Smelly Relations: Measuring and Understanding Database Schema Quality

Tushar Sharma
Athens University of Economics and Business, Athens, Greece
tushar@aueb.gr

Marios Fragkoulis
Athens University of Economics and Business, Athens, Greece
mfg@aueb.gr

Stamatia Rizou
Singular Logic
Athens, Greece
srizou@singularlogic.eu

Magiel Bruntink
Software Improvement Group
Amsterdam, The Netherlands
m.bruntink@sig.eu

Diomidis Spinellis
Athens University of Economics and Business, Athens, Greece
dds@aueb.gr

ABSTRACT

Context: Databases are an integral element of enterprise applications. Similarly to code, database schemas are also prone to smells — best practice violations.
Objective: We aim to explore database schema quality, associated characteristics and their relationships with other software artifacts.
Method: We present a catalog of 13 database schema smells and elicit developers' perspective through a survey. We extract embedded SQL statements and identify database schema smells by employing the DbDeo tool which we developed. We analyze 2925 production-quality systems (357 industrial and 2568 well-engineered open-source projects) and empirically study quality characteristics of their database schemas. In total, we analyze 629 million lines of code containing more than 393 thousand SQL statements.
Results: We find that the *index abuse* smell occurs most frequently in database code, that the use of an ORM framework doesn't immune the application from database smells, and that some database smells, such as *adjacency list*, are more prone to occur in industrial projects compared to open-source projects. Our co-occurrence analysis shows that whenever the *clone table* smell in industrial projects and the *values in attribute definition* smell in open-source projects get spotted, it is very likely to find other database smells in the project.
Conclusion: The awareness and knowledge of database smells are crucial for developing high-quality software systems and can be enhanced by the adoption of better tools helping developers to identify database smells early.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; *Software maintenance tools*;

KEYWORDS

Database schema smells, Code smells, Antipatterns, Software quality, Software maintenance

ACM Reference Format:
Tushar Sharma, Marios Fragkoulis, Stamatia Rizou, Magiel Bruntink, and Diomidis Spinellis. 2018. Smelly Relations: Measuring and Understanding Database Schema Quality. In *Proceedings of International Conference on Software Engineering (ICSE).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Databases are an integral element of enterprise applications. The effective use of database affects vital quality parameters, such as performance and maintainability, of these applications.

Code smells [10, 28, 30] indicate the presence of quality problems in a software system. The smell metaphor has been extended to other similar domains such as configuration management [26], spreadsheets [13], and presentations [25]. Typically, the use of a database in a software system manifests itself as a series of DDL (Data Definition Language — e.g. CREATE TABLE) or DML (Data Manipulation Language — e.g. SELECT) SQL statements. Similar to code, these SQL statements can also indicate smells. Bill Karwin [14] documents a catalog of database anti-patterns. However, their presence in software systems and their relationships with other software artifacts have not been explored yet.

In this context, we present our study on mining database smells in production-quality systems including both the industrial as well as the open-source software. We analyze SQL statements to measure schema quality of relational databases with a focus on performance and maintainability quality attributes. Specifically, we explore occurrence patterns of database schema smells and figure out the degree of co-occurrence among schema smells. We also study the factors that affect the density of database smells. These factors are the size of the project and database and the nature of code.

To study these aspects, we compiled a catalog of 13 database schema smells. We carried out a developers survey to understand software developers' perspective on these schema smells. We developed a tool viz. *DbDeo* to extract embedded SQL statements from host source code (in which the SQL statements are embedded) and to identify cataloged database smells. We analyzed 357 industrial and 2568 open-source projects containing SQL statements and provide

ICSE, May 2018, Gothenberg, Sweden

ACM ISBN 123-4567-24-567/08/06...$15.00
https://doi.org/10.475/123_4

empirical answers to each of the posed research questions. Some key observations from the study are listed below.

- In our developers survey, the responses emphasize that the awareness and knowledge of database smells is crucial for software developers to develop high quality software. Further, the survey reveals the subjectivity involved in the database smell detection and interpretation.
- The smell *index abuse* is the most frequently occurring database smell in both industrial and open-source projects. A high number of *index abuse* smell instances indicate the need of an effective index management in database systems for higher performance and maintainability.
- Some database smells, such as *adjacency list*, are more prone to occur in industrial projects compared to open-source projects by a significant margin.
- The size of the host application has no impact on density of database smells; however, smell density shows positive correlation with the size of the database. Application type (*i.e., Mobile*, *Desktop*, or *Web*) doesn't have a significant impact on database smell density.
- Use of an ORM framework doesn't prevent database smells.

Studying database code quality can help us understand the usage characteristics of database code. The study offers contributions to both research and practice. For researchers, it provides a method to investigate code quality of embedded SQL statements by mining repositories. The method also outlines the challenges involved (such as extracting embedded SQL statements). At the same time, practitioners can learn the potential quality issues that may arise in their database schema to avoid them. Furthermore, practitioners can identify database schema smells using the open-source tool developed in this study. Finally, our results pinpoint areas where improvements in database APIs, tool support, and training can increase the quality of database schemas.

2 RESEARCH OBJECTIVE

The goal of this study is to understand database code quality by mining database schema smells and explore their relationship with other software artifacts. The chosen subject systems are a wide variety of industrial as well as open-source software systems. We keep the focus of the study on performance and maintainability quality attributes of relational database code.

Characteristics of smells, such as frequency (or the occurrence pattern) of smells [1, 13, 17], provide dimensions of prioritization and refactoring. Similarly, relationships of smells with domains, frameworks, and other application characteristics [9, 17], help us understand the interplay of smells with application characteristics. In the context of database programming, ORM (Object-Relational Mapping) frameworks simplify database access by providing an abstraction. However, it is not understood whether the usage of an ORM framework in an application will lead us to fewer number of smells. Further, studying co-occurrence of database schema smells will complement the existing studies exploring properties of co-occurrence among smells [19, 26]. With this background, we explore the following research questions.

RQ1. *What are the occurrence patterns of database smells?* We examine the distribution of database smells to find out whether there exists a set of database smells that occurs more frequently in general than another set of database smells.

RQ2. *Does the size of the project or the database play a role in smell density?* Smell density [26] is a normalized metric defined as average number of smells occurred per a fixed number of lines (say, a thousand lines) of code in a project. We investigate the relationship of the size of the project (both the total lines of code as well as total number of CREATE TABLE statements) and smell density.

RQ3. *Does the nature of code (type of the application, or usage of ORM frameworks) affect the smell density?* The usage of an ORM framework makes it easier to work with a database. We explore whether the usage of ORM frameworks and the type of the application influence database smell density.

RQ4. *What is the degree of co-occurrence among database smells?* Patterns and smells tend to occur together [3, 30]. We examine the degree of co-occurrence among database smells to find out a set of database smells that is likely to occur when a database smell gets detected.

We present a catalog of database schema smells as a theoretical model to study the above stated research questions. We attempt to understand developers' perspective on database schema smells through an online survey. We developed a tool viz. *DbDeo* to detect 9 smells belonging to the presented catalog. We extracted SQL statements from 2925 repositories, analyzed them with our tool, and documented our quantitative and qualitative observations.

3 DATABASE SMELLS

We define database smells as follows:

Database smells are the characteristics of database code (either DDL *or* DML SQL *statements), database system, or stored data that indicate violation of the recommended best practices and potentially affect the quality of the software system in a negative way.*

We categorize database smells in three categories to understand them better.

Schema smells: Smells that arise due to poor schema design are classified as database schema smells. Smells presented in this section such as *compound attribute*, *index abuse*, and *god table* are examples of database schema smells.

Query smells: Smells arising from poorly written SQL queries are specified as database query smells. *Misused null* [14] (when null is used as an ordinary value in SQL queries) and *non-grouped column reference* [14] (when a query references at least one non-grouped column in the presence of *group by* clause) are examples of database query smells.

Data smells: Data smells arise from poor data handling in databases. *Intermingled data types* (where numbers and alphabets are intermingled leading to confusion and subtle bugs; for instance, using 'O' instead of '0' in 7O34) is an example of data smells.

In this paper, we focus only on database schema smells.

3.1 A Catalog of Database Schema Smells

We carried out a comprehensive exploration of resources that discuss best practices as well as common database smells or antipatterns. We studied wide variety of resources including books [14], research literature [4, 8, 23], industrial white-paper [24], and discussions on question-answer sites [7]. We summarize the result of our exploration in the form of a catalog of database schema smells.

CA: Compound attribute This smell arises when a column is used to store a non-atomic attribute. For instance, storing comma-separated lists for an attribute to avoid creating an intersection table for a many-to-many relationship [14, 24] or storing a JSON file which is not used atomically [7].
Rationale: Each attribute value must be stored and retrieved atomically. If a table does not adhere to this practice, the resultant schema introduces multiple problems. For instance, a user has to write more complex queries (using pattern-matching expressions) to retrieve data from this table. Such complex queries are prone to inaccurate results. Also, such queries cannot exploit available indexes. Even further, these queries are not portable due to vendor specific support to pattern-matching expressions.

AL: Adjacency list The smell occurs when an attribute in a table refers another row in the same table *i.e.,* a table has a recursive relationship to model hierarchical structure [14, 24].
Rationale: Querying a tree with adjacency list is quite difficult and error-prone. Specifically, deleting a node from a tree which is modelled using adjacency list is non-trivial and prone to introduce errors in the database.

SK: Superfluous key This smell arises when an unnecessary superfluous pseudo key is defined in a table where other attribute(s) in the table may serve as a primary key [14].
Rationale: Choosing an appropriate primary key is an essential requirement for a table. A pseudo key could be defined when the present set of attributes could not serve as a primary key. However, a pseudo key is unnecessary and even erroneous (leads to duplicate rows) when the existing set of attributes of the table could be used as a primary key.

MC: Missing constraints This smell arises when constraints for a foreign key are missing from a schema definition [14, 24].
Rationale: Referential integrity is an essential property of relational databases. Values referenced in a foreign key column must exist in the columns of primary or unique keys of the parent table. It can be easily achieved by defining constraints on foreign keys. However, when such constraints are missing for a foreign key it leads to compromized referential integrity of the database.

MD: Metadata as data This smell occurs when metadata is stored as data in the form of EAV (Entity-Attribute-Value) pattern [14, 24].
Rationale: In a relational table, all the attributes are equally applicable for all the rows in the table. It is tempting to implement EAV pattern when a subset of attributes applicable for a subset of rows and the rest of attributes for rest of the rows. However, this arrangement introduces many deficiencies in the database; for example, one can't use native SQL data types (leading to invalid data), enforce referential integrity, or make up attribute names.

PA: Polymorphic association This smell occurs when a table uses a multi-purpose foreign key [14, 24].
Rationale: Relational database schema does not allow us to declare polymorphic association. However, many times developers define an additional column in a table as a tag to realize a polymorphic association. This arrangement makes it difficult to query the table and compromises readability and understandability.

MA: Multicolumn attribute This smell arises when multiple serial columns are created for an attribute [7, 14].
Rationale: In cases when an attribute may have one or more values, it is tempting to create multiple columns for the attribute in a table. However, such a schema design makes querying the table very difficult and verbose.

CT: Clone tables This smell occurs when a table is split horizontally in multiple tables using some criterion (for example, year) to achieve scalability [14].
Rationale: This smell not only makes the querying difficult but also introduces problems managing data integrity.

VA: Values in attribute definition This smell arises when specific values are defined in an attribute definition to restrict possible values of the attribute [14].
Rationale: Specifying all possible values for an attribute in schema definition mixes metadata with data which is not recommended. This smell makes it difficult to extend or modify the list of accepted values for an attribute.

IA: Index abuse This smell arises when the indexes are used poorly [14, 24]. This smell has the following variants: 1) Missing indexes 2) Insufficient indexes (indexes must be prepared at least for primary and foreign keys), and 3) Unused indexes
Rationale: Creating effective indexes is not trivial; it requires judicious planning. A database with a deficient plan for indexes performs poorly.

GT: God table This smell arises when a table contains excessive number of attributes [7, 24].
Rationale: Excessive number of attributes tend to violate the principles of normalization which in turn introduce a variety of problems. Additionally, it impacts maintainability of the database.

MN: Meaningless name This smell occurs when a table or an attribute name is cryptic or meaningless [7].
Rationale: Meaningless or cryptic names hamper readability of the database's schema.

OA: Overloaded attribute names This smell occurs when two or more attributes are defined with identical names but as distinct data types in different tables [24].
Rationale: Identical names with different data types create confusion and could lead to subtle bugs in queries.

4 DEVELOPERS' SURVEY ON DATABASE SMELLS

We carried out an online survey targeting software developers to understand their perspective about the significance of various database schema smells. We divided the survey in three sections. In the first section, we collected information about participants' experience. In the second section, we asked the participants to read the description of each potential smell presented (total 13 questions based on the catalog presented in Section 3.1) and to rate each of them based on their *importance* (*i.e.,* the degree of smell's association with software quality issues), and *usefulness* (*i.e.,* the degree of accuracy of the smell in predicting software quality issues). All the questions in this section were Likert scale questions. We asked the respondents whether they consider the presented practice as a database schema smell, a recommended practice, both a smell and a recommended practice depending on the context, or neither a smell nor a recommended practice. The third section presented a couple of open-ended questions to get participants' view on the presented catalog and missing database schema smells. The questionnaire that we used is available online.[1]

We ran a pilot for the survey, collected the feedback, and improved the survey. We shared the survey to all online social media channels and sought participation from the developer community. We received 52 complete responses with completion rate 38%.

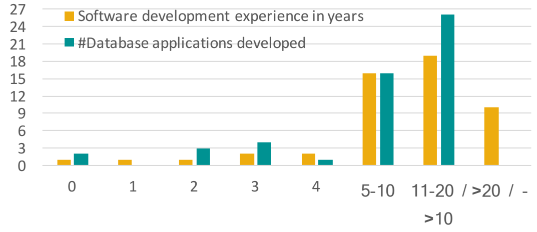

Figure 1: Experience of respondents in terms of number of years as well as the number of database applications developed by them

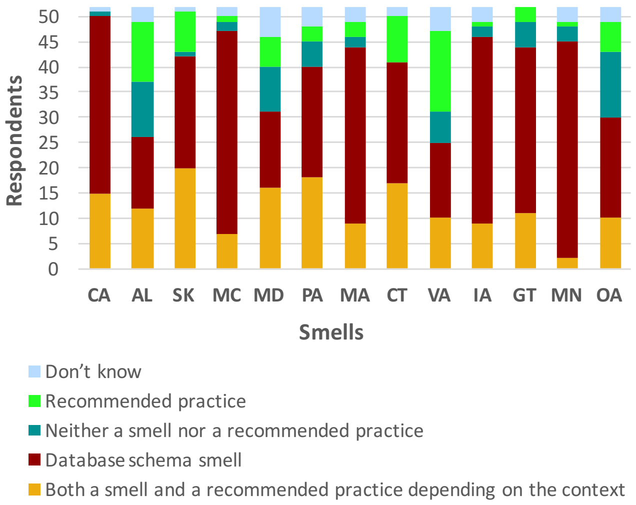

Figure 2: Respondents' perspective of considered database smells

Most of the respondents belong to experienced developer groups. Figure 1 shows the distribution of respondents' experience in terms of number of years and number of database applications they have developed. We summarize our findings from the survey below.

A large majority of 88% agrees (42% strongly agree and 46% agree) that the awareness and knowledge of database smells is crucial for software developers to develop high quality applications. None of the respondents marked disagree or strongly disagree options.

Figure 2 shows a consolidated perspective provided by the respondents for section 2 questions. Based on the responses we infer that some practices, such as *meaningless name* (83%) and *missing constraints* (77%), are clearly marked as database smells. However, we found that practices such as *values in attribute definition* and *adjacency list* are more context-sensitive.

The respondents had the option to add their views either in terms of smells that we have not included but they have seen in practice as well as their feeling, objection, or reservation on the presented smells. A few respondents underline the subjectivity involved in database smells detection. For instance, one respondent said that *"... database smells in general depend much more on an assessment of the need and end use of data... "*. Similarly, another respondent shared an instance of duplicating values in a table (which is a smell) to avoid querying 60 tables to load a single record. Yet another respondent provided his/her opinion on *index abuse* smell: *"... the proper use of indexes is dependent on many things and without regular profiling it's not possible to decide whether indexes are actually being misused."*

As a conclusion of our survey, developers seem to acknowledge the need for detecting database smells. However, their systematic identification remains an open problem. This points to the need for a tool that automatically detects the database smells. Developers may then, considering the context of the smell, decide whether the detected smells are indeed quality issues or serving a required purpose.

5 QUANTITATIVE ANALYSIS

In this section, we discuss our method to select and mine repositories as well as the detection strategies that we employed in *DbDeo*.

5.1 Mining Repositories

We used the following protocol to select the subject systems. We also illustrate the mechanism that we employed in extracting SQL statements and detecting smells.

Selecting Industrial Repositories We approached two organizations SIG (Software Improvement Group) and SILO (Singular Logic) and sought access to their (or their clients') projects to analyze them. We analyzed a total of 840 projects that belong to various domains including banking, CRM, and telecom.

Selecting Open-source Repositories We employed RepoReaper [22] to select subject systems for the study. RepoReaper provides assessment about GitHub open-source repositories on ten dimensions (architecture quality, community, continuous integration, documentation, history, license, management, state (active or dormant), unit tests, and number of stars). We selected all the 16,057 repositories that score greater than zero or *true* for nine or ten dimensions. We downloaded these repositories one by one, looked for SQL statements in each repository, and discarded the repositories that didn't have any SQL statements.

Extracting SQL Statements We used regular expressions to extract SQL statements from the acquired repositories in *DbDeo*. We implement a two-step process to extract SQL statements. In the first step, we used relaxed regular expressions optimized for speed and in the second step we used stringent regular expression optimized for correctness.

Analyzing and Detecting Smells We found 357 industrial projects and 2568 open-source projects that contained SQL statements. Then, we computed metrics such as the number of SELECT, CREATE TABLE, and INSERT statements as well as the number of files

[1] https://github.com/tushartushar/dbSmellsData

belonging to each programming language and corresponding total lines of code. Finally, we analyzed all the SQL statements from all the repositories using our tool *DbDeo* to detect database schema smells. The raw data generated by the tool can be accessed online.[2]

Table 1 shows some characteristics of the analyzed repositories. On average, industrial projects are 3.87 times bigger than open-source projects by LOC (average LOC for industrial and open-source projects are 617, 617 and 159, 328 respectively) and 5.05 times bigger by number of SQL statements (average number of SQL statements for industrial and open-source projects are 455 and 90 respectively). Although, CREATE TABLE statements are the major source of information to detect schema smells, many times other SQL statements are required to detect these smells. For example, we require CREATE TABLE, CREATE INDEX, and SELECT statements in a repository to detect *index abuse* smell. Therefore, we extracted SELECT, INSERT, UPDATE, and CREATE INDEX statements also in addition to CREATE TABLE statements. We analyzed 393,989 SQL statements from 2925 repositories (on average ≈ 135 SQL statements per repository).

Table 1: Characteristics of the analyzed industrial (I) as well as open-source (OSS) repositories

Attributes	I	OSS
Initial set of repositories	840	16,057
Repositories with SQL statements	357	2,568
Files	2,559,984	3,297,932
Lines of code (source code only)	220,489,273	409,155,497
SELECT statements	51,652	74,096
CREATE TABLE statements	18,907	50,682
INSERT statements	74,416	66,830
UPDATE statements	10,454	29,002
CREATE INDEX statements	7,152	10,798

5.2 DbDeo and Detection Strategies for Database Smells

We developed *DbDeo* — an open-source database smell detection tool[3]. The tool has a meta-model generator component that uses the third-party library SQLParse[4] to parse SQL statements and prepare a meta-model. The meta-model component defines abstractions such as *CreateTableStmt* and *TableColumn* and organizes them in a hierarchical structure. For instance, a *CreateTableStmt* object contains a list of *TableColumn* objects. These abstractions contain information about the parsed SQL statements. For example, one of the attributes belonging to *CreateTableStmt* is *totalColumnsInTable.* The smell detection module in turn uses the meta-model to detect database schema smells.

In the rest of section, we discuss detection strategies employed by *DbDeo* to detect database smells.

Compound attribute: We look for pattern-matching expressions in an SQL query. In a SELECT statement, we check the presence of REGEX in a WHERE clause. We inquire whether a comma is used to separate values that are inserted against an attribute using an INSERT statement. For UPDATE statements, we check the use of a comma in SET clause.

Adjacency list: We look for a foreign key constraint referring to an attribute in the same table.

Metadata as data: We look for a schema definition containing only three attributes. We detect the smell if we find two of the attributes, among three, of type VARCHAR.

Multicolumn attribute: We check the schema for a pattern '<attribute>'N where N is a number. We detect this smell in the table, if the schema has more than one attribute that matches with the above pattern.

Clone tables: We check all the schema definitions within a database for a pattern '<Table name>'N where N is a number. We conclude that a database has this smell when the database has two or more tables matching with the above pattern.

Values in attribute definition: We detect the smell by checking the schema for "enum" or "check" where the construct imposes a restriction on the possible values that can be entered for an attribute.

Index abuse: *Missing indexes* — We identify this variant of the smell when there exists at least one table and the number of indexes in the database are zero.
Insufficient indexes — commonly available database vendors support creating indexes for primary keys implicitly. We look for missing indexes for foreign keys to detect this smell variant.
Unused indexes — We identify this variant when the indexed attributes don't appear in any query.

God table: We count the total number of attributes defined in a schema definition. The table suffers from this smell if the number of attributes defined in the table crosses a threshold (currently we use 10 attributes as a threshold).

Overloaded attribute names: We scan all the attributes and their properties in schema definitions. If we find two or more attributes that have an identical name but defined as different data types, we report this smell.

We also considered detecting the remaining four smells automatically. However, we found it technically challenging to detect them automatically with high accuracy. For instance, *superfluous key* can be detected automatically if we have both the database schema and the data. However, devising heuristics without looking into data is prone to high false-positives.

6 RESULTS

In this section, we discuss the results observed from the analysis on the gathered data with respect to each research question posed.

RQ1. *What are the occurrence patterns of database smells?*

Approach: We use *DbDeo* to detect 9 types of database schema smells in the 357 industrial and 2568 open-source repositories. We collate all the detected instances of smells by their type and we compute average smell density for each type of smell.

Results: Table 2 summarizes the detected instances of database schema smells and corresponding average occurrences per repository in all the analyzed repositories.

We make the following observations from the collected data in the context of this question.

[2] https://github.com/tushartushar/dbSmellsData
[3] https://github.com/tushartushar/DbDeo
[4] https://github.com/andialbrecht/sqlparse

Table 2: Occurrence pattern of database schema smells for industry (I) as well as open-source (OSS) repositories

Smells	Occurrences		Avg. smell density	
	I	OSS	I	OSS
CA	5,517	7,966	0.04	0.04
AL	733	297	0.15	0.02
GT	4,428	5,507	0.44	0.24
VA	85	326	0.00	0.02
MD	944	1,003	0.16	0.09
MA	1,624	3,137	0.10	0.07
CT	101	3,704	0.00	0.05
OA	1814	7,300	0.20	0.21
IA	12,643	9,475	1.25	1.76

We find that *index abuse* is the most frequently occurring smell in both industrial as well as open-source projects. However, it is interesting to note that although the number of instances of *index abuse* smell are higher in industrial projects, they occur relatively less frequently than open-source projects considering their density. On the other hand, *values in attribute definition* in industrial projects and *adjacency list* in open-source projects are the least frequently occurring smells.

In industrial projects, some smells show significantly higher proneness to occur compared to open-source projects. For instance, smell density of *adjacency list* smell is approximately seven times more in industrial projects than the open-source projects. A potential reason of the observation is the higher size and complexity of the industrial projects. On the other hand, *clone table* tends to occur in open-source projects considerably more frequently than in industrial projects.

From the developers' survey, we learned that smells CA (*compound attribute*) and IA (*index abuse*) are the least subjective smells (*i.e.,* context matters the least for such smells) whereas smells AL (*adjacency list*) and VA (*values in attribute definition*) are most subjective in nature. This observation implies that a developer might be hesitant to introduce CA or IA and more open to adopt a solution that involve smells such as AL or VA. Interestingly, the occurrence patterns show exactly the opposite trend with respect to these smells; *i.e.,* smells CA and IA occur the most and smells AL and VA occur the least frequently in both industrial and open-source systems.

RQ2. *Does the size of the project or the database play a role in smell density?*

Approach: We computed smell density for all the detected database smells. In this paper, we define smell density as the number of database smells detected per 10 SQL statements. We then compute the Spearman's correlation coefficient between total LOC (Lines Of Code) and smell density of the repository. We also compute the Spearman's coefficient between size of the database (*i.e.,* number of CREATE TABLE statements) and smell density of the repository.

Results: The Spearman's correlation coefficient (ρ) for the dataset is 0.2420 (p-value $= 3.724 \times 10^{-06}$) for industrial projects and 0.0006 (p-value $=$ 0.9731) for open-source projects. This indicates that density of database smells has low correlation for the industrial projects and no correlation for the open-source projects with the total lines of code in the repository.

We also explore the relationship between smell density and size of the database where size of a database is measured by the number of CREATE TABLE statements in the repository. The Spearman's correlation analysis provides us $\rho = 0.7338$ (p-value $< 2.2 \times 10^{-16}$) for industrial projects and $\rho = 0.6174$ (p-value $< 2.2 \times 10^{-16}$) for open-source projects. The values of the correlation coefficient show that smell density and size of the database share a fairly strong correlation *i.e.,* as the size of database increases, density of database smells tends to increase.

RQ3. *Does the nature of code (type of the application, or usage of* ORM *frameworks) affect the smell density?*

Approach: We extract information concerning nature of subject systems; specifically, we infer the type of application and used ORM (Object-Relational Mapping) framework in each repository.

We infer the type of application among the following set — *Desktop*, *Mobile* (either IOS or Android), or *Web*. We use the following heuristics to classify a repository to one of the application types.

- We figure out the programming language used primarily in a repository. To know the programming language used primarily in a repository, we scanned all the files in the repository, detect the files containing source-code using their file extensions, and count the number of files for each programming language that we detect. We looked for the following programming languages: ASP, C, C#, C++, HTML, Java, JavaScript, Objective C, PHP, Perl, Python, Ruby, SQL, VB, and XML.
- If the prime programming language is Java and there exists a manifest file with name 'AndroidManifest.xml', we conclude that the application is of type *Mobile(Android)*.
- If the prime programming language is Objective C, we tag the application as a *Mobile(*IOS*)* application.
- If the repository contains one of the folders 'Static', 'css', or 'public_html' and primarily used programming language is one of the PHP, ASP, XML, or Python, then we classify the application type as *Web*.
- If the prime language is HTML, then also we interpret the application type as *Web*.
- If none of the above conditions meet for a repository, we classify it as a *Desktop* application.

Once we identify the type of all the repositories, we measure the average smell density for each application type. We select a list of 19 well-known ORM frameworks targeting different programming languages — C++ (LiteSQL, ODB, QxOrm), Java (ActiveJDBC, Apache Cayenne, Eclipse Link, Enterprise JavaBeans, Hibernate, Mybatis), Objective C (Core Data), C# (Dapper, Entity Framework, LINQ to SQL, NHibernate), PHP (Doctrine, Propel), and Python (SQLAlchemy, Django, SqlObject). We scan the dependencies of a repository specified in *import* (or similar) statements to detect whether the repository uses an ORM framework. For instance, we look for import statements in Java applications for the presence of *import org.apache.Cayenne* to know that the application is using Apache Cayenne framework. We measure and compare the average smell density for both ORM-based and non-ORM-based repositories.

Results: Figure 3 (left) shows average smell density for different types of applications. The figure shows that 1998 open-source and 346 industrial repositories are classified as *Desktop*, 40 open-source and 2 industrial repositories as *Mobile*, and 530 open-source and 9 industrial repositories as *Web* applications. For open-source repositories, all three application types exhibit similar database schema smell density. It indicates that application type is not a significant factor that affect database smell density for open-source repositories. On the other hand, industrial *Web* applications show significantly lower smell density than the industrial *Desktop* applications although the sample for mobile and web applications in industrial projects is not significant from a statistical perspective.

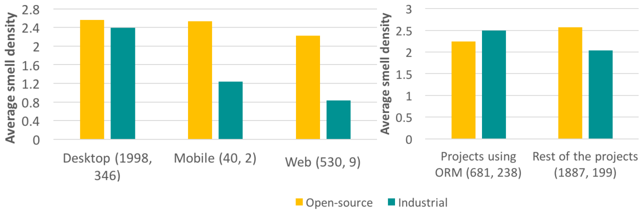

Figure 3: Average smell density of different types of applications (left) and projects using ORM frameworks and rest of the projects (right)

Right side of figure 3 shows average smell density for repositories separated based on whether they use an ORM framework or not. We observed that 681 open-source and 238 industrial projects use ORM frameworks among the analyzed projects. For industrial projects, non-ORM-based projects show lower average smell density than the projects based on ORM frameworks whereas we observe an opposite trend for open-source projects. However, Mann-Whitney U test shows that the difference in the average smell density is not statistically significant (p-value = 0.0252 for industrial and p-value = 0.1612 for open-source projects). Thus, ORM frameworks do not bring immunity from database schema smells.

RQ4. *What is the degree of co-occurrence among database smells?*

Approach: For each detected smell, we count occurrences of rest of the smells in the repository to investigate the degree of co-occurrence among database smells. We compute average co-occurrence for each smell across all the repositories. We take the average of the co-occurrences taking into consideration only those values where the smell has occurred at least once. Further, we normalize the average co-occurrence values with number of detected smells. This exercise reveals the normalized co-occurrence patterns among database smells.

Results: Figure 4 shows average co-occurrence among database smells. The figure reveals that *clone table* for industrial projects and *values in attribute definition* for open-source projects show highest co-occurrence with other smells. *Index abuse* smell exhibits lowest co-occurrence with other smells for both the categories of projects. It implies that whenever a *clone table* in an industrial project or *values in attribute definition* smell in an open-source project gets spotted, it is very likely to find other database smells in the project. On the other hand, *index abuse* smell occurs more independently.

Another interesting observation from figure 4 is that smells shows considerably higher correlations in industrial projects. A potential reason of the fact could be the larger size of industrial projects than the open-source projects (industrial projects are five times larger on average compared to open-source projects).

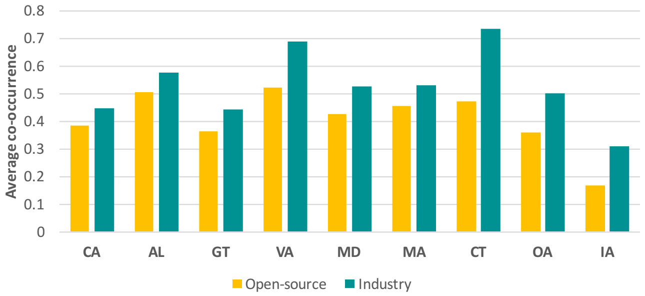

Figure 4: Average co-occurrence among database smells

7 DISCUSSION

In this section, we first discuss our observations about accuracy of the developed tool *DbDeo*. We also present our qualitative analysis of the results presented in Section 6.

7.1 Accuracy of the Developed Tool

We selected ten repositories randomly, performed each step listed in Section 5.1 (*i.e.,* extract SQL code, compute basic metrics, and detect smells) on these repositories, and analyzed the output of each step.

7.1.1 Accuracy of the SQL *Statements Extraction.* An SQL statement may appear in host source code either independently (in separate files) or embedded in the host source code. Majority of times, an embedded SQL statement receives some or all arguments dynamically by the host code. This property, along with diverse vendor-specific syntax of SQL statements, makes it difficult to cover all forms of SQL statements and extract them accurately using regular expressions. Brink *et al.* [2] also reveals challenges in separating embedded SQL statements from host source code considering possible variations in host programming language and vendor specific SQL syntaxes. Given the importance of the extracted SQL statements' quality and associated challenges, we first assess the quality of the extracted SQL statements.

As mentioned earlier, *DbDeo* extracts SQL statements in two steps. In the first step, it extracts the SQL statements embedded in the source code using generic regular expressions. The tool employs a few heuristics and stringent regular expressions in the second step. The second step is rigorous and relatively more time consuming. Extracting potential SQL statements in the first step and then cleanse them gives us performance without compromising on the quality of the extracted statements.

We manually analyzed all the statements in the selected ten repositories and classified them either as an SQL statement, or as an incomplete SQL statement, an extraneous SQL statement, or a non-SQL statement. An extraneous SQL statement has valid SQL statement followed by extraneous text or code that is not part of the SQL statement but was matched by the used regular expression.

Table 3 shows the performance of the SQL statement extraction process. We found two incomplete and two non-SQL statements in the extracted statements. One of the incomplete SQL statements is *“CREATE TABLE xxx.yyy (...)”*. Similarly, one of the non-SQL statements is *“select range from Archive..”*. The statement is written as a comment but fulfils SQL grammar and thus gets extracted.

Table 3: Performance of the SQL extraction process

Total SQL statements	818
Incomplete SQL statements	2
Extraneous SQL statements	0
Non-SQL statements	2

7.1.2 *Accuracy of Smell Detection.*

We detect database smells in all the ten repositories using *DbDeo*. We then verify each detected smell manually to measure the accuracy of the tool. Table 4 shows the total number of detected instances for each smell as well as the identified false-positive instances.

Table 4: Detected smells and identified false-positives

Smells	#Instances	Smells	#Instances
CA	4 (0)	AL	0 (0)
GT	26 (0)	VA	0 (0)
MD	3 (0)	MA	15 (0)
CT	23 (0)	OA	26 (2)
IA	30 (0)		

As the table shows, we identified two false-positive instances in detected smells. The first false-positive instance of *overloaded attribute names* smell is found in the following CREATE TABLE statement (shown partially).

CREATE TABLE *‘sqrl_nonce‘* (* *‘id‘* INT UNSIGNED AUTO_INCREMENT NO NULL PRIMARY KEY, * *‘nonce‘* CHAR(64) NOT NULL, ...

The tool detects the smell because the employed parser interprets ‘*’ as the name of an attribute and tool found another such attribute defined as different type in a different table. However, a manual inspection reveals that this SQL statement exists in a repository written mainly in C. The above SQL statement appears in a comment and the parser used in the tool doesn’t differentiate comments from the rest of the code. Similarly, the source of another false-positive is also a misinterpretation by the parser. Apart from these instances, we find other detected instances as genuine cases of schema smells.

7.2 Qualitative Analysis of the Results

In this section, we discuss the results obtained from our quantitative analysis presented in Section 6 from a qualitative perspective.

Our analysis found a considerable number of *overloaded attribute names* smells. Interestingly, many times developers declare attributes, even the primary keys, with identical names but with different types in a repository. We found that ID is the most popularly used name for a primary key. More than 40% of the analyzed tables belonging to open-source projects use ID as a primary key. For industrial projects, it is considerably lower (11%). An interesting observation is that their type differs significantly. We found 13 and 12 different types being used for the attribute ID across all the analyzed open-source and industrial repositories respectively.

During manual exploration, we also observed one of the reasons for smells *clone table* and *overloaded attribute names* to occur. We observed that these smells occur often in test or example code. This observation highlights the quality deficit introduced in test or example code and possibly reveals the casual mindset of developers while writing test or example code.

Parameterized queries (where values or even sometimes attribute names are supplied dynamically) are very common for embedded SQL statements in source code. We observed CREATE TABLE statements are majorly defined statically; however, understandably, majority of SELECT statements are defined as parameterized queries. This observation has an impact on *index abuse* smell. Our analysis reveals that more than 77% detected instances of *index abuse* smell belong to the third variant of the smell (*i.e.,* unused indexes). When parameterized queries expect attribute names dynamically, our tool cannot identify the used attribute names and produce false-positive instances of *index abuse* smell.

7.3 Opportunities

In the context of this study, we outline possible ways to improve the state of scientific and industrial practice.

Tool support IDEs can provide support, native or extended (via plug-ins), for SQL statements. This may allow developers to spot common problems, such as *index abuse* and *multicolumn attribute*, early on and rectify them. Along the same lines, ORM frameworks may raise an alarm, for instance in the form of warnings, to attract developers’ attention towards potential flaws in the database design. Sophisticated external tools may extend their support to detect database smells and improve the quality of database schemas. Further, language extensions may support the native treatment to embedded SQL statements. The native treatment allows a developer to employ existing tools (the ones used for the host programming language) for embedded SQL code.

Training and awareness The role of focused training sessions to increase awareness of database quality among developers cannot be denied. Such sessions would enable them to learn from existing peer knowledge and keep themselves updated with the changing technology.

Database standards: Standards are a collection of common practices followed globally or within an organization to ensure the consistency and effectiveness of the database environment. A database element naming convention is an example of such a standard. Organizations may adopt stringent standards for designing database schema to ensure the quality of the database system. Across the industry, a move toward stricter and comprehensive standards would prohibit some of the smells we identified.

Database APIs Database APIs can also be improved to support high quality schema design. Apart from deprecating obsolete features and issuing a warning for common mistakes, APIs may offer a new mechanism to verify the schema design. For example, a new CHECK statement (or an optional clause) may allow interested developers to check their schema design upfront and refactor the detected smells before they make their way to the production code.

8 RELATED WORK

The presented work is related to studies of code quality practices in traditional software engineering and software applications backed by relational database systems.

8.1 Traditional Code Quality Practices

Kent Beck [10] introduced the term "code smell" and defined it as "certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring". Code smells are poor design and implementation choices that impact the quality of a software system. Girish *et al.* [30] provides a comprehensive catalog of structural design smells classified based on the principle that they violate. Similarly, Garcia *et al.* [11] present a catalog of architecture smells.

Smells make a software system to decay. They cause technical debt [16] and impair maintainability [32]. Many attempts have been made to detect smells using static code analysis. Metrics-based methods [20, 27] compare metrics computed over code with specified thresholds in order to identify code smells. Decor [21] formulates rules provided through a domain specific language for detecting smells such as *blob* and *swiss army knife*. Machine learning-based approaches, such as Bayesian Belief Networks [15] and Support Vector Machines [18], have been used to detect smells.

Apart from traditional source code, smells have been detected in other related domains. For instance, Hermans *et al.* [12] and Cheung *et al.* [6] detect smells in spreadsheets using metric-based and machine learning-based methods respectively. Similarly, Puppeteer [26] relies on static code analysis for detecting code smells in software configuration code. Our tool *DbDeo* also employs static code analysis to extract and cleanse SQL statements from the host source code and to identify database schema smells.

8.2 Code Quality Practices in Database Applications

There is scant research that explores the quality characteristics of database code. Karwin [14] presents a comprehensive catalog of database antipatterns drawn from industry experience. He organizes antipatterns in four categories: logical database design, physical database design, query, and application development antipatterns. We build on the antipatterns illustrated in this book, especially the first category of database smells *i.e.,* logical database design, and complement it with smells (or best practices) gathered from other resources.

Authors have attempted studies to explore the quality aspect of database code. Brink *et al.* [2] discusses the challenges in extracting SQL statements from the host source code and presents a method to extract and distil SQL statements. The study provides a set of basic metrics concerning database such as number of tables and nested queries. Chen [4] proposes strategies for reducing the impedance mismatch between the relational and object-oriented model in order to improve database performance and integrity.

The knowledge and experience accumulated in popular question and answer sites can be leveraged to help developers avoid smells in SQL queries. Nagy *et al.* [23] mine Stack Overflow questions that are relevant to SQL queries. The study extracts SQL error patterns as a first step towards a recommendation system that aids developers to construct correct queries. Eessaar [8] also discusses a few heuristics that can be employed to detect some of the database smells outlined by Karwin [14]. Many authors have explored object-relational mapping in the context of their implications on application design [31] and performance [5].

Our work differs from the ones described above in that it presents a large scale empirical analysis that studies quality characteristics (database schema smells and their relationship with application characteristics) of database code.

9 THREATS TO VALIDITY

Construct validity concerns the appropriateness of observations made on the basis of measurements taken during the study. Static code analysis is always prone to false-positives and false-negatives. We employed a comprehensive set of tests for the tool to rule out obvious deficiencies. Additionally, we measured accuracy of the developed tool manually; we found the results of the accuracy analysis very satisfactory.

One may adopt one of the numerous techniques to parse and collect relevant source code information. These techniques include AST parsing, string matching, and reflection [29]. Due to the lack of an available tool to extract cleansed SQL statements from a host source code, we implemented the extraction functionality in our tool using regular expressions. Although, the regular expression-based solution cannot be as efficient as AST parsing (for example, separating SQL statements that are appearing in comments is inherently difficult with regular expressions). We employed two-step extraction process to overcome the deficiency. Additionally, we checked the results using both automated and manual tests.

The extraction of the full schema of a database is not guaranteed using the employed method. The implication of such a limitation is that our smell detection method will not report smells that may exist in the uncovered SQL statements. The presented smell detection mechanism uses a threshold (for *god table* smell). Choosing an appropriate threshold is challenging given its subjective nature. We chose the threshold carefully based on personal experience and in such a way that it is neither too lenient nor very stringent.

External validity concerns generalizability and repeatability of the produced results. We cover syntaxes used for major database providers and new syntaxes can be adopted by modifying the currently used regular expressions. Also, the experiment is repeatable; we have made the tool open-source under a liberal license. Further, the raw data generated by the presented analysis has been made available online.

10 CONCLUSIONS

The paper presents a comparative study of relational database schema smells and its relationship with application and database characteristics. We present a catalog of 13 database schema smells based on commonly known best practices to design databases. We carried out a survey to understand developers perspective on database schema smells. We downloaded 16,052 open-source and acquired 840 industrial repositories, selected total 2925 repositories containing SQL statements, analyzed more than 629 million lines of code, extracted more than 393 thousand SQL statements, and detected more than 66 thousand instances of database schema smells. We investigated four research questions and provided empirical observations based on the data obtained.

We observed that 1) the smell *index abuse* occurs most frequently in database code, 2) in industrial projects, some smells such as *adjacency list* show significantly higher proneness to occur compared to open-source projects, 3) the size of the host application has no impact on the density of database smells; however, smell density shows positive correlation with the size of the database, 4) application type (*Desktop*, *Mobile*, or *Web*) has no significant impact on database smell density, 5) use of an ORM framework doesn't avoid database schema smells, and 6) the smell *clone table* in industrial projects and smell *values in attribute definition* in open-source projects exhibit the highest co-occurrence with other database smells.

We also outline a few opportunities to improve the state of the scientific and industrial practice. Specifically, tools to analyze embedded SQL statements and identify potential quality issues could significantly improve the schema quality. Organizations may also contribute to this pursuit by defining appropriate internal standards and training programs. Finally, innovations to database APIs may improve the quality schema design.

We envision the following potential directions for the future. 1) In this paper, we restricted the scope of the study to database schema smells. In the future, we would like to perform a study with expanded scope including query and data smells as well. Additionally, it will be interesting to observe inter-category relationships among database smells. 2) We would like to quantify the impact of the smells on key quality attributes such as performance and maintainability. 3) Finally, we would like to catalog and identify database smells that impair portability.

ACKNOWLEDGMENT

We would like to thank Prof. Damianos Chatziantoniou from the Athens University of Economics and Business for reviewing an earlier draft of the paper and providing improvement suggestions. We would like to thank Fragkiska Goulandri for helping us with related work. We also would like to convey our sincere thanks to all the participants of our online survey for their insightful perspective on database smells. This work is partially funded by the SENECA project, which is part of the Marie Skłodowska-Curie Innovative Training Networks (ITN-EID). Grant agreement number 642954.

REFERENCES

[1] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *IEEE International Conference on Software Maintenance, ICSM*. Universita di Salerno, Salerno, Italy, IEEE, 56–65.

[2] Huib Van Den Brink, Rob Van Der Leek, and Joost Visser. 2007. Quality Assessment for Embedded SQL. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '07)*. IEEE Computer Society, 163–170. https://doi.org/10.1109/SCAM.2007.18

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns* (1 ed.). Wiley.

[4] T. Chen. 2015. Improving the Quality of Large-Scale Database-Centric Software Systems by Analyzing Database Access Code. *2015 31st IEEE International Conference on Data Engineering Workshops (ICDEW)* 00 (2015).

[5] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 1001–1012.

[6] Shing-Chi Cheung, Wanjun Chen, Yepang Liu, and Chang Xu. 2016. CUSTODES: Automatic spreadsheet cell clustering and smell detection using strong and weak features. In *Proceedings - International Conference on Software Engineering*. ACM Press, 464–475.

[7] Database smells on StackOverflow 2010. What are the most common SQL anti-patterns? http://stackoverflow.com/questions/346659/what-are-the-most-common-sql-anti-patterns. (2010). [Online; accessed 12-Oct-2017].

[8] Erki Eessaar and Janina Voronova. 2015. *Using SQL Queries to Evaluate the Design of SQL Databases*. Springer International Publishing, 179–186.

[9] Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bartosz Walter, and Pawel 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 (ICSM)*. 260–269.

[10] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Programs* (1 ed.). Addison-Wesley Professional.

[11] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Toward a Catalogue of Architectural Bad Smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems (QoSA '09)*. 146–162.

[12] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2012. Detecting and visualizing inter-worksheet smells in spreadsheets. In *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*. Delft University of Technology, 441–451.

[13] F. Hermans, M. Pinzger, and A. van Deursen. 2012. Detecting code smells in spreadsheet formulas. In *28th IEEE International Conference on Software Maintenance (ICSM)*. 409–418.

[14] Bill Karwin. 2010. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming* (1st ed.). Pragmatic Bookshelf.

[15] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2009. A Bayesian Approach for the Detection of Code and Design Smells. In *QSIC '09: Proceedings of the 2009 Ninth International Conference on Quality Software*. IEEE Computer Society, 305–314.

[16] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software* 29, 6 (2012), 18–21.

[17] Mario Linares-Vásquez, Sam Klock, Collin McMillan, Aminata Sabané, Denys Poshyvanyk, and Yann-Gaël Guéhéneuc. 2014. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps. In *ICPC 2014: Proceedings of the 22nd International Conference on Program Comprehension*. 232–243.

[18] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. 2012. Support vector machines for anti-pattern detection. In *ASE 2012: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 278–281.

[19] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. 2003. A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*.

[20] Radu Marinescu. 2005. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 701–704.

[21] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Software Eng.* 36, 1 (2010), 20–36.

[22] N Munaiah, S Kroh, C Cabrey, and M Nagappan. [n. d.]. Curating GitHub for engineered software projects. ([n. d.]). Preprint available at PeerJ https://doi.org/10.7287/peerj.preprints.2617v1.

[23] Csaba Nagy and Anthony Cleve. 2015. Mining Stack Overflow for discovering error patterns in SQL queries. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* 00 (2015), 516–520.

[24] Redgate. 2017. 119 SQL Code Smells. http://assets.red-gate.com/community/books/sql-code-smells.pdf. (2017). [Online; accessed 12-Oct-2017].

[25] Tushar Sharma. 2016. Presentation smells: How not to prepare your conference presentation. http://xrds.acm.org/blog/2016/06/presentation-smells-to-avoid-in-conference-presentation/. (2016). [Online; accessed 12-Oct-2017].

[26] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does Your Configuration Code Smell?. In *Proceedings of the 13th International Workshop on Mining Software Repositories (MSR'16)*. 189–200.

[27] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. Designite — A Software Design Quality Assessment Tool. In *Proceedings of the First International Workshop on Bringing Architecture Design Thinking into Developers' Daily Activities*.

[28] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158 – 173. https://doi.org/10.1016/j.jss.2017.12.034

[29] Diomidis Spinellis. 2015. Tools and Techniques for Analyzing Product and Process Data. In *The Art and Science of Analyzing Software Data*, Tim Menzies, Christian Bird, and Thomas Zimmermann (Eds.). Morgan-Kaufmann, 161–212.

[30] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for Software Design Smells: Managing Technical Debt* (1 ed.). Morgan Kaufmann.

[31] Alexandre Torres, Renata Galante, Marcelo Soares Pimenta, and Alexandre Jonatan B. Martins. 2017. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information & Software Technology* 82 (2017), 1–18.

[32] Aiko Yamashita. 2014. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. *Empirical Software Engineering* 19, 4 (2014), 1111–1143.