On the Prevalence, Impact, and Evolution of SQL Code Smells in Data-Intensive Systems

Biruk Asmare, Muse
Polytechnique Montréal
biruk-asmare.muse@polymtl.ca

Mohammad Masudur Rahman
Polytechnique Montréal
masud.rahman@polymtl.ca

Csaba Nagy
Software Institute
Università della Svizzera italiana
csaba.nagy@usi.ch

Anthony Cleve
Namur Digital Institute
University of Namur
anthony.cleve@unamur.be

Foutse Khomh
Polytechnique Montréal
foutse.khomh@polymtl.ca

Giuliano Antoniol
Polytechnique Montréal
antoniol@ieee.org

ABSTRACT

Code smells indicate software design problems that harm software quality. Data-intensive systems that frequently access databases often suffer from SQL code smells besides the traditional smells. While there have been extensive studies on traditional code smells, recently, there has been a growing interest in SQL code smells. In this paper, we conduct an empirical study to investigate the prevalence and evolution of SQL code smells in open-source, data-intensive systems. We collected 150 projects and examined both traditional and SQL code smells in these projects. Our investigation delivers several important findings. First, SQL code smells are indeed prevalent in data-intensive software systems. Second, SQL code smells have a weak co-occurrence with traditional code smells. Third, SQL code smells have a weaker association with bugs than that of traditional code smells. Fourth, SQL code smells are more likely to be introduced at the beginning of the project lifetime and likely to be left in the code without a fix, compared to traditional code smells. Overall, our results show that SQL code smells are indeed prevalent and persistent in the studied data-intensive software systems. Developers should be aware of these smells and consider detecting and refactoring SQL code smells and traditional code smells separately, using dedicated tools.

CCS CONCEPTS

• **Software and its engineering** → *Empirical software validation.*

KEYWORDS

Code smells, database access, SQL code smells, data-intensive systems

ACM Reference Format:
Biruk Asmare, Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. 2020. On the Prevalence, Impact, and Evolution of SQL Code Smells in Data-Intensive Systems. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3379597.3387467

1 INTRODUCTION

In Software Engineering, *smells* are *poor* solutions to commonly occurring problems in a software system. They could be found within the design and implementation of the system, and are frequently referred to as design smells and code smells, respectively. Regardless of whether they originate from the design or the source code, previous work have shown that they can negatively affect the performance [20] or the maintainability [29] of a software system. They should therefore be handled with special care, and refactored [15] as soon as possible. We refer to these smells as *traditional smells* throughout the paper. While there have been extensive studies on traditional smells [23, 29, 48, 57], recently, there has been a growing interest in a particular type of smells, namely *SQL code smells* [13, 26, 43].

SQL code smells are found within SQL code as a result of misuses in queries (e.g., *Implicit Columns* [43]). Data-intensive software systems that frequently interact with databases are particularly prone to these smells. The SQL queries are often embedded in the application code and remain hidden from the developers, which makes it harder to spot mistakes in them. As SQL is the principal language to communicate with relational databases, which represent the Top-5 databases according to the DB-Engine Ranking[1], many systems might suffer from such a type of smells. Although traditional code smells are widely studied, there have been only a few studies on the prevalence and impact of SQL code smells [13, 35, 56]. The objective of our study is to address this gap in the literature of code smells.

In this paper, we conduct an empirical study on SQL code smells, that investigates their (1) prevalence, (2) impact, (3) evolution and (4) co-occurrence with the traditional code smells within hundreds of open-source software systems. To the best of our knowledge, this is the first study investigating the prevalence, impact, and evolution of SQL code smells and also contrasting with the traditional smells.

Our study relies on the analysis of 150 open-source software systems that manipulate their databases through popular database access APIs – Android Database API, JDBC, JPA and Hibernate. We

[1]https://db-engines.com/en/ranking

analysed the source code of each project and studied 19 traditional code smells using the DECOR tool [17] and 4 SQL code smells using the SQLInspect tool [44]. We also collected bug-fixing and bug-inducing commits from each project using PyDriller [60].

By analyzing the collected data, we answer the four following research questions:

RQ1: What is the prevalence of SQL code smells across different application domains?

We study the prevalence of SQL code smells in the selected software systems by categorizing them into four application domains – Business, Library, Multimedia, and Utility. We find that SQL code smells are prevalent in all four domains. Some SQL code smells are more prevalent than others.

RQ2: Do traditional code smells and SQL code smells co-occur at class level?

We investigate the co-occurrence of SQL code smells and traditional code smells using association rule mining. The results show that while some SQL code smells have statistically significant co-occurrence with traditional code smells, the degree of association is low.

RQ3: Do SQL code smells co-occur with bugs?

We investigate the potential impact of SQL code smells on software bugs by analysing their co-occurrences within the bug-inducing commits. We perform Cramer's V test of association and build a random forest model to study the impact of the smells on bugs. We find that there is a weak association between SQL code smells and software bugs. Some SQL code smells tend to show higher association with bugs compared to others.

RQ4: How long do SQL code smells survive?

We perform a survival analysis of SQL and traditional code smells using Kaplan-Meier survival curves to compare their survival time. It is interesting to know the lifespan of SQL code smells as software evolves. It indicates whether the smells stay longer without getting fixed or not. We find that the survival time of SQL code smells is higher compared to that of traditional code smells. Furthermore, significant portions of the SQL code smells are created at the very beginning and then persist in all subsequent versions of the systems.

2 BACKGROUND

2.1 SQL Code Smells

Besides many white papers and blog posts about common mistakes or bad practices in SQL queries [34], an extensive catalogue of SQL code smells was published by Karwin in 2010 [26].

There are also tools (e.g., TOAD and SQL Enlight) typically designed for database administrators that can statically analyse queries and identify common mistakes. These techniques require the SQL code as input. For our study, to investigate SQL code embedded in the source code of data-intensive systems, we rely on SQLInspect [44], a tool able to extract SQL code from Java applications and detect SQL code smells belonging to Karwin's catalogue. In the following, we briefly describe the SQL code smells that SQLInspect can detect.

Implicit Columns smell occurs when columns of a table are unnecessarily queried, e.g., the usage of `*` in the column list of a `SELECT` statement. Although it is fast to write, it may cause performance issues such as network bandwidth wastage or even more serious problems when the table column order is modified and the change is not propagated to the application code [26].

Fear of the Unknown is a smell that occurs due to improper handling of `NULL` values. `NULL` has a special meaning in relational databases as it indicates the absence of data, and it is often misinterpreted by developers. For example, developers should check for `NULL` values using the `IS NULL` operator instead of the otherwise syntactically correct `!= NULL` expression that always returns `UNKNOWN` in SQL [43].

Ambiguous Groups smell occurs when developers misuse the `GROUP BY` aggregation command. For example, adding columns in the select list other than the ones used in aggregation function or in `GROUP BY` clause may generate erroneous results [26].

Random Selection occurs when developers query a single random row. This operation requires a full scan of the required table. This will have a negative impact on the performance as the size of the table increases [26].

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 catalogue 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, 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% [38]. 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. [43, 44].

2.2 Apriori: Association Rule Mining Algorithm

Apriori is an algorithm devised for mining frequent itemsets and relevant association rules [3]. It has been successfully used to mine association between items in many problems such as market basket analysis [27], intrusion detection [22], supply chain management [1] and requirement engineering [4]. The Apriori algorithm first scans the dataset (i.e., transactions) and generates frequent itemsets based on filtering criteria set by users. Then, a list of association rules is generated from the frequent itemsets.

We use the support [2], confidence [2], lift [9], leverage [52] and conviction [9] parameters to quantify the degree of association between two items (or smells). The range of values for support and confidence is between 0 and 1. Lift can take any value between 0 and ∞. If the value of lift is 1, it means that the smell pairs are independent. Leverage has a range between -1 and 1. A leverage value of zero shows independence. Conviction has a range of 0 and ∞. Independent occurrences have a conviction of 1.

2.3 Cramer's V Test for Association

The Cramer's V test measures the level of association between categorical variables [12]. It has a value between 0 and 1. A value of 0 indicates complete independence, and a value of 1 indicates complete association. The Cramer's V test takes into account sample size when comparing two variables. The formula is given in Equation 1 where X^2 is the Pearson's Chi-square coefficient, n is the total number of samples and *row* and *col* represent the number of distinct values of the categorical variables whose association is to be computed.

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

2.4 Survival Analysis

Survival analysis [39] is a statistical analysis technique that provides the expected time of the occurrence of an event of interest. 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.

Time to event (T) is defined as the time interval between the starting of observation and the occurrence of an event or the censoring of data. This time can be measured in any unit. Thus, T is a random variable with positive values [39]. **Status** is a boolean variable that indicates whether an event is observed or the data is censored. If the event occurs during the observation period, status takes a value of 1 and otherwise 0. **The Survival function S(t)** gives the probability ($P(T > t)$) that a subject will survive beyond time t. After we arrange our data in increasing order of T, we can plot the survival curve and estimate the survival probability using one of the commonly used survival estimators (e.g., Kaplan-Meier estimator [25]). The Kaplan-Meier estimation is computed following Equation 2, where t_i is the time duration up to event-occurrence point i, d_i is the number of event occurrences up to t_i, and n_i is the number of subjects that survive just before t_i. n_i and d_i are obtained from the aforementioned ordered data.

$$S(t) = \prod_{i:t_i \le t} [1 - \frac{d_i}{n_i}] \quad (2)$$

3 STUDY METHOD

In this section, we describe our study method where we select appropriate projects for our study, detect traditional and SQL code smells within their source code, extract bug-fixing and bug-inducing commits from their version history, and then analyse all these items to answer our research questions. Figure 1 shows the process of our empirical study.

3.1 Project Selection

We limited our study to Java because the SQL code smell detection tool we selected for our study, SQLInspect [44] can only process programs written in Java. We select our projects from GitHub using four steps as follows.

Phase-I: We use GitHub search mechanism and collect the software repositories labelled 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.

Table 1: Selected projects & their database access statistics

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

DAQC = Database Access Query Count

Phase-II: We performed code search on each project selected in the first phase using GitHub code search API [21]. 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 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 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 where 11% have sub-queries.

We also classify our selected projects 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 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 of all selected projects.

3.2 Code Smell Detection

It is not practical to detect code smells from every commit of each project due to the large number of projects and commits. Therefore,

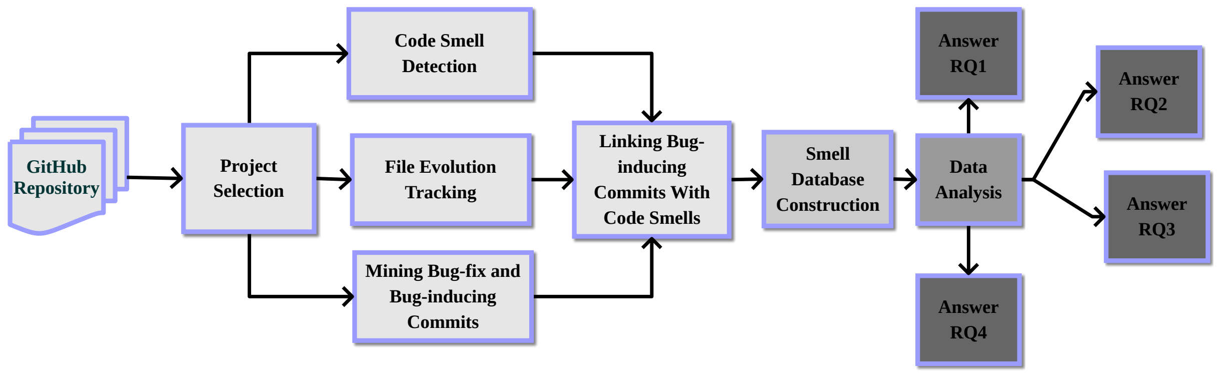

Figure 1: Process followed to conduct our empirical study

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 backwards. A similar approach was followed by Aniche et al. [5].

We use SQLInspect [44], 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 – 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*.

We use DECOR [17], 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% [41].

3.3 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 of file status can help us resolve the 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 on 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 [23]. Finally, each source file in each of our projects is tagged with a unique identifier generated from the file tracking information.

3.4 Mining Bug-Fix and Bug-Inducing Commits

We use PyDriller [60] 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 were selected based on the work of Mockus and Votta [40] and Antoniol et al. [6], who showed

Table 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**

that those keywords have a tendency to be associated with bug-fix commits. These keywords were also used in multiple previous studies to identify bug-fixing commits [18, 24, 32]. The complete keyword list is available in the replication package [42]. Our tool searches for each keyword in the commit messages, and separates the commits containing the keywords as bug-fixing commits.

Table 2 shows the proportion of bug-fix commits that are identified using the top six prevalent keywords.

PyDriller implements the SZZ algorithm [59] 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.

3.5 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 actually 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.

3.6 Construction of a Smell Database

In order 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 of a vector that stores the statistics on traditional code smells, SQL code smells found within a source code file and its bug-inducing related meta data. 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.

3.7 Experimental Data Analysis

Association between SQL and Traditional Code Smells: For Apriori analysis, we consider each entry (i.e., a record from our database that has at least one database access query) containing code smell statistics as a transaction. Then, the frequent itemsets are 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 to collect numerical, comparable association values between these two classes of code smells (RQ2).

Co-occurrence between SQL Code Smells and Bugs: To investigate the co-occurrence (or potential causation) between SQL code smells and software bugs (RQ3), we employ both Chi-squared test and Cramer's V test. We also develop a RandomForest model to investigate the importance of various code smells in determining co-occurrence with bugs.

Survival Analysis of Code Smells: We analyze the survival rates of traditional and SQL code smells during the evolution of our selected systems (RQ4). For survival analysis, we use the Kaplan-Meier curve [25] (e.g., Fig. 4). 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 corresponding code smell.

3.8 Replication Package

We made our collected data and results publicly available [42]. We provide (i) the database of smells in the projects under question, (ii) the list of keywords used for the identification of bug-fixing commits, (iii) data collection and analysis scripts.

4 STUDY RESULTS

In this section, we present our study findings and answer each of the four research questions as follows.

4.1 RQ1: What is the Prevalence of SQL Code Smells Across Different Application Domains?

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. Our projects from each domain have varying complexity in terms of project size and interactions with the database. We determine *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. We present our prevalence analysis in Figures 2, 3 and Table 3.

Table 3: 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%

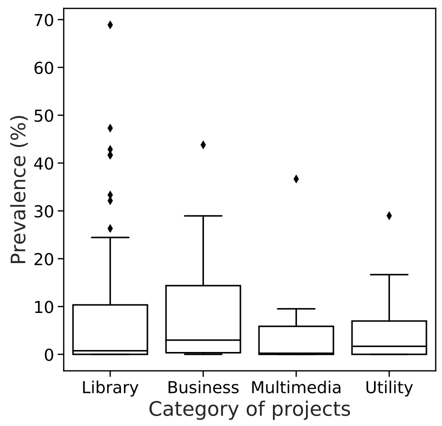

Figure 2: Prevalence of SQL code smells (Implicit Columns) across different application domains

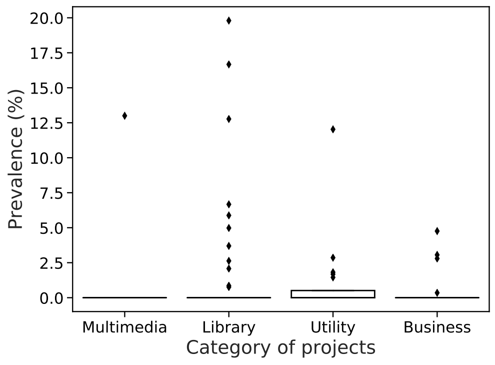

Figure 3: Prevalence of SQL code smells (Fear of the Unknown) across different application domains

We detect four types of SQL code smells (e.g., Section 2.1) 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, two queries are affected by this smell. 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.

We analyse the prevalence of *Implicit Columns* across four application domains. Table 3 and Fig. 2 summarize our findings. From Table 3, we see that projects from *Business* and *Utility* domains have the highest median prevalence of 2.98% and 1.68%. Fig. 2 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. 2. 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 2 and Table 3, we see that SQL code smells such as *Implicit Columns* have the least prevalence in the subject systems from *Multimedia* domain.

We further analyse the distribution of prevalence for *Fear of the Unknown* SQL code smell across the four application domains. Fig. 3 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 analyse. 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. We also randomly selected 10 projects and manually investigated 98 *Implicit Columns* 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.

***Implicit Columns* smells are the most prevalent SQL code smells 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.**

4.2 RQ2: Do Traditional Code Smells and SQL Code Smells Co-occur at Class Level?

We determine co-occurrences between SQL code smells and traditional code smells within our subject systems where we consider multiple versions of the source code files (a.k.a., revisions). Table 4 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.

Table 4: Source code file versions with database access

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

We use Apriori algorithm for determining the association (co-occurrence) between traditional code smells and SQL code smells. To generate frequent itemsets, we selected a minimum support of 0.01 (1%) considering the 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 itemset 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 5 shows our frequent itemsets where each itemset comprises 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.

We also conduct Chi-squared and Cramer's V tests to check whether the associations between traditional code smells and SQL code smells (e.g., Table 5) are statistically significant or not. Table 6 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 6, 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 6, we have weak evidence to reject the null hypothesis, i.e., such code smell pairs might not be associated.

Table 5: 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 Pais	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

Table 6: Chi-square and Cramer's V value of smell pairs computed on the combined dataset for each smell pair in Table 5. 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

We also further investigate 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 6 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.

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 and our extensive analysis.

4.3 RQ3: Do the SQL Code Smells Co-occur with Software Bugs?

We determine the association between SQL code smells and software bugs by analysing 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. Table 7 shows our investigation details.

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 7, 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.

Although the *Implicit Columns* smell is known to cause performance issues and software bugs [26, 43], 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.

Table 7: 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**

ComplexClass was also reported to be associated with software bugs by the earlier studies [33, 63].

We also determine the degree of association between any code smells and software bugs using Cramer's V test. As shown in Table 7, 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.

We also develop 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 [11] 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. The last column of Table 7 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.

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.

4.4 RQ4: How Long do the SQL Code Smells Survive?

We perform survival analysis [39] to determine how long the SQL code smells survive throughout the life cycle of a subject system. Fig. 4 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.

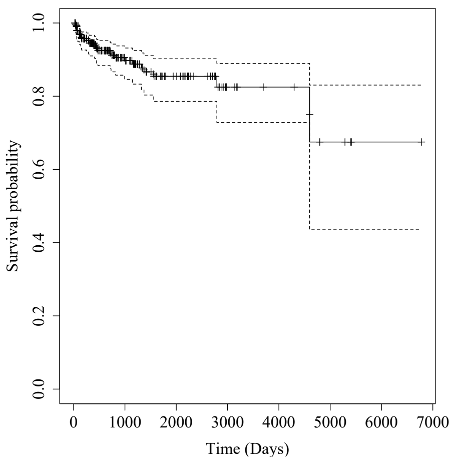

Figure 4: Kaplan-Meier survival curve for *Implicit Columns* SQL code smell. The X-axis is the time in days and the vertical axis shows the survival probability value. The Censoring time and the Confidence interval are marked in the plot.

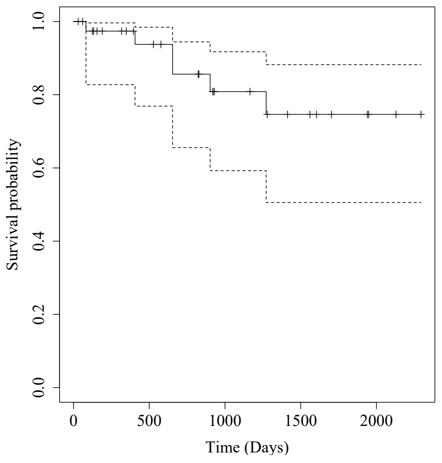

Figure 5: Kaplan-Meier survival curve for *Fear of the Unknown* SQL code smell. The X-axis is the time in days and the vertical axis shows the survival probability value. The Censoring time and the Confidence interval are marked in the plot.

Fig. 5 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 small number of instances of this smell in the dataset.

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. 6 shows our comparative analysis between traditional and SQL code smells. We see that SQL code smells have 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 longer time duration.

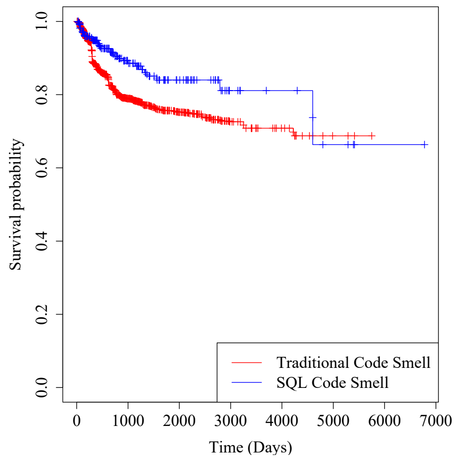

Figure 6: Kaplan-Meier survival curve for traditional code smells and SQL code smells. The Censoring time for censored files is marked in the plot.

We performed *Logrank test* [51] to determine whether the difference between these two survival curves in Fig. 6 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 large number of 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.

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 *LongParameterList* and 65% of source code files with the *LongMethod* contain those traditional code smells in all snapshots. This confirms the observation that a large number of 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.

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.

5 IMPLICATION OF FINDINGS

The result of RQ1 shows that not all SQL code smells are equally prevalent in data-intensive projects. 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 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 and potentially detecting the occurrence of SQL code smells given some traditional code smells or vice versa.

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.

The result of RQ4 shows that little attention is given to SQL code smells. 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.

6 THREATS TO VALIDITY

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 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 Johannes et al. [23]). To link bugs with file versions, 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 [54]. 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.

Threats to internal validity: We did not claim any causation as this is an exploratory study. We only discussed the co-occurrence or association. Hence, our study is not subjected to threats to internal validity.

Threats to conclusion validity: To avoid conclusion threats to validity we only used non-parametric statistical tests.

Threats to external validity: To make our findings generalizable, 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 is generalizable to any programming language. It is desirable to study if our conclusions can be extended to different programming languages.

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 [42].

7 RELATED WORK

We discuss the state of the art from two perspectives. First, we overview the related work on empirical studies about traditional code smells. Then, we discuss the state of the art research on SQL code or database-related smells.

Traditional Code Smells: Since the initial introduction of the term "design flaws" [53] and "code smell" [15], 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. [55] 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. Important to mention here are *correlation studies* on code smells and quality attribute such as number of bugs or number of modifications in classes [14, 33, 36, 62, 63] and empirical studies on the evolution of code smells [23, 29, 45, 48, 50, 57], their influence on defects [46], maintenance effort [58] or how humans perceive them [37, 49].

For the detection of traditional code smells, we rely on the DECOR tool [17]. However, different traditional smell detection tools are also available. For example, an artificial immune system-based smell detection tool was developed by Hassaine et al. [19], a Bayesian network-based expert system by Khomh et al. [30], and a textual data mining approach by Palomba et al. [47]. Kessentini and Ouni proposed an approach to automatically generate smell detection rules using genetic algorithm [28]. Another popular traditional smell detection tool is JDeodorant developed by Tsantalis [61]. JDeodorant focuses on the detection and refactoring of Feature Envy, God Class and Duplicated Code, Type Checking and Long Method smells. We choose DECOR because it covers more traditional code smells and was reported to reach a 100% recall.

SQL Code Smells: Although researchers studied common errors in SQL queries before [8, 16], the book of Karwin [26] is the first to present SQL antipatterns in a comprehensive catalogue. This catalogue inspired researchers to further investigate such smells. Khumnin et al. [31] present a tool for detecting logical database design antipatterns in Transact-SQL queries. Nagy and Cleve [43] propose a static analysis approach to detect SQL code smells in queries extracted from Java code. They also provide additional analyses (e.g., metrics) about the detected smells [44]. Another tool, DbDeo [56], implements the detection of *database schema* smells. DbDeo is evaluated on 2925 open-source repositories; their authors identified 13 different types of smells, among which 'index abuse' was found to be the most prevalent one. In another recent work, De Almeida Filho et al. [13] investigate the prevalence and co-occurrence of SQL code smells in PL/SQL projects. Arzamasova et al. propose to detect antipatterns in SQL logs [7] and demonstrate their approach through the refactoring of a project containing more than 40 millions of queries. Let us also mention the work by Burzanska et al. [10], who question whether the 'Poor Man's Search Engine' smell should still be considered as a poor practice today, as relational databases evolved since Karwin's catalogue.

There exist several other SQL analysis and smell detection tools, including TOAD[2], SQL Prompt[3], and SQL Enlight[4]. However, those tools require a set of SQL queries as input, and they cannot analyze the queries embedded in source code. SQLInspect can extract the queries and detect SQL code smells given the project source code, which justifies our choice.

Summary: In contrast with the studies discussed above, this constitutes – to the best of our knowledge – the first empirical study investigating the prevalence of SQL code smells, their association with bugs and with other traditional code smells, as well as their evolution over time. We expect this study to serve as a baseline for further studies on the impact and persistence of SQL code smells in data-intensive systems.

8 CONCLUSION AND FUTURE WORK

In this study, we investigated the prevalence of SQL code smells and their association with bugs and 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 linked bug-inducing commits to those smells using the SZZ algorithm to study their association with bugs. We 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. 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 from 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 in order to avoid potential harm.

Our study is exploratory in nature. We believe that further investigation is needed to better understand the consequences of SQL code smells. This includes, in particular, their impact on the performance and the maintainability of data-intensive systems.

Acknowledgements: This research was partly supported by the Excellence of Science project 30446992 SECO-ASSIST, funded by the F.R.S.-FNRS, FWO and Natural Sciences and Engineering Research Council of Canada (NSERC).

[2]http://www.toadworld.com/
[3]https://www.red-gate.com/hub/product-learning/sql-prompt
[4]https://sqlenlight.com/

REFERENCES

[1] R Agarwal. 2017. Decision making with association rule mining and clustering in supply chains. *International Journal of Data and Network Science* 1, 1 (2017), 11–18.

[2] R Agrawal, T Imielinski, and A Swami. 1993. Mining associations between sets of items in large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 207–216.

[3] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, Vol. 1215. 487–499.

[4] Shadi AlZu'bi, Bilal Hawashin, Mohammad EIBes, and Mahmoud Al-Ayyoub. 2018. A novel recommender system based on apriori algorithm for requirements engineering. In *2018 Fifth International Conference on Social Networks Analysis, Management and Security (SNAMS)*. IEEE, 323–327.

[5] Maurício Aniche, Gabriele Bavota, Christoph Treude, Marco Aurélio Gerosa, and Arie van Deursen. 2018. Code smells for model-view-controller architectures. *Empirical Software Engineering* 23, 4 (2018), 2121–2157.

[6] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is it a bug or an enhancement? A text-based approach to classify change requests. In *CASCON*, Vol. 8. 304–318.

[7] N. Arzamasova, M. Schäler, and K. Böhm. 2018. Cleaning Antipatterns in an SQL Query Log. *IEEE Transactions on Knowledge and Data Engineering* 30, 3 (March 2018), 421–434.

[8] S. Brass and C. Goldberg. 2004. Semantic errors in SQL queries: a quite complete list. In *Fourth International Conference on Quality Software, 2004. QSIC 2004. Proceedings.* 250–257.

[9] Sergey Brin, Rajeev Motwani, Jeffrey D Ullman, and Shalom Tsur. 1997. Dynamic itemset counting and implication rules for market basket data. *Acm Sigmod Record* 26, 2 (1997), 255–264.

[10] Marta Burzańska and Piotr Wiśniewski. 2018. How Poor Is the "Poor Man's Search Engine"?. In *Beyond Databases, Architectures and Structures. Facing the Challenges of Data Proliferation and Growing Variety*, Stanisław Kozielski, Dariusz Mrozek, Paweł Kasprowski, Bożena Małysiak-Mrozek, and Daniel Kostrzewa (Eds.). Springer International Publishing, Cham, 294–305.

[11] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research* 16 (2002), 321–357.

[12] Harald Cramer. 1946. Mathematical methods of statistics. *Princeton U. Press, Princeton* (1946), 500.

[13] Francisco Gonçalves de Almeida Filho, Antônio Diogo Forte Martins, Tiago da Silva Vinuto, José Maria Monteiro, Ítalo Pereira de Sousa, Javam de Castro Machado, and Lincoln Souza Rocha. 2019. Prevalence of bad smells in PL/SQL projects. In *Proceedings of the 27th International Conference on Program Comprehension.* IEEE Press, 116–121.

[14] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka. 2013. 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.* 260–269.

[15] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, and Erich Gamma. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., USA.

[16] Christian Goldberg. 2009. Do You Know SQL? About Semantic Errors in Database Queries. In *In 7th Workshop on Teaching, Learning and Assessment in Databases.* 13–19.

[17] Yann-Gaël Guéhéneuc. 2007. Ptidej: A flexible reverse engineering tool suite. In *2007 IEEE International Conference on Software Maintenance.* IEEE, 529–530.

[18] Latifa Guerrouj, Zeinab Kermansaravi, Venera Arnaoudova, Benjamin CM Fung, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2017. Investigating the relation between lexical smells and change-and fault-proneness: an empirical study. *Software Quality Journal* 25, 3 (2017), 641–670.

[19] Salima Hassaine, Foutse Khomh, Yann-Gaël Guéhéneuc, and Sylvie Hamel. 2010. 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, 343–348.

[20] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2016. An empirical study of the performance impacts of android code smells. In *Proceedings of the International Conference on Mobile Software Engineering and Systems.* ACM, 59–69.

[21] GitHub Inc. 2019. Search. Retrieved December 28, 2019 from https://developer.github.com/v3/search/

[22] Zichuan Jin, Yanpeng Cui, and Zheng Yan. 2019. Survey of Intrusion Detection Methods Based on Data Mining Algorithms. In *Proceedings of the 2019 International Conference on Big Data Engineering.* ACM, 98–106.

[23] David Johannes, Foutse Khomh, and Giuliano Antoniol. 2019. A large-scale empirical study of code smells in JavaScript projects. *Software Quality Journal* (2019), 1–44.

[24] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[25] Edward L Kaplan and Paul Meier. 1958. Nonparametric estimation from incomplete observations. *Journal of the American statistical association* 53, 282 (1958), 457–481.

[26] Bill Karwin. 2010. *SQL Antipatterns: Avoiding the pitfalls of database programming.* Pragmatic Bookshelf.

[27] Manpreet Kaur and Shivani Kang. 2016. Market Basket Analysis: Identify the changing trends of market data using association rule mining. *Procedia computer science* 85 (2016), 78–85.

[28] Marouane Kessentini and Ali Ouni. 2017. Detecting android smells using multi-objective genetic programming. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems.* IEEE Press, 122–132.

[29] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. 2009. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering.* IEEE, 75–84.

[30] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software* 84, 4 (2011), 559–572.

[31] P. Khumnin and T. Senivongse. 2017. SQL antipatterns detection and database refactoring process. In *2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD).* 199–205.

[32] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[33] Wei Li and Raed Shatnawi. 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software* 80, 7 (2007), 1120–1128.

[34] Red Gate Software Ltd. 2014. 119 SQL Code Smells.

[35] Y. Lyu, A. Alotaibi, and W. G. J. Halfond. 2019. Quantifying the Performance Impact of SQL Antipatterns on Mobile Applications. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 53–64.

[36] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. 2012. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In *2012 16th European Conference on Software Maintenance and Reengineering.* 277–286.

[37] Mika V. Mäntylä and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering* 11, 3 (01 Sep 2006), 395–431.

[38] Loup Meurice, Csaba Nagy, and Anthony Cleve. 2016. Static analysis of dynamic database usage in Java systems. In *International Conference on Advanced Information Systems Engineering.* Springer, 491–506.

[39] Rupert G Miller Jr. 2011. *Survival analysis.* Vol. 66. John Wiley & Sons.

[40] Audris Mockus and Lawrence G Votta. 2000. Identifying Reasons for Software Changes using Historic Databases.. In *Proc. of the 2000 International Conference on Software Maintenance.* 120–130.

[41] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. 2010. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing* 22, 3-4 (2010), 345–361.

[42] Biruk Asmare Muse. 2020. Replication package. https://github.com/Biruk-Asmare/MSR_2020_SQLSmells_Prevalence

[43] Csaba Nagy and Anthony Cleve. 2017. A static code smell detector for SQL queries embedded in Java code. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM).* IEEE, 147–152.

[44] Csaba Nagy and Anthony Cleve. 2018. SQLInspect: A static analyzer to inspect database usage in Java applications. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings.* ACM, 93–96.

[45] Steffen Olbrich, Daniela S Cruzes, Victor Basili, and Nico Zazworka. 2009. 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, 390–400.

[46] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *Proc. of the 2010 IEEE International Conference on Software Maintenance.* 1–10.

[47] Fabio Palomba. 2015. Textual analysis for code smell detection. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2.* IEEE Press, 769–771.

[48] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.

[49] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *2014 IEEE International Conference on Software Maintenance and Evolution.* 101–110.

[50] Ralph Peters and Andy Zaidman. 2012. Evaluating the lifespan of code smells using software repository mining. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 411–416.

[51] Richard Peto and Julian Peto. 1972. Asymptotically efficient rank invariant test procedures. *Journal of the Royal Statistical Society: Series A (General)* 135, 2 (1972), 185–198.

[52] Gregory Piatetsky-Shapiro. 1991. Discovery, analysis, and presentation of strong rules. *Knowledge discovery in databases* (1991), 229–238.

[53] Arthur J. Riel. 1996. *Object-Oriented Design Heuristics* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[54] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M González-Barahona. 2018. 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* 99 (2018), 164–176.

[55] José Amancio M. Santos, João B. Rocha-Junior, Luciana Carla Lins Prates, Rogeres Santos do Nascimento, Mydiã Falcão Freitas, and Manoel Gomes de Mendonça. 2018. A systematic review on the code smell effect. *Journal of Systems and Software* 144 (2018), 450 – 477.

[56] T. Sharma, M. Fragkoulis, S. Rizou, M. Bruntink, and D. Spinellis. 2018. 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)*. 55–64.

[57] Raed Shatnawi and Wei Li. 2006. An investigation of bad smells in object-oriented design. In *Third International Conference on Information Technology: New Generations (ITNG'06)*. IEEE, 161–165.

[58] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering* 39, 8 (Aug 2013), 1144–1156.

[59] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *ACM sigsoft software engineering notes*, Vol. 30. ACM, 1–5.

[60] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. 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)*. 908–911.

[61] Nikolaos Tsantalis. 2010. 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).

[62] Aiko Yamashita and Leon Moonen. 2013. Exploring the Impact of Inter-Smell Relations on Software Maintainability: An Empirical Study. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, 682–691.

[63] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 17–23.