Studying the characteristics of SQL-related development tasks: An empirical study

Daniel Alencar da Costa[1] · Natalie Grattan[1] · Nigel Stanger[1] · Sherlock A. Licorish[1]

Abstract

A key function of a software system is its ability to facilitate the manipulation of data, which is often implemented using a flavour of the Structured Query Language (SQL). To develop the data operations of software (i.e, creating, retrieving, updating, and deleting data), developers are required to excel in writing and combining both SQL and application code. The problem is that writing SQL code in itself is already challenging (e.g., SQL anti-patterns are commonplace) and combining SQL with application code (i.e., for SQL development tasks) is even more demanding. Meanwhile, we have little empirical understanding regarding the characteristics of SQL development tasks. Do SQL development tasks typically need more code changes? Do they typically have a longer time-to-completion? Answers to such questions would prepare the community for the potential challenges associated with such tasks. Our results obtained from 20 Apache projects reveal that SQL development tasks have a significantly longer time-to-completion than SQL-unrelated tasks and require significantly more code changes. Through our qualitative analyses, we observe that SQL development tasks require more spread out changes, effort in reviews and documentation. Our results also corroborate previous research highlighting the prevalence of SQL anti-patterns. The software engineering community should make provision for the peculiarities of SQL coding, in the delivery of safe and secure interactive software.

Keywords SQL anti-patterns · SQL code · SQL development task · Software maintenance

✉ Daniel Alencar da Costa
danielcalencar@otago.ac.nz

Natalie Grattan
natalie.grattan@postgrad.otago.ac.nz

Nigel Stanger
nigel.stanger@otago.ac.nz

Sherlock A. Licorish
sherlock.licorish@otago.ac.nz

[1] Department of Information Science, University of Otago, Dunedin, New Zealand

1 Introduction

Database storage and operations are key components of any software project requiring data persistence, ranging from small to large scale applications. Relational database management systems (RDBMSs) normally use the *Structured Query Language* (SQL) to access and manipulate data (Melton and Simon 1992). SQL has reached such a degree of importance that despite developments in NoSQL DBMSs, SQL still remains a popular data manipulation mechanism (Gaspar and Coric 2017). Evidence of the importance of SQL is that some NoSQL DBMSs have adapted to support SQL or SQL-like languages (Gaspar and Coric 2017). Other researchers have explored ways of bridging NoSQL to SQL through middleware (Rith et al. 2014; Mason and Lawrence 2005). Altogether, SQL code consistently remains a vital part of persistent data storage and is used for a variety of tasks. Thus, poorly written SQL can be the root cause of performance issues in applications (Faroult and Robson 2006), especially in resource-constrained environments, such as in mobile apps (Lyu et al. 2019).

Recognizing the importance of SQL code, researchers have studied SQL in general (Edmundson et al. 2013; Muse et al. 2020; Karwin 2010; de Almeida Filho et al. 2019). A main line of research is the study of *SQL anti-patterns*, which are recurrent mistakes when writing SQL (Nagy and Cleve 2015). The term SQL anti-pattern can also be interchangeably used with *SQL bad smell* or *SQL code smell* (Muse et al. 2020).

The prevalence of SQL anti-patterns (Muse et al. 2020; de Almeida Filho et al. 2019) hints that developing SQL is a complex task within the already cognitive-heavy task of software development (Walenstein 2002; Mens 2012). Further evidence that writing SQL code is not trivial is that there are roles dedicated to the development of SQL code (e.g., database administrators) (Yilmaz et al. 2015; Miller and Voas 2008). On the other hand, a full-stack developer, who writes and mixes both SQL and application code, faces the unique challenge of excelling in both worlds when performing *SQL development tasks* (i.e., development tasks that involve both application and SQL code). However, we have little empirical knowledge regarding the characteristics of SQL development tasks. For example, do SQL development tasks require more code changes than non-SQL development tasks? Do SQL development tasks take longer to be completed than non-SQL development tasks? Do SQL development tasks require different dimensions of development effort? Knowing these characteristics is important to inform software development tools and practices. For instance, if SQL development tasks typically require more code changes, *quality assurance* (QA) teams may use this information to adjust their code-reviewing priorities. As observed by Kononenko et al. (2018), code size has a statistically significant association with code review time. Another area that could potentially benefit from our study is the area of software estimation (Usman et al. 2014), as understanding the characteristics of SQL development tasks could benefit the reasoning behind estimating story points for user stories that involve SQL. Finally, empirical research like ours is important to better understand special characteristics of atypical software development tasks. In this regard, our study strives to better understand software development tasks, but focuses on SQL development tasks given their importance.

To investigate the characteristics of SQL development tasks, we performed an exploratory *Mining Software Repository* (MSR) study informed by qualitative document analyses (Bowen 2009; O'Leary 2017). Through a study of 20 carefully selected Apache projects, we performed comparisons between SQL and non-SQL development tasks. In the quantitative part of the study, we investigated two characteristics of SQL development tasks:

time-of-completion and *size of changes* (Mende and Koschke 2010; Kamei et al. 2010; Weiss et al. 2007; Giger et al. 2010). In the qualitative part of our study, we analyzed a stratified sample of 687 issue reports to better understand the nature of efforts invested in both SQL and non-SQL development tasks.

–   **RQ1: How many changes and how much time do issues require to be addressed when SQL code is involved?** Issue reports can describe bugs, enhancements, or new features to be addressed (da Costa et al. 2014). Among our main observations are the fact that SQL-related issues may take slightly longer to be addressed when compared to SQL-unrelated issues. Regarding the size of changes, we observe a clear trend that SQL-related issues require significantly more changes than SQL-unrelated issues.
–   **RQ2: How many changes and how much time do pull requests require to be merged when SQL code is involved?** Another core activity in software development, especially in open source projects, is the review of pull requests (also known as merge requests). Our results do not reveal a clear trend whether SQL-related pull requests have a longer time-of-completion when compared to SQL-unrelated pull requests. Although most projects reveal that SQL-related pull requests take longer to be completed, two projects show the opposite trend. Regarding the change size of SQL-related and SQL-unrelated pull requests, we did not observe significant differences in our data.
–   **RQ3: Are the dimensions of effort invested in SQL development tasks different from those of non-SQL development tasks?** We qualitatively investigate a representative sample of 687 issue reports to better understand the dimensions of effort invested in SQL-related and SQL-unrelated tasks. Our main observation is that SQL-related task are more likely to have a larger scope when compared to SQL-unrelated tasks.

Our paper is organized as follows. In Section 2, we survey the research related to our work. In Section 3, we describe the methodology of our empirical study. We present our obtained results in Section 4 and discuss them in Section 5. In Section 6, we reflect on the threats to validity of our study and we conclude our paper in Section 7.

2 Related Work

In this section, we explore two research themes that are closely related to our work: SQL anti-patterns detecting tools or anti-pattern automators, and empirical studies analysing SQL code and anti-patterns.

2.1 Tools or Anti-Pattern Automators

Identifying anti-patterns in source code is a time consuming task. Therefore, much research have invested in the creation of new tools to help automate the process of identifying SQL or *Object-Relational Mapping* (ORM) anti-patterns.

Chen et al. (2014) proposed a framework for detecting anti-patterns in ORM usage. They also identified the most common anti-patterns that have a performance impact. These anti-patterns are "one-by-one processing" and "excessive data" (Chen et al. 2014). In a follow-up study, Chen et al. (2016) investigated the impact of redundant data on performance in Java applications, further emphasising how inefficiencies in data access, retrieval, and storage can impact performance in a system. Cheung et al. (2013) aimed to optimize how SQL is used in software projects. They proposed a tool that parses source code and locates areas where code could be moved into the database layer rather than being carried out in the

application code, since it is a costly operation to communicate from the application to the database.

Nagy et al. (2015) designed a tool for locating the parts of source code where SQL queries are sent to the database. The tool can be a potential solution to the problem of not being aware of the query that is executed by the database, which can mask the complexity of the query. Lyu et al. (2018) investigated the *Repetitive Autocommit Transaction* (RAT) anti-pattern, which involves repeated database transactions rather than batch transactions in mobile applications. This anti-pattern can have a significant impact on performance, so they proposed a tool which identifies this anti-pattern and refactors instances that can be changed without causing deadlocks.

Much research in this area involve SQL static code analyzers, which automate the extraction of SQL code or anti-patterns from a source code repository. The tool Alvor is an Eclipse IDE plug-in that checks whether SQL statements embedded into code are syntactically correct, through a static analysis of the code to find SQL statements (Annamaa et al. 2010). Wassermann et al. (2007) presented a technique to evaluate dynamic SQL queries. SQLInspect, designed by Nagy and Cleve (2018), is another Eclipse plug-in that extracts SQL queries and finds anti-patterns (based on Karwin (2010) anti-patterns). The authors used benchmark projects and compared the performance against Alvor and other similar tools. SQLCheck, developed by Dintyala et al. (2020), is a similar tool that extracts SQL queries and detects anti-patterns. However, SQLCheck adds further functionality by ranking the anti-patterns based on their impact in the application, and fixing the anti-patterns by suggesting alternative approaches. The authors used SQL queries from real-world contexts to evaluate SQLCheck.

In contrast to previous research, we are not interested in detecting anti-patterns in SQL code, but in understanding whether development tasks that involve both SQL and application code (i.e., SQL development tasks) have special characteristics (e.g., a longer time-to-completion or different dimensions of effort). These investigations can help us further improve software development processes and practices.

2.2 Empirical Studies on SQL Anti-Patterns

Recent studies have conducted empirical analyses of SQL anti-patterns. Lyu et al. (2019) examined the impact of SQL anti-patterns on code performance in mobile applications. To compile a list of anti-patterns, the authors conducted an extensive literature review. They then used a benchmark design with values for performance-affecting factors and anti-pattern instances from a sample of Google Play applications. They tested whether performance improved by measuring changes in resource consumption. They found the anti-patterns "unbatched-writes" and "loop-to-join" (Lyu et al. 2019) to be prominent in terms of performance impact.

The anti-patterns outlined by Karwin (2010) provided a turning point in SQL anti-pattern research. Eessaar (2015) investigated 12 of Karwin's SQL anti-patterns, both logical and physical. To investigate the anti-patterns, they ran SQL queries based on a test database with design flaws and investigated the *Information Schema* tables to analyze the design of the database. The authors were able to identify the majority of the anti-patterns. The purpose of the study was not to test the performance impact of these anti-patterns but rather find methods for detecting them.

Nagy and Cleve (2015) parsed a dataset of STACK OVERFLOW posts related to MySQL. They extracted code snippets and filtered for SQL keywords. Next, they used a pattern detector, which involved breaking the elements into a tree structure and sorting nodes into

buckets based on their similarity. The authors did not find particular patterns that were likely to lead to errors.

Another source of data for empirical investigations is OSS projects hosted on GITHUB.[1] Studies conducted by de Almeida Filho et al. (2019) and Muse et al. (2020) used GITHUB to collect their data. de Almeida Filho et al. (2019) investigated SQL anti-patterns in PL/SQL projects. They studied 20 popular GITHUB projects and found that certain anti-patterns tended to be highly correlated with one another and that they tended to fall into clusters around syntax bad smells and code structure bad smells. Muse et al. (2020) used `SQLInspect` (Nagy and Cleve 2018) to extract SQL anti-patterns. They also extracted traditional code smells to measure correlations between SQL anti-patterns and traditional code smells. They found that "implicit columns" was the most common SQL anti-pattern. They also found that SQL anti-patterns tended to persist throughout iterations of the project. Given that Muse et al. (2020) used an automated approach to identify SQL anti-patterns, their investigations were limited to only four anti-patterns.

The existing empirical research on SQL anti-patterns has shown that SQL anti-patterns are indeed prevalent. However, what is missing is a closer analysis regarding the development process when it comes to SQL development tasks. For instance, are developers exerting as much effort when it comes to performing SQL development tasks? Our study, for the first time, provides an in-depth analysis regarding the characteristics of SQL development tasks. Our qualitative analysis helps us better understand the effort invested in development activities involving SQL code.

3 Methodology

In this section, we review our subject projects and explain how the data used in this study were collected. We subsequently outline our research questions and associated measures.

3.1 Subject Projects

To perform our empirical investigations, we studied Apache projects (ASF 2019). We selected Java Apache projects because Java has been the main language used by Apache. Apache projects have been frequently used in prior software engineering research (Rigby and Hassan 2017; Roberts et al. 2006; Neto et al. 2018; Vandehei et al. 2021). We chose Apache projects for our study as a strategy to avoid the inclusion of toy, personal, or non-engineered projects that can be found among GITHUB projects (Kalliamvakou et al. 2014; Pickerill et al. 2020).

Another goal related to studying Apache projects is to control for quality, since the Apache Foundation has a thorough process to select its contributors (i.e., committers).[2] For example, Apache works as "*a meritocracy. Once someone has shown sufficient sustained commitment to a project by helping out and contributing work to it (and the ASF), the project may vote to invite that person to become a committer.*"[3] Moreover, Apache has led
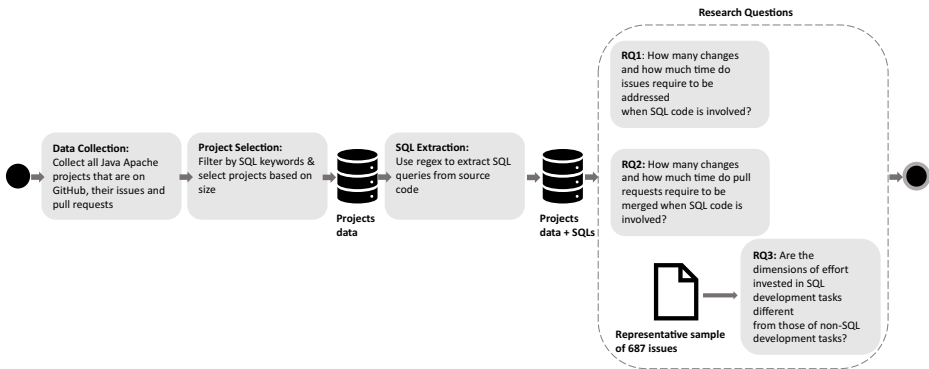

Fig. 1 High-level overview of steps involved in our study

the development of open source software for 22 years, including the implementation of *Java Specification Requests* (JSRs), such as the Java Persistence API, or simply JPA.[4,5]

Provided that we can ensure a certain level of professionalism in software development by selecting Apache projects, the second aspect of our project selection is to study projects with a substantial number of SQL queries. In this regard, we aim to carefully select projects with a substantial number of meaningful SQL statements. We explain the steps to select such projects in the next subsection.

3.2 Procedure and Data Collection

The first step to answer our research questions was to collect the required data. Each of the following steps refers to the corresponding heading in Fig. 1.

Step 1: Data Collection. We used the GHTORRENT database (Gousios 2013) to obtain the names of all the Apache projects that have repositories available on GITHUB. From here, we cloned each repository to a local computer and scanned each repository for SQL query keywords including SELECT, DELETE, UPDATE and INSERT. This initial keyword scan of projects provided a baseline filter that we built upon later with additional regular expression filters. This formed a subset of Java Apache projects that are likely to contain SQL. The commands to perform this initial scan can be found in our replication package.[6]

Step 2: Project Selection. Similar to de Almeida Filho et al. (2019), we selected 20 Apache projects to perform our empirical study. We believe 20 is a reasonable number because our goal was to include all projects in our analyses, including the qualitative analyses of SQL development tasks in RQ3.

Our protocol for sampling our 20 Apache projects is as follows. First, we found 2,100 projects related to Apache on GITHUB. We excluded 1,100 projects as they were not mainly written in Java. We also excluded forks and kept only original repositories. Next, we scanned projects based on SQL keywords. Projects for which SQL keywords could not be found were excluded (778 projects were excluded). To select our final set of projects we counted the number of Java files in each of the remaining 208 projects. We observed that some

---

[4]https://www.jcp.org/en/jsr/detail?id=317

[5]http://openjpa.apache.org/

[6]https://zenodo.org/record/7343828#.Y55ttXZBxPZ

projects contained an exorbitant number of files (e.g., Netbeans with about 37,500 files). As in this first research we do not intend to base our empirical observations on extreme cases, we removed outlier projects based on the upper limit of the inter quantile range of file counts, i.e., the same process used by boxplots to infer outliers (4 projects were removed). Finally, we selected the top 10 projects (in terms of file counts) and the 10 projects right above the median file count. This last step was intended to enrich our dataset with projects of different sizes and applications.

Step 3: SQL Extraction. We developed a Python script to obtain all the string literals in the Java files of the selected projects. Our intention was to find queries written in native SQL. The script created a dataset of string literals, alongside details of what file and line number the string literal was from, and which project. The intention was to ensure that we could later revisit the context in which the SQL query was found, which is important for our analyses in answering RQ3.

In order to extract the SQL from the string literals, we crafted regular expressions. A different regular expression was made for each of the following cases: a `SELECT` statement, an `UPDATE` statement, an `INSERT` statement, and a `DELETE` statement. These were executed using the SQL regex tool to match the patterns. The regular expressions were validated by the first and second authors using the OODT project, which is close to the median with 143 extracted SQL queries. A manual analysis was performed in this project by analyzing all the Java files and counting the number of SQL queries. Next, we compared the set of SQL queries from the manual analyses with those found by the regular expressions. The regular expressions performed well with a precision of 96.5% and recall of 97.9%. While we validated our regular expressions on only one Apache project, due to the high accuracy obtained, we are fairly confident that the majority of SQL queries can be extracted. Moreover, we also performed another manual validation of the SQL queries found by our regular expressions at a later stage in answering RQ3.

Table 1 shows the statistics related to our selected projects, i.e., the number of SQL statements, number of SQL-related issues and pull requests (PRs), and number of SQL-unrelated issues and pull requests. Overall, our selected projects contain a mean of 722 SQL queries, with a maximum of 6,010 queries (Hive) and a minimum of 5 queries (Tuscany). These numbers suggest that our studied projects are data-intensive systems (Muse et al. 2020).

Step 4: Data Mining. After obtaining our list of projects, we collected additional data related to pull requests (from GITHUB) and issue reports (from JIRA). We collected pull requests marked as *closed* or *closed and merged* and considered only pull requests that have been merged to the main branch for the quantitative analyses as we are interested in computing time-of-completion and change size metrics only for those pull requests that eventually made it to the end users. Additionally, even if any of those pull requests involved other branches, because the work has eventually been merged to the main branch, we did not lose information when computing the *time-to-completion* for such pull requests. Regarding issue reports, we collected reports marked as closed and fixed. This is because our goal is to investigate the time taken for pull requests and issue reports to be marked as closed. These collection tasks were facilitated using Python libraries such as `PyGithub` and `jira`.

3.3 Research Questions

We reiterate below our three research questions along with their motivations, approaches and measures.

Table 1 General characteristics of our selected Apache projects

Project	SQL-Statements	SQL-rel. issues	SQL-unrel. issues	SQL-rel. PRs	SQL-unrel. PRs
CLOUDSTACK	1,049	93	2,329	353	7,599
ASTERIXDB	18	4	187	2	16
GEODE	4,356	172	3,404	16	6
HADOOP MAP-REDUCE	37	6	621	–	–
GROOVY	58	19	3,812	12	3,260
HARMONY	406	–	110	–	–
HIVE	6,010	697	6,803	556	1,840
KARAF	67	3	436	27	3,551
TOMEE	230	5	263	57	1,955
FLINK	354	198	5,254	269	5,111
KNOX	11	5	1,124	16	1,216
LUCENE-SOLR	325	116	9,301	49	3655
MANIFOLDCF	108	–	4	–	23
MARMOTTA	34	2	253	2	6
NIFI	933	–	31	193	3,726
OODT	143	2	24	5	121
SYNAPSE	24	–	42	1	32
OOZIE	262	97	1,364	12	28
OPENWEBBEANS	6	2	724	–	5
TUSCANY	5	4	835	–	–

Note that some projects, such as Harmony and Tuscany, did not have any pull requests in their GITHUB repositories at the time our dataset was collected

RQ1: How many changes and how much time do issues require to be addressed when SQL code is involved? Issue reports can describe bugs, enhancements, or new features to be addressed (da Costa et al. 2014). Therefore, addressing issue reports sits at the core of the software development endeavour. We study the time-of-completion and size of issue reports to better understand whether there are significant differences when issue reports involve SQL code. This is our preliminary investigation to understand whether SQL development tasks are addressed in a different manner when compared to non-SQL development tasks.

RQ2: How many changes and how much time do pull requests require to be merged when SQL code is involved? Another core activity in software development, especially in open source projects, is the review of pull requests (also known as merge requests). Pull requests are normally reviewed by other (typically more experienced) members of the project development team (Rahman and Roy 2014). In RQ2, we study time-of-completion and size of pull requests to better understand whether there are significant differences when pull requests involve SQL code. As in RQ1, RQ2 help us better understand whether SQL development tasks are addressed differently when compared to non-SQL development tasks.

RQ3: Are the dimensions of effort invested in SQL development tasks different from those of non-SQL development tasks? We further investigated our data to better understand why the *time-of-completion* and *change size* of SQL development tasks significantly differ from those of non-SQL development tasks. This investigation is important not only to triangulate the results obtained in RQ1 and RQ2 but also to provide deeper insights

for researchers and practitioners. For example, although the *time-to-completion* of SQL development tasks is typically longer when compared to non-SQL development tasks, the difference could arise either because (i) SQL development tasks are less important, taking longer to receive attention from developers; or because (ii) SQL development tasks require more effort from developers, thus taking longer to complete. The goal of RQ3 is to shed more light on discussions of a similar nature as (i) and (ii).

4 Results

RQ1: How many changes and how much time do issues require to be addressed when SQL code is involved?

Approach. To find which issues were related to SQL, we linked GITHUB commits to issue reports from JIRA. We used an approach repeatedly used in previous research (Posnett et al. 2011; da Costa et al. 2016; Neto et al. 2018; Yatish et al. 2019) to link JIRA issues with commits. For example, for the project OODT, we used Python scripts to search each commit message for "OODT-" followed by a number, which is the convention used by Apache projects to refer to issue IDs within commit logs.[7] Afterwards, for each commit, we ran our SQL regular expressions on only the patches for Java files. If any SQL queries were found within a patch, the issue was marked as containing SQL. To measure the *time-to-completion* of issue reports, we measured the difference between the date that an issue report was opened and the date that the issue report was fixed. To compute the *size* of an issue report, we summed all the added and removed lines in the commits linked to that issue report.

We used beanplots to visualize the distributions of *time-to-completion* and *size* (Kampstra 2008). To check whether distributions were statistically different (i.e., between SQL issues vs. non-SQL issues), we used the Mann Whitney Wilcoxon (MWW) test (Wilks 2011) and the Cliff's delta effect-size measurement (Cliff 1993). Both MWW and Cliff's delta are non-parametric, as Shapiro-Wilks tests indicated the non-normality of our data (Shapiro and Wilk 1965). The MWW test checks whether two distributions come from the same population and the Cliff's delta measures the probability that a randomly selected value of one distribution is higher (or lower) than another distribution (Fig. 2).

Results. *SQL-related issues have a slightly longer time-to-completion when compared to SQL-unrelated issues.* Figure 2 shows the distributions of time-to-completion for SQL-related issues versus SQL-unrelated issues. The left-hand distribution of each beanplot represents issues not involving SQL, whereas the right-hand distribution represents issues that involve SQL. We observe that, in terms of *time-to-completion*, SQL-related issues take slightly longer to be completed. For issues, we obtained $p = 5.33 \times 10^{-54}$ and Cliff's delta $d = 0.241$, indicating a small but significant difference (Table 2).

SQL-related issues involve substantially more code modifications than other issues Fig. 3 shows the distributions of change size (i.e., the sum of code additions and code deletions) for SQL-related issues versus SQL-unrelated issues. We observe that SQL-related issues receive significantly more code modifications than SQL-unrelated issues: $p = 1.45 \times 10^{-230}$ and $d = 0.507$, indicating a *large* significant difference between the distributions.
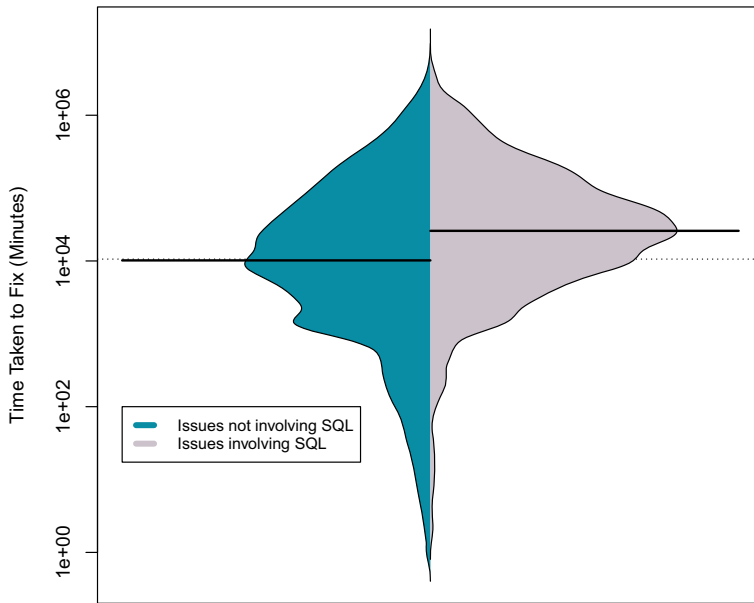
---

[7]https://issues.apache.org/jira/browse/OODT-1031

Fig. 2 Time-to-completion in minutes. SQL-related issues and pull requests both take longer to be fixed in our studied projects

Table 3 shows descriptive statistics for the distributions shown in Fig. 3. In line with the results above, we observe a large difference between the medians when it comes to issue reports: 422 lines modified for SQL-related issues against 68 lines modified for SQL-unrelated issues. These results suggest that SQL-related issues typically involve more code modifications than other issues.

Per project analysis (time-to-completion). Our observations made so far concern the general trend of our data, i.e., we considered all projects in an aggregated manner. Although it is valuable to know the general trend in the data, it is also important (especially in software engineering) to understand the specificities and contextual profiles of each project, since each project uses different processes, tools and are from different domains.

Table 4 shows the median time-to-completion of SQL-related issues versus SQL-unrelated issues in each of our studied projects. Table 4 also shows whether the results are statistically significant (i.e., as indicated by the MWW tests and Cliff's delta values).

Table 2 Descriptive statistics for time-to-completion (in minutes)

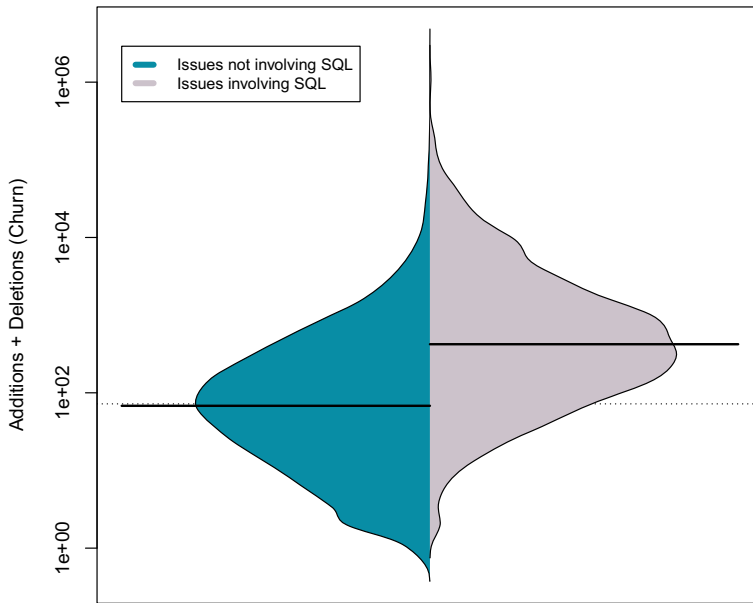|	Time-to-Completion of Issues (All projects)	
	SQL-related	*SQL-unrelated*
Minimum	2	0
Median	25,897 (17.9 days)	10,161 (7 days)
Mean	121,427 (84 days)	93,368 (65 days)
Maximum	3,888,797	6,129,740

Fig. 3 The size of changes (additions plus deletions). SQL-related issue reports receive more code changes

Four projects (NIFI, SYNAPSE, HARMONY, and MANIFOLDCF) were not included in the per-project analysis as we could not find SQL-related issues in these projects.

Out of 16 analysed projects, 5 projects obtained significant p-values (GEODE, HIVE, FLINK, LUCENE-SOLR, and OOZIE). FLINK obtained the smallest effect-size (i.e., considered negligible) whereas LUCENE-SOLR obtained the highest effect-size (i.e., considered medium). Although not all projects obtained a significant p-value, 12 out of 16 projects had a longer time-to-completion when it comes to SQL-related issues, which may explain why the general trend of time-to-completion (i.e., considering all the projects together) indicated a *small* but significant difference in the time-to-completion between SQL-related issues and SQL-unrelated issues. Overall, our per-project analysis suggests that the time-to-completion of SQL-issues may be slightly higher than SQL-unrelated issues in certain projects.

Per project analysis (size of changes). In Fig. 3, we observed a *large* significant difference in the size of changes between SQL-related issues and SQL-unrelated issues. To gain further insights, we study the differences in the size of changes in each of our analysed

Table 3 Descriptive statistics for change size (number of lines)

	Size of Issues (All projects)	
	SQL-related	*SQL-unrelated*
Minimum	2	0
Median	422	68
Mean	4,338	790
Maximum	1,115,413	1,810,320

Table 4 Time-to-completion of issues per project

	Median time-to-completion of issues in days			
	SQL-unrelated	SQL-related	p-value	Cliff's delta
ASTERIXDB	18.4	83.2	0.29	—
CLOUDSTACK	6	4.6	0.92	—
GEODE	9.9	20.7	**$7.82e^{-09}$**	**0.26 (small)**
GROOVY	7.1	19.9	0.07	—
HADOOP-MAPREDUCE	27.9	51.3	0.20	—
HIVE	7.1	17.9	**$9.01e^{-32}$**	**0.27 (small)**
FLINK	8.9	14.6	**0.003**	0.12 (negligible)
KARAF	2.9	0.4	0.86	—
KNOX	0.1	39.5	0.09	—
LUCENE-SOLR	5.8	48.8	**$8.40e^{-13}$**	**0.39 (medium)**
MARMOTTA	7	1.5	0.41	—
OODT	2.2	52	0.89	—
OOZIE	15	26.9	**0.01**	**0.15 (small)**
OPENWEBBEANS	1.9	62.2	0.49	—
TOMEE	0.3	12.5	0.32	—
TUSCANY	4	28.1	0.11	—

projects. Table 5 shows the median change size (for both SQL-related and SQL-unrelated issues) as well as the p-values obtained through our statistical tests.

Our results revealed a clear trend of SQL-related issues requiring significantly more changes than SQL-unrelated issues. It is observed that 75% of our projects ($\frac{12}{16}$) obtained a significant p-value with (at least) *medium* effect size measurements. Indeed, 9 projects obtained a *large* difference according to their effect size measurements. Our results suggest that SQL-related issues indeed require significantly more changes in order to be addressed.

RQ2: How many changes and how much time do pull requests require to be merged when SQL code is involved?

Approach. To identify whether pull requests were related to SQL, we ran our regular expressions on the commits associated with each pull request (again, only on patches for Java files). To compute the *time-to-completion* of each pull request to be merged or closed, we computed the difference between the date that a pull request was submitted and the date it was closed or merged. To compute the *size* of a pull request, we summed all the added and removed lines in the commits associated with that pull request. As in RQ1, we used beanplots to visualize the distributions of *time-to-completion* and *size*. We also use the MWW test and the Cliff's delta measurement to compare our distributions.

Results. *SQL-related pull requests have a slightly longer time-to-completion when compared to SQL-unrelated pull requests.* Figure 4 compares the distribution of time-to-completion between SQL-related pull requests and SQL-unrelated pull requests. We obtained a $p = 6.14 \times 10^{-22}$ and a $d = 0.28$, indicating a *small* but significant difference. The descriptive statistics for our distributions of time-to-completion are shown in Table 6. Our results suggest that SQL-related pull requests may take slightly longer to be completed when compared to their SQL-unrelated counterparts.

Table 5 Size of issues per project

	Median size of addressed issues			
	SQL-unrelated	*SQL-related*	*p-value*	*Cliff's delta*
AsterixDB	208	3017	**0.008**	**0.77 (large)**
CloudStack	18.4	83.2	**3.23e^{-13}**	**0.45 (medium)**
Geode	85	854	**1.72e^{-41}**	**0.61 (large)**
Groovy	50	293	**3.75e^{-07}**	**0.67 (large)**
Hadoop-MapReduce	92	836	0.006	**0.65 (large)**
Hive	77	406	**4.61e^{-84}**	**0.45 (medium)**
Flink	82	403	**1.37e^{-28}**	**0.46 (medium)**
Karaf	27	133	0.08	—
Knox	45	1,117	0.09	—
Lucene-Solr	90	529	**0.001**	**0.80 (large)**
Marmotta	87	968	0.06	—
OODT	24	19,239	**0.02**	**0.90 (large)**
Oozie	41	384	**1.36e^{-24}**	**0.62 (large)**
OpenWebbeans	91	16,241	**0.01**	**0.97 (large)**
Tomee	92	661	**0.02**	**0.59 (large)**
Tuscany	77	356	0.11	—
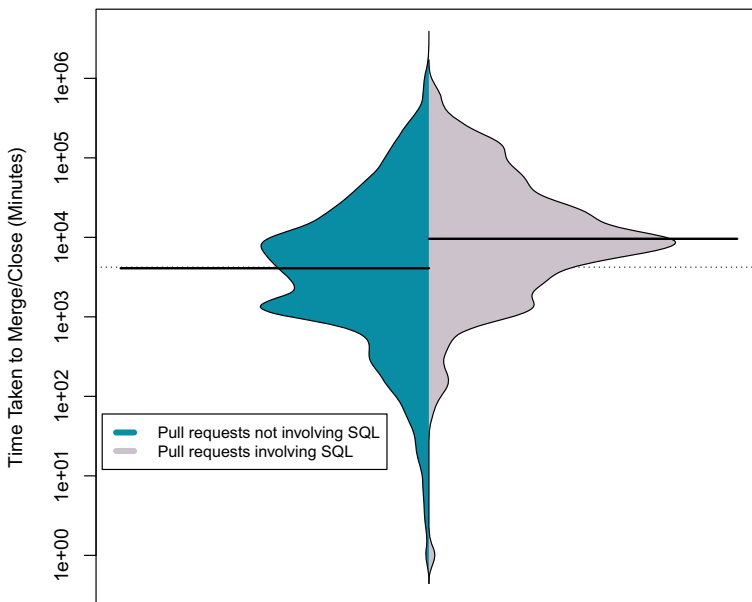
Fig. 4 Pull requests. Time-to-completion in minutes. SQL-related issues and pull requests both take longer to be fixed in our studied projects

Table 6 Descriptive statistics for time-to-completion (in minutes)

	Time-to-Completion of Pull requests (All projects)	
	SQL-related	*SQL-unrelated*
Minimum	0	0
Median	9,588 (7 days)	4,086 (3 days)
Mean	36,940 (26 days)	25,265 (17 days)
Maximum	757,374	1,731,191

SQL-related pull requests have a negligible difference in terms of change size when compared to SQL-unrelated pull requests. Figure 5 shows the comparison of change size between SQL-related pull requests and SQL-unrelated pull requests (i.e., added lines plus removed lines within patches). We observe that SQL-unrelated pull requests received a slightly higher amount of code modifications compared to SQL-related pull requests. However, while the p-value for the comparison is significant ($p = 6.42 \times 10^{-5}$), our Cliff's delta measurement indicates a *negligible* difference ($d = 0.12$), meaning that the observed difference is likely inconclusive. Table 7 shows the descriptive statistics of the size of changes for our pull requests.

Per project analysis (time-to-completion). To better understand the general distribution of time-to-completion shown in Fig. 4 we compare the distributions of time-to-completion in each studied project. In this per-project analysis, we did not include the TUSCANY, HADOOP-MAPREDUCE, HARMONY, MANIFOLDCF, and OPENWEBBEANS projects. In
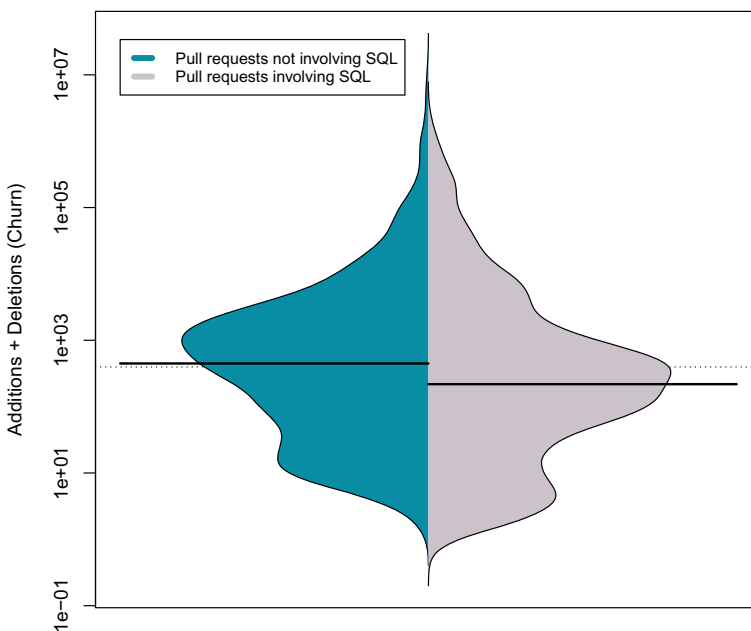

Fig. 5 Pull requests. The size of changes (additions plus deletions). SQL-related issue reports receive more code changes to be addressed

Table 7 Descriptive statistics for change size (number of lines)

	Change size of Pull requests (All projects)	
	SQL-related	*SQL-unrelated*
Minimum	0	1
Median	218	446
Mean	18,124	29,300
Maximum	1,563,226	8,434,616

the cases of TUSCANY, MANIFOLDCF, and OPENWEBBEANS, we could not find SQL-related pull requests, whereas there were no closed pull requests in HADOOP-MAPREDUCE and HARMONY. Table 8 shows our obtained results for the per-project analysis (i.e., median time-to-completion and p-values).

We observed that 60% of the analysed projects ($\frac{9}{15}$) obtained statistically significant difference. While the majority of these projects tend to have a longer time-to-completion for SQL-related pull requests, two projects (GEODE and SYNAPSE) show the opposite trend (i.e., SQL-unrelated pull requests have a longer time-to-completion than SQL-related pull requests).

Overall, our results suggest that more projects need to be investigated to draw stronger conclusions related to the time-to-completion of pull requests. For example, reflecting on our previous analysis regarding issue reports, although not all projects obtained significant p-values, the results were consistent (i.e., all projects had a longer time-to-completion for SQL-related issue reports). However, when it comes to the per-project analysis of pull requests, GEODE and SYNAPSE revealed an opposite trend from the general trend shown in Fig. 4. Lastly, given that we did not observe a significant difference between SQL-related

Table 8 Time-to-completion of pull requests per project

	Median time-to-completion of issues in hours			
	SQL-unrelated	*SQL-related*	*p-value*	*Cliff's delta*
ASTERIXDB	4.4	within 1 hour	0.16	—
CLOUDSTACK	170	465	**$5.40e^{-8}$**	**0.17 (small)**
GEODE	1,179	within 1 hour	**0.001**	**0.97 (large)**
GROOVY	32	82	0.75	—
HIVE	435	568	0.10	—
FLINK	72	142	**$6.64e^{-11}$**	**0.23 (small)**
KARAF	20	14	0.77	—
KNOX	15	86	**0.001**	**0.45 (medium)**
LUCENE-SOLR	121	2,801	**$1.09e^{-13}$**	**0.37 (medium)**
MARMOTTA	within 1 hour	within 1 hour	1	—
OODT	53	125	0.30	—
OOZIE	294	66	**0.03**	**0.42 (medium)**
SYNAPSE	30	within 1 hour	**$7e^{-4}$**	**0.98 (large)**
NIFI	52	186	**$2.53e^{-16}$**	**0.25 (small)**
TOMEE	48	175	**0.01**	**0.18 (small)**

and SQL-unrelated pull requests when it comes to the size of changes, we do not perform a per-project analysis regarding the size of changes for pull requests.

RQ3: Are the dimensions of effort invested in SQL development tasks different from those of non-SQL development tasks?

Approach. To answer RQ3, we pushed beyond the quantitative realm and used a qualitative approach known as *document analysis* (Bowen 2009; O'Leary 2017). Analyzing documents entails coding their content into themes (similar to how interview transcripts would be analyzed) (Bowen 2009). One of the main advantages of document analysis is that documents are "non-reactive," meaning that documents can be read and revisited multiple times without being changed by the research process ((Bowen 2009), p. 31). As the input for our document analysis, we used *digital issue reports* and their related documents, e.g., pull requests (from all branches), code review boards, or commit logs related to the issue reports. From the total of 38,160 issue reports obtained from our 20 studied projects, we created two representative samples: one containing issue reports involving SQL code, and the other issue reports that do not involve SQL code. Considering a confidence level of 95% and a confidence interval of 5%, we obtained 304 issue reports involving SQL code and 383 issue reports not involving SQL code. Instead of simply building random samples, we used a *Stratified Random Sample* (SRS) strategy because of the variability in the number of issue reports involving SQL per project in our population (Lohr 2009). Therefore, the number of issue reports involving SQL code per project in our samples is representative of the number of issue reports per project in the original population. To calculate the size of each stratum, we used the formula $n_h \approx n W_h$, where n_h is the sample size of stratum h; n is the size of the sample; and $W_h = N_h / N$ where N is the population size and N_h is the size of stratum h in the population (Podgurski et al. 1999).

To investigate whether our representative sample maintained the properties of its population (i.e., the 38,160 issue reports), we show in Figs. 6 and 7 the comparisons regarding time-to-completion and size of changes (which are equivalent to Figs. 2 and 3). Indeed, the statistical properties hold as both comparisons exhibit a similar behaviour, i.e., $p = 2.53x10^{-9}$ with $d = 0.27$ (small) for time-to-completion and $p = 3.43x10^{-24}$ with $d = 0.45$ (medium) for size of changes. Therefore, we proceeded with our document analysis.

Figure 8 shows an overview of our document analysis process. Our document analysis consists of two main steps. In the first step, the goal is to find different dimensions of effort invested to address the issue reports within each sample. This inductive approach generates a set of themes related to the different dimensions of effort invested in addressing issue reports. In this step, the first author ("main coder" in Fig. 8) analyzed all the issue reports from both samples and created *themes* based on the observed dimensions of effort invested to address the issue reports. For instance, if an issue report required intense discussions before a solution was proposed (which can be observed in the comments section of an issue report), the theme *"intense discussions"* was created (more details are provided in the results). Once the entire set of themes was generated and documented through several iterations and reflections (see Appendix A for the complete set of themes), the main coder discussed the themes and their meanings with two other authors ("secondary coders"). The secondary coders then used the existing set of themes to code the issue reports from both samples. At this step, the secondary coders had the opportunity to suggest new themes. Next, all coders collaboratively discussed the generated themes (e.g., merging themes, or accommodating new themes), producing a final set of themes.
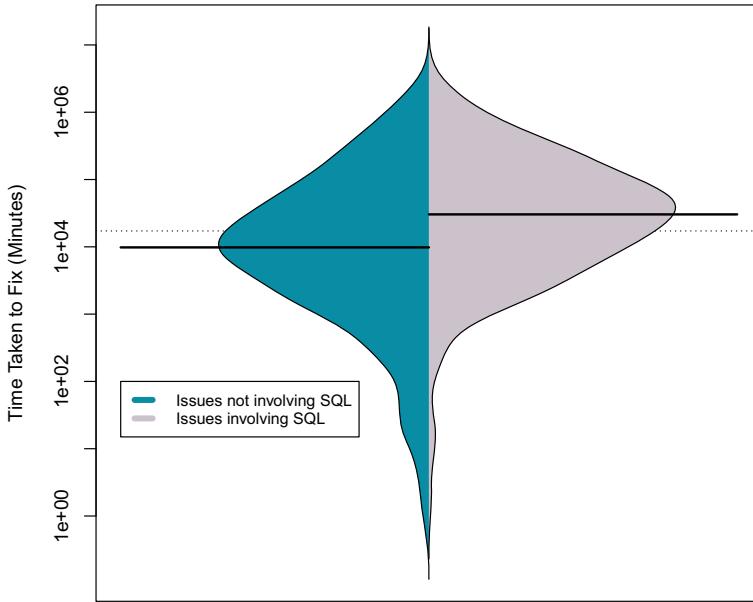
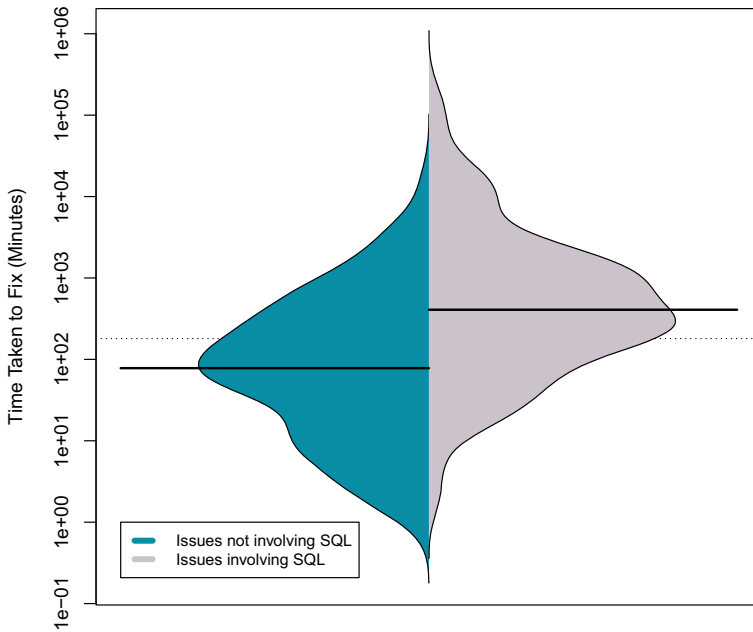**Fig. 6** Time-to-completion of the issues in our representative sample

Fig. 7 The size of changes (additions plus deletions) of our representative sample
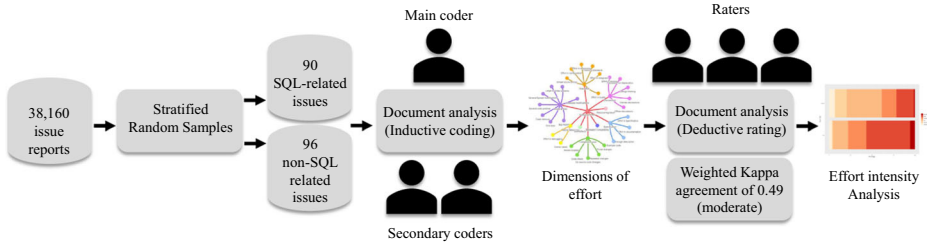
Fig. 8 An overview of the document analysis process

It is important to note that our goal with the inductive analysis is to widen our understanding regarding the effort invested in addressing the investigated issue reports instead of finding an *absolute truth* regarding what would be the most accurate themes produced in our data. For this reason, we position our inductive approach as a *reflexive thematic analysis* (Braun and Clarke 2020; 2019) and, hence, we are not interested in *coding reliability* at this stage (e.g., measuring inter-rater reliability). Instead, we are interested in generating qualitative results that can help us better reflect on the phenomenon of effort invested in the addressed issues.

Once the set of themes is created, the second step is to use the generated themes as a guide for our deductive analysis. The goal of our deductive analysis is to obtain a sense of the intensity of effort invested to address the issue reports. In this deductive analysis, three authors (i.e., three coders) assessed all issue reports separately. Each coder used a five-point Likert scale to indicate their perception of the intensity of effort invested in addressing an issue report; one of "very low", "low", "medium", "high", or "very high." In contrast to the previous step, at this stage, we are interested in measuring the accuracy of our perception of exerted effort within an issue report. For this reason, we used the *weighted kappa* (Cohen 1968) as our inter-rater reliability measure. We chose the weighted kappa because it is sensitive to the distance between disagreements, e.g., a disagreement between "very low" and "low" does not have the same weight as a disagreement between "very low" and "high." Finally, we compared the required effort for issues involving SQL code to that of issues that do not involve SQL code.

Results. Figure 9 visualises the dimensions of effort (i.e., the themes) that emerged from our inductive analysis (see Section 3.3). The central (or root) theme is the *perceived effort*, which is the main object of analysis in this RQ. The second-level (or axial) themes comprise effort related to: *scope & size*, *discussions*, *specification*, *test & debugging*, *complexity level*, *backward compatibility*, *reoccurring issue*, *side-effects*, and *side-efforts*. The third level themes are more specific and are grouped based on their relationship with the second-level themes. For instance, the theme *known location* refers to issues where the developers already knew where to fix the problem from the start. Therefore, the *known location* theme falls within the *complexity level* axial theme, since already knowing where the problem is located from the start indicates that the issue is not complex after all. Another example is the *discussions* axial theme, which groups all themes related to effort invested into discussions (e.g., discussing the design of a solution thoroughly as in the *design thinking* theme). In Appendix A, we provide in-depth details about each theme that emerged from our inductive analysis.

Fig. 9 An overview of the dimensions of effort (themes) that emerged from our inductive analysis

SQL development tasks require more widely spread modifications with a larger scope than non-SQL development tasks. In the next step of our analysis, we compare the themes for issues that involved SQL code versus issues that did not involve SQL code. Figure 10a visualises the key themes for SQL development tasks. The thickness of the edges and size of nodes are based on the number of times a theme occurred over the total number of theme occurrences.[8] For example, the theme *several/spread modifications* has the highest number of occurrences (188 times) within issues involving SQL code. Hence, the node and edge for *several/spread modifications* is the thickest in Fig. 10a. Regarding the high-level themes (i.e., the root theme and the Complexity Level, Discussions, Specification, Scope & Size, Side Effort, and Test & Debug themes), we consider their number of occurrences as 1 (one), since their role is mostly to communicate how the lower level themes are related to each other.

Figure 10b visualises the key themes for non-SQL development tasks. The most notable difference compared to Fig. 10a is in the *scope & size* theme: while SQL development tasks require more *several/spread modifications* (i.e., the changes involved different files, classes, or packages, or several code changes), non-SQL development tasks require more *localized modifications* (i.e., the changes were mostly within a file, or, even if they were
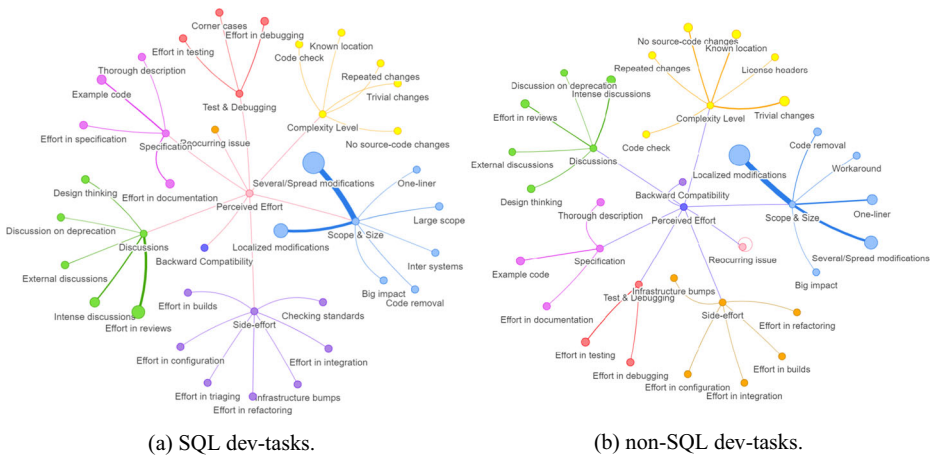
---

[8]https://github.com/neo4j-contrib/neovis.js/

(a) SQL dev-tasks.

(b) non-SQL dev-tasks.

Fig. 10 Dimensions of effort

in different files, they were only a few), simple *workarounds* (i.e., quickly crafted but not ideal solutions) or even *one-liner* modifications (Karampatsis and Sutton 2020) (i.e., modifications that required only one line of code). This last theme (*one-liner*) emerges mostly for non-SQL development tasks. Indeed, because the themes *several/spread modifications* and *localized modifications* are, in essence, mutually exclusive, we perform a χ^2 test of independence (McHugh 2013). Our goal was to check whether there exists a significant difference in occurrences of *several/spread modifications* and *localized modifications* between SQL and non-SQL development tasks. We obtain a $p = 1.161 \times 10^{-18}$ indicating that the observed difference is indeed significant — Table 9 shows our $2x2$ matrix used to compute our χ^2 test. Other interesting differences are related to the *complexity level* and *side-effort* themes. For example, SQL development tasks require different kinds of *side-effort* from non-SQL development tasks, such as *checking SQL standards*, *effort in configuration*, and *infrastructure bumps*. For instance, in issue HIVE-15982,[9] a developer comments *"I tested it on Postgres and it agrees with Oracle. So, its [sic] worth rechecking the standard for this."* They then proceed to check the standard and report back on the results. As for the differences in *complexity level*, we observe that non-SQL development tasks can involve problems for which the *location is known* from the start as well as problems that require *no source-code changes*.

Development tasks that involve SQL require a higher amount of effort compared development tasks that do not involve SQL. Figure 11 shows the result of our deductive analysis through stacked bar charts. We observe that SQL development tasks involve a substantially higher proportion of issues (53%) rated as "high" or "very-high" in terms of perceived effort compared to non-SQL development tasks (14%). In terms of inter-coder agreement, we obtained a weighted Kappa of 0.49, which signifies a good agreement beyond chance (Cohen 1968).

---

[9]https://issues.apache.org/jira/browse/HIVE-15982

Table 9 $2x2$ matrix used to compute our χ^2 test of independence

	Several/Spread modifications	Localized modifications
Non-sql	89	204
Sql	188	90

5 Discussion and Implications

Interpreting time-to-completion and change size. In RQ1, we observed that issue reports involving SQL code have a slightly longer *time-to-completion* and a significantly higher *change size* when compared to issue reports not involving SQL. In RQ2, we did not observe results that are as conclusive as the results of RQ1, since some projects revealed opposite tendencies in terms of time-to-completion of pull requests. Notwithstanding these observations, one has to be careful when interpreting *time-to-completion* and *change size*.
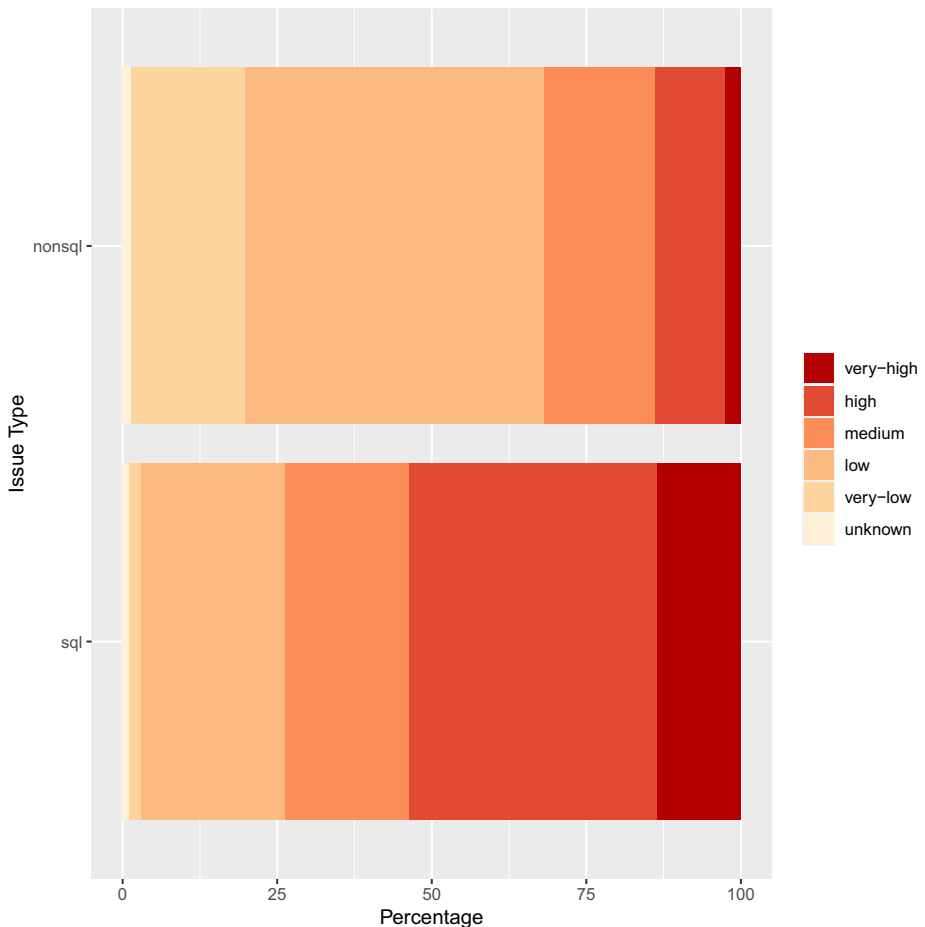
Fig. 11 The perceived effort invested in SQL development tasks vs. non-SQL development tasks.
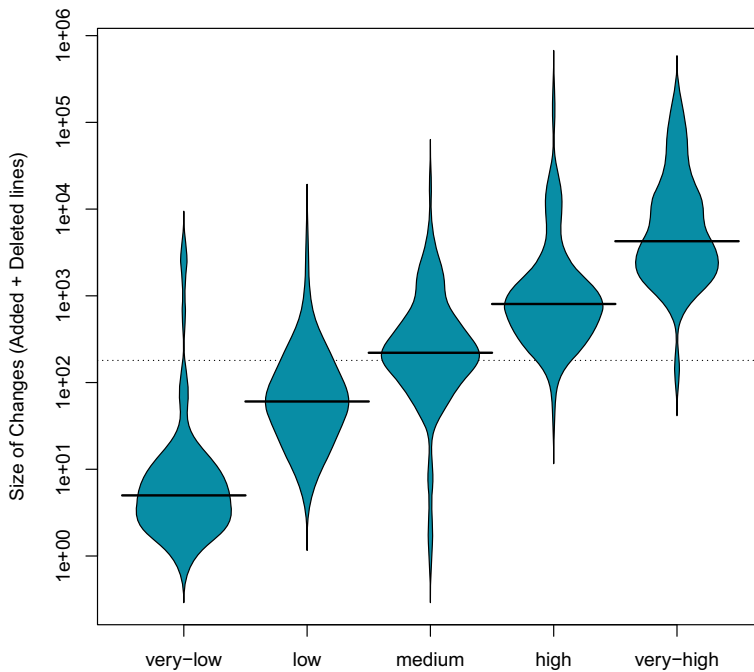
Fig. 12 The relationship between the perceived effort and the size of changes of issues

In terms of a longer *time-to-completion*, the interpretations can take opposite directions. A task may deliberately take longer simply because the task has a lower priority instead of being more complex. As for *change size*, while a higher *change size* may indicate that more effort has been invested, the nature of the changes also plays a role. For example, even if a high number of changes have been made, if they were mostly copies of the same piece of code (i.e., represented by the theme *repeated changes* in RQ3), the higher number of changes is not necessarily indicative of higher effort. We can see a practical example of this issue where the HIVE-1928 issue report received a perceived effort rating of "medium" in our analysis in RQ2 despite having a high number of code changes, as most of the changes were repetitive (e.g., changing the function `priv.getPriv()` to `priv.toString()` in different locations).[10] Otherwise, we probably would have classified HIVE-1928 as "high" perceived effort.

Given the prudence required to interpret *time-to-completion* and *change size*, we further analyse the relationship between our *perceived effort* (from RQ3) and the *time-to-completion* and *change size* of our representative sample. Figure 12 shows the distributions of change size (y-axis) per category of perceived effort (x-axis).

It is interesting to observe that the size of changes required by issues tend to increase as our perceived effort also increases. Indeed, an issue deemed as requiring a *high* effort may need fewer code modifications than a *medium* one (as one can note from the variations of the distributions), which can be explained by those issues that, despite not requiring as many code changes, involved more intense discussions or other type of efforts (e.g.,

---

[10] https://issues.apache.org/jira/browse/HIVE-1928

effort in reviews). We ran a Kruskal Wallis test (Kruskal and Wallis 1952) to check whether the different distributions (i.e., very-low, low, medium, high, and very-high) are significantly different from one another. We obtain a statistically significant outcome $p = 6.127 \times 10^{-94}$, indicating that the distributions are likely different. We then ran several pair-wise Wilcoxon tests — using Bonferroni-Holm corrections to counteract the problem of multiple comparisons) — and observed that all distributions are statistically different from one another.

Figure 13 shows the distributions of time-to-completion (y-axis) per category of perceived effort (x-axis). Indeed, the time-to-completion tends to be longer as the perceived effort increases. Again, given the variations in the distributions, a *very-low* effort issue could take longer than a *high* effort issue, which can be explained by those cases where, although an issue would be easy to fix, the priority of such an issue may not be high. Another interesting observation is that the variation in distributions tends to reduce as the effort increases. For example, it is more likely that a *very-high* effort issue will take a longer time to be addressed, whereas a *very-low* effort issue may be addressed very quickly or may equally take a longer time. Our Kruskal Wallis test hints that the distributions are statistically significantly different from one another ($p = 4.595 \times 10^{-24}$). Next, our pair-wise Wilcoxon tests reveal that, when performing pair-wise comparisons, only the *very-low vs. low* and *high vs. very-high* distributions are not statistically different.

Overall, these result helps us triangulate the results obtained in RQ1. For example, it is less likely that the difference in *change size* or *time-to-completion* in issue reports involving SQL code is solely due to issues being less important or of a lower priority. Overall, we believe that we have sufficient evidence to conclude that SQL development tasks require more effort from developers.
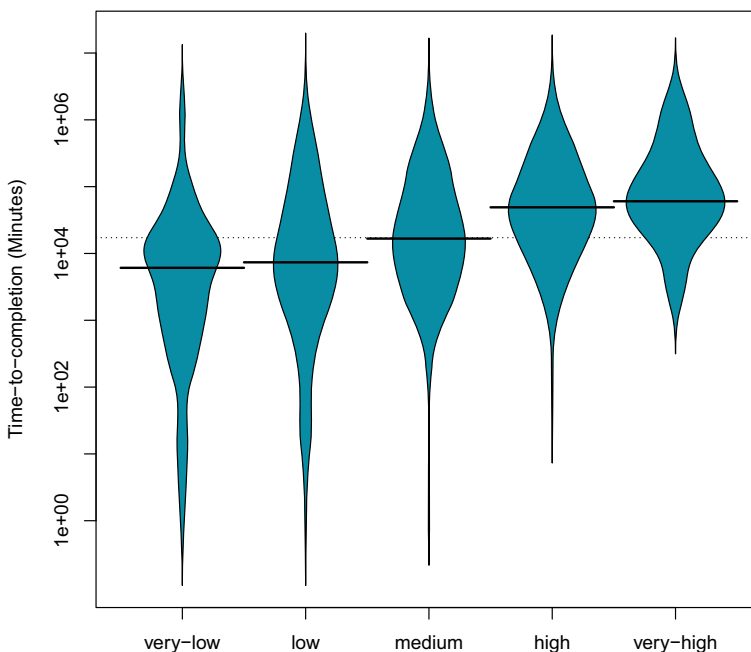

Fig. 13 The relationship between the perceived effort and the time-to-completion of issues

Interpreting the spread-out nature of SQL-related issues. In RQ3, we observe that SQL-related issues require more changes and/or are more spread-out when compared to SQL-unrelated issues. One might argue that SQL-related issues require more changes not necessarily because of the amount of *code* changes involved but because they may co-occur with changes in UI and configuration files (i.e., involving HTML or XML code), which can even involve auto-generated code. Due to this reason, we show in Fig. 14 a comparison of *change sizes* similar to RQ1 (issues) and RQ2 (pull requests), but considering Java code changes only.

In terms of issues, we obtain a significant p-value ($p = 1.071 \times 10^{-222}$) with a *large* effect-size ($d = 0.55$), meaning that the discrepancy in the difference between SQL-related issues and SQL-unrelated issues actually increased when compared to the difference observed in RQ1 (see Fig. 2). Regarding pull requests, we obtain an insignificant p-value ($p = 0.789$), which is a comparable result to the one observed in RQ2.

Implications for practice and research. The general trend of our results has implications for practice. For instance, developers, being aware that changes involving SQL code may take longer (or need more coding), may prioritise these changes accordingly (e.g., when prioritising the tasks to tackle in the next sprint). More developers may be added to these tasks, or they may be tagged for enhanced oversight. Another implication of our results is that, knowing that SQL-related tasks may involve more code changes, developers or project managers can consider this information when *creating and communicating software estimations* (e.g., when playing planning poker, rules may be carefully followed to ensure the most reliable estimates are entered on the system). The software estimation aspect is particularly important because accurate estimations are directly related to the delivery of a successful software project (Whigham et al. 2015; Shepperd 2014). Additionally, if an issue is likely impacting SQL code, developers may consider this information in their decision to prioritise the code review of the changes involved in such an issue.

Regarding implications for research, our work opens interesting avenues for future studies and the development of approaches. For example, research has been invested in predicting which issues and pull requests should be addressed next (i.e., prioritisation) (Zhao et al. 2019; Van Der Veen et al. 2015). It would be interesting to investigate whether the presence of SQL-code in a patch associated with an issue (or in a pull-request) could help such models to improve their predictions. While this has not been previously considered a
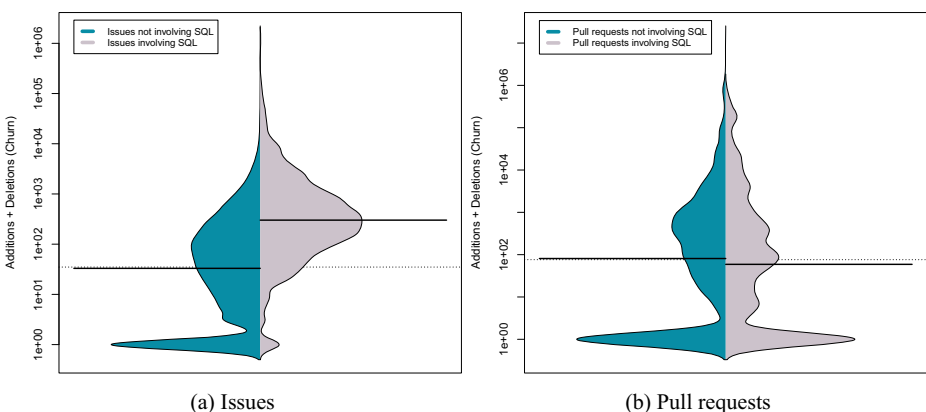

(a) Issues   (b) Pull requests

Fig. 14 Distributions of change size for Java-only changes

pertinent issue, our evidence suggests that it is worth consideration, at least for evaluating relevance.

Lastly, when considering other contexts, such as in open source development, contributors should be patient that SQL-related issues may take a bit longer or may require more changes to be addressed. Evidence for RQ3 somewhat supports the added challenges presented by SQL-code, including at times the need to consult with SQL standards. This has implications for times when such standards are not readily available too, and so, teams should organise to ensure that project resources are visible to everyone to reduce delays.

6 Threats to Validity

In this section, we discuss the threats to validity of our work.

Construct validity concerns how our conclusions are based on our evidence. The main construct threat of our work is related to the information that we *cannot see*. For example, as discussed in Section 5, the results obtained in RQ1 can have several interpretations, one of which is that issue reports may take longer to be addressed because they are less important. However, it is challenging to gauge the importance of an issue report. For instance, developers do not always express their opinion about the importance or urgency of an issue report, which makes it harder for researchers to extract this information from the data that is available. For example, in issue GROOVY-3832, a developer commented *"John, any chance you'll be able to look at this in time for 1.7.2 (approx 1 week away) otherwise I'll attempt to take a look."*[11] Such a statement can be indicative of some urgency, but it is challenging to identify the degree of urgency or whether the absence of such a statement in other issue reports is indicative of less urgency. To mitigate the limitations related to construct threats, we use both quantitative and qualitative analyses, including inductive and deductive approaches as explained in Section 3.3.

An additional construct threat is the regular expressions we developed to capture SQL statements within source code repositories, which support our investigations throughout this work. These regular expressions may not return all the SQL code in every project (false negatives) and they may return some false positives, which are pieces of code identified as SQL without actually having any SQL code. However, based on the high precision (96.5%) and recall (97.9%) of our reliability checks, we expect this risk to be low.

Another threat is related to the relevancy of the SQL code within commits. For example, one may question whether the SQL code included in the commits was relevant to the changes represented by these commits or whether the inclusion of the SQL code in the patch was inconsequential (e.g., the changes in the commit spanned areas with SQL code, but did not actually change the logic of the queries). In this regard, we note that 80% of the SQL-related tasks in our data (i.e., this is the case for both issues and pull requests) contain direct changes to SQL code (i.e., SQL code was directly involved in deletions/additions/modifications). On the other hand, for the cases where the SQL statements were not directly involved in modifications (i.e., they were present in the patch but within context lines), there is no direct way to determine whether the changes were completely inconsequential, unless we were able analyse the patches case by case and had the domain knowledge of the studied projects to do so.

---

[11] https://issues.apache.org/jira/browse/GROOVY-3832

Nevertheless, to account for this threat, we examined 72 of the issues (i.e., a sample with 95% confidence level and 10% confidence interval) where SQL statements were not directly involved but were present in the context lines of the fixing patches. We observed that for 90% ($\frac{65}{72}$) of these issues, developers are required to understand the SQL statement in the vicinity, meaning that these changes fixing these issues can still be regarded as SQL-related changes. For example, in change `a206ef9ea1126cb4ab3239853633546e93b6f3f8`, which fixed issue HIVE-862, a developer introduced the code `d.run(cmd).getResponse()`, where `cmd` is a string holding a `SELECT` query. It is reasonable to conclude that the developer must understand the query as they introduced code that executes such a query. Another interesting observation is that 19% ($\frac{14}{72}$) of these issues have fixing commits that actually directly changed SQL statements but were not identified as such by our regular expressions. For example, in change `b6218275b00b64aed7efaf470784cc0441464f67`, which fixed issue HIVE-4924, a developer modified a multi-line SQL-query. However, because the beginning of the query, (i.e., the part containing the `SELECT * FROM`) was not involved in the change, our regular expression did not identify such a change as directly modifying an SQL statement, but as having an SQL statement in its vicinity instead. As such, we are confident that our results are unlikely to suffer from bias and are properly based on true SQL-related changes.

Lastly, we acknowledge that addressing pull requests and addressing issue reports can be an interconnected process, i.e., before marking an issue report as *addressed* a developer may need to merge a pull request related to that issue report. Nevertheless, for the purpose of our research, we studied issue reports and pull requests separately because our investigated projects do not use pull requests and issue reports in a consistent manner. For example, in the GEODE project, developers tend to link a pull request to an issue report.[12] However, in the CLOUDSTACK project, commits are performed directly to the code-base and are mentioned in issue reports, i.e., not necessarily via pull requests.[13] Thus, although there may be an overlap between the tasks represented by issue reports and pull requests, we analysed issue report and pull requests separately to avoid the intricacies of different projects' processes.

External validity concerns the extent to which we can generalize our results to other environments (e.g., other software projects). A clear limitation of our research is that we only used Apache projects to conduct our study. Because of this, our conclusions are less generalizable to a wider population of projects and communities. For instance, it may be the case that because Apache projects adhere to a certain rigour in their development process, their SQL code is more stable than in projects of other communities. This potential higher stability of SQL code may influence our results, since SQL code would only be touched by tasks of higher impact and scope.

However, we weighed the choice of studying only Apache projects against the issues of selecting GITHUB projects more randomly and the possibility of selecting small or unusual projects. An interesting future work would be to investigate a greater variety of projects or to

---

[12] See the "links to" section on https://issues.apache.org/jira/browse/GEODE-4894

[13] See comments section of https://issues.apache.org/jira/browse/CLOUDSTACK-4146

compare and contrast two different OSS communities. Different open-source communities may have different practices. For example, in the Mozilla open-source community, they specifically mark performance bugs, whereas in the Apache community there is no option for this in JIRA (Jin et al. 2012).

Another limitation is focusing only on OSS communities. It is possible that closed software communities have entirely different practices altogether. For example, closed source communities tend to have a few dedicated testers (Bachmann and Bernstein 2009) rather than the more crowd-sourced approach to testing that is seen in OSS projects. This could mean issues are not fixed as quickly or pull requests are not merged as quickly, thus producing different results to our research.

Lastly, although our Apache projects have a substantial number of SQL queries, they cover very different domains. For example, SQL-related tasks developed for Groovy (a programming language) may be different in nature from SQL-related tasks developed for Hive (a data warehouse project). Nevertheless, in this study, we are interested in studying the general trends in the characteristics of SQL-related tasks. Studying whether the characteristics of SQL-related tasks are different for projects of different domains is an interesting possibility for future work.

Internal validity is generally concerned with the potential confounding factors involved in the relationship between independent and dependent variables (especially in the case of causal relationships).

In this research, we are not necessarily interested in investigating causal relationships between a dependent variable and its explanatory variables. Instead, our study is more exploratory in the sense that our aim is to better understand the characteristics of SQL development tasks (i.e., tasks that involve both SQL code and application code). In RQ1 and RQ2, the dependent variables were *change size* and *time-to-completion* and the independent variables were *SQL development task* or *non-SQL development task*. Nevertheless, we refrain from drawing a causal relationship in RQ1 and RQ2, tapping into more qualitative investigations in RQ3 to help us collect enough evidence to explore the relationship between SQL development tasks and development effort.

Regarding the qualitative analyses, especially those employed in the inductive analysis of RQ3, there is always the potential for bias from the authors' subjective experiences. For example, had the authors had different experiences, the themes that emerged in RQ3 might have been different. Despite this potential threats we employed rigorous qualitative methods to conduct our analyses (e.g., recruiting several coders and computing inter-rater agreement measures). Regarding the inductive analysis in RQ3, we acknowledge that the emergent themes may not be exhaustive and should be taken as a base framework that other researchers can further improve or refine.

Lastly, we acknowledge that, as the authors of this work, we do not have the domain knowledge of the investigated Apache projects in order to fully understand the code-base and the decisions made in the design of these projects. Nevertheless, we are able to search for "clues" that indicate the effort invested in an issue. Such clues are present in our scheme described in Appendix A. Such a scheme for our manual analysis was developed through our document analysis process described in the approach section of RQ3. To conclude, although our approach is not perfect (in the sense that we do not have the same technical or domain knowledge that a developer from our chosen projects would have), we are still able to gauge the relative required effort when comparing issues.

7 Conclusion

The goal of our research is to better understand the characteristics of SQL development tasks (i.e., development tasks involving both SQL code and application code). For this purpose, we set out to conduct an empirical study using 20 Apache projects to investigate whether SQL development tasks have a longer *time-to-completion* or whether they require more code modifications (as indicated by a larger *change size*) when compared to other development tasks. Indeed, we observe that SQL development tasks may take slightly longer to be completed and likely require more code modifications. To dive deeper into these findings, we employ a document analysis strategy with two main steps: an inductive analysis and a deductive analysis. As a result of our inductive analysis, we observe several dimensions of effort related to SQL development tasks and non-SQL development tasks. The major difference in terms of dimensions of effort is that SQL development tasks require more spread out changes and extra effort related to SQL-specific tasks such as checking SQL standards. According to our deductive analysis, SQL development tasks also require more effort from developers in general.

Overall, our empirical research suggests that SQL development tasks not only require extra effort, but also different types of effort. As potential implications of our research, we envision that software development practitioners may take into account the nature of SQL development tasks when performing effort estimations (e.g., when using planning poker in agile) or planning code reviews. Furthermore, those awaiting fixes for SQL-related issues should anticipate the involvement of more rigour from reviewers. As future research, we plan to investigate the interplay between developers' experience and the effort required in SQL development tasks. There is also scope for us to replicate this study using data sets from other communities.

Appendix A: Dimensions of Effort: Themes

Below we explain the meaning of each of the themes found in the inductive analysis of our *document analysis* (as explained in Section 3.3). The themes outlined below along with their descriptions served as a guide to perform our deductive analysis, i.e., gauging the required effort that was necessary to address a development task.

Several/spread modifications. Whether the changes involved different files (e.g., within different classes or packages) or whether there were many code changes.
Several code patches. Sometimes fixes involve several patch iterations. This might indicate a higher effort invested in the fix. However, several patches with trivial changes do not indicate a high effort.
Code check. Whether the fix involved adding only a code check.
Debug effort. Whether there was a considerable effort related to debugging the problem. This criterion can be mostly verified in the comments.
Intense discussions. Sometimes the issue does not require large code changes, but the discussions revolving around the changes were intense (i.e., discussing alternative solutions or ideas). This may indicate that the solution required a high effort (even if not many lines of code had been changed).
Localized modifications. The opposite of *Several/spread modifications*. This criterion indicates that the changes were mostly local (within a file), or, even if they were in different files, they were only a few. This criterion normally denotes "low" effort.

Offline discussions.   Sometimes there is evidence that further discussions were taken offline and not registered in the issue report itself. This criterion may help in judging whether extra effort was required to fix the issue.

Effort in documentation.   Sometimes, addressing certain issues requires not only new source code, but also new documentation. This criterion represents the effort to produce new documentation.

Effort in reviews.   This criterion denotes when higher than usual effort was invested in code review (e.g., they explicitly mention an external system where reviews were held and many comments were placed there).

Backward compatibility.   This criterion denotes when extra effort was needed to ensure backward compatibility.

Specification effort.   This criterion denotes when extra effort was needed to elaborate on the problem at hand. For example, when a developer does not fully understand the problem at hand, they might ask for more clarification. Consider the following quote as a practical example of this criterion: "What's VC, by the way? Venture Capital?"[14] The meaning was actually "virtual columns".

Effort in configuration.   This criterion denotes when there was more than usual effort in creating or changing the configuration of the system.

Effort in integration.   When there was more than usual effort required to integrate things (e.g., the software suddenly broke because of the new patches or conflicting merges).

Big impact.   This criterion emerges whenever there was an explicit indication that the feature or bug fix would have a large impact on the current system. For example, the bug fix or new feature would require a carefully thought out change.

Checking standards.   Whenever there is effort in checking standards in order to address the issue. For example, if the issue requires a new SQL function to be developed, developers may need to check what the SQL standard says prior to deciding on a solution.

Corner cases.   Whenever there was effort in covering corner/specific cases (usually in tests) to increase the quality of the fix or the new feature.

Infrastructure bumps.   Whenever there was a problem in the infrastructure (not necessarily physical infrastructure, but also soft infrastructure, such as server containers, or database servers) because of the issue. This criterion denotes that there was more than usual effort in fixing/updating the infrastructure.

Trivial changes.   This criterion is not much related to the size of the change, but denotes when a change was "easy" to apply. For example, to address an issue, the developers exposed information that was already computed but had not been exposed before. Another example is the addition of getters/setters, i.e., although the code changes can be large, adding getters/setters is trivial and often automated by IDEs. This criterion normally denotes "low" effort.

Effort in refactoring.   Whenever there was extra effort in performing refactoring operations while fixing/addressing the issue.

Large scope.   Whenever an issue required changes to several modules of the system. This criterion is an exacerbated version of *several/spread modifications*.

Design thinking.   Whenever there are clear discussions related to the design of a fix or new feature. For example, one developer might argue that an inner class should be sufficient to fix something, but another developer argues that actually creating a new package would be best.

---

[14]https://issues.apache.org/jira/browse/HIVE-8186

Unknown.   Whenever there is not enough information on the issue report (normally this criterion is only used when there are really no comments within an issue report).

Inter systems.   Whenever the issue require developers to think of the communication/interaction/integration of diverse systems in order to propose a solution.

Thorough description.   This criterion denotes when there was more than usual effort to describe the issue at hand. For instance, a long description was provided with code examples, etc.

Discussion on deprecation.   Whenever there is additional effort in discussing whether certain functionalities should be deprecated or not.

Effort in builds.   Whenever there is additional (and more than usual) effort in creating/modifying the build system (or running the builds) of the software project.

One-liner.   This criterion denotes when the issue was addressed with only one line of code. This is an exacerbated version of *localized changes*. Other variations of this criterion are *two-liners* or *three-liners*. This criterion normally denotes "very-low" effort.

Side effects.   This criterion denotes when there was an additional problem/bug because of the solution that was being developed for the issue at hand. Consider the following excerpt as a concrete example: "I found a hidden bug in FieldSortedHitQueue that materialized when writing a TestCase for a SortField.BYTE sorting with custom parser (it took me a long time to find out whats [*sic*] happening)."[15]

Example code.   This criterion applies whenever there was extra effort in providing codes examples in a discussion of an issue report.

Repeated changes.   Although code changes may be numerous, they may only be repetitions of the same code, which does not require as much effort. This criterion normally represents lower effort.

Code removal.   This criterion normally goes along with the *known location* criterion. *Code removal* normally describes when an issue was addressed by only removing code—normally the developer already knew the location of the code to be removed. This criterion denotes "low" effort.

No source-code changes.   This code refer to issues that were addressed without touching any source code. This criterion normally denotes "low" effort.

Known location.   When the developers already knew exactly where to change the code to fix an issue. This criterion normally denotes "low" effort.

Reoccurring issue.   When there is evidence that an issue keeps reoccurring even after effort invested in potential fixes.

---

[15]https://issues.apache.org/jira/browse/LUCENE-1478

References

Annamaa A, Breslav A, Kabanov J, Vene V (2010) An interactive tool for analyzing embedded SQL queries. In: Asian symposium on programming languages and systems, Springer, pp 131–138

ASF (2019) Community-led development "the Apache way" https://www.apache.org/foundation/. Accessed 10 Jan 2022

Bachmann A, Bernstein A (2009) Software process data quality and characteristics: a historical view on open and closed source projects. In: Proceedings of the joint international and annual ERCIM workshops on principles of software evolution (IWPSE) and software evolution (Evol) workshops, pp 119–128

Bowen GA (2009) Document analysis as a qualitative research method. Qual R J 9(2):27–40. https://doi.org/10.3316/QRJ0902027

Braun V, Clarke V (2019) Reflecting on reflexive thematic analysis. Qual Res Sport Exerc Health 11(4):589–597

Braun V, Clarke V (2020) One size fits all? What counts as quality practice in (reflexive) thematic analysis? Qual Res Psychol 18(3):328–352. https://doi.org/10.1080/14780887.2020.1769238

Chen TH, Shang W, Jiang ZM, Hassan AE, Nasser M, Flora P (2014) Detecting performance anti-patterns for applications developed using object-relational mapping. In: Proceedings of the 36th international conference on software engineering, pp 1001–1012

Chen TH, Shang W, Jiang ZM, Hassan AE, Nasser M, Flora P (2016) Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. IEEE Trans Softw Eng 42(12):1148–1161

Cheung A, Solar-Lezama A, Madden S (2013) Optimizing database-backed applications with query synthesis. ACM SIGPLAN Notices 48(6):3–14

Cliff N (1993) Dominance statistics: ordinal analyses to answer ordinal questions. Psychol Bull 114(3):494

Cohen J (1968) Weighted kappa: nominal scale agreement provision for scaled disagreement or partial credit. Psychol Bull 70(4):213

da Costa DA, Abebe SL, McIntosh S, Kulesza U, Hassan AE (2014) An empirical study of delays in the integration of addressed issues. In: 2014 IEEE international conference on software maintenance and evolution, IEEE, pp 281–290

da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE (2016) A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Trans Softw Eng 43(7):641–657

de Almeida Filho FG, Martins ADF, da Silva Vinuto T, Monteiro JM, de Sousa IP, de Castro Machado J, Rocha LS (2019) Prevalence of bad smells in PL/SQL projects. In: 2019 IEEE/ACM 27Th international conference on program comprehension (ICPC), IEEE, pp 116–121

Dintyala P, Narechania A, Arulraj J (2020) SQLCheck: automated detection and diagnosis of SQL anti-patterns. In: Proceedings of the 2020 ACM SIGMOD international conference on management of data, pp 2331–2345

Edmundson A, Holtkamp B, Rivera E, Finifter M, Mettler A, Wagner D (2013) An empirical study on the effectiveness of security code review. In: International symposium on engineering secure software and systems, Springer, pp 197–212

Eessaar E (2015) On query-based search of possible design flaws of SQL databases. In: Innovations and advances in computing, informatics, systems sciences, networking and engineering, Springer, pp 53–60

Faroult S, Robson P (2006) The Art of SQL. https://doi.org/10.1604/9780596008949

Gaspar D, Coric I (2017) Bridging relational and NoSQL databases. https://doi.org/10.4018/978-1-5225-3385-6.ch001

Giger E, Pinzger M, Gall H (2010) Predicting the fix time of bugs. In: Proceedings of the 2nd international workshop on recommendation systems for software engineering, pp 52–56

Gousios G (2013) The GHTorent dataset and tool suite. In: 2013 10th Working Conference on Mining Software Repositories (MSR). IEEE, pp 233–236. Accessed 10 March 2022

Jin G, Song L, Shi X, Scherpelz J, Lu S (2012) Understanding and detecting real-world performance bugs. ACM SIGPLAN Notices 47(6):77–88

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining GitHub. In: Proceedings of the 11th working conference on mining software repositories, pp 92–101

Kamei Y, Matsumoto S, Monden A, Matsumoto K, Adams B, Hassan AE (2010) Revisiting common bug prediction findings using effort-aware models. In: 2010 IEEE international conference on software maintenance, IEEE, pp 1–10

Kampstra P (2008) Beanplot: a boxplot alternative for visual comparison of distributions. J Stat Softw 28(1):1–9

Karampatsis RM, Sutton C (2020) How often do single-statement bugs occur? The manysstubs4j dataset. In: Proceedings of the 17th international conference on mining software repositories, pp 573–577

Karwin B (2010) SQL antipatterns: avoiding the pitfalls of database programming. Pragmatic Bookshelf

Kononenko O, Rose T, Baysal O, Godfrey M, Theisen D, De Water B (2018) Studying pull request merges: a case study of Shopify's active merchant. In: Proceedings of the 40th international conference on software engineering: software engineering in practice, pp 124–133

Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47(260):583–621

Lohr SL (2009) Sampling: design and analysis. Brooks/Cole. https://doi.org/10.1604/9780495105275

Lyu Y, Li D, Halfond WG (2018) Remove rats from your code: automated optimization of resource inefficient database writes for mobile applications. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 310–321

Lyu Y, Alotaibi A, Halfond WG (2019) Quantifying the performance impact of SQL antipatterns on mobile applications. In: 2019 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 53–64

Mason T, Lawrence R (2005) Dynamic database integration in a JDBC driver. In: Seventh international conference on enterprise information systems (ICEIS), vol 1. Miami, USA, pp 326-333

McHugh ML (2013) The chi-square test of independence. Biochemia medica 23(2):143–149

Melton J, Simon AR (1992) Understanding the New SQL: A Complete Guide. https://doi.org/10.1604/9781558602458

Mende T, Koschke R (2010) Effort-aware defect prediction models. In: 2010 14Th european conference on software maintenance and reengineering, IEEE, pp 107–116

Mens T (2012) On the complexity of software systems. IEEE Comput Archit Lett 45(08):79–81

Miller KW, Voas J (2008) Computer scientist, software engineer, or IT professional: which do you think you are? IT Professional 10(4):4–6

Muse BA, Rahman MM, Nagy C, Cleve A, Khomh F, Antoniol G (2020) On the prevalence, impact, and evolution of SQL code smells in data-intensive systems. In: Proceedings of the 17th international conference on mining software repositories (MSR 2020), ACM Press, pp 327–338

Nagy C, Cleve A (2015) Mining Stack Overflow for discovering error patterns in SQL queries. In: 2015 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 516–520

Nagy C, Cleve A (2018) SQLInspect: a static analyzer to inspect database usage in Java applications. In: Proceedings of the 40th international conference on software engineering: companion proceedings, pp 93–96

Nagy C, Meurice L, Cleve A (2015) Where was this SQL query executed? a static concept location approach. In: 2015 IEEE 22Nd international conference on software analysis, evolution, and reengineering (SANER), IEEE, pp 580–584

Neto EC, da Costa DA, Kulesza U (2018) The impact of refactoring changes on the SZZ algorithm: an empirical study. In: 2018 IEEE 25Th international conference on software analysis, evolution and reengineering (SANER), IEEE, pp 380–390

O'Leary Z (2017) The essential guide to doing your research project (Third). SAGE Publications Ltd. ISBN: 1473952085

Pickerill P, Joshua JH, Mirosław O, Michał M, Miroslaw S (2020) Phantom: curating github for engineered software projects using time-series clustering. Empir Softw Eng 25(4):2897–2929

Podgurski A, Masri W, McCleese Y, Wolff FG, Yang C (1999) Estimation of software reliability by stratified sampling. ACM Trans Softw Eng Methodol (TOSEM) 8(3):263–283

Posnett D, Hindle A, Devanbu P (2011) Got issues? do new features and code improvements affect defects? In: 2011 18Th working conference on reverse engineering, IEEE, pp 211–215

Rahman MM, Roy CK (2014) An insight into the pull requests of GitHub. In: Proceedings of the 11th working conference on mining software repositories, pp 364–367

Rigby PC, Hassan AE (2017) What can OSS mailing lists tell us? a preliminary psychometric text analysis of the Apache developer mailing list. In: Fourth international workshop on mining software repositories (MSR'07: ICSE workshops 2007), IEEE, pp 23–23

Rith J, Lehmayr PS, Meyer-Wegener K (2014) Speaking in tongues: SQL access to noSQL systems. In: Proceedings of the 29th annual ACM symposium on applied computing, pp 855–857

Roberts JA, Hann IH, Slaughter SA (2006) Understanding the motivations, participation, and performance of open source software developers: a longitudinal study of the Apache projects. Manag Sci 52(7):984–999

Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3/4):591–611

Shepperd M (2014) Cost prediction and software project management. Softw Proj Manag Chang World, pp 51–71

Usman M, Mendes E, Weidt F, Britto R (2014) Effort estimation in agile software development: a systematic literature review. In: Proceedings of the 10th international conference on predictive models in software engineering, pp 82–91

Van Der Veen E, Gousios G, Zaidman A (2015) Automatically prioritizing pull requests. In: 2015 IEEE/ACM 12Th working conference on mining software repositories, IEEE, pp 357–361

Vandehei B, da Costa DA, Falessi D (2021) Leveraging the defects life cycle to label affected versions and defective classes. ACM Trans Softw Eng Methodol (TOSEM) 30(2):1–35

Walenstein A (2002) Cognitive support in software engineering tools: a distributed cognition framework. Ph.D. thesis, Citeseer

Wassermann G, Gould C, Su Z, Devanbu P (2007) Static checking of dynamically generated queries in database applications. ACM Trans Softw Eng Methodol (TOSEM) 16(4):14–es

Weiss C, Premraj R, Zimmermann T, Zeller A (2007) How long will it take to fix this bug? In: Fourth international workshop on mining software repositories (MSR'07: ICSE workshops 2007), IEEE, pp 1–1

Whigham PA, Owen CA, Macdonell SG (2015) A baseline model for software effort estimation. ACM Trans Softw Eng Methodol (TOSEM) 24(3):1–11

Wilks DS (2011) Statistical methods in the atmospheric sciences, 3rd edn. Academic Press, New York

Yatish S, Jiarpakdee J, Thongtanunam P, Tantithamthavorn C (2019) Mining software defects: should we consider affected releases? In: 2019 IEEE/ACM 41St international conference on software engineering (ICSE), IEEE, pp 654–665

Yilmaz M, O'Connor RV, Clarke P (2015) Software development roles: a multi-project empirical investigation. ACM SIGSOFT Softw Eng Notes 40(1):1–5

Zhao G, da Costa DA, Zou Y (2019) Improving the pull requests review process using learning-to-rank algorithms. Empir Softw Eng 24(4):2140–2170