Titre: Title:	Data-Access Technical Debt: Specification, Refactoring, and Impact Analysis
Auteur: Author:	Biruk Asmare Muse
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Muse, B. A. (2022). Data-Access Technical Debt: Specification, Refactoring, and Impact Analysis [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10720/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/10720/
Directeurs de recherche: Advisors:	Foutse Khomh, & Giuliano Antoniol
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Data-Access Technical Debt: Specification, Refactoring, and Impact Analysis

BIRUK ASMARE MUSE
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Décembre 2022

© Biruk Asmare Muse, 2022.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Data-Access Technical Debt: Specification, Refactoring, and Impact Analysis

présentée par **Biruk Asmare MUSE**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Michel DESMARAIS, président
Foutse KHOMH, membre et directeur de recherche
Giuliano ANTONIOL, membre et codirecteur de recherche
Maxime LAMOTHE, membre
Nikolaos TSANTALIS, membre externe

DEDICATION

To my wife
To my daughter
To my family
For their endless love, support, and encouragement

. . .

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to **GOD** and our lady **Virgin Mary** for the blessings and support in every aspect of my life. Next, I would like to express my sincere thanks and my deepest gratitude to my supervisors, Professor **Foutse Khomh** and Professor **Giuliano Antoniol** for their economical, emotional support, encouragement and for their rigorous feedback during the long journey of this research.

I would like to thank the members of my Ph.D. committee, Professor **Maxime Lamothe**, Professor **Nikolaos Tsantalis**, and Professor **Michel Desmarais**, who accepted to review my dissertation with enthusiasm. I would like to thank all the reviewers participated in evaluating my research publications for their invaluable feedback and suggestions to improve my work.

I would also like to give my gratitude to colleagues and friends who collaborated with me in the studies, some of them are co-authors while others participate anonymously. My Sincere gratitude also goes to the developers and practitioners who anonymously provided their feedback during the surveys in this research.

Last but not definitely the least, I would like to express my deepest gratitude to my wife **Ageritu Kassa** who were there for me during the good and bad times and for her love, technical and emotional support. I would not be where I am without her continuous encouragement and understanding. I would like to thank my daughter **Hildana** and all my family. I would not be able to achieve any of my dreams without their love and support.

RÉSUMÉ

L'exploitation de la grande quantité de données hétérogènes générées par les humains et les machines pour obtenir des informations exploitables devient le centre d'attention de l'industrie et de la communauté de la recherche. Les décisions commerciales et les politiques de gouvernance sont guidées par les informations et les recommandations obtenues à partir de l'analyse de ces mégadonnées. En raison de la taille, de l'hétérogénéité et de la complexité, la gestion de ces données avec des logiciels et des applications traditionnels devient plus difficile. En conséquence, des systèmes très consommateurs en données tirant parti de la disponibilité des infrastructures infonuagiques ont été introduits pour relever ce défi. Le développement de systèmes à forte intensité de données implique l'intégration de cadres de stockage, de traitement et de présentation de données. Code d'accès aux données, mettant en œuvre des interactions directes avec des bases de données ou d'autres systèmes de persistance via des appels aux fonctions de pilote ou aux API, joue un rôle central dans les systèmes à forte intensité de données en reliant les composants de traitement et de présentation aux composants de stockage de données. Le développement de systèmes à forte intensité de données pose plusieurs défis lors de la conception, de la mise en œuvre et de l'assurance qualité. Par conséquent, les développeurs de systèmes à forte intensité de données, comme les systèmes logiciels traditionnels, pourraient intentionnellement ou non introduire des dettes techniques en raison de la pression de déploiement habituelle sur les développeurs et se concentrer sur la satisfaction des exigences fonctionnelles. Les dettes techniques sont des raccourcis de conception et de mise en œuvre pour répondre rapidement aux besoins actuels mais compromettent la qualité du logiciel à long terme. En plus des dettes techniques courantes dans les systèmes logiciels traditionnels, les systèmes à forte intensité de données pourraient être sujets à des dettes techniques d'accès aux données compromettant la qualité des opérations d'accès aux données.

Bien que la caractérisation et l'analyse d'impact des dettes techniques traditionnelles soient bien étudiées, peu l'attention a été portée sur les dettes techniques d'accès aux données. Compte tenu de l'importance cruciale du code d'accès aux données pour les systèmes à forte intensité de données, nous pensons que les résultats de la spécification, de la caractérisation et de l'analyse d'impact des dettes techniques d'accès aux données contribueront de manière significative à l'amélioration de la qualité des systèmes à forte intensité de données.

Dans ce thèse, nous visons à soutenir l'assurance qualité des systèmes à forte intensité de données en (1) spécifiant les dettes techniques d'accès aux données ; (2) en caractérisant les

dettes techniques d'accès aux données en termes de prévalence, d'évolution et de contexte; (3) en enquêtant sur les impacts des dettes techniques d'accès aux données sur la qualité des logiciels; et (4) en enquêtant sur les pratiques de refactoring dans les systèmes à forte intensité de données pour comprendre si/comment la dette technique d'accès aux données est résolue par la refactorisation. Nous avons étendu la taxonomie des dettes techniques auto-admises (SATD) en identifiant de nouveaux types liés à l'accès aux données. Nous avons également spécifié de nouveaux anti-modèles de performances d'accès aux données dans les systèmes NoSQL et polyglottes gourmands en données en analysant les problèmes de performances signalés. Pour caractériser les SATD d'accès aux données, nous avons mené une étude empirique sur la prévalence et l'évolution des SATD et les circonstances de leur introduction et de leur suppression. Nous avons également mené une étude quantitative de la prévalence, de la cooccurrence et de l'évolution des odeurs de code SQL qui sont un type de dettes techniques d'accès aux données qui ne sont pas admises par les développeurs. Nous avons également étudié les impacts des odeurs de code SQL sur l'introduction de bogues et mené une enquête auprès des développeurs pour comprendre la criticité des odeurs de code SQL et les anti-modèles de performances d'accès aux données du point de vue du développeur. Les refactorings étant des remèdes aux dettes techniques, nous avons mené une analyse quantitative et qualitative des refactorings pour investiguer la prévalence, l'évolution et le contexte des pratiques de refactoring. De plus, nous avons mené une enquête auprès des développeurs pour trianguler nos résultats d'analyse avec l'expérience des praticiens.

Dans l'ensemble, nos résultats montrent que les dettes techniques d'accès aux données sont répandues, persistantes et ont un impact sur la qualité des logiciels. Les dettes techniques d'accès aux données ne sont généralement pas traitées lors des opérations de refactoring. Nous pensons que nos résultats constituent une première étape importante vers l'étude des dettes techniques d'accès aux données et de leur impact sur la qualité des systèmes à forte intensité de données. Nous avons fourni plusieurs recommandations à la communauté des chercheurs et aux praticiens sur la base de nos conclusions qui peuvent être exploitées lors de la conception, de la mise en œuvre et de l'assurance qualité des systèmes à forte intensité de données.

ABSTRACT

Leveraging the vast amount of heterogeneous data generated by humans and machines to obtain actionable insights is becoming the center of attention in industry and the research community. Business decisions and governance policies are driven by the insights and recommendations obtained from analyzing this big data. Due to the size, heterogeneity, and complexity, handling this data with traditional software and applications is becoming more challenging. As a result, data-intensive software systems leveraging the availability of cloud infrastructures were introduced to address this challenge. The development of data-intensive systems involves the integration of data storage, processing, and presentation frameworks. Data-access code, implementing direct interactions with databases or other persistence systems via calls to driver functions or APIs, plays a pivotal role in data-intensive systems by linking processing and presentation components with data-storage components. The development of data-intensive systems poses several challenges during design, implementation, and quality assurance. Hence, developers of data-intensive systems, like traditional software systems, could intentionally or unintentionally introduce technical debts due to the usual release pressure on developers and focus on addressing the functional requirements. Technical debts are design and implementation shortcuts to quickly address current requirements, but they compromise software quality in the long run. In addition to the technical debts prevalent in traditional software systems, data-intensive systems could be prone to data-access technical debts, compromising the quality of data-access operations.

While the characterization and impact analysis of traditional technical debts are well investigated, not much attention was given to data-access technical debts. Considering the critical importance of data-access code to data-intensive systems, we believe that the findings from specification, characterization, and impact analysis of data-access technical debts will have a significant contribution towards improving the quality of data-intensive systems.

In this dissertation, we aim to support the quality assurance of data-intensive systems by (1) specifying data-access technical debts; (2) characterizing data-access technical debts in terms of their prevalence, evolution, and context; (3) Investigating the impacts of data-access technical debts on software quality; and (4) investigating refactoring practices in data-intensive systems to understand if/how data-access technical debt is resolved by refactoring.

We extended the taxonomy of self-admitted technical debts (SATDs) by identifying new types of SATDs related to data-access. We also specified new data-access performance anti-patterns in NoSQL-based and polyglot data-intensive systems by analyzing the reported

performance issues. To characterize data-access SATDs, we conducted an empirical study on the prevalence and evolution of SATDs and the circumstances behind their introduction and removal. We also conducted a quantitative study of the prevalence, co-occurrence, and evolution of SQL code smells, which are one kind of data-access technical debts that are not admitted by developers. We also investigated the impacts of SQL code smells on introducing bugs and conducted a developer survey to understand the criticality of SQL code smells and data-access performance anti-patterns from the developer's point of view. Since refactorings are remedies to technical debts, we conducted a quantitative and qualitative analysis of refactorings to investigate the prevalence, evolution, and context of refactoring practices. Moreover, we conducted a developer survey to triangulate our analysis findings with the experience of practitioners.

Overall, our results show that data-access technical debts are prevalent, persistent, and impact software quality. Data-access technical debts are generally not addressed during refactoring operations. We believe that our findings are an important first step toward the study of data-access technical debts and their impact on the quality of data-intensive systems. We provided several recommendations to the research community and to practitioners based on our findings that can be leveraged during the design, implementation, and quality assurance of data-intensive systems.

Keywords: Data-intensive systems, data-access classes, empirical study, technical debt, self-admitted technical debt, code smells, SQL code smells, performance anti-patterns, refactoring

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE OF CONTENTS ix

LIST OF TABLES xvi

LIST OF FIGURES xix

LIST OF SYMBOLS AND ACRONYMS xxii

CHAPTER 1 INTRODUCTION 1

1.1 Problem statement 6
1.2 Thesis statement 7
1.3 Research objectives 8
1.4 Research contributions 12
1.5 Articles related to the dissertation 13
1.6 Dissertation organization 14

CHAPTER 2 BACKGROUND 15

2.1 Chapter overview 15
2.1.1 SQL code smells 15
2.2 Survival analysis 17
2.3 Topic modelling 18
2.4 Apriori algorithm 19
2.5 Cramer's V test of association 21
2.6 Chapter summary 21

CHAPTER 3 LITERATURE REVIEW 22

3.1 Chapter overview 22
3.2 Specification of technical debt 22

3.2.1 Traditional code smells 22
3.2.2 Self-admitted technical debts (SATD) 23
3.2.3 Multi-language design smells 23
3.2.4 Deep learning design smells 23
3.2.5 Video game smells 24
3.2.6 Specification of data-access technical debts 24
3.2.7 Specification of SQL code smells 25
3.2.8 Specification of data-access performance anti-patterns 25
3.3 Technical debt detection and refactoring detection approaches 26
3.3.1 Traditional code smell detection and refactoring approach's 26
3.3.2 SATD detection and removal approaches 28
3.3.3 SQL code smell detection 30
3.3.4 Refactoring detection approaches 31
3.4 Characterization of technical debts 32
3.4.1 Prevalence and evolution of SATDs 34
3.4.2 Prevalence and evolution of SQL code smells 35
3.5 Impacts of technical debts on software quality 36
3.5.1 Impacts of traditional code smells on software quality 36
3.5.2 Impacts of SATDs on software quality 38
3.6 Refactoring practices in traditional software systems 38
3.7 Chapter summary 40

CHAPTER 4 STUDY DESIGN 41
4.1 Chapter overview 41
4.2 Characterization and impact analysis of SQL code smells 41
4.2.1 Selection of subject systems 41
4.2.2 Code smell detection 43
4.2.3 Tracking project file evolution 44
4.2.4 Mining Bug-fix and Bug-inducing commits 44
4.2.5 Linking Bug-inducing commits with code smells 45
4.2.6 Construction of a smell dataset 45
4.3 Specification and characterization of data-access SATD 46
4.3.1 Subject systems 46
4.3.2 Tracking source file genealogy 47
4.3.3 SATD detection 47
4.3.4 Identifying Data-Access SATD 48

4.3.5 SATD dataset construction 48
4.4 Specification and criticality analysis of data-access performance anti-patterns 48
4.4.1 Subject systems 49
4.4.2 Filter non-English repositories 50
4.4.3 Select repositories with the highest number of issues 50
4.4.4 Manually filter out irrelevant repositories 51
4.4.5 Collect issues 51
4.4.6 Filter data-access performance issues 51
4.4.7 Data-access performance issues dataset 52
4.4.8 Survey on data-access performance anti-patterns 52
4.5 Refactoring practices in data-intensive systems 54
4.5.1 Subject systems 54
4.5.2 Extracting list of revisions 57
4.5.3 Extracting commit information 57
4.5.4 Detecting refactoring 57
4.5.5 Construction of the refactoring dataset 58
4.5.6 Identifying data access refactoring instances 58
4.5.7 Linking refactoring dataset with commit information 59
4.5.8 Detecting SQL query and smell 59
4.5.9 Linking refactoring dataset with SQL query and smell dataset 59
4.5.10 Developer survey on refactoring practices 59
4.6 Chapter summary 64

CHAPTER 5 SPECIFICATION OF DATA-ACCESS TECHNICAL DEBTS 65
5.1 Chapter overview 65
5.2 RQ 1.1: Composition of data-access SATD 65
5.2.1 Analysis approach 65
5.2.2 Taxonomy of data-access SATDs 67
5.3 RQ 1.2: Specification of data-access performance anti-patterns 74
5.3.1 Analysis approach 74
5.3.2 Data-access performance anti-patterns 75
5.4 Discussion 83
5.5 Threats to validity 84
5.6 Chapter summary 85

CHAPTER 6 CHARACTERIZATION OF DATA-ACCESS SATDS 86
6.1 Chapter overview 86

6.2 RQ 2.1 : Prevalence of SATDs in data-intensive systems 87
6.2.1 Analysis approach 87
6.2.2 Findings 87
6.3 RQ 2.2: Persistence of SATDs in data-intensive systems 92
6.3.1 Analysis approach 92
6.3.2 Findings 93
6.4 RQ 2.3: Circumstances behind the introduction and removal of data-access SATD 96
6.4.1 Analysis approach 97
6.4.2 Findings 98
6.5 Discussion 103
6.6 Threats to validity 105
6.6.1 Threats to construct validity 105
6.6.2 Threats to internal validity 106
6.6.3 Threats to conclusion validity 106
6.6.4 Threats to external validity 106
6.6.5 Threats to reliability validity 106
6.7 Chapter summary 106

CHAPTER 7 CHARACTERIZATION OF SQL CODE SMELLS 108
7.1 Chapter overview 108
7.2 RQ 2.4: Prevalence of SQL code smells 108
7.2.1 Analysis approach 109
7.2.2 Findings 109
7.3 RQ 2.5: Co-occurrence of traditional code smells and SQL code smells 112
7.3.1 Analysis approach 112
7.3.2 Findings 112
7.4 RQ 2.6: Survival analysis of SQL code smells 114
7.4.1 Analysis approach 115
7.4.2 Findings 115
7.5 Discussion 118
7.6 Threats to validity 118
7.6.1 Threats to construct validity 118
7.6.2 Threats to conclusion validity 119
7.6.3 Threats to external validity 119
7.6.4 Threats to reliability validity 119

7.7 Chapter summary 119

CHAPTER 8 IMPACT OF DATA-ACCESS TECHNICAL DEBTS ON SOFTWARE QUALITY 121
8.1 Chapter overview 121
8.2 RQ 3.1: Co-occurrence of SQL code smells with bugs 122
8.2.1 Analysis approach 122
8.2.2 Findings 123
8.3 RQ 3.2: Perceived criticality of SQL code smells 124
8.3.1 Analysis approach 124
8.3.2 Findings 124
8.4 RQ 3.3: Perceived criticality of data-access performance anti-patterns 126
8.4.1 Analysis approach 126
8.4.2 Findings 127
8.5 Discussion 136
8.6 Threats to validity 137
8.6.1 Threats to construct validity 137
8.6.2 Threats to internal validity 138
8.6.3 Threats to conclusion validity 138
8.6.4 Threats to external validity 138
8.6.5 Threats to reliability validity 139
8.7 Chapter summary 139

CHAPTER 9 QUANTITATIVE ANALYSIS OF DATA-ACCESS REFACTORINGS 140
9.1 Chapter overview 140
9.2 RQ 4.1: Prevalence of refactorings in data-access classes 141
9.2.1 Analysis approach 142
9.2.2 Findings 142
9.3 RQ 4.2: Evolution of refactorings 147
9.3.1 Analysis approach 147
9.3.2 Findings 148
9.4 RQ 4.3: Data-access refactoring activities and SQL code smells 153
9.4.1 Analysis approach 153
9.4.2 Findings 153
9.5 RQ 4.4: Co-occurrence of refactorings in data-access classes 155
9.5.1 Analysis approach 155
9.5.2 Findings 156

9.6 RQ 4.5: Profile of developers performing data-access refactorings 159
9.6.1 Analysis approach 159
9.6.2 Findings 161
9.7 Discussion 169
9.8 Threats to validity 170
9.8.1 Threats to construct validity 170
9.8.2 Threats to internal validity 170
9.8.3 Threats to conclusion validity 171
9.8.4 Threats to external validity 171
9.8.5 Threats to reliability validity 171
9.9 Chapter summary 171

CHAPTER 10 QUALITATIVE ANALYSIS OF DATA-ACCESS REFACTORINGS 173
10.1 Chapter overview 173
10.2 RQ 4.6: Data-access class code elements prone to refactorings 174
10.2.1 Analysis approach 174
10.2.2 Findings 174
10.3 RQ 4.7: Context of data-access refactorings 178
10.3.1 Analysis approach 179
10.3.2 Findings 180
10.4 RQ 4.8: Developers' opinion about refactoring practices in data-access classes 180
10.4.1 Analysis approach 181
10.4.2 Findings 182
10.5 Discussion 187
10.6 Threats to validity 188
10.6.1 Threats to construct validity 188
10.6.2 Threats to internal validity 188
10.6.3 Threats to conclusion validity 189
10.6.4 Threats to external validity 189
10.6.5 Threats to reliability validity 190
10.7 Chapter summary 190

CHAPTER 11 CONCLUSION 191
11.1 Summary of the study findings 191
11.2 Implication of the findings 196
11.3 Future research opportunities 198

REFERENCES 200

LIST OF TABLES

Table 4.1 Selected projects and their database access statistics. **DAQC** = Database Access Query Count . . . 43
Table 4.2 Most prevalent keywords used to detect bug-fix commits . . . 45
Table 4.3 Distribution of repository level metrics for NoSQL and Polyglot subject systems . . . 51
Table 4.4 List of SQL subject systems with a number of commits, number of queries, and number of data access files and number of refactoring instances . . . 56
Table 4.5 List of NoSQL subject systems with a number of commits, number of data access files and number of refactoring instances. . . . 56
Table 5.1 Distribution of categories in the manually classified dataset . . . 73
Table 6.1 Project groups . . . 88
Table 6.2 Summary of the distribution of data-access and regular SATDs over the number of commits in Group 1 subject systems . . . 89
Table 6.3 Summary of the distribution of data-access and regular SATDs over the number of commits in Group 2 subject systems . . . 89
Table 6.4 Summary of the distribution of data-access and regular SATDs over the number of commits in Group 3 SQL subject systems . . . 89
Table 6.5 Data-access SATD introduction time for SATD categories . . . 100
Table 6.6 Distribution of data-access SATD removal time among the data-access categories . . . 101
Table 6.7 Data-access introducing commit goals in NoSQL and SQL subject systems . . . 102
Table 6.8 Data-access SATD introducing commit goals grouped by data-access SATD categories . . . 102
Table 6.9 Data-access SATD removing commit goals grouped by data-access SATD categories . . . 103
Table 6.10 Data-access SATD removing commit goals for SQL and NoSQL subject systems . . . 103
Table 7.1 Prevalence of Implicit Columns across four application domains . . . 110
Table 7.2 Source code file versions with database access . . . 113

Table 7.3 Top-3 SQL code smells, and traditional code smells based on lift value across the application domains. A leverage value close to 0 indicates weak association. 113

Table 7.4 Chi-square and Cramer's V value of smell pairs computed on the combined dataset for each smell pair in Table 7.3. We reject H_0 for all smell pairs in bold. 114

Table 8.1 Result of statistical tests and random forest model of association between smells and buggy files. 124

Table 9.1 Top ten most prevalent refactoring types. The table shows the number of refactoring instances (count) and percentage against the total number of data-access and regular class refactoring instances. 145

Table 9.2 Distribution of *Relative Commit Time* for the top ten prevalent data-access refactoring types. 150

Table 9.3 Distribution of distance from release for the top ten prevalent data-access refactoring type 152

Table 9.4 Top ten co-occurrence of data-access refactorings ranked by the support and associated co-occurrence metrics 156

Table 9.5 Result of Chi-squared test and cramer's V test for the most co-occurring refactoring types in data-access classes 157

Table 9.6 Apriori algorithm result for top ten regular refactoring types and corresponding statistical test 158

Table 9.7 Median feature importance values in percent and Logistic regression coefficient of developers' profile metrics 162

Table 9.8 Comparison of developer profile metrics between data-access refactoring developers and regular refactoring developers. We reject the null hypothesis, with a large effect size for the bolded metrics. The Cliffs Delta value is also bolded for metrics with a large effect size 162

Table 9.9 Summary of developers' contributions in the subject systems. It shows the number of developers and the summary of the percentage contributions of all developers for each subject system 165

Table 9.10 Summary of the contribution of developers in data-access refactoring. The table shows the number of developers involved, the percentage against total number of developers, and the summary of the distribution of their contribution in percentage 166

Table 9.11 Summary of the contribution of developers in regular refactoring. The table shows the number of developers involved, the percentage against the total number of developers, and the summary of the distribution of their contribution in percentage 167

Table 9.12 Percentage of data-access and regular refactoring performed by the main contributor of the involved classes. The subject systems are sorted in alphabetical order. 168

Table 10.1 Most prevalent refactoring types for Fetch data and Insert data functionalities. 178

LIST OF FIGURES

Figure 4.1 Overview of the study method for characterization and impact analysis of SQL code smells . . . 42

Figure 4.2 Overview of the study method for Specification and characterization of data-access SATDs. . . . 46

Figure 4.3 Overview of the study method for specification and criticality analysis of data-access performance anti-patterns . . . 49

Figure 4.4 Example survey question regarding *Duplicate requests* anti-pattern. . . 54

Figure 4.5 Overview of the study method for refactoring practices in data-intensive systems . . . 55

Figure 5.1 SATD classification hierarchy extended from Bavota and Russo [1]. White boxes are newly added categories to existing categories (gray boxes). Boxes marked with a database icon (🛢) are categories closely related to database accesses. . . . 69

Figure 5.2 Catalog of data-access performance anti-patterns prevalent in the analyzed data-access performance issues. . . . 76

Figure 6.1 Distribution of the number of commits in SQL and NoSQL subject systems. The y-axis is on a log scale. . . . 88

Figure 6.2 Prevalence of regular and data-access SATD in $Group_1$. The horizontal lines in this and subsequent violin plots show the 25%, median, and 75% quantiles respectively from bottom to top. . . . 90

Figure 6.3 Prevalence of regular and data-access SATD in $Group_2$. . . . 91

Figure 6.4 Prevalence of regular and data-access SATD in $Group_3$. . . . 92

Figure 6.5 The distribution of average time interval between successive snapshots taken every 500 commits for SQL and NoSQL subject systems. The y-axis time unit is in days. . . . 93

Figure 6.6 Kaplan–Meier survival curve for data-access SATDs in SQL subject systems. The x-axis is the number of commits. The censoring time and confidence intervals are marked on the plot. The Logrank test's p-value is indicated. . . . 94

Figure 6.7 Kaplan–Meier survival curve for data-access SATDs in NoSQL subject systems. The x-axis represents the number of commits. The censoring time and confidence interval are marked on the plot. The Logrank test's p-value is indicated. . . . 95

Figure 6.8 Kaplan–Meier survival curve for SQL subject systems by grouping them into data-access and regular SATD comments. The x-axis represents the number of commits. The censoring time is marked on the plot. . . . 95
Figure 6.9 Kaplan–Meier survival curve for NoSQL subject systems by grouping them into data-access and regular SATD comments. The x-axis represents the number of commits. The censoring time is marked on the plot. . . . 96
Figure 6.10 Distribution of data-access SATD introduction time . . . 99
Figure 6.11 Distribution of data-access SATD introduction time in SQL and NoSQL subject systems . . . 99
Figure 6.12 Distribution of data-access SATD removal time in SQL and NoSQL subject systems . . . 101
Figure 7.1 Prevalence of SQL code smells (Implicit Columns) across different application domains . . . 110
Figure 7.2 Prevalence of SQL code smells (Fear of the Unknown) across different application domains . . . 111
Figure 7.3 Kaplan-Meier survival curve for *Implicit Columns* SQL code smell. The X-axis is the time in days, and the vertical axis shows the survival probability value. The Censoring time and the Confidence interval are marked in the plot. . . . 116
Figure 7.4 Kaplan-Meier survival curve for *Fear of the Unknown* SQL code smell. The X-axis is the time in days, and the vertical axis shows the survival probability value. The Censoring time and the Confidence interval are marked in the plot. . . . 116
Figure 7.5 Kaplan-Meier survival curve for traditional code smells and SQL code smells. The Censoring time for censored files is marked in the plot. . . . 117
Figure 8.1 Criticality rating of *Sequential lookup of multiple keys* performance anti-pattern. The number of respondents (38) is indicated next to the anti-pattern name. . . . 128
Figure 8.2 Criticality rating of performance anti-patterns under database connection category. . . . 129
Figure 8.3 Criticality rating of performance anti-patterns under database driver or API access anti-pattern category. . . . 131
Figure 8.4 Criticality rating of performance anti-patterns under caching category. 132

Figure 8.5 Criticality rating of *Non-optimal indexing logic* data-access performance anti-pattern. . . . 133
Figure 8.6 Criticality rating of data node configuration and management anti-patterns 134
Figure 8.7 Criticality rating of *Inefficient query translation* data-access performance anti-pattern 135
Figure 9.1 Violin plot of distribution of code size between data-access classes and regular classes. 143
Figure 9.2 Violin Plot of the distribution of refactoring density in data-access classes and regular classes 146
Figure 9.3 Violin Plot of the distribution of *Relative Commit Time* in data-access refactorings (DAC) and regular refactorings 148
Figure 9.4 Violin Plot of the distribution of *distance from release* in data-access refactorings (DAC) and regular refactorings 151
Figure 10.1 Functionalities of code artifacts associated with refactoring in data-access classes. The sub-categories are ordered from the most prevalent to the least prevalent. 176
Figure 10.2 Bar plot showing the proportion of each label in the analyzed samples in percentage. 181
Figure 10.3 Survey respondent's opinion on refactoring frequency and release time. Total agreement is obtained by summing the *strongly agree* and *agree* responses. Similarly, total disagreement is obtained by summing *strongly disagree* and *disagree*. 184
Figure 10.4 Respondents' opinion on the context during which data-access refactoring is applied. Total agreement is obtained by summing *strongly agree* response and *agree* response. Similarly, total disagreement is obtained by summing *strongly disagree* and *disagree*. 185
Figure 10.5 Motivations behind data-access refactoring with the number and proportion of endorsement by the survey respondents. 186

LIST OF SYMBOLS AND ACRONYMS

SATD	Self-admitted technical debt
Non-SATD	Non-self-admitted technical debt
DAC	data-access classes
NDC	non-data-access classes
JNI	Java Native Interface

CHAPTER 1 INTRODUCTION

The advancement in storage technology, data communication, and IOT resulted in the generation of big data that is growing at exponential rates. The Internet reached more than 63% of the world population in 2022. Moreover, over 93% of Internet users use social media generating huge amounts of text and multimedia data[1]. Processing such data is an integral part of intelligent business solutions. Companies also generate a huge amount of data as a result of business transactions and manufacturing processes. It is estimated that 181 Zettabytes of data need to be analyzed in 2025. Furthermore, the global market in business analytics services is currently worth $274 billion and projected to generate $103 billion revenue by 2027 [2]. Processing this big data is beyond the capability of traditional software systems. This created an opportunity for the development of data-intensive systems.

Data-intensive systems

Data-intensive systems are software systems that devote a large component of their functionality to collecting, storing, analyzing, and visualizing high-volume, high-velocity, and high-variety data. Data-intensive systems integrate *software systems* and *data storage systems* [2]. The software systems handle data manipulation, transformation, data generation, analysis, and learning. The software systems are powered by big data analytics engines such as Spark. Data-storage systems are responsible for persisting data and metadata during data ingestion, manipulation, or reporting. Data storage systems are realized using relational, NoSQL-based databases and distributed file systems. The software systems interact with the data storage systems using **data-access classes**. Data-access classes are responsible for direct interaction with databases and other persistence systems through their drivers. They also provide a read/write interface with the storage systems. Data-access classes contain SQL queries or API calls to relational mapping frameworks in SQL-based database systems or interact with the associated read/write API of NoSQL databases. Data-access classes are critical components of data-intensive systems as their performance and robustness affect the performance of data-intensive systems.

Data-intensive systems that handle structured data utilize relational databases as data storage. SQL is the dominant data access language in relational databases. Hence, we refer to data-intensive systems that use relational databases as *SQL-based data-intensive systems*.

[1] https://www.domo.com/data-never-sleeps
[2] https://explodingtopics.com/blog/big-data-stats

Relational databases aim to provide strong data integrity by enforcing ACID (Atomicity, Consistency, Isolation, Durability) transactions. However, most relational databases may not be ideal solutions for handling ever-increasing data in amount and variation.

To exploit the scalability offered by current cloud solutions, NoSQL databases are getting increasing attention. One of the principles behind NoSQL databases is sacrificing consistency for availability and partition tolerance by providing BASE (Basically Available Soft state and Eventual consistency) transactions. NoSQL databases can be used to store structured or unstructured data. We refer to data-intensive systems that rely on NoSQL databases as *NoSQL-based data-intensive systems*.

Some data-intensive systems deploy both SQL and NoSQL databases as persistence layers. We refer to them as polyglot data-intensive systems. One of the reasons behind deploying database systems from different paradigms is to combine the advantages obtained from both SQL and NoSQL databases. For example, NextCloud combines different SQL databases with Redis (NoSQL) for caching. However, combining multiple database systems has its costs in terms of the complexity of the software systems. Also, the intention to improve the performance of one database could affect the performance of another.

The development of data-intensive systems typically requires the integration of a multitude of specialized frameworks for data storage (e.g., relational or NoSQL databases), processing (e.g., Hadoop, Spark), and learning (e.g., TensorFlow, Scikit-learn) which poses several design, implementation, and quality assurance challenges [2–4]. Developing data-intensive systems also often faces the usual release pressures that force programmers to compromise software quality, introducing technical debt [2] in data-intensive systems.

Technical Debt

Technical debts in the software development process represent the design and implementation of short-term solutions that easily address required functionality but introduce problems in software quality in the long run if they are not fixed [5]. Technical debts can occur in all stages of the software development cycle including requirement analysis, architecture, design, implementation, testing, building, documentation, deployment, and post-deployment stages [6]. Technical debts also affect software quality attributes such as reliability, efficiency, and flexibility. Sometimes developers admit technical debts by mentioning them in source code comments. Such type of technical debt is called Self-admitted technical debt (SATD) [7].

Some technical debts may not be necessarily known or admitted by developers in the source code or documentation artifacts refereed in this work as **non-self-admitted technical**

debts (Non-SATDs). Non-SATDs can occur at the design level or code level of software systems. Non-SATDs observed at the design level are referred to as **anti-patterns**. On the other hand, non-SATDs that are observed at the code or implementation level are referred to as **code smells**. Code smells are indicators of the underlying poor design choice. Examples of those smells include *Complex class, Long method, Long parameter list, Lazy class, and so on.* The detection, refactoring as well as the impacts of such anti-patterns and code smells are well-studied [8–11]. To distinguish those with smells that are related to data access, we refer to them as *Traditional smells and anti-patterns.*

Data-access technical debt

In this dissertation, we are interested in special kinds of technical debts that occur in the design and implementation of data-access classes referred to as **data-access technical debt**. Like traditional technical debts, data-access technical debts could be self-admitted (called **data-access SATDs**) and non-self-admitted (called **non-self-admitted data-access technical debts**). One example of data-access SATD is shown in Listing 1.1. The SATD is extracted from *Carbon-apimgt*[3] and notes a pending task to filter results by the status of the APIs. The query marked with the todo comment returns unnecessary records when only a specific API context is needed.

[3]Carbon-apimgt, `https://bit.ly/2NvDZvQ`

```
public ArrayList<URITemplate> getAllURITemplatesOldThrottle(String apiContext, String version)
    throws APIManagementException {
      Connection connection = null;
      PreparedStatement prepStmt = null;
      ResultSet rs = null;
      ArrayList<URITemplate> uriTemplates = new ArrayList<URITemplate>();

      //TODO : FILTER RESULTS ONLY FOR ACTIVE APIs
      String query = SQLConstants.GET_ALL_URL_TEMPLATES_SQL;
      try {
          connection = APIMgtDBUtil.getConnection();
          prepStmt = connection.prepareStatement(query);
          prepStmt.setString(1, apiContext);
          prepStmt.setString(2, version);

          rs = prepStmt.executeQuery();

          URITemplate uriTemplate;
          while (rs.next()) {
              uriTemplate = new URITemplate();
              String script = null;
              uriTemplate.setHTTPVerb(rs.getString("HTTP_METHOD"));
              uriTemplate.setAuthType(rs.getString("AUTH_SCHEME"));
              uriTemplate.setUriTemplate(rs.getString("URL_PATTERN"));
              uriTemplate.setThrottlingTier(rs.getString("THROTTLING_TIER"));
              InputStream mediationScriptBlob = rs.getBinaryStream("MEDIATION_SCRIPT");
              if (mediationScriptBlob != null) {
                  script = APIMgtDBUtil.getStringFromInputStream(mediationScriptBlob);
              }
              uriTemplate.setMediationScript(script);
              uriTemplate.getThrottlingConditions().add("_default");
              uriTemplates.add(uriTemplate);
          }
      } catch (SQLException e) {
          handleException("Error while fetching all URL Templates", e);
      } finally {
          APIMgtDBUtil.closeAllConnections(prepStmt, connection, rs);
      }
      return uriTemplates;
  }
```

Listing 1.1 Instance of data-access SATD

Non-self-admitted data-access technical debts that occur at the design level are called **data-access anti-patterns**. For example, the *Missing index* anti-pattern is a data-access performance anti-pattern that occurs when necessary database indexes are not defined. This anti-pattern degrades read performance [12].

Non-self-admitted data-access technical debt that occurs at the implementation level is referred to as **data-access smell**. For data-access classes that use SQL to interact with relational or SQL-based databases, data-access smells are specifically called **SQL code smells**. One example of SQL code smell is *implicit columns* smell. Listing 1.2 shows an example data access code excerpt from *WordPress-mobile* project[4]. This code contains two instances of *Implicit columns* smell in its data access class, i.e., the SELECT statements do not explicitly mention column names. This forces the database to return all columns, including columns that are not needed. Such data access causes performance problems for large-size tables and creates unnecessary coupling between application code and database. That means, if the order of columns is changed in the database, the corresponding data access logic must be updated to reflect the change.

```
public static Cursor getQueryStringCursor(String filter, int max) {
        String sql;
        String[] args;
        if (TextUtils.isEmpty(filter)) {
            sql = "SELECT * FROM tbl_search_suggestions";
            args = null;
        } else {
            sql = "SELECT * FROM tbl_search_suggestions WHERE query_string LIKE ?";
            args = new String[]{filter + "%"};
        }

        sql += " ORDER BY date_used DESC";

        if (max > 0) {
            sql += " LIMIT " + max;
        }

        return ReaderDatabase.getReadableDb().rawQuery(sql, args);
    }
}
```

Listing 1.2 Instance of SQL code smell

[4] https://github.com/wordpress-mobile/WordPress-Android/blob/develop/WordPress/src/main/java/org/wordpress/android/datasets/ReaderSearchTable.java

Refactoring practices

Refactoring is a way of removing or reducing technical debts at the design level or implementation level. The technical debts are addressed by changing the internal behaviour of the system while preserving the external-behaviour [13]. Studying refactoring practices help provide insights into if/how technical debts are addressed. Several studies specified refactorings, proposed automated refactoring approaches, and studied the characteristics of refactorings and the impact of refactorings in mitigating technical debts in traditional software systems. Similarly, investigating refactoring practices in data-access classes under the context of data-intensive systems will provide insights into how data-access technical debts are managed.

1.1 Problem statement

The major functionality of a data-intensive system is data management and processing. Hence, the efficiency of a data-intensive system is highly affected by the efficiency of data-access operations. Technical debts introduced in data-access operations could harm the overall software quality of data-intensive systems. Several studies addressed the specification of traditional technical debt [1,7,13,14], characterization of traditional technical debts [1,8,10, 11,15–17], impact of traditional technical debts on software quality [8,18–21], and refactoring practices in traditional software systems [22–26]. On the other hand, data-access technical debts are just getting attention recently with few works on specification, characterization, and impact on software quality.

While there exist some studies on performance anti-patterns in SQL-based applications, [12, 27, 28], to the best of our knowledge, we did not find a study that specifies data-access SATDs. Also, the existing specifications of performance anti-patters considered only SQL-based data-access code, and they may not generalize to NoSQL-based and polyglot persistence systems. Due to the variation in the data representation and access mechanism in NoSQL-based databases, it is difficult to come up with a catalogue of NoSQL data-access anti-patterns that are not specific to certain NoSQL databases. There are some data-access performance anti-patters specific to some NoSQL databases such as MongoDB, Riak, or Redis. Hence, more work is needed to specify data-access performance anti-patterns that are not specific to a certain NoSQL database.

The characterization of data-access technical debts is not well explored compared to traditional technical debt. Sharma et al. [29] investigated database schema quality on 357 industrial and, 2568 open source projects. However, they did not consider data manipulation queries. In another study, Filho et al. [30] conducted an exploratory study on the prevalence

and co-occurrence of bad smells in PL/SQL (Procedural Language for SQL) projects. While this work sheds light on the prevalence of data-access technical debts, the authors did not consider queries embedded in data-access classes of data-intensive software systems. Furthermore, the co-occurrence of data-access technical debts with traditional technical debts and their evolution is not yet explored.

Besides the specification and characterization of data-access technical debts, it is necessary to investigate the impacts of data-access technical debts on software quality. The impacts of such debts can be investigated by correlating the technical debts with quality attributes (Eg. bugs) or by asking software developers about the perceived criticality of the anti-patterns. Studying the impact helps to prioritize critical technical debts, as addressing all technical debts may not be practical in large-scale software systems.

Technical debts at the design level and code level are removed or addressed by refactoring. Hence, studying refactoring practices help provide insight into how developers manage technical debts. Several studies investigated refactoring activities in traditional software systems and if/how traditional technical debts are addressed. However, we did not find similar studies for data-access technical debts.

1.2 Thesis statement

The goal of this research is to improve the quality of data-intensive systems. While data-intensive systems are introduced to address big data challenges, the complexity of their design and implementation, and the heterogeneity of their components added to the release pressures on their developers make them prone to both traditional technical debts and data-access technical debts. While there are plenty of studies about traditional technical debts, data-access technical debts are just getting attention recently. We believe that specifying and investigating the characteristics and impacts of data-access technical debts will provide awareness to stakeholders involved in data-intensive systems about the management of technical debts and help researchers to provide tool support for refactoring or technical debt management.

Thesis statement: Data-access technical debts (1) are prevalent and persistent in data-intensive systems (2) negatively impact software quality, and (3) they are not addressed during refactoring.

1.3 Research objectives

The general objective of this research is to investigate data-access technical debts in the context of data-intensive systems. The specific objectives that will help achieve our general objective, the corresponding research questions, and links to the chapters in this dissertation are outlined below.

Objective 1. Specify data-access technical debts

Our goal here is to complement existing specifications of technical debts by specifying new types of data-access SATDs and performance anti-patterns. We answer the following research questions to achieve this objective.

RQ 1.1: **What is the composition of data-access SATD?**

We conducted an inductive coding qualitative study on sample data-access technical debts from SQL-based and NoSQL-based open-source data-intensive systems to extend the categories of traditional SATDs [1] with new data-access SATDs (Chapter 5).

RQ 1.2: **What are the data-access performance anti-patterns prevalent in data-intensive systems?**

We collected and manually analyzed issues reported from NoSQL-based and polyglot open-source data-intensive systems to label the root causes of the issues and build the taxonomy of the data-access performance anti-patterns using inductive coding (Chapter 5).

Objective 2. Study the characteristics of data-access technical debts

Once we specify data-access technical debts, the next goal is to characterize such debts by investigating their prevalence, co-occurrence with traditional smells, and their evolution using open-source data-intensive systems as subject systems. We have two sub-objectives under this objective that are (2.1) to characterize data-access SATDs and (2.2) to characterize SQL code smells.

Sub-objective 2.1 Characterization of data-access SATDs

We conducted an empirical study on data-access SATDs to investigate their prevalence and evolution using open-source data-intensive systems as subject systems. We achieved this objective by answering the following research questions.

RQ 2.1: **How prevalent are SATDs in data-intensive systems?**

To answer this RQ, we collected SATD comments from multiple snapshots of the subject systems and compared the prevalence of data-access SATDs against traditional SATDs and between SQL-based and NoSQL-based data-intensive subject systems (Chapter 6).

RQ 2.2: **How long do SATDs persist in data-intensive systems?**

We performed survival analysis on SATDs to understand how long SATDs persist before getting addressed. We compared the survival curves of data-access SATDs against traditional SATDs and survival rates of data-access SATDs between SQL-based and NoSQL-based subject systems (Chapter 6).

RQ 2.3: **What are the circumstances behind the introduction and removal of data-access SATD?**

We used the introduction and removal of SATD comments as a proxy for the corresponding introduction and removal of SATDs. We first identified data-access SATD introducing commits and removal commits and use the corresponding commit time to identify when they are introduced or removed. We also manually analyzed the introduction and removal commit messages to understand why data-access SATDs are introduced or removed (Chapter 6).

Sub-objective 2.2 Characterization of SQL code smells

While data-access non-SATDs include SQL code smells and data-access performance anti-patterns, we do not have a detection tool for data-access performance anti-patterns. Hence, we restricted our analysis to SQL code smells. We conducted a quantitative study to characterize SQL code smells by investigating the prevalence of SQL code smells, the co-occurrence of SQL code smells with traditional code smells and the evolution of SQL code smells. We answered the following research questions to achieve this sub-objective.

RQ 2.4: **What is the prevalence of SQL code smells across different application domains?**

We extracted SQL code smells from the latest snapshots of SQL-based data-intensive subject systems to compute the prevalence as the ratio of the number of SQL code smells to the number of queries. We compared the prevalence of several SQL code smells across different application domains (Chapter 7).

RQ 2.5: **Do traditional code smells and SQL code smells co-occur at class level?**

We extracted SQL code smells, and traditional code smells from multiple snapshots of SQL-based data-intensive systems and applied *Apriori algorithm* and statistical test of association to understand the co-occurrence of SQL code smells, and traditional code smells (Chapter 7).

RQ 2.6: **How long do SQL code smells survive?**

We conducted a survival analysis of SQL code smells to understand how long SQL code smells persist in data-intensive systems (Chapter 7).

Objective 3. Investigate impacts of data-access technical debts on software quality

Once we characterize data-access technical debts, it is important to understand the impacts of such debts on software quality. We specifically investigated the impacts of SQL code smells on bugs and the impacts of data-access performance anti-patterns as perceived by developers. We answered the following research questions to understand the impacts of data-access technical debts.

RQ 3.1: **Do SQL code smells co-occur with bugs?**

To investigate the impact of SQL code smells on bug proneness, we first started with a co-occurrence analysis between SQL code smells and bugs using the Apriori algorithm and statistical test of association. For SQL code smells to have an impact on bug-proneness, they first need to have a significant co-occurrence with bugs. We identified bug-fixing commits and their corresponding bug-inducing commits using SZZ and applied the Apriori algorithm and statistical test of association (Chapter 8).

RQ 3.2: **What is the perceived criticality of SQL code smells?**

We evaluated the criticality of prevalent SQL code smells as perceived by developers using a developer survey. In the survey, we asked the developers to rate the criticality of the SQL code smells and to justify their rating with an open-ended question. We quantitatively and qualitatively analyzed the survey responses to answer this RQ (Chapter 8).

RQ 3.3: **How do developers perceive the criticality of data-access performance anti-patterns?**

After we specified data-access performance anti-patterns, we surveyed developers to rate the criticality of those anti-patterns based on their experience and asked for their

justification in open-ended questions. We analyzed the criticality rating and justifications to answer this RQ (Chapter 8).

Objective 4. Study data-access refactoring practices

Since design-level and code-level technical debts are addressed by refactoring, it is important to investigate data-access refactoring practices to understand if and how refactorings are applied to address data-access technical debts. We conducted a quantitative and qualitative study of several aspects of data-access refactoring and contrasted it with refactoring performed in non-data-access (regular) classes. We answered the following research questions to achieve this objective.

RQ 4.1: **How prevalent are refactorings in data access classes?**

To answer this RQ, we computed the prevalence of data-access refactorings in absolute numbers, as well as normalizing by code size. We also compared the prevalence of data-access refactorings and non-data-access refactorings between SQL-based and NoSQL-based subject systems (Chapter 9).

RQ 4.2: **How do refactoring activities change during the lifetime of the subject systems?**

To investigate what types of refactorings are applied as systems evolve, we extracted the commit time for each refactoring time and plotted the distribution of the commit times. We compared the distributions between data-access refactorings and non-data-access refactorings. We also investigated how the prevalence of refactorings is affected by release deadlines (Chapter 9).

RQ 4.3: **Do data access refactoring activities touch SQL queries and SQL code smells?**

In this RQ, we investigated if SQL code smells are addressed during data-access refactorings. We computed the co-occurrence of refactorings with SQL queries and SQL code smells using both line-level matching and method-level matching (Chapter 9).

RQ 4.4: **Do different types of refactorings co-occur in data access classes?**

We investigated if multiple refactoring types are applied together (composite refactorings). We computed the co-occurrence of prevalent refactoring types at the commit level using the Apriori algorithm and also performed a statistical test of association between different refactoring types in data-access classes (Chapter 9).

RQ 4.5: **What is the profile of developers performing data-access refactorings?**

To understand the profile of developers involved in data-access refactoring, we extracted the developer information from the refactoring commits and collected several metrics evaluating the contribution and refactoring experience of developers. Understanding the profile of developers involved in refactoring sheds light on how refactoring tasks are assigned to developers and if/how familiar the developers need to be to apply the data-access refactorings (Chapter 9).

RQ 4.6: **What do code elements targeted by data access refactorings implement?**

We conducted a qualitative analysis of sample data-access refactoring instances to understand what components of data-access logic are often prone to refactoring. We manually analyzed the code changes associated with the refactorings to identify the functionality of the target codes (Chapter 10).

RQ 4.7: **What is the context in which data access refactoring occur?**

In this RQ, We investigated the context of data-access refactorings by manually analyzing sample data-access refactoring commit messages and labelling the context of the refactorings using deductive coding. Answering this RQ, helps us to understand if refactorings are applied just to address technical debts or if the refactorings are done together with other software development activities such as bug-fixing, adding a new feature, changing feature, or a combination of such activities (Chapter 10).

RQ 4.8: **What is developers' opinion about refactoring practices in data-access classes?**

In the previous RQs, we characterized data-access refactorings using data collected from open-source systems. We conducted a developer survey on refactoring practices to triangulate the findings with the current practice in the industry. This RQ helps us to understand to what extent the findings from the qualitative and quantitative analysis are in line with the actual practices in the industry (Chapter 10).

1.4 Research contributions

We outline the contributions from the dissertation and the corresponding chapters.

1. A catalogue of data-access performance anti-patterns (Presented in Chapter 5)

2. Taxonomy of data-access SATD (Presented in Chapter 5).

3. Empirical study on prevalence, co-occurrence, and evolution of SQL code smells (Presented in Chapter 7).

4. Empirical study on prevalence and evolution of data-access SATDs (Presented in Chapter 6).

5. Empirical study on the impacts of SQL code smells on bug proneness (Presented in Chapter 8).

6. Mixed method analysis of developer survey to understand the criticality of prevalent SQL code smells (Presented in Chapter 8).

7. Mixed method analysis of developer survey to understand the criticality of data-access performance anti-patterns (Presented in Chapter 8).

8. Empirical study to characterize data-access refactoring practices (Presented in Chapter 9 and Chapter 10).

9. A catalogue of data-access functionalities prone to refactoring (Presented in Chapter 10).

10. Mixed method analysis of developer survey to understand practical aspects of refactoring activities from developer's point of view (Presented in Chapter 10).

1.5 Articles related to the dissertation

In this section, we outline the list of published articles and under review that are included in this dissertation in the chronological order of publication and submission.

1. B. A. Muse, M. M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "On the prevalence, impact, and evolution of sql code smells in data-intensive systems," in Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 327–338. (The research questions **RQ 2.4, RQ 2.5, RQ 2.6, RQ 3.1** are presented in this article.)

2. B. A. Muse, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "FIXME: synchronize with database! an empirical study of data access self-admitted technical debt," Empir. Softw. Eng., vol. 27, no. 6, p. 130, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10119-4 (The research questions **RQ 1.1, RQ 2.1, RQ 2.2, RQ 2.3** are presented in this article.)

3. B. A. Muse, F. Khomh, and G. Antoniol, "Do developers refactor data access code? an empirical study," in IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022. IEEE, 2022, pp. 25–35. [Online]. Available: https://doi.org/10.1109/SANER53432.2022.00014 (Parts of the research questions **RQ4.1, RQ4.2 RQ4.3 and RQ 4.6** are presented in this article.)

4. B. A. Muse, F. Khomh, and G. Antoniol, "Refactoring Practices in the Context of Data-intensive Systems" (A journal extension of the third publication is accepted to **Empirical software engineering (EMSE) journal**. The research questions **RQ 4.1, RQ 4.2, RQ 4.3, RQ 4.4, RQ 4.5, RQ 4.6, RQ 4.7, and RQ 4.8** are presented in this article).

5. Muse, B. A., Nafi, K. W., Khomh, F., & Antoniol, G. (2022). "Data-access performance anti-patterns in data-intensive systems". (Submitted to **Empirical software engineering (EMSE) journal**. The corresponding **Registered report** presenting the methodology received **continuity acceptance** to the Empirical software engineering journal and was presented at the International Conference on Software Maintenance and Evolution (ICSME 2022) in the registered reports track. The research questions **RQ 1.2 and RQ 3.3** are presented in this article.

1.6 Dissertation organization

The remainder of the dissertation is organized as follows. We provide the background information about concepts and analysis methods used in this dissertation in **Chapter 2**. We review the related literature in **Chapter 3**. In **Chapter 4**, we describe the methodology we followed to identify subject systems, collect, extract data and prepare the datasets used in the empirical studies. We present the specification of data-access technical debts in **Chapter 5**. We present the characterization of data-access SATDs in **Chapter 6** and the characterization of SQL code smells in **Chapter 7**. In **Chapter 8**, we present the impact of data-access technical debts on software quality. We present the findings of a quantitative analysis of refactoring practices in **Chapter 9** and qualitative analysis of refactoring practices in **Chapter 10**. We finally provide the conclusion of our work, the limitations, and future research extensions in **Chapter 11**.

CHAPTER 2 BACKGROUND

2.1 Chapter overview

In this chapter, we provide background information about SQL code smells. We also introduce the algorithms and statistical methods utilized in this dissertation including survival analysis, topic modeling, the Apriori algorithm, and Cramer's V test of association.

2.1.1 SQL code smells

Data-access smells are bad practices in data-access code that impact the performance and robustness of data-access operations. Data-access classes based on relational databases are prone to SQL code smells. SQL code smells are data-access smells manifesting in SQL statements that are standalone or embedded in a data-access code. The category of SQL code smells is defined in the book of Karwin [27]. However, we briefly describe SQL code smells that are covered in this study and can be detected by SQLInspect [31], the SQL code smell detection tool used in this case study.

Implicit Columns:

Implicit Columns smell occurs when select queries fetch unnecessary columns from the database by using select all (*). It may cause performance issues such as bandwidth wastage and creates unnecessary coupling between the database and application code [27]. For example, the select query in Listing 2.1 is prone to *Implicit Columns*. If the customer table contains many columns, explicitly specifying the needed columns instead of selecting all would fix this smell.

```
Example: SELECT * FROM customer;
```

Listing 2.1 Example of Implicit Columns smell

Fear of the unknown:

Fear of the unknown smell occurs when improper handling of null values and null check during data access causes unexpected error [32]. The query in Listing 2.2 will not return rows where customer_id is NULL as intended. The correct way would be to use *IS NULL* operator for a null check.

```
SELECT * FROM customer WHERE customer_id=NULL;
```

Listing 2.2 Example of Fear of the unknown smell

Ambiguous Groups:

Ambiguous Groups occurs when developers misuse the GROUP BY statement by adding columns in the select statement that are not aggregated [27]. Handling not aggregated columns in queries with GROUP BY is different among different database systems. For example, running the query in Listing 2.3 on SQLite database will let the query engine pick one random phone number from any of the rows of the employee table, which may not be desirable.

```
SELECT city, phone, AVG(salary) FROM employee GROUP BY city;
```

Listing 2.3 Example of Ambiguous Groups smell

Random Selection:

Random Selection smell occurs when querying a single random row from the database, which forces a full scan, which has a negative performance impact for large size tables [27]. The query in Listing 2.4 fetches a random single row from the customer table which forces a full scan of the customer table.

```
SELECT customer_name FROM customer ORDER BY RAND() LIMIT 1;
```

Listing 2.4 Example of Random Selection smell

SQLInspect supports the detection of these four smells out of the total six types of query smells from the catalog of Karwin; hence, we also rely on these. We notice that the catalog of Karwin groups smells into the following categories: Logical Database Design, Physical Database Design, Query, and Application Development. As our goal is to investigate the application code, the relevant ones for us are the last two categories. However, SQLInspect does not implement the detection of smells in the Application Development category, as they are not explicitly in the SQL code. SQL Code smell detection in SQLInspect relies on SQL query extraction, which has a minimum precision of 88 % and a minimum recall of 71.5% [33]. Hence, the aforementioned precision and recall values can be considered as an upper bound

for SQL Code smell detection performance. More details on SQLInspect and the supported smells can be found in the related papers of Nagy et al. [31, 32].

2.2 Survival analysis

Survival analysis [34] is a statistical analysis technique that provides the expected time for an event's occurrence. The event of interest could be anything as long as it is clearly defined. We define a study observation window and track events of interest that occur within the window. If the subjects under study leave during the period of observation, the corresponding data will be censored. If the event is not observed during the observation period, the corresponding subject will be censored at the end of the period. *Time to event* and *status* are two important variables for survival analysis. To compute each variable, we first need to define an event of interest that depends on the problem we want to analyze. In our case, an event of interest is the *removal of a SATD or removal of SQL code smell*.

Time to event (T) is defined as the time interval between the starting of observation (the first instance of the SATD) and the occurrence of an event of interest or the censoring of data. Time to event T is a random variable with only positive values and can be measured in any unit [34]. The most common approach is to use time in minutes, hours, days, months, or years. However, we will use the number of commits to consider that the actual time may not correctly reflect software evolution compared to the number of commits. Projects have different activities at different times. Commits could be made more frequent at specific periods of time and less frequent at other times. Using time for T in those cases has a limited capability to reflect project evolution. On the contrary, the *number of commits* directly measures the project activity regardless of activity variation in some periods of time.

It is important to define an observation window and flag events outside it as censored. In our case, we define the observation window to cover all our snapshots of the subject systems. We flag SATDs and SQL code smells that persist in the latest snapshots as censored, since we do not know if the event of interest will occur or not. Similarly, when an entire source file with one or more SATD comments or SQL code smells is deleted within the observation window, we flag the SATDs and SQL code smells as censored. The reason is that in this case, it cannot be determined whether the technical debts are removed intentionally or only because of the file deletion. This is also supported by the observation of Zampetti et al., who found that 20%–50% of the removals of SATDs are accidental and are even unintended [35].

Survival analysis takes a boolean variable called *status* to distinguish between censored data and non-censored data. For instance, it takes a value of 1 when the event of interest occurred

and 0 otherwise.

The survival function S(t) gives the probability ($P(T > t)$) that a subject (SATD in our case) will survive beyond time t.

After we computed T and *status*, we can choose our survival estimator. We selected one of the commonly used survival estimators, the Kaplan-Meier estimator [36]. The Kaplan-Meier estimation is computed following Equation 2.1. t_i is the time duration (in the number of commits) up to event-occurrence (removal of SATD or SQL code smell) point i, d_i is the number of event occurrences up to t_i, and n_i is the number of SATDs or SQL code smells that survive just before t_i. n_i and d_i are obtained from the input data.

$$S(t) = \prod_{i:t_i \leq t} [1 - \frac{d_i}{n_i}] \tag{2.1}$$

2.3 Topic modelling

Topic modeling [37] is one of the unsupervised machine learning techniques that, given a set of documents (document corpus), can detect the word and phrase patterns and cluster the documents based on word similarity. In our case, the corpus will be our dataset, and each comment will be one document in the corpus. Topic modeling works by counting the words and grouping documents with similar word patterns. Topic modeling is one of the frequently used techniques in *natural language processing* (NLP).

Latent semantic analysis (LSA) [37] and *latent Dirichlet allocation* (LDA) [38] are commonly used topic modeling algorithms. We also rely on LDA to assign topics to a set of words, assuming that the arrangement of words determines the topic. LDA model is trained using a tokenized and pre-processed set of documents. After the LDA is trained, it can assign a document to a topic group with a certain probability. In this paper, we use LDA to cluster comments based on similarity so that our sampled data for manual analysis is not biased toward a specific topic.

LDA has hyper-parameters such as the number of topics, alpha, and beta to control the similarity levels that affect the model's performance. The first one determines the *number of topics* generated by LDA after training. It can take any positive integer value. An insufficient value results in a too-general model that makes topic interpretation difficult. An excess number of topics creates many topics that are too fine-grained for classification and subjective evaluation [39]. *Alpha* controls the document topic density. A higher alpha makes the documents contain many topics. On the contrary, a smaller alpha makes the documents

have a few topics. *Beta* controls the topic word density, determining the number of words in the corpus associated with a topic. The higher the *beta* value, the more words are associated with a topic. All those parameters need to be tuned using the target dataset by optimizing for the best performance of the LDA model.

Performance evaluation of LDA:

A topic model can be evaluated by human judgment and intrinsic methods such as *perplexity and coherence*. *Perplexity* measures how well a probability model predicts a sample. It is computed by assessing the LDA model with unseen or held-out data. The lower the perplexity, the better the performance of the model. While perplexity measures the prediction of the LDA model, it does not evaluate the interpretation of the generated topics [40]. Another approach is to use coherence for evaluation. The coherence score is computed following segmentation, probability estimation, confirmation measure, and aggregation [41]. The coherence score is calculated by summing the scores of a pair of words that describe a topic on the assumption that words that often appear together in the document are more coherent. Coherence takes a value between 0 and 1. The higher the score, the better the model.

2.4 Apriori algorithm

Apriori algorithm is used for mining association rules by incrementally building item sets that co-occur frequently in a dataset. It was proposed by Agrawal and Srikant in 1994 [42]. Apriori algorithm can be applied in different domains including market basket analysis and software engineering. The market basket analysis aims to determine how to arrange items in a store by putting together items that are usually purchased together in supermarkets or online stores.

Apriori algorithm works by scanning the dataset and identifying frequent item sets based on different metrics such as support [43], confidence [43], lift [44], leverage [45] and conviction [44]. The thresholds for such metrics are context-dependent and often set as hyperparameters of the algorithm. We will describe the metrics of the Apriori algorithm used in our co-occurrence analysis of refactoring types. Our dataset contains the occurrence of each refactoring type in each snapshot of a file. The occurrence is a binary value of 1 and 0. We will use refactoring types labeled A and B for defining the metrics.

Support:

Support measures the proportion of rows that contain the candidate refactoring type to all the rows in the dataset. It has a value between 0 and 1. Support is defined in Equation 2.2

for a single item and 2.3 for two items.

$$Support\{A\} = \frac{no\ of\ rows\ containing\ A}{Total\ no\ of\ rows} \tag{2.2}$$

$$Support\{A, B\} = \frac{no\ of\ rows\ containing\ A\ and\ B}{Total\ no\ of\ rows} \tag{2.3}$$

Confidence:

Confidence measures how likely rows that contain one refactoring type also contain another refactoring type. Equation 2.4 defines Confidence. Confidence also has a value between 0 and 1. However, this metric only considers the popularity of one item. For instance, if B is more popular, it will raise confidence regardless of the actual association.

$$Confidence\{A \rightarrow B\} = \frac{Support\{A, B\}}{Support\{A\}} \tag{2.4}$$

Lift:

Lift addresses the limitation of confidence by accounting for the support of the second item. Lift is defined in Equation 2.5 A lift value of 1 means there is no association between the refactoring types and more than 1 indicates some association. A value less than 1 indicates refactoring type B is unlikely to co-occur with A.

$$Lift\{A \rightarrow B\} = \frac{Support\{A, B\}}{Support\{A\} * Support\{B\}} \tag{2.5}$$

Leverage:

Leverage measures the difference in probability of items occurring together and the case if the items are occurring independently. Equation 2.6 defines leverage. A leverage value of 0 indicates the items are independent. A leverage value above 0 indicates some degree of association.

$$Leverage\{A \rightarrow B\} = Support\{A, B\} - Support\{A\} * Support\{B\} \tag{2.6}$$

Conviction:

Conviction measures the probability that A appears without B assuming they are dependent on the actual frequency of A without B. A conviction value of 1 shows that A and B are

independent. Conviction is defined in Equation 2.7

$$Conviction\{A->B\} = \frac{(1 - support\{B\})}{(1 - Confidence\{A \rightarrow B\})} \quad (2.7)$$

2.5 Cramer's V test of association

Cramer's V is a statistical test of association between nominal variables [46]. It assumes a value between 0 and 1 inclusively. A value of 0 indicates no association, and a value of 1 indicates maximum association. The Cramer's V test of association is based on the result of Chi-squared test, taking into account sample size when comparing two nominal variables. Equation 2.8 defines Cramer's V test of association where X^2 is the Pearson's Chi-squared coefficient, n is the total number of samples, and A and B represent the number of distinct values of the categorical variables A and B respectively.

$$V = \sqrt{\frac{X^2}{n * min(A-1, B-1)}} \quad (2.8)$$

2.6 Chapter summary

In this chapter, we presented background information regarding SQL code smells and provided an introduction to algorithms and statistical analysis methods utilized in this dissertation.

CHAPTER 3 LITERATURE REVIEW

3.1 Chapter overview

In this chapter, we present the previous work related to the research problem described in Section 1.1. In particular, we discuss studies about specification of technical debts, characterization of technical debts, impacts of technical debt on different software development activities and studies on the refactoring practices in traditional software systems.

3.2 Specification of technical debt

Li et al. conducted a systematic mapping study on technical debt and its management [6]. They examined 49 papers, classified technical debts into ten categories and identified eight activities and 29 technical debt management tools.

Rios et al. performed a tertiary study and evaluated 13 secondary studies dating from 2012 to March 2018 [47]. As a result, they developed a taxonomy of technical debt types and identified a list of situations in which debt items can be found in software projects.

Alves et al. performed a systematic mapping study by evaluating 100 studies dating from 2010 to 2014 [48]. They also proposed a taxonomy of technical debt types and created a list of indicators to identify technical debt.

Alves et al. [49] conducted a systematic literature review identifying 29 technical debt prioritization approaches. Among the 29 approaches, 70.83% address a specific type of technical debt, while the remaining approaches can be applied to any kind of technical debt. 33.33% of the approaches address code debt, 16.67% address design debt, 12.5% address defect debt and 1% of the approaches is shared by SATDs, database normalization debt, requirement debt and architectural debt. Among all approaches, 54.17% consider value and cost as prioritization decision factors, 29.17% rely on value only, and 16.67% of approaches are based on value cost and constraint.

3.2.1 Traditional code smells

Fowler et al. [50] described code smells as “certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring.“ They discussed 22 different traditional code smells such as duplicate code, long method, and large class. They also discussed possible refactoring approaches for the described code smells. We call such smells traditional code

smells to distinguish them from data-access related code smells.

3.2.2 Self-admitted technical debts (SATD)

The pioneering work on SATD was done by Potdar et al. [7] where they identified comment patterns that indicate SATDs. They manually analyzed 101,762 comments to identify the patterns prevalent in SATDs including "TODO", "FIXME" and "Ugly". They also studied the prevalence of SATDs in four open-source projects.

Bavota and Russo [1] conducted a differentiated replication of the work of Potdar and Shihab [7]. They proposed a taxonomy of SATDs. The type of SATDs identified include but are not limited to *test debt, design debt, code debt, requirement debt, and defect debt*. They also identified several sub-categories under those SATDs. While this work is the first effort to categorize traditional SATDs, they need to be extended to cover data-access SATDs.

3.2.3 Multi-language design smells

Practical real-world software systems nowadays are composed of multiple modules possibly from different programming languages and multiple persistence technologies. As a result, multi-language systems are getting attention in recent literature. Multi-language systems are being developed to leverage the advantages of the component languages, promote code reuse, and to integrate legacy code [51]. Java Native Interface (JNI) is one example of multi-language support for Java. JNI provides an interface for java to invoke and be invoked by code in other programming languages. Grichi et al. [52] studied the usage of Java Native Interface (JNI) in open source projects and found that JNI is often used during loading libraries, exception management, and implementing native methods.

Abidi et al. [51] investigated different software artifacts and documentations and proposed a catalog of multi-language anti-patterns that include *Excessive Inter-language communication*, *Too much scattering*, *Too much clustering*, *unnecessary use of Multi-language programming*, and *language and paradigm mismatch*.

3.2.4 Deep learning design smells

Machine learning in general and deep learning in particular are typically used in data-intensive systems in the data-processing component. Recently Nikanjam and khomh [53] specified eight design smells on feedforward neural networks collected from literature review and analysis of 659 deep learning programs with performance issues obtained from stack

overflow. The identified smells include: *Non-expanding feature map, Losing local correlation, heterogeneous blocks of CNNs, too much down-sampling and so on.*

3.2.5 Video game smells

Modern video games can be considered as data-intensive systems as they process, store and transfer large amount of data. Like traditional softwares, video games are also prone to technical debt and specifically video game smells. Borrelli et al. [54] specified and detected video game smells on Unity engine based games. They specified video game smells including but not limited to: *Allocating and destroying GameObjects in updates, Coupling objects through the IDE Inspector, and Heavyweight Update methods.*

In another work, Nardone et al. [55] expanded the proposed video game smells by Borrelliet al. analyzing forums related to video game development. Their work added several game design smells obtained from several popular game engines. They identified 28 video game smells with 5 high-level categories, related to multiplayer, game design and logic, animation, physics, and rendering.

3.2.6 Specification of data-access technical debts

Albarak and Bashoon defined the concept of database design debt as "*the immature or suboptimal database design decisions that lag behind the optimal/desirable ones, that manifest themselves into future structural or behavioral problems, making changes inevitable and more expensive to carry out*" [56]. They develop a taxonomy of debts related to the conceptual, logical, and physical design of a database. For example, they claim that ill-normalized databases (*i.e.,* databases with tables below the fourth normal form) can also be considered technical debt [57]. To tackle this specific type of debt, they propose an approach to prioritize tables that should be normalized.

Foidl et al. claim that technical debt can be incurred in different parts (*i.e.,* software systems, data storage systems, data) of data-intensive systems and different parts can further affect each other [2]. They propose a conceptual model to outline where technical debt can emerge in data-intensive systems by separating them into three parts: software systems, data storage systems and data. They present two smells as examples. Missing constraints, when referential integrity constraints are not declared in a database schema; and metadata as data, when an entity-attribute-value pattern is used to store metadata (attributes) as data. While this study provided a conceptual model for components of data-intensive systems prone to technical debt, it did not provide empirical evidence for the existence of the technical debt. We

contribute to addressing this gap by investigating SATDs in data-intensive systems.

Weber et al. [28] also identified relational database schemas as potential sources of technical debt. In particular, they provided a first attempt at utilizing the technical debt analogy for developing processes related to the missing implementation of implicit foreign key (FK) constraints. They discuss the detection of missing FKs, propose a measurement for the associated TD, and outline a process for reducing FK-related TD. As illustrative case study, they consider OSCAR, a large Java medical record system used in Canada's primary health care.

3.2.7 Specification of SQL code smells

Bill Karwin [27] described Database anti-patterns at different levels of database systems which are physical database design, logical database design, SQL (query) anti-patterns, and application development anti-patterns. Rounding errors, 31 Flavors, phantom files, and Index Shotgun were discussed as physical-level anti-patterns. Karwin also discussed logical database design anti-patterns such as JayWalking, Multi-column attributes, and keyless entry. Anti-patterns such as SQL Injection, Readable Passwords, and Magic beans are described as application-level anti-patterns.

For the SQL query anti-patterns, Karwin described *Fear of the unknown* smell related to improper null checks; *Ambigious groups*, related to fetching columns that are not part of the group by clause; *Random selection* smell that occurs when a single random row is fetched requiring expensive indexing and sorting operations; *Poor man's search engine* which is a case of using wildcards for searching strings; Implicit columns smell that occurs when column names are not explicitly specified in select, insert and update queries; *Spaghetti Query* smell which occurs when developers try to solve the complex task with a single complex query similar to God class traditional code smell. Karwin also demonstrated the impacts of those smells with examples and cases where the smells are not necessarily bad. He also provided some detection and refactoring insights for all the smells.

3.2.8 Specification of data-access performance anti-patterns

There are several studies on performance bugs [12, 58–61], their root cause [12, 58, 62], fixing strategy [12, 58, 59] their impact or relevance [12, 61] and both static and dynamic analysis based detection approaches [63–66]. Researchers also suggested various ways of data-access optimization to improve the performance of database-backed web applications using caching and prefetching techniques. [67–70]. Most of the performance studies either focus only on

systems that use ORM driven relational databases or consider a subset of the performance anti-patterns, hence their findings may not be generalized to the case of NoSQL and polyglot persistence based data-intensive subject systems.

The closest work to this study is the work of Shao et al. [12] where they provided a catalog of data-access performance anti-patterns obtained from (1) a literature survey (24 anti-patterns) and (2) 10 new anti-patterns, analyzing real-world performance bugs collected from seven open-source relational database backed web applications (BugZilla, DNN, Joomla!, MediaWiki, Moodle, WordPress, and Odoo). Our keyword-based approach to identifying data-access performance bugs is similar to the method used in this work. However, we extended the keywords to cover the case of NoSQL databases. While most anti-patterns are associated with SQL queries, some performance anti-patterns like moving computation to the server or not caching could also be observed in NoSQL-based systems. Hence, we extended the data-access performance anti-patterns in this study to the case of NoSQL and polyglot persistence data-intensive systems.

3.3 Technical debt detection and refactoring detection approaches

In this section, we present proposed approaches to detect technical debts including traditional code smells, SATDs and SQL code smells and approaches to detect refactorings.

3.3.1 Traditional code smell detection and refactoring approach's

There are many proposed approaches to detect traditional code smells. The underlying detection mechanism can be grouped into rule-based, textual analysis-based, and machine learning based.

Moha et al. proposed DECOR [71], a rule-based detection approach. In this paper, they described the steps required for the specification and detection of code and design smells. The general steps of smell detection are to describe and provide the specification of a smell, translate the specification to detection algorithms and finally return code artifacts with smells and manually validate the obtained smells. The authors instantiated Decor approach and provided DETEX concrete algorithm of smell detection. The first step in the algorithm is domain analysis which is a process of collecting descriptions of smells from literature. The second step is the specification to identify rules to describe smells. The rules are defined as thresholds in software metrics. The rules are implemented in a high-level domain-specific language (DSL). The DSLs were translated into a meta-model smell definition language. The authors evaluated their approach on 11 open-source systems and independent developers. The

result shows that the proposed detection algorithm has a precision of 60% and a Recall of 100% in most subject systems. Recent implementations of their approach can detect around 18 different traditional code smells.

Palomba proposed a textual analysis-based approach to detect code smells [72]. Palomba detected Long Method traditional code smell. The proposed detection approach extracts comments and identifiers from method blocks and performs the common NLP pre-processing steps. Finally, the similarity matrix between the documents in a method is computed using Latent Semantic Indexing (LSI). Long method smells are detected when an entry's similarity value is below a threshold value of 0.4. The proposed approach had a precision of 65% and a recall of 61%.

Machine learning-based approaches rely on different software metrics as input features. Hassaine et al. [73] proposed Bio-inspired modeling of code smells based on immune system response. The approach consists of learning from existing design smells (vaccines). The AIS (Auto-immune system) supervised learning system was used as a model with metrics values as input features. This approach achieved 65 to 90% precision and 100% recall.

Khomh et al. [74] Proposed a Bayesian Belief Networks (BBN) based machine learning model to detect anti-patterns. The classification result is a probability distribution for each class i.e., anti-pattern or not an anti-pattern. Inputs to the classifiers are software metrics and lexical properties converted to probability distributions. The model can be tuneable with expert knowledge. The evaluation shows this approach is superior in some cases to DECOR [71].

Maiga et al. proposed SMURF traditional code smell detection tool [75]. SMURF is built using A Support Vector Machine (SVM) model based on the polynomial kernel. SMURF can detect Blob, functional decomposition, Spaghetti code and Swiss Army Knife smells. The input features for the model are object-oriented metrics similar to other machine learning-based models. The model's accuracy can be improved using expert feedback and re-training. The evaluation result shows better detection performance compared to Bayesian-based approaches like Khomh et al. [74].

Kessentini et al. [76] proposed multi-objective genetic programming to detect traditional code smells. This algorithm was used to generate optimal threshold rules that cover certain instances of code smells given quality metrics as features. The objective functions are precision and recall. They evaluated their approach on 184 open-source android projects and were able to identify 10 android code smells with an accuracy of 82% and relevance of 77% obtained from developers' feedback.

Another popular code smell detection and refactoring tool is JDeodorant by Tsantalis et al.

[77]. JDeodorant focuses on the detection and refactoring of Feature Envy, God Class, Duplicated Code, Type Checking and Long Method smells. This tool can suggest refactoring decisions for those traditional code smells. Another refactoring tool is FaultBuster [78]. Szőke et al. proposed FaultBuster refactoring toolset. FaultBuster automatically identifies smells using static analysis and automatic refactoring on some of the smells. It has a feature to periodically scan source code, analyze and provide automatic refactoring, and is integrated with many IDES.

3.3.2 SATD detection and removal approaches

Current SATD detection approaches are either pattern-based or machine-learning-based approaches. Since machine learning-based approaches came after pattern-based approaches we will discuss them in a chronological order.

Pattern based SATD detection

De Freitas et al. [79] Proposed a contextualized vocabulary model to identify TD from source code analysis. The model consists of software-related terms, adjectives that describe the terms, verbs to model actions in comments, adverbs, and tags such as Fixme and Todo. The combined terms can be used for searching comments in a pattern-based approach. They tested the feasibility of their approach using jEdit and Apache Lucene projects and identified technical debts of different categories.

As an extension of the work in [79], De Farias et al. conducted an empirical study on the effectiveness of contextual vocabulary models (CVM) [80]. Besides evaluating the accuracy of the pattern-based approaches, they studied the impacts of language skills and developer experience on finding SATDs using a controlled experiment. Their result shows that the accuracy of the pattern-based approach looks promising but it needs further improvement. English reading skills affected the identification of SATDs using pattern-based approaches.

Machine learning based SATD detection

Maldonado et al. proposed NLP based approach to automatically identify SATDs [81]. Their approach can detect design SATDs and requirement SATDs. Furthermore, they build a manually labeled dataset of SATDs which consists of 62566 comments. This dataset is used as a benchmark in most of the subsequent studies. They built a multi-class regression model using their dataset. They evaluated their approach and achieved F1-measure between 40% to 60%. They observed that words related to sloppy code indicate design SATD while words

related to incomplete code as requirement SATD.

Huang et al. proposed a machine learning-based detection approach that combines the decisions of multiple Naive-Bayes-based classifiers into a composite classifier using majority vote [82]. The comments from source codes are represented using vector space modeling (VSM) features selected from SVM using Information gain. They achieved F1-Score of greater than 73.7%.

Liu et al. [83] proposed the SATD detector tool which is a concrete implementation of Huang et al. [82] approach. They provided this tool as a java back-end library implementing the model to train and classify comments and the corresponding eclipse plugin as a front end.

Most of the existing approaches rely on detecting SATDs at the file level. However, SATDs are often introduced during the evolution of software, and hence incorporating change history is also important. To address this, Yan et al. [84] extracted 25 features from the information of SATD introducing commits. They deployed a random forest classifier for detection. The authors achieved AUC of 0.82. Also, they found that features related to code diffusion were more deterministic features for the classification of SATD.

Yu et al. proposed the Jitterbug framework that combines pattern-based detection for easy-to-find SATDs and machine learning-based techniques for other SATDS [85]. The relevant patterns are chosen using precision as a fitness function. The objective of their approach is to reduce the manual inspection effort. The result showed that easy SATDs were identified with precision between 99% and 100%.

Zampetti et al. proposed SARDELE [86], a deep learning-based approach to automatically recommend how to remove SATDs, the deep learning model contains a convolutional neural network trained with comments and a recurrent neural network trained with source code. Their system is capable of recommending removal strategies such us changing API calls, conditionals, and return statements. SARDLE achieved an average precision of 55% and a recall of 57%.

Noiseux proposed TEDIOUS (TEchnical Debt IdentificatiOn System) [87] that recommends SATDs to developers to be admitted. TEDIOUS works by combining software quality metrics at the method level as features and comparing different machine learning classifiers. The recommendation system achieved average precision of up to 67% and recall of up to 55% using a random forest classifier.

As evident in recent works, SATDs can indicate places for potential improvement in software quality just like smells. However, SATDs is already admitted by developers which further stresses the importance of addressing SATDs as a significant part of maintenance activities.

The current detection approaches have limitations in terms of precision and recall. Hence, more work needs to be done to improve the detection performance as well as the generalizability of the detection models.

3.3.3 SQL code smell detection

Van Den Brink et al. [88] described different metrics to assess the quality of embedded SQL statements in source code. Those metrics can help measure the occurrence, structure, interrelation, and variation points of SQL queries. Example metrics such as the Number of queries, number of operations, number of input variables, Number of tables used, and number of nested queries are discussed in the paper. Those metrics can be mapped to quality attributes related to maintainability. The authors also implemented those metrics and measurement toolsets. They used PL/SQL, COBOL, and Visual Basic applications as case studies.

Khumnin and Senivongse proposed, Transact-SQL, to analyze database schema and detect database anti-patterns. Transact-SQL automatically detects some of the logical anti-patterns mentioned in the book of Karwin et al. [27]. This tool also proposes refactoring approaches for the smells detected in the database schema. Authors used different heuristics that look for certain patterns in the column names and table names for smell detection. Khumnin and Senivongse evaluated Transact-SQL using 3 industry database schemas, the result showed that Transact-SQL has high recall but smaller precision. Authors, further mentioned that detection is dependent on the semantics of the data and hence needs a semi-automatic approach that involves domain experts. The major drawback of their approach besides the low recall is that relying on patterns in naming for detection may not work in cases where naming conventions are not followed.

Delplanque et al. [89] proposed DBCritics a static database schema analyzer tool that detects database smells given any entity such as a table, view, function, triggers, etc.. of a database schema. The rules used for smell detection define the thresholds to decide smelly code from normal. Those rules define thresholds to detect smells such as unused functions, a table without a primary key, a foreign key referencing the primary key, and so on. The authors demonstrated DBCritics using two real-world databases as a case study.

One challenge in SQL code smell detection is parsing SQL statements embedded in other source codes. To address this Nagy and Cleve [32] proposed a static analysis approach to identify SQL code smells embedded in Java Code. SQL smell detection was achieved in two steps that are SQL extraction from java source code followed by SQL code smell detection that augments information obtained from database schema and data content besides the

extracted SQL. The smell detection algorithm looks for patterns corresponding to each SQL code smell in an abstract semantic Graph (ACG).

Nagy and Cleve [31] proposed SQLInspect query analysis and SQL smell detection tool as an implementation of their approach in [32]. Given a source code, it extracts the embedded SQL queries and generates SQL-related metrics besides smell detection. SQLInspect is an eclipse plugin with command-line interface support for batch analysis. SQLInspect has a precision of 88% and recall of 71.5% [33].

3.3.4 Refactoring detection approaches

Demeyer et al. [90] proposed a set of heuristics for detecting refactorings based on object-oriented quality metrics. They relied on object-oriented metrics including method size, class size, number of inherited methods, and number of immediate children class. The proposed heuristics utilize the changes in such object-oriented quality metrics to detect refactorings. They evaluated this approach on 3 object-oriented software systems but they did not report the precision and recall of their approach.

Antoniol et al. [91] proposed a refactoring detection approach based on vector space information retrieval to identify discontinuities in class evolution and potential refactorings. They focused on class renaming, class merging, and factoring out classes. They evaluated their approach on a single software system with 40 releases. Weißgerber and Diehl [92] proposed a refactoring detection approach in which they first looked for added, changed, and removed entities between versions as refactoring candidates and utilize clone detection techniques to rank the refactoring candidates based on the likelyhood of refactoring. This approach works for structural refactorings (refactorings that change the class structure of software) and local refactorings (refactorings performed inside classes). They evaluated this approach on a refactoring dataset and reported a maximum recall of 77% and precision of 92%.

Dig et al. [93] combined a syntactic analysis and semantic analysis to detect refactoring candidates. The static analysis is based on the Shingles encoding information retrieval technique to evaluate the similarity between code elements while the semantic analysis is based on reference graphs to track semantic relationships. They reported an accuracy of 85%.

Xing and Stroulia proposed JDEvAn tool sute [94] that implements UMLDiff algorithm to compare the UML representation of two versions and queries the algorithm for identifying refactoring instances.

Kim et al. [95] proposed Ref-Finder eclipse plugin that expresses the detected refactoring types in terms of template logic queries and logic programming engine for inference. This

tool supports at least 66 refactoring types.

Falleri et al. proposed GumTReeDiff tool [96] that compares generic abstract syntax trees of revisions and detects changes at the level of abstract syntax tree leaf nodes. While this tool does not detect refactorings it can be combined with refactoring detection rules to detect refactorings [97].

Silva et al. proposed RefDiff 2.0 [98] multilanguage refactoring detection tool relying on code structure tree source code representation. The refactoring detection first analyzes source code changs between two revisions and builds the code structure tree for each. Next, it computes the relationship between the entities to determine refactorings. They reported a precision of 96% and recall of 80%.

Tsantalis et al. proposed RefactoringMiner [97, 99] to detect refactorings in Java projects. Given two revisions of a system, this tool relies on abstract syntax tree matching algorithm to determine refactoring candidates. The matching algorithms follow a bottom-up approach starting from leaf statements and moving to composite statements. The matching is performed in multiple rounds utilizing abstraction and argumentization pre-processing steps. In our study, we relied on Refactoring Miner for the study in refactoring practices due to its state-of-the-art detection accuracy, faster execution time, and because it does not require similarity thresholds as an input.

3.4 Characterization of technical debts

In this section, we present recent works on the characterization of technical debts. We presented empirical studies regarding the prevalence and evolution of traditional code smells, SATDs, and SQL code smells.

Prevalence and evolution of traditional code smells

Since the initial introduction of the term “design flaws” [100] and “code smell” [50], many studied their impact on development; i.e., how they affect performance, source code quality or maintainability. A recent literature review “on the code smell effect” by Santos et al. [101] gives an overview of these studies. They identify a total number of 3530 papers in this area and after removing duplicated and short papers they do an in-depth examination of 64 papers in their survey.

Palomba et al. [10] Studied 395 releases of 30 open source projects to understand the prevalence and impacts of traditional code smells. They mined 17350 instances of 13 code smells

manually validated the smells. The result shows that traditional code smells such as Long Method and Spaghetti code had a higher prevalence in the subject systems. They also showed that smelly files are more prone to code change.

There are several empirical studies on the evolution of traditional code smells [8,10,11,15–17]. Olbrich et al. [16] studied the evolution of God class and Shotgun Surgery smells and their impact on change frequency and size. They showed that studying the evolution of smells provides insight on whether traditional code smells are intentionally refactored or not.

Peters et al. investigated the evolution of code smells in seven open-source projects [15]. The results showed that on average traditional code smells had a life-span of up to 50% of the lifetime of the subject systems. Furthermore, they, demonstrated that smell removals were associated with the removal of the smelly files rather than an intentional fix in most cases hinting towards developers' lack of concern or awareness of traditional code smells.

Understanding the circumstances when smells are introduced is important as it could lead to better refactoring recommendation systems. Tufano et al. conducted a large scale empirical study on when and why traditional code smells are introduced by developers [102]. Authors studied all revisions from 200 representative projects from android, Apache, eclipse using their “HistoryMiner“ tool. They identified traditional code smell introducing commits using their own heuristics and DECOR smell detection tool. The authors identified 9164 commits that are origins to the detected smells. The results show that a large portion of the smells is introduced in the first version of the classes. Furthermore, the result showed that smells are mainly introduced for the purpose of feature enhancement and less than 16% as a result of bug fixing. Most of the smells were introduced one month before the official release dates and by experienced developers compared to beginners.

The evolution of God class smell was also studied by Vaucher et al. [103]. In this study, the authors investigated the life cycle, prevalence, circumstances behind introduction of those smells and their evolution over time. The aim of the study was to help distinguish God classes that are introduced by accident from God classes introduced as a result of strict design requirements.

Palomba et al. analyzed the co-occurrence of 13 traditional code smells in 395 releases of 30 open source projects [17]. The result shows that 59% of the classes contained more than one traditional code smells together. Some of those smell pairs such as Message Chains and Spaghetti code tend to co-occur more frequently. In addition, the result showed that in most cases the co-occurring smells are removed together as a result of maintenance activities.

Johannes et al. [11] studied the evolution and impact on fault-proneness of 12 types of

JavaScript code smells in 1807 releases of 15 client-side and server-side projects. They conducted survival analysis on the smells considering co-founding factors such as code size. The result showed that smells such as Variable Re-assign tend to survive longer in the evolution of the subject systems. Those smells were often introduced when the file is introduced and persisted for a long time.

Abidi et al. [104] conducted an emprical study on the prevalence and impact of multi-language design smells on fault-proneness. They analyzed 98 releases of nine open-source JNI projects and found that multi-language design smells are prevalent, persistent and bug-prone.

3.4.1 Prevalence and evolution of SATDs

Potdar and Shihab [7] used source-code comments to study self-admitted technical debt in four large open-source software projects. They found that different types of self-admitted technical debts exist in up to 31% of the studied project files. They showed that developers with higher experience tend to introduce most of the self-admitted technical debt and that time pressures and complexity of the code do not correlate with the amount of self-admitted technical debt introduced. They also observed that between 26.3% and 63.5% of self-admitted technical debt are removed from the projects after their introduction.

Bavota and Russo [1] conducted a differentiated replication of the work of Potdar and Shihab [7]. They considered 159 software projects and investigated the diffusion (prevalence) and evolution of self-admitted technical debt and its relationship with software quality. Their results show that (1) SATD is diffused in software projects; (2) the most diffused SATDs are related to code, defect, and requirement debt; (3) the amount of SATD increases over time due to the introduction of new SATDs that developers never fix; and (4) SATD has very long survivability (over 1,000 commits on average). They also proposed a SATD taxonomy, which is used as a base for this work. We extended their taxonomy by identifying data-access-specific SATDs.

Kamei et al. assessed ways to measure the interest of SATDs as a function of LOC and fan-in measures [105]. They examined JMeter as a case study and manually classified its SATD comments, then compared the metric values after the introduction and removal of SATDs to compute their interest. They found that up to 44% of SATDs have positive interest implying that more effort is needed to resolve such debt.

Maldonado et al. [106] studied the removal of SATDs on five open-source projects. They relied on NLP based approach to detect SATDs in the subject systems. The results show that majority of SATDs were removed by the same developers who introduced them. The

median survival of SATDs was 18 to 172 days. The results from the developer survey showed that SATDs are used by developers to track places where code improvement is needed and implementations that may have potential bugs in the future.

Zampetti et al. performed a quantitative and qualitative study of how developers address SATDs in five Java open source projects [35]. They found that a relatively large percentage (20%–50%) of SATD comments are accidentally removed while entire classes or methods are dropped. Developers acknowledged in commit messages the SATD removal in only 8% of the cases. They also observed that SATD is often addressed by specific changes to method calls or conditionals, not just complex source code changes. Like Zampetti et al., we utilize the information obtained from commit messages to understand why data-access SATDs are introduced or removed.

3.4.2 Prevalence and evolution of SQL code smells

Sharma et al. [29] investigated database schema quality on 357 industrial and 2568 open source projects. They deployed the DbDeo tool to extract embedded SQL statements in source code and detect smells. The extraction and smell detection of SQL queries is based on regular expressions. This study is focused on database schema smells. Sharma et al. found that index abuse smell is the most prevalent smell and Adjacency list smell was more prevalent in industrial projects compared to open-source projects. This work did not consider data access and manipulation queries in their studies. However, they mentioned it as future work.

Filho et al. [30] conducted an exploratory study on the prevalence and co-occurrence of bad smells in PL/SQL (Procedural Language for SQL) projects. They analyzed 20 PL/SQL open-source projects using the Manduka code analyzer tool. Manduka tool can detect SQL code smells using a rule-based approach. They used correlation tests and the Apriori algorithm for association rule mining. The result showed that not all smells are equally prevalent in the subject systems and some database smells tend to co-occur together. Although this study is the first study on the prevalence of SQL code smells, the subject systems are small PL/SQL projects and hence this work did not address the prevalence of SQL code smells in queries that are embedded into application codes.

Besides the aforementioned works, the prevalence and impact of SQL code smell on DI systems are not well explored. Particularly the following important aspects of SQL code smells in DI systems were not studied. The first aspect is the diffusion or prevalence of SQL code smells in real-world applications which can provide potential insights into what smells need more attention from developers considering their application domain or context.

It also guides future research on indicating the focus areas to propose useful SQL smell refactoring approaches. The second aspect is studying the co-occurrence of SQL code smells with traditional code smells which could provide insights into the interaction of traditional code smells and SQL code smells and if refactoring approaches to fix traditional code smells affect SQL code smells and vice versa.

Studying the evolution of SQL code smells is another extension of the state of the art as it provides insights into developers' awareness of SQL code smells and their tendency to refactor them as the projects evolve.

3.5 Impacts of technical debts on software quality

In this section, we present related work on the impacts of traditional code smells and SATDs on software quality.

3.5.1 Impacts of traditional code smells on software quality

There are many empirical studies on the impacts of traditional code smells on different aspects of software quality including maintenance, fault proneness, change proneness, and performance. A recent literature review on the effects of traditional code smells was conducted by Santos et al. [101]. The authors collected 3739 conference and journal papers that were published between 2002 and 2017 and filtered 64 papers for final analysis. They categorized papers related to traditional code smells into those dealing with development issues, Human aspects, Programming, and Detection. We focus on papers related to traditional code smell detection approaches and empirical studies on the prevalence, and evolution of traditional code smells, and their correlation with various software quality aspects.

The co-occurrence of traditional code smells with software systems quality was studied in the work of Fontana et al. [19]. They studied 68 projects. Firstly, they studied the prevalence by dividing the projects into data visualization, software development, Application software, and client-server software. Their result shows that traditional code smells are equally prevalent in those domains. Some of those smells such as God class, duplicate code, and data class were the most prevalent. Furthermore, they found that the smells often co-relate with software metrics that measure software quality attributes.

Sjoberg et al. [20] conducted a controlled study to investigate the impact of traditional code smells on maintenance efforts using 6 professional software developers. Maintenance effort was measured in terms of the time taken to complete a maintenance task. The result showed that none of the studied smells show significant association with maintenance effort but other

metrics such as code size contribute to variation in maintenance effort.

Yamashita et al. [21] Studied how interactions between code smell impact software maintainability. They hired developers to implement a change request on subject systems and measured their maintenance effort comparing smelly and non-smelly artifacts. They found that the co-occurrence of code smells in artifacts affects the maintainability of software.

Traditional code smells are sometimes associated with bugs in addition to maintainability issues. For instance, the impacts of God class smells on software quality in terms of maintainability and error-proneness were investigated by Zazworka et al. [107]. The results of their study show that God class traditional smell has a positive correlation with both change proneness and error-proneness. In another study, Li et al. [18] investigated the impact of traditional code smells on error probability. The authors analyzed the post-release evolution of eclipse as a sample object-oriented system. They found that God Class and God Method smells were positively associated with error proneness of classes in the subject systems.

Khomh et al. [8] Studied the impacts of 29 code smells in 9 releases of Azureus and 13 releases of Eclipse as subject systems. They measured change proneness as the number of changes between releases. The result shows that classes with a higher number of smells are more change-prone and some smells are more change prone than others.

Traditional code smells may also cause performance problems in performance-critical applications. Hecht et al. studied the performance impacts of code smells in android applications [108]. They studied Internal Getter/Setter, member Ignoring Method, and HashMap usage using two open-source android applications as subject systems. They measured user interface performance (Frame time and number of delayed frames) and memory usage-related performance. The study showed that from 4 to 12 percent performance improvement was obtained by refactoring such smells.

In another study, Morales et al. [109] proposed an energy-aware refactoring approach (EARMO). They investigated smells on mobile applications and refactoring. They also evaluated EARMO and obtained a precision of 84 %. The authors also showed the impact of the smells on power consummation by comparing the power usage before and after refactoring.

The studies show that code smells have considerable impacts on software quality factors such as maintainability, change proneness, and even contribute to defects in some cases. Traditional code smells also negatively affect performances in terms of execution time and energy usage.

3.5.2 Impacts of SATDs on software quality

There are few studies on the impacts of SATD on software quality. Wehaibi et al. [110] Studied the relation between SATD and software quality in terms of defects and maintenance effort. SATD's were identified using pattern-based approaches. They studied 5 popular open-source projects and found that the defectiveness of files with SATDs increased after the SATDs were introduced and SATD changes are more difficult to perform.

Kamei et al. assessed ways to measure the interest of SATDs as a function of LOC and fan-in measures [105]. They manually classified the comments in the subject system as SATD or not. They compared the metric values during the introduction and removal of SATDs to compute the interest. They computed SATD interest in JMeter project as a case study and found that up to 44% of SATDs have positive SATD interest implying that more effort is needed to resolve SATDs.

The impacts of data-access SATDs on software quality is not well explored in the current state of the art. Exploring SATDs related to data access can provide insights into the patterns and anti-patterns of data access in particular and the overall quality of DI systems in general. The work of Wehaibi et al. [110] and Zampetti et al. [35] motivated us to investigate the circumstances behind the introduction, evolution, and removal of data-access SATDs as such factors affect the interest of the technical debt. We are interested in generalizing the findings of [35] to the context of data-intensive systems.

3.6 Refactoring practices in traditional software systems

Many empirical studies explored the prevalence, nature, co-occurrence, and impact of refactoring activities on software quality. Since Fowler proposed a catalog of refactoring types [13], there have been many studies on refactoring activities.

Silva et al. [111] surveyed open-source developers to identify the motivations behind applying refactoring and found that refactorings are not motivated by code smells. They are rather motivated by changes in software requirements such as bug fixing and feature enhancement.

Chávez et al. [112] studied the impact of refactoring activities on internal quality attributes such as cohesion, coupling, complexity, and inheritance and found that 65% of refactoring instances improved the associated internal quality attributes.

Ferreira et al. [113] analyzed 20,689 refactoring instances from 5 open-source projects to study the relationship between refactoring activity and bugs and found that code elements involved in floss refactoring are more bug-prone compared to root canal refactoring.

Mahmoudi et al. [114] conducted an empirical study to investigate the impact of refactoring activities on merge conflicts using 3000 java subject systems and found that 22% of the refactoring instances were involved with merge conflicts.

Vassallo et al. [22] studied 200 open-source projects belonging to different java ecosystems and showed that the type of refactoring operations applied by developers depends on the support of development environments. Furthermore, they showed that planning for refactoring activities is done based on the age of the software component and proximity to the software release.

Peruma [23] explored refactoring activities in android applications and found that *rename attribute* is the most common refactoring in android applications. They also found that the overall motivation of refactoring is quality improvement by exploiting refactoring commit messages.

Iammarino et al. [24] studied the co-occurrence of refactoring activities and SATD removals using a curated SATD dataset and Refactoring Miner tool and found that refactorings are more likely to co-occur with SATD removal commits than with other commits, however, in most cases, they belong to different quality improvement activities rather than part of the SATD removal. Rename refactorings are specifically studied given the importance of such refactorings on program comprehension.

Peruma et al. [25] analyzed 524,113 rename refactorings and found that most rename refactorings narrow the meaning of the identifiers for which they are applied. In another study, Peruma et al. [26] found that rename refactorings are preferred by less experienced developers and that developers frequently change the semantic meaning after rename refactoring. They also investigated the co-occurrence of rename refactorings with other types of refactorings and found that there are some refactoring types such as *Move Class, Extract Method and Move Attribute* and significant portions of rename refactorings are associated with a change in variable type.

AlOmar et al. [115] conducted a large-scale empirical study involving 111,884 refactoring commits from 800 open source java projects. Their result demonstrated that fixing code smell is not the main driver for refactoring. Based on their analysis of the commit messages, they found that Bug fixing shares 24.3% and functional requirement shares 22.3%.

While several aspects of refactoring activities in traditional software systems are well investigated, to the best of our knowledge, we did not find studies on refactoring practices in data-intensive systems and if/how data-access technical debts are addressed by the applied refactorings.

3.7 Chapter summary

In this chapter, we discussed literature on specification of technical debts, their detection and refactoring approaches, characterization and impact analysis of technical debts. We also discussed literature on refactoring practices in traditional software systems. While several aspects of traditional technical debts are well investigated in the literature, several aspects of data-access technical debts are not yet explored. In this research, we complement existing studies on specification of data-access technical debts and investigate the characterization, impact analysis of such debts. We also explore refactoring practices in data-access classes, to understand if/how data-access technical debts are addressed via refactoring.

CHAPTER 4 STUDY DESIGN

4.1 Chapter overview

In this chapter, we describe our study design to answer all research questions in this dissertation. We describe the details of the subject systems, data collection, and extraction method to generate datasets utilized in the specification, characterization, and impact analysis of data-access technical debts and the analysis of data-access refactoring practices in data-intensive systems.

4.2 Characterization and impact analysis of SQL code smells

In this section, we describe the process we follow to select the study subject systems, detect traditional and SQL code smells within their source code, and extract bug-fixing and bug-inducing commits from their version history. This dataset is utilized to answer **RQ 2.4, RQ 2.5, RQ 2.6** discussed in chapter 7 and **RQ 3.1 discussed in Chapter 8**. Figure 4.1 shows the overview of the study method for characterization and impact analysis of SQL code smells.

4.2.1 Selection of subject systems

We limited our subject systems to Java applications because the SQL code smell detection tool that we selected for our study, SQLInspect [31] can only process programs written in Java. In addition, SQLInspect can only detect SQL code smells, and hence we only considered SQL-based software systems to generate the code smell dataset. We identified the final list of subject systems using the following selection process.

Phase-I: We use GitHub search mechanism and collect the software repositories labeled with four keywords – `android app, hibernate, JPA, Java`. We choose these keywords (a.k.a., categories) since we were interested in data-intensive software systems and also wanted to study SQL code smells in their embedding code.

Phase-II: We performed a code search on each project selected in the first phase using GitHub code search API [116]. In particular, we look for the import statements (e.g., `import android.database .sqlite.SQLiteDatabase`) that SQLInspect can analyze to detect potential SQL code smells.

Phase-III: Once Phase-I and Phase-II are completed, we collect the projects that (1) fall

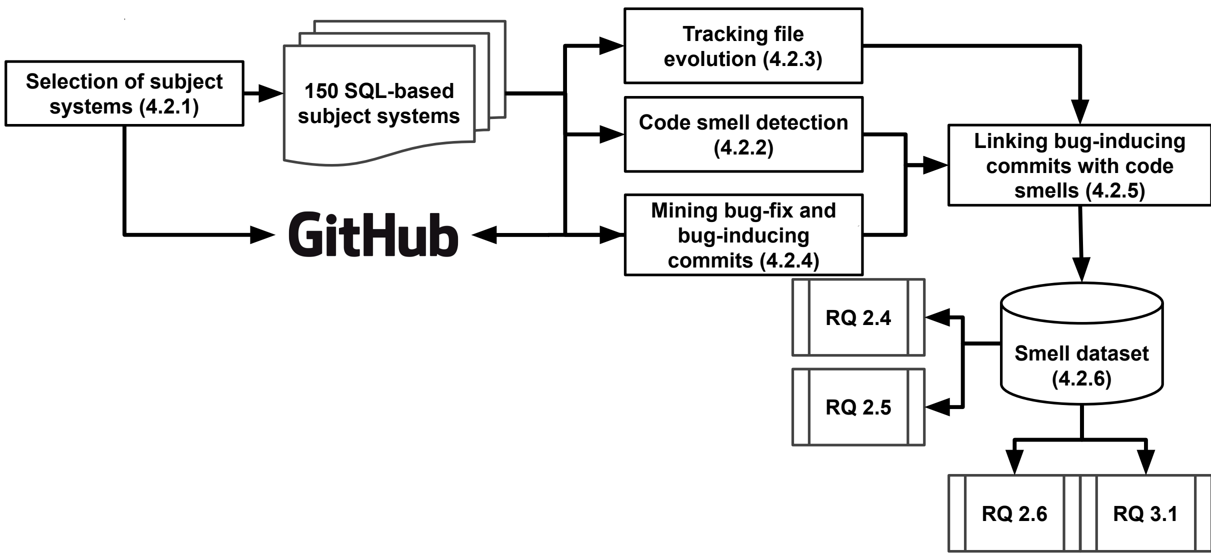


Figure 4.1 Overview of the study method for characterization and impact analysis of SQL code smells

into the four categories above and (2) pass the constraint of import statements in their source code.

Phase-IV: Since the project collection in the above three phases was not significantly high, we thus collect all the labels from each project and build a word-count dictionary to identify the most common keywords. Then we select Top-50 keywords from each of the four categories and repeat Phase-I, which delivers a large collection of 35,000 projects. Then we look for import statements in their source code again and separate 800 projects that contained the required import statements.

Phase-V: We ran the SQLInspect tool on 800 projects and selected the projects with at least 10 database access queries. We choose this threshold to capture the projects that vary in size and complexity and to obtain a dataset with a significant number of queries for analysis. Finally, we ended up with a total of 150 data-intensive software projects. On average, each

project has a size of 146 KLOC, 121 SQL queries and 15 data-access classes. Overall, 13% of these projects have more than 500 KLOC and 30% of them use more than 73 SQL queries. About 48% of SQL queries in those projects perform SELECT operations, whereas 11% have sub-queries.

We also classify the selected SQL subject systems into four application domains – *Business, Library, Multimedia* and *Utility* – to capture the domain-related aspects.

We assign each project to any of these four groups by consulting their overview on the GitHub pages.

Software projects that are used for business and educational purposes (e.g., data analysis) are kept in the *Business* category. Open-source libraries or tools used by developers are categorized into the *Library* category. Games and media player systems are categorized under *Multimedia*. Finally, software projects for personal uses (e.g., task management, scheduling or social networking) are categorized into the *Utility* category.

Table 4.1 shows, for each application domain, the total number of projects and their median number of database access queries. The median is calculated by considering the latest version, at the time of data collection, of all selected projects.

Table 4.1 Selected projects and their database access statistics. **DAQC** = Database Access Query Count

Application domain	**# Projects**	**Median DAQC**
Library	97	32
Business	23	46
Utility	19	50.5
Multimedia	11	19.5

4.2.2 Code smell detection

It is not practical to detect code smells from every commit of each project due to many projects and commits. Therefore, we detect the traditional and SQL code smells from each project by taking their snapshots after every 500 commits, starting from the most recent commits backward. A similar approach was followed by Aniche et al. [117].

We use SQLInspect [31], a static analysis tool, for SQL code smell detection. SQLInspect extracts SQL queries from the Java code and then detects four types of SQL code smells that are Implicit Columns, Fear of the Unknown, Random Selection, and Ambiguous Groups. The tool can detect smells from the SQL code targeting several database access frameworks

– *Android Database API*, *JDBC*, *JPA*, and *Hibernate*. SQLInspect relies on SQL query extraction, which has a minimum precision of 88 % and a minimum recall of 71.5% [33]. Hence, the aforementioned precision and recall values can be considered as an upper bound for SQL Code smell detection performance. More details on SQLInspect and the supported smells can be found in the related papers of Nagy et al. [31, 32].

We use DECOR [118], a reverse engineering tool, for detecting the traditional code smells. DECOR can detect 18 different traditional code smells from Java source code. DECOR has a recall of 100% and a precision > 60% [119].

4.2.3 Tracking project file evolution

Software projects change, and so are their source code files as they evolve over time. To ensure a reliable analysis of software evolution, file genealogy tracking is important. Tracking file status can help us resolve issues involving file renaming or file location changes during evolution. We use the `git diff` command to compare two consecutive project snapshots using their commit identifiers. The command shows a list of files that are either added, deleted, modified, or renamed between two given commits. It also provides a numerical estimation of how likely a file has been renamed. We consider a threshold of 70% accuracy to detect file renaming, as was used by an earlier study [11]. Finally, each source file in each of our projects is tagged with a unique identifier generated from the file tracking information.

4.2.4 Mining Bug-fix and Bug-inducing commits

We use PyDriller [120] to mine bug-fixing and bug-inducing commits from our selected projects. PyDriller offers a Python API that interacts with any GitHub repository using a set of Git commands. To identify bug-fix commits using PyDriller, we employed a set of 57 keywords that indicate possible fixing of bugs, errors, and software failures (e.g., **fix, fixed, fixes, bug, error, except, issue, fail, failure, crash**). The set of keywords was selected based on the work of Mockus and Votta [121] and Antoniol et al. [122], who showed that those keywords tend to be associated with bug-fix commits. These keywords were also used in multiple previous studies to identify bug-fixing commits [123–125]. The complete keyword list is available in the replication package [126]. Our tool searches for each keyword in the commit messages, and separates the commits containing the keywords as bug-fixing commits.

Table 4.2 shows the proportion of bug-fix commits that are identified using the top six prevalent keywords. PyDriller implements the SZZ algorithm [127] to pinpoint a bug-inducing commit from a given bug-fix commit within the version-control history. We use PyDriller to

detect the bug-inducing commits for the bug-fixing commits detected above.

Table 4.2 Most prevalent keywords used to detect bug-fix commits

Keywords	**Bug-Fix Commits**
fix, fixed, fixes	66.16%
bug	7.93%
issue	6.16%
except	4.84%
error	4.51%
fail, failure	3.55%
Total bug-fix commits	**110,747**

4.2.5 Linking Bug-inducing commits with code smells

To determine any association between code smells and software bugs, the smells have to be present in the code before the bugs occur. We determine such potential causal associations using bug-inducing commits. Let T_0 be the snapshot date of the smelly code file and T_n be the commit date of the next snapshot that tracks the same code file. Now, we identify the bug-inducing commits between T_o and T_n that contain the smelly code file from version T_o. If any bug-inducing commit touches the smelly file which is later fixed in the corresponding bug-fixing commit, then we mark such smells as linked with the target bug-inducing commits.

4.2.6 Construction of a smell dataset

To perform our analysis reliably, we store the information extracted from the earlier steps in a relational database. A record in the smells' table of our database is identified using a combination of file identifier and project version number (a.k.a., file-version-ID). Each record comprises a vector that stores the statistics on traditional code smells, SQL code smells found within a source code file, and its bug-inducing related metadata. Our database contains a total of 1,077,548 records for 139,017 source files from 150 projects with 1648 versions. However, our study analyzes only such records where the source code files deal with database access, and might contain SQL code smells. Thus, in practice, we deal with a subset of 29,373 records for our study.

4.3 Specification and characterization of data-access SATD

In this section, we describe the process we followed to identify subject systems and extract SATDs to prepare the SATD dataset utilized to answer research questions **RQ 1.1** discussed in Chapter 5, and **RQ 2.1, RQ 2.2 and RQ 2.3** discussed in Chapter 6. Figure 4.2 shows the overview of the study method.

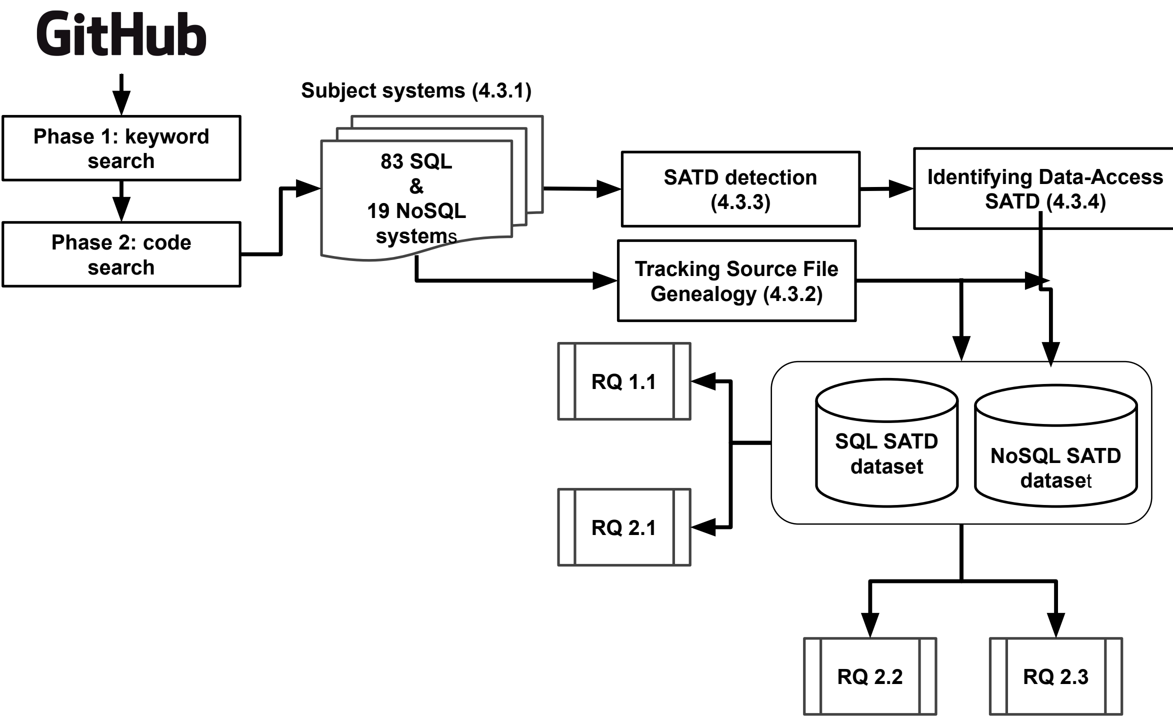


Figure 4.2 Overview of the study method for Specification and characterization of data-access SATDs.

4.3.1 Subject systems

To generate the SATD dataset, we utilized both SQl-based subject systems and NoSQl-based subject systems. The process we followed to identify SQL based subject systems is described in Section 4.2.

To identify NoSQL-based subject systems, we first collected NoSQL database management

systems popular in open-source projects such as MongoDB, Redis, Riak and Neo4J. The database systems are collected from the supported databases of Eclipse JNoSQL,[1] a popular Java framework in the Eclipse ecosystem that streamlines the integration of Java applications with NoSQL databases. At the time of data collection, JNoSQL supported around 30 databases. We ran a GitHub search for projects mentioning these database engines. To avoid "toy" projects, we set the following criteria for the projects: (1) they had to have at least one open issue, (2) more than 1,000 commits, and (3) at least one recent commit within one year from the time of data collection (*i.e.,* 2020). We also applied similar quality filtering to the SQL-based subject systems.

we compiled a list of import statements that are used to access NoSQL persistence systems, *e.g.,* `com.mongodb.MongoClient`, `org.neo4j.driver`, `org.apache.hbase`. To determine the imports, we started with the list of supported NoSQL databases from JNoSQL. For each database, we explored code snippets provided as instructions to connect that database to Java applications. We collected the import statements mentioned in such snippets to compile the list of NoSQL import statements. We ran GitHub code search on the identified projects and counted the import statements for each system. We were finally left with 83 SQL-based and 19 NoSQL-based subject systems satisfying our criteria.

4.3.2 Tracking source file genealogy

We followed a similar procedure described in Sub-section 4.2.3 to track file genealogy of both the SQL-based and NoSQl-based subject systems.

4.3.3 SATD detection

Due to the large number of subject systems with many commits, we took snapshots of each system's every 500th commit. Another approach would have been to select only a few projects and study every commit of each subject system. However, our research is exploratory and, therefore, we emphasize the generalizability of our results and conclusions.

We used the SATD detection tool by Liu et al. [83]. This tool uses a machine learning-based detection approach that combines the decisions of multiple Naive-Bayes classifiers into a composite classifier using a majority vote. During the tool's training phase, the source codes comments are represented using vector space modeling (VSM), and features are selected from VSM using information gain. The details of their approach are discussed in [82]. The tool has a Java API as well as an Eclipse plugin to support developers. Given a source code

[1]http://www.jnosql.org/docs/introduction.html

comment, the tool returns a boolean indicating whether it is a SATD comment or not. We chose this detection tool because it has the highest accuracy (average F_1 score of 73.7%) among different approaches, and the rest of the approaches were not realized as a tool to the best of our knowledge.

SATD detection was carried out in two phases. In the first phase, we extracted the comments of each snapshot of all the projects using srcML.[2] SrcML initially converts the source code into XML format. The comments can then be identified by running XPath queries. In the second phase, we run the SATD detection on the identified comments. The output of the SATD detection tool is binary: it classifies the comment as SATD-related or not.

4.3.4 Identifying Data-Access SATD

We relied on SQL and NoSQL database import statements to identify data-access classes in both subject systems. We considered a class with at least one SQL/NoSQL database access import statement as a data-access class. To identify such classes, we ran a code search using the `egrep` command on all studied snapshots of the projects. A SATD comment that belongs to any of the identified data access classes is considered a data-access SATD.

4.3.5 SATD dataset construction

We built two SATD datasets corresponding to SQL and NoSQL subject systems. A row in each dataset contains *fileId*, *version*, *commentId*, *projectName*, *commentMessage* and *isDataAccess*. The *version* attribute is needed because we study multiple versions of each subject system. Overall, our dataset contains 35,284 unique comments from SQL subject systems, out of which 4,580 are from data-access classes. Our dataset also contains 2,386 unique comments from NoSQL subject systems, out of which 436 are comments from data-access classes.

4.4 Specification and criticality analysis of data-access performance anti-patterns

In this section, we describe the approach we followed to identify NoSQL-based and polyglot subject systems, to extract issues and filter data-access performance issues, and prepare the **data-access performance issue dataset** utilized to answer **RQ 1.2** discussed in Chapter 5 and **RQ 3.1** discussed in Chapter 8. Figure 4.3 shows the overview of the study method.

[2]https://www.srcml.org/

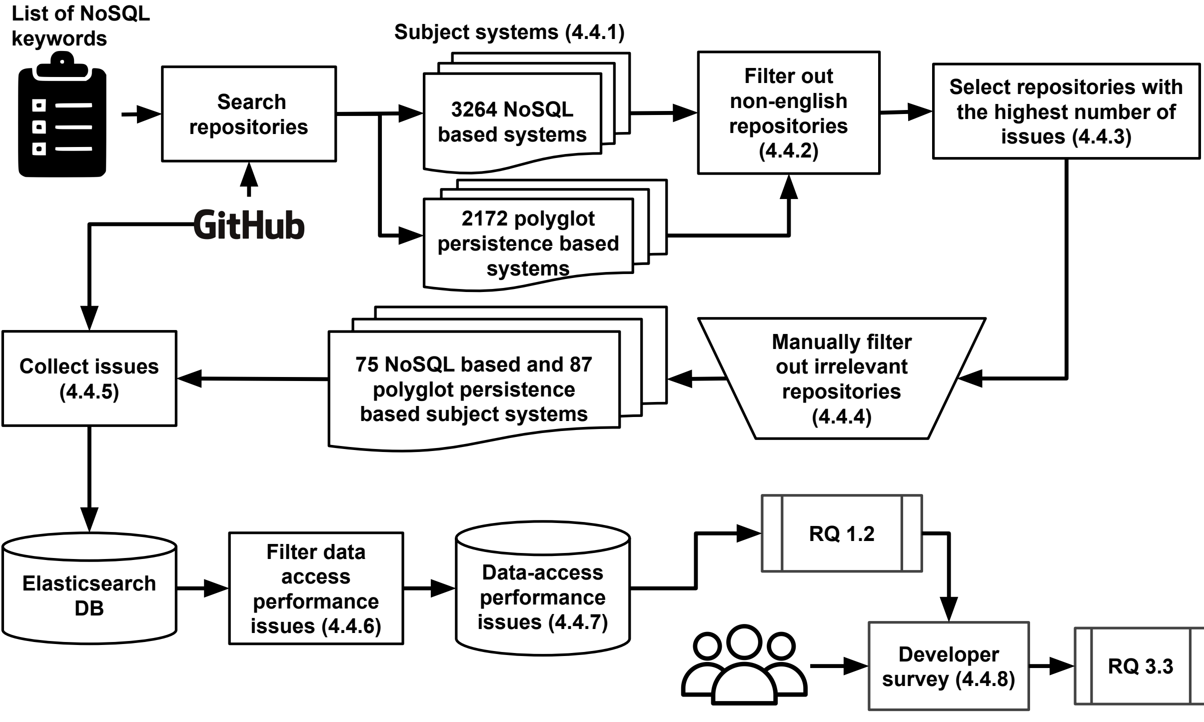


Figure 4.3 Overview of the study method for specification and criticality analysis of data-access performance anti-patterns

4.4.1 Subject systems

We followed a similar approach to identify NoSQL-based systems described in 4.3, we conducted a repository search using GitHub rest API V3 and NoSQL import statements, and we looked for repositories that mention at least one of the import statements, corresponding to the NoSQL databases in our list, in their title, description, or README file. To avoid tutorials and toy projects we set the search criteria to consider active repositories (whose latest push is not older than one year from the data collection date), repositories that are not mirrored to other repositories, repositories whose code size is at least 100 KB and repositories with at least two stars [128].

We ran the repository search on March 1, 2022, and we found 20340 candidate repositories before applying the quality filters and 3264 remaining after applying the quality filters. The subject systems utilize different programming languages, including but not limited to Java

and JavaScript.

For identifying polyglot persistence data-intensive systems, we started with the work of Benats et al. [128] where they investigated the usage of multi-database models using projects collected from Libraries.IO[3]. They assessed the popularity of different database models including polyglot persistence models. As part of their replication package, they released the database of projects and their corresponding usage of relational and key-value, wide-column, document-oriented, and graph-based NoSQL models. In particular, we used two tables from the provided database, the first one is a view (FILTERED_REPOSITORIES_VIEW) that provides the list of repositories after quality filtering (projects having at least 100 KB code size and at least two stars). This list contains 42176 projects. The second table (SQL_NOSQL_REPOSITORY_WITH_DBMS_TYPE) provides the usage of the aforementioned database models in all projects. Combining the two, we obtained the database model usage of all the 42K systems. Since our goal is to identify systems that use polyglot persistence databases, we selected repositories that have at least two database models resulting in 6877 polyglot persistence-based applications. Once we collected the metadata of each repository, we removed repositories whose latest push is older than one year from the data collection date, similar to the criteria of NoSQL subject systems. We finally ended up with 2172 polyglot subject systems.

4.4.2 Filter non-English repositories

We observed that the description and readme files are not written in English for some repositories. We filtered out repositories whose descriptions are not written in English using Langdetect[4] python library. After the language filtering, 2498 NoSQL and 1604 polyglot persistence-based repositories remained.

4.4.3 Select repositories with the highest number of issues

Since analyzing all issues from the 4102 repositories is not feasible and since there is no automatic way to measure the relevance of the systems, we need to restrict the number of subject systems by sorting the systems in decreasing order of the number of issues and picking the top projects. We selected the top 150 NoSQL based and 150 polyglot persistence-based systems as candidate subject systems.

[3]https://libraries.io/

[4]https://pypi.org/project/langdetect/

4.4.4 Manually filter out irrelevant repositories

We manually went through each of the 300 subject systems' repositories on GitHub and investigated the description, code, README, and issues to understand their functionality and relevance to our study as data-intensive systems. We also filtered out repositories with non-English language descriptions missed by the Langdetect. After this filtering, the remaining 75 NoSQL-based data-intensive systems and 87 polyglot persistence-based data-intensive systems were selected as subject systems, 162 in total.

4.4.5 Collect issues

For each of the 162 subject systems, we collected all issues from their GitHub repositories using PyGithub[5], which provides a python wrapper for GitHub REST API[6]. For each issue, we collected the title, body, issue number, URL, state, creation time, comments, labels, and closing time if the issue is closed. Since we want to investigate the solutions proposed to mitigate the issues, we are only interested in closed issues. We collected a total of 526672 closed issues from NoSQL-based systems and 257840 closed issues from the polyglot persistence-based systems and stored them in the Elasticsearch database. Table 4.3 provides the mean, standard deviation and the five-number summary of the number of stars, number of forks, code size, and number of closed issues for our candidate subject systems.

Table 4.3 Distribution of repository level metrics for NoSQL and Polyglot subject systems

Metric	Group	Minimum	25%	Median	Mean	75%	Maximum	std
Stars	Polyglot	5	71.5	508	4616.91	2896	61167	10747.29
	NoSQL	7	405	2229	6083.81	9043.5	58918	9880.86
Forks	Polyglot	0	29	127	1157.80	621	36393	4072.99
	NoSQL	1	174	428	1318.23	954	21435	2863.24
Code size (Kb)	Polyglot	744	7912	22,727	130741.72	138677.5	1719944	297229.78
	NoSQL	1126	8515	23,010	132062.04	85,238.50	3836513	458862.46
Closed issues	Polyglot	30	476	1028	2963.41	2163.5	33914	5541.40
	NoSQL	303	1001	1936	6761.79	4366	128001	18274.97
Age (years)	Polyglot	2.32	4.56	6.09	6.4	8.3	12.99	2.4
	NoSQL	1.4	4.71	6.97	7.1	9.52	13.76	3.03

4.4.6 Filter data-access performance issues

Similar to the work of Shao et al. [12], we use the following heuristics to identify issues related to data access performance. We constructed an Elasticsearch query string using keywords

[5]https://pygithub.readthedocs.io/en/latest/index.html
[6]https://docs.github.com/en/rest

associated with performance (performance, slow, timeout, sluggish), keywords associated with data access (read, write, fetch, update, delete, load), and database-related keywords (data, database, query, schema, index, cache, table, partition, document) connected by AND operator. The query is formed in multiple rounds by examining the returned results and modifying the query to minimize false positives. We applied the query against the issue title and the issue body. Listing 4.1 shows the final query we used to filter data access performance issues where "|" represents OR operator and " + " represents AND operator. We obtained 3760 issues from the NoSQL based systems and 2645 issues from the polyglot persistence-based systems and prepared the data-access performance issues dataset.

```
"simple_query_string":{
  "query":"(performance | timeout | slow | sluggish) + (read | write | fetch
     | update | delete | load)+ (data | database | query | schema | index |
   cache | table | partition | document)",
  "fields":["title","body"]
}
```

Listing 4.1 Elasticsearch query to identify data access performance issues

4.4.7 Data-access performance issues dataset

Each issue in the data-access performance issues dataset is identified by its unique issue id and contains the title, body, the associated GitHub issue URL, the repository name, the relevance score obtained from the Elasticsearch query, and the issue type as polyglot or NoSQL.

4.4.8 Survey on data-access performance anti-patterns

In this Subsection, we describe the procedures we followed to conduct a survey on the criticality of data-access performance anti-patterns to answer RQ 3.3 reported in Chapter 8. We describe the recruitment of survey participants and the details of the survey questionnaire.

Recruitment of survey participants

Our inclusion criteria are that developers should have at least one year of back-end or full-stack software development experience. We utilize the convenience sampling method to recruit survey participants. We recruited participants from two sources. We extracted the name and emails of authors who contributed to the development of NoSQL-based and polyglot

subject systems, from which we extracted data-access performance issues. We sent the survey to 4470 valid email addresses associated with the subject system contributors.

As a second source, we also utilized LinkedIn[7] as a platform to recruit survey participants. We first searched for "backend development" and "fullstack development" in Linkedin. Next, we manually went through the profiles of the matched participants to make sure their experience is relevant to our study as per the inclusion criteria. Since it is not possible to directly send messages to LinkedIn users before connecting in the free version, We first sent a connection request to the profiles that pass the selection criteria and sent the survey link to the ones that accept the connection request.

Survey questionnaire

The survey questionnaire contains ten sections. The first section provides the introduction to the survey and details the information and consent form. It finally asks respondents if they accept to participate in the survey.

Sections two to section nine correspond to the number of high-level anti-pattern categories shown in Figure 5.2. At the beginning of each section, we described the high-level anti-pattern categories. Next, we outlined the name of each anti-pattern, its description, criticality rating, optional justification/comments, and question about fixing strategy "*if you encounter this anti-pattern, how do you fix it?*" Figure 4.4 shows the sample survey questions about *Duplicate requests* performance anti-pattern. We repeated similar questions for each of the 14 newly identified performance anti-patterns under their respective categories. We added the "I don't know" option for each criticality rating to avoid forcing participants to pick one value when they don't have enough information to rate the criticality.

At the end of the section we provide a chance for respondents to mention a new anti-pattern based on their experience by asking: "Please describe any non-listed design issues related to < high-level anti-pattern category> you encountered reducing data-access performance".

In the last section, section nine, we ask the following demographic questions.

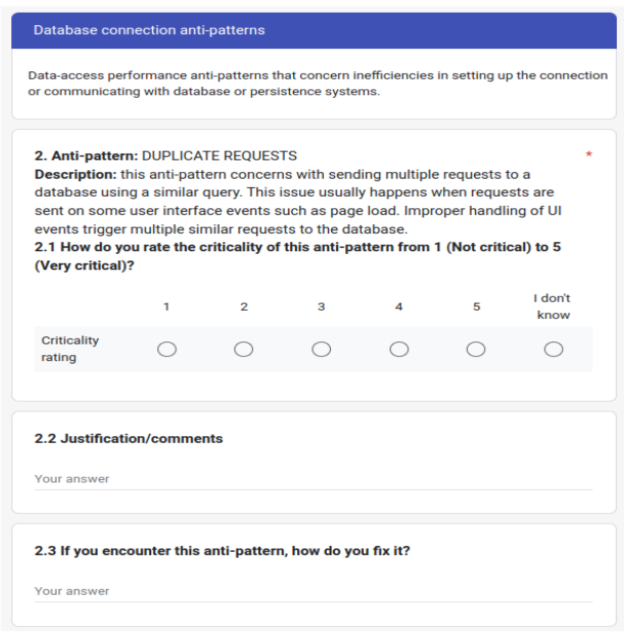
1. What best describes your role in your current organization/projects?

   Options: Software Developer, Product Owner, Manager, and other (respondents can specify their own roles)

2. How many years of experience do you have in software development?

   Options: Less than 1, Between 1 and 5, Between 5 and 10 and More than 10

[7] https://www.linkedin.com/



Database connection anti-patterns

Data-access performance anti-patterns that concern inefficiencies in setting up the connection or communicating with database or persistence systems.

2. Anti-pattern: DUPLICATE REQUESTS *
Description: this anti-pattern concerns with sending multiple requests to a database using a similar query. This issue usually happens when requests are sent on some user interface events such as page load. Improper handling of UI events trigger multiple similar requests to the database.
2.1 How do you rate the criticality of this anti-pattern from 1 (Not critical) to 5 (Very critical)?

	1	2	3	4	5	I don't know
Criticality rating	○	○	○	○	○	○

2.2 Justification/comments

Your answer

2.3 If you encounter this anti-pattern, how do you fix it?

Your answer

Figure 4.4 Example survey question regarding *Duplicate requests* anti-pattern.

3. How many years of experience do you have in backend and database access code development?

   Options: Less than 1, Between 1 and 5, Between 5 and 10 and More than 10

4. Which database access (persistence) frameworks, if any, do you use for development? (open-ended question)

4.5 Refactoring practices in data-intensive systems

In this section, we outline the process we followed to select the subject systems and extract commit information and refactoring dataset. This dataset is used to answer **RQ 3.1** discussed in Chapter 8), **RQ 4.1, RQ 4.2, RQ 4.3, and RQ 4.4** discussed in Chapter 9, and **RQ 4.5, RQ 4.6, RQ 4.7, and RQ 4.8** discussed in Chapter 10. Figure 4.5 shows the overview of the study method.

4.5.1 Subject systems

We utilized both SQL-based systems from Section 4.2 and NoSQL based subject systems from Section 4.3 to study data-access refactorings. However, this analysis requires data-extraction from all snapshots of the subject systems, which is not feasible for 150 SQL-based subject systems due to time and resource constraints. Hence, using the number of SQL queries as a

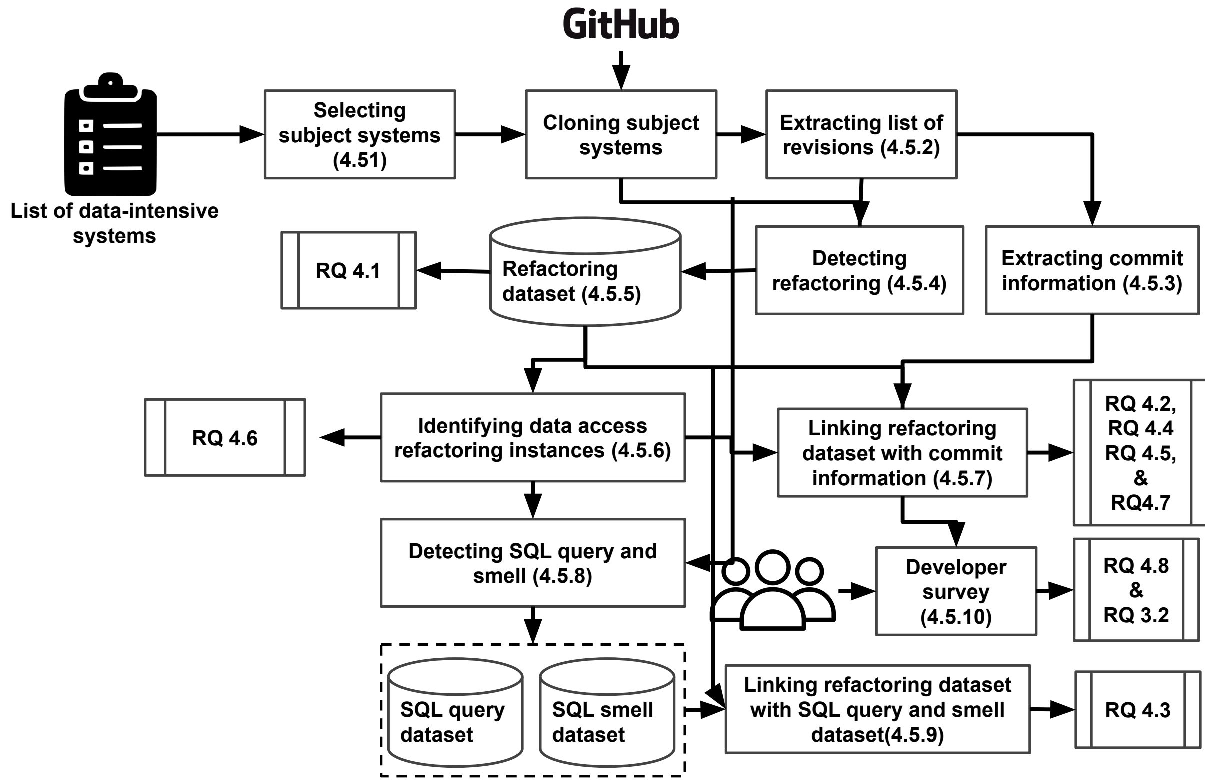


Figure 4.5 Overview of the study method for refactoring practices in data-intensive systems

proxy to pick the most data-intensive systems, we ranked the systems in decreasing order of the number of SQL queries and took the first 12 subject systems. Although *bio2rdf-scripts* [8] project has the highest number of queries, the number of refactoring instances detected was only 6. Hence, we removed this project from our list of subject systems. Table 4.4 shows the summary of our SQL-based data-intensive subject systems. We analyzed 2, 473, 090 refactoring instances from 174776 commits. The systems have an average of 519 queries and 67 data access classes. Furthermore, the subject systems have 206, 091 refactoring instances on average.

Out of the 19 NoSQL-based subject systems identified in Section 4.3, two were not active at the time of data collection (July 2022) and hence we removed them and ended up with 17 NoSQL-based data-intensive systems as subject systems. Table 4.5 shows the summary of our NoSQL-based data-intensive subject systems. We analyzed 189, 205 refactoring in-

[8]https://github.com/bio2rdf/bio2rdf-scripts

stances from 41, 626 commits. The systems have an average of 220.8 data-access classes. Furthermore, the subject systems have 11, 129.7 refactoring instances on average.

Table 4.4 List of SQL subject systems with a number of commits, number of queries, and number of data access files and number of refactoring instances

Project Name	Number of commits	Number of Queries	Number of data access files	Number of refactoring
Eclipse-ee4j/eclipselink	10403	1371	43	80, 311
Adempiere/adempiere	15754	941	365	2,104,700
Appirio-tech/direct-app	3073	876	95	1492
DotCMS/core	17957	740	40	60, 211
Wso2/carbon-apimgt	33174	656	12	32,101
Oltpbenchmark/oltpbench	1110	303	131	2396
Mtotschnig/MyExpenses	9065	287	3	9949
Querydsl/querydsl	7874	249	23	38,065
Wordpress-mobile/ WordPress-Android	59048	221	17	31, 647
AppLozic/ Applozic-Android-SDK	2298	202	6	2136
Xipki/xipki	6328	193	21	88, 623
Deegree/deegree3	8692	190	45	21, 459

Table 4.5 List of NoSQL subject systems with a number of commits, number of data access files and number of refactoring instances.

Project Name	Number of commits	Number of data access files	Number of refactoring
hazelcast_hazelcast-jet	2680	1448	28637
spring-projects_spring-data-elasticsearch	1597	663	9520
hazelcast_hazelcast-simulator	5743	483	15785
pietermartin_sqlg	2371	377	8217
eMoflon_emoflon-neo	1588	303	4027
IHTSDO_snowstorm	2418	124	7422
Flipkart_foxtrot	2861	81	4171
Hurence_logisland	2538	76	5517
spring-projects_spring-data-redis	2592	52	23380
neo4j-contrib_neo4j-apoc-procedures	2090	47	5523
personium_personium-core	1405	31	7266
gisaia_ARLAS-server	1460	21	4087
codelibs_fess	4089	13	48741
OpenSextant_Xponents	1930	13	1887
US-CBP_GTAS	2876	9	4867
RyanSusana_elepy	1729	8	3776
romanchyla_montysolr	1659	5	6382

4.5.2 Extracting list of revisions

After we cloned each subject system, we run the *git rev-list 'branch'* command to get the list of revisions in the default branch of each system. This command gets commit IDs of revisions from recent to oldest for the given branch. We focused our analysis on the default or master branch of each system.

4.5.3 Extracting commit information

One of the independent variables in this study is the commit time. To collect metadata associated with a commit including the committer time for each revision, we used PyDriller [120], a python framework for mining software repositories. This framework provides an API to collect information from a remote or locally cloned GitHub repository. We also marked the commits tagged as official releases by GitHub for each subject system.

4.5.4 Detecting refactoring

There are several refactoring detection tools available such as refactoring crawler (Precision= 85% and Recall= 85%) [93], RefFinder (Precision= 35% and Recall=24%) [95], Refdiff 2.0 (Precision= 93.8% and Recall=76.9%) [98], GUMTREEDIFF 2.1.2 (Precision= 60.1% and Recall=63%) [96] and RefactoringMiner (Precision= 99.6% and Recall=94%) [97]. In our analysis, we used Refactoring Miner for the following advantages. One, it has state-of-the-art average precision of 99.6% and an average recall of 94% [97]. Two, the detection approach does not require code similarity thresholds with default values obtained from empirical studies using relatively small subject systems, as a result, the thresholds could overfit to the specific subject systems and may not generalize to new subject systems. Hence, it is necessary to calibrate the thresholds for new subject systems. This calibration process requires a tedious and time-consuming manual effort, which makes large-scale studies using different subject systems difficult [97]. Three, Refactoring Miner does not require building snapshots before analysis, which greatly reduces the data collection time. Refactoring Miner relies on AST-based statement matching techniques to list refactoring instances between successive revisions in a commit history. We deployed the latest version, 2.2, which supports 85 different types of refactoring instances [9]. Refactoring Miner takes a repository and branch to analyze and generates a JSON document containing all the detected refactoring types on the history of the specified branch.

[9]https://github.com/tsantalis/RefactoringMiner

4.5.5 Construction of the refactoring dataset

We merged the result of the refactoring detection for all systems and build the refactoring dataset. Each row in this dataset contains *repository name, commit ID, refactoring ID, refactoring type, refactoring description, file name, method lines, refactoring lines.* The refactoring ID is automatically generated to give a unique ID for each refactoring instance; the refactoring type takes one of the 85 refactoring types; the refactoring description contains the description of the refactoring generated by Refactoring Miner; the file name contains the list of files associated with the refactoring; method lines contains the list of method line ranges for all the methods associated with the refactoring; refactoring lines contains the list of lines touched by the refactoring, and a Boolean indicating if the refactoring instance is data-access refactoring or not.

4.5.6 Identifying data access refactoring instances

We relied on import statements to identify data access classes in our subject systems. We particularly looked for import statements for SQL projects corresponding to the underlying persistence technologies such as Android SQLite API, JDBC, and Hibernate. The import statements include but are not limited to, `android.database.sqlite`, `android.database.DatabaseUtils`, `org.hibernate.Query`, `org.hibernate.SQLQuery`, `java.sql.Statement`, `javax.persistence.Query`,`javax.persistence.TypedQuery`, `org.springframework`. This approach was used in the work of Naggy et al. [31] who proposed the SQLInspect tool for static analysis and SQL code smell detection. To avoid the possibility of unused import statements in the code, we added another criterion for a data access class to be associated with at least one SQL query. Using this approach, we identified 18,892 refactoring instances as data access refactoring instances.

Similarly, to identify data-access classes in NoSQL-based subject systems, we looked for the import statements corresponding to the NoSQL databases, including but not limited to `com.mongodb.MongoClient`, `org.apache.tinkerpop.gremlin`,`org.elasticsearch.client` and `org.neo4j.driver`. Unfortunately, we don't have a similar tool to SQLInspect for NoSQL databases to the best of our knowledge, hence we relied on the import statements to identify data access and regular refactorings. Using the import statements, we identified 61,182 data-access refactoring instances from the NoSQL-based subject systems.

4.5.7 Linking refactoring dataset with commit information

To answer the second research question, we combined the refactoring dataset and the collected commit information using the commit ID. The commit information contains author time and committer time. However, we used committer time for our analysis to represent the time of a system revision.

4.5.8 Detecting SQL query and smell

To answer **RQ4**, we run SQLInspect [31] on our subject systems, to extract SQL queries and SQL smells. We run SQLInspect on all snapshots that are associated with at least one data access refactoring instance. We obtain a separate dataset for query and smell instances. Each query instance is associated with commit ID, class name, query value, and line number of the query location. Similarly, each smell instance is associated with commit ID, class name, smell type, and location.

4.5.9 Linking refactoring dataset with SQL query and smell dataset

We linked the data access refactoring dataset and the query dataset using line-level matching and method-level matching, respectively. The common criteria for both approaches are that both the refactoring and query instances must be from the same repository, the same snapshot, and belong to the same class. Line level matching is more strict in the sense that the line number of the query should be one of the lines involved in the target refactoring instance. In method-level matching, we match a query instance whose location is inside one of the target methods of the target refactoring instance. We used a similar approach to match the smell dataset with the refactoring dataset. While line-level matching provides a more accurate representation of association, many refactoring instances are applied to a method that contains a query. The method level matching captures the indirect association between refactoring activities and queries or smells.

4.5.10 Developer survey on refactoring practices

We conducted a developer survey to understand practitioners' opinion regarding refactoring practices in data-access classes. In this Sub-section, we outline the details of how we recruited survey participants and the content of the survey.

Recruitment of survey participants

Since we are interested in data-access refactoring practices, we considered developers involved in data-access refactoring as survey candidates (358 developers) from the **refactoring dataset**. We extracted the email address of all the developers from our dataset. Out of the 358 developers emails, only 246 emails were valid. We shared the survey link to the developers with a working email address. We also relied on LinkedIn to recruit more survey participants. To identify relevant survey candidates, we first searched for "backend developer" with the aim to obtain developers with good experience in data-access code development. Next we manually went through the returned profiles as ranked by the system based on relevance, checked their current profession and experience and sent a connection request for the developers with relevant skill and experience to this study. We needed to send a connection request before sharing the survey, as we can't directly send the message without first connecting with participants. We also had to limit to a maximum of 100 connection requests per week imposed by LinkedIn. We shared the survey link for those developers that accepted the connection request. We continued this process for the survey duration of four weeks.

Survey questionnaire

We prepared the survey on Google forms composed of six sections. The first section contains information about the research objective, a description of data-access refactoring, and informed consent statements. The second section asks participants about the context and motivations behind applying data-access refactoring and contains the following questions.

1. *When do you usually perform refactoring in data-access classes (classes that contain code to query the database or to store data on the database)?*

   We provided a multiple choice grid where the rows contain the motivations that are : *as part of a bug fix, When adding new feature, when changing existing implementation and dedicated only to improve the code quality in data-access classes* and the columns provide an option to specify agreement as : *strongly disagree, disagree, neither agree nor disagree, agree, strongly agree, and I don't know*. We added the 'I don't know' option in this and similar subsequent questions to avoid forcing the respondents to pick a choice.

2. *What is your justification for your choice in the previous question?*

3. *Do you have cases other than the previously mentioned ones when you could apply data-access refactorings?*

4. *What do you aim to achieve by refactoring data-access classes?* We provided the following motivations as a checkbox where respondents can select one or more motivations. The motivations are : *Improve the data access performance, fix bad code smells, improve code readability, improve maintainability, and I don't know*

5. *What is your justification for your choice in the previous question? (open-ended question)*

6. *Do you have other motivations to do data-access refactorings that are not mentioned above? (open-ended question)*

Section three asks the survey respondents about when refactoring in general and data-access refactoring, in particular, are applied relative to releases. This section contains the following questions.

1. *From your experience and opinion, please state your agreement with the following statements.*

   We provided a multiple choice grid with rows containing the statements : *Refactorings are often made shortly after releases, Most refactorings are done just before releases, Refactoring activities are not affected by release deadlines, Data-access refactorings are often made shortly after releases, Most data-access refactorings are done just before releases and Data-access refactorings are not affected by release deadlines.* We provided a multiple choice of agreement rating similar to section one question one.

2. *What is your justification for your choice in the previous question?* (open-ended question)

3. *Do you have more comments on the relationship between refactoring activities and release time?* (open-ended question)

Section Four contains questions regarding factors that should be considered when assigning developers to perform data-access refactoring. This section contains the following questions.

1. *What factors do you think should be considered when assigning a developer to perform refactoring on data-access classes? Rate the following factors from 1 (lowest impact) to 5 (highest impact) based on your opinion and experience.* This question has a multiple choice grid with rows containing factors including: *The coding experience of the developer, The ownership, or familiarity of the developer to the target class, Developer's*

availability, Refactoring contribution, experience of developers, and Random assignment of developers. Each row is associated with a Likert scale ranging from one to five and an option 'I don't know'.

2. *What is your justification for your choice in the previous question?* (open-ended question)

3. *What additional factors should be considered to assign developers for refactoring data-access classes?*

In section five, we asked survey participants about SQL query optimization, SQL and NoSQL data-access smells and associated data-access refactoring practices. Section five contains the following questions.

1. *I consider optimizing SQL queries during data access refactoring* We provided a multiple-choice agreement level with options similar to section one question one.

2. *What is your justification for your choice in the previous question?* (open-ended question)

3. *During SELECT query, fetching all columns using (*) is considered as SQL code bad smell as it creates unnecessary coupling between database columns and data access logic and as it could impact performance during high workloads. How do you rate the criticality of this smell from 1 (Barely critical) to 5 (Very critical)* For this question, we provided a five-point Likert scale in addition to 'I don't know'.

4. *What is your justification for your rating in the previous question?* (open-ended question)

5. *I consider refactoring queries to fix the above-mentioned SQL code smell.* We provided a multiple-choice agreement level with options similar to section one question one.

6. *What is your justification for your rating in the previous question?* (open-ended question)

7. *The NULL marker is used in a relational database to indicate that data does not exist in the database. Improper usage of the marker in query (e.g., column=NULL or column <> NULL) is considered a bad SQL code smell as it could return unexpected results. It is recommended to use "IS NULL" to check for null in select queries. How do you rate the criticality of this smell from 1 (Barely critical) to 5 (Very critical)* For this question, we provided a five-point Likert scale in addition to 'I don't know'.

8. What is your justification for your rating in the previous question? (open-ended question)

9. *I consider refactoring queries to fix the above-mentioned SQL code smell.* We provided a multiple-choice agreement level with options similar to section one question one.

10. *What is your justification for your choice in the previous question?*

11. *Did you encounter data-access smells (bad practices) in data-access classes that interact with NoSQL databases?* For this question, we provided options as "yes" and "no" and we proceeded to a section for respondents that answered yes to this question.

12. *Please mention the anti-patterns you have observed in data-access code of NoSQL databases* (open-ended question) Only respondents that answered yes to the previous question see the following questions about NoSQL anti-patterns.

13. *I consider refactoring to fix the anti-patterns I observed in NoSQL database access code.*

    We provided a multiple-choice of agreement level with options similar to section one question one.

    What is your justification for your choice in the previous question? (open-ended question)

In the last section, we asked the survey participants demographic questions about their software development and refactoring experience. This section contains the following questions.

1. *What best describes your role in your current organization/projects?* We provided a multiple-choice containing *Software Developer, Product Owner, and Manager.* We also allowed participants to enter roles that are not listed.

2. *How many years of experience do you have in software development?*

   For this question, we provided a multiple choice containing the following experience ranges that are *Less than 1, Between 1 and 5, Between 5 and 10 and More than 10*

3. *How often do you perform refactoring in current or past projects?*

   For this question, we provided a multiple choice with options that include *Never, Rarely, Sometimes, Often, and Always.*

4. *How often do you perform refactoring on data-access classes in current or past projects?*

   We provided a multiple choice with options that include *Never, Rarely, Sometimes, Often, and Always.*

4.6 Chapter summary

In this chapter, we presented the details of how we identified subject systems, collected and extracted data to prepare the datasets utilized to answer all research questions in this study. All the research questions are discussed in the following Chapters.

CHAPTER 5 SPECIFICATION OF DATA-ACCESS TECHNICAL DEBTS

5.1 Chapter overview

In this chapter, we present the analysis approach and findings regarding specification of data-access SATDs and specification of data-access performance anti-patterns. We answer the following research questions to complete research objective one.

RQ 1.1: **What is the composition of data-access SATD?**

> **Main finding:** Besides SATDs categorized in the taxonomy of Bavota and Russo [1], we found several SATDs pertaining to data access operations such as query construction, data synchronization, index management and transactions. We also found that Low internal quality code debt has the highest prevalence among data-access SATDs in both SQL and NoSQL subject systems.

RQ 1.2: **What are the data-access performance anti-patterns prevalent in data-intensive systems?**

> **Main finding:** We identified 14 new data-access performance anti-patterns categorized in to high-level anti-patterns regarding database connection, interacting with database driver API, caching, indexing, and query. We also find several instances of performance anti-patterns identified in the previous studies.

5.2 RQ 1.1: Composition of data-access SATD

Bavota and Russo [1] proposed a taxonomy of SATDs by conducting large scale analysis on traditional software systems. We extended their taxonomy by identifying data-access SATDs and improving the generalization of the taxonomy to data-intensive systems. In this subsection, we outline our analysis approach and our findings regarding composition of data-access SATD (RQ1.1).

5.2.1 Analysis approach

To answer this research question, we first identified unique data-access SATD comments in our dataset. We built an LDA topic model on the dataset to generate the strata needed for

stratified sampling. Finally, We conducted a manual analysis on the sample SATD comments. We provide a detailed description in the following paragraphs.

Build LDA model: We then applied common NLP preprocessing techniques. In particular, we removed punctuation, common English stop words, and the words "todo" and "fixme," as they are very common in most comments. Then, we applied lemmatization and stemming using the Python NLTK library. The final output of this pre-processing is a tokenized comment.

The tokenized comments were transformed using TF-IDF transformation. The input of the LDA was the TF-IDF representation of the comments in our dataset. After the preprocessing, we run the LDA topic model, using the Gensim Python library to cluster the SATD comments based on similarity. We experimented with different hyper-parameters, namely the number of topics, alpha, and beta using coherence score as model performance evaluation. First, we experimented with topics from 5 to 75, increasing by 5 every iteration. The coherence score gradually increased as we increased the number of topics and reached a maximum value of 20 topics (0.39%) for SQL systems. For NoSQL systems, we started with less than five topics since the corpus of NoSQL comments was smaller, then we continued until 150 because we saw some fluctuations in coherence score as the number of topics increased. We obtained the highest coherence score (0.45%) when the number of topics was set to 4. Next, we experimented with alpha and beta using a range from 0.01 to 0.1 with 0.3 intervals. We did not see a significant change in the coherence score. Hence, we used the default values on Gensim (alpha and beta: 'symmetric' meaning alpha and beta are set as the inverse of the number of topics). Both LDA models achieved a lower coherence score, below 0.5. However, we did not consider the interpretation of the topics. Instead, we used the LDA to cluster similar comments before sampling. After the LDA model training, we assigned each document to a specific topic. The overall output of the LDA model was documents clustered under each topic group. We used stratified random sampling from the clusters to generate our dataset for manual analysis.

Stratified sampling: We prepared a dataset for manual analysis using stratified sampling from each LDA topic group. The dataset contains 361 data-access SATD comments, composed of 183 data-access SATD comments from SQL systems and 178 data-access SATD comments from NoSQL systems. that represent our entire dataset with 95% confidence. We used stratified random sampling to pick representative samples from each LDA topic group or cluster.

Manual analysis: The manual analysis was conducted using deductive coding to assign themes to the comments. We started with the themes identified by Bavota and Russo [1]

and extended them with themes specific to data access. To have a common interpretation of the labels among authors, we conducted iterative sessions to label sample SATDs and resolve conflicts. After that, the first author labeled all the 361 SATDs, which were then divided into three sets and reviewed by three more authors to ensure that at least one additional person checks each label. Finally, all conflicts were resolved through face-to-face discussions.

During the labeling process, we found some comments that were identified as SATD comments by the detector tool but were not related to technical debt. Recall that Liu et al. reported an average F_1 score of 73.7% for the tool [83]. A common reason was that they contained one of the keywords (*e.g.,* "fix"), but the developer meant it for a different purpose (*e.g.,* "*// import release fix into the release branch*"[1]). We found 105 instances (29%) of these comments and marked them as *false positives*.

We found 12 comments in which the information from the comments and source code did not give enough context to assign the comments to the appropriate category. We marked such instances as *unclear*.

We also found 4 comments that belonged to more than one category, as they typically ordered tasks in a list under a "todo" comment. These tasks often belonged to various SATD categories; hence, we decided to mark them as *multi-label comments*.

For multi-label comments, we cannot identify one specific category. Hence, we exclude them for this analysis. After we excluded false positives, unclear comments and multi-label comments, the final dataset contained a total number of 240 data-access SATD comments.

5.2.2 Taxonomy of data-access SATDs

Figure 5.1 shows the taxonomy we extended from the work of Bavota and Russo [1]. In particular, we added a new high-level category called *data-access debt* and provided more specialized categories for code debt, test debt and documentation debt. While our primary focus is on the newly added categories, especially on the data-access debt categories, we also provide a brief description of the original categories [1] for completeness. We describe the composition of SATDs categorized in Figure 5.1 in the following paragraphs. We start with the SATDs identified by Bavota and Russo [1], then we move to the newly added categories.

Code debt: *Code debt* includes "*problems found in the source code which can affect negatively the legibility of the code, making it more difficult to be maintained*" [129]. It is divided into *low internal quality* and *workaround* categories. SATD comments that mention code quality issues related to program comprehension are categorized as *low internal quality*.

[1] https://bit.ly/3siSWzX

For example, a comment from the *low internal quality* category in *Blaze-Persistence*[2] says:

```
// TODO this is ugly think of a better way to do this
```

Comments justified by the developers as a workaround to address specific requirements are categorized under *workaround*. For example, quick fixes that mention a hack or workaround belong to this category. We extended *workaround* SATDs with a *workaround on hold* category. An "on-hold" SATD comment describes a problem that can be fixed once an issue referenced in the comment is addressed [130].

We found a specific case of an "on-hold" SATD when the issue holding back the developers was due to synchronization problems with the database schema. We dedicated the *workaround on hold due to database schema* category for similar SATDs. As an example, the comment in *OpenL Tablets*[3] says:

```
// TODO It should be removed when the table can be resolved by the ID
```

Defect debt: Comments that mention bugs or defects that should be fixed but are postponed to another time are categorized under *defect debt*. The main causes of this debt can be *defects* or *low external quality* issues.

Defects are further divided into *known defects to fix* and *partially fixed defects*. An example of a *partially fixed defect* can be seen in *Snowstorm*:[4]

```
// TODO Remove this partial ESCG support
```

We found two specific cases when the issue was due to a *known defect of external library*; thus, we introduced a sub-category for these cases. *Low external quality* SATD comments describe problems with a high probability of becoming a bug or defect [1], as they may affect user experience.

Design debt: SATDs related to *code smells* or *design patterns* are grouped in this category.

Comments that discuss the violation of object-oriented design or mention refactoring as a solution are categorized under *code smells*. Comments suggesting the usage of a design pattern are classified under *design patterns*.

Documentation debt: This type of SATD can be identified in comments by looking for "*missing, inadequate, or incomplete documentation of any type*" [129]. Comments referring to issues already addressed are also categorized under documentation debt. This might happen when developers forget to update the documentation or comments after some source

[2]Blaze-Persistence, `https://bit.ly/3qGaXbb`
[3]OpenL Tablets, `https://bit.ly/3sioFkX`
[4]Snowstorm, `https://bit.ly/3dEUMXN`

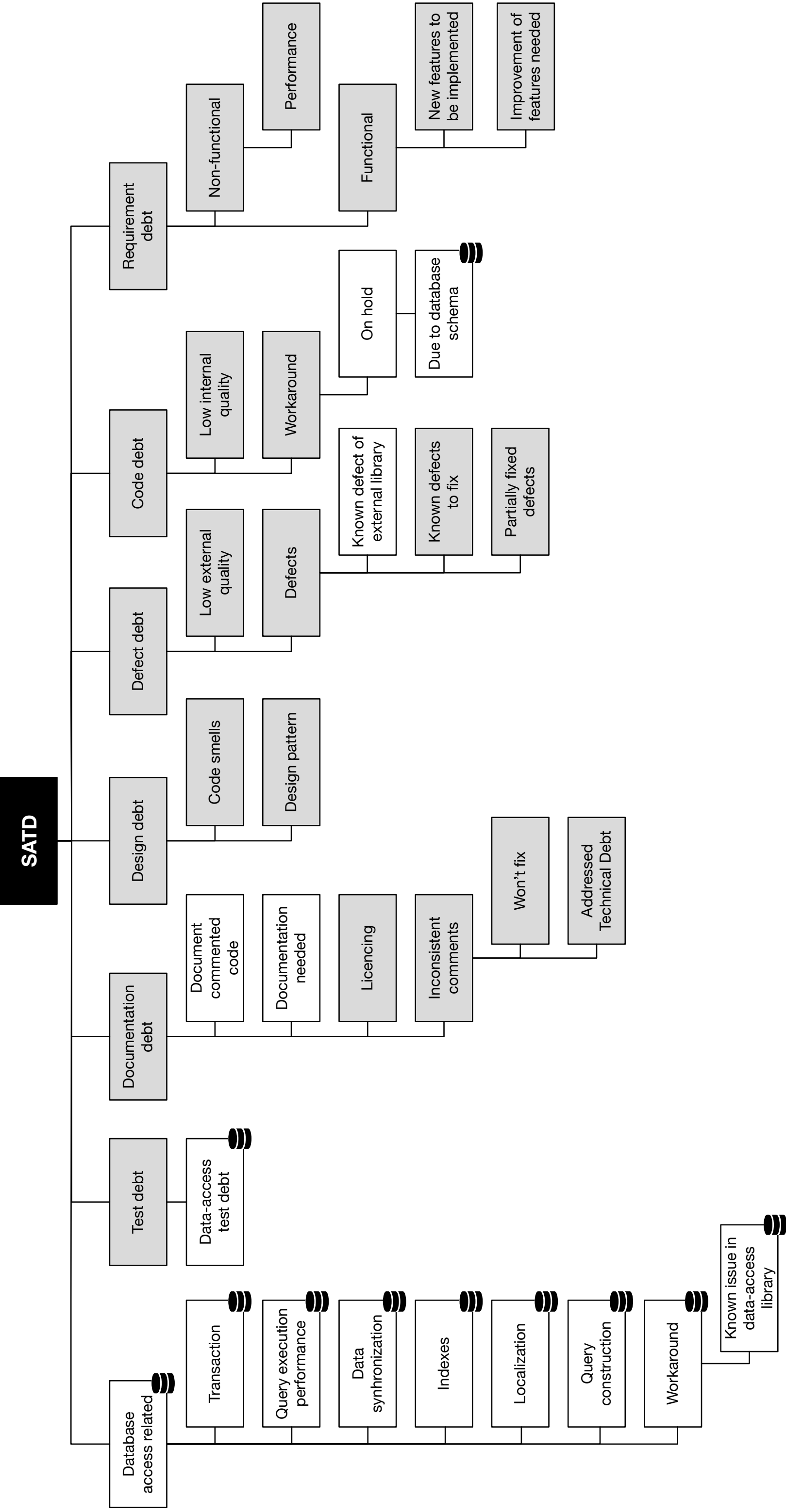


Figure 5.1 SATD classification hierarchy extended from Bavota and Russo [1]. White boxes are newly added categories to existing categories (gray boxes). Boxes marked with a database icon (🛢) are categories closely related to database accesses.

code changes. *Documentation debt* is divided into *inconsistent comments* and *licensing* categories. *Inconsistent comments* are further divided into *addressed technical debt* and *won't fix* categories [1].

We added two new sub-categories, *document commented code* and *documentation needed*, as we found multiple instances of such cases. *Document commented code* comments explain the rationale of code that was commented out but still needed due to a pending "todo" or "fixme." Comments labeled as *documentation needed* mention the necessity of providing documentation to a piece of code.

Requirement debt: Comments that describe the need for new features to be implemented are categorized under *requirement debt*. Bavota and Russo [1] further classified these to *functional* and *non-functional* requirement debt. *Functional* requirement debt includes the *new feature to be implemented* and *improvement to features needed* categories.

Additionally, under *non-functional* requirement SATDs, we also observed a few issues related to *performance* requirements.

Test debt: *Test debt* affects the quality of testing activities [129]. These comments are typically found in testing classes and indicate low quality of testing code, *e.g.,* in terms of readability or the appropriateness of test cases and testing conditions.

We identified several *test debt* comments in the test code related to data accesses. We grouped these under the *data access test debt* category. Examples of these are related to the testing of database access operations, such as transactions and query syntax. For example, a comment in *Sqlg*[5] says:

```
// TODO this really should execute limit on the db and finally in the step. That way less results are
    returned from the db
```

The comment follows a query in a test method of the *TestRangeLimit* class. Sqlg provides graph computing capabilities on SQL databases, and the method tests the range specification of a query. As the comment suggests, the query in the test could be optimized to return fewer results.

Database access related SATDs

We added *database access related* as a new category that groups together SATDs dealing with the implementation of data-access logic. This category is further divided into sub-categories. We describe each sub-category and provide examples from the subject systems.

[5] Sqlg, `https://bit.ly/3wxbAqW`

Query execution performance: We found SATD comments dealing with issues about the execution performance of database queries. For example, a comment in *GnuCash Android*[6] says:

```
// Relies ON DELETE CASCADE takes too much time
```

The comment belongs to a method that deletes all accounts and transactions from the database. As the developers note, the cascade operation takes too much time and affects the method's performance.

Transactions: We identified comments about code that deal with transactions or rollback operations. An example of this type of debt was found in *Sqlg*:[7]

```
// TODO undo this in case of rollback?
```

The comment appears in a method that removes a schema from a database. The operation is performed in a transaction; however, the implementation does not undo the operation in case of a rollback.

Workaround on known issue in data-access library: We found comments that described workarounds of problems existing in the data-access libraries. In such comments, the developers explicitly reference the issue, pointing to the library's issue tracking system.

The following comment in *Foxtrot*[8] explains a workaround by directing the developer to an issue of Hazelcast, a key-value store implementation.

```
// HACK::Check https://github.com/hazelcast/hazelcast/issues/1404
```

Data synchronization: These SATD comments describe a synchronization issue between the application and the database. An example comment can be found in *UPortal*:[9]

```
// todo Figure out if we should instead return the id of the system user in the DB
```

The comment appears in a method called *getUserIdForUsername(...)* that is supposed to return a user's ID. However, as an additional comment says, the method "*returns 0 consistent with prior import behavior, not the id in the database.*"

Indexes: Comments about issues related to indexes in the database are grouped under this category. For example, the following comment in *Sqlg*[10] describes the need for support for indexes.

[6] GnuCash Android, `https://bit.ly/37H1PeV`
[7] Sqlg, `https://bit.ly/3pRCOnK`
[8] Foxtrot, `https://bit.ly/3urWcey`
[9] UPortal, `https://bit.ly/3qY5OX2`
[10] Sqlg, `https://bit.ly/3aIqEcc`

```
// TODO Sqlg needs to get more sophisticated support for indexes i.e. function indexes on a property etc.
```

Localization: We found comments about localization issues in the database, *i.e.,* problems with character sets or collation. The following comment in *Robolectric*[11] highlights the need for creating a collator as part of registering a localized collator.

```
// TODO: find a way to create a collator
// http://www.sqlite.org/c3ref/create_collation.html
// xerial jdbc driver does not have a Java method for sqlite3_create_collation
```

Query construction: We found comments that mentioned issues about the construction of database queries. The following comment in *Carbon-apimgt*[12] notes a pending task to filter results by the status of the APIs.

```
// TODO FILTER RESULTS ONLY FOR ACTIVE APIs
```

The query marked with the todo comment returns unnecessary records when only a specific API context is needed.

Distribution of manually categorized data-access SATDs

Table 5.1 shows the distribution of the final labels in the sample dataset. The comments under each category were presented separately for SQL and NoSQL subject systems. We mark SATDs related to database accesses with a database icon (🛢) and regular SATDs with a file icon (🗋). The categories are sorted according to the *total number of comments*.

Table 5.1 shows that a large portion of the comments belongs to sub-categories of *code debt*, *requirement debt* and *defect debt*. This is a similar observation with Bavota and Russo [1]. We can also see that *data-access debts* are also found in smaller quantities compared to the traditional SATDs. The most considerable *data-access debt* is *data-access test debt*, followed by *query construction*.

When we contrast SATDs between SQL and NoSQL systems, we can see that most categories have a higher occurrence in SQL systems than in NoSQL systems.

[11] Robolectric, https://bit.ly/3umvXpD
[12] Carbon-apimgt, https://bit.ly/2NvDZvQ

Table 5.1 Distribution of categories in the manually classified dataset

Category	SQL	NoSQL	Total	Percent
Low internal quality	21	19	40	16.39
Improvement to features needed	16	14	30	12.30
Known defects to fix	9	16	25	10.25
Workaround	12	11	23	9.43
New features to be implemented	13	8	21	8.61
Low external quality	15	3	18	7.38
Code smells	10	6	16	6.56
Test debt	3	12	15	6.15
Data-access test debt	5	3	8	3.28
Query construction	6	1	7	2.87
Document commented code	1	4	5	2.05
On hold	1	4	5	2.05
Query execution performance	3	2	5	2.05
Performance	1	2	3	1.23
Addressed technical debt	2	1	3	1.23
Documentation needed	3	0	3	1.23
Known issue in data access library	1	1	2	0.82
Data synchronization	2	0	2	0.82
Transactions	1	1	2	0.82
Known defect of external library	1	1	2	0.82
Partially fixed defects	0	1	1	0.41
Due to database schema	1	0	1	0.41
Localization	1	0	1	0.41
Indexes	0	1	1	0.41
Design patterns	1	0	1	0.41

5.3 RQ 1.2: Specification of data-access performance anti-patterns

The first step towards characterizing data-access performance anti-patterns is their specification. While there exist several efforts to specify data-access performance anti-patterns (e.g., [12, 61, 131]), the studies did not consider NoSQL-based and polyglot persistence systems. We complement existing specifications of data-access performance anti-patterns by analyzing performance issues collected from several NoSQL-based and polyglot data-intensive systems. We outline the manual analysis approach we followed to identify data-access performance anti-patterns and the description of the prevalent performance anti-patters in the following Sub-sections.

5.3.1 Analysis approach

We used an open coding approach to come up with the taxonomy of data access performance anti-patterns. I and another co-author participated in the labeling. We prepared a web application to help us manage the issue labeling process. We utilized the **data-access performance issues dataset** for this analysis. We leveraged the relevance score obtained from the Elasticsearch to sort the performance issue dataset from most relevant to least relevant. The relevance score measures the relevance of the issues to the search query. We next performed the labeling of the sorted dataset in multiple rounds with 100 labels per round until label saturation is achieved. We examined all conversations associated with the issue to identify and label the root causes of the issue.

To minimize the impact of researcher bias, we assigned the issues to each labeler under the constraint that each issue will be reviewed by both labelers. Furthermore, we labelled the issues independently and resolved labeling-conflicts by discussing among us. We utilized the list of performance anti-patterns from the work of [12] as a seed and continue adding new labels as we continue the labeling.

Once labeling saturation happened, we built the taxonomy from the ground up using the card sorting approach to come up with the taxonomy of data access performance anti-patterns shown in Figure 5.2. We achieved labeling saturation after labelling 250 instances, but we labeled 150 more issues to make sure we don't miss a new performance anti-pattern. Once we achieved labeling saturation, we computed the inter-rater reliability agreement using Cohen's Kappa for the 150 newly labeled performance issues after saturation, excluding 99 false positive issues. We obtained a Cohen's kappa value of 0.764 indicating a substantial labeling agreement.

5.3.2 Data-access performance anti-patterns

Figure 5.2 shows the catalog of data-access performance anti-patterns prevalent in the analyzed issues. The performance anti-patterns are grouped into seven high-level categories namely *Data fetching and update anti-patterns, Database driver or API access anti-patterns, Database connection anti-patterns, Query anti-patterns, Indexing anti-patterns, Data node configuration and management anti-patterns, and Caching anti-patterns.* In addition to 12 data-access performance anti-patterns from previous studies, we identified 14 new data-access performance anti-patterns from our analysis. The newly identified anti-patters are written in **Blue** to distinguish them from the ones that were identified in the previous studies (written in **Brown**). We will provide the description and example for each of the newly identified anti-patterns. We will also briefly describe the performance anti-patterns from other studies and provide a reference of the papers that introduced and described them in detail for readers.

Data fetching and update anti-patterns

We categorize anti-patterns that occur during data-fetching, rendering on some user interface (UI) component as well as during updating some data residing in database under *Data fetching and update anti-patterns.* We identified one new anti-pattern, *Sequential lookup of multiple keys*, in addition to performance anti-patterns identified in previous studies that are *Inefficient rendering* , *Inefficient updating* ,*Unnecessary column retrieval* , and *Unnecessary row retrieval* under this category.

Inefficient rendering anti-pattern [12, 61] occurs when an inefficient API is used to load the data that populates a user interface component from databases. This anti-pattern increases the page load time which negatively affects the user experience. This anti-pattern is fixed by improving the implementation of the data-access APIs and the rendering logic.

Inefficient updating anti-pattern [12, 61, 131] occurs when issuing multiple queries to update multiple database records at the same time. This creates unnecessary multiple requests to the database, increasing the overall response time. This anti-pattern is fixed by batching multiple update operations into a single query.

Unnecessary column retrieval anti-pattern [12,131,132] concerns with retrieving more columns or fields from the database than needed, which wastes band-width and increases response time. Tables that contain many columns are more prone to this anti-pattern. Fetching only the necessary columns from the database fixes this data-access performance anti-pattern. Similarly, *Unnecessary row retrieval* anti-pattern [12, 133] concerns with fetching more rows

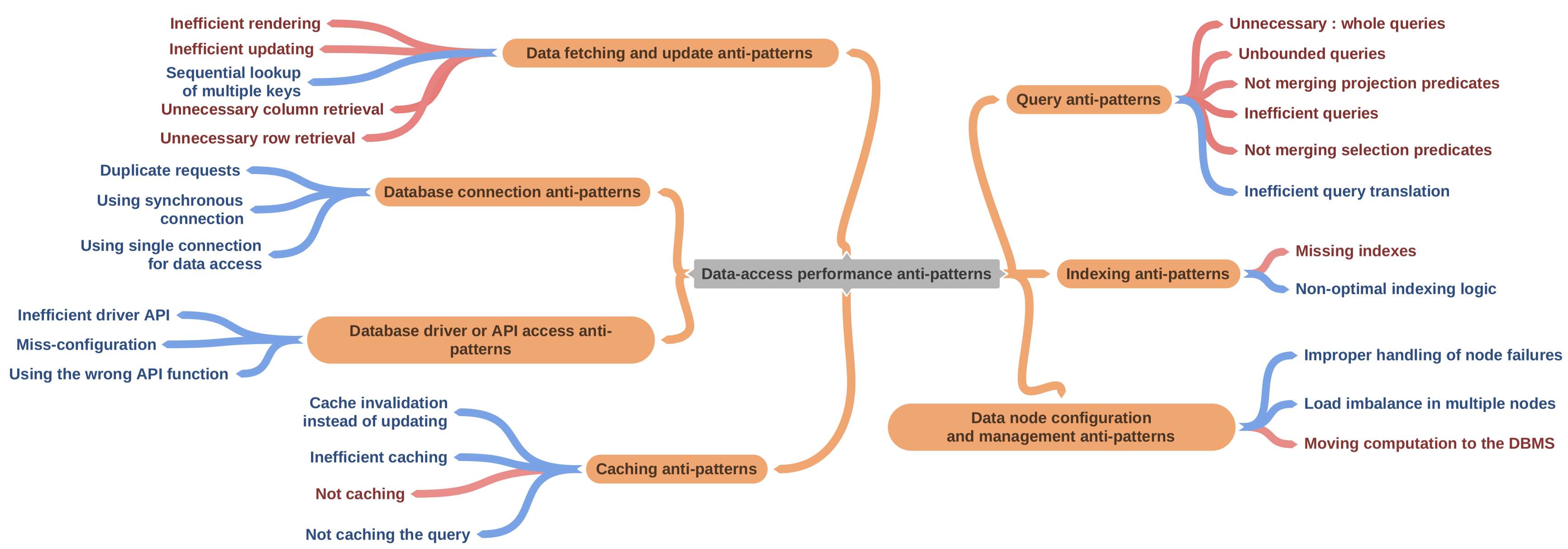


Figure 5.2 Catalog of data-access performance anti-patterns prevalent in the analyzed data-access performance issues.

or records from the database than needed wasting network bandwidth and increasing response time. This anti-pattern is fixed by modifying the queries to include filtering clauses that discard unwanted rows.

We identified *Sequential lookup of multiple keys* a new anti-pattern that occurs when users search for a match from a list of keywords, the data-access API performs multiple scans of the table, one scan for each keyword. For tables that contain many records, running multiple scans degrade performance. This anti-pattern is fixed by modifying the data-access logic to match multiple keywords in a single scan. For example, an issue reported in *Cht-core* application [13] where the API creates separate query for each keyword slowing down data loading. This issue was fixed by a workaround of introducing an index to mitigate the loss of performance caused by this anti-pattern.

Database connection anti-patterns

Data-access performance anti-patterns that concern inefficiencies in setting up the connection or communicating with database or persistence systems are categorized under *database connection anti-patterns*. We categorized the newly identified *Duplicate requests*, *Using synchronous connection*, and *Using single connection for data access* data-access performance anti-patterns under this category.

Duplicate requests anti-pattern concerns with sending multiple requests to a database using a similar query. This issue usually happens when requests are sent on some user interface events, such as page load. Improper event handling often leads to multiple similar requests to be sent to the database. This anti-pattern impacts the response and throughput of the persistence system by sending unnecessary and redundant queries to the database. When there are numerous clients at the same time, this anti-pattern will exhaust connection pools, creating a timeout in the worst case. This anti-pattern can be fixed in client code by adding a logic that tracks if the same request is sent before. One instance of this issue was reported on Kibana[14] where they send duplicate requests on a page load event and then show a particular component. In this example, they fixed the issue by creating a logic that tracks already sent requests to avoid redundant connection requests to the back end.

Using synchronous connection anti-pattern occurs when a request is sent from the main application or back-end thread to the database, which makes the back-end request handler block until the database processes the request. This anti-pattern can be observed during reading and writing operations. It is fixed by modifying the data-access logic to use *asyn-*

[13] https://github.com/medic/cht-core/issues/4981
[14] https://github.com/elastic/kibana/issues/120219

chronous connections for complex requests that take time to be processed. The issue reported in Hazelcast[15] shows one manifestation of this anti-pattern where the partition thread(back-end thread) hangs during a read request from slow external sources. This issue was fixed by making an asynchronous request from the partition thread, freeing the partition thread to handle other requests until the data is loaded from the external source. Once the data is fully loaded in the background, the task will be reassigned to the partition thread.

Using single connection for data access anti-pattern occurs when sequential requests are sent instead of parallel requests to read data from different sources or to modify data. This anti-pattern is relevant when the data nodes are replicated to improve the scalability and robustness of the data-intensive systems. This anti-pattern is fixed by using parallel connection requests whenever possible. One instance of this anti-pattern is observed in the issue reported in *Presto* application[16] in which the current implementation uses a single connection to read data from multiple tables, which could slow down the application. The suggested fix was to implement parallel data read operations.

Database driver or API access anti-patterns

Performance anti-patterns concerning inappropriate choice of the databases, lack of support of features, misuse and miss-configuration of data-access drivers or APIs are categorized under *Database driver or API access anti-patterns.* We identified new types of anti-patterns that are *Inefficient driver API*, *Miss-configuration*, and *Using the wrong API function* under this category.

Inefficient driver API performance anti-pattern that occur due to inefficient database access drivers or APIs. The impact of the inefficient implementations of the driver or API will affect the overall performance of applications that rely on them. Furthermore, all applications that use the drivers and APIs could be impacted by the introduced inefficiencies. *Inefficient driver API* anti-patterns could occur in all types of data-access operations including read, write, delete, update or insert. This performance anti-patterns can be fixed by requesting a feature update from the API, updating the driver or considering alternative options. One issue reported in *Mongoengine*[17] mentions inefficient implementation of referential integrity during bulk insert operation which was fixed by updating the MongoEngine.

Miss-configuration anti-pattern concerns the incorrect configuration of data-access drivers and APIs resulting in slow performance or timeout in the worst case. Such anti-patterns are

[15]https://github.com/hazelcast/hazelcast/issues/9507
[16]https://github.com/prestodb/presto/issues/10832
[17]https://github.com/MongoEngine/mongoengine/issues/361

fixed by choosing appropriate configuration of data-access drivers, considering the type and amount of workload of the applications. One user reported an issue on *Redisson*[18] about large number of connections opened to Redis database when the application loads for the first time. The suggested fix was to modify the configuration of the driver to reduce the connection pool size in the configuration file.

Using the wrong API function anti-pattern is concerned with calling an inefficient data-access API function when there exists a more efficient alternative API function. The performance loss will be caused by using the less efficient API functions. One user of *Prisma* reported a slow performance during bulk write operation [19]. The slow performance was due to the usage of the *create* API that performs a single write instead of the *createMany* function that batches multiple writes. The recommended fix was to use the *createMany* API. In addition, appropriate documentation of the APIs will help users to identify the correct API to use.

Caching anti-patterns

We categorized data-access performance anti-patterns associated with catching data. We identified three new performance anti-patterns under this category that are *Cache invalidation instead of updating*, *Inefficient caching*, and *Not caching the query*. We also observed instances of *Not caching*, performance anti-pattern identified in previous works, in our issue dataset.

Not caching anti-pattern [12, 70, 134, 135] occurs when multiple syntactically equivalent data-access queries are sent to the database without caching the results. This anti-pattern is fixed by introducing a caching layer which avoids hitting the database when the query result is already known.

Not caching the query anti-pattern is observed when query builders are used in the data-access code. If the query is not cached, it is rebuilt and validated every time a read or write request is sent to the database, creating unnecessary load on the database system. The recommended solution would be to build and cache the query once and bind that query to all the records to be inserted or updated in a batch. One pull request reported in *Stargate* [20] application highlights the performance improvement obtained by caching the query during bulk insert/update operations. This anti-pattern should not be confused with, *Not caching* which is about not catching the result data not the query.

Cache invalidation instead of updating anti-pattern occurs when developers perform cache

[18] https://github.com/redisson/redisson/issues/2265
[19] https://github.com/prisma/prisma/issues/9612
[20] https://github.com/stargate/stargate/issues/1333

invalidation instead of updating the content of the cache, causing an unnecessary database hit and reducing the data-access performance. In situations, where the cache contains large number of records, invalidating the cache when a single or few records are changed degrades the read performance. This anti-pattern is fixed by using cache update functions that can target a subset of the records that are modified. An example issue that highlights this performance anti-pattern is reported in *Kibana*[21]. In this issue, the implementation invalidates cache when user roles are changed, causing a reload of the rules on client machines. The performed fix was to modify the caching logic to only update the modified user roles.

Inefficient caching anti-pattern is concerned with an inefficient implementation of the caching logic, which negatively affects the data-access performance. This anti-pattern is fixed by improving the techniques and implementation logic of data caches. For instance, one pull request in *Hazelcast*[22] optimizes the serialization used in the second level cache for Hibernate to improve the performance of the cache.

Indexing anti-patterns

We categorized data-access performance anti-patterns concerning index usage and management under this category. We identified *Non-optimal indexing logic* anti-pattern under this category. We also observed instances of *Missing indexes* anti-pattern which is identified in previous studies.

Non-optimal indexing logic performance anti-pattern occurs when the indexing logic is implemented or utilized in a sub-optimal way that degrades data-access performance. One issue reported in *Typesense*[23] demonstrates the impact of sub-optimal indexing logic when deleting a table with large number of records. They obtained dramatic improvement from, tens of minutes to seconds, in deletion speed by optimizing the indexing logic. This performance anti-pattern is fixed by optimizing the implementation and utilization of indexes.

Missing indexes anti-pattern [12, 27, 29, 61] occurs when necessary indexes are not included in the database schema implementation. The lack of the indexes will result in sub-optimal read and write performance. This anti-pattern is fixed by adding and using the necessary indexes in the schema.

[21] https://github.com/elastic/kibana/issues/125574
[22] https://github.com/hazelcast/hazelcast/issues/2248
[23] https://github.com/typesense/typesense/issues/515

Data node configuration and management anti-patterns

Data-access performance anti-patterns associated with inefficient configuration, load balancing and failure handling of data nodes. These anti-patterns are relevant when the persistent system is implemented in multi-node database clusters to improve the scalability and robustness of the applications. We identified two new anti-patterns namely *Improper handling of node failures*, and *Load imbalance in multiple nodes*. We also found instances of *Moving computation to the DBMS* anti-pattern identified in previous studies.

Moving computation to the DBMS anti-pattern [12, 132] occurs when developers process the results of multiple queries in the back-end server instead of using database management system when the database management system can process the queries more efficiently. This anti-pattern is fixed by moving the computation to the database management system.

Improper handling of node failures anti-pattern concerns with the improper handling of node failures in multi-node persistence systems. In a multi-node cluster configuration, when the data disk of a node fails while the node is still communicating, the node will be considered active and will not be flagged as failed. Consequently, other requests will be sent to the node, creating unnecessary delay and timeout. This anti-pattern is fixed by adding a logic to track disk failures in data-nodes and notify disk failure to the coordinator. One issue reported in *Elasticsearch* application[24] highlights this performance anti-pattern. This issue reports improper handling of disk failures in data node, causing accumulation of the requests in the coordinator node. This issue was fixed by adding a logic to flag data-nodes with disk failure as failed nodes.

Load imbalance in multiple nodes performance anti-pattern occurs when requests are not fairly distributed to the data nodes, causing overload in some data nodes while other data nodes are underutilized. This anti-pattern is fixed by optimizing the implementation of load-balancing logic to fairly distribute the workload to all data nodes. One pull request reported in *Presto*[25] implements new load-balancing logic, which changes the previous approach of picking the first available data node from a list by randomly shuffling the data nodes before picking.

Query anti-patterns

We categorized performance anti-patterns due to inefficient formulation and translation of data-access query or statements in this category. We identified a new anti-pattern *Inefficient*

[24]https://github.com/elastic/elasticsearch/issues/60037
[25]https://github.com/prestodb/presto/pull/3087

query translation in this category. We also found instances of *Unbounded queries*, *Unnecessary whole queries*, *Not merging projection predicates*, *Inefficient queries*, and *Not merging selection predicates* anti-patterns in our analysis of the issues.

Unbounded queries performance anti-pattern [12, 66, 131, 136] occurs when queries do not include pagination, resulting in unbounded results to be returned at once. This anti-pattern is fixed by splitting the query result into pages using pagination.

Unnecessary whole queries performance anti-pattern [12, 64, 65, 137] occurs when the results of some queries are not used in the application. Such queries create unnecessary load on the database and degrades the application performance. Removing unnecessary queries fixes this anti-pattern.

Not merging projection predicates anti-pattern [12,131,132] is concerned with issuing multiple requests where each request loads a subset of the needed columns, creating unnecessary load on the database. This anti-pattern is fixed by listing all the needed columns in one query.

Inefficient queries anti-pattern [12, 131, 137] is concerned with using queries that incur high performance costs when more efficient and semantically equivalent queries are available. This anti-pattern is fixed by using more optimal queries.

Not merging selection predicates anti-pattern [12,132,138] is concerned with issuing multiple select queries with each request loading a subset of the result rows causing unnecessary load on the database. This anti-pattern is fixed by creating a single query and merging the selection predicates.

Inefficient query translation anti-pattern occurs when user queries are translated into inefficient but semantically equivalent form by the database system. The performance degradation occurs due to the inefficient translation of user queries. One issue in *Prisma* application[26] reports that the user queries are slowed down due to inefficient translation by the database management system. This issue is fixed by optimizing the translation logic implemented by the database system.

We also analyzed the prevalence of the data-access performance anti-patterns using our manually analyzed dataset, excluding the false-positives. The results show that:

⋄ ***Inefficient driver API*** **performance anti-pattern is the most prevalent data-access performance anti-pattern.**

Out of the 400 issues that we labeled, we found 141 true positive data-access performance issues. The remaining issues have some match with the search query, but are not data-access

[26] https://github.com/prisma/prisma1/issues/4948

performance issues. The most prevalent data-access performance anti-pattern was *Inefficient driver API*(15.6%) followed by *Inefficient caching*, *Not caching*, and *Inefficient queries* with a prevalence of 10.64% each. The following anti-patterns had only one instance each: *Missing indexes*, *Cache invalidation instead of updating and Sequential lookup of multiple keys anti-patterns* are some performance anti-patterns with only one instance each.

⋄ **73.4% of the data-access performance issues are from NoSQL based subject systems.**

Out of the 141 data-access performance issues, 103 are from NoSQL subject systems (73.4%). The remaining 38 are from polyglot subject systems. *Missing indexes*, *Moving computation to the DBMS*, and *Unnecessary row retrieval* performance anti-patterns were only found in polyglot based systems. Whereas, some anti-patterns such as *Improper handling of node failures*, *Inefficient updating* and *Using the wrong API function*performance anti-patters were only found in NoSQL subject systems.

5.4 Discussion

The state-of-the-art SATD detection systems do not differentiate between different types of SATDs. One reason for this could be the lack of information on the specific types of SATDs. In this direction, Bavota and Russo [1] provided a taxonomy of SATDs, including design debt, code debt, defect debt, requirement debt, and test debt. While they addressed most of the software development workflow, they did not cover data-access debts, since the subject systems were not data-intensive. We extended their taxonomy, incorporating 11 new database access-specific SATDs generalizing their taxonomy to data-intensive systems. This taxonomy can be utilized for proposing SATD detection approaches that provide more information than their mere existence. This, in turn, helps practitioners in their effort to manage technical debts and future researchers to investigate the impacts of specific types of SATDs on software quality. We find that low internal quality code debts were the most prevalent SATDs among our subject systems. Code debts are also found to be dominant SATDs in traditional software systems [1]. Hence, future research efforts should focus more on code debts as they are more prevalent SATDs in software systems. Data-access SATDs are also important in the context of data-intensive systems.

We extended the list of data-access performance anti-patterns from previous researches (Eg. [12]) by analyzing 400 data-access performance issues extracted from NoSQL based and polyglot data-intensive subject systems. We specified 14 new performance anti-patterns categorized under seven high-level categories. We observed instances of identified performance

anti-patterns in addition to the newly identified anti-patterns. Hence, we improved the generalization of the data-access performance anti-patterns by incorporating NoSQL-based and polyglot-based data-intensive systems. While the identified anti-patterns are obtained by analyzing real world reported performance issues, the criticality of the new performance anti-patterns need to be assessed. Hence, we prepared a developer survey to evaluate the criticality of each of the new anti-patterns. The result of the survey will be discussed in Chapter 8.

Our manual analysis finding shows that NoSQL based systems are more prone to data-access performance anti-pattern compared to polyglot persistence systems. However, further investigation of the prevalence is needed, perhaps after automatic data-access performance detection tool or approach is available.

5.5 Threats to validity

In this section, we discuss threats to validity related to the specification of data-access SATD (RQ1.1) and specification of performance anti-patterns (RQ1.2).

Threats to construct validity

Threats to construct validity refer to the extent to which the experiment setting actually reflects the construct under study. the potential researcher bias in the manual analysis of **RQ1.1 and RQ1.2**. To overcome this threat, the manual analysis conducted by the first author was evaluated by another co-author and all the labeling conflicts were resolved by discussion. We made sure that at least two authors label each data-access SATD and performance issue. Another threat to construct validity is that the accuracy of the SATD detection is not 100%. Hence, our dataset could contain false positives. However, we checked each data-access SATD it is not true positive and excluded all the false positive SATDs from our analysis. Similarly, we used a keyword base heuristics to identify data-access performance issues which could introduce several false positive performance issues, and we could also lose some true data-access performance issues. To mitigate this threat, we first checked if the data-access performance issue is true positive before labeling and excluded the false positives from our analysis.

Threats to external validity

Threats to external validity concern the ability to generalize experiment results outside the experiment setting [139]. The extent to which the produced catalog of data-access perfor-

mance anti-pattern exhaustively covers all performance issues is a threat to external validity. To mitigate this threat, we did not fix the number of samples to manually analyze. We rather sorted the performance issues dataset in decreasing order of relevance and started labeling until we achieve a labelling saturation. While we achieved a labelling saturation after analyzing 250 issues, we continued our analysis and labeled 150 more issues, and we did not generate a new performance anti-pattern. The specification of data-access SATD is also prone to a similar threat to external validity, and we manually analyzed statistically significant data-access SATD samples to mitigate this threat.

Threats to reliability validity

Threats to reliability validity concern the possibility for independent researchers to replicate this study. To minimize potential threats to reliability, all our subject systems are open source and available on GitHub. Furthermore, we provided all the necessary materials to replicate our study in our replication packages [140, 141].

5.6 Chapter summary

In this chapter, we discussed the specification of data-access SATDs and performance anti-patterns obtained by analyzing a sample data-access SATD comments and data-access performance issues. We identified 8 new data-access SATDs and 14 new data-access performance anti-patterns. We believe that the newly identified anti-patterns improve the generalization of existing non-data-access SATDs and performance anti-patterns. Our findings could be leveraged in researches regarding automated detection and refactoring of data-access technical debts, and practitioners can leverage the findings to improve the quality of data-intensive software systems.

CHAPTER 6 CHARACTERIZATION OF DATA-ACCESS SATDS

6.1 Chapter overview

In this chapter, we present the analysis approach and findings regarding the characterization of data-access SATDs to achieve **Sub-objective 2.1**. We present the analysis and result of research questions **RQ 2.1, RQ 2.2, and RQ 2.3**.

RQ 2.1: **How prevalent are SATDs in data-intensive systems?**

> **Main finding:** Data-access SATD has lower prevalence than regular SATD in both SQL and NoSQL subject systems. We observed that the number of data-access SATDs tends to increase as systems evolve, regardless of the database type. In most cases, NoSQL systems have higher median data-access SATD compared to SQL systems.

RQ 2.2: **How long do SATDs persist in data-intensive systems?**

> **Main finding:** We found statistically significant differences between the survival curves of data-access and regular SATDs in both SQL and NoSQL systems, which indicates that data-access SATDs are fixed sooner than regular SATDs. However, we also found a significant number of data-access SATDs introduced in the first versions of the systems (5% for SQL and 7% for NoSQL systems). Many persisted until the latest versions (68% for SQL and 39% for NoSQL).

RQ 2.3: **What are the circumstances behind the introduction and removal of data-access SATD?**

> **Main finding:** Most SATD comments in data-access classes are introduced at the later stages of change history. However, SATD comments where database access is explicitly mentioned (*i.e.,* database access related categories in the taxonomy) are introduced earlier than SATD comments unrelated to database accesses. We observed similar distribution between SQL and NoSQL data-access SATDs in introduction time. *Bug fixing* and *refactoring* are the main reasons behind the introduction of data-access SATDs, followed by *feature enhancement* and *supporting new features*. Data-access debt removal commits are often associated with *feature enhancements*, *new features*, and *bug fixing*. None of the observed removal events was associated with *refactoring*. We did not find removed *database access related SATD* comments.

6.2 RQ 2.1 : Prevalence of SATDs in data-intensive systems

In this section, we discuss the analysis approach and findings regarding the prevalence of SATDs in data-intensive systems. Further studies to characterize data-access SATDs are needed only if they are prevalent in data-intensive systems. Hence, investigating the prevalence of data-access SATDs is the first step towards their characterization. We used the **SATD dataset** described in Section 4.3 for this analysis.

6.2.1 Analysis approach

To answer this RQ, we computed the total number of data-access SATD comments and non-data-access SATDs for both SQL and NoSQL subject systems using the SATD dataset described in Section 4.3. We collected the number of SATDs for each snapshot of the subject systems' change history. We used violin plots to show how the prevalence of SATDs change as systems evolve and compared data-access and regular SATDs as well as SQL and NoSQL systems.

6.2.2 Findings

The results show that:

◇ **Data-access SATD has lower prevalence than regular SATD in both SQL and NoSQL subject systems. The number of data-access SATDs tends to increase as**

systems evolve, regardless of the database type. In most cases, NoSQL systems have higher median data-access SATD compared to SQL systems.

Figure 6.1 shows the distribution of the number of commits for SQL and NoSQL systems. We can see a significant difference in the number of commits between the two types of systems. SQL systems have a median of 4,501 and a mean of 7,066.5 commits. The maximum number of commits is 53,501 for SQL subject systems. On the other hand, for NoSQL systems, the median number of commits is 1,501, and the mean is 1,869.42. The maximum number of commits is 5,501 for NoSQL systems.

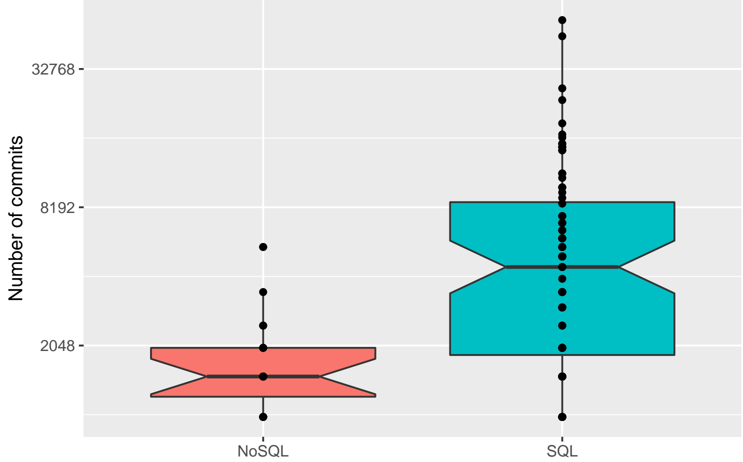


Figure 6.1 Distribution of the number of commits in SQL and NoSQL subject systems. The y-axis is on a log scale.

Table 6.1 Project groups

Group	Min. Commits	Max. Commits	NoSQL projects	SQL projects
$Group_1$	1001	1,500	12	21
$Group_2$	1,501	6,750	7	37
$Group_3$	6,751	53,501	0	26

The quantile analysis of the distribution of commits shows that 25% of the projects have less than 1,501 commits, 50% of the projects less than 3,001, and 75% of the projects less than 6,751 commits. We grouped the projects based on the quantiles into three for the purpose of visualization. Table 6.1 presents a summary of the systems in each group. For example, all projects with a maximum of 1,500 commits are included in $Group_1$, including 12 NoSQL and 21 SQL subject systems.

Table 6.2 Summary of the distribution of data-access and regular SATDs over the number of commits in Group 1 subject systems

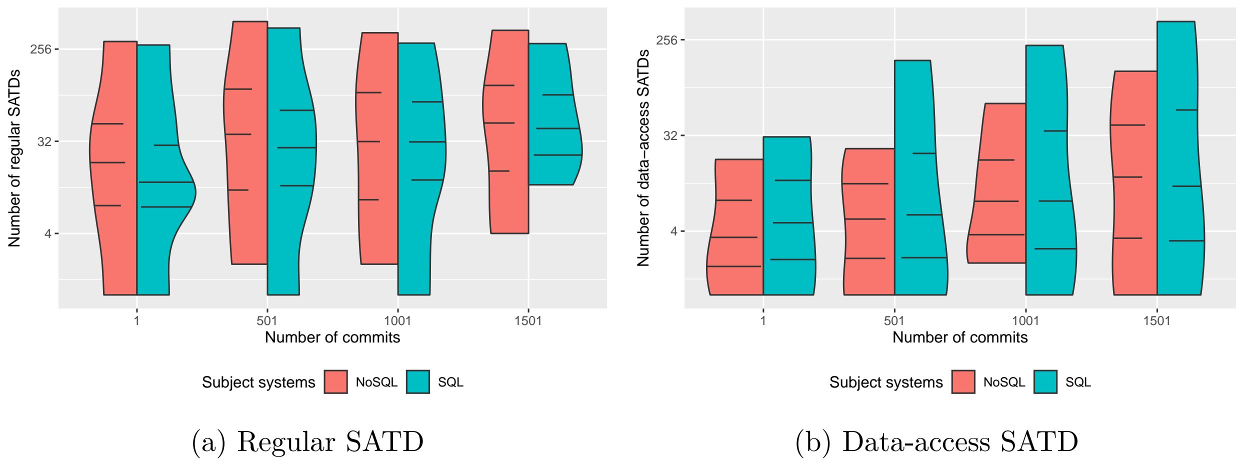
		Data-access SATD						Regular SATD					
Commit	System	Min	25%	Mean	**Median**	75%	Max	Min	25%	Mean	**Median**	75%	Max
1	NoSQL	0	0	1.92	**0**	0.25	19	1	8.25	47.83	**23.5**	47.25	304
	SQL	0	0	5.05	**0**	3.5	31	1	8	35.26	**12**	31.5	281
501	NoSQL	0	0	5.75	**1**	8.5	24	0	7.5	79.67	**38.5**	87	477
	SQL	0	1	17.20	**1.5**	8.25	163	1	13.75	57.75	**28.5**	64.25	412
1001	NoSQL	0	2	13.50	**4**	14.5	64	2	6	74.42	**35**	95.5	370
	SQL	0	1	27.76	**4**	17	226	1	13	63.67	**35**	74	293
1501	NoSQL	1	2	34.71	**22**	43.5	129	4	17	96.57	**71**	88	391
	SQL	1	2.5	63.17	**5.5**	75.25	380	12	20.5	73.67	**40**	84.5	290

Table 6.3 Summary of the distribution of data-access and regular SATDs over the number of commits in Group 2 subject systems

		Data-access SATD						Regular SATD					
Commit	System	Min	25%	Mean	**Median**	75%	Max	Min	25%	Mean	**Median**	75%	Max
1	NoSQL	0	0	0.57	**0**	0	4	4	5	21.14	**10**	29	66
	SQL	0	0	6.89	**0**	0.25	203	1	7	64.61	**24**	86.25	580
1001	NoSQL	0	0	7.00	**4**	12	21	2	7.5	30.86	**15**	46.5	91
	SQL	0	0	8.14	**0**	2	121	3	18	111.16	**40**	164	586
2001	NoSQL	0	3	13.14	**7**	11.5	56	0	15.5	35.43	**26**	50	91
	SQL	0	0	19.03	**2**	6	316	5	29	147.76	**51**	239	1015
3001	NoSQL	1	5.5	10.00	**10**	14.5	19	17	25.5	34.00	**34**	42.5	51
	SQL	0	1	31.30	**5**	9	506	4	37	160.37	**87**	224.5	857
4001	NoSQL	2	2	2.00	**2**	2	2	20	20	20.00	**20**	20	20
	SQL	0	1	40.17	**2.5**	10.5	555	24	40.75	169.83	**85**	220.25	923
5001	NoSQL	18	18	18.00	**18**	18	18	11	11	11.00	**11**	11	11
	SQL	0	1.5	54.13	**4**	12	588	3	39.5	179.60	**95**	266	941
6001	SQL	1	1.25	26.67	**2.5**	3	150	11	27.5	48.67	**33.5**	76.25	98

Table 6.4 Summary of the distribution of data-access and regular SATDs over the number of commits in Group 3 SQL subject systems

	Data-access SATD						Regular SATD					
Commit	Min	25%	Mean	**Median**	75%	Max	Min	25%	Mean	**Median**	75%	Max
1	0	0	3.46	**0**	0	60	4	27.5	203.96	**67.5**	153.75	1485
10001	0	1	71.00	**18**	50.5	519	99	203	552.47	**307**	648.5	2263
20001	0	2.5	10.00	**5**	15	25	177	183	189.33	**189**	195.5	202
30001	0	0.75	1.50	**1.5**	2.25	3	180	192.75	205.50	**205.5**	218.25	231
40001	0	0.75	1.50	**1.5**	2.25	3	202	223.75	245.50	**245.5**	267.25	289
50001	4	4	4.00	**4**	4	4	308	308	308.00	**308**	308	308



(a) Regular SATD

(b) Data-access SATD

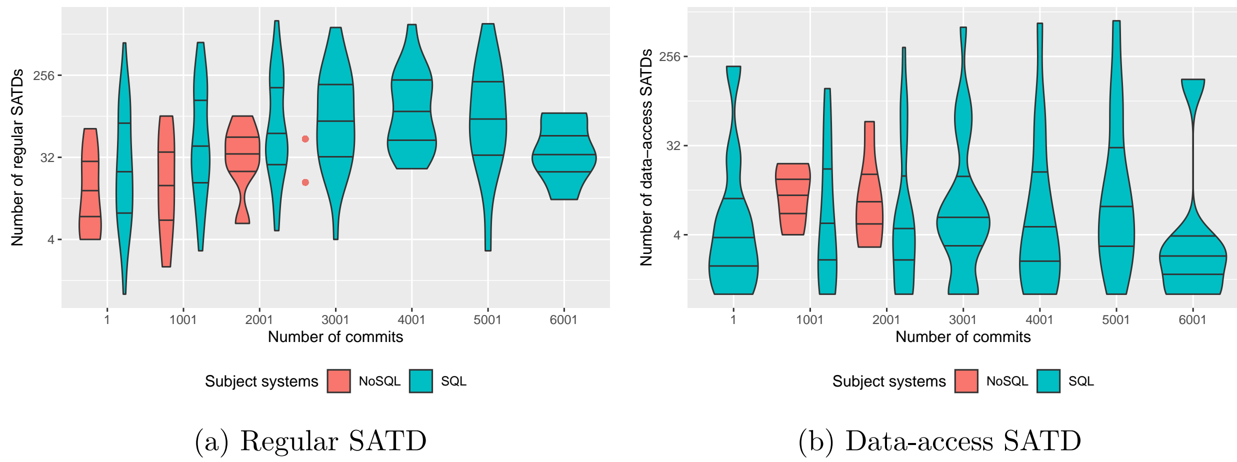
Figure 6.2 Prevalence of regular and data-access SATD in $Group_1$. The horizontal lines in this and subsequent violin plots show the 25%, median, and 75% quantiles respectively from bottom to top.

Tables 6.2, 6.3 and 6.4 show the summary of the distribution of SATDs in our subject systems by the project groups. The distribution was computed over the snapshots of the subject systems. Figure 6.2a shows the distribution of regular SATDs in $Group_1$. We observe that the number of regular SATDs increases for SQL systems as the number of commits increases. For NoSQL systems, an increase in the SATDs is observed, moving from 1 to 501 and 1,001 to 1,501. The median of regular SATDs in NoSQL systems (23.5, 38.5, 71) is higher than in SQL systems (12, 28.5, 40) for snapshots at commits 1 and 501 and 1,501, respectively. The highest number of regular SATDs (477) was observed at the 501st commit of a NoSQL system, *Bboss*.[1] Bboss is a framework that provides API support for developing enterprise and mobile applications.

Figure 6.2b shows the distribution of data-access SATDs in $Group_1$. The number of data-access SATDs in $Group_1$ increases with the number of commits. The median data-access SATD for SQL systems is 0, 1.5, 4, and 5.5 for commits 1, 501, 1,001 and 1,501. For NoSQL systems, the median is 0, 1, 4, and 22 for commits 1, 501, 1,001 and 1,501, respectively. We can see that the median of data-access SATDs is roughly similar between SQL and NoSQL subject systems except for commit 1,501, where we observe a large difference in magnitude between SQL and NoSQL subject systems. The highest number of data-access SATDs (380) was observed at commit 1,501 by the SQL subject system *Blaze-persistence*.[2] Blaze-persistence is a criteria API provider project for applications that rely on JPA for data

[1]https://github.com/bbossgroups/bboss
[2]https://github.com/Blazebit/blaze-persistence

persistence.



(a) Regular SATD

(b) Data-access SATD

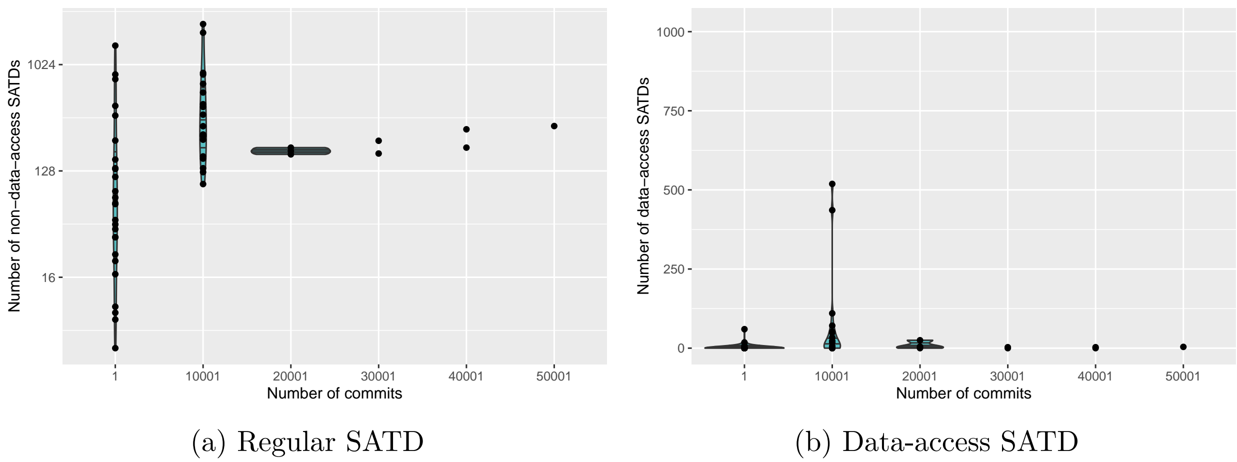
Figure 6.3 Prevalence of regular and data-access SATD in $Group_2$.

Figure 6.3a shows the distribution of regular SATDs in $Group_2$. We observe an increasing trend in the number of regular SATDs for both SQL and NoSQL systems. SQL systems have a higher median number of regular SATD in all snapshots. The median number of regular SATD of SQL systems is 24, 40, 51 and 87 for commits 1, 1,001, 2,001 and 3,001, respectively. For NoSQL systems, the median is 10, 15, 26 and 34, respectively. The maximum number of regular SATD (1,015) was registered in an SQL system, *Jena-sparql-api*,[3] at commit 2,001. *Jena-sparql-api* provides a SPARQL processing stack for building Semantic Web applications.

We observe a similar trend of increase in the number of data-access SATDs on $Group_2$, as shown in Figure 6.3b. The median number of data-access SATD in NoSQL systems is 0, 4, and 7 for commits 1, 1,001, and 2,001. SQL systems have a median number of data-access SATD 0, 0, and 2, respectively. The largest data-access SATD (588) was registered at commit 5,001 by SQL system *Threadfix*,[4] a software vulnerability management application.

Figure 6.4 shows the distribution of regular and data-access SATD in $Group_3$. We only have SQL systems in $Group_3$. After commit 10,001, we have two projects where we observe SATD, and only one project, WordPress-Android,[5] remains after commit 20,001. The violin plot is not needed for such cases. Figure 6.4a shows that the number of regular SATDs rises between commit 1 (median=67.5) and commit 10,001 (307), then decreases at 20,001 (189). The most significant regular SATDs (2,263) were observed at version 10,001 in *ControlSystemStudio*,[6]

[3]https://github.com/SmartDataAnalytics/jena-sparql-api
[4]https://github.com/denimgroup/threadfix
[5]https://github.com/wordpress-mobile/WordPress-Android
[6]https://github.com/ControlSystemStudio/cs-studio



(a) Regular SATD

(b) Data-access SATD

Figure 6.4 Prevalence of regular and data-access SATD in $Group_3$.

a repository of applications to operate large-scale industrial control systems. In Figure 6.4b, we can see an increasing median number of data-access SATDs (0, 18) at commits 1 and 10,001.

6.3 RQ 2.2: Persistence of SATDs in data-intensive systems

In RQ2.1, we showed that data-access SATDs are prevalent. In this RQ, we investigate the persistence of data-access SATDs as the subject systems evolve using survival analysis. We utilized the **SATD dataset** for this analysis. We present the details of our analysis approach and our findings in the following sub-sections.

6.3.1 Analysis approach

We analyzed the persistence of SATDs using survival analysis discussed in Section 2.2. There are two cases when we automatically check if SATDs in File X are addressed between two versions A and B. Case 1: if an SATD comment in File X is similar between version A and B, we consider it as “not fixed” at version B. Case 2: if the comment found in version A is missing in version B, we consider it “fixed” at version B.

Metrics for measuring developers activity in time

Code repositories track changes in software artifacts through commits. The distribution of commits in time co-relates to developer activity and is used to study the evolution of software and the associated technical debts (*e.g.,* [11, 102]). For our analysis, we took a

snapshot of projects every 500 commits. Figure 6.5 shows the distribution of the average time interval between successive snapshots of our subject systems. The average time interval between successive snapshots is 535 days for SQL subject systems and 423 days for NoSQL subject systems. The variation in time interval across and inside subject systems led to other approaches for measuring developer activity, such as using the number of commits.

We choose the number of commits over time in days for the survival analysis because different projects have different activities in time. As we discussed before (see Section 2.2), the number of commits suits better than time for our purpose to reflect the projects' activity [1]. While we use the number of commits to measure developer activity in our analysis, the above typical values can be used to interpret the commit time span in days.

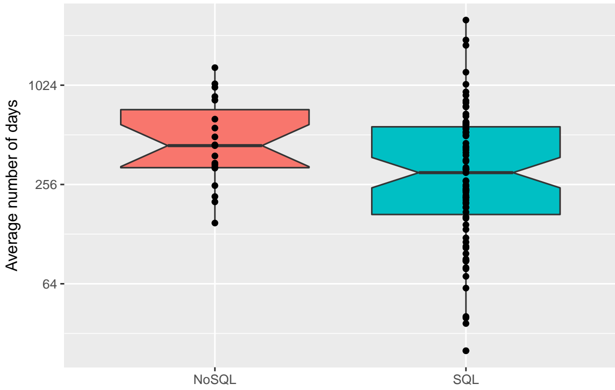


Figure 6.5 The distribution of average time interval between successive snapshots taken every 500 commits for SQL and NoSQL subject systems. The y-axis time unit is in days.

We used the Kaplan-Meir curve to visualize the survival of subject SATDs. The Kaplan-Meir curve shows the survival probability $S(t)$ of a given SATD at a time t. We define the addressing of a SATD as our event of interest. The occurrence of this event determines the survival probability. SATDs that persisted up to the latest versions and those removed with the source files are flagged as censored (see Section 2.2).

6.3.2 Findings

The results show that:

◇ **We found statistically significant differences between the survival curves of data-access and regular SATDs in both SQL and NoSQL systems, which indicates that data-access SATDs are fixed sooner than regular SATDs.**

Figure 6.6 shows the survival probability of data-access SATDs in SQL projects. The median

survival is 1,000 commits. Given the average value of 500 commit time span of 535 days for SQL subject systems, described in the analysis (Subsection 6.3.1), the average median survival time is 2.93 years. The steeper slope before 10,000 commits has two potential explanations. One possibility is that several data-access SATDs are fixed/censored at the early stages of the projects. Alternatively, several subject systems have a few commits. The distribution of the total number of commits (median=3,729, mean=7,005, skewness=3.07) of SQL subject systems is right-skewed. Hence, the steep slope is not likely due to small project activities. The number of "data-access SATD fixed" events is 3,914, with the remaining 608 being censored. This shows that many SATD comments are introduced and fixed at the early stages of the projects.

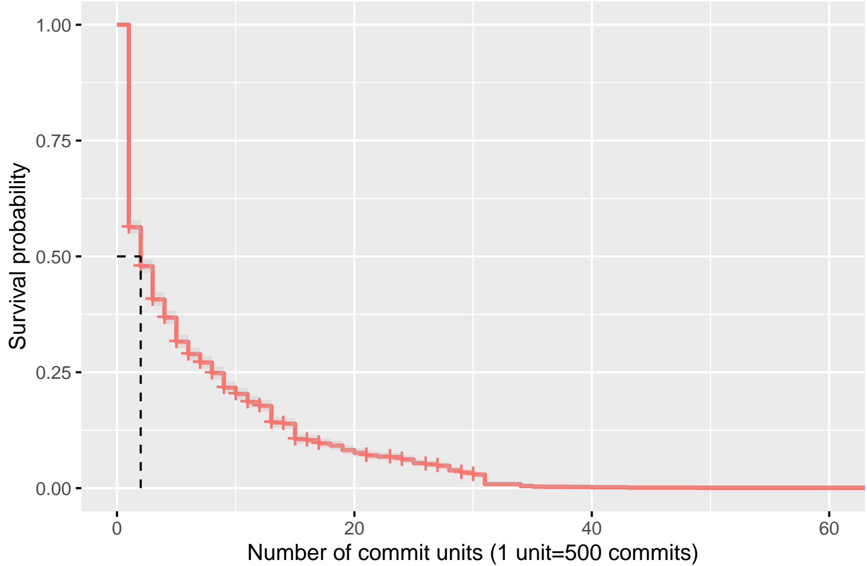


Figure 6.6 Kaplan–Meier survival curve for data-access SATDs in SQL subject systems. The x-axis is the number of commits. The censoring time and confidence intervals are marked on the plot. The Logrank test's p-value is indicated.

Figure 6.7 shows the survival probability of data-access SATDs in NoSQL subject systems. The median survival time of NoSQL data-access SATDs is 1,000 commits (2.3 years using an average 500 commit time span for NoSQL subject systems as described in Subsection 6.3.1). The number of events is 391 out of 441, with the remaining data being censored. The number of commits of NoSQL projects has a right-skewed distribution (median=1,927, mean=2,114, skewness=2.16). The smaller median survival value aligns with the smaller median number of commits of NoSQL subject systems.

Figure 6.8 compares the survival curves of data-access and regular SATD comments in SQL systems. This comparison provides an insight into the prioritization of addressing technical

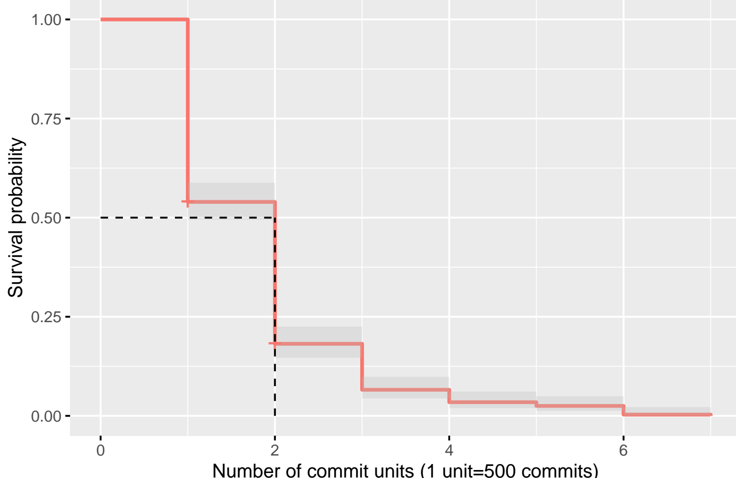


Figure 6.7 Kaplan–Meier survival curve for data-access SATDs in NoSQL subject systems. The x-axis represents the number of commits. The censoring time and confidence interval are marked on the plot. The Logrank test's p-value is indicated.

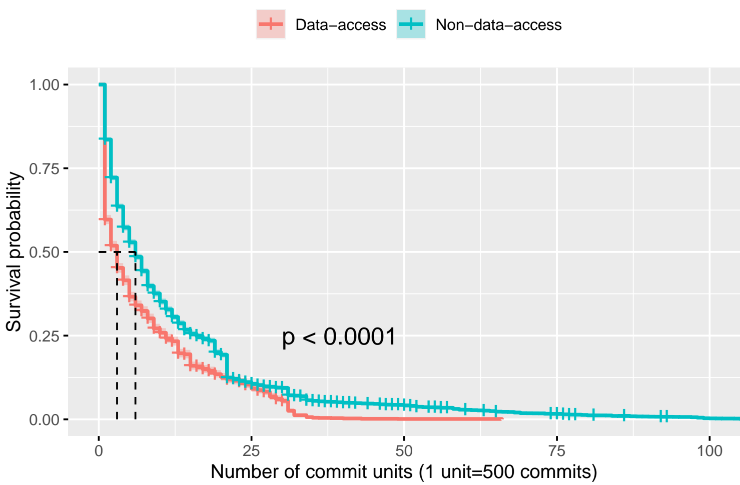


Figure 6.8 Kaplan–Meier survival curve for SQL subject systems by grouping them into data-access and regular SATD comments. The x-axis represents the number of commits. The censoring time is marked on the plot.

debt. Data-access comments have a lower survival curve compared to their regular counterparts. We run the Log-Rank test to compare the survival curves statistically. The p-value of the log-rank test is $< 2e - 16$. Hence, we can reject the null hypothesis that there is no difference between the survival curves of data-access and regular SATD comments. Data-access SATDs tend to get more priority in addressing compared to regular SATDs.

Similarly, Figure 6.9 shows for NoSQL subject systems that data-access SATDs tend to get

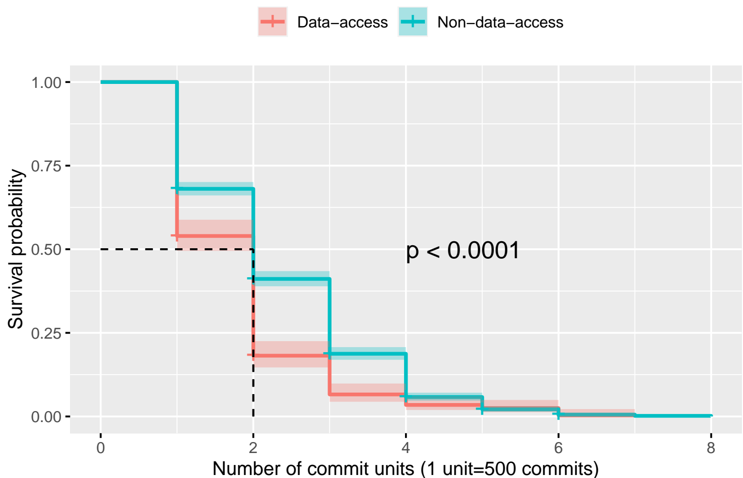


Figure 6.9 Kaplan–Meier survival curve for NoSQL subject systems by grouping them into data-access and regular SATD comments. The x-axis represents the number of commits. The censoring time is marked on the plot.

fixed quicker than regular SATDs. The Log-Rank test's p-value was $< 2e - 16$. Hence, we can reject the null hypothesis that there is no difference in survival curves.

⋄ **A significant number of data-access SATDs introduced in the first versions of the systems (5% for SQL and 7% for NoSQL systems). Many persisted until the latest versions (68% for SQL and 39% for NoSQL**

Many data-access SATD comments are introduced in the first versions of the systems, and several of them persisted until the latest versions. For SQL systems, 223 (4.93%) comments were introduced in the first version, and 152 (68.16%) persisted until the latest version. For NoSQL systems, 31 (7.02%) comments were introduced in the first version, out of which 12 (38.7%) lasted in all versions.

6.4 RQ 2.3: Circumstances behind the introduction and removal of data-access SATD

We present the result of our quantitative and qualitative analysis of data-access SATDs to determine the circumstances behind the introduction and removal of such anti-patterns. The findings from this RQ provide insight into what type of changes introduce or remove data-access SATDs and when. We utilized the **SATD dataset** for this analysis.

6.4.1 Analysis approach

In our analysis, we use the introduction or removal of SATD comments as a proxy to the introduction or removal of SATDs, respectively. We are particularly interested in investigating when and why the data-access SATDs are introduced and removed. Hence, we first identified the SATD introduction and removal commits and then computed the commit time. Then we conducted manual labeling of the commit messages. We outline the details of our analysis in the following paragraph.

Identify SATD introducing and removing commits

Using this labeled data from **RQ 1.1**, we extracted the commits that introduced the comments and commits that removed them from the change history of the subject systems. We used the PyDriller repository mining framework [120] for our analysis. PyDriller is used to analyze both local and remote repositories and extract information related to their change history. We looked for the *SATD introducing commit* given the path of a file by looking at the change history starting from the beginning to the end and looking for the first occurrence of the SATD under study. Similarly, we looked for the *SATD removal commit*, the commit in which a SATD is removed from the system, by looking for the first commit in which the SATD is no longer present given that the SATD occurred in the previous versions. To check if the SATD is removed together with the hosting class, we also keep track of the commit where the hosting class is removed (if it is removed).

When are data-access SATDs introduced or removed?

For our purpose, the number of commits is better than the absolute time at reflecting software evolution (see sub section 6.3.1). Hence, we measure introduction time and removal time in terms of number of commits.

We define *introduction time* (t'_i) as the number of commits that occurred before and including the first occurrence of the SATD under study. Similarly, we define *removal time* (t'_r) as the number of commits that occurred before and including the commit that removed the SATD. t'_i and t'_r are measured in the number of commits.

Since the total number of commits varies across the projects, we normalize the *introduction time* and *removal time* with the total number of commits for each subject system (see Equations 6.1 and 6.2). We use a similar normalization for the removal time. For example, a SATD introduction time of 20% for a project with 1,000 commits means the SATD was introduced in the 200th commit from the beginning. The smaller the value, the closer the

introduction of SATD to the early stages of the project evolution and vice versa.

$$\textit{Introduction time} = \frac{t_i' \cdot 100}{\textit{Total number of commits}} \tag{6.1}$$

$$\textit{Removal time} = \frac{t_r' \cdot 100}{\textit{Total number of commits}} \tag{6.2}$$

We use *Introduction time* and *Removal time* to investigate when SATDs are introduced or removed.

Why are data-access SATDs introduced or removed?

To investigate why data-access SATDs are introduced, we collected the commit messages of SATD introduction and removal commits, then manually categorized their goal or purpose. We use similar categories to Tufano et al. [142]: *bug fixing*, *enhancement*, *new feature*, and *refactoring*. In our case, we added *merging* and *multiple goals* to account for merging commits and commits whose messages have more than one goal. In this way, the commit goal can be mapped to more than one of the categories from Tufano et al.. *Bug fixing* commits mention that the commit was made to fix an existing bug or issue. *Enhancement* commits aim at enhancing existing or already implemented features. Commits with the goal *new feature* describe their goal as introducing or supporting a new functionality. Commits that mention refactoring operations are categorized under *refactoring*. Finally, commits made for merging pull requests and branches are categorized under *merging*. We labeled SATD removing commits similarly.

6.4.2 Findings

The results show that:

⋄ **Most SATD comments in data-access classes are introduced at the later stages of change history. However, SATD comments where database access is explicitly mentioned (i.e., database access related categories in the taxonomy) are introduced earlier than SATD comments unrelated to database access**

Figure 6.10 shows the overall distribution of data-access SATD introduction time. The distribution is right-skewed, with the median introduction time (72.53%) and mean (64.14%). This indicates that most of the data-access SATD introducing commits did not happen at the beginning of the change history. This also confirms our observation of the survival analysis

SATDs. Data-access SATDs seem to be introduced at later stages in the change history. We also identified SATDs committed in the most recent snapshots of the subject systems (introduction time=100%).

◇ **We observed similar distribution between SQL and NoSQL data-access SATDs in introduction time.**

Figure 6.11 shows the distribution of introduction time for SQL and NoSQL systems. For both SQL and NoSQL data-access SATDs, introduction time is right-skewed. The notches of the SQL and NoSQL overlap, which means that the difference in the median is not significant. SQL data-access SATDs have a slightly higher median (80.31%) than in NoSQL systems (71.67%).

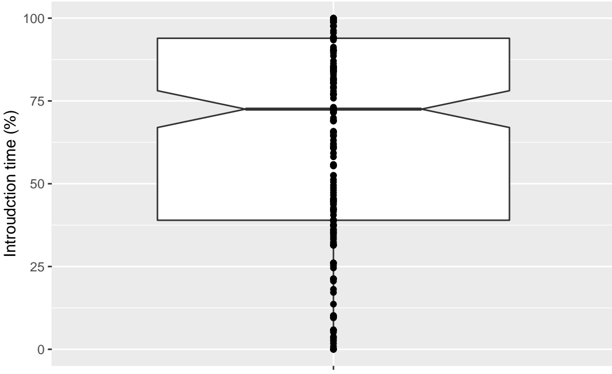


Figure 6.10 Distribution of data-access SATD introduction time

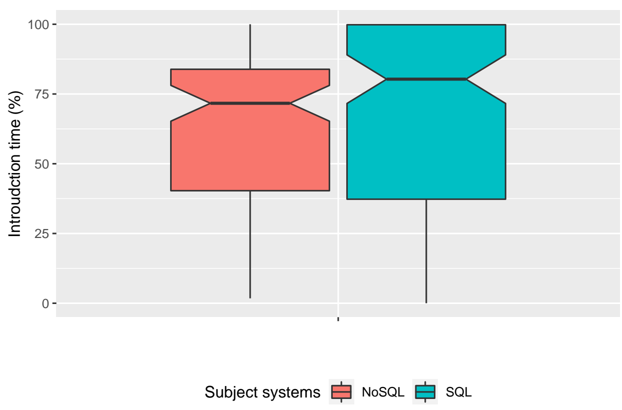


Figure 6.11 Distribution of data-access SATD introduction time in SQL and NoSQL subject systems

Table 6.5 shows the number of comments, mean, and median introduction time for all data-

access SATD categories. The categories are ordered by the median introduction time from highest to lowest. *Low external quality* and *design patterns* data-access SATDs are introduced in the latest stages of change history among all the categories. On the other extreme, most of the database access related SATDs tend to be introduced at the early stages of change history. Compared to regular SATDs, most of the database access related SATDs are introduced earlier. *Transactions*, *indexes*, and *data-access test debt* tend to be introduced at later stages. *Addressed technical debt* comments tend to be introduced at the very beginning of the subject systems' development.

Table 6.5 Data-access SATD introduction time for SATD categories

Category	**Comments**	**Mean**	**Median**
🗋 Low external quality	18	81.34	99.83
🗋 Design patterns	1	99.83	99.83
🗋 Performance	3	87.86	82.96
🗋 Workaround	23	62.76	82.12
🛢 Data-access test debt	8	74.76	81.31
🗋 Known defects to fix	25	75.24	80.43
🗋 New features to be implemented	21	64.25	80.43
🗋 Code smells	16	72.08	77.28
🛢 Transactions	2	72.62	72.62
🛢 Indexes	1	72.53	72.53
🗋 Document commented code	5	49.15	72.31
🗋 Test debt	15	71.10	71.67
🗋 Improvement to features needed	30	59.89	70.66
🗋 Known defect of external library	2	68.15	68.15
🗋 Multi-label	4	63.09	64.26
🛢 Localization	1	61.19	61.19
🗋 Low internal quality	40	56.84	58.75
🗋 Documentation needed	3	66.39	50.54
🗋 On hold	5	46.73	48.09
🛢 Due to database schema	1	47.30	47.30
🛢 Data synchronization	2	45.64	45.64
🛢 Query execution performance	5	48.93	44.68
🛢 Known issue in data access library	2	43.57	43.57
🛢 Query construction	7	48.08	37.29
🗋 Partially fixed defects	1	21.30	21.30
🗋 Addressed technical debt	3	28.90	2.51

⋄ **The median removal time is 99.58% for SQL and 98.58% for NoSQL data-access SATDs indicating that SATD's persist thought the revision history of the**

systems without getting addressed.

We found 12 data-access SATDs that were removed at different stages of the change history. Figure 6.12 shows the distribution of data-access SATD removal time for SQL and NoSQL subject systems. Both SQL and NoSQL SATDs were removed at the latter stages, close to the most recent versions.

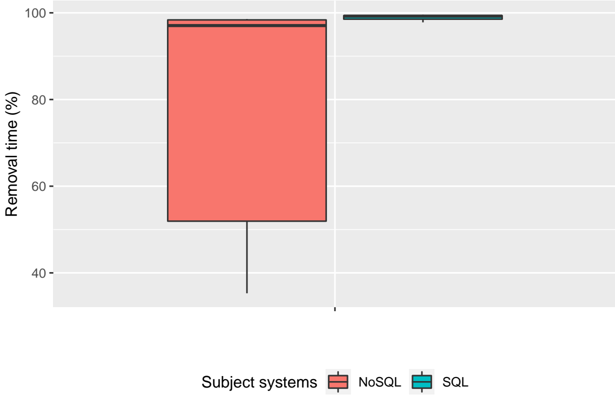


Figure 6.12 Distribution of data-access SATD removal time in SQL and NoSQL subject systems

Table 6.6 Distribution of data-access SATD removal time among the data-access categories

Category	Comments	Mean	Median	Minimum	Maximum
🗋 Improvement to features needed	2	99.47	99.47	99.36	99.58
🗋 Code smells	1	98.54	98.54	98.54	98.54
🗋 Known defects to fix	2	98.19	98.19	97.80	98.58
🗋 Test debt	1	98.15	98.15	98.15	98.15
🗋 Low internal quality	5	77.34	97.06	35.29	99.22
🗋 Document commented code	1	47.32	47.32	47.32	47.32

Table 6.6 shows the distribution of data-access SATD removal time grouped by categories. We did not have any removed comments from the *database access related* SATD category. *Improvement of features needed* comments tend to be removed at later stages of change history with the highest median removal time of 99.58%. On the other hand, *document commented code* comments were introduced in the middle stages of the change history (median=47.32%).

◇ ***Bug fixing* and *refactoring* are the main reasons behind the introduction of data-access SATDs, followed by *feature enhancement* and *supporting new features.***

We now focus on the potential reasons for data-access SATDs' introduction and removal. We manually labeled the data-access SATDs' introducing/removing commit messages to classify

their purposes. We classified the goal of the commit messages as *bug fixing*, *enhancement*, *new feature*, *refactoring*, and *merging*. Some commit messages described *multiple goals*, and some comments were labeled *unclear* as they did not contain enough information in the commit message for categorization.

Table 6.7 Data-access introducing commit goals in NoSQL and SQL subject systems

Systems	**Bug Fixing**	**Enhancement**	**Multiple Goals**	**New Feature**	**Refactoring**	**Unclear**	**Merging**
NoSQL	17	22	3	30	38	5	0
SQL	45	6	3	32	38	1	4

Table 6.7 summarizes the various goals of data-access SATDs' introductions. Considering NoSQL data-access SATDs, *refactoring* is the most associated reason with 38 instances (33.04%). It is followed by *new feature* with 30 cases (26.09%) and 22 *enhancements* (19.13%). For SQL, *bug fixing* was the most often mentioned reason in comments, with 45 instances (34.88%). It is followed by *refactoring* with 38 cases (29.46%) and 30 *new features* (23.26%). Overall, *bug fixing* and *refactoring* are the main reasons behind the introduction of data-access SATDs.

Table 6.8 Data-access SATD introducing commit goals grouped by data-access SATD categories

Categories	**Bug Fixing**	**Enhancement**	**Multiple Goals**	**New Feature**	**Refactoring**	**Unclear**	**Merging**
Low internal quality	11	7	2	4	14	2	0
Workaround	4	3	1	6	9	0	0
On hold	1	1	3	0	0	0	0
Due to database schema	0	1	0	0	0	0	0
Query execution performance	1	0	1	1	2	0	0
Transactions	1	0	0	1	0	0	0
Known issue in data-access library	0	0	0	1	1	0	0
Data synchronization	0	0	0	0	2	0	0
Indexes	0	0	0	1	0	0	0
Localization	1	0	0	0	0	0	0
Query construction	0	1	0	5	1	0	0
Known defect of external library	1	1	0	0	0	0	0
Known defects to fix	7	3	1	5	8	1	0
Low external quality	9	0	0	3	4	0	2
Partially fixed defects	0	0	0	0	1	0	0
Code smells	5	1	0	5	5	0	0
Design patterns	1	0	0	0	0	0	0
Document commented code	3	0	0	1	0	1	0
Documentation needed	2	0	0	0	1	0	0
Addressed technical debt	1	0	0	0	2	0	0
Multi-label	1	1	0	1	0	0	0
Improvement to features needed	7	2	0	10	10	1	0
New features to be implemented	2	3	0	5	9	1	1
Performance	1	0	0	0	2	0	0
Test debt	1	5	1	6	2	0	0
Data-access test debt	2	0	1	3	2	0	0

Table 6.8 shows the introduction goals grouped by data-access debt categories. In general, *refactoring*, *new feature* and *bug fixing* appear to be the most common reasons. However,

only considering the database access related SATDs, they are mainly introduced during *refactoring*. Another interesting observation is that *code smells* are introduced during *refactoring* (31.25%), *bug fixing* (31.25%) and *new feature* (31.25%). This means that refactoring, which is supposed to fix code smells, could also introduce other code smells and SATDs.

◇ **Data-access debt removal commits are often associated with *feature enhancements*, *new features*, and *bug fixing***

We present the removal goals of SATD categories in Table 6.9. *Low internal quality* is associated with *enhancement* (60%) and *new feature* (40%). The remaining SATD categories have 6 instances combined.

Table 6.9 Data-access SATD removing commit goals grouped by data-access SATD categories

Category	**Commit Goal**	**Comments**
Low internal quality	Enhancement	3
	New Feature	2
Known defects to fix	Enhancement	1
	New Feature	1
Code smells	Unclear	1
Document commented code	Bug fixing	1
Improvement to features needed	Bug fixing	1
	New Feature	1
Test debt	Bug fixing	1

Table 6.10 Data-access SATD removing commit goals for SQL and NoSQL subject systems

Commit Goal	**Enhancement**	**New Feature**	**Bug Fixing**	**Unclear**
SQL	1	2	1	1
NoSQL	3	2	2	0
Total	**4**	**4**	**3**	**1**

Table 6.10 summarizes the goals of the removals of data-access SATDs. Several comments were removed for feature *enhancements* and *new features*. *Bug fixing* commits also contribute to the reduction of data-access SATD. Both SQL and NoSQL systems follow a similar distribution of commit goals.

6.5 Discussion

We investigated the prevalence, persistence, and introduction and removal circumstances. The results show that SATDs are prevalent in data-intensive systems, and their prevalence

increases as systems evolve. This pattern is similar to traditional software systems. Bavota and Russo [1] showed that SATDs are prevalent and increase as new ones are introduced during software evolution. This indicates that in both traditional and data-intensive systems, developers tend to introduce new SATDs instead of addressing existing ones. In addition, our results show that the prevalence of SATDs is different between SQL and NoSQL data-intensive systems. Given that NoSQL persistence systems are getting higher preference due to the advantages they offer in terms of schema flexibility and scalability and our result showing more prevalent SATDs in some NoSQL-based systems, our findings motivate further investigation of the impact of the persistence technologies on SATD.

Our results regarding the persistence of SATDs in data-intensive systems are similar to traditional systems. Bavota and Russo [1] found that the median survival time of SATDs to be 1000 commits for traditional software systems. We also find similar median survival times for both SQL and NoSQL subject systems. On the other hand, Maldonado et al. [106] reported that SATDs persist up to 173 days on average using five open-source traditional software systems. This implies that SATDs in data-intensive systems have even higher persistence (more than two years on average in our case). We also found that a significant number of SATDs persisted in all versions without getting addressed. Since the longer the SATD stays in the system, the higher the cost of repaying, practitioners should incorporate fixing technical debts as part of their workflow. This result highlights the importance of research work in SATDs in terms of providing tool support, raising awareness of the costs of technical debts, and providing processes and frameworks for monitoring technical debt.

Our fine-grained analysis on data-access SATDs showed that most data-access SATD comments are introduced as the subject systems evolve rather than at the initial stages, indicating that they are introduced as a result of software evolution. A software system can evolve for various reasons such as bug fixing, adding new features, improving features, and refactoring activities. Developers should take care to assess the cost of the SATD they introduce with such activities. Our results also show that the introduction of data-access SATDs is mainly associated with refactoring. However, this motivates further investigation on how and why refactoring operations are associated with SATDs. This could be done by extracting refactoring information using refactoring detection tools and co-relating with the SATD's introduced. This, in turn, leads to the development of refactoring tools that also suggest developers when to admit technical debts.

6.6 Threats to validity

In this section, we discuss threats to research validity related to our analysis and findings regarding characterization of data-access SATDs (RQ 2.1, RQ 2.2 and RQ 2.3).

6.6.1 Threats to construct validity

Threats to construct validity concern the relation between theory and observation. We relied on a list of keywords and import statements to select subject systems and distinguish data-access classes (DAC) from non-data-access classes (NDC) within those systems. We may have missed some keywords and import statements, which would lead us to overestimate the set of NDCs. Conversely, it is possible that some classes are considered as DACs (*i.e.,* that import database-related packages belong to our list), but do not (directly) query the database. Hence, we may also slightly overestimate the actual set of DACs in the software systems considered. We checked 100 randomly selected data-access classes and found that 82% of those directly query the database.

Another threat to construct validity is the precision of the SATD detector tool. The 73.7% F_1 score shows that the tool could introduce a significant number of false positives. Indeed, we conducted a manual analysis and identified a considerable number of false positives. However, The SATD detector is a state-of-the-art tool whose base approach was also used in other studies (*e.g.,* [1]). Improving the accuracy of the SATD detector is out of the scope of the paper. However, the conclusions from this paper are carefully formulated and need to be interpreted taking into account the imprecise nature of the tool.

There might be cases when SATD comments are removed without code changes in effect. This may mean that the SATD admitted earlier is no longer viewed as technical debt by the developers, or they may not be interested in keeping track of that SATD [35]. Such cases are not actual removals of SATDs. Zampetti et al. [35] conducted an empirical study on Java open-source systems and observed that such cases are not frequent ($< 10\%$) in most cases and the maximum being 17%.

We used the number of commits as a metric to measure developer activity instead of time due to the variations in commit time span across subject systems and in between different snapshots of a subject system. However, the number of commits may not accurately represent the time spent by developers on technical debt. To help mitigate this threat, we provided the typical 500 commit time span for each subject system in the replication package as an indication of time.

6.6.2 Threats to internal validity

Internal validity concerns how one can be confident on claimed cause and effect relation. We did not claim any causation in our study. We only analyzed the diffusion and survival of SATD in SQL and NoSQL subject systems. Hence, our study is not subjected to threats to internal validity.

6.6.3 Threats to conclusion validity

Conclusion validity concerns the degree to which the statistical conclusions about the claimed relationships are reasonable. To avoid conclusion threats to validity, we only used non-parametric statistical tests.

6.6.4 Threats to external validity

External validity concerns the generalizability of findings outside the study context. Our study considers different types of projects in terms of database technology (SQL or NoSQL), application domain, size, and the number of database interactions. We also covered projects that use different drivers and frameworks to interact with the database. We only considered Java projects for analysis. However, our investigation approach is generalizable to other programming languages.

6.6.5 Threats to reliability validity

Reliability validity concerns factors that cause an error in data collection and analysis. To minimize potential threats to reliability, we analyzed open-source projects available on GitHub and provided a replication package that contains our dataset and analysis reported in this chapter scripts [140].

6.7 Chapter summary

In this chapter, we investigated the prevalence and evolution of data-access SATDs using traditional SATDs as a baseline. We also performed quantitative and qualitative analyses of data-access SATDs to understand the circumstances behind the introduction and removal of data-access SATDs. Results show that data-access SATDs are introduced as software gets more mature, and many instances of SATDs persist for a more extended time.

Bug fixing and refactoring are the main reasons behind the introduction of data-access SATDs followed by feature enhancements and new features. The observed SATD removal activities

are not associated with refactoring, which implies that the removals are merely parts of bug fixing or feature enhancement activities. SATDs in general and data-access SATDs, in particular, are critical to data-intensive systems as they determine the quality of the subject systems in terms of robustness and efficiency of data-access operations.

Supporting more functionalities and maintaining code quality at the same time is a general problem for any software system. Having the right balance would help maintain software quality and reduce technical debt costs in the long run.

CHAPTER 7 CHARACTERIZATION OF SQL CODE SMELLS

7.1 Chapter overview

In this chapter, we present the analysis approach and findings regarding the characterization of SQL code smells to achieve **Sub-objective 2.2**. While SQL code smells are not the only technical debts under data-access non-SATDs, only SQL code smell detection tool is available to the best of our knowledge. Hence, we restricted our analysis to SQL code smells. However, this study can be replicated for other types of non-SATDs once the detection tools are available. We present the analysis and result of research questions **RQ 2.4, RQ 2.5, and RQ 2.6**.

RQ 2.4: **What is the prevalence of SQL code smells across different application domains?**

> **Main finding:** ***Implicit Columns*** smell is the most prevalent SQL code smell in the data-intensive systems across four application domains, followed by the *Fear of the Unknown* smells. The remaining two SQL code smells are not prevalent in the 150 subject systems under our study.

RQ 2.5: **Do traditional code smells and SQL code smells co-occur at class level?**

> **Main finding:** Several traditional code smells (e.g., LongParameterList) and SQL code smells (e.g., Implicit Columns) could co-occur within the data-intensive subject systems. However, their association is rather weak according to multiple statistical tests.

RQ 2.6: **How long do SQL code smells survive?**

> **Main finding:** SQL code smells have higher tendency to survive for longer period of time compared to traditional code smells. A large fraction of the source files affected by SQL code smells (80.5%) persist throughout the whole snapshots, and they hardly get any attention from the developers during refactoring.

7.2 RQ 2.4: Prevalence of SQL code smells

In this section, we present the analysis approach and results about the prevalence of SQL code smells on data-intensive systems. Studying the prevalence of SQL code smells is the

first step to characterize SQL code smells. If the smells are not prevalent, then there is no need to further investigate their evolution or impact. If SQL code smells are prevalent, then further characterization is needed.

7.2.1 Analysis approach

To study the prevalence of SQL code smells in data-intensive systems, we started with the code smells dataset, and we collect SQL code smells from each of the projects (i.e., latest tracked versions) and provide the summary statistics on code smells for each of the four application domains described in Section 4.2.

Our projects from each domain have varying complexity in terms of project size and interactions with the database. We measure *prevalence* of SQL code smells *as the ratio between total number of SQL code smells and total number of database access queries in a subject system.*

7.2.2 Findings

The results show that:

◇ ***Implicit Columns* was the most prevalent data-access smell across all projects followed by *Fear of the Unknown.***

We detected four types of SQL code smells described in Section 2.1.1 with SQLInspect in our data-intensive subject systems. Out of these four smell types, *Implicit Columns* was the most frequent across all projects, with a median prevalence of 1.67%. That is, out of every 100 database access queries, this smell affects two queries. The second most frequent code smell – *Fear of the Unknown* – has a median prevalence of 0.8%. We did not find any *Ambiguous Groups* or *Random Selection* in the most recent tracked version of our subject systems. However, our analysis identified a few *Ambiguous Groups* code smells in the older versions of the systems.

◇ **Business and utility applications contain higher median prevalence of *Implicit Columns* smells among all the studied application domains.**

We analyze the prevalence of *Implicit Columns* across four application domains. Table 7.1 and Fig. 7.1 summarize our findings. From Table 7.1, we see that projects from *Business* and *Utility* domains have the highest median prevalence of 2.98% and 1.68%. Fig. 7.1 further shows the distribution of prevalence for *Implicit Columns* across the application domains. We see that *Business* and *Library* have the highest median and 75% quantile in the prevalence

ratio measure. We also investigated the nature of the outlier projects from Library domains, as shown in the box plot of Fig. 7.1. We notice that the Library project with the highest prevalence ratio, *Tablesaw* data visualization library, has 45 SQL queries, out of which 31 queries are smelly. This project has more than 2K stars and 39 contributors. The second highest in prevalence, *calcite-elasticsearch*, is another library project that has a total of 167 SQL queries, out of which 79 queries are smelly. Since library projects are often reused by other applications, the impact of these SQL code smells could be much more serious. In both Figure 7.1 and Table 7.1, we see that SQL code smells such as *Implicit Columns* have the least prevalence in the subject systems from *Multimedia* domain.

Table 7.1 Prevalence of Implicit Columns across four application domains

Domain	**Median Prevalence**	**Mean Prevalence**
Business	2.98%	8.49%
Multimedia	0.23%	5.47%
Utility	1.68%	5.27%
Library	0.75%	7.93%

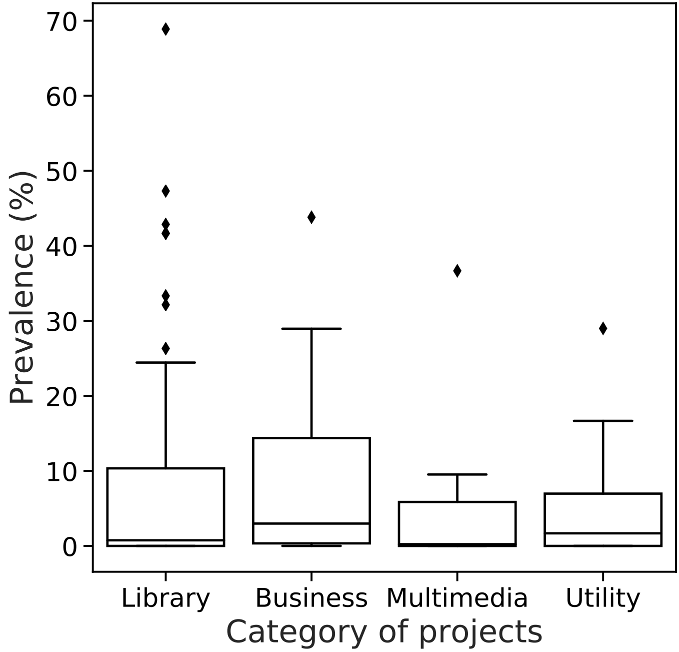


Figure 7.1 Prevalence of SQL code smells (Implicit Columns) across different application domains

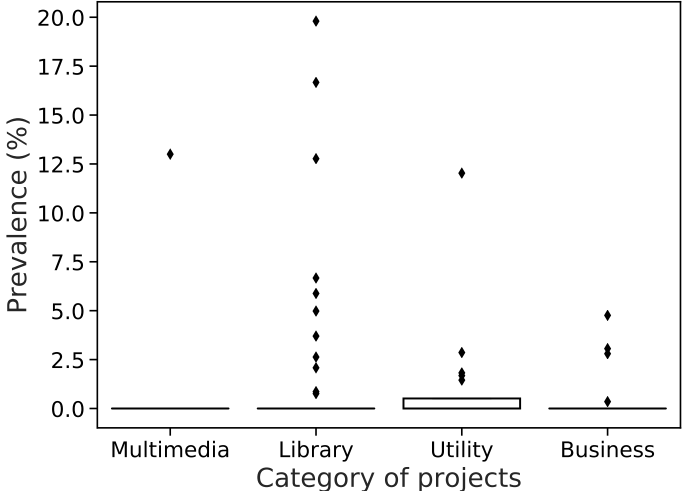


Figure 7.2 Prevalence of SQL code smells (Fear of the Unknown) across different application domains

◇ **The median prevalence of *Fear of the Unknown* is zero on all applications. However, some business and utility applications show higher prevalence of *Fear of the Unknown.***

We further analyze the distribution of prevalence of *Fear of the Unknown* SQL code smell across the four application domains. Fig. 7.2 shows our prevalence ratio distribution for this smell. We see that the median prevalence for all domains is zero. However, there exist a significant number of outlier projects in the library, business and utility domains that we analyze. The project with the highest prevalence of *Fear of the Unknown* smell is a real-time chat and messaging Android SDK library, *Applozic-Android-SDK*, that has at least 295 forks and 18 contributors. The project has 202 SQL queries in the most recent tracked version, out of which 40 queries are affected with the target smell. All our analyses above show that *Implicit Columns* and *Fear of the Unknown* are the two prevailing SQL code smells across all four application domains.

◇ **Large number of *Implicit Columns* smell instances retrieved three or more columns that were not utilized.**

We also randomly selected 10 projects and manually investigated 98 *Implicit Columns* of smells from them. We found that at least 70% of these smelly SQL queries retrieved three or more columns that were unused and 15% retrieved nine or more table columns that were unused. Such a counter-productive data access could lead to a performance bottleneck. *Implicit Columns* smells might also create unnecessary coupling between a front-end and its

back-end database, which could negatively affect the maintainability of the system. Although the prevalence of *Fear of the Unknown* smell is not as high as for *Implicit Columns*, their impact on maintenance and performance could also not be ignored.

7.3 RQ 2.5: Co-occurrence of traditional code smells and SQL code smells

In this section, we present the co-occurrence analysis between SQL code smells and traditional code smells. We utilized the Apriori algorithm and Cramer's V test of association to measure the co-occurrence. Investigating the co-occurrence between SQL code smells, and traditional code smells will provide insights into the potential interaction between data-access technical debts and traditional technical debts.

7.3.1 Analysis approach

To study the co-occurrence of SQL code smells with traditional code smells, we utilized the **code smells dataset** (described Section 4.2) and applied Apriori association rule mining algorithm described in Section 2.4. For Apriori analysis, we consider each entry (i.e., a record from our dataset that has at least one database access query) containing code smell statistics as a transaction. Then, the frequent item sets were generated from all the transactions that involve traditional code smells and SQL code smells. Besides the Apriori algorithm, we employ Cramer's V association test described in Section 2.5 to collect numerical, comparable association values between these two classes of code smells.

7.3.2 Findings

The results show that:

◇ **Implicit Columns smells co-occur with LongMethod across among all application domains.**

Table 7.2 shows the statistics on file versions for each application domain. We see that business systems have the highest number of file versions that deal with database access, while multimedia systems have the lowest number. Business systems have more database interactions since they are often involved in data processing and data visualization. We have only 11 Multimedia systems in our dataset, which might explain their low number.

We use Apriori algorithm for determining the association (co-occurrence) between traditional code smells and SQL code smells.

To generate frequent item sets, we selected a minimum support of 0.01 (1%) considering the

Table 7.2 Source code file versions with database access

Application Domain	**# File Versions**
Business	16,225
Library	11,839
Multimedia	156
Utility	1,153

small number of occurrences of SQL code smells compared to that of traditional code smells. We also restrict the maximum number of items in every item set to 2 since we were interested in the association between one traditional smell and one SQL code smell. We also set the minimum lift threshold to 1 to generate the *relevant* association between SQL code smells and traditional code smells.

Table 7.3 shows our frequent item sets where each item set consists of one traditional code smell and one SQL code smell. When all subject systems are considered, we see an association (i.e., Lift$>$ 1.00) between *Implicit Columns* and *LongMethod*. We also repeat the same experiments for each of the four application domains. We see that *Implicit Columns* smells co-occur with *LongMethod* across both business and library domains.

They also co-occur with *ComplexClass* in all application domains except business. However, the leverage value is close to zero for each of the mined association rules, which indicates that the association between SQL code smells, and traditional code smells is not strong.

Table 7.3 Top-3 SQL code smells, and traditional code smells based on lift value across the application domains. A leverage value close to 0 indicates weak association.

Application Domain	**Smell Pairs**	**Support**	**Confidence**	**Lift**	**Leverage**	**Conviction**
Combined	Implicit Columns:LongMethod	0.0507	0.528	1.03	**0.0015**	1.0336
Business	Implicit Columns:ComplexClass	0.0169	0.445	1.2169	**0.003**	1.1429
	Implicit Columns:LongMethod	0.0207	0.5437	1.031	**0.0006**	1.0358
Library	Implicit Columns:LongParameterList	0.0295	0.1804	1.0261	**0.0007**	1.0056
	Implicit Columns:LongMethod	0.0854	0.5228	1.0377	**0.0031**	1.0398
Multimedia	Fear of the Unknown:AntiSingleton	0.01923	0.2143	5.5714	**0.0158**	1.2238
	Fear of the Unknown:ComplexClass	0.0705	0.7857	2.7238	**0.0446**	3.32
	Implicit Columns:ComplexClass	0.1474	0.5476	1.8984	**0.0698**	1.5729
Utility	Fear of the Unknown:AntiSingleton	0.0208	0.31169	4.6074	**0.0163**	1.3545
	Fear of the Unknown:LongParameterList	0.01908	0.2857	1.8	**0.0085**	1.1778
	Implicit Columns:ComplexClass	0.0928	0.4693	1.5593	**0.0333**	1.3173

◇ ***Implicit Columns* smell has a weak but statistically significant association with several traditional code smells such as *LongParameterList* and *ComplexClass*.**

We also conduct Chi-squared and Cramer's V tests to check whether the associations between

Table 7.4 Chi-square and Cramer's V value of smell pairs computed on the combined dataset for each smell pair in Table 7.3. We reject H_0 for all smell pairs in bold.

Smell Pairs	Chi-square P-value	Cramer's V
Implicit Columns:LongParameterList	**< 0.0001**	**0.0708**
Fear of the Unknown:LongMethod	**< 0.0001**	**0.048**
Fear of the Unknown:LongParameterList	**< 0.0001**	**0.03864**
Implicit Columns:ComplexClass	**< 0.0001**	**0.02925**
Implicit Columns:AntiSingleton	**< 0.0001**	**0.0282**
Fear of the Unknown:ComplexClass	**0.02217**	0.01335
Fear of the Unknown:AntiSingleton	**0.04868**	0.0115
Implicit Columns:LongMethod	0.0796	0.01

traditional code smells and SQL code smells (e.g., Table 7.3) are statistically significant or not. Table 7.4 shows the p-values from our Chi-squared tests. We assume this null hypothesis – H_0: traditional code smells and SQL code smells occur independently. However, given the p-values (< 0.05) in Table 7.4, we have strong evidence to reject the null hypothesis for each of the five emboldened smell pairs. That is, *Implicit Columns* has a significant association with several traditional code smells, such as LongParameterList and ComplexClass. It should be noted that each of these code smells is a result of bad programming practices by the developers.

Given the p-values ($>= 0.05$) in Table 7.4, we have weak evidence to reject the null hypothesis, i.e., such code smell pairs might not be associated.

◇ **The highest degree of association with a Cramers'V value of 0.07 was observed by the pair *Implicit Columns :LongParameterList*. This shows that SQL code smells and traditional code smells have a very weak association.**

We further investigated the statistically significant associations between traditional and SQL code smells within our subject systems, and determine the degree of associations using Cramer's V tests. Table 7.4 shows the results from these tests. We see that

Implicit Columns:LongParameterList pair has the highest degree of association with a *V* value of 0.07, which is still a weak association. The smell pairs for which we accept the null hypothesis have also small Cramer's V values, which is expected.

7.4 RQ 2.6: Survival analysis of SQL code smells

In this section, we present the findings of our survival analysis of SQL code smells in data-intensive systems.

7.4.1 Analysis approach

We conducted a survival analysis of SQL code smells, and traditional code smells, to investigate for how long SQL code smells persist in data-intensive systems. We utilize the **code smells dataset** for this analysis. We analyze the survival rates of traditional and SQL code smells during the evolution of our selected systems. For survival analysis, we use the Kaplan-Meier curve [36] (e.g., Fig. 7.3). The curve shows the survival probability $S(t)$ of a given code smell at a time t. We define the fixing of a code smells as our *event of interest*. That is, if a source code file contains a target smell in an earlier version snapshot and does not contain the same smell in the current snapshot, our event of interest occurs at the current snapshot. The occurrence of this event determines the survival probability of the corresponding code smell.

7.4.2 Findings

The results show that:

◇ **Significant number of SQL code smells persist for more than eight years in the subject systems.**

Figure 7.3 shows the Kaplan-Meier survival curve of *Implicit Columns* SQL code smells against 150 data-intensive subject systems. We see that the survival curve has a steeper slope at the beginning and becomes flat after 3000 days. This indicates that a large fraction of this smell was either fixed or censored without getting fixed in this time. However, the large number of censored data points indicate that a significant part of *Implicit Columns* smells persist without getting fixed.

Fig. 7.4 shows the Kaplan-Meier survival curve for another prevalent SQL code smell, namely *Fear of the Unknown*. It has a similar trend to that of *Implicit Columns*, but the events are more visible due to the small number of instances of this smell in the dataset.

◇ **SQL code smells have higher tendency to survive for longer period of time compared to traditional code smells.**

In order to achieve further insights, we compare the survival time of SQL code smells with that of traditional code smells. We run survival analysis on the two most prevalent traditional smells that are *LongMethod* and *LongParameterList*.

Fig. 7.5 shows our comparative analysis between traditional and SQL code smells. We see that SQL code smells have a gentler survival curve than that of traditional smells. That is, SQL code smells have longer lifespan. Thus, they persist within the subject systems for a

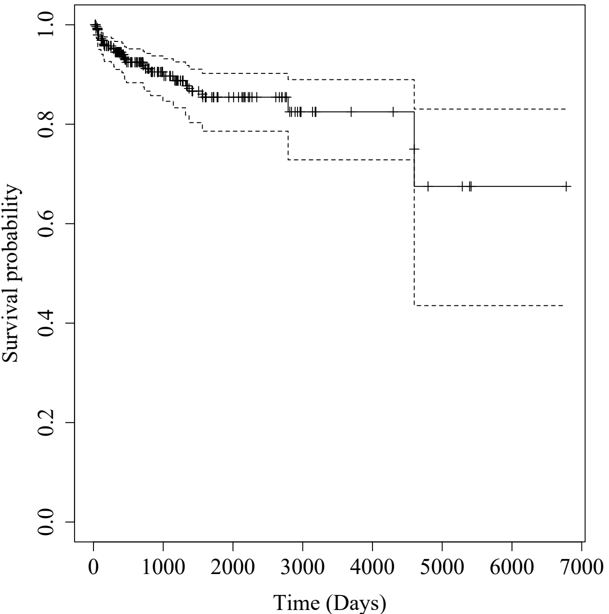


Figure 7.3 Kaplan-Meier survival curve for *Implicit Columns* SQL code smell. The X-axis is the time in days, and the vertical axis shows the survival probability value. The Censoring time and the Confidence interval are marked in the plot.

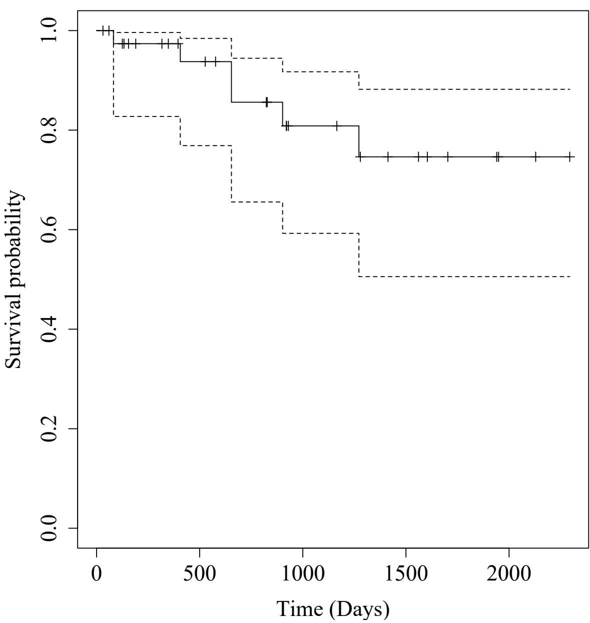


Figure 7.4 Kaplan-Meier survival curve for *Fear of the Unknown* SQL code smell. The X-axis is the time in days, and the vertical axis shows the survival probability value. The Censoring time and the Confidence interval are marked in the plot.

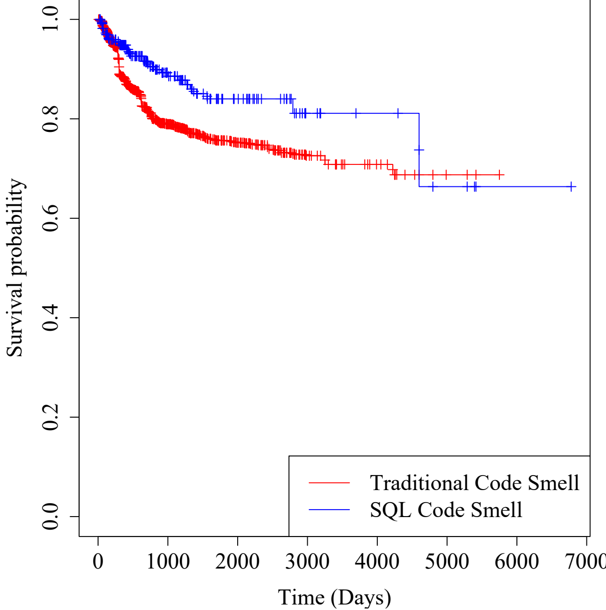


Figure 7.5 Kaplan-Meier survival curve for traditional code smells and SQL code smells. The Censoring time for censored files is marked in the plot.

longer time duration.

We performed *Logrank test* [143] to determine whether the difference between these two survival curves in Fig. 7.5 is statistically significant or not. We obtained a Chi-squared test p-value of 0.002, which provides a strong evidence that these survival curves are significantly different. In both curves, we see numerous censored data. By censored, we mean those files whose smells either persist in all tracked snapshots or they are deleted from the projects during the observation window, which is a rare case in our data.

◇ **A large fraction of the source files affected by SQL code smells (80.5%) persist throughout the whole snapshots of the subject systems without getting fixed.**

We also track the SQL code smells that occur across the versions of each single subject systems. Based on our investigation, we found that a large percentage of SQL code smells occurred in early versions of the subject systems. For instance, 89.5% of the source code files with *Implicit Columns* had their smells introduced in their first tracked snapshots. Similarly, 72.5% of the source code files with *Fear of the Unknown* had their smells introduced in their first tracked snapshot.

Our analysis shows that 80.5% of source code files with *Implicit Columns* smell contained this smell in all snapshots. Similarly, 65% of source code files with the *Fear of the Unknown* smell contained this smell in all snapshots. In contrast, 54% of source code files with *Long-*

ParameterList and 65% of source code files with the *LongMethod* contain those traditional code smells in all snapshots. This confirms the observation that many files with SQL code smells, and traditional code smells were censored before they are getting fixed. All these findings above suggest that SQL code smells get a little to no attention from the developers for refactoring.

7.5 Discussion

Our findings show that *Implicit Columns* and *Fear of the Unknown* smells are prevalent in the subject systems. This shows that not all SQL code smells are equally prevalent in data-intensive systems. Developers need to focus their attention on smells that are prevalent such as *Implicit Columns* which may lead to unexpected issues in the production environment. The prevalence of SQL code smells on Library projects is more concerning, as it may propagate to other application domains.

Our findings also show a small, but statistically significant co-occurrence between some SQL code smells and some traditional code smells. This result can be a starting point for investigation of the relation between SQL code smells, and traditional code smells by expanding the dataset and potentially detecting the occurrence of SQL code smells given some traditional code smells or vice versa.

The result of the survival analysis of SQL code smells, and traditional code smells, shows that little attention is given to SQL code smells by developers. Large portions of those smells are created in the first tracked snapshot of our subject systems and tend to persist for longer period of time. This implies that smells in general and SQL code smells in particular get a low priority in refactoring. The reasons for this could be developers' lack of awareness about those smells and their potential negative impact, or developers' engagement in higher priority tasks such as bug fixing tasks.

7.6 Threats to validity

In this section, we describe threats to research validity regarding characterization of SQL code smells (RQ 2.4, RQ 2.5, RQ 2.6).

7.6.1 Threats to construct validity

We relied on the accuracy of SQLInspect and DECOR detection tools. Both tools may miss some smells. While the results reflect the minimum case, the actual number of smells could

be higher. We also used git diff for file history tracking, which might fail to track some files if they are moved using *mv* command instead of *git move*. We did not include such files in our study. We also used a 70% similarity threshold for rename detection, which may lead to false rename assumption in some cases. However, the same threshold was used by the literature (e.g., by [11]).

7.6.2 Threats to conclusion validity

Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment [139]. This threat is associated with the choice of statistical tests. We used a non-parametric test in this study. While non-parametric tests are more general than parametric tests, they have lower statistical power. We did not claim a causal relationship between the variables, as we measured the association between them.

7.6.3 Threats to external validity

To make our findings generalize, we selected different types of projects in terms of application domain, size, and number of interactions with a database. We also covered projects that use different drivers and frameworks to interact with the database. We also tried to select representative projects with relevant data access. We only considered Java projects for analysis. However, our investigation approach generalizes to any programming language. It is desirable to study if our conclusions can be extended to different programming languages.

7.6.4 Threats to reliability validity

To minimize potential threats to reliability, we analyzed open-source projects available on GitHub and provide a replication package that contains our dataset [126].

7.7 Chapter summary

In this chapter, we investigated the prevalence of SQL code smells and their association with other traditional code smells. We collected 150 open-source Java projects, extracted both SQL and traditional code smells, and then jointly analyzed their prevalence and co-occurrence. We also performed a survival analysis to study how SQL code smells are handled throughout the lifetime of these projects.

Our results show that SQL code smells are prevalent in open-source data-intensive systems,

but at different levels. In particular, we found that the *Implicit Columns* SQL code smell is the most prevalent in our subject systems. With some exceptions, however, we did not see a significant difference in the prevalence of SQL code smells among application domains. Also, we found only a weak association between SQL code smells and traditional code smells.

Our survival analysis showed that a significant portion of SQL code smells was created in the first tracked snapshot of the studied systems and persisted in all snapshots without getting fixed.

Overall, our findings indicate that SQL code smells exist persistently in data-intensive systems, but independently of traditional code smells. As a consequence, developers have to be aware of SQL code smells, so that they can identify those smells and refactor them to avoid potential harm.

CHAPTER 8 IMPACT OF DATA-ACCESS TECHNICAL DEBTS ON SOFTWARE QUALITY

8.1 Chapter overview

In this chapter, we present the analysis approach and findings regarding the impacts of data-access technical debts on software quality (**Objective 3**). In particular, we investigate the co-occurrence of SQL code smells with bugs (RQ 3.1) as a first step to understand their impact on defects. To claim that SQL code smells have an impact on bugs, we first need to see if they have a strong co-occurrence with bugs because if they have a weak co-occurrence we can conclude that SQL code smells do not have a significant impact on defects. We also analyzed the perceived criticality of SQL code smells using a developer survey (RQ 3.2) and the perceived criticality of data-access performance anti-patterns (RQ 3.3).

RQ 3.1: **Do SQL code smells co-occur with bugs?**

> **Main finding:** SQL code smells (e.g., Implicit Columns, Fear of the Unknown) do not have statistically significant association with software bugs. On the contrary, traditional code smells (e.g., ComplexClass, SpaghettiCode) have a statistically significant association with the bugs, according to the results of the performed two statistical tests and RandomForest-based feature contribution analysis.

RQ 3.2: **What is the perceived criticality of SQL code smells?**

> **Main finding:** The Majority of the respondents agree that SQL code smells are critical by providing a criticality rating of 4 and 5. Majority of the respondents mention that they consider removing the SQL code smells during refactoring. The respondents also specify some NoSQL data-access performance anti-patterns and mention that they consider fixing them during refactoring.

RQ 3.3: **How do developers perceive the criticality of data-access performance anti-patterns?**

> **Main finding:** Among the 14 newly identified data-access performance anti-patterns *Improper handling of node failures*(55.5% respondents gave criticality rating > 3), *Using synchronous connection*(55% respondents gave criticality rating > 3), and *Inefficient driver API*(52.78% respondents gave criticality rating > 3) were the most critical data-access performance anti-patterns with more than 52% of the respondents confirming that the anti-patterns are critical and impact performance.

8.2 RQ 3.1: Co-occurrence of SQL code smells with bugs

In this section, We investigate the potential impact of SQL code smells on software quality and more specifically on bugs by first analyzing the occurrence of SQL code smells in bug-introducing changes. On one hand, if the analysis does not show a strong association between SQL code smells and bugs, it implies that SQL code smells do not have a significant impact on the bug-proneness of data-intensive systems. On the other hand, if the result indicates a strong association, it implies that SQL code smells could impact bug-proneness, and further analysis is required to determine if the SQL code smells or other co-founding factors are responsible for bug-proneness of data-intensive systems. We utilized Apriori algorithm and Cramer's V test of association to analyze the co-occurrence of SQL code smells with bugs. We present the details of our analysis and findings in the following sub-sections.

8.2.1 Analysis approach

To investigate the co-occurrence (or potential causation) between SQL code smells and software bugs, we utilized the **code smells dataset** (Section 4.2) and employed both Chi-squared test and Cramer's V test of association between SQL code smells and bugs. We determine the association between SQL code smells and software bugs by analyzing smelly code, bug-fixing code, and bug-inducing code. Our dataset contains a total of 21,973 file revisions, out of which 3,215 revisions were found in the bug-inducing commits. It should be noted that bug-inducing commits lead to software bugs, which are confirmed by the bug-fixing commits later. We thus separate the bug-inducing commits, and determine the pair-wise co-occurrence (association) between SQL code smells and bugs within these commits. We conduct Chi-squared test and Cramer's V test to check the significance and degree of the association.

We also developed a RandomForest model to investigate the contribution of each code smell on determining whether a file revision is bug-inducing (e.g., true class) or not (e.g., false class). Since the dataset was not balanced, we used SMOTE-based oversampling [144] and 10-fold cross-validation for our machine learning model. Finally, we collect the feature importance values from our trained model. These values indicate the importance of code smells (i.e., predictors) on determining whether a file version being bug-inducing or not.

8.2.2 Findings

To determine the association between SQL code smells and software bugs, we assume this null hypothesis – H_0: The presence of SQL code smells in a file version and the file version being bug-inducing are independent phenomena. We test this hypothesis with Chi-squared test using $\alpha = 0.05$. As shown in Table 8.1, we notice that both *Implicit Columns* and *Fear of the Unknown* are two SQL code smells that occur independently of the bug-inducing commits. They have p-values greater than our significance threshold of 0.05.

◇ **SQL code smells (e.g., Implicit Columns, Fear of the Unknown) do not have statistically significant association with software bugs. On the other hand, traditional code smells (e.g., ComplexClass, SpaghettiCode) have a statistically significant association with the bugs.**

Although the *Implicit Columns* smell is known to cause performance issues and software bugs [27, 32], our empirical analysis did not show a strong correlation with bugs. On the contrary, the traditional code smells such as *SpaghettiCode, ComplexClass* and *AntiSingleton* have significant p-values < 0.05, which indicates that they have a stronger association with bugs. The traditional code smell namely *ComplexClass* has the lowest and the most significant p-value, which indicates its significant association with the bugs. *ComplexClass* was also reported to be associated with software bugs by the earlier studies [18, 107].

We also determine the degree of association between any code smells and software bugs using Cramer's V test. As shown in Table 8.1, we see that the traditional code smells (e.g., *LongMethod*, *LongParameterList*) have a relatively higher V-values than that of SQL code smells. That is, SQL code smells might be less associated with the bugs than the traditional code smells.

◇ ***ComplexClass* traditional code smell has the highest feature importance of 46% while SQL code smells have combined feature importance less than 13%.**

The last column of Table 8.1 shows how each of the code smells could turn its containing file to be bug-inducing. We see that *ComplexClass* has the highest importance of 46%. Despite the

low Cramer's V values, *LongMethod* and *LongParameterList* are pretty important (≈10%) in our trained model. On the other hand, SQL code smells (e.g., Fear of the Unknown, Implicit Columns) might be less important according to our model, which clearly indicates their low association with the software bugs.

Table 8.1 Result of statistical tests and random forest model of association between smells and buggy files.

Smell	**Chi-square P-value**	**Cramer's V value**	**Feature Contribution (%)**
Implicit Columns	0.2377	0.0069	5.095
Fear of the Unknown	0.1671	0.008	7.695
LongMethod	0.1162	0.0003	9.86
LongParameterList	**0.0034**	0.0034	9.1
AntiSingleton	**< 0.001**	0.0001	7.69
SpaghettiCode	**< 0.001**	0.0227	5.87
ComplexClass	**< 0.001**	0.0846	**46.65**

8.3 RQ 3.2: Perceived criticality of SQL code smells

In this section, we present the analysis approach and findings from the survey in refactoring practices (Section 4.5.10) regarding the criticality of SQL code smells as perceived by software developers. Analyzing the criticality of SQL code smells helps to prioritize SQL code smells for refactoring, as well as to understand the perceived impacts of SQL code smells based on the developers' point of view and experience.

8.3.1 Analysis approach

As part of the survey in refactoring practices, we asked practitioners about the criticality of prevalent SQL code smells, *Implicit Columns* and *Fear of the Unknown*, in data-intensive systems and provided an opportunity for participants to specify critical NoSQL anti-patterns. The survey questions are outlined in Section 4.5.10 (see section five of the survey).

8.3.2 Findings

The results show that:

⋄ 65% of the respondents agreed that they consider optimizing SQL queries during data-access refactoring.

We obtained 20 complete responses after running the survey on refactoring practices for one month. We asked if the survey participants consider improving SQL queries during data-

access refactoring, 10 participants agree and 3 participants strongly agree, totaling 65%. One participant mentioned that optimized query is more important than optimized code in the context of data-intensive systems. On the other hand, one respondent disagrees, justifying that SQL queries must always be optimized whether refactorings are applied or not. The remaining 6 participants Neither agree nor disagree with justifications mentioning that optimizing SQL queries should be a separate task and not associated with refactoring (1 respondent) and two respondents mention that it depends on the underlying data-access framework.

◇ 90% of the respondents confirm the criticality (rating > 3) of both *Implicit Columns and Fear of the unknown* SQL code smells. Furthermore, the majority of the respondents confirm that they consider refactoring both smells during data-access refactoring operations.

When considering the criticality of *Implicit Columns* SQL code smell, 16 participants (80%) rate the criticality more than three out of five, indicating that most of the participants agree that *Implicit Columns* SQL code smells are relevant technical debts that impact the quality of data-intensive systems. In the follow-up justification question, respondents outlined that this smell could impact performance in cases where there is huge data being involved. This smell is also associated with security issues, for man in the middle attack, having this smell exposes more data to the attacker. Furthermore, this smell creates unnecessary coupling between the database and the data-access logic. One respondent who gave a criticality of 4 mentioned that *"this smell could be avoided by using an Object-relational mapper (ORM) and mitigated by using caching, load balancing, and scalable cloud solutions"*. However, we believe that the smell still affects data-access codes using ORMs especially if they only use the subset of the fetched data. Unfortunately, the respondents who gave a lower criticality value (2) did not provide their justification.

We asked the participants if they consider fixing SQL code smells during data-access refactoring, and 14 respondents agree that they consider refactoring this smell, out of which five strongly agree. On the other hand, two respondents disagree, with one respondent selecting 'I don't know'. Three respondents neither agreed nor disagreed, and one respondent argue to justify his response that sometimes it is hard to list all columns for a table with many columns. Another respondent also mentioned that they will not refactor the code just to fix this smell, but that they will do it from time to time.

We asked similar questions for the *Fear of the unknown* smell, as it is the second most prevalent SQL code smell in data-intensive systems. Similar to *Implicit Columns* smell, the criticality of the *Fear of the unknown* smell was rated more than three by 16 respondents

(80%) out of which ten respondents gave the highest criticality (five) and six respondents gave four. Only one respondent gave a criticality of one and another respondent gave a criticality of two. However, both respondents did not justify their rating. One of the respondents that gave a criticality rating of three mentioned that *"I don't consider it a big issue, but I had to change because of an unexpected result"*.

18 respondents confirm that they consider refactoring *Fear of the unknown* SQL code smell, with eight of them strongly agree. Only two respondents mentioned that they neither agree nor disagree. Respondents mentioned that this smell is critical and should be fixed as a critical issue or a bug.

◇ **We obtained several NoSQL anti-patterns from the suggestion of survey participants related to indexing, maintaining consistency and synchronization.**

We also asked if the respondents encountered data-access smells in NoSQL database-backed applications, out of which only eight participants confirmed that they encountered NoSQL data-access anti-patterns. The mentioned NoSQL anti-patterns include *unnecessary indexes, missing indexes, handling consistency using locks in BASE transactions, not using optimistic locking when needed, and using light-weight transactions in real-time read/write loads.*

Seven out of the eight respondents agreed that they consider refactoring the aforementioned anti-patterns out of which two of them strongly agree. One respondent answered 'I don't know' to this question. Another respondent also mentioned that *"the anti-patterns usually ruin performance and cause a technical debt, and they need to be addressed during data-access refactoring"*.

8.4 RQ 3.3: Perceived criticality of data-access performance anti-patterns

In this section, we discuss the result of the survey on data-access performance anti-patterns (described in Sub-section 4.4.8) to understand the perceived criticality of data-access performance anti-patterns.

8.4.1 Analysis approach

We analyzed the responses of the **survey on data-access performance anti-patterns**. The recruitment procedure and the content of the survey were described in Sub-section 4.4.8. We obtained **40** complete responses after running the survey for one month. We extracted all survey responses from Google forms and downloaded them as a CSV file. We will report the quantitative and qualitative analysis of the survey responses. We present the summary of

critical ratings for each data-access performance anti-pattern using stacked bar charts. **We mapped the criticality score of 1 to Not Critical, 2 to Lowly critical, 3 to Neutral, 4 to Critical, and 5 to Highly critical.** In the following discussion, we consider criticality ratings of critical and highly critical as **positive** critical ratings and criticality ratings not critical and lowly critical as **negative** criticality ratings.

8.4.2 Findings

We first report the demographics of participants and discuss the summary of the survey response analysis, categorizing them into seven high-level data-access performance anti-patterns described in **RQ 3.3**.

Survey demographics

◇ **Majority of the survey respondents are software developers with significant experience in software development and backend development.**

60% of the survey respondents described their organizational role as *software developers,* while two participants mention their role as product owner and manager. Other participants mentioned roles such as developer and researcher, testing manager, system analyst, data analyst, and student.

All survey participants mentioned that they have at least one year of software development experience. 29 participants (72.5%) have between one and five years of experience, while 5 participants (12.5%) have between five and ten years of software development experience. Six participants (15%) have more than ten years of software development experience.

Regarding experience in backend development, 31 participants(77.5%) mentioned that they have backend development experience between one and five years. Six participants (15%) mentioned that they have more than ten years of backend development experience. The remaining three participants mentioned that they have between 5 and 10 years of backend development experience.

Participants mention several database access (persistence) frameworks they use for development. Some mentioned frameworks are *Hibernate, Spring, Django, ado.net, Entity framework, mongoose*. Some respondents also mention that they work with databases such as MongoDB, Oracle, Redis, and MySQL. One participant mentioned that he/she preferred to use custom helpers for ease of use and control.

Data fetching and update anti-patterns

◇ ***Sequential lookup of multiple keys* performance anti-pattern was rated as critical/highly critical by 50% of the respondents.**

We only identified one new anti-pattern, *Sequential lookup of multiple keys*, under this category. Figure 8.1 shows the summary of the criticality rating for this anti-pattern. 38 respondents provided a criticality rating between 2 and 5 for this anti-pattern while 2 respondents did not rate (they chose the "I don't know" option). On the other hand, it was rated as lowly critical by 21.05% of the respondents. One of the respondents that rate this anti-pattern as Lowly critical mentioned that *"Sure, it's a performance issue, and in rare cases might make something infeasible, but it doesn't fundamentally affect what you can do in most cases."* which shows that they may be considered critical performance issues in some contexts but not always. When we look at the justifications provided by some of the respondents that rated this anti-pattern as highly critical mentioned that this anti-pattern slows read performance and prevents scalability. 21.05% of the respondents provided a neutral rating of 3. Some neutral raters justified their response by mentioning that this anti-pattern won't have much impact for smaller size tables, and the search is linear in the number of rows.

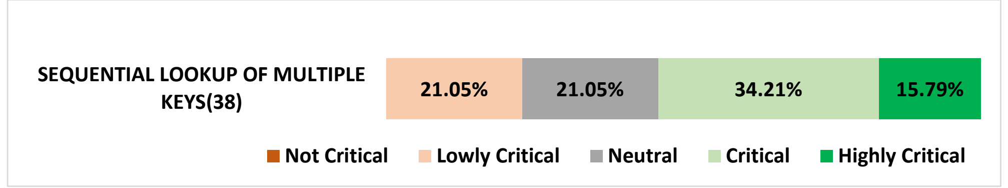


Figure 8.1 Criticality rating of *Sequential lookup of multiple keys* performance anti-pattern. The number of respondents (38) is indicated next to the anti-pattern name.

The respondents suggested *using indexes, caching the table, using bulk queries, and splitting a large table using the keywords* as an index as possible fixes for this anti-pattern.

The respondents did not suggest a new performance anti-pattern under this category, that is not covered in this study or previous literature.

Database connection anti-patterns

Figure 8.2 shows the criticality rating of the three newly identified database connection anti-patterns. Among the anti-patterns, *Using synchronous connection* was rated the most

critical, with 55% respondents giving a positive criticality rating (*i.e.*, critical/highly critical).

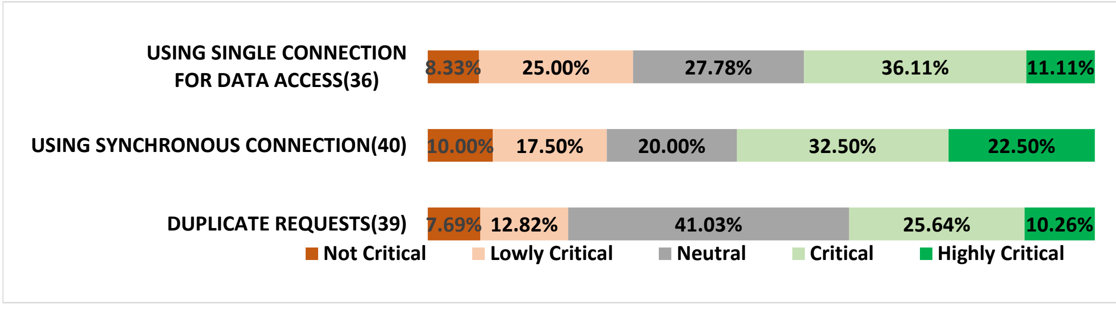


Figure 8.2 Criticality rating of performance anti-patterns under database connection category.

◇ ***Using synchronous connection* was the most critical database connection anti-pattern. 55% of the respondents gave a positive criticality rating.**

We analyzed the provided justifications for the criticality rating of *Using synchronous connection* anti-pattern. One respondent who rated this anti-pattern as lowly critical mentioned that *"most backend code is already multi-threaded or multi-process, so it's less important that the db request be separately threaded."*. On the other hand, raters who provided positive criticality ratings for this anti-pattern mentioned that this anti-pattern slows down the system, wastes resources, and stalls the back-end during the time of database lookups.

Some of the respondents who rated this anti-pattern as neutral mentioned that the criticality of this anti-pattern depends on the problem context. One respondent mentioned that *"This depends a lot on the problem context. In UI, it can be a problem. If it's a batch task running from time to time, and it does not need another thread, it is not a problem."*. Another respondent also mentioned that *"This depends on the back-end architecture. If threads are the unit of concurrency, there's not a significant problem. If using an event-driven async I/O architecture, this is a major problem."*

When we asked about possible fixes for this anti-pattern, many respondents suggested using the asynchronous connection as a fix for this anti-pattern especially if asynchronous data-access API is available.

Using single connection for data access performance anti-pattern gets the second-highest critical rating. 47.22% of the respondents provided a positive criticality rating, with 11% of the respondents rating this anti-pattern as highly critical.

Some justifications provided by the respondents who gave a positive critical rating for this

anti-pattern include hindering the scalability and performance. On the other hand, the justifications provided by raters with negative critical ratings (*i.e.,* Not critical/lowly critical) mention that this anti-pattern is not critical when the performance of the individual queries is fast and the latency introduced due to this anti-pattern is not a major contributor to the overall performance of the system. Respondents suggested using parallel requests, adding more connection pools, and using asynchronous APIs, as possible fixes for this anti-pattern.

For the *Duplicate requests* performance anti-pattern, 41.3% of respondents rated this anti-pattern as Neutral. 35.9% of the respondents rated this anti-pattern with a positive rating, while 20.51% rated this anti-pattern with a negative criticality rating. Some mentioned justifications for raters who gave positive rating mention that the anti-pattern causes unnecessary consumption of resources, slows down the application when the duplicated request takes a lot of time, is not necessary, and may cause inconsistency in the application. On the other side, one respondent who gave a criticality rating of lowly critical mentioned that this anti-pattern won't have much impact on requests with a small database processing time. Regarding the possible fixes for this anti-pattern, proper handling of UI events, request validation, and caching were suggested by the respondents.

We asked the respondents to suggest new performance anti-patterns under this category, but none of them suggested any new performance anti-pattern.

Database driver or API access anti-patterns

Figure 8.3 shows the criticality rating of the performance anti-patterns under database driver or API access anti-patterns. Among the performance, anti-patterns, *Inefficient driver API* gets the highest positive criticality rating. This anti-pattern obtained a positive criticality rating from 52.78% of the survey respondents. On the other hand, *Using the wrong API function* performance anti-pattern received the lowest amounts of positive criticality rating (*i.e.,* 30.55%).

◇ ***Inefficient driver API*** **get the highest positive criticality rating among database driver or API access anti-patterns.**

Some of the respondents who gave a positive criticality ratings for *Inefficient driver API* anti-pattern provided justifications that include difficulty to recognize the source of this anti-pattern, hard to work around, and causing performance issues. Furthermore, one respondent mentioned that *"Most apps will rely on the API to access data on the database. Therefore if the API has known issues those will directly affect the functionality and effectiveness of the app."* highlighting that the impact of this anti-pattern spans all applications that rely on

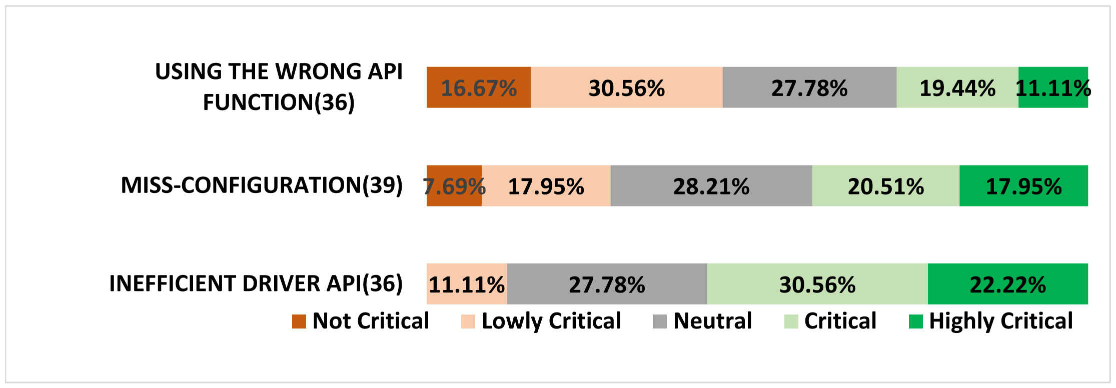


Figure 8.3 Criticality rating of performance anti-patterns under database driver or API access anti-pattern category.

the API. The justifications from the respondents who gave a negative criticality rating are not clear, as most of them still admitted that this anti-pattern causes performance problems. *Changing or updating the driver or API and trying to get support from the vendors of the API were suggested by most respondents as a fixing strategy for this anti-pattern.*

The *Miss-configuration* anti-pattern obtained the second-highest amount of positive critical rating by 38.46% respondents. One respondent who gave a positive criticality rating for this anti-pattern justified by saying, *"vendors should provide solid defaults, but unfortunately often don't. again, can take significant effort to remedy."* highlighting that this anti-pattern is difficult to remedy. On the other hand, respondents that gave a negative criticality rating mentioned that this anti-pattern is easily fixed with proper guidance to the developer. The majority of the respondents mentioned that correcting the configuration issues by following guidelines from the vendors is a possible fix for this anti-pattern.

Using the wrong API function anti-pattern got the lowest amount of positive rating (*i.e.,* rated positive only by 30.55% of the respondents). 47.23% of the respondents gave a negative criticality rating for this anti-pattern and justified their rating by saying that this anti-pattern can be easily fixed and that the severity of this anti-pattern is highly dependent on the functionality of the API. Conversely, some respondents with positive criticality ratings mentioned that this anti-pattern is more common than expected and could lead to maintenance problems besides performance problems.

The survey respondents did not provide us with any new performance anti-pattern under the database driver or API access anti-patterns category.

Caching anti-patterns

Figure Figure 8.4 shows the summary of the criticality rating of the data-access performance anti-patterns under caching category. *Cache invalidation instead of updating* anti-pattern and *Inefficient caching* anti-pattern get the highest positive criticality rating among caching anti-patterns. 42.86% of respondents gave a positive criticality rating for, *Cache invalidation instead of updating* while 42.85% of the respondents gave a positive criticality rating for *Inefficient caching*. On the other hand, *Not caching the query* anti-pattern obtained a positive criticality rating only from 27.02% of the respondents.

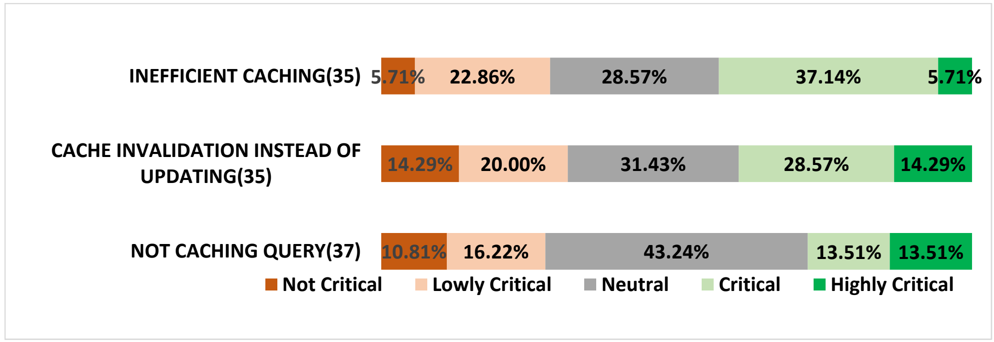


Figure 8.4 Criticality rating of performance anti-patterns under caching category.

◇ ***Cache invalidation instead of updating*** **anti-pattern obtained the highest positive criticality rating among caching anti-patterns. 42.86% of the respondents gave a positive criticality rating for this anti-pattern.** ***Inefficient caching*** **anti-pattern obtained the next highest rating(42.85% of the respondents gave positive criticality rate)**.

Respondents who gave a positive rating for *Cache invalidation instead of updating* anti-pattern mentioned that this anti-pattern leads to performance problems and is often implemented incorrectly. Conversely, respondents with negative ratings mentioned that updating the cache could lead to even worse problems by converting the cache into a database, so they consider this anti-pattern as benign. Participants suggested updating the cache for large cache size or reducing the cache size as possible fixes for this anti-pattern.

Respondents who gave a positive criticality rating for *Inefficient caching* anti-pattern mentioned that this anti-pattern slows down read data-access operations and since caching is an important component of data-access, one needs an efficient implementation. The respondents also mentioned that this anti-pattern negatively affects application users' experience. On the other hand, respondents who gave a negative criticality rating for this anti-pattern admitted

that it degrades performance but has a minimal overall effect.

Not caching the query anti-pattern obtained a negative criticality rating from 27.03% of the respondents while obtaining a positive criticality rating from 27.02% of the respondents. The remaining 43.24% gave a neutral rating for this anti-pattern. One respondent who gave a positive rating mentioned that this anti-pattern is critical, especially when the same query is sent multiple times. Conversely, respondents who gave negative ratings mentioned that building the query is not a performance bottleneck and caching the query could introduce unwanted security vulnerabilities. Caching complex queries is the fix suggested by the survey respondents.

The respondents did not provide us with a new performance anti-pattern related to caching when they are asked to mention new performance anti-patterns under this category.

Indexing anti-patterns

We only have *Non-optimal indexing logic* data-access performance anti-pattern under the indexing category. Figure 8.5 shows the criticality rating of this anti-pattern.

◇ 47.06% of the respondents gave a positive criticality rating for this anti-pattern while only 26.47% negative criticality rating.

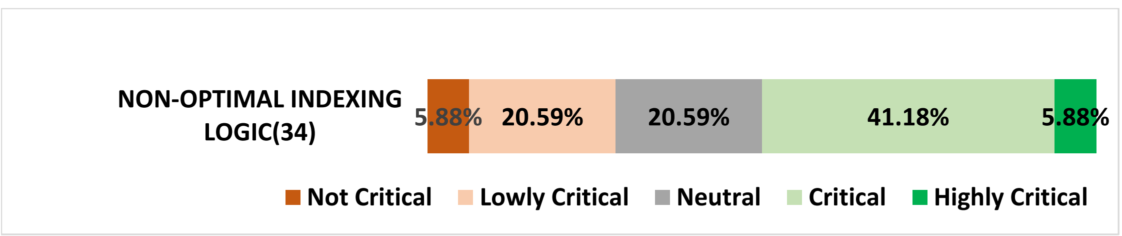


Figure 8.5 Criticality rating of *Non-optimal indexing logic* data-access performance anti-pattern.

Respondents with positive criticality ratings mentioned that as indexing improves data-access performance, having a non-optimal indexing logic will negatively impact performance, especially when there are large databases with huge tables. One respondent also mentioned that this anti-pattern is *"very common, and can evolve as the application's data patterns morph over time."*. One respondent who gave a negative rating mentioned that the impact of this anti-pattern *"Depends on how "non-optimal" Indexing is done, may affect everything, may affect little"*. *continuously monitoring the performance of indexes and optimizing inefficient implementations* were suggested by respondents as a fix for this anti-pattern. The

respondents did not provide us with any new data-access performance anti-pattern related to indexing.

Data node configuration and management anti-patterns

We have two new performance anti-patterns under this category: *Improper handling of node failures*, and *Load imbalance in multiple nodes* performance anti-patterns under this category. Figure 8.6 shows the criticality ratings of both anti-patterns. *Improper handling of node failures*, anti-pattern was given a positive criticality rating by 55.55% of the respondents, while only 19.44% of the respondents gave it a negative criticality rating.

◇ **55.5% of the respondents gave a positive criticality rating for *Improper handling of node failures*, performance anti-pattern. 36.11% of the respondents rated this anti-pattern as highly critical.**

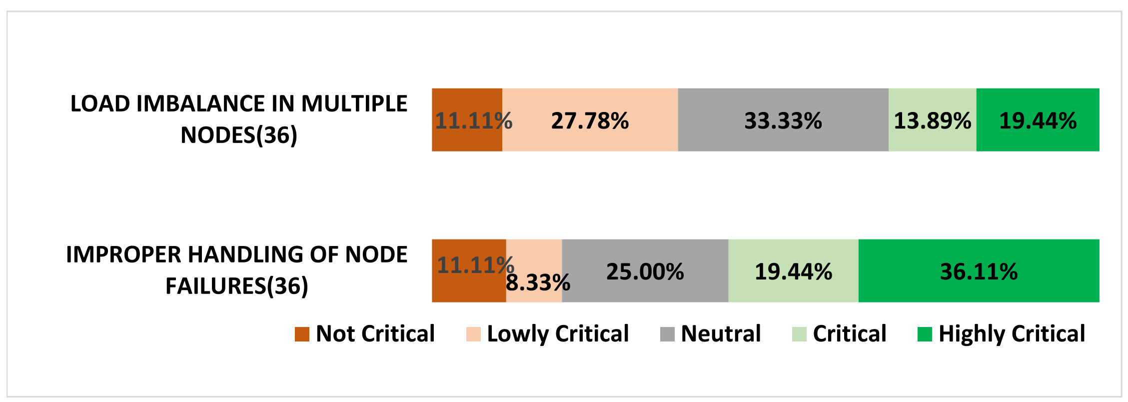


Figure 8.6 Criticality rating of data node configuration and management anti-patterns

Respondents who gave positive ratings mentioned that the *Improper handling of node failures*, anti-pattern is a common problem that negatively impacts data-access performance, and it may even lead to data loss in the worst case. Conversely, one respondent who gave a negative criticality rating mentioned that *"it's a big issue, but algorithms for managing quorum and health are fairly robust most of the time. any effort spent here is almost always better spent somewhere else where it'll have a better return."*; highlighting that this anti-pattern is problematic, but that fixing it is not a high priority. *Implementing proper node failure handling mechanism* is suggested as a fix for this anti-pattern by most respondents.

The *Load imbalance in multiple nodes* performance anti-pattern obtained a negative criticality rating from 38.89% of the respondents. One respondent who gave a negative rating mentioned that *"this is less an anti-pattern and more a trade-off of architectural decisions*

in the database engine.”. On the contrary, respondents who gave positive ratings mentioned that it wastes resources and causes performance issues, and may even lead to a system crash in the worst case. *Using or implementing load balancer, dynamic load balancing, validating node configuration, choosing appropriate shading keys* were suggested to fix this data-access performance anti-pattern. The respondents did not suggest any new data-access performance anti-pattern under this category.

Query anti-patterns

Figure 8.7 shows the criticality rating of the *Inefficient query translation* anti-pattern which is the only new anti-pattern we have among query anti-patterns. This anti-pattern obtained a positive criticality rating from 39.39% of the survey respondents, while 27.27% of the respondents gave it a negative criticality rating.

◇ ***Inefficient query translation*** **anti-pattern obtained a positive criticality rating by 39.39% of the respondents.**

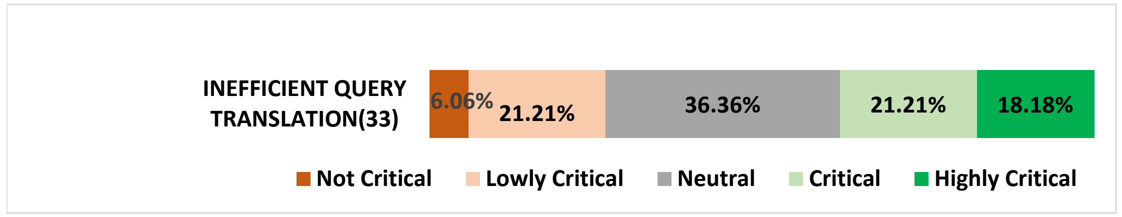


Figure 8.7 Criticality rating of *Inefficient query translation* data-access performance anti-pattern

Respondents who gave a positive criticality rating mentioned that inefficient translation reduces the performance of query translation, plus it is expensive to fix this anti-pattern. One respondent mentioned that *“Ineffective query management is detrimental to translation quality and also time-consuming and expensive to rectify post-delivery.”*. Another respondent mentioned that *“I’ve typically noticed this in a slightly reversed form where the query optimizer can’t find an optimization that it should be able to find (so more like a failure to translate).”*. Hence, an inefficient query optimizer could cause this anti-pattern. One respondent who gave a negative criticality rating mentioned that the criticality of this anti-pattern depends on the size of data processed by the query. *Improving the efficiency of query translation or manipulating, if possible, the query optimizer that caused the inefficient translation* were suggested as possible fixes.

8.5 Discussion

We did not see a strong co-occurrence between SQL code smells and bugs. Our result shows that some traditional code smells have a higher association with bugs compared to SQL code smells. This implies that, future investigation should focus on the impact of SQL code smells on maintainability and performance instead of their link with bugs.

Our findings show that data access refactorings do not generally touch SQL queries and SQL code smells Since improving data access performance is one of the desired improvements in the data-intensive system and query optimization is one mechanism to improve performance, it is normally expected that code elements that are associated with queries should get more refactoring attention. Indeed, the majority of the surveyed developers confirmed that they will consider optimizing SQL queries during data-access refactoring. However, our analysis of the subject systems shows that code elements associated with queries do not get refactoring attention. In chapter 7, we showed that SQL code smells are prevalent in data-intensive systems, and they tend to persist across the evolution of the systems without getting fixed. Furthermore, the majority of the survey participants consider both *Implicit Columns and Fear of the unknown SQL code smells* as relevant and critical in data-intensive systems and confirmed that they consider refactoring to fix those smells. However, in our subject systems, few of the data access refactorings involved methods containing SQL queries and none of them modified the queries. Consequently, we did not find any instance of SQL code smell removal. Due to the lack of NoSQL data-access smell detection tool, we were not able to do a similar relation analysis for NoSQL subject systems. Specification and detection of NoSQL anti-patterns is still not well addressed in the case of NoSQL-based systems. To contribute to the specification of NoSQL anti-patterns we asked the developers to specify some instances that they encountered. The provided NoSQL anti-patterns include inefficient indexing, missing indexes, inefficient handling of consistency in BASE transaction and synchronization issues. Those provided NoSQL data-access anti-patterns are associated with the performance and robustness of data-access code.

We conducted a developer survey to evaluate the criticality of the data-access performance anti-patterns discussed in **RQ 1.2** (Chapter 5). The result of the survey shows that among 14 newly identified performance anti-patterns *Improper handling of node failures*(55.5% positive rating), *Using synchronous connection*(55% positive rating), and *Inefficient driver API*(52.78% positive rating) were the most critical data-access performance anti-patterns with more than 52% of the respondents confirming that the anti-patterns are critical. Hence, those anti-patterns should be prioritized by developers, given their strong impact on data-access performance. Future works to develop data-access performance anti-pattern detection

approaches should prioritize those anti-patterns. *Improper handling of node failures* and *Using synchronous connection* can be easily detected by performing a static analysis on the corresponding code base. But detecting *Inefficient driver API* anti-pattern is relatively difficult as it may require dynamic analysis and profiling the driver API calls. Another anti-pattern with significant positive criticality rating is *Non-optimal indexing logic*(47.06% positive rating). This is not surprising as indexing is important functionality to accelerate read performance and inefficiency in the indexing logic negatively affects data-access performance. Automatically detecting this anti-pattern requires dynamic profiling of the indexing logic. On the other hand, some anti-patterns such as *Load imbalance in multiple nodes*and *Not caching the query*obtained more negative criticality rating. Unfortunately, we did not obtain a strong justification from the respondents on why such anti-patterns are less critical. Hence, it needs further investigation.

The survey respondents confirmed our proposed fixing strategy for most of the anti-patterns, but also suggested additional fixing strategies for some performance anti-patterns. Hence, practitioners can utilize the fixing strategies we suggested in RQ 1.2 and what developers suggested in RQ 8.4 to address the data-access performance anti-patterns. Furthermore, we did not obtain a new performance anti-pattern under all the seven high level categories. While we are not claiming that this are the only data-access performance anti-patterns, most of the root causes of data-access performance issues can be mapped to the performance anti-patterns obtained in this work as well as prior works that specified data-access performance anti-patterns.

8.6 Threats to validity

In this section, we discuss threats to research validity regarding our analysis and findings on the impact of data-access technical debts (RQ 3.1, RQ 3.2 and RQ 3.3).

8.6.1 Threats to construct validity

To link bugs with file versions, as part of the co-occurrence of SQL code smells with bugs, we relied on the SZZ algorithm, which might not be free from limitations. First, the heuristics of finding bug-fix commits using keywords may introduce false positives, which might incorrectly identify buggy lines [145]. We also manually checked 50 randomly-sampled, bug-inducing commits detected by the SZZ algorithm and found only three (6%) false-positives. Thus, the threat posed by SZZ might not be significant.

8.6.2 Threats to internal validity

We relied on LinkedIn to recruit some survey participants. There could be the case that those participants are not involved with data-access code development and refactoring. To mitigate this threat, we carefully evaluated the LinkedIn profile of the candidate participants to make sure they have experience working with data-access code and only shared the survey link to candidates that passed the evaluation. Another internal threat to validity is that studies based on questionnaires could be subjective. To mitigate this threat, we provided a five point Likert scale and also included "I don't know" option to avoid forcing the respondents to pick one answer for multiple choice questions.

8.6.3 Threats to conclusion validity

Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment [139]. This threat is associated with the choice of statistical tests. We used a non-parametric test in this study. While non-parametric tests are more general than parametric tests, they have lower statistical power. However, we did not claim a causal relationship between the variables as we measured the association between them.

8.6.4 Threats to external validity

Threats to external validity concern the ability to generalize experiment results outside the experiment setting [139]. We have a few completed responses (20) of the survey on refactoring practices. However, the participants come from different industries including open-source projects, which improves the generalization of the findings. Another threat to external validity is related to the extent to which the produced catalog of data-access performance anti-pattern exhaustively covers all performance issues. To mitigate this threat, we did not fix the number of samples to manually analyze. We rather sorted the performance issues dataset in decreasing order of relevance and started labeling until we achieve a labelling saturation. While we achieved a labelling saturation after analyzing 250 issues, we continued our analysis and labeled 150 more issues, and we did not generate a new performance anti-pattern. We also asked survey respondents for new performance anti-pattern that is not already covered, and we did not get any new performance anti-pattern.

8.6.5 Threats to reliability validity

To minimize potential threats to reliability, we analyzed open-source projects available on GitHub and provide a replication package that contains our dataset and analysis scripts [126, 141, 146].

8.7 Chapter summary

In this chapter, we discussed the impact of data-access technical debts on software quality. In particular, we investigated the co-occurrence of SQL code smells with software defects (bugs) and the perceived criticality of SQL code smells and data-access performance anti-patterns. Our results show that SQL code smells do not co-occur with bugs, but may have a significant performance impact as they are perceived as critical by software developers. Some data-access performance anti-patterns were perceived as critical by the majority of the software developers that participated in the survey.

CHAPTER 9 QUANTITATIVE ANALYSIS OF DATA-ACCESS REFACTORINGS

9.1 Chapter overview

In this chapter, we present the quantitative analysis and findings regarding the prevalence of refactorings (RQ 4.1), the evolution of refactorings (RQ 4.2), refactoring activities and SQL code smells (RQ 4.3), co-occurrence of data-access refactorings (RQ 4.4) and profile of developers involved with data-access refactoring (RQ 4.5). We utilized the **refactoring dataset** as a data source for the quantitative analysis. As part of achieving research objective 4, we answer the following research questions.

RQ 4.1: **How prevalent are refactorings in data access classes?**

> **Main finding:** Refactorings are slightly less prevalent in data-access classes compared to regular classes. Furthermore, ***Change Parameter Type*** is the most prevalent refactoring type in data-access classes. Most data-access refactoring activities are concentrated on few core classes. The distribution of refactoring density is similar between data-access and regular classes, indicating that developer's give similar refactoring attention to data-access and regular classes.

RQ 4.2: **How do refactoring activities change during the lifetime of the subject systems?**

> **Main finding:** The median *Relative Commit Time* for data-access refactorings (45.97%) is higher than that of the regular refactorings (9.89%), indicating that developers have a tendency to refactor data-access classes at later stages of the evolution of systems compared to regular classes. Among the most prevalent data-access refactoring types, *Move Method & Add Method Annotation* it has a tendency to be applied at the later stages of the software evolution. The prevalence of data-access refactorings is roughly similar, regardless of the *Relative Commit Time* from official releases. However, most of the regular refactorings happen far before official releases. Data-access refactorings in SQL-based subject systems have higher *Relative Commit Time* and *Distance Before Release* compared to data-access refactorings in NoSQL-based subject systems.

RQ 4.3: **Do data access refactoring activities touch SQL queries and SQL code smells?**

> **Main finding:** We observed that only a small fraction of data-access refactoring instances in SQL-based subject systems touched statements and methods containing SQL queries and SQL smells. Using line level matching, only 0.45% of refactoring instances were involved with SQL queries. Using a method level matching, we found 29.68% refactoring instances were performed on the methods containing SQL queries and 1.35% refactoring instances were performed on the methods containing a query with SQL code smell. However, the applied refactorings did not modify the SQL queries, and consequently the SQL code smells are never fixed once they are introduced.

RQ 4.4: **Do different types of refactorings co-occur in data access classes?**

> **Main finding:** In most cases, similar refactoring pairs co-occur in both data-access and regular refactorings. Refactorings that change the type of variable or parameter has a higher co-occurrence with refactorings that rename identifiers. Not more than two pairs of refactorings participate in composite refactorings for both data-access and regular refactorings.

RQ 4.5: **What is the profile of developers performing data-access refactorings?**

> **Main finding:** Among the considered developer profile metrics, *Refactoring contribution* metric was the most separating metric between developers involved in data-access refactoring and regular refactoring. Some refactoring types in data-access and regular classes are often associated with developers that have high *Refactoring contribution & Local contribution*. Developers involved in data-access refactoring are more experienced with the subject systems compared to developers involved only in regular refactoring. Not all developers have proportional refactoring responsibility. Large portion of both data-access and regular refactorings are dominated by single developers and class owners.

9.2 RQ 4.1: Prevalence of refactorings in data-access classes

In this section, we discuss our analysis of the prevalence of refactorings in data-access classes. We compared the prevalence of refactorings between data-access classes and non-data-access

classes. Since data-access classes are critical components of data-intensive systems linking the storage systems with the data processing systems, studying the prevalence of refactoring activities in such classes is critical. On one hand, if the refactorings are not prevalent, it could show that they are not getting enough attention. On the other hand, if they are prevalent, we further investigate the characteristics of such refactorings.

9.2.1 Analysis approach

We use two metrics to measure the prevalence of refactorings. These metrics are:

- *Number of refactoring instances:* absolute number of refactoring instances in data-access or regular classes.
- *Refactoring density:* defined in Equation 9.1. Where *no of refactorings* is the number of applied refactorings and *average code size* is the average number of software lines of code of the target class over all revisions. We applied Square root to reduce the scale of the code size to match the number of refactorings. The motivation behind using refactoring density is the significant variation in code size between data-access classes and regular classes. Figure 9.1 shows the distribution of average code size for data-access and regular classes. Given the subject systems are data-intensive, we expect to have higher median average code size for data-access classes (268) compared to regular classes (124.3).

$$Refactoring\ density = \frac{no\ of\ refactorings}{\sqrt{average\ code\ size}} \tag{9.1}$$

We utilize the refactoring dataset to answer this research question. In addition, we utilized the tool SlOCCount [1] to determine the code size of each snapshot of a class involved in refactoring. SLOCCount can count the physical source line of code of a java source file. We first compare the average refactoring frequency between data-access and regular classes. Second, we plot the distribution of refactoring density for data-access classes and regular classes using a violin plot. Third, we examine specific types of refactoring instances and compare their prevalence.

9.2.2 Findings

Results of our analysis show that:

[1] https://dwheeler.com/sloccount/

Figure 9.1 Violin plot of distribution of code size between data-access classes and regular classes.

◇ **Refactorings are not equally prevalent in data-access classes and regular classes.**

We have *80,074* refactoring instances associated with data-access classes and *2,582,221* refactoring instances associated with regular classes. The refactoring dataset contains *9,073* data-access classes and *94,570* regular classes. The average refactoring in data-access classes (8.82) is lower than regular classes (27.3) showing that data-access refactoring is less prevalent compared to regular refactoring. While the average refactoring frequency provides insight into the overall prevalence of refactoring, it gives equal weight to each class. However, there could be the case that some classes have more attention from developers compared to other classes. To investigate this, we plotted the distribution of refactoring density for data-access classes and regular classes.

◇ **The refactoring density of data-access classes and regular classes has a statistically significant difference in distribution. However, the difference is negligible considering the effect size.**

Figure 9.2 shows the distribution of refactoring density among data-access classes denoted by “DAC” and regular classes denoted by “Regular”. The 25, 50, and 75 percentiles are indicated in the plot. We can see that there is a difference in the distribution of refactoring density between data-access classes and regular classes. The maximum refactoring density is 63.59 for data-access classes and, 1518.9 for regular classes. We can see that for both regular and data-access classes, the refactoring density is small with median value of 0.2243 and 0.2261 respectively. Since the number of regular classes is much larger than the number of data-access classes, we randomly selected, 9073 regular classes and performed a Wilcoxon

rank-sum test to see if there is a statistically significant difference between the distribution of refactoring densities. We define the null hypothesis as H_0*: The distribution of refactoring density between data-access class and regular class is equal.* We rejected H_0

with a p-value $< 1.83e^{-9}$, 95 percent confidence interval between 0.0118 and 0.0229, and a difference in location of 0.0174. However, the Cliff's Delta effect size of 0.05 (negligible) indicates that the difference in distribution in refactoring frequencies is negligible.

We further analyzed the top five data-access classes in refactoring density and observed that they have large code sizes and long methods. Such classes implemented core data-access functions or schema definitions. The data-access class with one of the highest refactoring densities is ApiMgtDAO.java [2] from the project Carbon-apimgt. This class implements the API management data-access functionalities with average code size of 11, 156.55 SLOC and a refactoring density of 10.34.

⋄ **The distribution of data-access refactoring density is similar between SQL subject systems and NoSQL subject systems.**

The average data-access refactoring density of SQL subjects is (0.726) which is slightly lower than that of NoSQL systems (0.735). We performed a Wilcoxon rank-sum test to compare the distribution of refactoring density between SQL data-access classes and NoSQL data-access classes. We have a weak evidence to reject the hypothesis, H_0*: The distribution of refactoring density between SQL data-access classes and No-SQL data-access classes is equal.*, with (P-value 0.038, 95 percent confidence interval between 0.0007 and 0.0285, and a difference in location of 0.0142) with Cliff's Delta effect size of 0.047 (negligible). This shows that the distribution of data-access refactoring density is similar between SQL and NoSQL subject systems.

We analyzed the specific types of refactoring instances that are prevalent both in data-access classes and regular classes. Table 9.1 summarizes the most prevalent refactoring types in data-access and regular classes.

The most prevalent refactoring type in data-access classes is, *Change Parameter Type* (8%) which is associated with changing the type of objects passed as an input to methods. All the prevalent refactoring types in data-access classes are either variable level or method level, which shows that simpler refactoring activities that do not span more than one class are preferred by the developers over refactoring types that require touching multiple classes. Besides changing APIs, refactoring in data-access classes is often associated with improving program comprehension (Rename variable, method, and parameter (16.4% combined))

[2] https://bit.ly/36XxUyk

changing access level of methods and moving methods between classes.

We further analyzed the prevalence of data-access refactorings between SQL and NoSQL subject systems. The most prevalent data-access refactoring is *Rename Variable* for SQL systems (9.95%) and *Change Parameter Type* for NoSQL systems (9.06%). Most of the top prevalent data-access refactorings types are similar between SQL and NoSQL subject systems.

Move Attribute refactoring is the most prevalent, spanning 57% of all refactorings in regular classes. *Move Attribute* refactoring is aimed at removing smells such as *Shotgun Surgery* and reduces unnecessary class coupling. Another prevalent refactoring, *Change Attribute Access Modifier* (23.3%), focuses on improving encapsulation. This shows that most of the refactoring on regular classes is done to improve inter-class coupling and encapsulation.

Table 9.1 Top ten most prevalent refactoring types. The table shows the number of refactoring instances (count) and percentage against the total number of data-access and regular class refactoring instances.

Data-access class			**Regular class**		
Refactoring Type	**count**	**percentage**	**Refactoring Type**	**count**	**percentage**
Change Parameter Type	6455	8.062	**Move Attribute**	1472101	57.009
Change Variable Type	5655	7.063	Change Attribute Access Modifier	602074	23.316
Rename Method	4941	6.171	Change Parameter Type	37751	1.462
Rename Variable	4399	5.494	Add Method Annotation	35145	1.361
Rename Parameter	3788	4.731	Change Variable Type	31733	1.229
Change Return Type	3676	4.591	Rename Method	27235	1.055
Add Method Annotation	3371	4.21	Rename Parameter	25251	0.978
Add Parameter	3196	3.992	Change Return Type	24665	0.955
Change Method Access Modifier	3042	3.799	Rename Variable	23160	0.897
Move Method	2850	3.559	Add Parameter	22499	0.871

Figure 9.2 Violin Plot of the distribution of refactoring density in data-access classes and regular classes

9.3 RQ 4.2: Evolution of refactorings

In this section, we discuss our analysis of how the prevalence of refactorings varies as systems evolve and if the prevalence of refactorings is affected by the release deadline in the subject systems. Software becomes complex as it evolves due to the added and improved features. It is interesting to study if the evolution of software determines the type and prevalence of refactoring activities performed by developers on data-access classes. If the refactoring activities are equally frequent in all stages, it shows that developers considered refactoring as a regular activity. Otherwise, it may suggest that refactoring activities in data-intensive systems are triggered by the increasing complexity introduced during evolution. It is also interesting to study how refactoring activity is affected by distance from official releases. It indicates if the release pressure on developers affects their decision to refactor code.

9.3.1 Analysis approach

We use *Relative Commit Time* as a metric to express when a refactoring happens. Since every refactoring is associated with a commit, we use the associated commit time. Due to the variation in the maturity of the subject systems, we use relative time rather than an absolute time for correct comparison. *Relative Commit Time* is computed using Equation 9.2 where distance is computed as the number of commits the subject system has at the time of the refactoring and *totalCommits* is the total number of commits the subject system has at the time of the experiment.

$$RelativeCommitTime(\%) = \frac{distance * 100}{TotalCommits} \tag{9.2}$$

Another analysis we can do is to study if the prevalence of refactorings are affected by pressure from release deadlines. We hypothesize that developers due to the pressure from release deadlines may be forced to prioritize other activities such as bug fixing over refactoring. We use the metric *Distance before release* to measure how far the refactorings happened before the time of official release commits. Equation 9.3 defines *Distance before release* metric where R_n is the commit of the official release R, C_{ref} be the commit where the target refactoring happened, R_p is the commit where the predecessor official release happened and $index(x)$ is the number of commits of the system when x happened. For a refactoring, commit *Distance before release* can be greater than or equal to zero and less than 100%. The further the refactoring commit happens before release, the higher the value. We removed refactoring instances that happen after the latest official release of the subject systems because the *distance from release* can not be computed. We also excluded the system Oltpbench because

it does not have any tagged releases in GitHub.

$$Distance\ before\ release(\%) = \frac{(index(R_n) - index(C_{ref})) * 100}{(index(R_n) - index(R_p))} \tag{9.3}$$

We linked the commit information and the refactoring dataset using the commit ID. Then, we computed the *Relative Commit Time* for each refactoring using Equation 9.2 and computed *Distance before release* using Equation 9.3. We first show the distribution of *Relative Commit Time* for data-access refactoring and regular refactoring instances using violin plots. Second, we statistically compare the two distributions using Wilcoxon rank-sum test. Third, we compare the distributions between data-access and regular refactorings at the subject system level. Finally, we show the summary of the *Relative Commit Time* distribution for the most prevalent data-access refactoring types. We repeat a similar analysis for the *Distance before release* metric.

9.3.2 Findings

Results show that:

◇ **The median *Relative Commit Time* for data-access refactorings is higher compared to regular refactorings.**

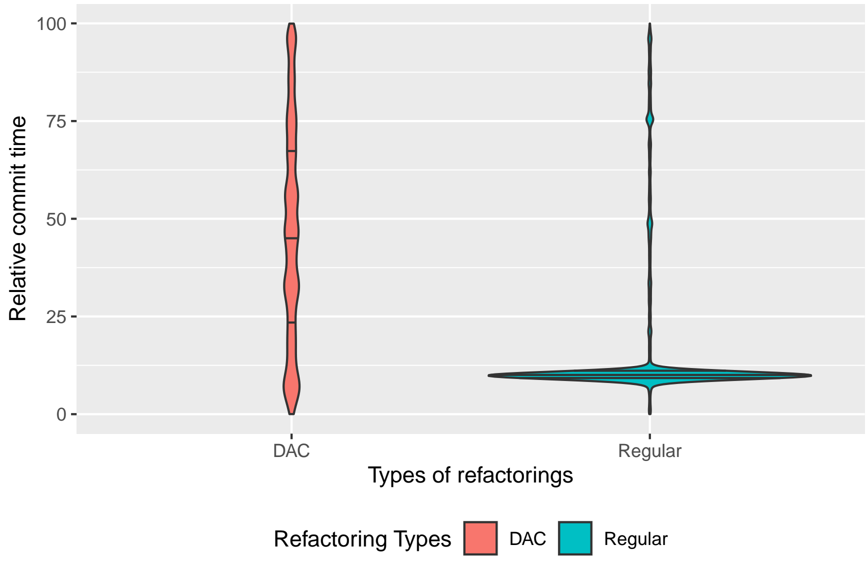


Figure 9.3 Violin Plot of the distribution of *Relative Commit Time* in data-access refactorings (DAC) and regular refactorings

Figure 9.3 shows the distribution of *Relative Commit Time* for data-access refactorings and

regular refactorings. As can be seen on Figure 9.3, the *Relative Commit Time* is different between data-access and regular refactorings. Many regular refactorings occurred in the first 25% *Relative Commit Time* or during the beginning stages of the projects. However, for data-access classes, the refactoring activities are distributed across all project evolution stages, with more tendency to later stages. The median *Relative Commit Time* for data-access refactorings (45.97%) is higher compared to regular counterparts (9.89%). Furthermore, 25% of data-access refactorings occurred below *Relative Commit Time* of 23.17% while 50% of regular refactorings occurred below 9.89%. This shows that developers often perform regular refactorings during the beginning stages of the subject systems' evolution, while they perform data-access refactorings throughout the evolution of the subject systems.

◇ **The difference in the distribution of *Relative Commit Time* between data-access and regular refactorings is statistically significant.**

We performed Wilcoxon rank-sum test on *Relative Commit Time* using statistically significant samples of both data-access and regular refactoring instances (Sample size=1000, confidence level=99%, and margin of error $<$ 4%) and rejected the null hypothesis with (W =758680, p-value $<$ 2.2e-16, and Cliff's Delta effect size of 0.55 (large). This indicates that the difference in distribution between the *Relative Commit Time* of data-access refactorings and regular refactorings is statistically significant with large effect size.

◇ **Data-access refactorings have a higher median *Relative Commit Time* in the majority of the subject systems.**

We further analyzed the *Relative Commit Time* between data-access refactorings and regular refactorings by splitting the data by subject systems. The median *Relative Commit Time* is higher for data-access refactorings compared to regular refactorings in 16 out of 29 subject systems (55.17%). Furthermore, 10 out of the 13 systems where the regular refactoring commit time is higher are associated with a low number of data-access classes and data-access refactorings. The low number of data-access refactorings is expected given the low number of data-access classes.

◇ **The** *Move Method & Add Method Annotation* **refactorings often occur at later stages of the evolution of the subject systems' data-access classes.**

Another interesting analysis would be to investigate what type of data-access refactorings are performed by developers at different stages of the subject systems' evolution. We focused our analysis on the top 10 most prevalent data-access refactoring types from **RQ 4.1**. Table 9.2 shows the summary of the distribution of *Relative Commit Time* for the top ten most prevalent data-access refactorings, sorted by the median. If we divide the median of *Relative*

Commit Time into four quartiles, we can see that *Move Method & Add Method Annotation* refactorings belong to the fourth quartile, indicating that this refactoring is often performed by developers at the latest stage of the evolution of the systems. The median *Relative Commit Time* of the other refactoring types are found at the second and third quartiles, indicating that they are performed in the middle stages of the evolution of the subject systems. On the other hand, *Add Parameter Modifier* is often performed at the early stages (median=40.6%) compared to other prevalent data-access refactorings.

◇ **Data-access refactorings have a higher median *Relative Commit Time* in SQL based systems compared to NoSQL based systems.**

We compared the distribution of *Relative Commit Time* of data-access refactorings between SQL based subject systems (Minimum=0.02%, first quartile=30.31%, Median=48.35%, third quartile=76.39% and Maximum=99.24%) and NoSQl-based subject systems (Minimum=0.03%,first quartile=21.04, Median=44.55%, third quartile=65.82%, and Maximum= 99.93%). The median *Relative Commit Time* of data-access refactorings is slightly higher in SQL-based data-intensive systems compared to NoSQL-based data-intensive systems. This shows that developers of NoSQL-based data-intensive systems often apply data-access refactorings at earlier stages of the system's evolution compared to SQL-based data-intensive systems' developers.

Table 9.2 Distribution of *Relative Commit Time* for the top ten prevalent data-access refactoring types.

Refactoring Type	count	mean	std	min	25%	50%	75%	max
Move Method	2850	49.802	26.679	0.237	30.298	**54.514**	67.038	99.408
Add Method Annotation	3371	51.364	28.092	0.587	26.940	**54.366**	73.682	99.113
Change Method Access Modifier	3042	49.674	26.380	0.214	29.600	**49.543**	71.225	99.408
Rename Variable	4399	47.911	28.062	0.051	25.318	**47.100**	71.792	99.408
Add Parameter	3196	48.783	27.180	0.051	27.666	**47.098**	70.468	99.066
Change Variable Type	5655	43.441	27.762	0.224	17.201	**42.799**	65.821	99.408
Change Return Type	3676	45.595	29.934	0.122	19.859	**42.799**	65.933	99.408
Change Parameter Type	6455	42.570	26.735	0.051	19.375	**42.761**	57.127	99.930
Rename Method	4941	44.502	27.633	0.122	22.817	**42.537**	67.482	99.578
Rename Parameter	3788	44.329	29.187	0.024	16.547	**40.597**	67.717	99.408

◇ **Most of the refactorings happen shortly after official releases.**

When we see the distribution of *distance from release* for the refactorings in our subject systems, the average value is 74.15% and 75% of the refactorings have more than 81% distance from release. This shows that refactorings mostly happened shortly after official releases and as the time of a release approaches the number of refactoring activities substantially decrease.

◇ **Data-access refactorings tend to be similarly prevalent regardless of the dis-**

tance from release while the majority of regular refactorings happen shortly after release.

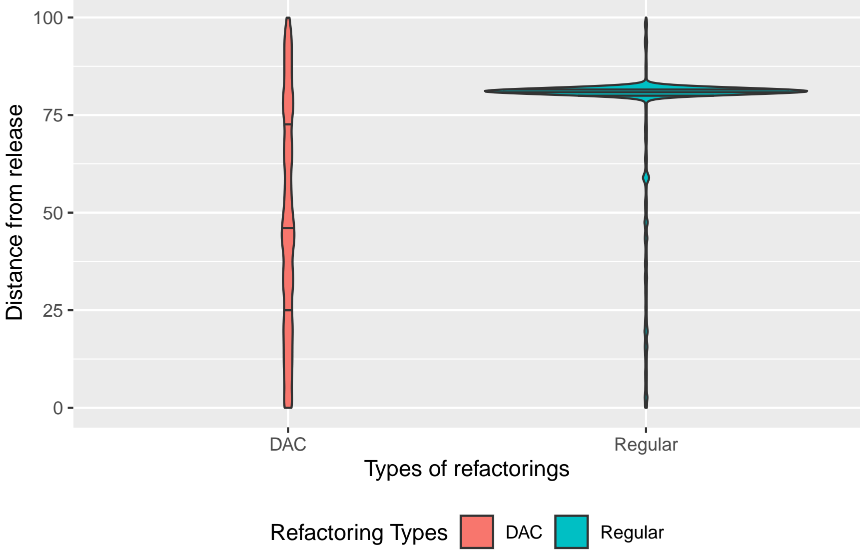


Figure 9.4 Violin Plot of the distribution of *distance from release* in data-access refactorings (DAC) and regular refactorings

Figure 9.4 shows the distribution of *distance from releases* for data-access and regular classes. 75% of the regular refactorings have *distance from releases* greater than 81%. On the other hand, the *distance from release* for data-access refactorings is more distributed across all values (median=44.7% and mean=47.4%) compared to regular refactorings. The mean and median being lower than 50% shows that data-access refactorings tend to be applied more before release than after release.

◇ **The difference in distribution of *distance from release* between data-access refactoring and regular refactoring is statistically significant with medium effect size.**

We performed Wilcoxon rank-sum test on *Distance before release* using 1000 random samples for both data-access and regular refactorings (confidence level = 99%, and margin of error < 4%), and rejected the null hypothesis with (W = 236202, p-value < 2.2e-16, and Cliff's Delta effect size of 0.53 (large). This indicates that the difference in distribution between the *distance from release* of data-access refactorings and regular refactorings is statistically significant, with a large effect size.

We compared the *distance from release* distribution between data-access and regular refactorings for each subject system. In the majority of the subject systems (65.4%), data-access

refactorings have a lower distance from release than regular refactorings. For instance, The median *distance from release* for the Adempiere project, with the largest number of refactoring instances, is 81% for regular and 54.6% for data-access refactorings.

◇ **The top ten most prevalent data-access refactoring types have roughly similar *distance from release* distribution. However,** *Move Class* **refactoring has a median value of 25.85% which shows that this refactoring is mostly applied immediately after releases.**

Table 9.3 shows the distribution of *distance from release* for the top ten prevalent data-access refactoring types. Most of the prevalent data-access refactoring types have a median *distance from release* between 50% and 44%. *Move Class* refactoring has the smallest *distance from release* (25.85%). This means *Move Class* refactoring is mostly applied close to release. On the other hand, *Extract Method* have the highest median *distance from release* (61%) which shows that this type of refactoring is often applied near (and before) the releases dates.

Table 9.3 Distribution of distance from release for the top ten prevalent data-access refactoring type

Refactoring type	**Count**	**Mean**	**Std**	**Min**	**25%**	**Median**	**75%**	**Max**
Extract Method	1911	54.259	28.359	0.0	30.265	**61.793**	78.635	99.752
Add Parameter	2102	51.537	27.387	0.0	29.483	50.367	75.306	99.939
Change Variable Type	3539	50.401	26.686	0.0	30.256	49.236	73.675	99.495
Rename Variable	3118	48.392	27.475	0.0	23.027	48.313	72.616	99.837
Change Parameter Type	2174	48.770	27.605	0.0	27.454	46.892	74.231	99.826
Rename Method	2515	46.055	28.479	0.0	21.619	44.652	70.416	99.752
Add Method Annotation	2370	45.597	30.721	0.0	17.964	44.114	74.025	99.035
Move Method	2092	42.676	24.486	0.0	24.876	44.114	56.995	99.939
Change Method Access Modifier	2143	42.415	29.546	0.0	15.816	**39.120**	67.188	99.752
Move Class	1968	36.015	24.520	0.0	21.769	**25.850**	49.483	97.907

◇ **Data-access refactorings have a higher median distance from release in SQL based systems compared to NoSQL based systems.**

Comparing the distribution of distance from release of data-access refactorings between SQL based subject systems (Minimum=0%, first quartile=27.74, Median=49.07%, third quartile=76.14% and Maximum=99.94%) and their NoSQL counterparts (Minimum=0%, first quartile=21.77, Median=44.11%, third quartile=71.26%, and Maximum=99.94%) show that developer's of NoSQL based data-intensive systems often apply data-access refactorings earlier after releases compared to SQL-based data-intensive system developers.

9.4 RQ 4.3: Data-access refactoring activities and SQL code smells

In this section, we investigate if SQL code smells are fixed during data-access refactoring activities by analyzing the intersection of SQL code smells dataset and the refactoring dataset. All our SQL-based subject systems use SQL statements for database interactions, including fetching and inserting data. Hence, it is interesting to investigate if SQL statements change during data-access refactoring activities. Furthermore, SQL code smells are shown to be prevalent in data-intensive systems(RQ2.4) and it is also interesting to see if queries that contain such smells are touched during data-access refactorings. Unfortunately, to the best of our knowledge, we did not find a tool that can extract NoSQL data-access statements and corresponding data-access anti-patterns. Hence, we excluded NoSQL subject systems from the analysis in this research question.

9.4.1 Analysis approach

To investigate if data-access refactoring instances touch SQL queries and SQL code smells, we matched the **refactoring dataset** excluding NoSQL subject systems with query dataset and smell dataset using *line level* and *method level* matching. *Line level* matching has more strict criteria than method level matching, since the line number of the query or smell should be the same as the line number of the code involved in refactoring. On the other hand, *Method level* matching is less restrictive, as it checks if the query or smell is part of the method involved with the refactoring. For both cases, the subject system, revision, and class name should match the query, smell, and refactoring dataset.

9.4.2 Findings

The results show that:

◇ **Only small fractions of data-access refactorings touch code lines containing SQL query.**

- Data-access refactoring and SQL queries

Our result shows that a few refactoring instances touched SQL queries (using line level matching). We have 18, 892 data-access refactoring instances, out of which only 86 instances (0.45%) touched an SQL query. When we further analyze the types of the 86 refactoring instances involved with SQL queries, the most prevalent type is *Change Return Type* with 17 refactoring instances (19.76%), followed by *Add Parameter* with 12 refactoring instances (13.95%), *Extract Method* with 10 refactoring instances (11.63%), and *Change Variable Type*

with 7 refactoring instances (8.14%).

◇ **30% of data-access refactorings were applied on a method that contains SQL queries.**

When we consider method level matching, we get more SQL queries associated with the refactoring instances as expected. Results show that, 5607 refactoring instances (29.68%) contained SQL queries inside the target methods. When we see the refactoring types, the most prevalent type is *Rename Variable* with 821 instances (14.64%), followed by *Change Variable Type* with 603 instances (10.75%), *Add Parameter* with 409 instances (7.29%), and *Change Parameter Type* with 355 instances (6.33%). Such types of refactorings focus on fixing linguistic smells or implementing API changes and are not associated with modifying query or improving SQL code smells.

Data-access refactoring and SQL smells

SQL smells are detected from SQL queries. Hence, the smell dataset is a subset of the query dataset. Although SQLInspect can detect *Implicit columns, Fear of the Unknown, Ambiguous Groups and Random Selection*, the smell dataset only contains instances of *Implicit columns* smell and *Fear of the unknown* SQL smell. Indeed, we showed that both types of SQL smells are prevalent in data-intensive systems in Chapter 7.

◇ **1.35% of data-access refactorings in SQL-based subjects were involved with a method containing queries with a SQL code smell.**

Using the line level matching, we did not find any instance of SQL code smell associated with data-access refactoring instances. On the other hand, we found 256 refactoring instances (1.355%) that target methods containing a query with SQL Code smell. From the 256 refactoring instances, *Rename Variable* takes the larger share with 60 instances (23.44%), followed by *Extract Method* with 30 instances (11.72%), *Rename Method* with 26 instances (10.156%), *Add Parameter* with 24 instances (9.38%), and *Change Variable Type* with 21 instances (8.2%).

When examining the specific types of SQL code smells associated with the refactoring instances, we observe that 200 instances (78.12%) are associated with only *Implicit Columns* SQL smell, and 49 instances (19.14%) are associated with only the *Fear of the unknown* SQL smell.

◇ **Developers of the SQL subject systems do not address SQL code smells.**

The previous findings indicate that SQL queries and SQL code smells are not modified during data-access refactorings. The next natural question would be to investigate how the SQL

code smells are removed if they are removed. We leveraged our query dataset and smell dataset to investigate how SQL code smells are addressed.

We define the candidate SQL smell removal commit for a source file as the commit which (1) modifies the target file; (2) no instance of the target smell was detected by SQLInspect for this file in this commit, and (3) there is at least one instance of the target smell detected in the immediate predecessor commit for the same file. With these criteria, we only found seven candidate *Implicit Columns* removal commits. We then manually examined each instance to verify if the removals are real removals. The result of our manual investigation shows that out of the seven removal candidates, one instance is a false-negative (the smell exists in the removal commit but SQLInspect missed it). The remaining six removals were accidental removals, which means that the method containing the queries is deleted by the removal commit. This shows that no instance addressed *Implicit Columns* smell removal by listing the columns in the SELECT clause. We did not find any removal candidate for the *Fear of the unknown* SQL code smell.

9.5 RQ 4.4: Co-occurrence of refactorings in data-access classes

The result of RQ 4.1 shows that different types of refactorings are applied to data-access classes. However, composite refactorings, 2 or more single refactorings applied together, could be performed on data-access classes. Investigating this could provide insights into the high-level intentions of the developers' refactorings in data-access classes, and how different are they compared to the regular refactorings. Furthermore, the findings of this research question could be leveraged in the automatic recommendation of refactorings and refactoring prioritization tasks. In this section, we present our analysis of the co-occurrence of different refactoring types using the Apriori algorithm and Cramer's V test of association.

9.5.1 Analysis approach

We started with the data-access refactoring dataset and prepared a matrix with a row corresponding to one revision (commit) of a repository and columns representing the occurrence of each refactoring type in the commit. The occurrence is either 1 (the refactoring type is performed) or 0 (the refactoring type is not performed). We run the Apriori algorithm on this dataset to compute the co-occurrence of different refactoring types. The considered co-occurrence metrics are *support, confidence, lift, leverage, and conviction.*

To generate frequent item sets, Apriori algorithm needs a hyper-parameter to consider the minimum co-occurrence. We set the minimum support to 0.01 considering the small number

of occurrences for most refactoring types. To generate the association rules, we used *lift* as a metric with a minimum threshold of 1. This means that Apriori only selects associations whose lift is greater than 1. A lift value of 1 indicates that the candidate refactoring types occur independently.

For the top ten most co-occurring refactoring types, we conducted a Chi-squared test to measure the statistical significance of their association and cramer's V test of association, which considers the effect size.

9.5.2 Findings

The results show that:

◇ ***Rename Variable* and *Change Variable Type* are the most co-occurring refactoring types in data-access classes.**

Table 9.4 summarizes the results of Apriori algorithm. This table shows top ten co-occurrences ranked by *support* metric. *Rename Variable* and *Change Variable Type* registered the highest support of 0.085 and leverage of 0.051. We can also observe that renaming both variables and parameters is associated with changing the types of variables or parameters, respectively. Furthermore, the highest Lift (3.965) and conviction (1.777) were registered by the pair *Change Attribute Type & Change Parameter Type*.

Table 9.4 Top ten co-occurrence of data-access refactorings ranked by the support and associated co-occurrence metrics

Item 1	Item 2	Support	Confidence	Lift	Leverage	Conviction
Rename Variable	Change Variable Type	**0.085**	0.443	2.492	**0.051**	1.477
Change Parameter Type	Rename Parameter	0.066	**0.513**	3.685	0.048	1.768
Change Parameter Type	Change Variable Type	0.058	0.454	2.552	0.035	1.506
Rename Variable	Rename Parameter	0.057	0.296	2.127	0.030	1.223
Change Attribute Type	Change Parameter Type	0.056	0.510	**3.965**	0.042	**1.777**
Change Return Type	Change Variable Type	0.054	0.469	2.636	0.033	1.548
Rename Method	Rename Parameter	0.053	0.311	2.235	0.029	1.250
Change Attribute Type	Change Variable Type	0.052	0.476	2.674	0.033	1.568
Rename Attribute	Change Attribute Type	0.051	0.480	4.351	0.039	1.711
Rename Method	Rename Variable	0.051	0.298	1.557	0.018	1.152

◇ **All pairs of refactoring types show statistically significant co-occurrence using the Chi-squared test.**

We performed a Chi-squared test of association using the following null hypothesis. H_0 : *for a given refactoring type pairs* $< R_1, R_2 >$, R_1 *and* R_2 *do not co-occur in data-access classes.* Table 9.5 shows that all refactoring pairs in 9.4 have a P-value < 0.0001. Hence, we reject H_0

Table 9.5 Result of Chi-squared test and cramer's V test for the most co-occurring refactoring types in data-access classes

Item 1	**Item 2**	**Chi-squared P-value**	**cramer's V**	**cramer's V interpretation**
Rename Variable	Change Variable Type	<0.0001	0.337	Moderate
Change Parameter Type	Rename Parameter	<0.0001	**0.414**	Moderate
Change Parameter Type	Change Variable Type	<0.0001	0.276	Low
Rename Variable	Rename Parameter	<0.0001	0.22	Low
Change Attribute Type	Change Parameter Type	<0.0001	**0.4**	Moderate
Change Return Type	Change Variable Type	<0.0001	0.273	Low
Rename Method	Rename Parameter	<0.0001	0.225	Low
Change Attribute Type	Change Variable Type	<0.0001	0.273	Low
Rename Attribute	Change Attribute Type	<0.0001	**0.406**	Moderate
Rename Method	Rename Variable	<0.0001	0.123	Low

for all of those cases showing that the co-occurrence of the refactoring pairs is statistically significant.

⋄ **All of the refactoring type pairs have either moderate or low association based on cramer's V test.**

cramer's V test returns association value between 0 (no association) and 1 (maximum association). An association value < 0.1 is interpreted as *very low* association. A value between 0.1 and 0.3 is interpreted as *low* association. A value between 0.3 and 0.5 is interpreted as *moderate* association. A value greater than 0.5 is interpreted as *high* association.

Table 9.5 shows that the association between the refactoring types *Change parameter type & Rename parameter* is the strongest (with a value of 0.414), followed by *Rename Attribute & Change Attribute Type* (0.406), and *Change Attribute Type & Change Parameter Type* (0.4). The rest of the refactoring type pairs have low associations.

⋄ **Similar sets of refactoring types are observed to co-occur in regular classes compared to data-access classes.**

Table 9.6 summarizes the output of the Apriori algorithm and corresponding statistical test of association for the top ten co-occurring refactoring types in regular classes. Most of the top refactoring type pairs in the table are similar to that of data-access refactoring pairs. All pairs of the refactoring types have a statistically significant co-occurrence based on the Chi-squared test. Furthermore, *Rename Variable & Change Variable Type* refactoring types showed the highest association value (0.449), followed by *Change Parameter Type & Rename Parameter* (0.462). Most of the co-occurring refactoring types have moderate association based on Cramer's V test.

The refactoring pairs *Rename Variable & Change Variable Type* co-occur in both data-access and regular refactorings with the highest support value. We can also observe in data-access

Table 9.6 Apriori algorithm result for top ten regular refactoring types and corresponding statistical test

Item 1	**Item 2**	**Support**	**Confidence**	**Lift**	**Leverage**	**Conviction**	**Chi-squared P-value**	**cramer's V**	**cramer's V interpretation**
Rename Variable	Change Variable Type	**0.0058**	0.4431	36.0668	**0.0056**	1.7735	<0.0001	**0.449**	Moderate
Change Parameter Type	Rename Parameter	0.0047	**0.4482**	**46.9225**	0.0046	**1.7949**	<0.0001	**0.462**	Moderate
Change Parameter Type	Change Variable Type	0.0038	0.3653	29.7353	0.0037	1.5562	<0.0001	0.328	Moderate
Rename Parameter	Rename Variable	0.0037	0.3909	29.9126	0.0036	1.6204	<0.0001	0.325	Moderate
Change Return Type	Change Variable Type	0.0035	0.4226	34.3994	0.0034	1.7106	<0.0001	0.341	Moderate
Rename Method	Rename Parameter	0.0029	0.3242	33.9375	0.0028	1.4655	<0.0001	0.304	Moderate
Change Parameter Type	Change Return Type	0.0028	0.2720	32.4293	0.0028	1.3621	<0.0001	0.296	Low
Change Parameter Type	Rename Variable	0.0027	0.2565	19.6244	0.0025	1.3274	<0.0001	0.219	Low
Rename Variable	Extract Method	0.0025	0.1884	23.2097	0.0024	1.2221	<0.0001	0.23	Low
Add Parameter	Change Variable Type	0.0024	0.2112	17.1895	0.0022	1.2521	<0.0001	0.19	Low

refactorings that rename refactorings are applied together with changing type of variables, attributes, or parameters. This shows that developers try to maintain code understandability by updating identifiers to reflect the change in type of the variables they identify.

◇ **In all cases, only two single refactoring types participate in composite refactoring for both data-access and regular refactorings.**

For both data-access and regular refactorings, we did not observe more than two refactoring pairs co-occurring. This shows that the applied composite refactorings do not involve more than two single refactoring instances.

9.6 RQ 4.5: Profile of developers performing data-access refactorings

In this section, we discuss our analysis regarding the profile of developers. We first define the developer profile metrics and then compare profiles of developers involved in data-access refactoring with developers that are not involved in data-access refactoring. Given that data-access operations are core components of data-access classes, it is interesting to characterize developers' profiles involved in data-access refactoring for the following reasons. One, we will learn the requirement of data-access refactoring in terms of developers' profiles. Two, the findings could help guide practitioners to pick the appropriate personnel to assign data-access refactoring tasks. Three, the findings could help in developing automated recommendations for developers for refactoring tasks. In addition, assessing the contribution of developers, as part of their profile, helps us to investigate if the responsibility of maintaining code quality by refactoring is shared by all developers or a few developers.

9.6.1 Analysis approach

We combined the refactoring dataset and commit information to answer this RQ. Our metrics for the profile of developers include metrics to evaluate the contribution of the developer over all projects, popularity of the developer, and contribution of the developer on the subject systems. We used the metrics provided by GitHub, on developers' profile page, as a measure of developer's popularity and global contribution. Similar metrics were also used to profile developers in the literature (Eg. [147,148]). We define the metrics in the following paragraph.

- *Followers* : The number of followers a developer have. This information can be obtained from GitHub based on the username of the developer.

- *Public repositories*: The number of public repositories owned by the developer. This measures the experience of the developer in diverse projects besides the subject systems.

This is obtained from GitHub.

- *Public gists*: The number of public gists made by the developer obtained from GitHub.
- *Global contribution*: The average number of contributions for the last five years. The contributions include: committing to a repository, opening an issue, opening a discussion, answering a discussion, proposing and submitting a pull request to review.
- *Local contribution (Loc.cont.)*: Measures the contribution of the developers on the subject systems by authoring a commit. Equation 9.4 defines this metric where *Authored commits* are the number of commits where the developer is the author and *Total Commits* are a total number of commits in the subject system.

$$Loc.cont.(\%) = \frac{Authored\ commits * \ 100}{Total\ commits} \tag{9.4}$$

- *Refactoring contribution (Ref.Cont.)*: This metric measures the contribution of the developer on refactoring the subject systems. Equation 9.5 defines this metric, where *Dev.refactorings* are a number of refactorings performed by the developer and *Total refactorings* are a total number of refactorings detected in the subject system.

$$Ref.Cont.(\%) = \frac{Dev.refactorings * \ 100}{Total\ refactorings} \tag{9.5}$$

Among the defined metrics, *Refactoring contribution* and *Local contribution* can be used as a proxy to measure the developer's experience in the subject system, while the remaining metrics can be used as a proxy to measure the global experience of developers.

We compare the profile of developers involved in data-access refactoring against developers involved only in regular refactoring (baseline) to identify developers that only performed regular refactorings (Regular refactoring developers) and developers that performed either data-access only or data-access and regular refactorings (data-access refactoring developers). Every refactoring is associated with a commit, and a commit has the associated author and committer. We can identify both types of developers; first by linking the refactoring dataset and commit information based on the refactoring commit ID and second, extracting the author of the commit. We identified 501 regular refactoring developers and 358 data-access refactoring developers from our subject systems.

After the developers were identified, we collected the metrics *followers, public repositories, public gists and global contributions* for all developers by querying the GitHub API and the remaining two metrics are computed from our refactoring dataset and commit information.

Since we have multiple metrics to profile developers, it is interesting to identify which metrics discriminates better between the two developer groups. To answer this, we evaluated the feature importance of each metric using multiple classification models on our developer dataset. In particular, We used Decision Tree, Random Forest, Gradient Boost, AdaBoost classifiers from ensemble models family, and Logistic Regression model implemented in SciKit-learn [149] library as classification models. We select those models because they provide feature importance values and are often used in other studies to compute future importance (e.g., [150,151]). We used the default hyper-parameters for all models and performed 10-fold cross validation to minimize the effect of random variation. In the following, we report the median feature importance for each metric. In addition, we report the median value of coefficients of the logistic regression model corresponding to each metric. Table 9.7 shows the feature importance in percentage and logistic regression coefficient for all developer profile metrics. We also normalized the training data before fitting to the logistic regression model.

In addition to the feature importance, we also conducted a statistical test of significance using Wilcoxon rank-sum test. We define the null hypothesis as *H0: the distribution of the metric value between data-access refactoring developers and regular refactoring developers is similar.* We also used Cliff's Delta non-parametric test to estimate the effect size of the metrics.

9.6.2 Findings

The results show that:

◇ ***Refactoring contribution*** **and** ***local contribution*** **metrics are the most distinguishing metrics between developers performing data-access refactoring and regular refactoring.**

As shown in Table 9.7, *Refactoring contribution* has the highest median feature importance in all ensemble models. The highest median feature contribution was 48.95% using the Gradient Boost model. Similarly, *Local contribution* has the second-highest median percentage contribution in all ensemble models (maximum=24.86 using Random Forest model). This metric also has the highest Logistic Regression coefficient of -0.49, followed by *Refactoring contribution* (-0.27). On the other hand, global profile metrics such as *Public repositories, public gists, Followers and Global contribution* have lower feature contribution.

Table 9.8 shows the result of the statistical tests and effect size associated with each metric. We reject H_0 for this metric with a large effect size of 0.51 for *Refactoring contribution* and *Local contribution* (0.49). We also reject H_0 for *Global contribution & Followers* metrics,

Table 9.7 Median feature importance values in percent and Logistic regression coefficient of developers' profile metrics

Model	Followers	Public gists	Public repositories	Local contribution	Refactoring contribution	Global contribution
DecisionTreeClassifier	12.13	5.81	10.99	18.09	**39.29**	12.89
RandomForestClassifier	11.15	7.03	12.7	24.86	**28.52**	15.55
GradientBoostingClassifier	8.43	6.77	5.09	21.33	**48.95**	9.22
AdaBoostClassifier	13.5	11	8.5	19.5	**32**	14.5
Logistic regression(coefficients)	0.01	0.12	-0.06	**-0.49**	-0.27	-0.06

however the effect size is small (0.156) and negligible (0.095), respectively. We failed to reject H_0 for the remaining metrics. This shows that both data-access developers and regular refactoring developers have roughly similar global profiles, but they are distinguishable based on local change and refactoring contributions. While these findings show that the local contribution and refactoring contribution of developers refactoring data-access classes is different from regular refactorings, it does not show how refactoring responsibility is shared among developers.

◇ **Developers performing data-access refactoring have higher local contribution as well as refactoring contribution compared to developers performing regular refactoring.**

Developers involved in data-access refactoring have higher refactoring contribution (mean =7.40%, median=0.686%) than developers involved in regular refactoring (mean=1.37%, median=0.04%). Data-access developers also showed higher local contribution (mean=6.66%, median=1.07%) compared to regular refactoring developers (mean=1.45%, median=0.12%).

Table 9.8 Comparison of developer profile metrics between data-access refactoring developers and regular refactoring developers. We reject the null hypothesis, with a large effect size for the bolded metrics. The Cliffs Delta value is also bolded for metrics with a large effect size

Metric	Wilcoxon rank-sum test					CliffsDelta	
	W	P-value	Difference in location	95% confidence interval		value	Interpretation
Refactoring contribution	**135278**	**<0.0001**	**0.4477**	**0.3391**	**0.6485**	**0.5100**	**Large**
Local contribution	**133882**	**<0.0001**	**0.6483**	**0.4594**	**0.8765**	**0.4950**	**Large**
Global contribution	103515	<0.0001	36.6000	13.8000	59.1000	0.1560	Small
Public repositories	90935	0.6958	0.0000	-2.0000	3.0000	0.0150	Very small
Followers	98058	0.0170	1.0000	0.0001	2.0000	0.0952	Very small
Public gists	92324	0.4069	0.0000	-0.0001	0.0000	0.0310	Very small

◇ *Remove Parameter Modifier, Remove Variable Modifier, & Remove Variable Annotation* **refactoring types were mostly performed by developers with high *Refactoring contribution and Local contribution* in data-access classes. In regular classes** *Move And Rename Method, Extract And Move Method & Change Parameter Type* **refactoring types were associated with developers with highest *Refactoring contribution***

and Local contribution.

Another interesting analysis would be to investigate if the contribution of developers is associated with the type of refactorings they apply. In particular, we observed the distribution of *Refactoring contribution, Local contribution, and Global contribution* of developers by grouping the developers based on the refactoring types. We computed the median of each metric and sorted the refactoring types based on the median metric value from high to low.

Considering the distribution of *Refactoring contribution* over data-access classes, *Remove Variable Modifier, Remove Variable Annotation, & Remove Parameter Modifier* refactoring types were performed by developers with the highest *Refactoring contribution* ($> 69\%$). On the other hand, refactoring types *Merge Variable, Remove Parameter Annotation, & Replace Attribute* were associated with developers with low *Refactoring contribution* ($< 20\%$). The distribution of *Refactoring contribution* over regular classes show that *Move And Rename Method, Extract And Move Method, & Change Parameter Type* refactoring types have the highest median value ($> 91\%$) and *Move And Rename Method, Extract And Move Method, & Change Parameter Type* refactoring types have the lowest median value ($< 8\%$).

When we consider the metric *Local contribution* in data-access classes, *Remove Parameter Modifier, Remove Variable Modifier, & Remove Variable Annotation* refactoring types were associated with developers that have higher median *Local contribution* ($> 61.7\%$). The refactoring types *Move And Rename Attribute, Replace Attribute, & Modify Parameter Annotation* were associated with low median value ($< 14\%$). On the other hand, similar analysis on regular refactorings show that *Move And Rename Method, Extract And Move Method, & Change Parameter Type* have the highest median value ($> 79\%$) and *Modify Method Annotation, Modify Parameter Annotation, & Add Variable Modifier* have the lowest median value ($< 4\%$).

Considering the *Global contribution* metric, refactoring types *Add Attribute Annotation, Split Attribute, & Push Down Attribute* were done by developers with high *Global contribution* (> 607) in data-access classes and *Add Variable Annotation, Modify Class Annotation, & Modify Parameter Annotation* refactorings were associated with developers with low median *Global contribution* (< 318). For regular refactorings, *Move And Rename Method, Rename Class, & Change Thrown Exception Type* refactoring types were performed by developers with high global contribution (> 597) and refactoring types *Extract Subclass, Push Down Method, & Inline Method* were performed by developers with low median global contribution (< 161). We observe that the top three refactorings associated with developers with high global contribution are different from that of *Local contribution and Refactoring contribution* for both data-access and regular refactorings.

◇ **The amount of local contribution is not similar among developers. In most cases, few developers are responsible for large portions of the commits in the subject systems.**

Table 9.9 shows the number of developers and the distribution summary of the percentage contributions of all the developers for each subject system. The number of developers per subject system ranges from 7 to 349 with an average of 74.07 and a median of 43. *Carbon-apimgt* has the largest number of developers among all the subject systems. The summary of the percentage contribution shows that in most cases very few developers account for the large number of commits. For example, 97% of the commits were done by one developer, while the rest of the 3% is shared by the other 36 developers in *MyExpenses* subject system.

◇ **On average, 54.6% of data-access refactorings and 58% of regular refactorings are performed by a single developer.**

We observe a similar distribution of individual contributions in data-access refactorings compared to Table 9.9 which shows that in most cases the most contributing developers are also responsible for data-access refactoring activities. For example, *Michael Totschnig* from the MyExpenses subject system has the maximum development contribution (97%) and the highest data-access refactoring contribution (100%).

When we see the maximum individual contribution in Table 9.10, the average value over all the subject systems is 54.6% with a median of 48.6%. This shows that a significant proportion of data-access refactoring is performed by a single developer. On the other hand, Table 9.11 shows that the average maximum individual contribution to regular refactorings is 58% with a median of 57.32% which is higher than data-access refactorings.

◇ **The responsibility of refactoring is more shared by developers in data-access refactoring compared to regular refactoring.**

We further investigate the impact of code ownership on the refactoring contribution of developers. We expect that code owners are most likely candidates to refactor their code. We define a class owner as the developer that applied the highest number of changes to a file. For each class involved in refactoring, we determined the class owner using a git command that lists the author names of each commit that changes the target file. The author that has the maximum number of changing commits is assigned as the owner of the file.

In 19 subject systems (65.5%), more than half of the data-access refactoring operations were applied by file owners. On average, 61.6% of data-access refactorings are performed by the file owners. On the other hand, some subject systems such as *Carbon-apimgt and Direct-app* tend to share the responsibility of data-access refactoring to developers other than the file

Table 9.9 Summary of developers' contributions in the subject systems. It shows the number of developers and the summary of the percentage contributions of all developers for each subject system

Subject system	Developers	Average	std	Minimum	25%	Median	75%	Maximum
AppLozic/Applozic-Android-SDK	36	2.78	4.41	0.04	0.15	0.91	3.12	15.61
Flipkart/foxtrot	36	2.78	5.65	0.03	0.07	0.24	1.3	24.22
Hurence/logisland	38	2.63	5.53	0.04	0.18	0.45	1.76	25.97
IHTSDO/snowstorm	35	2.86	11.88	0.04	0.04	0.08	0.52	70.39
OpenSextant/Xponents	8	12.5	28.34	0.36	0.88	1.63	4.6	82.23
RyanSusana/elepy	**7**	14.29	27.04	0.06	0.14	2.31	11.74	73.86
US-CBP/GTAS	43	2.33	3.61	0.03	0.12	0.66	3.93	19.4
adempiere/adempiere	116	0.86	2.89	0.01	0.01	0.05	0.31	26.19
appirio-tech/direct-app	86	1.16	3.74	0.03	0.07	0.16	0.69	30.07
codelibs/fess	41	2.44	12.38	0.02	0.05	0.15	0.46	79.53
deegree/deegree3	56	1.79	6.01	0.01	0.03	0.09	0.99	36.25
dotCMS/core	98	1.02	2.74	0.01	0.01	0.06	0.67	18.12
eMoflon/emoflon-neo	14	7.14	11.41	0.06	0.35	1.48	10.64	41.81
eclipse-ee4j/eclipselink	111	0.9	1.71	0.01	0.01	0.1	0.92	9.58
gisaia/ARLAS-server	25	4	4.63	0.07	0.41	2.6	6.37	16.58
hazelcast/hazelcast-jet	61	1.64	4.71	0.04	0.04	0.07	0.56	26.04
hazelcast/hazelcast-simulator	49	2.04	6.84	0.02	0.02	0.05	0.19	34.74
mtotschnig/MyExpenses	37	2.7	15.93	0.01	0.01	0.03	0.07	**96.98**
neo4j-contrib/ neo4j-apoc-procedures	154	0.65	3.03	0.05	0.05	0.05	0.14	26.41
oltpbenchmark/oltpbench	53	1.89	5.85	0.09	0.09	0.18	0.81	38.56
personium/personium-core	21	4.76	7.78	0.07	0.28	0.85	6.76	31.39
pietermartin/sqlg	30	3.33	14.19	0.04	0.04	0.08	0.4	77.94
querydsl/querydsl	138	0.72	6.79	0.01	0.01	0.03	0.05	79.6
romanchyla/montysolr	11	9.09	24.52	0.06	0.15	0.3	3.98	82.46
spring-projects/ spring-data-elasticsearch	143	0.7	3	0.06	0.06	0.06	0.13	26.55
spring-projects/ spring-data-redis	134	0.75	3.71	0.04	0.04	0.04	0.07	30.52
wordpress-mobile/ WordPress-Android	211	0.47	1.42	0	0	0.02	0.13	12.4
wso2/carbon-apimgt	**349**	0.29	0.92	0	0.01	0.05	0.26	15.4
xipki/xipki	**7**	14.29	25.28	0.02	0.02	0.03	19.11	61.71

Table 9.10 Summary of the contribution of developers in data-access refactoring. The table shows the number of developers involved, the percentage against total number of developers, and the summary of the distribution of their contribution in percentage

Subject systems	**Developers**	**Developers(%)**	**Average**	**std**	**Minimum**	**25%**	**Median**	**75%**	**Maximum**
AppLozic/Applozic-Android-SDK	14	38.89	7.14	9.78	0.35	0.44	2.1	10.84	33.57
Flipkart/foxtrot	10	27.78	10	18.05	0.3	0.75	1.27	8.08	56.33
Hurence/logisland	9	23.68	11.11	17.88	0.53	2.8	5.19	7.86	57.39
IHTSDO/snowstorm	4	11.43	25	39.76	1.59	1.59	7.14	30.56	84.13
OpenSextant/Xponents	4	50	25	43.26	0.34	1.88	4.97	28.08	89.73
RyanSusana/elepy	2	28.57	50	63.98	4.76	27.38	50	72.62	95.24
US-CBP/GTAS	5	11.63	20	14.7	4	16	16	20	44
adempiere/adempiere	50	43.1	2	4.56	0.02	0.04	0.24	1.17	25.42
appirio-tech/direct-app	20	23.26	5	9.21	0.47	0.93	1.87	3.27	39.25
codelibs/fess	5	12.2	20	43.93	0.3	0.3	0.3	0.5	98.59
deegree/deegree3	11	19.64	9.09	16.56	0.24	0.48	1.92	4.92	49.88
dotCMS/core	22	22.45	4.55	5.45	0.13	0.88	2.66	5.93	21.53
eMoflon/emoflon-neo	**9**	**64.29**	**11.11**	**16.63**	**0.05**	**0.15**	**0.71**	**18.06**	**42.77**
eclipse-ee4j/eclipselink	29	26.13	3.45	6.19	0.08	0.16	0.64	3.71	26.27
gisaia/ARLAS-server	8	32	12.5	8.66	2.54	6.64	10.97	17.38	27.02
hazelcast/hazelcast-jet	29	47.54	3.45	8.45	0	0.02	0.13	1.75	34.23
hazelcast/hazelcast-simulator	22	44.9	4.55	11.46	0.01	0.03	0.07	1.91	47.31
mtotschnig/MyExpenses	**1**	**2.7**	**100**	**NaN**	**100**	**100**	**100**	**100**	**100**
neo4j-contrib/ neo4j-apoc-procedures	12	7.79	8.33	11.82	0.25	0.44	2.61	11.26	35.57
oltpbenchmark/oltpbench	18	33.96	5.56	13.09	0.15	0.29	1.67	5.23	56.69
personium/personium-core	3	14.28	33.33	16.67	16.67	25	33.33	41.67	50
pietermartin/sqlg	18	60	5.56	21.11	0.01	0.03	0.1	0.77	90.05
querydsl/querydsl	9	6.52	11.11	22.93	0.1	0.48	1.54	9.06	70.81
romanchyla/montysolr	2	18.18	50	67.2	2.48	26.24	50	73.76	97.52
spring-projects/ spring-data-elasticsearch	25	17.48	4	14.3	0.07	0.14	0.43	1.3	72.27
spring-projects/spring-data-redis	13	9.7	7.69	13.64	0.03	0.1	0.27	4.4	36.61
wordpress-mobile/ WordPress-Android	27	12.8	3.7	7.73	0.12	0.36	0.61	2.48	33.9
wso2/carbon-apimgt	77	22.06	1.3	1.87	0.07	0.2	0.6	1.66	9.51
xipki/xipki	2	28.57	50	4.44	46.86	48.43	50	51.57	53.14

Table 9.11 Summary of the contribution of developers in regular refactoring. The table shows the number of developers involved, the percentage against the total number of developers, and the summary of the distribution of their contribution in percentage

Subject systems	Developers	Developers(%)	Average	std	Minimum	25%	Median	75%	Maximum
AppLozic/Applozic-Android-SDK	21	58.33	4.76	7.66	0.05	0.49	1.51	5.89	29.95
Flipkart/foxtrot	17	47.22	5.88	11.23	0.03	0.06	0.14	3.11	34.91
Hurence/logisland	22	57.89	4.55	10.1	0.02	0.1	0.65	3.12	37.64
IHTSDO/snowstorm	15	42.86	6.67	17.18	0.01	0.03	0.27	2.42	66.19
OpenSextant/Xponents	**6**	**75**	**16.67**	**37.67**	**0.06**	**0.91**	**1.41**	**2.48**	**93.54**
RyanSusana/elepy	3	42.86	33.33	46.61	0.05	6.7	13.34	49.97	86.6
US-CBP/GTAS	20	46.51	5	8.51	0.04	1.62	3.1	4.41	39.65
adempiere/adempiere	60	51.72	1.67	12.74	0	0	0	0.01	98.67
appirio-tech/direct-app	36	41.86	2.78	4.1	0.08	0.23	0.98	3.89	16.67
codelibs/fess	18	43.9	5.56	21.35	0	0.01	0.06	0.37	90.99
deegree/deegree3	34	60.71	2.94	8.34	0	0.01	0.12	0.62	39.51
dotCMS/core	49	50	2.04	8.04	0	0.01	0.17	0.76	55.8
eMoflon/emoflon-neo	8	57.14	12.5	19.8	0.1	0.27	8.41	11.34	59.87
eclipse-ee4j/eclipselink	78	70.27	1.28	3.52	0	0.01	0.17	0.98	22.68
gisaia/ARLAS-server	12	48	8.33	8.95	0.03	1.51	6.35	13.05	29.47
hazelcast/hazelcast-jet	**2**	**3.28**	**50**	**66.15**	**3.23**	**26.61**	**50**	**73.39**	**96.77**
mtotschnig/MyExpenses	3	8.11	33.33	57.23	0.01	0.29	0.57	49.99	99.42
neo4j-contrib/neo4j-apoc-procedures	75	48.7	1.33	5.06	0.02	0.04	0.08	0.26	35.77
oltpbenchmark/oltpbench	16	30.19	6.25	15.83	0.06	0.32	1.23	4.67	64.81
personium/personium-core	12	57.14	8.33	15.96	0.07	0.4	1.02	8.11	55.97
pietermartin/sqlg	9	30	11.11	27.28	0.07	0.33	0.65	0.85	82.94
querydsl/querydsl	57	41.3	1.75	12.23	0	0.01	0.01	0.05	92.41
romanchyla/montysolr	4	36.36	25	28.95	0.02	1.29	20.66	44.37	58.66
spring-projects/ spring-data-elasticsearch	42	29.37	2.38	11.6	0.01	0.02	0.11	0.62	75.29
spring-projects/spring-data-redis	34	25.37	2.94	8.89	0	0	0.01	0.07	45.28
wordpress-mobile/ WordPress-Android	113	53.55	0.88	2.75	0	0.01	0.05	0.38	24.11
wso2/carbon-apimgt	184	52.72	0.54	1.74	0	0.03	0.14	0.43	20.36
xipki/xipki	3	42.86	33.33	35.29	0.01	14.85	29.69	50	70.3

owners.

We observe a roughly similar distribution when we see the regular refactoring operations performed by the file owners. In 21 subject systems (72.4%), more than half of the regular refactoring was performed by the owners of the target class. Among the subject systems, *Adempiere* have the lowest percentage of 0.71% which shows that regular refactorings are not performed by the class owner but rather involve many developers in the context of the subject system. The average percentage of regular refactoring performed by the target class owner over all the subject systems (66.71%) is higher than that of data-access refactorings. This shows that the responsibility of data-access refactoring is more shared by developers compared to regular refactorings.

Table 9.12 Percentage of data-access and regular refactoring performed by the main contributor of the involved classes. The subject systems are sorted in alphabetical order.

Subject Systems	**Refactored by main contributor (%)**	
	DAC	**Regular**
AppLozic/Applozic-Android-SDK	33.916	42.108
Flipkart/foxtrot	55.44	41.229
Hurence/logisland	22.237	85.46
IHTSDO/snowstorm	84.127	76.736
OpenSextant/Xponents	89.726	93.103
RyanSusana/elepy	**100**	92.65
US-CBP/GTAS	52	58.418
adempiere/adempiere	37.743	**0.711**
appirio-tech/direct-app	22.43	48.983
codelibs/fess	98.593	90.977
deegree/deegree3	76.259	74.95
dotCMS/core	42.153	23.627
eMoflon/emoflon-neo	54.642	73.687
eclipse-ee4j/eclipselink	36.18	56.611
gisaia/ARLAS-server	55.196	61.056
hazelcast/hazelcast-jet	48.619	96.774
hazelcast/hazelcast-simulator	74.723	**99.415**
mtotschnig/MyExpenses	**100**	56.434
neo4j-contrib/neo4j-apoc-procedures	64.428	71.429
oltpbenchmark/oltpbench	66.279	26.645
personium/personium-core	33.33	26.645
pietermartin/sqlg	90.181	79.036
querydsl/querydsl	68.69	92.724
romanchyla/montysolr	94.215	87.366
spring-projects/spring-data-elasticsearch	72.122	78.929
spring-projects/spring-data-redis	43.041	63.798
wordpress-mobile/WordPress-Android	68.523	57.746
wso2/carbon-apimgt	**9.315**	41.182
xipki/xipki	91.574	96.127

9.7 Discussion

Previous research has shown that rename refactorings are dominant refactorings in traditional software systems (Example: [23, 152]). While rename refactorings are among the top five prevalent data-access refactorings, the most prevalent data-access refactoring is *Change Parameter Type* followed by *Change Variable Type.* Most of such refactoring types are often applied when developers change APIs and data types as part of adding or modifying features or during bug fixing activities. The prevalent refactorings are mostly low-level refactorings such as Rename method, variable, and parameter, change return type, and change parameter type which happens either at the statement level or method level. The reason for this could be their simplicity to apply, strong tool support for such refactorings, or the lack of tool support to perform complex refactorings that involve multiple classes or packages.

Given the subject systems represent data-intensive systems, it is expected that data-access classes should get more refactoring attention compared to regular classes. However, the results show roughly similar refactoring density between data-access and regular classes. This might indicate that developers give little consideration to the domain importance when choosing refactoring candidates.

Except for a few refactoring type pairs, composite refactoring is not practiced in data-access classes. However, few refactoring pairs have a strong association. For instance, *Change parameter type & Rename parameter* have the highest association, which is not surprising as developers perform rename to make sure the variable name reflects changes in the data type of variable or parameter. This finding is similar to traditional software systems. Peruma et al. [26] showed that rename refactorings have a tendency to co-occur with *Change Variable Type* refactorings in traditional software systems. They showed that 17.39% of 310,309 rename refactoring instances were associated with a change in variable type.

On average, 23% of the developers contributed to data-access refactoring and 46.8% contributed to regular refactoring. On average, 43.3% of data-access refactorings and 52.9% of regular refactorings are performed by a single developer. Data-access refactorings tend to involve more developers besides the main contributor to the target classes compared to regular refactorings

Regarding developers involved in refactoring, our findings show that not all developers perform refactoring and the number of developers involved in data-access refactoring is smaller than that of regular refactorings. Furthermore, most of the data-access refactoring is dominantly performed by a single developer. The data-access classes have a higher code size compared to regular classes in our subject systems. Our results also show that the overall

contribution of developers is not identical as demonstrated in Table 9.9 and most of the developers that have higher overall contributions also have higher refactoring contributions. We investigated the proportion of refactorings performed by the file owner (i.e., the developer that applied the highest number of commits that modified the target file) and found that a large portion of both data-access and regular refactorings are applied by file owners. Vassallo et al. [22] have similar findings for traditional software systems. Hence, our finding generalizes their finding to the context of data-access code and data-intensive systems.

9.8 Threats to validity

In this section, we present threats to validity associated with the quantitative study of refactoring practices in data-intensive systems (RQ 4.1, RQ 4.2, RQ 4.3, RQ 4.4 and RQ 4.5).

9.8.1 Threats to construct validity

We relied on state-of-the-art refactoring detection and SQL code smell detection tools to extract refactoring instances, SQL queries, and SQL code smells. Refactoring Miner is reported to achieve 99% precision and 94% recall [97]. However, we could still miss some refactoring instances in which the tool might introduce false positives. The SQL Inspect (precision $>$ 88% and recall $>$ 71.5% [33]) tool used to identify SQL queries and SQL code smells could also miss some queries and smells. Hence, the interpretation of our findings should take this into account. We also relied on import statements to identify data-access classes in NoSQL-based subject systems. There could be some cases where the import statements are not actually utilized in the code, leading to false data-access classes. However, we did not encounter any case when we perform the manual analysis in RQ 4.6 for NoSQL-based subject systems.

9.8.2 Threats to internal validity

The subject systems we selected may not represent data-intensive systems as they are obtained from open-source projects. To mitigate this threat, we applied a rigorous filtering of applications. For NoSQL subject systems, we relied on the number of SQL queries to rank the subject systems and considered the systems with the highest number of queries. Similarly, we ranked NoSQL subject systems based on the number of data-access classes and selected the top projects.

9.8.3 Threats to conclusion validity

This threat is associated with the choice of statistical tests. We used a non-parametric test in this study. While non-parametric tests are more general than parametric tests, they have lower statistical power. However, we did not claim a causal relationship between the variables as we measured the association between them.

9.8.4 Threats to external validity

Since we performed a longitudinal study, we have to limit our subject systems to 29. This could limit the external validity of our study. However, we carefully selected our subject systems to represent data-intensive systems by considering the number of SQL queries and number of data-access classes as a proxy. Furthermore, those systems come from different application domains and rely on different data-access technologies including JDBC, SQLite, and Hibernate. Hence, our findings can be generalized to the extent of open source data-intensive systems.

9.8.5 Threats to reliability validity

To minimize potential threats to reliability, all our subject systems are open source and available on GitHub. Furthermore, we provided all the necessary materials to replicate our study in our replication package [146].

9.9 Chapter summary

In this chapter, we analyzed the prevalence of refactoring activities between data-access classes and regular classes to investigate if developers perform different types of refactoring between them. We also analyzed the distribution of refactoring instances over the evolution of the subject systems to investigate if the refactoring applied by developers varies between data-access classes and regular classes and varies across systems' evolution. Third, we investigated if data-access refactoring activities touch SQL queries and SQL smells. Fourth, we investigated the co-occurrence of different refactoring types. Lastly, we compared the profile of developers involved in data-access refactoring with the ones that are not involved in data-access refactoring.

Our results show that different types of refactoring instances are prevalent in data-access classes and regular classes. The most prevalent refactoring is *Change Parameter Type* in data-access classes and *Move attribute* in regular classes, respectively. Furthermore, data-access

refactoring tends to be applied at later stages of the subject system's evolution compared to regular refactoring. We also find that data-access refactoring instances do not generally touch both SQL queries and SQL code smells. We also found that *Refactoring contribution metric* was the most separating metric between developers involved in data-access refactoring and regular refactorings.

CHAPTER 10 QUALITATIVE ANALYSIS OF DATA-ACCESS REFACTORINGS

10.1 Chapter overview

In this chapter, we present the qualitative analysis and findings regarding functionality of code elements prone to refactoring (RQ 4.6), context of data-access refactoring (RQ 4.7), and developer's opinion about refactoring practices (RQ 4.8). We utilized the **refactoring dataset** and **survey on data-access refactorings** as a data source for the qualitative analysis. As part of achieving research objective 4, we answer the following research questions.

RQ 4.6: **What do code elements targeted by data access refactorings implement?**

> **Main finding:** Data-access functionality is associated with many refactoring instances, accounting for 38.27% of the analyzed data-access refactorings. Among data-access functionalities, fetching data takes the largest share, accounting for 23.46% of the analyzed data-access refactorings followed by inserting data. Different refactoring types are prevalent in data-access code implementing data fetching and data insertion.

RQ 4.7: **What is the context in which data access refactoring occur?**

> **Main finding:** Large fractions of data-access refactoring commits are not pure refactorings. Only 23.8% are pure refactorings. The remaining refactorings are done together with other software activities such as bug fixing or changing feature. Changing feature development activity has the highest co-occurrence with data-access refactorings.

RQ 4.8: **What is developers' opinion about refactoring practices in data-access classes?**

> **Main finding:** The most common motivation to apply data-access refactoring was improving code readability, followed by improving data-access performance. Coding experience, refactoring contribution, and familiarity with the target code were considered as important factors to assign developers to data-access refactoring. Large number of respondents support that data-access refactorings are often applied during changing a feature.

10.2 RQ 4.6: Data-access class code elements prone to refactorings

In this section, we present the result of our manual analysis on sample data-access refactorings to identify functionalities of code elements that are prone to refactoring.

In **RQ 4.1**, we have shown that refactoring activities are prevalent in data-access classes. However, not all components of data-access classes are directly associated with data-access. It is expected that data-access classes could also contain constructors, accessors, mutators, and non data-access logic implementations. Hence, it is necessary to investigate if data-access refactorings focus on the actual data-access logic or other non data-access functionalities.

10.2.1 Analysis approach

To identify functionalities of code artifacts associated with refactoring in data-access classes, we randomly sampled 500 data-access refactoring instances (Confidence interval= 4.28 and Confidence level= 95%) and manually analyzed the code associated with each refactoring. Out of the 500 analyzed samples, 10 were false positives. In addition, we were not able to assign functionality to 4 instances since they were associated with empty methods and their method name is not descriptive enough. Finally, we remained with 486 refactoring instances. We first describe each of the identified functionalities and then discuss the prevalence of the functionalities using absolute numbers and percentages against the total sample.

10.2.2 Findings

The results show that:

◇ **Data-access functionalities were the most prevalent functionalities implemented by refactored codes in data-access classes.**

Figure 10.1 shows the identified functionalities categorized by seven higher-level categories. The number of refactoring instances is indicated for the categories as well as each functionality. We will describe the categories in the following paragraphs. ◇ *Data-access:* Refactoring instances that target code elements that perform read and write operations on the data stored in databases are categorized under this category. We categorized code elements that contain one or more data manipulation queries (Eg. SELECT, INSERT, UPDATE and DELETE) and code elements that are associated with managing transactions under data-access. Numerous refactoring instances (38.27%) are associated with data-access. In particular, *Fetch data*, a functionality associated with reading some data from a database is the most prevalent (23.46%) followed by *Insert data* (9.67%) and *Update data* (2.67%). One refactoring instance was associated with *Manage Transaction* and more specifically rolling back a transaction.

◇ *Initialize fields and components:* Refactoring instances that target code elements that initialize or set data fields, utility fields, or user interface (UI) components are categorized under this functionality. Data fields are variables that represent a database entity and whose values are used to capture data from a database or to populate a database. On the other hand, utility fields are used to capture non-data-access related entities. The *Initialize fields and components* functionality is associated with 19.13% of the refactoring instances. The *Store data element & Class constructor* functionalities associated with initializing data fields are the most prevalent functionalities associated with 7.2% refactoring instances each and followed by *Initialize UI element* (1.85%), associated with initializing graphical user interface components with data obtained from a database or as a result of some intermediate transformation of data. *Data model*, a functionality associated with providing an abstraction of database entities, is associated with 1.23% of the refactoring instances. The remaining functionalities i.e., *Initialize utility fields, data model (define the schema of a database entities) and function parameter (storing passed value as function parameters) make 2.88% of refactoring instances.*

◇ *Helper:* This category regroups refactoring instances targeting code elements that are involved in the implementation of business logic that is not directly associated with database access, such as parsing objects (access elements of complex objects), validation of user input, and handling UI events. The *Helper* functionality is associated with 18.72% of refactoring instances. *Implement non data-access business logic* is the most prevalent functionality in this group. It is associated with 9.87% refactoring instances, followed by *Field accessor*(get the value of a field) which is associated with 5.14% instances. *Input validation* and *Handle UI event* functionalities are less prevalent; each are associated with only 1 instance.

◇ *Manage query and result set*: Refactoring instances that target code elements associated with manipulating the query, parsing, and transforming the query result set are grouped un-

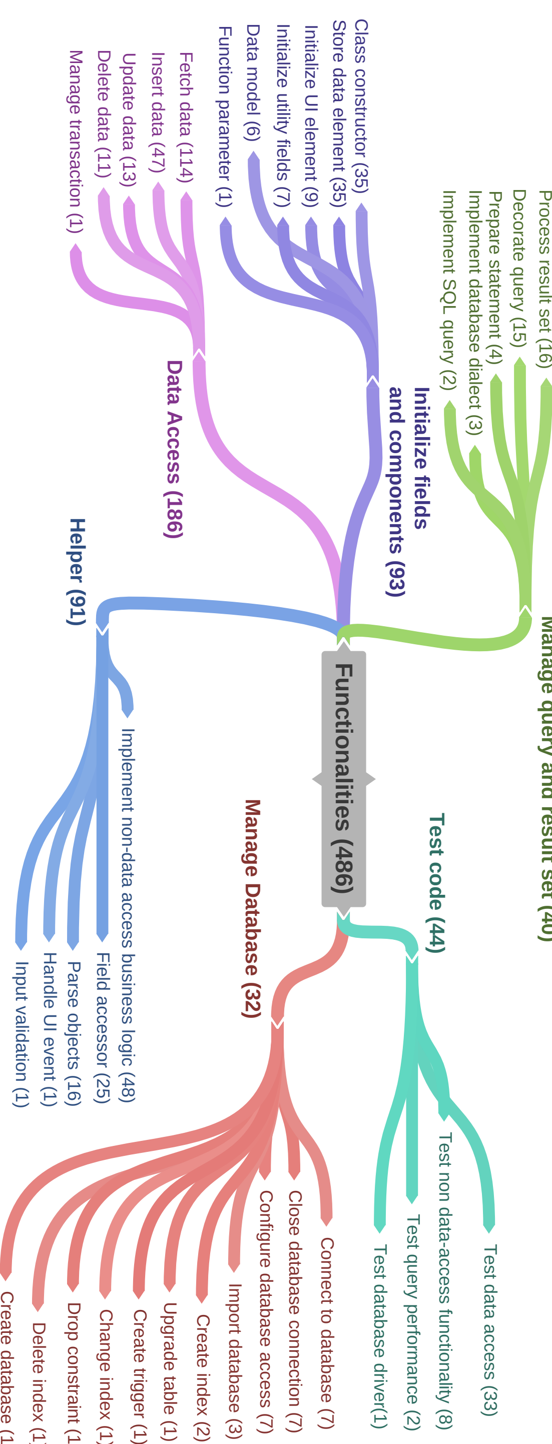


Figure 10.1 Functionalities of code artifacts associated with refactoring in data-access classes. The sub-categories are ordered from the most prevalent to the least prevalent.

der this category. This group accounts for 8.23% of refactoring instances. Specifically, *Process result set*, associated with parsing and iterating over query results, takes the largest share (3.29%) followed by *Decorate query* (3.09%), functionality associated with pre-processing a query before passing to the database. *Prepare statement*, associated with parameterizing queries, *Implement database dialect*, associated with defining particular features of the SQL query available when accessing a data entity, and *Implement SQL query*, associated with abstracting SQL data-access over data sources that do not directly support SQL queries, account for 1.85% in total.

◇ *Manage database*: We categorize refactoring instances that target code elements involved in connecting with database, managing the index, managing constraints, managing triggers, and importing the database under this group. The *Manage database* functionality is associated with 6.58% refactoring instances. *Close database connection* functionality, associated with terminating a database connection and releasing resources, *Configure database access*, associated with setting the connection parameters and credentials, and *Connect to database* functionalities are the most prevalent functionalities with each taking 1.44%. The *Import database* functionality is associated with creating a database with a template or data obtained from external files, and accounts for 0.61% of the refactoring instances. *Create index, Delete index, change index* functionalities are focused on managing indexes and account for 0.61% of the refactoring instances. The remaining functionalities (*Upgrade table, drop constraint, create trigger and create database* are associated with one refactoring instance each.)

◇ *Test code*: Refactoring instances that target code elements involved in testing production code elements are categorized under *Test code*. The *Test code* functionality is associated with 9.05% of the refactoring instances. The most common functionality in *Test code* is testing production code involved in database read and write operations, known as *Test data-access*. It accounts for 6.79% of refactoring instances. On the other hand, the *Test non data-access functionality*, associated with testing production code that is not directly associated with database access is associated with 8 (1.64%) refactoring instance. Code elements involved with testing the performance of a certain query are categorized as *Test query performance*, and they are associated with only two refactoring instances. Code elements that specifically test database drivers are categorized as *Test database driver*.

◇ *Rename Variable* **refactoring is the most prevalent refactoring applied on *data fetching* code while** *Add Parameter Modifier* **is the most frequent refactoring applied on *data insertion* code.**

Given that Fetch Data and Insert data are the most prevalent data-access functionalities,

accounting to 33.12% combined, it is interesting to study the prevalence of refactoring types associated with them as it shows what type of refactorings are frequently applied in a code that implements the aforementioned data-access functionalities. Table 10.1 shows the top ten prevalent refactoring types associated with Fetch data and Insert data functionalities. The percentage in the table was computed against the total number of refactoring instances associated with Fetch data (114) and insert data (47) functionalities. *Rename Variable* and *Add Parameter* refactorings are the most frequent refactorings (21.93% combined) applied to data-access code that implements data fetching, followed by *Change Variable Type*. This shows that in most cases, refactorings that change the name or type of variable are applied to data fetching code. On the other hand, *Add Parameter Modifier and Extract Variable* are the most prevalent refactorings applied to data-access codes that implement Insert data functionality. *Change Thrown Exception Type* is the third most prevalent refactoring associated with Insert data. Most of the refactoring types associated with code that implements insert data functionality focus on improving code comprehension (*Extract Variable*) and protecting input parameters from modification or overriding using *Add Parameter Modifier (Eg. final modifier)*. A significant number of refactorings associated with insert data functionality aim at changing the type of exception thrown by the target method.

Table 10.1 Most prevalent refactoring types for Fetch data and Insert data functionalities.

Fetch data Functionality			Insert data Functionality		
Refactoring Type	**count**	**percentage**	**Refactoring Type**	**count**	**percentage**
Rename Variable	13	11.404	**Add Parameter Modifier**	7	14.894
Add Parameter	12	10.526	Extract Variable	5	10.638
Change Variable Type	9	7.895	Change Thrown Exception Type	4	8.511
Change Return Type	8	7.018	Add Parameter	3	6.383
Remove Parameter	7	6.14	Rename Method	3	6.383
Extract Method	7	6.14	Add Variable Modifier	3	6.383
Add Parameter Modifier	6	5.263	Change Parameter Type	3	6.383
Rename Parameter	5	4.386	Parameterize Variable	2	4.255
Change Parameter Type	5	4.386	Move And Inline Method	2	4.255
Change Thrown Exception Type	5	4.386	Rename Parameter	2	4.255

10.3 RQ 4.7: Context of data-access refactorings

In this section, we present our analysis and findings regarding the contexts in which data-access refactorings occur. We manually analyzed a sample data-access refactoring commit messages to classify the purpose of the commits.

Developers could perform only refactoring activities in a commit or perform refactoring as part of other activities such as bug fixing, changing features, or adding new features. An-

swering this research question helps to understand if refactorings are applied for the sole purpose of improving the code or as part of addressing user requirements.

10.3.1 Analysis approach

From data-access refactoring commits, we randomly sampled 500 commits (Confidence interval= 4.24 and Confidence level= 95%) and manually investigated the commit message and associated changes to determine if the purpose of the commit is only for refactoring or if it is also associated with other activities. In particular, we assigned the purpose of the commit as one of the following: *Refactoring, adding a new feature, changing feature, merging, bug fixing or multi-purpose.* We followed a similar approach with RQ 4.6 to validate the labels and minimize researcher bias.

Refactoring: We assign a commit whose commit message and set of changes indicate only refactoring changes as refactoring. Such commit has the sole purpose of applying refactoring and is not associated with other development activities.

For example, the commit message *"TPCC formal code refactoring completed... now it is time to debug..."*[1] in the Oltpbench project is refactoring commit.

Adding new feature: We assign a commit that implements new features in addition to refactoring under this category. For example, the commit message *"#608105 :added batch support for SQLDeleteClause and SQLUpdateClause"*[2] in Querydsl project clearly shows that this commit implements new feature besides refactoring.

Changing feature: We assign the purpose of commits that update existing implementation, besides refactoring, by enhancing or reversing already implemented features as Change feature.

The commit message *"try safer approach for updating planinstance_transaction, the previous one lead to SQLiteConstraintException"*[3] in MyExpenses project performs changing existing features besides refactoring.

Merging: We assign all merging commits under this category. For example, *" Merge change from branches/adempiere341, revision 6036-6040"* [4] in Adempiere project is assigned merging.

Bug fixing: We assign commits that perform bug fixing besides refactoring under this

[1] https://bit.ly/3EzrZ0P
[2] https://bit.ly/3z7dXCg
[3] https://bit.ly/3Jhq0Si
[4] https://bit.ly/3qyWf75

category. We identify commits whose commit message indicates bug fixing as bug fixing commits.

For example, the commit message *"Fixed bug in ReaderTagTable that caused tbl_tag_updates to always overwrite the existing row when updating a date column"* [5] in WordPress-Android project.

Multi-purpose: Some commits involve refactoring and any of adding new feature, changing feature or bug fixing together. We assign the label of such commits as Multi-purpose.

The commit message *"Loads of changes: - Updates to the Dashboard interface - New WP-TitleBar class to re-use for each view that uses the action bar interface - Corrected the app blog ID (not wp blog id) to be an int instead of a string - Bug fixes"* [6] in WordPress-Android project is a good example for this category as it contains refactoring, bug fixing, adding new feature and feature change.

10.3.2 Findings

The results show that:

◇ **Only 23.8% of the analyzed commits are pure refactoring.**

Figure 10.2 shows the proportion of each category among the analyzed samples. 74.6% of the data-access refactoring commits applied refactoring together with other activities such as adding new feature, bug fixing, changing feature, and a combination of those. Only 23.8% of the commits are pure refactoring. The remaining 1.6% are merging commits. This shows that in most cases refactorings are applied together with other changes rather than for the sole purpose of refactoring.

◇ **Changing feature is the most co-occurring activity with data-access refactoring.**

25.8% of the analyzed data-access refactoring commits involved changing or updating an existing feature. Bug fixing (21%) and adding new feature (17.4%) activities have the next higher co-occurrence with data-access refactoring activities.

10.4 RQ 4.8: Developers' opinion about refactoring practices in data-access classes

In this section, we report our analysis and findings from the data-access refactoring survey. We first report the demographics of participants and then proceed to the main survey

[5] https://bit.ly/3ewi3dU
[6] https://bit.ly/3HhviLN

Figure 10.2 Bar plot showing the proportion of each label in the analyzed samples in percentage.

questions.

In this Chapter and Chapter 9, we presented different findings about the motivation, timing associated with data-access refactoring and if data-access smells are addressed during refactoring. We relied on empirical analysis of our subject systems to obtain the findings. The aim of this research question is to complement the previous findings with the opinions and experiences of software developers involved with data-access code development and refactoring.

10.4.1 Analysis approach

We analyzed the responses of the **survey on refactoring practices**. The recruitment procedure and the content of the survey were described in Section 4.5.10. We obtained **20** complete responses after running the survey for one month. We extracted all survey responses from Google forms and download them as a CSV file. We separated the downloaded file based on the survey sections for easier analysis. Next, we present the findings in the next sub-section.

10.4.2 Findings

We first report the demographics of participants and discuss the responses separating into the following sections: *Data-access refactoring and releases, Data-access refactoring context, Motivation behind refactoring data-access classes, and Developers assignment to refactoring.*

Demographics of participants

We asked about the organizational roles, software development experience, and how often they apply refactoring in general and data-access refactoring in particular to assess the demographic distribution of the survey participants.

◇ **Most survey participants are software developers with experiences ranging from one to five years to more than ten years.**

When we see the organizational role of the survey participants, most of them have software developer roles including development and testing, i.e., 15 participants (75%), and the remaining 5 participants mentioned their roles as data scientists and data engineers. Considering the software development experience of the survey participants, 11 participants (55%) have between one and five years of development experience. Six participants (30%) have more than ten years of experience. The remaining three (15%) have between 5 and 10 years of software development experience.

◇ **Most survey participants frequently perform refactoring and data-access refactoring activities.**

We also asked the survey participants how often they perform refactoring in general and data-access refactoring in particular. Nine participants mentioned that they often perform refactorings, while nine participants mentioned that they sometimes perform refactorings. In the extreme case, two participants mentioned that they always perform refactorings. Considering the specific case of data-access refactoring, 14 respondents (70%) mentioned that they perform data-access refactoring more frequently (sometimes and often) while six participants mentioned that they rarely perform data-access refactorings.

Data-access refactoring and releases

In Chapter 9, we found that data-access refactorings have similar distribution regardless of how far or how close they are applied to official releases, while regular refactorings are often applied shortly after releases (Section 9.3). We asked the survey participants if the refactoring decisions they made are affected by release deadlines.

◇ **Survey respondents have a mixed opinion about the distribution of refactorings relative to release points with no clear winner.**

Figure 10.3 shows the survey respondents' agreement to various statements describing when refactorings in general and data-access refactorings in particular are applied relative to releases. The statement that refactorings are not affected by release deadline was not supported by nine participants. Similarly, eight participants did not support the statement that data-access refactorings are not affected by release deadlines. On the other hand, eight respondents supported the statement that refactorings are not affected by release deadlines and seven agreed with the statement that data-access refactorings are not affected by release deadlines. This shows that neither of the statements have strong support. For refactorings in general and data-access refactorings in particular, the statement that refactorings are made shortly after releases got very small support (three) from participants.

We also asked the justification behind the respondent's answer to when refactorings are applied, and received 11 responses. Two respondents mentioned that since data-access is a critical part of data-intensive applications, it is a good idea to apply data-access refactorings before release. On the other hand, one participant highlighted that refactorings could only be applied before releases when there is enough time for regression testing. Another respondent also mentioned that major refactoring before releases would be risky. Two respondents mention that the decision when to refactor depends on the goal of the refactoring (code smell removal, part of bug fixing), the criticality of the target code varies from project to project. This shows that other confounding factors such as code criticality, refactoring goal, and complexity of the target code plays a significant role on the timing of refactoring activities, besides release deadlines.

Data-access refactoring context

In RQ 4.7 We showed that large portions of data-access refactoring commits are not pure refactorings. They are often associated with other software development activities such us changing features. We asked the survey respondents in what context do they often apply data-access refactorings.

◇ **Large number of respondents support that data-access refactorings are often applied when changing a feature.**

Figure 10.4 shows the distribution of the survey responses with respect to the context in which data-access refactorings are applied. Most of the respondents agree that data-access refactorings are performed when changing a feature (16 participants, 80%), followed by im-

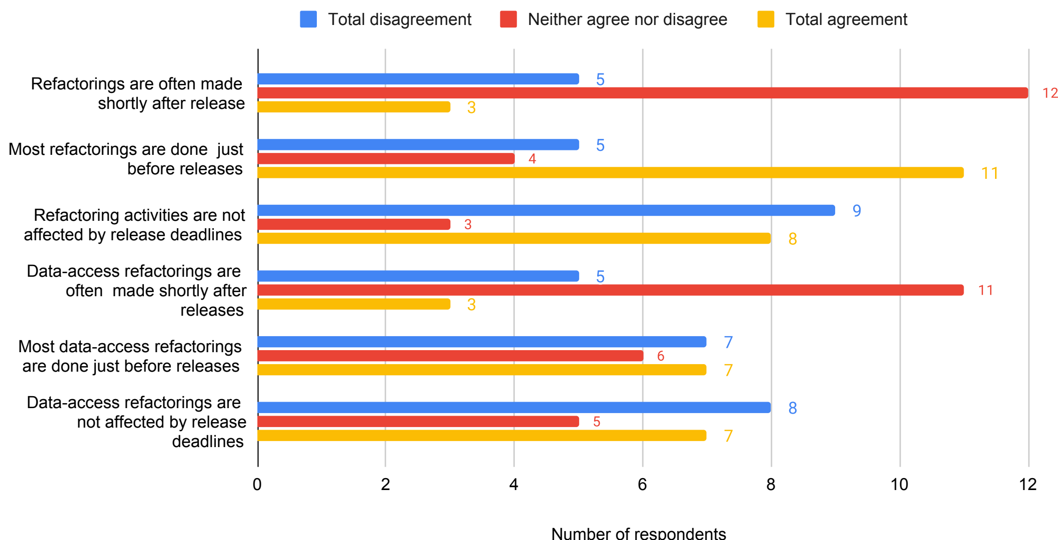


Figure 10.3 Survey respondent's opinion on refactoring frequency and release time. Total agreement is obtained by summing the *strongly agree* and *agree* responses. Similarly, total disagreement is obtained by summing *strongly disagree* and *disagree*.

provement to code quality with 14 supports (70%). Bug fixing and adding new feature obtained a support of 13 responses each (65%). This result is in alignment with our finding in RQ5 that data-access refactoring is mostly applied when changing feature, followed by pure refactoring to improve code quality.

We also noticed that a significant number of respondents neither agreed nor disagreed with the statement that data-access refactorings are applied during bug fixing (6) or only to improve code quality (5). We asked the respondents the reasons behind their decision to apply a refactoring, and most of the responses indicate that data-access refactorings are often applied together with any of the contexts.

One respondent mentioned that data-access refactorings rarely occur with adding new feature. Similarly, another respondent mentioned that *"When adding a new feature, my focus primarily remains on getting the feature work. So generally don't think of refactoring during that time"*. This is in line with our finding in RQ 4.7 that adding a new feature has lower association with data-access refactoring, compared to changing features or improving code quality.

We also asked the respondents to mention other contexts in which data-access refactoring could be applied. Some respondents mentioned that they apply data-access refactoring during migration between databases, when migrating the code to a newer version, to increase code

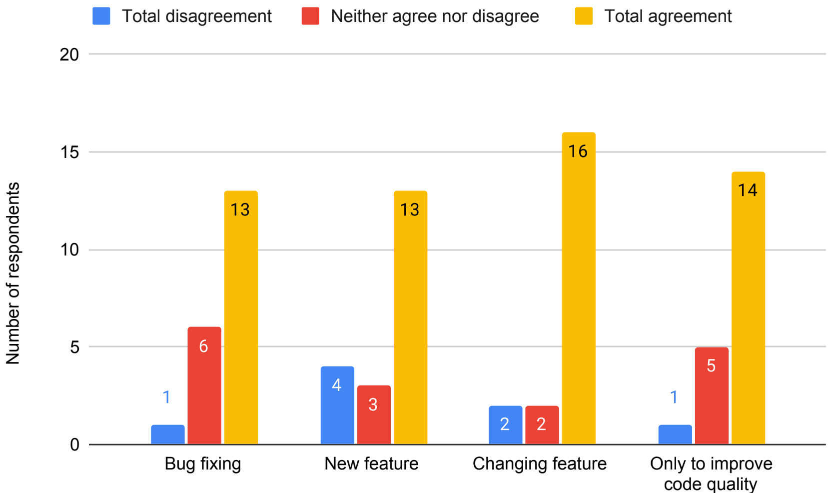


Figure 10.4 Respondents' opinion on the context during which data-access refactoring is applied. Total agreement is obtained by summing *strongly agree* response and *agree* response. Similarly, total disagreement is obtained by summing *strongly disagree* and *disagree*.

coverage, and for performance optimization.

Motivation behind refactoring data-access classes

We asked our survey participants what is the motivation behind applying data-access refactorings. We provided them a list of common refactoring motivations that are *to improve data-access performance, to fix bad code smells, improve code readability and improve maintainability.* We also provided a place for the respondents to add other motivations.

◇ **The most popular motivation behind applying data-access refactoring is improving code readability followed by improving data-access performance.**

Figure 10.5 shows the popularity of the data-access refactoring motivations. The most popular motivation is *improve code readability* with 18 votes (90%) followed by *improve data-access performance* with 17 votes (85%) and *fix bad code smells* also with 17 votes (85%). *Improve maintainability* obtained 15 votes (75%). The motivations *to reduce complexity, use a newer/better API and Improve testability* were suggested by the survey participants.

We also asked the survey participants for a justification behind the refactoring motivations they voted. One respondent highlighted the importance of improving code readability promotes collaboration. Another respondent mentioned that *"data-access refactorings are performed only when there is a performance issue"*.

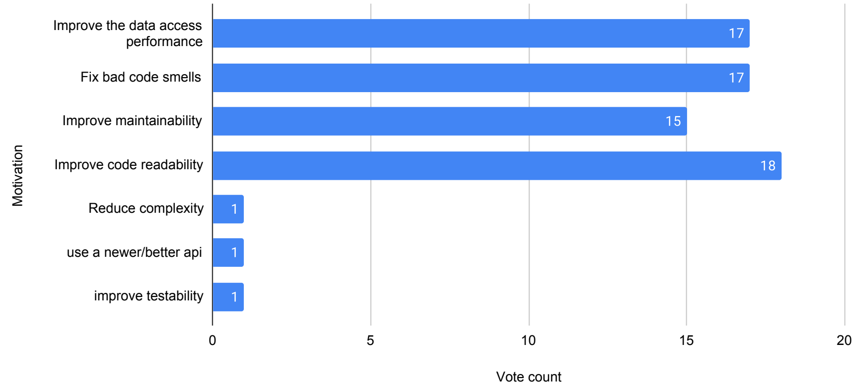


Figure 10.5 Motivations behind data-access refactoring with the number and proportion of endorsement by the survey respondents.

Developers assignment to refactoring

In Chapter 9, we found that among the developer profile metrics, *Refactoring contribution* metric was the most distinguishing metric between refactoring data-access and regular classes, and data-access refactorings are dominated by developers that own the target code. To investigate if these obtained results match the reality of data-access refactoring assignment tasks, we asked our survey participants to rate the importance of the profile metrics.

⋄ **The coding experience, refactoring contribution, and familiarity of developers with the target code was considered important factors by the survey respondents when assigning a data-access refactoring task to the developers.**

16 respondents gave a rating above three for *the coding experience of the developer* followed by *refactoring contribution and experience of the developer* (15 respondents). The *ownership or familiarity of the developer to the target code* and *developer's availability* each rated above three by 12 respondents. This shows that the coding experience and refactoring contribution of developers are the main factors considered by respondents to assign data-access refactoring tasks. However, seven respondents rate five for the factor *ownership or familiarity of the developer to the target code,* which implies that it is still a significant factor to consider.

We also asked about the justification behind the rating of the aforementioned factors. Some respondents highlighted that refactoring needs to be supervised (under code review) which mitigates the issues associated with assigning developers with lower experience. In addition, the respondents mention that it is useful to involve developers with average experience in refactoring to improve their skills and code familiarity. The respondents also pointed out that familiarity with the code reduces the refactoring time and effort. However, assigning

developers that are not familiar with the code will give an advantage of fresh pair of eyes to identify some issues or to have another perspective on the refactoring tasks.

10.5 Discussion

We found 6 high-level functionalities and 36 low-level functionalities of the code elements targeted by data-access refactorings. The dominant functionality is data-access, involving reading and writing on databases. Furthermore, fetching data is more frequent compared to insert or update operations. This finding is important as it shows that not all data-access operations are equally important in data-intensive systems. On one hand, this helps developers to prioritize refactorings based on domain importance. This has the potential to better manage their refactoring effort by focusing on code elements that have more contribute to improving the overall performance of data-intensive systems. On the other hand, Most of the SQL code smells affect data fetching performance, which shows that addressing such smells might have a higher contribution to improving the systems.

Our findings show that only 23% of the data-access refactorings are pure refactorings. The remaining refactorings are floss refactorings as they are performed as part of other development activities such as bug fixing, adding a new feature, and changing features. Hence, addressing requirement changes is the main motivation behind data-access refactorings rather than improving software quality attributes such as maintainability and program comprehension. The findings of the developer survey also support that refactorings are mostly applied during changing features. This finding supports the common wisdom on the motivation behind refactoring. Multiple studies show that changes in requirements are the main motivations behind refactoring in traditional software systems [22, 111, 115]. Our findings that changing feature has the highest association with data-access refactoring is similar to the finding of [22]. Hence, our result generalizes the findings of previous research to the context of data-access classes.

Such findings imply that improvement of software quality by refactoring has less priority compared to addressing changes in requirements in software systems in general and data-intensive systems in particular. One of the reasons could be that addressing requirements have more immediate benefits for developers rather than maintaining software quality, which usually has long-term benefits by reducing the impacts of technical debts. Hence, more research work is required to educate all stakeholders about the long-term impacts of failure to apply regular software maintenance operations such as refactoring and to recommend ways to give more emphasis on incorporating software quality as an important factor for software acceptance.

We find that the most popular motivation to apply data-access refactorings is to improve code readability, followed by improving data-access performance and fixing bad code smells. Improving code readability and fixing bad code smells is also common motivation to refactor traditional software. It is not surprising that improving data-access performance was the most considered motivation as the overall performance of data-intensive systems is affected by the performance of data-access operations and performing refactoring to improve performance will have an immediate benefit and business value.

10.6 Threats to validity

In this section, we present threats to validity associated with the qualitative study of refactoring practices in data-intensive systems (RQ 4.6, RQ 4.7, and RQ 4.8).

10.6.1 Threats to construct validity

Construct validity: Threats to construct validity refer to the extent to which the experiment setting actually reflects the construct under study. We relied on state-of-the-art refactoring detection and SQL code smell detection tools to extract refactoring instances, SQL queries, and SQL code smells. Refactoring Miner is reported to achieve 99% precision and 94% recall [97]. However, we could still miss some refactoring instances in which the tool might introduce false positives. The SQL Inspect (precision > 88% and recall > 71.5% [33]) tool used to identify SQL queries and SQL code smells could also miss some queries and smells. Hence, the interpretation of our findings should take this into account. We also relied on import statements to identify data-access classes in NoSQL-based subject systems. There could be some cases where the import statements are not actually utilized in the code, leading to false data-access classes. However, we did not encounter any case when we perform the manual analysis in RQ 4.6 for NoSQL-based subject systems. Another potential threat to construct validity comes from the potential researcher bias in the manual analysis of **RQ 4.6 and RQ 4.7**. To overcome this threat, the manual analysis conducted by the first author was evaluated by an independent researcher and all the disagreements in labeling were resolved with discussion.

10.6.2 Threats to internal validity

Threats to internal validity concern issues that may indicate a causal relationship, although there is none [139]. We relied on LinkedIn to recruit some survey participants. There could be the case that those participants are not involved with data-access code development

and refactoring. To mitigate this threat, we carefully evaluated the LinkedIn profile of the candidate participants to make sure they have experience working with data-access code and only shared the survey link to candidates that passed the evaluation. The subject systems we selected may not represent data-intensive systems as they are obtained from open-source projects. To mitigate this threat, we applied a rigorous filtering of applications. For NoSQL subject systems, we relied on the number of SQL queries to rank the subject systems and considered the systems with the highest number of queries. Similarly, we ranked NoSQL subject systems based on the number of data-access classes and selected the top projects. Another internal threat to validity is that studies based on questionnaires could be subjective. To mitigate this threat, we provided a five point Likert scale and also included "I don't know" option to avoid forcing the respondents to pick one answer for multiple choice questions.

10.6.3 Threats to conclusion validity

Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment [139]. This threat is associated with the choice of statistical tests. We used a non-parametric test in this study. While non-parametric tests are more general than parametric tests, they have lower statistical power. However, we did not claim a causal relationship between the variables as we measured the association between them.

10.6.4 Threats to external validity

Threats to external validity concern the ability to generalize experiment results outside the experiment setting [139]. Since we performed a longitudinal study, we have to limit our subject systems to 29. This could limit the external validity of our study. However, we carefully selected our subject systems to represent data-intensive systems by considering the number of SQL queries and number of data-access classes as a proxy. Furthermore, those systems come from different application domains and rely on different data-access technologies including JDBC, SQLite, and Hibernate. Hence, our findings can be generalized to the extent of open source data-intensive systems. Another threat to external validity is the small number of survey participants (20). However, the participants come from different industries including open-source projects, which improves the generalization of the findings.

10.6.5 Threats to reliability validity

Threats to reliability validity concern the possibility for independent researchers to replicate this study. To minimize potential threats to reliability, all our subject systems are open source and available on GitHub. Furthermore, we provided all the necessary materials to replicate our study in our replication package [146].

10.7 Chapter summary

In this chapter, we presented our qualitative analysis of refactoring practices. We manually analyzed a sample of data-access refactoring commits to identify data-access functionalities that are prone to data-access refactoring. Next, we manually analyzed a sample data-access refactoring commits to determine the context in which the refactorings are applied. Finally, we analyzed the data-access refactoring survey to study the characteristics of refactorings from the developers' point of view.

The results show that data-access functionality such as data fetching and update code elements are refactoring prone as they are associated with numerous data-access refactoring instances. Large fractions of data-access refactorings are applied as part of other software development activities such as bug fixing, changing a feature, or adding a new feature. This shows that developers do not usually perform refactoring just to address technical debts. The survey analysis shows that the most common motivation to apply data-access refactorings is improving code readability, followed by improving data-access performance.

CHAPTER 11 CONCLUSION

In this chapter, we conclude the dissertation by summarizing the main findings and discussing the implications of the findings to practitioners and researchers. We finally outline the limitations of this study and future research extensions.

This year, the Internet reached more than 63% of the world population, with a large majority of users generating huge amounts of text and multimedia data every second in addition to the data generated by IoT devices. Modern business, research, and governance are data-driven, requiring analysis of this high volume, high velocity, and high variety data to obtain insights that guide business decisions and governance policies. The big data analytics market is projected to generate $103 Billion by 2027[1]. This big data can't be efficiently analyzed using traditional software systems. Hence, data-intensive software systems were introduced to answer the challenges. Data-intensive systems leverage modern cloud infrastructure to process big data at scale. Data-intensive systems devote their functionality to collecting, storing, and analyzing high volume, high velocity and variety data [2]. Data-intensive systems integrate data-storage systems (databases and distributed file systems) and software systems to analyze and transform the data. The design and implementation of data-intensive systems pose various design, implementation, and quality assurance challenges [2–4]. In addition, developers of data-intensive systems like traditional software systems face the usual release pressures that force them to compromise software quality, introducing technical debt [2]. Data-access classes link the storage components with the processing components in data-intensive systems. Hence, they are critical components of data-intensive systems. Data-access classes could be prone to both traditional technical debts and data-access technical debts. While traditional technical debts are well investigated, data-access technical debt is just getting attention recently.

The goal of this dissertation is to specify and characterize data-access technical debts, analyze the impacts of data-access technical debts on software quality, and investigate if/how data-access technical debts are addressed during refactoring.

11.1 Summary of the study findings

We summarize the main findings by dividing them based on the study objectives. We set four research objectives to study data-access technical debts that are (1) Specify data-access

[1] https://explodingtopics.com/blog/big-data-stats

technical debts (2) study the characteristics of data-access technical debts (3) Investigate the impacts of data-access technical debts on software quality and (4) Study data-access refactoring practices.

Specification data-access technical debts

We conducted a qualitative study on data-access SATDs and data-access performance issues to specify data-access SATDs and data-access performance anti-patterns respectively. We extended the category of traditional SATDs by Bavota et al. [1] by identifying 8 new data-access SATDs such as query construction, data synchronization, index management, and transaction. We also identified 14 new data-access performance anti-patterns categorized under high-level anti-patterns regarding database connection, interacting with database driver API, caching, indexing, and query.

Characterization of data-access technical debts

We analyzed the prevalence, evolution, and circumstances behind the introduction and removal of SATDs in data-intensive systems to characterize data-access SATDs. The results show that (1) for both SQL and NoSQL-based subject systems, data-access SATD is less prevalent compared to traditional SATD; (2) data-access SATDs increase in prevalence as systems evolve and (3) Data-access SATDs are more prevalent in NoSQL-based data-intensive systems compared to SQL based systems.

We conducted a survival analysis to investigate the evolution of SATDs and found that (1) the survival curves of data-access SATDs and regular SATDs have a statistically significant difference; (2) Data-access SATDs tend to be fixed sooner compared to regular ones. (3) Significant data-access SATDs were introduced in the first version, and the majority of them persisted to the most recent version of the subject systems.

We conducted both quantitative and qualitative analyses of sample data-access SATDs to investigate when (1) data-access SATDs are introduced and removed and (2) what are the circumstances behind the introduction and removal of data-access SATDs. Our findings show that: (1) the Majority of data-access SATDs are introduced at later stages of the system's evolution; (2) The distribution of data-access SATD introduction time is similar between SQL-based and NoSQL-based systems; (3) commits introducing data-access SATDs are mostly associated with *Bug fixing* and *refactoring*; (4) Commits removing data-access SATDs are mostly associated with *feature enhancements* and *adding new feature* and surprisingly, none of the removal commits were associated with refactoring. We did not notice

the removal of database access-related data-access SATDs.

We quantitatively analyzed the prevalence of SQL code smells, the co-occurrence of SQL code smells with traditional code smells, and the evolution of SQL code smells to characterize SQL code smells. The results show that: (1) ***Implicit Columns*** smell is the most prevalent SQL code smell in the data-intensive systems, followed by the ***Fear of the Unknown*** SQL code smell. We did not find an instance of other SQL code smells in the latest versions of the SQL-based subject systems.

We analyzed the co-occurrence of SQL code smells with traditional code smells using Apriori algorithm and Cramer's V statistical test of association. The results show that (1) Some SQL code smells (e.g., Implicit Columns) co-occur with traditional code smells (e.g., Long-ParameterList) but their association is weak according to Cramer's V test.

We conducted a survival analysis of SQL code smells to investigate their evolution. The results show that: (1) SQL code smells survive longer than traditional smells; and (2) many SQL code smells persist through all versions of the subject systems without getting fixed.

Overall, our results and analysis show that data-access technical debts are prevalent and persist for a long period of time during the evolution of subject systems.

Impacts of data-access technical debts

We provided empirical evidence that data-access technical debts are prevalent and persistent in data-intensive systems. The next objective was to analyze the impacts of data-access technical debts on software quality. We investigated the potential impact of SQL code smells on bugs and the perceived impacts of SQL code smells. We also investigated the perceived impacts of data-access performance anti-patterns on performance.

We analyzed the co-occurrence of SQL code smells with software bugs using the Apriori algorithm and Cramer's V association test. We did not find a statistically significant association between SQL code smells and software bugs, while some traditional smells have a significant co-occurrence with bugs (e.g., ComplexClass, SpaghettiCode).

We asked software developers to rate the criticality of SQL code smells as part of the survey on refactoring practices. The majority of the respondents rate SQL code smells as critical and highly critical, and mentioned that they consider addressing SQL code smells during refactoring. This shows that although SQL code smells do not introduce defects, they affect data-access performance and create software maintainability issues that need further investigation.

We asked software developers to rate the criticality of the 14 newly identified data-access

performance anti-patterns. *Improper handling of node failures*, *Using synchronous connection*, and *Inefficient driver API* were rated critical and highly critical by the majority of the respondents. This shows that some identified data-access performance anti-patters are viewed as a critical impacting data-access performance.

Overall, our results show that SQL code smells and some data-access performance anti-patterns are perceived as critical technical debts impacting the performance of data-access classes.

Data-access refactoring practices

We have shown that data-access technical debts are prevalent and persistent and impact performance of data-access classes. Since refactorings are remedies of technical debts, We finally analyzed refactoring practices in data-access classes to investigate if/how developers refactor data-access classes and address data-access technical debts.

We first analyzed if refactoring is prevalent in data-access classes using both SQL-based and NoSQL-based data-intensive subject systems. The results show that: (1) refactoring in data-access classes are slightly less prevalent compared to regular classes ; (2) The distribution of refactoring density (number of refactorings divided by code size) is similar between data-access and regular classes; (3) Among different refactoring types we analyzed, ***Change Parameter Type*** is the most prevalent type in data-access classes.

We also analyzed the prevalence of refactorings as subject systems involve and if/how it is affected by software release schedules. The findings show that (1) Data-access refactorings have a tendency to be applied at later stages of the systems' evolution compared to regular refactorings. In particular, prevalent data-access refactoring types *Move Method & Add Method Annotation* has a tendency to be applied at the later stages of the software evolution. (2) The prevalence of data access refactorings is roughly similar regardless of release schedules, while regular refactorings tend to be applied far before official releases. (3) Data-access refactorings in SQL-based data-intensive systems tend to be applied at later stages compared to data-access refactorings in NoSQL-based data-intensive systems.

We investigated if data-access refactoring changes touch SQL queries and SQL code smells using both line-level and method-level matching. Our results show that only 0.45% of data-access refactorings were applied on statements containing SQL query while 29.68% of data-access refactorings were performed on methods containing SQL queries. However, the applied refactorings did not modify the queries and consequently did not address SQL code smells. This shows that developers generally do not address SQL code smells during refactoring.

We investigated if composite refactoring, applying different types of refactorings in the same commit, is prevalent in data-intensive systems leveraging Apriori algorithm and Cramer's V test of association. Our findings show that: (1) refactorings that change the type of variable or parameter tend to co-occur with refactorings that rename identifiers in both data-access and regular classes. (2) Not more than two refactoring types co-occur in the same commit for both data-access and regular refactorings.

We analyzed the profile of developers involved in refactoring using several developer profile metrics and compared the profile of developers involved in data-access refactoring with regular refactorings. Our findings show that: (1) *Refactoring contribution* profile metric is the most distinguishing metric between developers that involve in data-access refactoring against developers that involve in only regular refactoring. (2) Developers involved in data-access refactoring are more experienced with the subject systems than developers involved in regular refactoring only. (3) Not all developers equally contribute to refactoring. Most of the refactorings is done by developers that are class owners (most of the changes performed on the class is done by them).

We manually analyzed the code elements targeted by data-access refactorings to understand which data-access class functionalities are refactoring prone. Our finding shows that: (1) Data-fetching and inserting data operations are more refactorings prone. (2) *Rename Variable* refactoring is the most prevalent refactoring applied on data fetching code, while *Add Parameter Modifier* is the most frequent refactoring applied on data insertion code.

We manually analyzed a sample of data-access refactoring commits and labeled the purpose of the commit from the commit message and changed files. Our result shows that: (1) Only 24% of data-access refactorings are pure refactorings the remaining refactorings are done as part of other software development activities such as changing features and bug fixing.

We prepared a developer survey to characterize refactorings from the point of view of developers and to complement our empirical findings. We analyzed the survey responses, and the findings show that: (1) Improving data-access performance is the most common motivation behind data-access refactoring after improving code readability. (2) Developers confirmed that coding experience, refactoring contribution, and code familiarity are important factors to assign developers for refactoring. (3) Large number of respondents mentioned that they apply data-access refactoring during changing features, which aligns with our empirical analysis findings.

Overall, while data-access refactorings are prevalent in data-intensive systems, they are not performed just to address technical debts rather they are performed as part of addressing functional requirements and the applied refactorings do not modify data-access statements

(E.g. SQL queries) and do not generally address data-access technical debts.

11.2 Implication of the findings

We have demonstrated that data-access technical debts are (1) prevalent and persistent, (2) impact software quality and (3) are not addressed by refactoring. In this section, we discuss the implication of our findings for the research community and practitioners.

Implication to researchers

We extended the SATD taxonomy proposed by Bavota et al. [1] incorporating data-access SATDs. Researches in SATD detection approach can leverage this taxonomy to identify the type of SATD rather than just classifying the comment as SATD or not. Having a more granular SATD detection tool will help software developers by providing more information about the SATD and help them to prioritize SATDs based on their criticality. The tools could also allow researchers to conduct more fine-grained analysis of the prevalence, evolution, and impact of specific types of SATDs and data-access SATDs in particular.

We also specified data-access performance anti-patterns by analyzing performance issues collected from NoSQL-based and polyglot data-intensive systems. Researchers can leverage data-access performance anti-patterns from this study and previous studies to propose automatic detection approach and possibly tools. Furthermore, not all the anti-patterns are critical. Hence, we recommend prioritizing *Improper handling of node failures*, *Using synchronous connection*, and *Inefficient driver API* performance anti-patterns as they are more critical than others. Once data-access performance detection tools are available, developers can leverage them to efficiently identify and prioritize them for refactoring and allows investigating of the prevalence, evolution, and impact of such data-access anti-patterns.

We investigated data-access SATDs, SQL code smells and specified data-access performance anti-patterns. However, technical debts could occur at all stages of software development. Hence, researchers should complement our findings on data-access technical debts by investigating other types of technical debts that could impact data-access performance.

Our analysis of developer's survey show that SQL code smells and data-access performance anti-patterns are critical to data-intensive systems and could impact performance. But the findings are based on the experience of software developers. Hence, more research is needed to further highlight the impacts of such anti-patterns on the software quality of data-intensive systems. For instance, the performance impact of those smells could be investigated by analyzing the operation logs and performance reports of the systems and correlating them with

code changes. Another way could be to measure performance using a controlled experiment before and after applying refactoring, or to profile applications performance before and after the introduction of SQL code smells. We particularly recommend researchers to analyze the performance impact of *Implicit Columns* with this approach since this smell is prevalent and perceived by developers as critical and most likely impact performance.

We observed that data-access technical debts such as SQL code smells are not addressed during refactoring. This calls for further research to find out if it is due to lack of developer's awareness or due to external factors such as pressures from functional requirements. Another reason for having low instances of data-access refactorings touching SQL statements or fixing SQL code smells could be due to the limitation of existing refactoring detection tools to detect changes in queries. One way to incorporate data-access technical debts in refactoring detection is to extend existing detection tools to record syntactic and semantic changes in data-access statements. The first step for such an approach could be to extract data-access statements from the source code, which is not trivial to the case of NoSQL-based systems. While there are tools that can extract SQL statements from source code (E.g. SQLInspect), more research is required to have a similar approach for NoSQL database-based systems. The second step is to compare the changes in the data-access statements to identify changes and potential refactorings. This step requires the specification of refactoring corresponding to each data-access technical debt. For some SQL code smells such as *Implicit columns* and *Fear of the unknown*, the refactoring detector may need to consult the underlying data-source schema to identify the refactoring needed.

Complex refactoring operations that involve multiple classes and packages are not prevalent in data-intensive systems. One reason for this is that most of the refactoring tools support low-level refactoring operations, assisting developers and improving productivity. However, having strong support for more complex refactoring types will be beneficial to developers.

We demonstrated that some refactoring types have a strong association with each other. Hence, the performance of refactoring recommendation tools could be enhanced by making the recommendation systems consider related refactorings as a feature. For example, we found that *rename parameter* is frequently associated with *changing parameter*. If the tools detect that changing parameter refactoring is applied, then the tools could recommend renaming the target code entity. It is also interesting to investigate if leveraging information obtained from one refactoring when detecting its composite refactoring pair improves the detection performance.

The process to assign developers to refactor data-access code can be automated by leveraging features such as developer experience, refactoring contribution, and code ownership. A

machine learning model leveraging those features in addition to the features extracted from the target code could provide reliable developer recommendations to refactor data-access classes. This recommendation tool will benefit the software development process by reducing the time needed to assign refactoring to a developer and improving the quality of the applied refactoring.

Implication to practitioners

Our results show that data-access technical debts are prevalent and persistent in data-intensive systems, and they are not removed during refactoring operations. Hence, we recommend developers to consider addressing such technical debts to minimize their impact on software quality and maintainability. Developers could leverage our findings to identify and prioritize data-access technical debts that are more critical to the overall performance of data-intensive systems.

We showed that Bug fixing and refactoring commits often introduce data-access SATDs which shows that addressing technical debts in one element could introduce another technical debt. Hence, we recommend practitioners to assess if new changes introduce technical debts during code reviews.

We also showed that refactorings are done as part of other software development activities such as bug fixing, adding new feature or changing feature. However, we recommend developers to consider regularly addressing technical debts, as some data-access technical debts may not get attention and could later be more problematic. We recommend quality assurance teams to leverage our findings and technical debt detection tools to regularly evaluate the software product, prioritize technical debts and enforce regular refactoring. In most cases, the product owners ask developers to focus on addressing functional requirements because the associated business impact is apparent. However, this is not the case for technical debts. Hence, a business value must be associated with refactoring activities to improve code quality and address data-access technical debts. Both developers and quality assurance teams need to be aware of the importance of addressing technical debts in general and data-access technical debts with regular refactoring.

11.3 Future research opportunities

We believe that this dissertation is the first step toward a comprehensive analysis of the characteristics and impacts of data-access technical debts on software quality. Our study opens several research directions.

We specified data-access performance anti-patterns and evaluated their perceived impact on performance. One research direction is to work on automatically detecting the data-access performance anti-patterns leveraging static or dynamic analysis of software artifacts. The outcome of this work will pave the way to a more comprehensive analysis of the prevalence, co-occurrence, evolution, and impact of data-access anti-patters.

We evaluated the perceived criticality of SQL code smells and data-access performance anti-patterns in this study. Our analysis approach can be replicated to data-access SATDs using a mixed method combining analysis of developer survey and quantitative study of the impact of data-access SATDs on software quality.

Another extension of the work on characterization of SQL code smells is investigating the circumstances behind the introduction and removal of SQL code smells. The findings will provide insights on when they are introduced and removed and what is the context of the commits that introduced or removed them.

Another interesting extension would be to investigate if/how data-access SATDs are addressed during refactoring. The findings will give insights into how data-access SATDs are resolved.

We relied on open-source data-intensive systems as subject systems in this study. Replicating the studies in this dissertation to industrial data-intensive systems will improve the generalization of our findings regarding the prevalence, evolution, and impact of data-access technical debts.

REFERENCES

[1] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 315–326.

[2] H. Foidl, M. Felderer, and S. Biffl, "Technical debt in data-intensive software systems," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2019, pp. 338–341.

[3] O. Hummel, H. Eichelberger, A. Giloj, D. Werle, and K. Schmid, "A collection of software engineering challenges for big data system development," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018, pp. 362–369.

[4] B. Park, D. L. Rao, and V. N. Gudivada, "Dangers of bias in data-intensive information systems," in *Next Generation Information Processing System*, P. Deshpande, A. Abraham, B. Iyer, and K. Ma, Eds. Singapore: Springer Singapore, 2021, pp. 259–271.

[5] W. Cunningham, "The wycash portfolio management system," in *OOPSLA '92*, 1992.

[6] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, no. C, p. 193–220, Mar. 2015.

[7] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 91–100.

[8] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 75–84.

[9] R. Shatnawi and W. Li, "An investigation of bad smells in object-oriented design," in *Third International Conference on Information Technology: New Generations (ITNG'06)*. IEEE, 2006, pp. 161–165.

[10] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[11] D. Johannes, F. Khomh, and G. Antoniol, "A large-scale empirical study of code smells in javascript projects," *Software Quality Journal*, pp. 1–44, 2019.

[12] S. Shao, Z. Qiu, X. Yu, W. Yang, G. Jin, T. Xie, and X. Wu, "Database-access performance antipatterns in database-backed web applications," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 58–69.

[13] M. Fowler, "Refactoring: Improving the design of existing code," *Extreme Programming and Agile Methods–XP/Agile Universe 2002*, p. 256, 2002.

[14] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[15] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 411–416.

[16] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proc. of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2009, pp. 390–400.

[17] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, 2018.

[18] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[19] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, "Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains," in *2013 IEEE International Conference on Software Maintenance*, Sep. 2013, pp. 260–269.

[20] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.

[21] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 682–691.

[22] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, "A large-scale empirical exploration on refactoring activities in open source software projects," *Science of Computer Programming*, vol. 180, pp. 1–15, 2019.

[23] A. Peruma, "A preliminary study of android refactorings," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2019, pp. 148–149.

[24] M. Iammarino, F. Zampetti, L. Aversano, and M. Di Penta, "Self-admitted technical debt removal and refactoring actions: Co-occurrence or more?" in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 186–190.

[25] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "An empirical investigation of how and why developers rename identifiers," in *Proceedings of the 2nd International Workshop on Refactoring*, 2018, pp. 26–33.

[26] M. W. M. Anthony Peruma, M. J. Decker, and C. D. Newman, "Contextualizing rename decisions using refactorings, commit messages, and data types," *J. Syst. Softw.*, vol. 169, p. 110704, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110704

[27] B. Karwin, *SQL Antipatterns: Avoiding the pitfalls of database programming.* Pragmatic Bookshelf, 2010.

[28] J. H. Weber, A. Cleve, L. Meurice, and F. J. B. Ruiz, "Managing technical debt in database schemas of critical software," in *2014 Sixth International Workshop on Managing Technical Debt*, 2014, pp. 43–46.

[29] T. Sharma, M. Fragkoulis, S. Rizou, M. Bruntink, and D. Spinellis, "Smelly relations: Measuring and understanding database schema quality," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, May 2018, pp. 55–64.

[30] F. G. de Almeida Filho, A. D. F. Martins, T. d. S. Vinuto, J. M. Monteiro, Í. P. de Sousa, J. de Castro Machado, and L. S. Rocha, "Prevalence of bad smells in pl/sql projects," in *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 2019, pp. 116–121.

[31] C. Nagy and A. Cleve, "SQLInspect: A static analyzer to inspect database usage in Java applications," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings.* ACM, 2018, pp. 93–96.

[32] N. Csaba and A. Cleve, "A static code smell detector for SQL queries embedded in java code," pp. 147–152, 2017. [Online]. Available: https://doi.org/10.1109/SCAM.2017.19

[33] L. Meurice, C. Nagy, and A. Cleve, "Static analysis of dynamic database usage in Java systems," in *International Conference on Advanced Information Systems Engineering.* Springer, 2016, pp. 491–506.

[34] R. G. Miller Jr, *Survival analysis.* John Wiley & Sons, 2011, vol. 66.

[35] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal? an in-depth perspective," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. Association for Computing Machinery, 2018, p. 526–536.

[36] E. L. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American statistical association*, vol. 53, no. 282, pp. 457–481, 1958.

[37] C. H. Papadimitriou, P. Raghavan, H. Tamaki, and S. Vempala, "Latent semantic indexing: A probabilistic analysis," *Journal of Computer and System Sciences*, vol. 61, no. 2, pp. 217–235, 2000.

[38] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[39] W. Zhao, J. J. Chen, R. Perkins, Z. Liu, W. Ge, Y. Ding, and W. Zou, "A heuristic approach to determine an appropriate number of topics in topic modeling," *BMC Bioinformatics*, vol. 16, no. 13, p. S8, Dec 2015.

[40] J. Chang, S. Gerrish, C. Wang, J. Boyd-graber, and D. Blei, "Reading tea leaves: How humans interpret topic models," in *Advances in Neural Information Processing Systems*, Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, Eds., vol. 22. Curran Associates, Inc., 2009.

[41] M. Röder, A. Both, and A. Hinneburg, "Exploring the space of topic coherence measures," in *Proceedings of the eighth ACM international conference on Web search and data mining*, 2015, pp. 399–408.

[42] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[43] R. Agrawal, T. Imielinski, and A. Swami, "Mining associations between sets of items in large databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1993, pp. 207–216.

[44] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," *Acm Sigmod Record*, vol. 26, no. 2, pp. 255–264, 1997.

[45] S. Piatetsky, G. Frawley, and J. William, "Discovery, analysis and presentation of strong rules, knowledge discovery in databases," 1991.

[46] H. Cramer, "Mathematical methods of statistics," *Princeton U. Press, Princeton*, p. 500, 1946.

[47] N. Rios, M. G. de Mendonça Neto, and R. O. Spínola, "A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners," *Information and Software Technology*, vol. 102, pp. 117 – 145, 2018.

[48] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt," *Inf. Softw. Technol.*, vol. 70, no. C, p. 100–121, Feb. 2016.

[49] R. Alfayez, W. Alwehaibi, R. Winn, E. Venson, and B. Boehm, "A systematic literature review of technical debt prioritization," in *Proceedings of the 3rd International Conference on Technical Debt*, ser. TechDebt '20. Association for Computing Machinery, 2020, p. 1–10.

[50] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, and E. Gamma, *Refactoring: Improving the Design of Existing Code.* USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[51] M. Abidi, F. Khomh, and Y.-G. Guéhéneuc, "Anti-patterns for multi-language systems," in *Proceedings of the 24th European Conference on Pattern Languages of Programs*, 2019, pp. 1–14.

[52] M. Grichi, M. Abidi, Y.-G. Guéhéneuc, and F. Khomh, "State of practices of java native interface," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, 2019, pp. 274–283.

[53] A. Nikanjam and F. Khomh, “Design smells in deep learning programs: An empirical study,” in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*. IEEE, 2021, pp. 332–342. [Online]. Available: https://doi.org/10.1109/ICSME52107.2021.00036

[54] A. Borrelli, V. Nardone, G. A. D. Lucca, G. Canfora, and M. D. Penta, “Detecting video game-specific bad smells in unity projects,” in *MSR ’20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, S. Kim, G. Gousios, S. Nadi, and J. Hejderup, Eds. ACM, 2020, pp. 198–208. [Online]. Available: https://doi.org/10.1145/3379597.3387454

[55] V. Nardone, B. A. Muse, M. Abidi, F. Khomh, and M. D. Penta, “Video game bad smells: What they are and how developers perceive them,” *ACM Transactions on Software Engineering and Methodology*, 2021.

[56] M. Al-Barak and R. Bahsoon, “Database design debts through examining schema evolution,” in *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, 2016, pp. 17–23.

[57] M. Albarak and R. Bahsoon, “Prioritizing technical debt in database normalization using portfolio theory and data quality metrics,” in *Proceedings of the 2018 International Conference on Technical Debt*, ser. TechDebt ’18. Association for Computing Machinery, 2018, p. 31–40.

[58] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, “Understanding and detecting real-world performance bugs,” *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 77–88, 2012.

[59] A. Nistor, T. Jiang, and L. Tan, “Discovering, reporting, and fixing performance bugs,” in *2013 10th working conference on mining software repositories (MSR)*. IEEE, 2013, pp. 237–246.

[60] Y. Liu, C. Xu, and S.-C. Cheung, “Characterizing and detecting performance bugs for smartphone applications,” in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1013–1024.

[61] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung, “How not to structure your database-backed web applications: a study of performance bugs in the wild,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 800–810.

[62] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: an empirical study," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 61–72.

[63] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 562–571.

[64] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1001–1012.

[65] T. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. N. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Trans. Software Eng.*, vol. 42, no. 12, pp. 1148–1161, 2016. [Online]. Available: https://doi.org/10.1109/TSE.2016.2553039

[66] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung, "Powerstation: Automatically detecting and fixing inefficiencies of database-backed web applications in ide," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 884–887.

[67] I. T. Bowman and K. Salem, "Optimization of query streams using semantic prefetching," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 4, pp. 1056–1101, 2005.

[68] A. Cheung, O. Arden, S. Madden, and A. C. Myers, "Automatic partitioning of database applications," *arXiv preprint arXiv:1208.0271*, 2012.

[69] X. Xiao, S. Han, D. Zhang, and T. Xie, "Context-sensitive delta inference for identifying workload-dependent performance bottlenecks," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 90–100.

[70] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 666–677.

[71] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, “Decor: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[72] F. Palomba, “Textual analysis for code smell detection,” in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 769–771.

[73] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, and S. Hamel, “Ids: An immune-inspired approach for the detection of software design smells,” in *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 2010, pp. 343–348.

[74] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, “Bdtex: A gqm-based bayesian approach for the detection of antipatterns,” *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[75] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, and E. Aimeur, “Smurf: A svm-based incremental anti-pattern detection approach,” in *2012 19th Working Conference on Reverse Engineering*, 2012, pp. 466–475.

[76] M. Kessentini and A. Ouni, “Detecting android smells using multi-objective genetic programming,” in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 2017, pp. 122–132.

[77] N. Tsantalis, “Evaluation and improvement of software architecture: Identification of design problems in object-oriented systems and resolution through refactorings,” *Diss. Ph. D. dissertation, Univ. of Macedonia*, 2010.

[78] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, “Faultbuster: An automatic code smell refactoring toolset,” in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, pp. 253–258.

[79] M. A. de Freitas Farias, M. G. de Mendonça Neto, A. B. da Silva, and R. O. Spínola, “A contextualized vocabulary model for identifying technical debt on code comments,” in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2015, pp. 25–32.

[80] M. A. de Freitas Farias, J. A. Santos, M. Kalinowski, M. Mendonça, and R. O. Spínola, “Investigating the identification of technical debt through code comment analysis,” in

International Conference on Enterprise Information Systems. Springer, 2016, pp. 284–309.

[81] E. da Silva Maldonado, E. Shihab, and N. Tsantalis, “Using natural language processing to automatically detect self-admitted technical debt,” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, 2017.

[82] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, “Identifying self-admitted technical debt in open source projects using text mining,” *Empirical Software Engineering*, vol. 23, no. 1, pp. 418–451, 2018.

[83] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, “Satd detector: A text-mining-based self-admitted technical debt detection tool,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 9–12.

[84] M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin, and X. Yang, “Automating change-level self-admitted technical debt determination,” *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1211–1229, 2018.

[85] Z. Yu, F. M. Fahid, H. Tu, and T. Menzies, “Identifying self-admitted technical debts with jitterbug: A two-step approach,” *arXiv preprint arXiv:2002.11049*, 2020.

[86] F. Zampetti, A. Serebrenik, and M. Di Penta, “Automatically learning patterns for self-admitted technical debt removal,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2020, pp. 355–366.

[87] C. Noiseux, “Recommending when design technical debt should be self-admitted,” Ph.D. dissertation, École Polytechnique de Montréal, 2017.

[88] H. Van Den Brink, R. Van Der Leek, and J. Visser, “Quality assessment for embedded sql,” in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007).* IEEE, 2007, pp. 163–170.

[89] J. Delplanque, A. Etien, O. Auverlot, T. Mens, N. Anquetil, and S. Ducasse, “Code-critics applied to database schema: Challenges and first results,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2017, pp. 432–436.

[90] S. Demeyer, S. Ducasse, and O. Nierstrasz, “Finding refactorings via change metrics,” pp. 166–177, 2000. [Online]. Available: https://doi.org/10.1145/353171.353183

[91] G. Antoniol, M. Di Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.* IEEE, 2004, pp. 31–40.

[92] P. Weißgerber and S. Diehl, "Identifying refactorings from source-code changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06).* IEEE, 2006, pp. 231–240.

[93] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *European conference on object-oriented programming.* Springer, 2006, pp. 404–428.

[94] Z. Xing and E. Stroulia, "The jdevan tool suite in support of object-oriented evolutionary development," in *Companion of the 30th international conference on Software engineering*, 2008, pp. 951–952.

[95] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 371–372.

[96] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[97] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020.

[98] D. Silva, J. Silva, G. J. D. S. Santos, R. Terra, and M. T. O. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, 2020.

[99] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206

[100] A. J. Riel, *Object-Oriented Design Heuristics*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[101] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonça, "A systematic review on the code smell effect," *Journal of Systems and Software*, vol. 144, pp. 450 – 477, 2018.

[102] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[103] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 145–154.

[104] M. Abidi, M. S. Rahman, M. Openja, and F. Khomh, "Are multi-language design smells fault-prone? an empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 29:1–29:56, 2021. [Online]. Available: https://doi.org/10.1145/3432690

[105] Y. Kamei, E. d. S. Maldonado, E. Shihab, and N. Ubayashi, "Using analytics to quantify interest of self-admitted technical debt." in *QuASoQ/TDA@ APSEC*, 2016, pp. 68–71.

[106] E. d. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 238–248.

[107] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 17–23.

[108] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the international conference on mobile software engineering and systems*, 2016, pp. 59–69.

[109] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: An energy-aware refactoring approach for mobile apps," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1176–1206, 2017.

[110] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 179–188.

[111] D. Silva, N. Tsantalis, and M. T. Valente, “Why we refactor? confessions of github contributors,” in *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, 2016, pp. 858–870.

[112] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, “How does refactoring affect internal quality attributes? a multi-project study,” in *Proceedings of the 31st Brazilian symposium on software engineering*, 2017, pp. 74–83.

[113] I. Ferreira, E. Fernandes, D. Cedrim, A. Uchôa, A. C. Bibiano, A. Garcia, J. L. Correia, F. Santos, G. Nunes, C. Barbosa *et al.*, “The buggy side of code refactoring: Understanding the relationship between refactorings and bugs,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 406–407.

[114] M. Mahmoudi, S. Nadi, and N. Tsantalis, “Are refactorings to blame? an empirical study of refactorings in merge conflicts,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 151–162.

[115] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, and M. Kessentini, “How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation,” *Expert Systems with Applications*, vol. 167, p. 114176, 2021.

[116] G. Inc, “Search,” December 2019. [Online]. Available: https://developer.github.com/v3/search/

[117] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen, “Code smells for model-view-controller architectures,” *Empirical Software Engineering*, vol. 23, no. 4, pp. 2121–2157, 2018.

[118] Y.-G. Guéhéneuc, “Ptidej: A flexible reverse engineering tool suite,” in *2007 IEEE International Conference on Software Maintenance*. IEEE, 2007, pp. 529–530.

[119] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, and A. Tiberghien, “From a domain analysis to the specification and detection of code and design smells,” *Formal Aspects of Computing*, vol. 22, no. 3-4, pp. 345–361, 2010.

[120] D. Spadini, M. Aniche, and A. Bacchelli, “PyDriller: Python framework for mining software repositories,” in *Proc of the 26th ACM Joint European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2018, p. 908–911.

[121] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases." in *Proc. of the 2000 International Conference on Software Maintenance*, 2000, pp. 120–130.

[122] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? a text-based approach to classify change requests," in *CASCON*, vol. 8, 2008, pp. 304–318.

[123] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[124] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[125] L. Guerrouj, Z. Kermansaravi, V. Arnaoudova, B. C. Fung, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Investigating the relation between lexical smells and change- and fault-proneness: an empirical study," *Software Quality Journal*, vol. 25, no. 3, pp. 641–670, 2017.

[126] B. A. Muse, "Replication package," 2020. [Online]. Available: https://github.com/Biruk-Asmare/MSR_2020_SQLSmells_Prevalence

[127] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[128] P. Benats, M. Gobert, L. Meurice, C. Nagy, and A. Cleve, "An empirical study of (multi-) database models in open-source projects," in *International Conference on Conceptual Modeling*. Springer, 2021, pp. 87–101.

[129] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *2014 Sixth International Workshop on Managing Technical Debt*, 2014, pp. 1–7.

[130] R. Maipradit, C. Treude, H. Hata, and K. Matsumoto, "Wait for it: identifying "on-hold" self-admitted technical debt," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3770–3798, 2020.

[131] B. Chen, Z. M. Jiang, P. Matos, and M. Lacaria, "An industrial experience report on performance-aware refactoring on a database-centric web application," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 653–664.

[132] Y. Lyu, A. Alotaibi, and W. G. J. Halfond, "Quantifying the performance impact of SQL antipatterns on mobile applications," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 53–64.

[133] S. Chaudhuri, V. Narasayya, and M. Syamala, "Bridging the application and dbms profiling divide for database application developers," in *Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 1252–1262.

[134] Z. Scully and A. Chlipala, "A program optimization for automatic database result caching," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 271–284.

[135] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding database performance inefficiencies in real-world web applications," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 1299–1308.

[136] J. Yang, C. Yan, C. Wan, S. Lu, and A. Cheung, "View-centric performance optimization for database-backed web applications," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 994–1004.

[137] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang, "Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: an experience report," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 1–12.

[138] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan, "Program transformations for asynchronous and batched query submission," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 2, pp. 531–544, 2014.

[139] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[140] B. A. Muse, C. Nagy, F. Khomh, A. Cleve, and G. Antoniol, "Replication package for: FIXME: Synchronize with Database. An Empirical Study of

Data Access Self-Admitted Technical Debt," Jan. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.5825671

[141] B. A. Muse, "Replication package," 2022. [Online]. Available: https://github.com/Biruk-Asmare/data_acess_performance_antipatterns_in_data_intensive_systems_RR.git

[142] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 403–414.

[143] R. Peto and J. Peto, "Asymptotically efficient rank invariant test procedures," *Journal of the Royal Statistical Society: Series A (General)*, vol. 135, no. 2, pp. 185–198, 1972.

[144] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[145] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *Information and Software Technology*, vol. 99, pp. 164–176, 2018.

[146] B. A. Muse, F. Khomh, and G. Antoniol, "Replication package: Refactoring Practices in the Context of Data-intensive Systems," Sep. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7140854

[147] N. McDonald and S. Goggins, "Performance and participation in open source software on github," in *CHI'13 extended abstracts on human factors in computing systems*, 2013, pp. 139–144.

[148] C. Zhou, S. K. Kuttal, and I. Ahmed, "What makes a good developer? an empirical study of developers' technical and social competencies," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 319–321.

[149] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[150] J. Zhou, S. Wang, C.-P. Bezemer, Y. Zou, and A. E. Hassan, “Studying the association between bountysource bounties and the issue-addressing likelihood of github issue reports,” *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2919–2933, 2021.

[151] Z. Kurtanović and W. Maalej, “On user rationale in software engineering,” *Requirements Engineering*, vol. 23, no. 3, pp. 357–379, 2018.

[152] G. C. Murphy, M. Kersten, and L. Findlater, “How are java software developers using the elipse ide?” *IEEE software*, vol. 23, no. 4, pp. 76–83, 2006.