|0-9]
IBM DB2	jdbc:db2
Redis	redis:
ElasticSearch	jdbc:es jdbc:elasticsearch
SQLite	jdbc:sqlite
MS Access	jdbc:ucanaccess
Maria DB	jdbc:mariadb
Cassandra	-
Ehcache	[ˆa-zA-Z|ˆ\/"_#|0-9]CacheManager[ˆa-zA-Z|ˆ\/"_#|0-9]

Table 6: Libraries heuristics that indicate DBMS adoption.

DataBases	Libraries
Oracle	ojdbc com\.oracle
MySQL	mysql-connector-java
MS SQL Server	mssql-jdbc
PostgreSQL	\s*artifactId\s*\s*postgresql
MongoDB	org\.mongodb mongodb-java-driver mongodb-driver
IBM DB2	com\.ibm\.db2
Redis	<\s*artifactId\s*>\s*lettuce <\s*artifactId\s*>\s*jedis commons\-pool2
ElasticSearch	\s*artifactId\s*\s*elasticsearch
SQLite	sqlitejdbc
MS Access	\s*artifactId\s*\s*ucanaccess
Maria DB	mariadb-java-client
Cassandra	<\s*artifactId\s*>\s*cassandra com\.datastax\.oss
Ehcache	<\s*artifactId\s*>\s*ehcache <\s*artifactId\s*>\s*terracotta-toolkit

Table 7: Comparison between the results found by our Heuristics (H) and the project's documentation (D or d). Cells in red indicate false positives. We abbreviate the DBMS names due to space constraints: (O)racle; (MyS)QL; (S)QL Server; (P)ostgreSQL; (M)ongo(DB); IBM (DB2); (R)edis; (E)lasticsearch; SQ(Lite); (A)ccess; (Ma)ria DB; and (C)assandra.

Projects	**O**	**MyS**	**S**	**P**	**MDB**	**DB2**	**R**	**E**	**Lite**	**A**	**Ma**	**C**
Activiti	d	d	d	d		d			d	d	d	
Che	H	H		**HD**	H		H	H				
Skywalking	**Hd**	**HD**	**Hd**	**Hd**	**Hd**	d	**Hd**	**HD**	d	d	**Hd**	**Hd**
Storm	d	**Hd**	d	d	**HD**	d	**HD**	**HD**	d	d	d	**HD**
Pinpoint	**HD**	**HD**	**HD**	**HD**	**HD**		**HD**	**HD**			**HD**	**HD**

specific heuristics to define which of the models was used by each project that adopts such DBMSs. Thus, each multi-model DBMS has two sets of heuristics, one to identify its use as a relational DBMS and another for its use as a non-relational DBMS. In this way, to identify the 50 selected DBMSs in our project corpus, we defined 53 sets of heuristics, balancing our search between the two models: 26 sets of heuristics for relational DBMSs and 27 for non-relational.

We consider a given project adopts a certain DBMS when at least one of the respective heuristics is found on that project source code. Going back to the Oracle example, if we find the string referring to the driver, "`oracle.jdbc.driver.OracleDriver`" or "`oracle.jdbc. OracleDriver`," or the connection string, "`jdbc:oracle`," or the strings referring to the libraries, "`ojdbc`" or "`com.oracle`," we assume that Oracle is used in that project. The selected DBMS list with the respective heuristics is available at the replication package.

We manually validated the heuristics for the top 12 DBMSs (see Tables 3, 4, 5, and 6) on a sample of five randomly selected projects of our corpus. To accomplish this, we analyzed the documentation of each project to determine which DBMSs they support. We, then, ran our heuristics and compared the DBMSs we identified with the ones used by the projects according to their documentation.

Table 7 compares the results obtained by our heuristics and those found manually in the projects' documentation. Each cell (r, c) in this table contains an **H** when the DBMS on column c was found by our heuristics for the project at row r, and a **D** when the DBMS on column c is specifically mentioned in the project's documentation. Also, a cell contains a **d** when the DBMS of column c may or may not be used for the project on row r. This happens when the project documentation mentions the possibility of utilizing other DBMSs not explicitly mentioned (e.g. when they mention the possibility of using a JDBC connection mechanism commonly used by relational DBMSs).

Cells marked with **HD** or **Hd** means the DBMS was found by our heuristics and was mentioned, specifically or as a possibility, in the project documentation, characterizing the true positives. Cells marked with a single **d** are not considered false negatives since **d** are possibilities instead of obligations. Lastly, cells with a red **H** represent false positives since our heuristics detected the corresponding DBMS, but it was not mentioned in the project's documentation.

For example, the Activiti Project documentation mentions Oracle, MySQL, MS SQL Server, PostgreSQL, IBM DB2, and H2 as examples of supported DBMSs, with H2 being the default option. It also states the support for Java Database Connectivity (JDBC), indicating that relational DBMSs are potential options for this project. Therefore, the fact that our heuristics did not find signs of usage of one of the eight relational DBMSs we used in our validation step was not considered a false negative. In the case of the Skywalking project, its documentation cites MySQL, H2, and Elasticsearch as existing implementations, and it mentions the user may "implement [their] own." Thus, we considered that this project allows the implementation of other DBMSs, and thus the heuristic results as correct — it found DBMSs that are not explicitly mentioned by the project's documentation, but we understand they are valid possibilities. Storm's documentation mentions the integration with JDBC, Cassandra, Redis, Elasticsearch, and MongoDB. Therefore, we considered relational DBMSs as possible options, and the results obtained by our heuristics were considered correct. All the DBMSs mentioned in the documentation for the Pinpoint project were found by our heuristics. Finally, for the Che project, we found evidence of Oracle, MySQL, PostgreSQL, MongoDB, Redis, and Elasticsearch. Given that PostgreSQL is the only DBMS mentioned in the project's documentation, the results indicating evidence of the other DBMSs diverged, so we considered them false positives.

As shown in Table 7, we found evidence of the adoption of 29 DBMSs in the 5 projects, with only 5 false positives. This corresponds to a precision of 82.75% with 100% recall. Therefore, we consider our heuristics adequate and sought mechanisms to mitigate possible failures. For instance, we found situations where generic strings, such as *DatabaseClient*, *CosmoClient*, and *MongoClient*, were present in the results. Although these examples could indicate the connection methods of different DBMSs, the application developer might have created a method or variable containing these strings as substrings. To mitigate this issue and focus specifically on identifying the connection establishment, we applied filters at the beginning and end of the search string.

For the projects that mention JDBC connection support, we face a challenge since this type of connection can be used generically without explicitly specifying the DBMSs. However, our heuristics are designed to be specific for each DBMS, allowing us to identify clues even when the documentation mentions a generic JDBC connection. Therefore, when we found a clue associated with a DBMS that was not specified in the documentation but is considered a possibility due to its usage of the JDBC mechanism, we considered it a successful identification. This means that despite the project mentioning a generic connection that could potentially obscure the specific DBMS being used, our DBMS-specific heuristics uncovered the clue associated with the DBMS.

3.4 Infrastructure

Most steps of our research were automatized to reduce error-prone and time-consuming manual executions and to increase the reproducibility and auditability of the results. We first implemented a series of scripts and Jupyter notebooks for collecting, filtering, and analyzing projects' metadata from GitHub, as described in Section 3.1. Then, we implemented a script to automatically download the repositories of our corpus. It took about 12 hours to clone the 317 repositories on a machine with an i7 4790k processor (4GHz) with 4 cores and 16 GB of RAM – the run times we mention later in this section were all measured using this same machine.

The execution of each heuristic over the projects in our corpus was also automatized. This script first checks the existing heuristics and decides whether a new execution is necessary. This step populates a relational DBMS with information about the projects, the heuristics, and the execution of each heuristic for each project. Finally, we implemented a web application that allows to manually validate the results. This application shows the pending matches and, for each match, the output generated by `git grep`. The researchers could analyze the results and confirm it or not. In this step, running all heuristics in the 10 slices of the 317 projects took about 30 hours.

To mine association rules, we adopted the Apriori algorithm, a popular algorithm for extracting frequent item sets, proposed by Agrawal, Srikant, et al. (1994) in 1994 and implemented by the MLxtend library. According to Agarwal (2013), frequent item sets' mining has allowed the discovery of interesting associations and correlations between large amounts of business transaction records assisting in the decision-making processes of many businesses. Thus, we understood that the employment of the Apriori algorithm would be appropriate. We then use it to find the correlation between the DBMSs, i.e., which DBMSs are used together in a given project. This algorithm requires a one-hot data frame as input. Consequently, we created three datasets containing the results of the heuristics discovered at three stages of the projects' history (beginning, middle and end) to capture the evolving DBMS adoption in the projects over time. To achieve this, we developed Python scripts for building and pre-processing datasets, using scripts from the Pandas library for the pre-processing. Afterward, we applied this algorithm to each dataset to generate the correlations between DBMSs, with a run time of a few seconds.

To mine sequential patterns, we adopted the Prefixspan algorithm, the most popular pattern-growth algorithm for sequential pattern mining (HAN et al., 2001). Specifically, we used the implementation provided by the SPMF library. Given our objective of discovering subsequences in sequential datasets (FOURNIER-VIGER et al., 2017), we considered this algorithm as the most suitable choice. We aimed to identify the DBMS that was most frequently replaced over time, treating each project as a sequence composed of 10 items. Each item represents a slice of the project history (as explained in Section 3.2), indicating the DBMSs that were added, kept, or removed from the project. Consequently, by combining the sequence records from all projects, we created a sequential dataset that served as the input for the PrefixSpan algorithm. This algorithm took between 10 and 30 minutes to generate the sequential patterns, depending on the support (e.g., 2) and the size of the sequence (e.g., 4). We also established three flags, namely Init, In, and Out, to indicate the addition or removal of a DBMS. The rules we used to generate the input file are defined as follows:

- The existence of an **Init** flag indicates that the DBMS occurs in that project since its first slice. For example, to denote that Oracle appears in the first slice of a given project, we add $Oracle_{Init}$.
- The existence of an **In** flag indicates the first occurrence of the DBMS in any slice other than the first. For example, if the first appearance of SQLite in a given project occurs in its third slide, we add $SQLite_{In}$.
- The existence of the DBMS in the previous slice and in the current slice indicates it was kept from one slice to the other. We denote this by using the name of the DBMS. For example, if MariaDB was present in the third slice of a given project, and it also appears in the fourth slide, we add $MariaBD$.
- The existence of an **Out** flag indicates the DBMS was removed from the project in

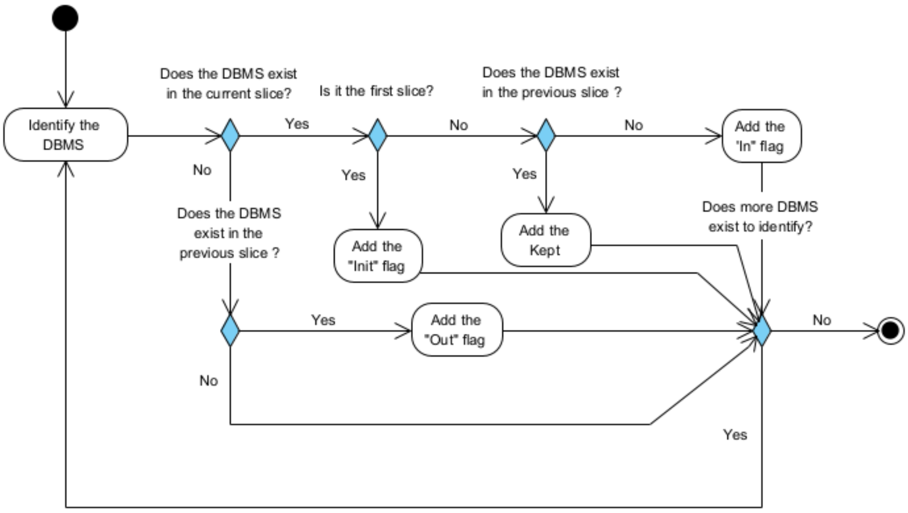
Identify the DBMS
Does the DBMS exist in the current slice?
Yes
Is it the first slice?
No
Does the DBMS exist in the previous slice ?
No
Add the "In" flag
No
Yes
Yes
Does the DBMS exist in the previous slice ?
Add the "Init" flag
Add the Kept
Does more DBMS exist to identify?
Yes
Add the "Out" flag
No
No
Yes

Figure 3: Activity Diagram for identifying the DBMSs in a given project slice

that slice. For example, if Redis was present in the fifth slice of a given project but was not found in the sixth slice, we add $Redis_{Out}$.

Figure 3 shows an activity diagram illustrating how we identify the DBMSs in a given project slice, considering the DBMSs added, kept, and removed, as explained above. We emphasize that this activity diagram describes only part of the process implemented by our script that generates the data entry file required by SPMF. Using Zendesk's Maxwell project as an example, we came up with the following notation to represent how the established flags contributed to identifying the DBMS:

$$\mathbf{MySQL}_{\boldsymbol{Init}} \rightarrow \mathbf{MySQL} \rightarrow \mathbf{MySQL} \rightarrow \mathbf{MySQL} \rightarrow \mathbf{MySQL} \rightarrow \mathbf{MySQL} \rightarrow \mathbf{Redis}_{\boldsymbol{In}}\ \mathbf{MySQL} \\ \rightarrow \mathbf{Redis\ MySQL} \rightarrow \mathbf{Redis\ MySQL} \rightarrow \mathbf{Redis}_{\boldsymbol{Out}}\ \mathbf{MySQL}$$

In this notation, slices are separated by an arrow (→). In the above example, MySQL was present in the first slice (MySQL_{Init}) and was kept as the only DBMS until the sixth slice (MySQL). In the seventh slice, Redis appears for the first time (Redis_{In}), and MySQL is kept (MySQL). In the eighth and ninth slices, Redis and MySQL are still present (Redis MySQL). Finally, in the last slice, Redis was removed (Redis_{Out}) while MySQL was kept (MySQL).

For our analysis, we also needed to count the occurrences of a given DBMS in a sequence, allowing for both the validation of the mined results and the execution of additional analyses. For that, we have used a tool called Pattern Counter [4]. Pattern Counter allows counting patterns in a sequence of items using rules and variables. The

[4] https://patterncounter.readthedocs.io/en/latest/

tool uses the same input file we generated to work with the SPMF library. In addition, Pattern Counter provides filtering capabilities, allowing data extraction through various combinations of parameters.

3.5 Replacement Patterns

When examining two DBMSs X and Y, we can identify a potential replacement pattern. This replacement occurs when DBMS X exists in a specific slice, then in a subsequent slice, X does not exist, but instead, DBMS Y appears, and Y is kept in a subsequent slice. We can formalize this replacement using the following rule:

$$\boldsymbol{X \rightarrow Y_{In}X_{Out} \rightarrow Y}$$

where X and Y represent the DBMS, the suffix $_{In}$ represents the DBMS was added, the $_{Out}$ suffix represents the DBMS was removed, the absence of $_{In}$ and $_{Out}$ represents the permanence of the DBMS, and → separates the slices.

For instance, consider the sequential pattern $PostgreSQL \rightarrow Oracle_{In}PostgreSQL_{Out}$ $\rightarrow Oracle$. This notation signifies that *PostgreSQL* was present in a particular slice, and in a subsequent slice, *PostgreSQL* was replaced by *Oracle*, which was kept in a following slice.

Another way to perceive a DBMS replacement is when DBMS X exists in a specific slice, then in a subsequent slice, DBMS Y enters, and in a later slice, DBMS X is removed, while DBMS Y is kept. The following equation formalizes this situation:

$$\boldsymbol{X \rightarrow Y_{In} \rightarrow X_{Out}Y}$$

For example, consider the sequence $Oracle \rightarrow PostgreSQL_{In} \rightarrow Oracle_{Out}PostgreSQL$. This indicates that *Oracle* was present in a particular slice, *PostgreSQL* entered in a subsequent slice, and in a following slice, *Oracle* was removed while *PostgreSQL* remained. This formulation allows us to capture and analyze the replacement patterns among DBMSs over time. By identifying such replacements, we gain insights into the dynamics of DBMSs' usage and their transitions within the dataset.

3.6 Threats to Validity

As with any empirical work, our study has limitations due to design decisions and the nature of this research. In this Section, we discuss the threats to the validity of our study.

Our corpus may not be representative of all Java open source projects that are popular, mature, and active. We tried to avoid toy projects and inactive projects by filtering out projects with less than 1,000 stars, no pushes in the last 3 months (relative to the time our search was done on GitHub), more than 10 contributors, and more than 1,000 commits in the default branch. Also, the list of projects returned by our search query was manually inspected to filter out projects that did not contain end-user applications. Although this analysis was conducted by two researchers and revised by another two, there may exist misclassified projects in our corpus.

Still, we collected metadata from the repositories from GitHub in March 2021 and downloaded the repositories to collect data for our analysis in March 2022. Our analysis represents the project's state during the data collection. After that time, projects may have become inactive, may have been removed, or may have received several commits, which may have included additional use of DBMSs that were not captured in our analysis.

We used DB Ranking to select the top 50 DBMSs of the ranking (as of February 2022), so our analysis does not cover all existing DBMSs (the DB-Engines ranking (DB-ENGINES, 2022) includes 422 DBMSs as of September 2023). Also, popular DBMSs for Java projects may have been left out of our analysis since we did not use Java popularity as a criterion to choose the DBMSs we would search for in our corpus. This is due to two reasons. First, we could not find such information. Second, using a pre-compiled list of DBMSs would possibly hide new insights, such as finding usage of a DBMS that is possibly out of the list (if such a list existed). We believe the results of our analysis can be used to build such a list.

The heuristics proposed may not encompass all possible ways to identify DBMS connections and may result in false positives. To assess the efficacy of the heuristics, we conducted a validation step for a subset of our heuristics using 5 projects. We manually inspected the project's website to find false positives and false negatives (see Table 7). We made several adjustments in how we built our heuristics based on those results and reflected them in the remaining heuristics to guarantee a high level of precision and recall for the analyzed DBMSs.

To find replacement patterns in the project life cycle, we sliced each project's history into 10 pieces. For projects with a long development history, this may mean we analyzed snapshots that are very large in size. For instance, project liferay/liferay-portal (see Table 8) has 866,693 commits, resulting in slices that are 86,669 commits apart from each other. We acknowledge that from one snapshot to the other, DBMSs may have been added, replaced, or removed more than once. As a side effect, we may not have a complete record of changes in our analysis. Even projects with shorter development histories (for instance, square/keywhiz – see Table 9 – which has only 1,748 commits) may suffer from this problem. We preferred to have a controlled and standard way to understand the history of changes in more projects to have an overall idea of the landscape of DBMS adoption instead of conducting a comprehensive and more granular analysis at this point. Future studies may go in-depth to analyze the changes in a more detailed fashion.

For the replacement pattern of the form $\boldsymbol{X} \rightarrow \boldsymbol{Y_{In}} \rightarrow \boldsymbol{X_{Out}Y}$, we did not require the DBMS maintenance ($\boldsymbol{X}$) to be explicit in the same slice where a DBMS is added ($\boldsymbol{Y_{In}}$). Guaranteeing this condition would require us to use sequences bigger than 4, which would imply a much larger execution time, and a huge set of mined rules to analyze. However, it is safe to assume that the DBMS is there (or was in the past) since we've indicated through the flags that a DBMS is only removed if it was present in a previous slice.

In our co-occurrence analysis, the fact that we found evidence of the use of more than one DBMS at the same time by a given project may not mean they are in fact used together. It may simply mean the project was designed in a way that it could work with several DBMSs—but the user had to choose a single one at installation time. To make sure more than one DBMS is used, we would need a deeper analysis of the source code, for example, to investigate which entities are stored in each DBMS.

We segmented each project's history into ten distinct parts. It is worth noticing that these projects possess varying ages, and as a result, each slice may not represent equivalent temporal durations across all projects (see Tables 8, 9). This discrepancy could introduce potential biases since trends change over time. Future analyses might delve deeper into the temporal dimensions of the evolution of the DBMSs to more accurately capture the progression and nuances over time.

3.7 Final Remarks

This Chapter describes the whole process of corpus selection, definition and validation of heuristics, mining repositories in search of heuristics, and characterization of substitution

Table 8: Commits inspected from the Liferay-Portal project

slice	sha1	date	diff commits (days)
1	cb9feb9c1475f14438fe638a1189423c609ec162	2014/02/18	1607
2	dadfb0e1f690e441554f85271b0305cb9def369e	2015/11/21	641
3	c0d9a6f2faed7cb054d6c65df5501857171c7790	2016/07/20	242
4	d231eb7bec08a3ee1c000f29162f8901b3060ca0	2017/09/04	411
5	561d271a1b9763c1f49e9703866e4217173bc83c	2018/07/17	316
6	db755cd7d363afcd389f574e5f37a70a684a712a	2019/03/29	255
7	1c587628b2576b4f1d87f3f7cbe1b87e3ba896b8	2019/09/04	159
8	9ad97a63ff06dbb4d6ca64611b10503bf169f408	2020/05/14	253
9	143250a4a43b5107ad206d11674aa3138bad2ab5	2021/05/21	372
10	84e6a40fe8bc249e7b574903bbb075e637ad7dad	2022/02/16	271

Table 9: Commits inspected from the Keywhiz project

slice	sha1	date	diff commits (days)
1	63c2b938937d76f85e632402454f7ab25ec39a2c	2015/04/24	29
2	45dc013ff3a34c777af58ded7cc3fb03dd3ad8b6	2015/11/10	200
3	22a50603301547fdb31576d89ee31b1111584069	2016/06/28	231
4	e3dfc26a1293a80db2863a28bb2cae4b4454f742	2017/03/08	253
5	fdd350ff38d79e78d6e1d81109d879f8a270deb3	2019/04/19	772
6	737c56e698d3d06c1382de8aecb73236a02a68e0	2019/09/06	140
7	7205b311339dec44a2aacf53a797bc1403f67dc9	2020/03/16	192
8	132140b2c7ad0615e38d6cf9a3550990160245b9	2020/07/13	119
9	ad55029e312a20ed980b887a5366206b92e9835c	2021/04/29	290
10	0f2ae34814874a88d5250d849489e9e49480997a	2022/01/31	277

patterns, in addition to discussing the threats to the validity of our study. Chapter 4 answers the research questions.

4 RESULTS AND DISCUSSION

This Chapter reports and discusses the research questions results about the most predominant DBMS choices throughout the project life cycle, highlighting the growing adoption of more than one DBMS in a single project and the occurrences of DBMS replacements.

4.1 What are the most commonly used DBMS during the life cycle of projects? (RQ1)

In this question, we identify the most adopted DBMSs throughout the projects' history, considering the established heuristics. It is important to highlight that we consider that a project adopted a DBMS if it appeared in any segment of the project's history, even if it was no longer present in the final segment.

Figure 4 presents the most popular DBMSs in the projects of our corpus, considering the historical analysis performed across all 10 slices of each project. As described in Section 3.3, we applied 53 sets of heuristics for the 50 surveyed DBMSs since some are multi-model. We found evidence of the adoption of 46 DBMSs in 197 of the 317 projects in our corpus. The top positions are occupied by MySQL, which appears in 107 projects, followed by a tie between H2 and PostgreSQL, appearing in 74 projects, then Oracle in 66 projects, Redis in 61 projects, and Ehcache in 57 projects. Thus, MySQL was present in about 54% of projects that use a DBMS, H2, and PostgreSQL occurred in about 37%, Oracle in about 33%, Redis in about 31%, and Ehcache in about 29% of these projects. As can be noted, the relational (SQL) model is the most commonly used (78.17% of the projects use a relational DBMS). Still, the non-relational (NoSQL) model is also present. We also found evidence of Ignite being adopted by 2.17% of the projects in both categories: one project uses it as a relational database, and three projects use it as a non-relational database.

We also observed that MySQL, PostgreSQL, and Oracle are among the most used

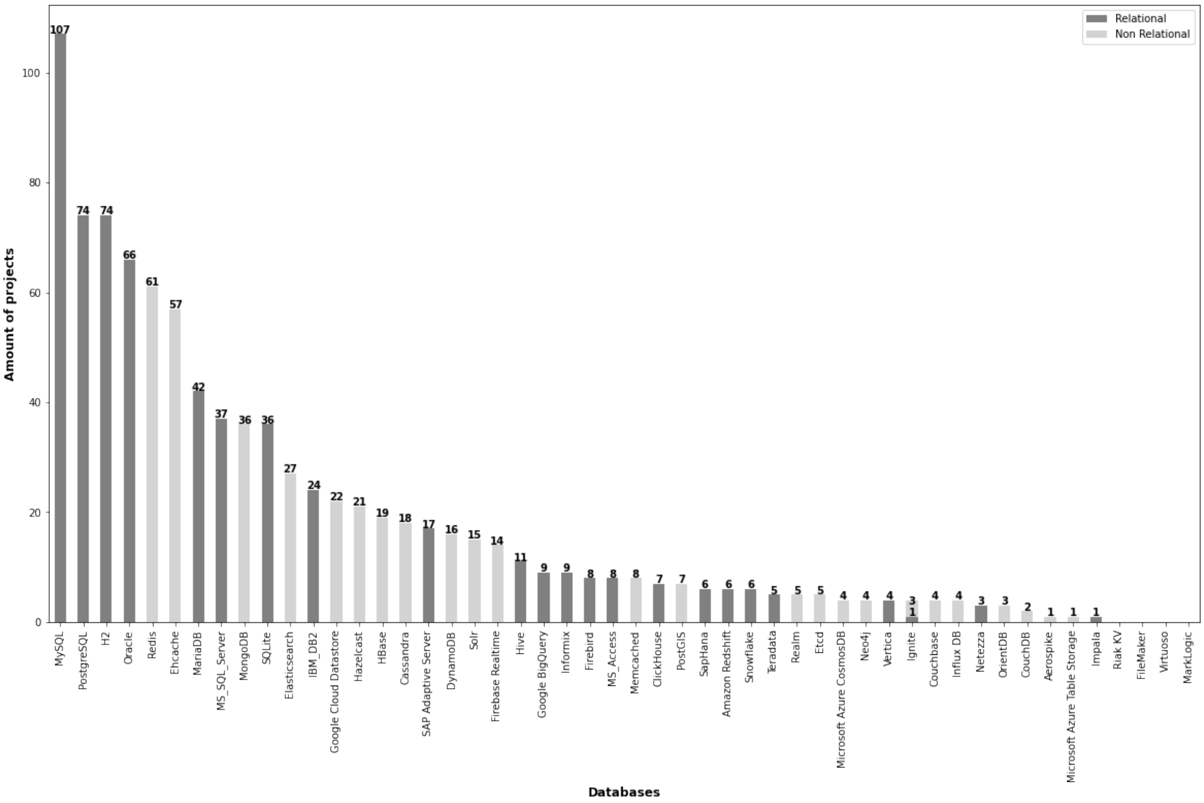

Figure 4: Most Popular DBMSs in Java projects

DBMSs in the DB-Engines ranking (DB-ENGINES, 2022), even considering the different contexts between this ranking and our research. On the other hand, H2 (Hypersonic 2) is our second most adopted DBMS and occupies the 47th position in this ranking. It might have happened due to the simplicity of integration of H2 in Java projects [1].

Given the 46 DBMSs found, we observed that the adopted DBMSs are distributed as follows: 22 relational DBMSs (47.82%), 23 non-relational DBMSs (50.00%), and one multi-model DBMS (Ignite – 2.17%). This information reinforces the increasing trend of non-relational model adoption among Java projects. According to Davoudian et. al (DAVOUDIAN et al., 2018), non-relational DBMSs are not designed to replace relational DBMSs but as a solution to the gaps regarding the need for scalability and availability that certain distributed applications require. Another fact we identified in our research that corroborates the authors' statement is presented in the Venn Diagram shown in Figure 5, which shows the number of projects classified by the type of data model they adopt.

Out of the 197 projects in which we found evidence of the use of DBMSs, 56 adopt only the relational model, and 43 adopt only the non-relational model. However, we found an

[1] (http://www.h2database.com/html/history.html)

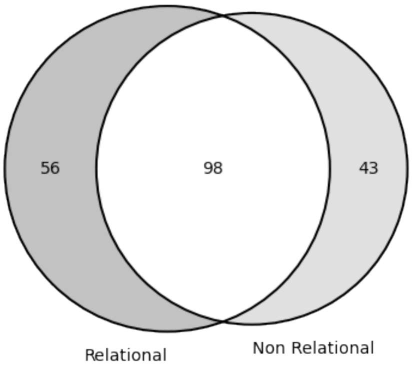

Figure 5: Venn Diagram to represent the adoption of data models in projects

intersection where both models are adopted (98 projects). Thus, 49.75% of these projects adopt both data models, while 50.25% use only one of the models. The understanding that both models can complement each other is reflected in about half of the projects that use a DBMS in our corpus.

Out of the 50 DBMS we searched for in our corpus, we did not find evidence of usage of 4 of them: FileMaker, RiakKV, Marklogic, and Virtuoso. Marklogic and Virtuoso are multi-model, while FileMaker is a relational DBMS, and RiakKV is a key-value DBMS (non-relational). We suspect that the absence of Marklogic, RiakKV, and Virtuoso in our corpus is related to their low popularity. In fact, in the DB-Engines ranking (DB-ENGINES, 2022), they appear at the 52nd, 69th, and 75th positions, respectively[2]. Regarding the absence of FileMaker, we have not identified any explanation from the available data, as it is a well-established Database Management System, holding the 23rd position in the aforementioned ranking.

Another interesting aspect concerns analyzing the characterization of the project domains (see Section 3.1) regarding adopting relational, non-relational, or both models. Figure 6 presents the analysis grouped by project domain and data model. The intention of this analysis was to discover which data models are mostly used in the various project domains of our corpus. We found 21 projects from the infrastructure management domain using multi-models, whereas only 7 projects used relational models and 4 used non-relational models. The infrastructure management domain showed the highest adoption of multi-model DBMSs. As examples, the *Apache's Dolphinscheduler* project showed adoption of MySQL, H2, PostgreSQL, Oracle, MS SQL Server, IBM DB2, Hive, ClickHouse, Redis, Ehcache, and HBase; the *Netflix's Conductor* project showed adoption

[2] Note that these positions refer to the BD-Engines ranking on February 2022, when we selected the DBMS that we would use in our research.

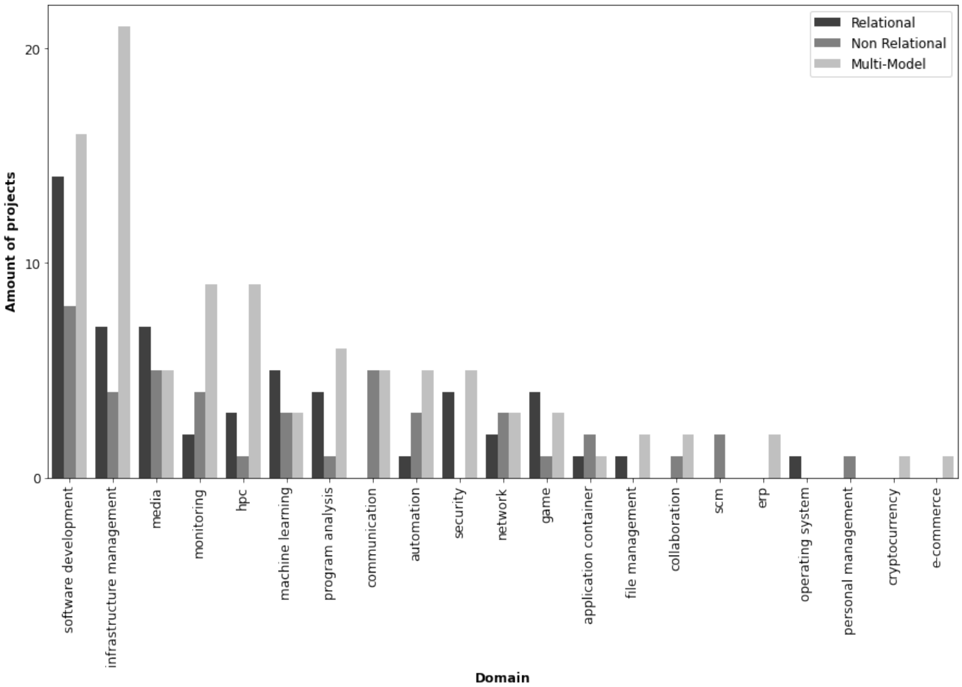

Figure 6: Distribution of DBMS models by project domains

of MySQL, PostgreSQL, Redis, Elasticsearch, and Cassandra; and the *Apache's Beam* project showed adoption of MySQL, PostgreSQL, Hive, Google BigQuery, ClickHouse, Snowflake, Redis, Ehcache, MongoDB, Elasticsearch, Google Cloud Data Store, Hazelcast, HBase, Cassandra, DynamoDB, Solr, and InfluxDB.

We also observed that multi-model DBMSs surpass relational and non-relational models in the Software development, Program analysis, Security, HPC, Monitoring, Automation, File management, and Collaboration domains. Thus, out of the 21 domains in our corpus, the usage of multi-model DBMSs is predominant in 9, corresponding to almost 43%. This discovery reinforces the trend of using more than one distinct data model and that they can complement each other (SAHATQIJA et al., 2018). For instance, all projects from the ERP, Cryptocurrency, and E-commerce domains use only multi-model DBMSs. We also observed that some specific domains use only one of the models. In this sense, projects from the SCM and Personal Management domains only use non-relational DBMSs, and projects from the Operating system domain only use relational models.

RQ1: What are the most commonly used DBMSs during the projects' life cycle?
Answer: MySQL, H2, and PostgreSQL are among the three most used relational DBMSs, while Redis and Cassandra are the most used non-relational DBMSs. Approximately half of the projects (49.75%) adopted both relational and non-relational databases. This co-occurrence of models was especially prevalent in infrastructure management projects.
Implications: Certain DBMSs have consistently maintained a high level of adoption, making them strong candidates to consider when choosing a DBMS.

4.2 Which DBMSs are often used together? (RQ2)

With this question, we explored whether the DBMSs were used simultaneously within the same project and how often this occurred. For this analysis, we adopted association rules to extract patterns that indicate the existence of concomitant use of DBMSs in our corpus. To perform our historical analysis, we made a snapshot from the beginning, middle, and end of the projects' life cycles and compared the results obtained in these three moments (slices). We mined the three slices by applying the Apriori algorithm and generated a heat map to represent the correlations between the DBMS in each slice, as is shown in Figures 7, 8, and 9. We used a minimum frequency of five projects (minimum support of 5), which means we only considered the correlations that occurred in at least five projects to characterize a pattern. Note that the upper diagonals of the heat maps for each slice were eliminated due to redundancy. In addition, the blank cells in the lower diagonal have a frequency below five projects.

Figure 7 presents an overview of the joint usage of 11 DBMSs early in the projects' life cycle (corresponding to the slice that cont first 10% commits as explained in Section 3.2). Among the most used DBMSs, we find PostgreSQL and MySQL in 27 projects, Oracle and MySQL in 20, and H2 and MySQL in 19. Notice that these are the same DBMSs most used individually, as we discussed in Section 4.1. This demonstrates that certain DBMS combinations are preferred choices for developers and are widely adopted in various projects. Several factors can be attributed to the popularity of these combinations, such as complementary features provided by these DBMSs, performance characteristics, integration facility, and even developer familiarity. Thus, using these DBMSs together

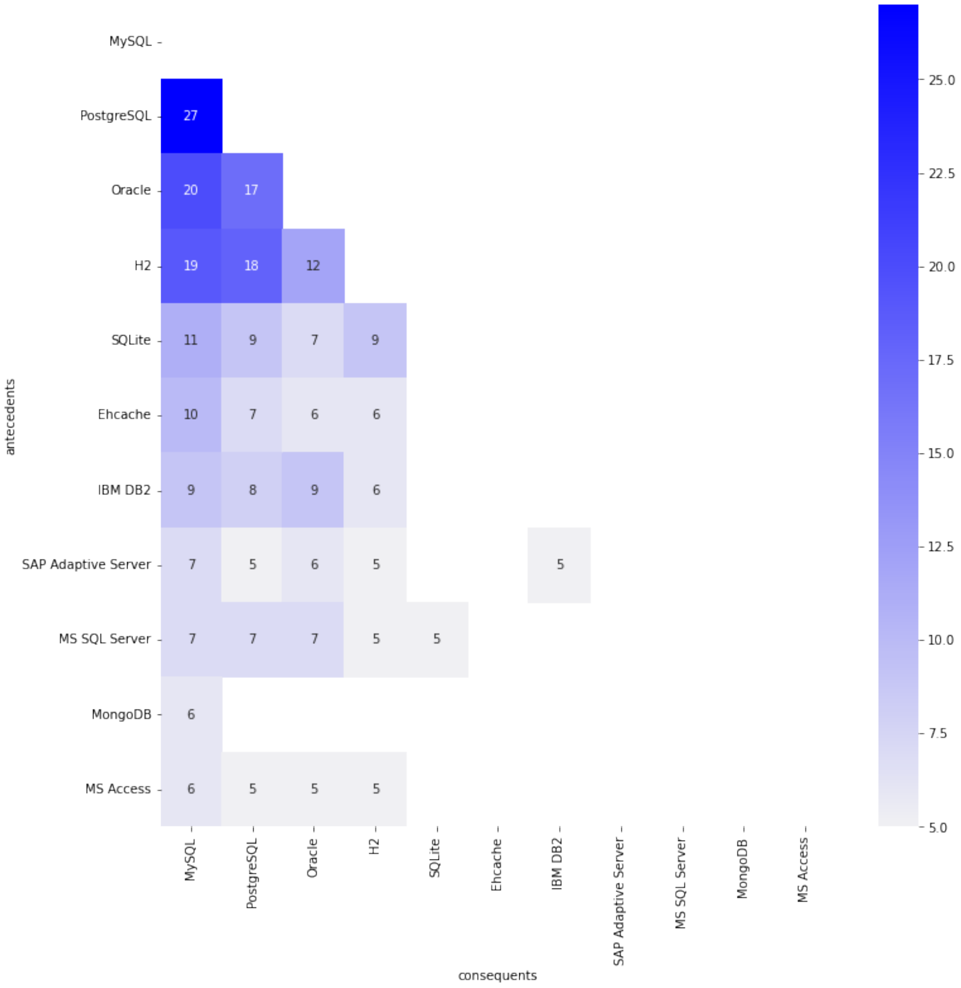

Figure 7: Correlation of the most frequent DBMSs at the beginning of the projects' life cycle

can provide more accurate solutions satisfying diverse requirements in different projects.

Out of the 11 DBMSs found, 9 are relational, and only 2 are non-relational. Consequently, the most frequent combinations involved relational DBMSs. Examples of combining relational and non-relational DBMSs are Ehcache and MySQL, which appear in 10 projects, and MongoDB and MySQL, in 6 projects. No combinations among non-relational DBMSs were observed in the projects, demonstrating that in the initial phase, the combinations between relational DBMSs are more commonly used than non-relational DBMSs. A possible explanation is the age of the projects and the fact that non-relational DBMSs has gained popularity over time.

Halfway through the life cycle of the projects, the amount of combined use of DBMSs more than doubled when compared to the life cycle's beginning. It went from 11 to 25 DBMSs, as shown in Figure 8. This means that the projects employed more DBMSs

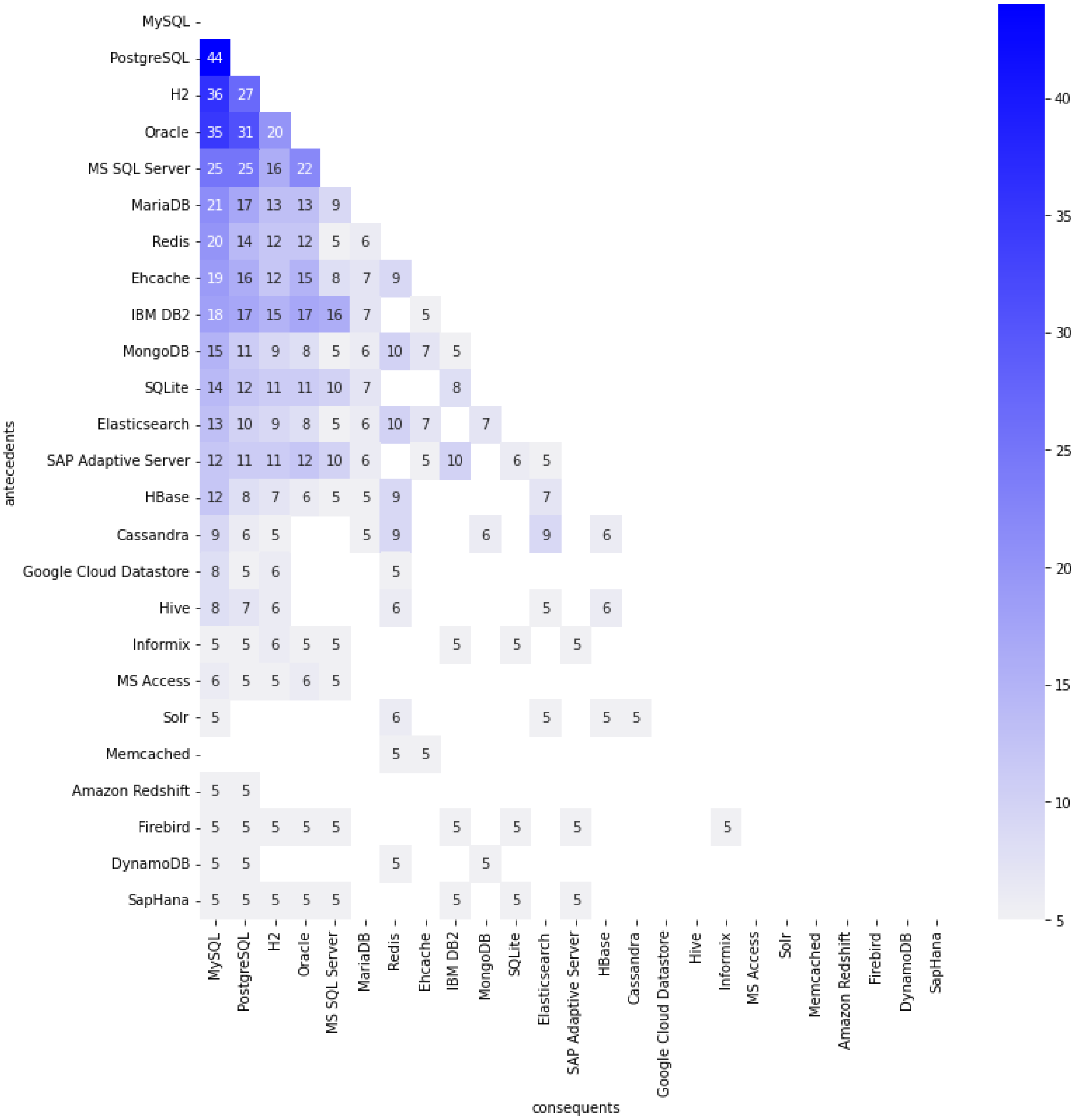

Figure 8: Correlation of the most frequent DBMSs in the middle of the projects' life cycle

together as they mature. This increase may be related to the more complex database needs arising as projects undergo significant changes during their life cycle. Despite the increase in the variety of DBMS combinations, the most frequent combinations remain consistent with those observed at the projects' beginning. PostgreSQL and MySQL, H2 and MySQL, and Oracle and MySQL remain the three most used combinations, significantly increasing to 44, 36, and 35 projects, respectively. We also highlight the increase in the frequency of combinations involving MS SQL Server and the emergence of combinations involving MariaDB and Redis. These may reflect the search for more customized solutions to the projects' database needs as they mature.

Moreover, the number of non-relational DBMSs increased to 10, i.e., five times more

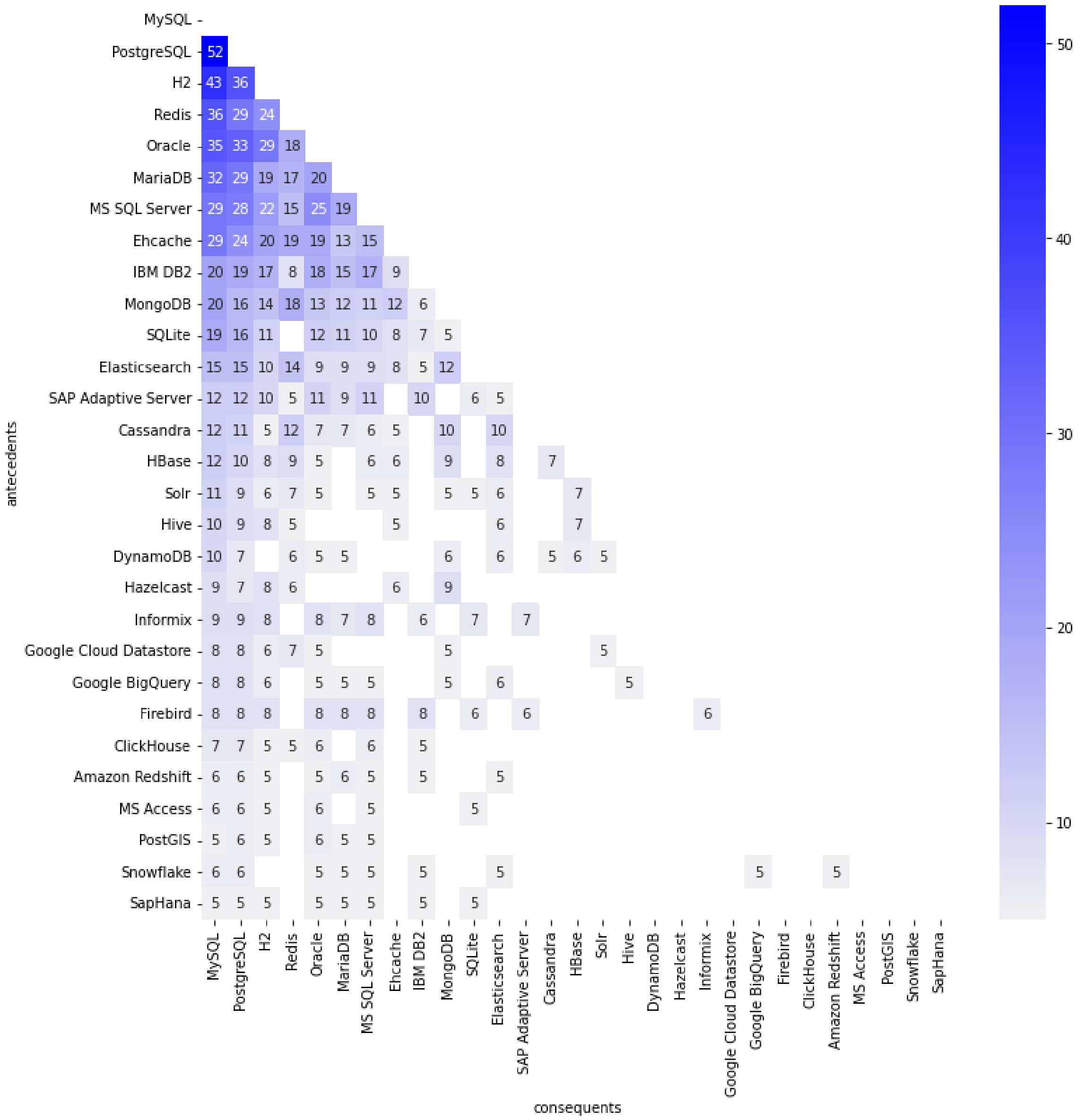

Figure 9: Correlation of the most frequent DBMSs at the end of the projects' life cycle

than the number found at the projects' beginning, indicating that these DBMSs are gaining force as complementary alternatives to relational DBMSs in various project scenarios. We noticed new usage combinations involving these DBMSs, such as Redis and MySQL in 20 projects, MongoDB and PostgreSQL in 11 projects, and Elasticsearch and Redis in 10 projects. In addition, the frequency of combinations involving Ehcache and MongoDB increased compared to the initial phase, Ehcache with MySQL in 19 and MongoDB with MySQL in 15 projects. Thus, we observed more combinations involving the two models and the emergence of combinations of non-relational DBMSs. However, the combinations containing only relational DBMSs remain predominant in this phase.

According to Figure 9, the number of combined DBMSs increases to 29 at the end of the project life cycle, showing moderate growth compared to the 25 DBMSs found at the middle of the project life cycle. However, the growth rate from the beginning (11 DBMSs) to the middle (25 DBMSs) is significantly higher than that from the middle to the end. This demonstrates that until the middle, there is a wider exploration that meets the projects' demands. However, towards the end of the project life cycle, the number of DBMS combinations stabilizes, perhaps due to identifying more efficient and effective DBMS combinations leading to a more controlled growth.

MySQL, PostgreSQL, H2, and Oracle remain among the most frequent combinations throughout the projects' life cycle. Although Oracle lost its position to Redis, the combinations involving these DBMSs remain popular choices in various projects. Interestingly, we could observe an increase in the number of projects adopting Redis together with other DBMS over time. Still, we could also observe a decrease in the number of DBMSs used with Redis over time. The Redis combinations became more frequent among projects, indicating its increasing popularity in joint usage at the end of the projects life cycle. According to Cattel (CATTELL, 2011), Redis is a single-node key-value storage DBMS suitable for applications that search objects by a single attribute. Thus, this increase in popularity may mean it is being used for simple data manipulation. Nevertheless, the amount of DBMSs used in combination with Redis in the middle of the projects (MySQL, PostgreSQL, H2, Oracle, MS SQL Server, and MariaDB) has reduced by half at the end of the projects (MySQL, PostgreSQL, H2). This decline may be related to projects replacing several relational DBMS choices with fewer ones instead of necessarily meaning that Redis had influenced that reduction.

The combinations involving non-relational DBMSs—such as Ehcache, MongoDB, Elasticsearch, Cassandra, and HBase—become more frequent from the middle to the end of the analyzed project histories. This growth is reflected in combinations among the two models (for instance, Ehcache with MySQL in 29, MongoDB with MySQL in 20 projects, Elasticsearch with PostgreSQL in 15 projects) and combinations containing only non-relational DBMSs, such as Cassandra with Redis in 12 projects, and HBase with MongoDB in 9 projects. This reaffirms an increasing trend of joint use of non-relational DBMSs in the projects' advanced stages.

Overall, these analyses indicate that the choice of DBMS combinations evolves over the projects' life cycle, with certain DBMSs gaining popularity while others seeing fluctuations in their usage patterns. Non-relational DBMSs, in particular, became more prevalent in combinations as projects progress, possibly due to their advantages in handling certain

Table 10: Top 10 rules with highest lifts in the *first slice* of the project life cycle

A	**B**	**Sup (A)**	**Sup (B)**	**Sup (A→B)**	**Conf (A→B)**	**Conf (B→A)**	**Diff**	**Lift (A→B)**
SAP Adaptive Server	IBM DB2	7	9	5	0.71	0.56	0.16	15.63
IBM DB2	Oracle	9	25	9	1.00	0.36	0.64	7.88
MS SQL Server	SQLite	8	16	5	0.63	0.31	0.31	7.70
MS SQL Server	Oracle	8	25	7	0.88	0.28	0.60	6.90
SAP Adaptive Server	Oracle	7	25	6	0.86	0.24	0.62	6.75
MS Access	Oracle	6	25	5	0.83	0.20	0.63	6.57
MS Access	H2	6	31	5	0.83	0.16	0.67	5.30
IBM DB2	PostgreSQL	9	36	8	0.89	0.22	0.67	4.86
MS SQL Server	PostgreSQL	8	36	7	0.88	0.19	0.68	4.79
MS Access	PostgreSQL	6	36	5	0.83	0.14	0.69	4.56

Table 11: Top 10 rules with highest lifts in the *fifth slice* of the project life cycle

A	**B**	**Sup (A)**	**Sup (B)**	**Sup (A→B)**	**Conf (A→B)**	**Conf (B→A)**	**Diff**	**Lift (A→B)**
Firebird	Informix	5	6	5	1.00	0.83	0.17	32.83
Firebird	SAP Adaptive Server	5	12	5	1.00	0.42	0.58	16.42
SapHana	SAP Adaptive Server	5	12	5	1.00	0.42	0.58	16.42
Informix	SAP Adaptive Server	6	12	5	0.83	0.42	0.42	13.68
Solr	Cassandra	7	11	5	0.71	0.45	0.26	12.79
Solr	HBase	7	13	5	0.71	0.38	0.33	10.82
SapHana	IBM DB2	5	19	5	1.00	0.26	0.74	10.37
Firebird	IBM DB2	5	19	5	1.00	0.26	0.74	10.37
Hive	HBase	9	13	6	0.67	0.46	0.21	10.10
Firebird	SQLite	5	22	5	1.00	0.23	0.77	8.95
SapHana	SQLite	5	22	5	1.00	0.23	0.77	8.95

data types and workloads.

We further analyzed whether using a particular DBMS increases the chance of using another one. To do this, we filtered the rules with the highest lift values for the three moments in the project history, as shown in Tables 10, 11, and 12. In Table 10, we find the rule Sap Adaptive Server → IBM DB2 with a lift of 15.63 in the first slice, the highest lift found early in the project history. This means that using SAP Adaptive Server increases by 15.63 times the chance of using IBM DB2 and vice versa. Besides, the rule's confidence is 71% (Conf(A→B) = 0.71) when we have Sap Adaptive Server as antecedent and IBM DB2 as consequent, and about 56% (Conf(B→A) = 0.56) the other way around. The slight difference of 16% (Diff = 0.16) between the confidences shows that both are close, so this rule has no majority direction. Thus, when SAP Adaptive Server is used, there is a high chance of using IBM DB2 and vice versa. The next rule with a lift of 7.88 is IBM DB2 → Oracle with 100% confidence (Conf(A→B) = 1.00) in this direction and 36% (Conf(B→A) in the opposite direction. This 64% difference (Diff = 0.64) between the confidence measures indicates that using IBM DB2 leads to using Oracle, but not the other way around. Thus, in 100% of the cases where IBM DB2 occurs, Oracle also occurs.

Furthermore, the presence of IBM DB2 increases the occurrence of Oracle by 7.88 times.

We noticed that during the middle of the projects' life cycle, the lift values of the top 10 rules (Table 11) nearly doubled when compared to the lift values of the top 10 rules (Table 10) identified at the beginning of the project's life. The increased lift suggests that specific DBMS correlations become stronger, indicating possible dependencies or synergies between them as the projects mature. For example, the rule Firebird → Informix has a lift of 32.83 and 100% confidence (Conf(A→B) = 1.00) in this direction, and 83% confidence (Conf(B→A) = 0.83) in the opposite direction. Due to the small difference between the two confidences (Diff = 0.17), we cannot perceive a majority direction in this rule, only a balanced dependency in both directions. This indicates a strong correlation between both DBMSs, meaning that when Firebird is chosen as a database, Informix will likely be used in the same project, and the reciprocal is true.

Both rules, Firebird → Sap Adaptive Server and Sap Hana → Sap Adaptive Server present a lift value of 16.42. This indicates that adopting Firebird or Sap Hana increases the Sap Adaptive Server adoption by 16.42 times and vice-versa. Additionally, when Sap Adaptive Server is in the consequent (B), the confidence is 100% (Conf(A→B) = 1.00), while it is 42% (Conf(B→A) = 0.42) when it is in the antecedent position (A). The significant difference (Diff = 0.58) between the confidences of these rules indicates a strong dependency. In other words, the adoption of Firebird or Sap Hana is highly influenced by Sap Adaptive Server adoption since in 5 of the 12 times it occurred, Firebird or Sap Hana occurred. Still, every time Firebird or Sap Hana occurred, Sap Adaptive Server also occurred. Moreover, a few rules—Firebird → IBM DB2, DB2 SapHana → IBM DB2, Firebird → SQLite, and SapHana → SQLite rules—although with lower lifts, present significant differences (Diff = 0.74; 0.74; 0.77; 0.77, respectively) in even greater confidence values than the previous ones, demonstrating that these DBMSs also have a strong dependency relationship. Again, Firebird and Sap Hana are only adopted when other DBMSs are adopted. This indicates that Firebird and Sap Hana are not typically the initial choices for projects. Instead, they are combined with IBM DB2, SQLite, and Sap Adaptive Server, suggesting Firebird and Sap Hana are suitable add-ons for these DBMSs, leading to their combined adoption.

Finally, in Table 12, we present the top 10 rules with the highest lifts found at the end of the projects' life cycle. We observe that lifts remain high towards the end of the projects' life cycles, demonstrating strong co-occurrences in adopting different DBMSs. Another relevant aspect is the null difference (Diff = 0.00) to confidence in both direc-

Table 12: Top 10 rules with highest lifts in the *last slice* of the projects' life cycle. On this table, SAP AS stands for SAP Adaptive Server.

A	**B**	**Sup (A)**	**Sup (B)**	**Sup (A→B)**	**Conf (A→B)**	**Conf (B→A)**	**Diff**	**Lift (A→B)**
Snowflake	Amazon Redshift	6	6	5	0.83	0.83	0.00	27.36
Snowflake	Google BigQuery	9	6	5	0.83	0.56	0.28	18.24
Firebird	Informix	9	8	6	0.75	0.67	0.08	16.42
Informix	SAP AS	9	12	7	0.78	0.58	0.19	12.77
Firebird	SAP AS	8	12	6	0.75	0.50	0.25	12.31
Google BigQuery	Hive	9	11	5	0.56	0.45	0.10	9.95
SapHana	IBM DB2	5	20	5	1.00	0.25	0.75	9.85
Firebird	IBM DB2	8	20	8	1.00	0.40	0.60	9.85
Hive	HBase	11	15	7	0.64	0.47	0.17	8.36
Amazon Redshift	IBM DB2	6	20	5	0.83	0.25	0.58	8.21
Snowflake	IBM DB2	6	20	5	0.83	0.25	0.58	8.21
SAP AS	IBM DB2	12	20	10	0.83	0.50	0.33	8.21

Table 13: Similarities between DBMSs of the same vendors

Slice	**A**	**B**	**Sup (A)**	**Sup (B)**	**Sup (A→B)**	**Conf (A→B)**	**Conf (B→A)**	**Diff**	**Lift (A→B)**
Fifth	MariaDB	MySQL	23	81	21	0.91	0.26	0.65	2.22
Last	MariaDB	MySQL	36	87	32	0.89	0.37	0.52	2.01
Last	PostGIS	PostgreSQL	6	61	6	1.00	0.10	0.90	3.23

tions of the first rule, Snowflake → Amazon Redshift, which makes it difficult to identify the dependency direction. In 83% (Conf(A→B) = 0.83) of the cases where Snowflake occurred, Amazon Redshift also occurred, and vice versa. Besides, Snowflake increases the usage of Amazon Redshift about 27 times and vice versa. Therefore, when one of these DBMSs is chosen, it highly influences the choice of the other. In contrast, we have the Snowflake → IBM DB2 rule, with a significant difference (Diff = 0.58) between the confidences, indicating a strong dependency of Snowflake on IBM DB2. This means that in 5 out of 20 of the cases where IBM DB2 was adopted, Snowflake was also adopted, and in the majority of cases (5 out of 6) where Snowflake was adopted, IBM DB2 was adopted. The DBMS adoption increases the adoption of the other by about 8 times. Snowflake also appears to be used as a complementary DBMS, as it is always combined with other DBMSs, as already mentioned. According to Snowflake's documentation [3], it is a relational DBMS that combines a completely new SQL query engine with an innovative architecture natively designed for the cloud. The IBM DB2 documentation [4] also mentions a data integration tool, DataStage, which allow it to build reliable data pipelines, orchestrate data in distributed environments and move and transform data between cloud sources and data warehouses, with a Snowflake connector, among other things. Possibly,

[3] https://docs.snowflake.com/en/user-guide/intro-key-concepts

[4] https://dataplatform.cloud.ibm.com/docs/content/wsj/getting-started/get-started-datastage-snowflake.html?context=cpdaas&audience=wdp

because of these features, it has become a suitable complement for some mature projects that needed to extend their support to cloud storage without compromising the properties of the relational model.

We also noted a fluctuation in the lift values and confidence measures of some rules throughout the different stages of the project history. For example, the rule SAP Adaptive Server → IBM DB2 appears with a lift of 15.63 and confidences of 71% (Conf(A→B) = 0.71) and 56% (Conf(B→A) = 0.56) at the beginning of the project history. In the middle, the lift decreases to 8.64, the **A→B** confidence increases to 83%, and the **B→A** confidence decreases to 53%. In the end, the lift decreases slightly to 8.21, **A→B**'s confidence remains at 83%, but **B→A**'s confidence decreases to 50%. Similarly, the rule Firebird → Informix, discovered in the middle of the project history, initially exhibits a high lift of 32.83 with 100% (Conf(A→B) = 1) and 83% (Conf(B→A) = 0.83) confidences, but the lift decreases by half, 16.42, with confidences 75% (Conf(A→B) = 0.75) and 67%(Conf(B→A) = 0.83) at the end. This fluctuation in lift values and confidence measures indicates that the associations and dependencies between certain DBMS combinations are not static and may vary during the project life cycles. Various factors might have influenced these variations, such as changes in project requirements, technological advancements, or shifts in the development team's preferences. Despite the decrease in the initial lift values, the highlighted rules continue to represent significant associations between the aforementioned DBMS combinations. Thus, although the strength of the relationships may change, there is still some level of dependency or association between certain DBMSs that may become more or less prevalent as projects mature. We also observe that, in the early stage of the projects' cycle, the usage of a relational DBMS combined with another relational DBMS was more frequent. As observed in Table 10, they were contained in the 10 rules that are relational. However, this situation differs from the middle to the end of the project life cycle because we find non-relational DBMSs in the rules highlighted in Tables 11 and 12. However, it is important to consider that even among primarily relational DBMSs, there might have been cases where a secondary model of one of the DBMSs is non-relational. For instance, Sap Adaptive Server and IBM DB2 are primarily relational but support non-relational data models. Sap Adaptive Server also supports the Spatial Store data types, while IBM DB2 supports the Document Store, RDF Store, and Spatial Store types. This may suggest that the joint utilization of these DBMSs might be driven by the need for a specific data type not supported by the other DBMS, or to complement certain capabilities that one of the DBMSs lacks. Additionally, the compatibility between the two DBMSs, both developed in C/C++, might have facilitated their combined usage.

In our analysis (Table 13), we have observed the occurrence of rules where DBMSs from the same vendors are used together. For instance, Maria DB and MySQL co-occur from the middle to the end of the projects' life cycle. Similarly, in the end, we noticed the co-occurrence of PostGIS and PostgreSQL. Although we address the evidence of MariaDB and MySQL usage individually in our study, we point out that MariaDB emerged as a fork of MySQL [5] to maintain high fidelity and acceptance. However, despite the current versions of MariaDB having additional functionality compared to MySQL, there remains a commitment to maintaining compatibility between the two DBMSs [6]. For this reason, the co-occurrence between the two DBMSs is expected.

When analyzing the differences (Diff =0.65, 0.52) in confidence between the two rules involving MariaDB and MySQL in the fifth and last slice, respectively, a strong dependence of MariaDB on MySQL is observed. At the projects' mid-life cycle (fifth slice), in 91% (Conf(A→B)= 0.91) of the cases where MariaDB occurred, MySQL also occurred, and in only 21% (Conf(B→A)= 0.21) of the cases where MySQL occurred, MariaDB also occurred. Furthermore, MariaDB's usage increased MySQL's usage by 2.22 times. By the end of the life cycle, the values vary slightly: the lift and **A→B** confidence decreases, respectively, to 2.01 and 89% (Conf(A→B)), but the **B→A** confidence increases to 37%(Conf(B→A) = 0.37). The fluctuation in these values indicates that the dependency loses a little strength but still remains significant. This suggests that projects tend to use MariaDB and MySQL together at the end of their life cycle.

Another expected co-occurrence between DBMSs happens at the end of the project life cycle: PostGIS and PostgreSQL. Since PostGIS is a spatial DBMS extension of PostgreSQL, it transforms PostgreSQL into a spatial DBMS by adding support for three features: spatial types, spatial indexes, and spatial functions[7]. The rule with PostGIS → PostgreSQL, whose confidences are 100%(Conf(A→B) = 1.00) and 10%(Conf(B→A) = 0.10), also demonstrates a strong dependency of PostGIS on PostgreSQL. Therefore, in 100% of the cases where PostGIS was used, PostgreSQL was also used, while in only 10% of cases where PostgreSQL was used, PostGIS was also used. Furthermore, using PostGIS increases the chance of using PostgreSQL by 3.23 times. Considering that PostGIS depends on PostgreSQL to work, this 100% confidence is expected. This information suggests that the projects that require spatial database functionality will probably adopt PostGIS and PostgreSQL together, reinforcing the idea that the co-occurrence between

[5] https://mariadb.org/en/

[6] https://mariadb.com/kb/en/mariadb-vs-mysql-compatibility/

[7] https://postgis.net/

DBMSs is intended to meet the specific demands of the projects.

RQ2: Which DBMSs are often used together?
Answer: We found co-occurrences involving 11 DBMSs at the project beginning, prevailing the combinations among relational DBMSs: PostgreSQL and MySQL, Oracle and MySQL, and H2 and MySQL. In the middle of the project life cycle, the number of DBMSs that appear in co-occurrence increases to 25. Ten of them include non-relational DMBSs, such as Redis and MySQL, MongoDB and PostgreSQL, and Elasticsearch and Redis. The combined usage of PostgreSQL and MySQL, Oracle and MySQL, H2 and MySQL get even more popular in the middle of the life cycle. At the end of the project life cycle, the number of DBMSs used in combination with others increased to 29 DBMSs, with MySQL, PostgreSQL, H2, and Oracle among the most popular relational ones. However, Oracle lost its position to Redis. Our analysis also revealed that using some DBMSs increases the chance of adopting another in parallel. This is the case for SapAdaptiveServer and IBM DB2, IBM DB2 and Oracle at the beginning of the project life cycle, Firebird and Informix in the middle, and Snowflake and Amazon Redshift towards the project life cycle end.
Implications: The results offer a valuable understanding of prevalent combinations and may help to perceive potential synergies or challenges in utilizing DBMSs together.

4.3 Which DBMS are frequently replaced by others? (RQ3)

In this question, we look for the existence of substitutions between DBMSs and quantify the frequency they occur throughout the projects' life cycle.

Figure 10 presents the frequency of DBMSs being kept or removed across the history of the projects. MySQL was kept throughout the life cycle of 76 projects but removed in 31 projects. This means that about 39% of the projects in which we found that a DBMS was used kept using MySQL, and it was removed in about 16%. The removals happened for the majority of DBMSs surveyed, except for ClickHouse (7 projects), Etcd (5 projects), Realm (5 projects), MicrosoftAzureCosmosDB (4 projects), Ignite_NoSQL (3 projects), Netezza (3 projects), OrientDB (3 projects), CouchDB (2 projects), Impala

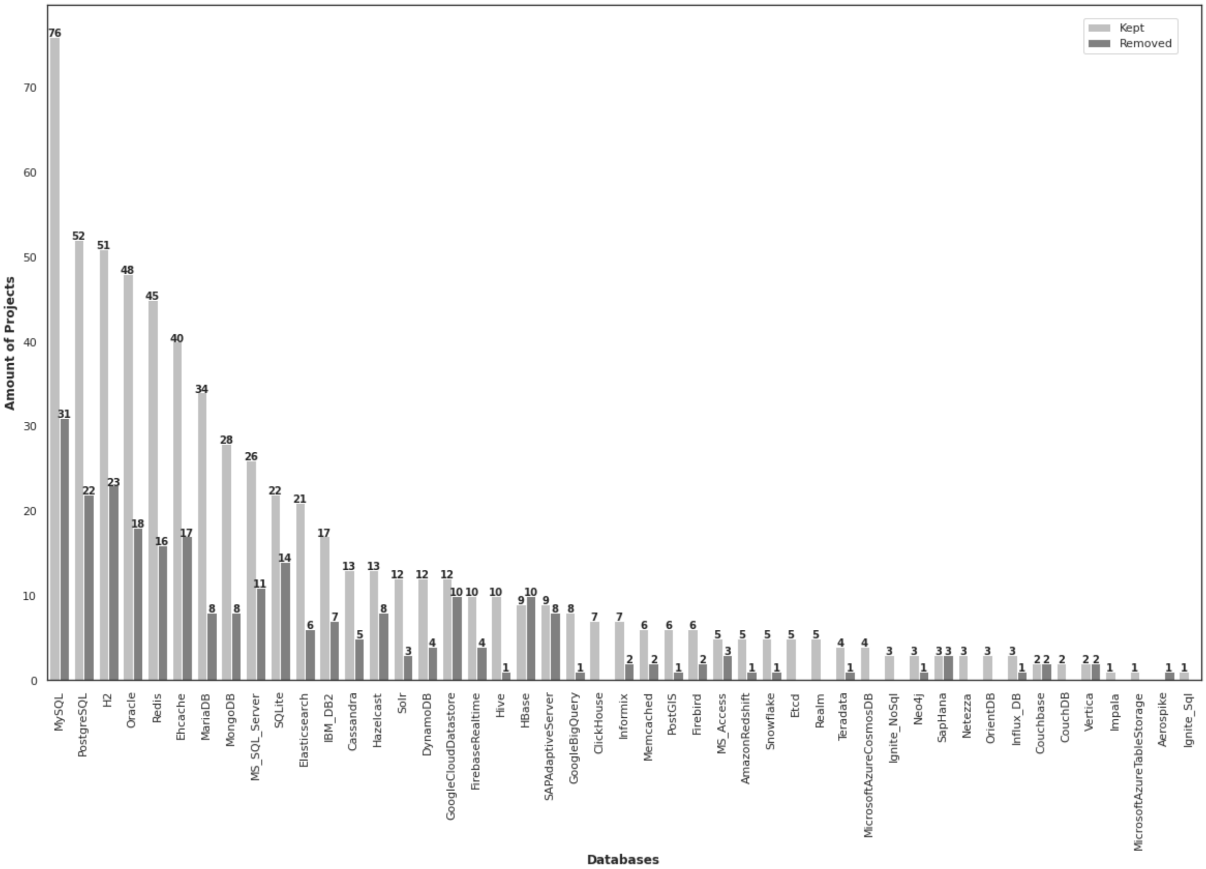

Figure 10: Distribution of DBMS permanences and removals in the projects

(1 project), MicrosoftAzureTavleStorage (1 project), and Ignite_SQL (1 project). As one of our goals was to understand the frequency of DBMS replacements, the significant amount of removals we found had created the expectation that substitutions are frequent. To investigate this, we utilized sequential patterns mining (FOURNIER-VIGER et al., 2017). Each item set in the sequence corresponds to ten moments (slices) of each of the 197 projects' history. We thus have 197 sequences in total that were coded according to the patterns shown in Section 3.5. After generating the patterns, we filtered out those that met the established patterns to characterize the replacements, as discussed in Section 3.5.

Table 14 shows 20 sequential patterns demonstrating DBMS replacements, of which we highlight: $PostgreSQL \rightarrow MariaDB_{In}PostgreSQL_{Out} \rightarrow MariaDB$ with support $= 3$, indicating that three projects used PostgreSQL, in a later slice they started using MariaDB and stopped using PostgreSQL, maintaining the MariaDB use in a later slice. The pattern $Oracle \rightarrow MySQL_{In} \rightarrow Oracle_{Out}MySQL$ with support $= 3$ indicates that three projects used Oracle; in a later slice, they started using MySQL, stopped using Oracle, and kept using MySQL in a later slice. These two examples present replacement patterns whose changes occur in different ways. While in the first example, MariaDB started at the same

Table 14: DBMS replacement patterns.

Pattern	**Support**	**Occurrences per DBMS**
Ehcache → Redis$_{In}$ Ehcache$_{Out}$ → Redis	2	2
GoogleCloudDatastore → MySQL$_{In}$ → GoogleCloudDatastore$_{Out}$ MySQL	2	2
H2 → MariaDB$_{In}$ H2$_{Out}$ → MariaDB	2	4
H2 → MySQL$_{In}$ → H2$_{Out}$ MySQL	2	
HBase → MariaDB$_{In}$ → MariaDB HBase$_{Out}$	2	4
HBase → HBase$_{Out}$ Cassandra$_{In}$ → Cassandra	2	
MS_SQL_Server → Redis$_{In}$ MS_SQL_Server$_{Out}$ → Redis	2	2
MySQL → H2$_{In}$ → H2 MySQL$_{Out}$	2	4
MySQL → Redis$_{In}$ MySQL$_{Out}$ → Redis	2	
Oracle → MariaDB$_{In}$ → MariaDB Oracle$_{Out}$	2	9
Oracle → MySQL$_{In}$ → Oracle$_{Out}$ MySQL	3	
Oracle → PostgreSQL$_{In}$ → Oracle$_{Out}$ PostgreSQL	2	
Oracle → SQLite$_{In}$ → Oracle$_{Out}$ SQLite	2	
PostgreSQL → Cassandra$_{In}$ → PostgreSQL$_{Out}$ Cassandra	2	11
PostgreSQL → MariaDB$_{In}$ PostgreSQL$_{Out}$ → MariaDB	3	
PostgreSQL → MySQL$_{In}$ → PostgreSQL$_{Out}$ MySQL	2	
PostgreSQL → Oracle$_{In}$ PostgreSQL$_{Out}$ → Oracle	2	
PostgreSQL → PostgreSQL$_{Out}$ MySQL$_{In}$ → MySQL	2	
Solr → MariaDB$_{In}$ → MariaDB Solr$_{Out}$	2	2
SQLite → H2$_{In}$ → H2 SQLite$_{Out}$	2	2

time that PostgreSQL left, in the second example, MySQL enters at any moment, and at a later time when Oracle leaves, MySQL is maintained. Although the patterns confirmed the replacement's existence, they do not indicate that a specific substitution occurred frequently—since each pattern demonstrates that DBMSs were replaced in at least 2 and at most 3 projects (Support = $\{2.3\}$).

Furthermore, we found 12 replacement patterns among relational DBMSs across 26 projects, 2 patterns of replacements among non-relational DBMSs across 4 projects, and 6 patterns of replacements involving both models across 12 projects. This indicates that in 70% of the cases, the replacements occur between DBMSs that have the same data model. This may mean that data integrity is the most relevant criterion when choosing a replacement. Replacements among non-relational DBMSs occur in only 10% of the cases, such as $HBase \rightarrow HBase_{Out}Cassandra_{In} \rightarrow Cassandra$. This indicates that the substitution between non-relational DBMSs is rarer, possibly because they provide different data types (graph, key-value, document, etc.), so replacing them is not an easy task. Finally, in 30% of the cases, the replacements occur among distinct data models, as $MSSQLServer \rightarrow Redis_{In}MSSQLServerr_{Out} \rightarrow Redis$. This may occur due to some new desirable property that the DBMS in use could not provide, as reported by Gessert et al. (2017).

As depicted in Table 14, it is possible to notice that some DBMSs are replaced by several others, so we analyzed the count of all that were replaced (*Occurrences per DBMS replaced*, see Table 14), regardless of the DBMS that replaced them. PostgreSQL was replaced 11 times in 8 projects, being replaced 4 times by MySQL. In the opposite direction, PostgreSQL replaced MySQL, we did not find a pattern within the defined support limit. This indicates that PostgreSQL tends to be more replaced than MySQL. MySQL was removed 4 times in 4 projects. In 2 of them, it was replaced by H2, and in 2 projects by Redis. H2 was replaced by MySQL 2 times in 2 projects, demonstrating that replacements may occur in both directions. Oracle is the second most replaced DBMS (9 times in 6 projects), 3 times replaced by MySQL, with no replacements in the opposite direction. Next, we have HBase being replaced 4 times in 4 projects, 2 times replaced by Cassandra, also with no replacements in the opposite direction. The remaining DBMSs were replaced only 2 times: Ehcache, GoogleCloudDataStore, MS SQL Server, Sorl, and SQLite. Given these results and the expectations generated by the significant amount of removals 10, we observe that substitutions between DBMSs are not frequent but do occur.

Table 15 presents a more comprehensive analysis by counting all replacements found. This analysis aimed to investigate the relevance of DBMS replacements, considering the frequency at which a particular DBMS was replaced, regardless of the substituting DBMS. PostgreSQL totalizes 25 replacements across 11 projects. Oracle sums up 19 replacements across 8 projects. Notably, HBase, MySQL, EhCache, GoogleCloudDataStore, SQLite, MS SQL Server, H2, and Sorl exceeded three times or more the number of the replacements shown in Table 14. This occurs because they are frequently replaced by another DBMS only once in a single project, resulting in support value = 1. Since the minimum threshold for a pattern to appear in Table 14 is set higher than 1, such occurrences are not included there. Thus, when we analyze only which DBMSs were replaced, without considering by which other DBMS they were replaced, the numbers we get are possibly higher than those presented in Table 14. In total, we observed 129 DBMSs being replaced.

Table 15: Occurrences of DBMSs replacements and count of projects where other DBMSs replaced them

DBMS	**Occurrences**	**Projects**
PostgreSQL	25	11
Oracle	19	8
HBase	16	7
MySQL	15	7
Ehcache	11	9
GoogleCloudDataStore	11	6
SQLite	10	5
MS SQL Server	9	4
H2	8	5
Solr	5	3

Table 16: Occurrences of DBMS substitutes and count of projects where they replaced other DBMSs

DBMS	**Occurrences**	**Projects**
Redis	17	9
MySQL	15	10
MariaDB	15	8
H2	12	7
Cassandra	11	4
HBase	7	4
MongoDB	5	4
Oracle	5	4
PostgreSQL	5	4
MS SQL Server	5	3
Elasticsearch	4	3
Hive	4	2
Join_Ignite_NoSql	4	2
DynamoDB	3	2
SQLite	3	2
GoogleCloudDatastore	2	2
ClickHouse	2	1
Hazelcast	2	1
IBM DB2	2	1
Amazon Redshift	1	1
Couchbase	1	1
Ehcache	1	1
Realm	1	1
Solr	1	1
Teradata	1	1

In summary, we have identified that PostgreSQL is the DBMS mostly susceptible to replacements during the projects' history. Additionally, we observed that all replaced DBMSs have experienced more than one replacement in certain projects. This means that when a DBMS is removed, it may be replaced by more than one alternative DBMS, either in the same slice or in later slices of the project history. For example, in the Airsonic project, we discovered three distinct patterns of Oracle substitutions for three different DBMSs: PostgreSQL, MySQL, and MariaDB.

Conversely, Table 16 presents the DBMSs that replaced others. This analysis aimed to find out which DBMS is mostly used as a replacement, regardless of which DBMS it replaced. Redis replaced other DBMSs 17 times across 9 projects, MySQL and MariaDB replaced other DBMSs 15 times across 10 and 8 projects, respectively. H2, Cassandra, and HBase replaced other DBMSs 12, 11, and 7 times, respectively. Some DBMSs, on the other hand, have replaced other DBMSs, such as Amazon Redshift, Couchbase, Ehcache, Realm, Solr, and Teradata, among others. From this viewpoint, we discovered the DBMS most chosen to replace others with is Redis. The tendency for non-relational DBMSs to outperform relational DBMSs as a substitute is also relevant. Figures 11 and 12, extracted from the DB-Engines website, presents this trend by showing the evolution in the adoption of relational (11) and non-relational (12) DBMSs from 2012 to 2023 (note the logaritmic scale). Although a certain stability is observed in using relational DBMSs, the growth of non-relational DBMSs is notable. Despite the fact that our analysis criterion is based on adoption and DB-Engines on the DBMS citation criterion, both show an increasing trend towards non-relational DBMS. This trend reinforces some of Cattel's predictions (CATTELL, 2011) that NoSQL DBMSs would not be a "passing fad" due to their simplicity, flexibility, and scalability as well as developers would accept these advantages to the detriment of ACID transactions. Thus, this growing preference for non-relational solutions can be related to the business's need to deal with fast lookup or more complex querying capabilities that require a greater volume of data (GESSERT et al., 2017), such as textual searches or choosing a standard for exchanging data.

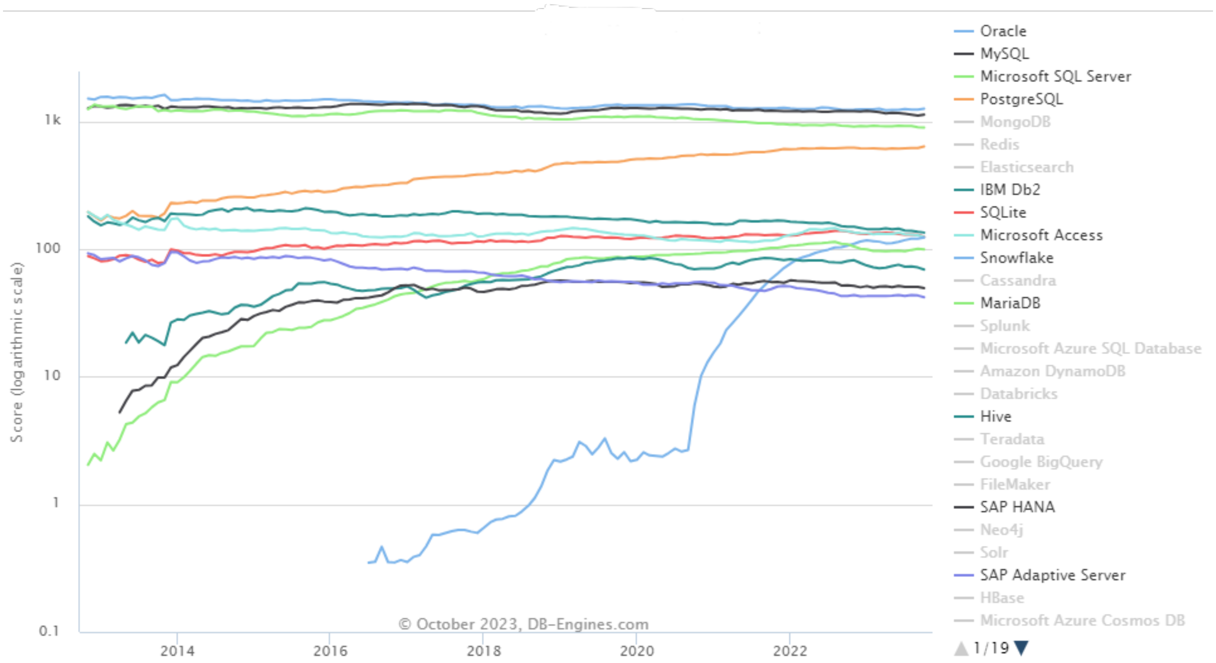

Figure 11: Relational DBMSs trend. Source: DB-Engines.

Comparing Tables 15 and 16, we observe PostgreSQL being replaced by other DBMSs 25 times but replacing other DBMSs only 5 times, indicating a decrease in its adoption. MySQL was replaced by another DBMS 15 times and replaced by another DBMS 15 times, too, while MariaDB replaced another DBMS 15 times but was not replaced. Besides, if we consider that MySQL and MariaDB are the same DBMS (MariaDB is a fork of MySQL), we could infer that MySQL usage remained steady or potentially increased. On the other hand, Redis was not replaced but substituted another DBMS 17 times, suggesting a significant increase in its usage among the projects. This way, we can perceive which

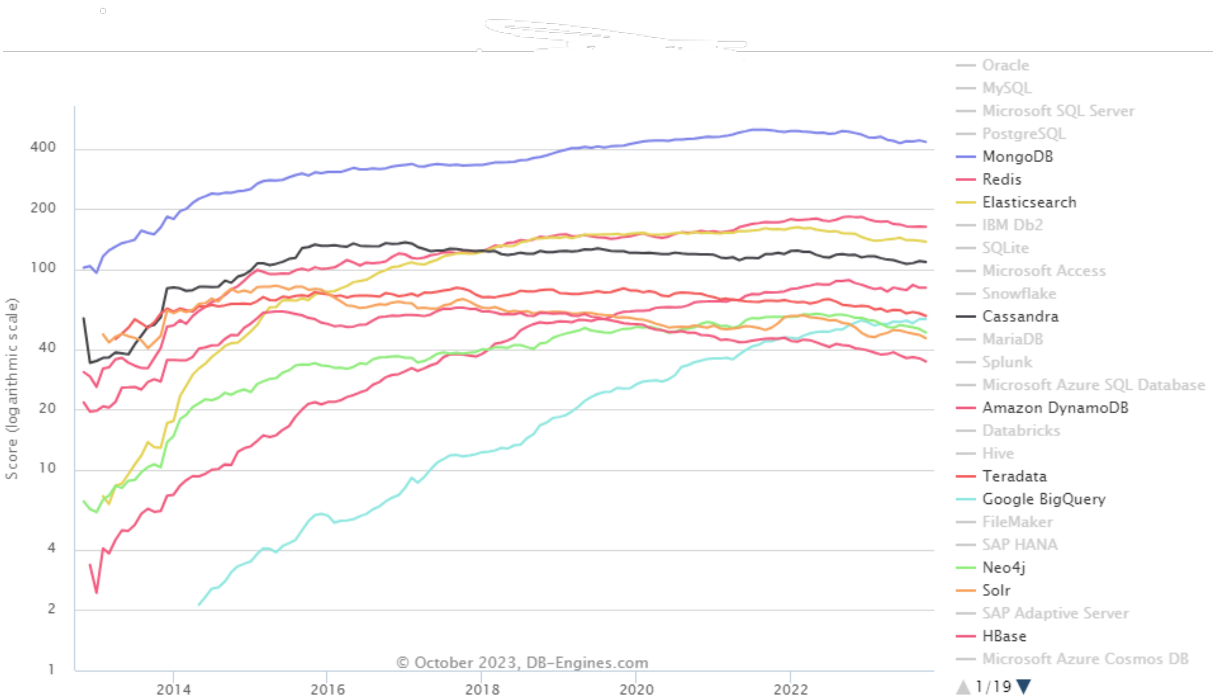

Figure 12: Non-relational DBMSs trend. Source: DB-Engines

DBMSs tend to be discontinued and which tend to be most selected by replacing others.

Nevertheless, despite discovering evidence of DBMS substitutions and some DBMSs being more prone to being replaced than others, we did not observe frequent patterns of specific DBMSs always being substituted by others. The growing co-occurrence among DBMSs, reported in Section 4.2, may influence the situation, making migrations from one DBMS to another unnecessary. This further supports the notion that DBMSs can complement each other, and the projects are utilizing multiple DBMSs to meet their specific needs.

RQ3: Which DBMSs are frequently replaced by others?

Answer: Our results reveal 20 situations of DBMS replacements in projects. The most frequent occurred in only 3 projects: PostgreSQL was replaced by MariaDB and Oracle by MySQL. PostgreSQL was the DBMS that underwent 11 replacements in 8 projects by 4 different DBMSs, including MySQL. In 70% of the cases, the replacements occurred between DBMSs that have the same model. Of that, only 10% of the replacements involve non-relational DBMSs. The remaining 30% of the replacements involve both data models. In a more comprehensive analysis, considering only the replaced DBMS, including the patterns that occurred in only 1 project, we found PostgreSQL and Oracle among the most susceptible to replacement. Also, considering just the DBMS that replaced another one, we found Redis, MySQL, and MariaDB among the most used as a replacement.

Implications: Some DBMSs undergo frequent replacement by other DBMSs. Investigating the existence of replacement patterns allows for knowing the DBMS migration trends in the projects.

4.4 Final Remarks

This Chapter reported the results we obtained for our research questions. Specifically, we report results about the popularity of DBMSs over the history of our projects corpus, the existence of simultaneous use of different DBMSs over the history of projects, and the existence of substitution between DBMSs. We consider this to be relevant information that provides valuable insights for professionals in the field. Chapter 5 presents the conclusions of this work.

5 CONCLUSION

This Chapter presents our conclusions, contributions, and opportunities for future work.

This work presented a broad and unprecedented investigation of the adoption of the 50 top DBMSs over the life cycle of 317 Open Source Java projects. To do so, we sliced the project history into 10 commit intervals and applied heuristics to indicate the presence of each DBMS in each slice. This information was processed to allow identification of which DBMSs were added, removed, or kept in each slice. Then, we identified replacement patterns that allowed us to determine whether some DBMSs were replaced by others. We also identified DBMSs that were used together in the project history.

We could observe that MySQL, H2, and PostgreSQL are among the three most used relational DBMSs, while Redis and Cassandra are the most used non-relational DBMSs. Projects that belong to the infrastructure management domain mostly use multi-model DBMSs.

The concomitant use of DBMSs grows as the projects mature. In the first slice of the project, we observed pair combinations of 11 DBMSs, prevailing combinations among relational DBMSs (e.g., PostgreSQL and MySQL in 27 projects, Oracle and MySQL in 20 projects, and H2 and MySQL in 19 projects). In the middle of the project history, we observed pair combinations of 25 DBMSs. Ten involved non-relational DBMSs (e.g., Redis and MySQL appear in 20 projects, MongoDB and PostgreSQL in 11 projects, and Elasticsearch and Redis in 10 projects). In the last slice of the project history, the number of DBMS pair combinations increased to 29, with MySQL, PostgreSQL, H2, and Oracle among the relational ones, but Oracle lost its position to Redis.

Additionally, we discovered 20 situations where DBMSs were replaced in projects. The most frequent replacement occurred in only three projects, with PostgreSQL being replaced by MariaDB and Oracle by MySQL. PostgreSQL was also the DBMS that underwent the most replacements: it was replaced eleven times in eight projects by four different DBMSs. MySQL was the DBMS that most replaced PostgreSQL. Furthermore, 70% of the replacements occur between DBMSs of the same model, of which only 10% is non-relational. We observed that, although DBMS replacements are not so frequent, they occur. We suppose the low frequency in the DBMS replacements may be directly related to the current tendency to use more than one DBMS together. According to Sahatqija et al. (2018), one possible explanation is that non-relational DBMSs were not created to substitute the relational ones but to complement them. This diminishes the need to migrate from one DBMS to another since two DBMSs of distinct models can be used concurrently.

Our findings offer strategic insights for organizations considering migration between DBMSs—understanding the prevalent frequencies and patterns of DBMS adoption can significantly inform such decisions. For professionals aspiring to specialize in the area, understanding how DBMSs are adopted can shape their educational and career paths. Educational institutions and training programs can utilize our results to refine their curriculum, emphasizing the most extensively adopted DBMSs and prevalent migration scenarios. Finally, our findings can guide projects seeking to enhance their integration and compatibility options, as well as DBMS tool developers, directing their efforts to broaden their potential user base.

5.1 Main Contributions

This research makes relevant contributions to the Software Engineering scientific community. These include:

- **The empirical research methodology**: The methods for slicing the history of projects, for defining the heuristics that indicate the use of DBMS in a given project, for automating the extraction of the results of the heuristics (scripts), for identifying the additions, keeping, and removals of DBMS, and for characterizing replaced patterns are part of the entire analysis process and can be applied to other related topics.

- **The identification of the most popular DBMS in Java Open Source Projects**: The fact that MySQL, H2, PostgreSQL, Redis, and Cassandra are the most common choices in open-source Java projects shows that software projects are increasingly moving towards non-relational models without abandoning relational ones.

- **Patterns of co-use**: The discovery of the co-use of DBMS throughout the projects life cycle reveals an increase in this use as projects mature. Thus, these patterns provide a new perspective on which DBMSs are used together for different needs in the same project.

- **DBMS replacement trends**: Our study reveals DBMS that replace and are replaced by others, albeit infrequently. However, this information is valuable for indicating the stability of DBMSs during the evolution of projects.

The results of our findings are reported in a paper that is currently under review in the Software: Practice and Experience journal.

5.2 Future Work

The DBMS adoption is a subject with a range of research possibilities that were not addressed in this work. Thus, we envision some interesting future work.

One of them is confirming whether two DBMSs are, in fact, used together by investigating which entities are stored in each DBMS. Using more than one DBMS in a given project may mean the project can work with several DBMSs, but the user chooses a single one at installation time. To make sure more than one DBMS is used, we would need a deeper analysis of the source code.

Another interesting study would be to find out why certain DBMSs are used together or substituted through qualitative analysis, for instance, by conducting interviews with the developers of the projects. Knowing the reasons underlying the concomitant use of DBMSs or their replacements can reveal valuable information for decision-making by development teams. An analysis of the application domains from the perspective of the DBMSs that are most commonly adopted together and the DBMSs that tend to be replaced would also be a relevant aspect to consider for future work.

This research can also be extended to projects developed in other programming languages to determine if there are differences in the most used DBMS and which DBMSs are most commonly used together in each language. Thus, language-specific preferences and practices would be revealed and contribute to a broader understanding of DBMS selection and use.

The historical study of the Object Relational Mapping (ORM) tools adoption together with the DBMSs adoption is another interesting perspective to consider. It would be possible to identify trends in the adoption of these tools in the context of different projects and their relationship with particular DBMSs, providing an understanding of common ways to interact with various DBMSs.

Historical analysis of vulnerabilities associated with the use of DBMSs in projects is another promising area. Understanding historical vulnerability trends can help identify failure patterns and correlations between DBMS choices, leading to best practices for selecting, configuring, and securing DBMSs in software projects. In this way, potential security problems can be avoided.

REFERENCES

AGARWAL, Shivam. Data Mining: Data Mining Concepts and Techniques. In: IEEE. 2013 International Conference on Machine Intelligence and Research Advancement. Katra, India: Institute of Electrical and Electronics Engineers, 2013. P. 203–207. DOI: 10.1109/ICMIRA.2013.45.

AGRAWAL, Rakesh; SRIKANT, Ramakrishnan, et al. Fast algorithms for mining association rules. In: SANTIAGO, CHILE. PROC. 20th int. conf. very large data bases, VLDB. [S.l.: s.n.], 1994. v. 1215, p. 487–499.

BERNSTEIN, Philip A.; HADZILACOS, Vassos; GOODMAN, Nathan. **Concurrency Control and Recovery in Database Systems**. [S.l.]: Addison-Wesley, 1987. ISBN 0-201-10715-5. Available from: <http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx>.

BORGES, Hudson; TULIO VALENTE, Marco. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. **Journal of Systems and Software**, Elsevier BV, v. 146, p. 112–129, Dec. 2018. ISSN 0164-1212. DOI: 10.1016/j.jss.2018.09.016.

CATTELL, Rick. Scalable SQL and NoSQL data stores. **Acm Sigmod Record**, ACM New York, NY, USA, v. 39, n. 4, p. 12–27, 2011.

CHEN, Peter Pin-Shan. The entity-relationship model—toward a unified view of data. **ACM transactions on database systems (TODS)**, Acm New York, NY, USA, v. 1, n. 1, p. 9–36, 1976. DOI: 10.1145/320434.320440.

CODD, E. F. A Relational Model of Data for Large Shared Data Banks. **Commununications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 13, n. 6, p. 377–387, June 1970. ISSN 0001-0782. DOI: 10.1145/362384.362685.

DATE, Cj. **An introduction to database systems**. [S.l.]: Pearson education india, 2006.

DAVOUDIAN, Ali; CHEN, Liu; LIU, Mengchi. A survey on NoSQL stores. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 51, n. 2, p. 1–43, 2018.

DEWITT, David; GRAY, Jim. Parallel database systems: The future of high performance database systems. **Communications of the ACM**, ACM New York, NY, USA, v. 35, n. 6, p. 85–98, 1992. DOI: 10.1145/129888.129894.

DIMOLIKAS, Konstantinos; ZARRAS, Apostolos V; VASSILIADIS, Panos. A study on the effect of a table's involvement in foreign keys to its schema evolution. In: SPRINGER. CONCEPTUAL Modeling: 39th International Conference, ER 2020, Vienna, Austria, November 3–6, 2020, Proceedings 39. [S.l.: s.n.], 2020. P. 456–470.

DB-ENGINES. **DB-Engines Ranking**. [S.l.: s.n.], 2022. https://db-engines.com/en/ranking. Accessed: 2022-02-28.

FOURNIER-VIGER, Philippe; LIN, Jerry Chun-Wei; GOMARIZ, Antonio; GUENICHE, Ted; SOLTANI, Azadeh; DENG, Zhihong; LAM, Hoang Thanh. The SPMF open-source data mining library version 2. In: SPRINGER. MACHINE Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part III 16. Cham: Springer International Publishing, 2016. P. 36–40.

FOURNIER-VIGER, Philippe; LIN, Jerry Chun-Wei; KIRAN, Rage Uday; KOH, Yun Sing; THOMAS, Rincy. A survey of sequential pattern mining. **Data Science and Pattern Recognition**, v. 1, n. 1, p. 54–77, 2017.

GESSERT, Felix; WINGERATH, Wolfram; FRIEDRICH, Steffen; RITTER, Norbert. NoSQL database systems: a survey and decision guidance. **Computer Science-Research and Development**, Springer, v. 32, p. 353–365, 2017.

GOEMINNE, Mathieu; DECAN, Alexandre; MENS, Tom. Co-evolving code-related and database-related changes in a data-intensive software system. In: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). Antwerp, Belgium: Institute of Electrical and Electronics Engineers, 2014. P. 353–357. DOI: 10.1109/CSMR-WCRE.2014.6747193.

GRAY, Jim. The transaction concept: Virtues and limitations. In: VLDB. [S.l.: s.n.], 1981. v. 81, p. 144–154.

HAN, Jiawei; PEI, Jian; MORTAZAVI-ASL, Behzad; PINTO, Helen; CHEN, Qiming; DAYAL, Umeshwar; HSU, Meichun. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: IEEE. PROCEEDINGS of the 17th international conference on data engineering. [S.l.: s.n.], 2001. P. 215–224.

HAN, Jiawei; PEI, Jian; TONG, Hanghang. **Data mining: concepts and techniques**. [S.l.]: Morgan kaufmann, 2022.

KALLIAMVAKOU, Eirini; GOUSIOS, Georgios; BLINCOE, Kelly; SINGER, Leif; GERMAN, Daniel M.; DAMIAN, Daniela. The Promises and Perils of Mining GitHub. In: PROCEEDINGS of the 11th Working Conference on Mining Software Repositories. Hyderabad, India: Association for Computing Machinery, 2014. (MSR 2014), p. 92–101. ISBN 9781450328630. DOI: 10.1145/2597073.2597074.

KLEPPMANN, M. **Designing Data-intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems**. [S.l.]: O'Reilly Media, 2017. ISBN 9781449373320. Available from: <https://books.google.com.br/books?id=BM7woQEACAAJ>.

PATNI, Jagdish Chandra; SHARMA, Hitesh Kumar; TOMAR, Ravi; KATAL, Avita. **Database Management System: An Evolutionary Approach**. [S.l.]: CRC Press, 2022.

QIU, Dong; LI, Bixin; SU, Zhendong. An empirical analysis of the co-evolution of schema and code in database applications. In: PROCEEDINGS of the 2013 9th Joint Meeting on Foundations of Software Engineering. Saint Petersburg, Russia: Association for Computing Machinery, 2013. (ESEC/FSE 2013), p. 125–135. ISBN 9781450322379. DOI: 10.1145/2491411.2491431.

RASCHKA, Sebastian. MLxtend: Providing machine learning and data science utilities and extensions to Python's scientific computing stack. **The Journal of Open Source Software**, The Open Journal, v. 3, n. 24, p. 638, Apr. 2018. DOI: 10.21105/joss.00638.

SADALAGE, PJ; FOWLER, M. **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**. [S.l.]: Addison-Wessley, 2013.

SAHATQIJA, Kosovare; AJDARI, Jaumin; ZENUNI, Xhemal; RAUFI, Bujar; ISMAILI, Florije. Comparison between relational and NOSQL databases. In: IEEE. 2018 41st international convention on information and communication technology, electronics and microelectronics (MIPRO). Opatija, Croatia: Institute of Electrical and Electronics Engineers, 2018. P. 0216–0221.

SCHERZINGER, Stefanie; SIDORTSCHUCK, Sebastian. An empirical study on the design and evolution of NoSQL database schemas. In: SPRINGER. CONCEPTUAL Modeling: 39th International Conference, ER 2020, Vienna, Austria, November 3–6, 2020, Proceedings 39. [S.l.: s.n.], 2020. P. 441–455.

SILBERSCHATZ, A; KORTH, HF; SUDARSHAN, S. **Database System Concepts, 7th edn, vol. 4**. [S.l.]: McGraw-Hill, New York, 2019.

STONEBRAKER, Michael; WONG, Eugene; KREPS, Peter; HELD, Gerald. The Design and Implementation of INGRES. **ACM Trans. Database Syst.**, v. 1, n. 3, p. 189–222, 1976. DOI: 10.1145/320473.320476. Available from: <https://doi.org/10.1145/320473.320476>.

TAN, Pang-Ning; STEINBACH, Michael; KUMAR, Vipin. **Introduction to data mining**. [S.l.]: Pearson Education India, 2016.

VASSILIADIS, Panos. Profiles of schema evolution in free open source software projects. In: IEEE. 2021 IEEE 37th International Conference on Data Engineering (ICDE). [S.l.: s.n.], 2021. P. 1–12.

WINSLETT, Marianne. Interview with Avi Silberschatz. In: 1. SIGMOD Record. [S.l.: s.n.], 2002. v. 31. Available from: <https://sigmodrecord.org/publications/sigmodRecord/0203/avi-revised-dbprofile.pdf>.

________. Interview with David Dewitt. In: 2. SIGMOD Record. [S.l.: s.n.], 2002. v. 31. Available from: <https://sigmodrecord.org/publications/sigmodRecord/0206/dewitt-interview.pdf>.

________. Interview with Jim Gray. In: 1. SIGMOD Record. [S.l.: s.n.], 2003. v. 32. Available from: <https://sigmodrecord.org/publications/sigmodRecord/0303/Gray_SIGMOD_Interview_Final.pdf>.

________. Interview with Michael Stonebraker. In: 2. SIGMOD Record. [S.l.: s.n.], 2003. v. 32. Available from: <https://sigmodrecord.org/publications/sigmodRecord/0306/D1-DBP-stonebraker-final.pdf>.

________. Interview with Peter Chen. In: 1. SIGMOD Record. [S.l.: s.n.], 2004. v. 33. Available from: <https://sigmodrecord.org/publications/sigmodRecord/0403/D.MW-chen-final.pdf>.

________. Interview with Phil Bernstein. In: 3. SIGMOD Record. [S.l.: s.n.], 2004. v. 33. Available from: <https://sigmodrecord.org/publications/sigmodRecord/0409/7.phil-bernstein-final.pdf>.

WITTEN, Ian H; FRANK, Eibe; HALL, Mark A; PAL, Christopher J; MINING, Data. Practical machine learning tools and techniques. In: MORGAN KAUFMANN. DATA Mining. [S.l.: s.n.], 2017. v. 4, p. 11–79.