FIXME

Muse, Biruk Asmare; Nagy, Csaba; Cleve, Anthony; Khomh, Foutse; Antoniol, Giuliano

Published in:
Empirical Software Engineering

DOI:
10.1007/s10664-022-10119-4

Publication date:
2022

Document Version
Early version, also known as pre-print

Noname manuscript No.
(will be inserted by the editor)

FIXME: Synchronize with Database

An Empirical Study of Data Access Self-Admitted Technical Debt

Biruk Asmare Muse · Csaba Nagy · Anthony Cleve · Foutse Khomh · Giuliano Antoniol

Received: date / Accepted: date

Abstract Developers sometimes choose design and implementation shortcuts due to the pressure from tight release schedules. However, shortcuts introduce technical debt that increases as the software evolves. The debt needs to be repaid as fast as possible to minimize its impact on software development and software quality. Sometimes, technical debt is admitted by developers in comments and commit messages. Such debt is known as self-admitted technical debt (SATD). In data-intensive systems, where data manipulation is a critical functionality, the presence of SATD in the data access logic could seriously harm performance and maintainability. Understanding the composition and distribution of the SATDs across software systems and their evolution could provide insights into managing technical debt efficiently. We present a large-scale empirical study on the prevalence, composition, and evolution of SATD in data-intensive systems. We analyzed 83 open-source systems relying on relational databases as well as 19 systems relying on NoSQL databases. We detected SATD in source code comments obtained from different snapshots of the subject systems. To understand the evolution dynamics of SATDs, we conducted a survival analysis. Next, we performed a manual analysis of 361 sample data-access SATDs, investigating the composition of data-access SATDs and the reasons behind their introduction and removal. We identified 15 new SATD categories, out of which 11 are specific to database access operations. We found that most of the data-access SATDs are introduced in the later stages of change history rather than at the beginning. We also observed that bug fixing and refactoring are the main reasons behind the introduction of data-access SATDs.

Keywords Data-intensive systems · Database accesses · Technical debt · Self-admitted technical debt

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

Csaba Nagy
Software Institute, Università della Svizzera italiana, E-mail: csaba.nagy@usi.ch

Anthony Cleve
Namur Digital Institute, University of Namur, E-mail: anthony.cleve@unamur.be

Foutse Khomh
Polytechnique Montréal, E-mail: foutse.khomh@polymtl.ca

Giuliano Antoniol
Polytechnique Montréal, E-mail: antoniol@ieee.org

1 Introduction

With the increasing data demand of novel technologies, modern systems often collect and process large data volumes with high velocity for various purposes. Such *Big Data* or *data-intensive systems* [15] are pervasive and virtually affect people in all walks of life [35]. They often have critical roles, too, calling for prime importance to ensure their quality. Data-intensive systems, however, have several peculiarities posing challenges to software engineering practitioners and researchers [10,12,17,35].

Developers of data-intensive systems are also often under pressure to deliver features on time. Although deadlines can increase productivity, a potential adverse side effect is decreased quality [22]. This phenomenon led to the concept of *technical debt*, *i.e.,* implementation trade-offs made by developers during rushed development tasks. Since Cunningham first described technical debt almost 30 years ago [11], many researchers have studied its impact on software development [3, 4,23,38]. In general, researchers agree that it leads to low quality (in particular maintainability), and makes further changes more expensive in the long run [24,51].

Technical debt is often admitted by the developers through comments with "todos" and "fixmes" left in the source code as reminders for the future. Such debt is referred to as *self-admitted technical debt* (SATD). Researchers often use SATD as a proxy to estimate technical debt because it can be identified by analyzing the source code [26,16] or issue reports [52].

Technical debt is pertinent to data-intensive systems too. In a recent study, Foidl et al. claim that technical debt can proliferate in data-intensive systems [12]. As they say, data-intensive systems have heterogeneous architecture divided into multiple parts (software systems, data storage systems, and data), and debt introduced in one part has unwanted effects on other parts as well. A similar phenomenon has been described by several authors [25,29,46], who found that changes in the database or application often remain unpropagated to the other side. In the end, the system's quality decays over time [46].

While these studies recognize the importance of technical debt in data-intensive systems, the problem has not received much attention. Although several researchers have investigated the detection, persistence and impact of technical debt in traditional software systems (*e.g.,* [23,4, 38,3,26,24,51]), data-intensive systems remained out of focus. In particular, we still do not know much about the prevalence and persistence of technical debt in data-intensive systems. Neither do we know their composition and the circumstances of their introduction or removal. This paper aims to fill this gap in the literature.

We conduct an empirical study to understand the prevalence and persistence of SATDs, their composition, and the circumstances of their introduction and removal. In particular, we seek answers to the following research questions.

RQ1: How prevalent are SATDs in data-intensive systems?
RQ2: How long do SATDs persist in data-intensive systems?
RQ3: What is the composition of data-access SATD?
RQ4: What are the circumstances behind the introduction and removal of data-access SATD?

We focus on data-accesses and define *data-access SATDs* as SATDs that occur in *data-access classes*, *i.e.,* classes with direct database interactions or other persistence systems via calls to driver functions or APIs. To differentiate against their counterparts, we refer to SATDs in non-data-access classes as *regular SATDs*. We are interested in data-access SATDs because mismanagement of SATDs in such classes can significantly impact the overall quality of a data-intensive system.

We examine SATDs in relational (SQL-based) and non-relational (NoSQL-based) data-intensive systems. The reason is the fundamental differences between SQL and NoSQL systems in terms

of schema, data-access approach, data representation, scalability, and the type of data they are manipulating [40–42, 49]. Relational systems have a predefined schema, use structured query language for data access, store data using tables, and capture the relationship between entities via the relationship between tables. They are vertically scalable and efficient for handling structured data. NoSQL systems have a dynamic schema, rely on document, key-value, graph, or column storage. They are horizontally scalable and efficient for handling unstructured data. SQL systems are often older and more mature compared to NoSQL systems. The conceptual differences in SQL and NoSQL systems result in differences in their APIs, affecting their data-access code, thus, the data-access SATDs too. Such differences motivated us to compare the prevalence and persistence of SQL-based and NoSQL-based data-intensive systems in our analysis. To the best of our knowledge, this is the first study on database-related technical debt that considers both relational and NoSQL software systems.

The primary contributions of this work can be summarized as follows.

1. We provide empirical evidence that SATDs are not equally prevalent between data-access and regular classes and between NoSQL and SQL systems.
2. Our results show that data-access SATDs have lower survival than regular SATDs
3. We extended the SATD taxonomy proposed by Bavota and Russo [7] with new SATD types, including 11 database access specific debts.
4. Our result also shows that data-access SATDs are introduced at later stages of software evolution, mainly during *bug fixing* and *refactoring* activities.

The rest of the paper is organized as follows. We present related work in Section 2. In Section 3, we provide background information on topic modeling and survival analysis. We describe the study methodology in Section 4, then we present the results in Section 5, and discuss their implications in Section 6. In Section 7, we identify the threats to the validity of our study. Finally, we provide concluding remarks and discuss directions for future work in Section 8.

2 Related work

Several researchers have investigated self-admitted technical debt in source code for various purposes including its identification [13, 14, 44, 16, 26, 53, 54, 1], removal [57, 28], prioritization [3, 2, 20], recommendation when to admit SATDs [55], or the analysis of its impact on source code quality [51] – to mention a few examples.

In this section, we first present an overview of recent literature and surveys related to technical debt. Then we summarize previous empirical studies on self-admitted technical debt. Finally, we discuss closely related papers focusing on technical debt in databases or data-intensive systems.

2.1 Surveys and Literature Reviews on Technical Debt

Li et al. conducted a systematic mapping study on technical debt and its management [23]. They examined 49 papers, classified technical debts into ten categories and identified eight activities and 29 technical debt management tools.

Rios et al. performed a tertiary study and evaluated 13 secondary studies dating from 2012 to March 2018 [38]. As a result, they developed a taxonomy of technical debt types and identified a list of situations in which debt items can be found in software projects.

Alves et al. performed a systematic mapping study by evaluating 100 studies dating from 2010 to 2014 [4]. They also proposed a taxonomy of technical debt types and created a list of indicators to identify technical debt.

Alves et al. [3] conducted a systematic literature review identifying 29 technical debt prioritization approaches. Among the 29 approaches, 70.83% address a specific type of technical debt, while the remaining approaches can be applied to any kind of technical debt. 33.33% of the approaches address code debt, 16.67% address design debt, 12.5% address defect debt and 1% of the approaches is shared by SATDs, database normalization debt, requirement debt and architectural debt. Among all approaches, 54.17% consider value and cost as prioritization decision factors, 29.17% rely on value only, and 16.67% of approaches are based on value cost and constraint.

Sierra et al. [43] present a survey of SATD studies from 2014 to 2019. They identified three main categories of research contributions: (1) papers that focus on the *detection* of SATD, (2) papers that aim to improve the *comprehension* of SATD, and (3) papers that focus on the *repayment* of SATD.

Interestingly, while the above literature reviews identified various types of technical debt (*e.g.,* service debt related to web services), none of them explicitly mention database communication-related debts in their taxonomies as well as evolution and management. As they constitute an overview of the state-of-the-art technical debt research, the *recent surveys indicate a lack of studies on database-related technical debt.* We address this gap in the literature by extending the SATD taxonomy [7] with database access debt. Furthermore, we study the evolution and management of data-access SATDs to complement the state of the art.

2.2 Empirical Studies on Self-Admitted Technical Debt

Potdar and Shihab [36] used source-code comments to study self-admitted technical debt in four large open-source software projects. They found that different types of self-admitted technical debts exist in up to 31% of the studied project files. They showed that developers with higher experience tend to introduce most of the self-admitted technical debt and that time pressures and complexity of the code do not correlate with the amount of self-admitted technical debt introduced. They also observed that between 26.3% and 63.5% of self-admitted technical debt are removed from the projects after their introduction.

A large-scale empirical study on removing self-admitted technical debt was performed by Maldonado et al., who examined 5,733 SATD removals in five large open source projects [28]. They found that the majority (40.5–90.6%) of SATD comments were removed from the systems, and the median amount of time that self-admitted technical debt stayed in the project ranged between 82–613.2 (18.2–172.8 days on average). While the above studies address the prevalence and evolution of SATDs, they rely on four or five subject systems, limiting the generalization of the results, especially to data-intensive systems. We investigate the prevalence and evolution of data-access SATDs using 102 data-intensive subject systems.

Bavota and Russo [7] conducted a differentiated replication of the work of Potdar and Shihab [36]. They considered 159 software projects and investigated the diffusion (prevalence) and evolution of self-admitted technical debt and its relationship with software quality. Their results show that (1) SATD is diffused in software projects; (2) the most diffused SATDs are related to code, defect, and requirement debt; (3) the amount of SATD increases over time due to the introduction of new SATDs that developers never fix; and (4) SATD has very long survivability (over 1,000 commits on average). They also proposed a SATD taxonomy, which is used as a base for this work. We extended their taxonomy by identifying data-access-specific SATDs.

Wehaibi et al. [51] studied the relation between SATD and software quality in terms of defects and maintenance effort. They identified SATDs in five popular open-source projects

using pattern-based approaches. They found that the defectiveness of files increased after the introduction of SATDs and that changes were more difficult when they were related to SATDs.

Kamei et al. assessed ways to measure the interest of SATDs as a function of LOC and fan-in measures [20]. They examined JMeter as a case study and manually classified its SATD comments, then compared the metric values after the introduction and removal of SATDs to compute their interest. They found that up to 44% of SATDs have positive interest implying that more effort is needed to resolve such debt.

Zampetti et al. performed a quantitative and qualitative study of how developers address SATDs in five Java open source projects [56]. They found that a relatively large percentage (20%–50%) of SATD comments are accidentally removed while entire classes or methods are dropped. Developers acknowledged in commit messages the SATD removal in only 8% of the cases. They also observed that SATD is often addressed by specific changes to method calls or conditionals, not just complex source code changes. Like Zampetti et al., we utilize the information obtained from commit messages to understand why data-access SATDs are introduced or removed.

The work of Wehaibi et al. [51] and Zampetti et al. [56] motivated us to investigate the circumstances behind the introduction, evolution, and removal of data-access SATDs as such factors affect the interest of technical debt. We are interested in generalizing the findings of [56] to the context of data-intensive systems.

2.3 SATD detection approaches

Most of the SATD detection approaches are either pattern- or machine-learning-based. Initial methods for detection were pattern-based. Machine learning approaches were introduced more recently to improve the performance of detection approaches and tools.

2.3.1 Pattern-based SATD detection

De Freitas et al. [13] proposed a contextualized vocabulary model to identify technical debt using source code analysis. The model consists of software-related terms, adjectives that describe the terms, verbs to model actions in comments, adverbs, and tags such as Fixme and Todo. The combined terms can be used for searching comments in a pattern-based approach. They tested the feasibility of their approach on jEdit and Apache Lucene and identified technical debt in various categories.

As an extension of the work of De Freitas et al. [13], De Farias et al. conducted an empirical study on the effectiveness of contextual vocabulary models (CVM) [14]. Besides evaluating the accuracy of the pattern-based approaches, they studied the impacts of language skills and developer experience on finding SATDs using a controlled experiment. Their result shows that the accuracy of the pattern-based approach looks promising, but it needs further improvement. English reading skills affected the identification of SATDs using pattern-based techniques.

2.3.2 Machine-learning-based SATD detection

Maldonado et al. proposed NLP based approach to identify SATDs [44] automatically. Their approach can detect design and requirement SATDs. Furthermore, they built a manually labeled dataset of 62566 SATD comments. This dataset is used as a benchmark in most of the subsequent studies. They proposed a multi-class regression model using their dataset. They evaluated their

approach and achieved an F1 measure between 40% to 60%. They observed that words related to sloppy code indicate design SATD while words related to incomplete code are associated with requirement SATD.

Huang et al. proposed a machine-learning-based detection approach that combines the decisions of multiple Naive-Bayes-based classifiers into a composite classifier using majority vote [16]. The comments from source codes are represented using vector space modeling (VSM), where features are selected utilizing Information Gain. They achieved an average F1-Score of 73.7%. Liu et al. [26] proposed a SATD detector tool which is a concrete implementation of Huang et al. [16] approach. They provided this tool as a Java back-end library implementing the model to train and classify comments and the corresponding Eclipse plugin as a front end. We also used this tool to detect SATDs in our subject systems due to its state-of-the-art detection performance and the availability of concrete implementation of the detection approach as a Java API and Eclipse plugin.

Zampetti et al. [55] presented TEDIOuS (TEchnical Debt IdentificatiOn System), a machine learning approach that provides recommendations to developers about "technical debt to be admitted". The method relies on source code structural metrics, readability metrics, and information from static analysis tools. They evaluated TEDIOus using nine open-source subject systems and achieved an average precision of 67% and recall of 55%.

2.4 Technical Debt in Data-Intensive Systems

Albarak and Bashoon defined the concept of database design debt as "*the immature or suboptimal database design decisions that lag behind the optimal/desirable ones, that manifest themselves into future structural or behavioral problems, making changes inevitable and more expensive to carry out*" [1]. They develop a taxonomy of debts related to the conceptual, logical, and physical design of a database. For example, they claim that ill-normalized databases (*i.e.,* databases with tables below the fourth normal form) can also be considered technical debt [2]. To tackle this specific type of debt, they propose an approach to prioritize tables that should be normalized.

Foidl et al. claim that technical debt can be incurred in different parts (*i.e.,* software systems, data storage systems, data) of data-intensive systems and different parts can further affect each other [12]. They propose a conceptual model to outline where technical debt can emerge in data-intensive systems by separating them into three parts: software systems, data storage systems and data. They present two smells as examples. Missing constraints, when referential integrity constraints are not declared in a database schema; and metadata as data, when an entity-attribute-value pattern is used to store metadata (attributes) as data. While this study provided a conceptual model for components of data-intensive systems prone to technical debt, it did not provide empirical evidence for the existence of the technical debt. We contribute to addressing this gap by investigating SATDs in data-intensive systems.

Weber et al. [50] also identified relational database schemas as potential sources of technical debt. In particular, they provided a first attempt at utilizing the technical debt analogy for developing processes related to the missing implementation of implicit foreign key (FK) constraints. They discuss the detection of missing FKs, propose a measurement for the associated TD, and outline a process for reducing FK-related TD. As illustrative case study, they consider OSCAR, a large Java medical record system used in Canada's primary health care.

Ramasubbu and Kemerer [37] empirically analyze the impact of technical debt on system reliability by observing a 10-year life cycle of a commercial enterprise system. They also examine the relative effects of modular and architectural maintenance activities in clients. They conclude that technical debt decreases the reliability of enterprise systems. They also add that modular

maintenance targeted to reduce technical debt is about 53% more effective than architectural maintenance in reducing the probability of a system failure due to client errors.

2.5 Summary

The various studies and approaches discussed above constitute an extensive and sound basis for measuring, detecting and removing (self-admitted) technical debt. To the best of our knowledge, this paper is the first large-scale study investigating the prevalence, nature, and evolution of self-admitted technical debt in *data-intensive* systems in general and in *data-access* code in particular. It is also the first to study database-related technical debt in both relational and NoSQL software systems. In addition, it proposes an extension of an existing SATD taxonomy [7] to incorporate data-access related SATDs.

3 Background

This section provides a background on the topic modeling and survival analysis techniques used in our study.

3.1 Topic Modelling

Topic modeling [34] is one of the unsupervised machine learning techniques that, given a set of documents (document corpus), can detect word and phrase patterns and cluster the documents based on word similarity. In our case, the corpus will be our dataset, and each comment will be one document in the corpus. Topic modeling works by counting the words and grouping documents with similar word patterns. Topic modeling is one of the frequently used techniques in *natural language processing* (NLP).

Latent semantic analysis (LSA) [34] and *latent Dirichlet allocation* (LDA) [8] are commonly used topic modeling algorithms. We also rely on LDA to assign topics to a set of words assuming that the arrangement of words determines the topic. LDA model is trained using a tokenized and pre-processed set of documents. After the LDA is trained, it can assign a document to a topic group with a certain probability. In this paper, we use LDA to cluster comments based on similarity so that our sampled data for manual analysis is not biased to a specific topic. LDA has hyper-parameters such as the number of topics, alpha, and beta to control the similarity levels that affect the model's performance. The first one determines the *number of topics* generated by LDA after training. It can take any positive integer value. An insufficient value results in a too general model that makes topic interpretation difficult. An excess number of topics creates many topics that are too fine-grained for classification and subjective evaluation [58]. *Alpha* controls the document topic density. A higher alpha makes the documents contain many topics. On the contrary, a smaller alpha makes the documents have a small number of topics. *Beta* controls the topic word density determining the number of words in the corpus associated with a topic. The higher the *beta* value, the more words are associated with a topic. All those parameters need to be tuned using the target dataset by optimizing for the best performance of the LDA model.

Performance evaluation of LDA: A topic model can be evaluated by human judgment and intrinsic methods such as *perplexity and coherence*. *Perplexity* measures how well a probability model predicts a sample. It is computed by assessing the LDA model with unseen or held-out data. The lower the perplexity, the better the performance of the model. While perplexity measures the prediction of the LDA model, it does not evaluate the interpretation of the generated

topics [9]. Another approach is to use coherence for evaluation. The coherence score is computed following segmentation, probability estimation, confirmation measure, and aggregation [39]. Coherence score is calculated by summing the scores of a pair of words that describe a topic on the assumption that words that often appear together in the document are more coherent. Coherence takes a value between 0 and 1. The higher the score, the better the model.

3.2 Survival Analysis

Survival analysis [30] is a statistical analysis technique that provides the expected time for an event's occurrence. *Time to event* and *status* are two important variables for survival analysis. To compute each variable, we first need to define an event of interest that depends on the problem we want to analyze. In our case, an event of interest is the *removal of an SATD.*

Time to event (T) is defined as the time interval between the starting of observation (the first instance of the SATD) and the occurrence of an event of interest or the censoring of data. Time to event T is a random variable with only positive values and can be measured in any unit [30]. The most common approach is to use time in minutes, hours, days, months, or years. However, we will use the number of commits to consider that the actual time may not correctly reflect software evolution compared to the number of commits. Projects have different activities at different times. Commits could be made more frequent at specific periods of time and less frequent at other times. Using time for T in those cases has a limited capability to reflect project evolution. On the contrary, the *number of commits* directly measures the project activity regardless of activity variation in some periods of time.

It is important to define an observation window and flag events outside it as censored. In our case, we define the observation window to cover all our snapshots of the subject systems. We flag SATDs that persist in the latest snapshots as censored since we do not know if the event of interest (*i.e.,* the removal of the SATD) will occur or not. Similarly, when an entire source file with one or more SATD comments is deleted within the observation window, we flag the SATDs as censored. The reason is that in this case, it cannot be determined whether the SATDs are removed intentionally or only because of the file deletion. This is also supported by the observation of Zampetti et al., who found that 20%–50% of the removals of SATDs are accidental and are even unintended [56].

Survival analysis takes a boolean variable called *status* to distinguish between censored data and non-censored data. For instance, it takes a value of 1 when the event of interest occurred and 0 otherwise.

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

After we computed T and *status*, we can choose our survival estimator. We selected one of the commonly used survival estimators, the Kaplan-Meier estimator [21]. The Kaplan-Meier estimation is computed following Equation 1. t_i is the time duration (in the number of commits) up to event-occurrence (removal of SATD) point i, d_i is the number of event occurrences up to t_i, and n_i is the number of SATDs that survive just before t_i. n_i and d_i are obtained from the input data.

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

3.3 Metrics for measuring developers activity in time

Code repositories track changes in software artifacts through commits. The distribution of commits in time co-relates to developer activity and is used to study the evolution of software and the associated technical debts (*e.g.,* [19,48]). For our analysis, we took a snapshot of projects every 500 commits. We provided the mean, standard deviation, and 95% confidence interval of the commit time span for each SQL system[1] and NoSQL system[2] respectively in the replication package.

Furthermore, Figure 1 shows the distribution of the average time interval between successive snapshots of our subject systems. The average time interval between successive snapshots is 535 days for SQL subject systems and 423 days for NoSQL subject systems. The variation in time interval across and inside subject systems led to other approaches for measuring developer activity, such as using the number of commits. While we use the number of commits to measure developer activity in our analysis, the above typical values can be used to interpret the commit time span in days.

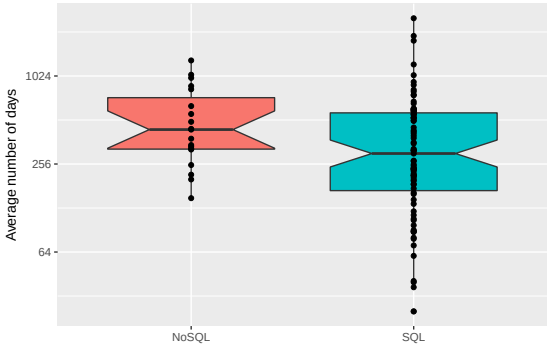


Fig. 1 The distribution of average time interval between successive snapshots taken every 500 commits for SQL and NoSQL subject systems. The y-axis time unit is in days.

4 Study Method

In this section, we discuss the approach we followed to answer our research questions. Fig. 2 gives an overview of our approach, including the subject system identification, data collection, and data analysis procedures. Each step is described in the coming sub-sections.

4.1 Data collection

We gathered SQL and NoSQL systems from GitHub for our study. We followed the following steps to identify subject systems.

Phase 1: We ran a GitHub search using the keywords related to the persistence libraries used by SQL projects such as SQLite, JDBC, Hibernate, and JPA. Those keywords are used

[1] `https://bit.ly/2YLrLnU`
[2] `https://bit.ly/3jj5JAH`

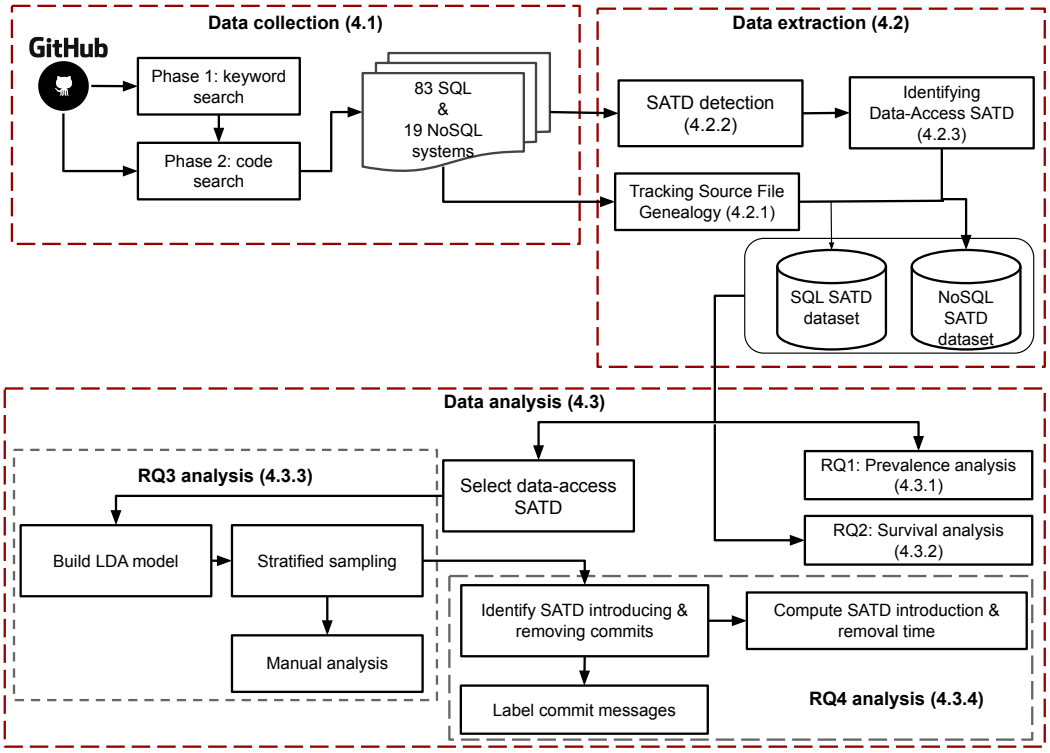


Fig. 2 Overview of the study method. We provided the subsection and sub-subsection numbers for easier matching with the description.

assuming that projects that mention those libraries in the project name, description or readme file have a high chance of being data-intensive or having significant data-access code.

For NoSQL projects, we first collected NoSQL database management systems popular in open-source projects such as MongoDB, Redis, Riak and Neo4J. The database systems are collected from the supported databases of Eclipse JNoSQL,[3] a popular Java framework in the Eclipse ecosystem that streamlines the integration of Java applications with NoSQL databases. Currently, JNoSQL supports around 30 databases. The complete list of the databases is available in our replication package [31]. We ran a GitHub search for projects mentioning these database engines.

To avoid "toy" projects, we set the following criteria for the projects: (1) they had to have at least one open issue, (2) more than 1,000 commits, and (3) at least one recent commit within one year from the time of data collection (*i.e.,* 2020).

Phase 2: We ran a code search on subject systems using the GitHub code search API [18]. We particularly looked for import statements for SQL projects corresponding to the persistence technologies such as Android SQLite API, JDBC, and Hibernate. The import statements were identified in the work of Nagy et al. [33]. The import statements include, among others, `android.database.sqlite`, `android.database.DatabaseUtils`, `org.hibernate.Query`, `org.hibernate.SQLQuery`, `java.sql.Statement`, `javax.persistence.Query`, `javax.persistence.TypedQuery`, `org.springframework`.

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

Similarly, for NoSQL projects, we collected a list of import statements that are used to access NoSQL persistence systems, *e.g.,* `com.mongodb.MongoClient`, `org.neo4j.driver`, `org.apache.hbase`. To determine the imports, we started with the list of supported NoSQL databases from JNoSQL. For each database, we explored code snippets provided as instructions to connect that database to Java applications. We collected the import statements mentioned in such snippets to compile the list of NoSQL import statements. We ran a code search on the identified projects and counted the import statements for each project.

We were finally left with 83 SQL and 19 NoSQL subject systems with the SQL and NoSQL import statements as mentioned above. The complete list of the projects and import statements is available in our replication package [31].

4.2 Data extraction

4.2.1 Tracking Source File Genealogy

To ensure reliable evolution analysis, we need to keep track of all subject systems' source code genealogy. We extracted file genealogy information using the `git diff` command. This command compares two snapshots and reports the added, modified, deleted and renamed files. Specifically, we used `git diff --name-status sha1 sha2`, where `sha1` and `sha2` are commit ids of the versions to be compared. `git diff` provides a numerical estimation of rename operations, which indicates the similarity percentage. In this work, renames with similarity thresholds greater than 70% are considered true renames. A similar threshold was used in previous studies too [19,32]. We tagged each source file with a unique ID generated from our file genealogy tracking database.

4.2.2 SATD Detection

Due to the large number of subject systems with a large number of commits, we took snapshots of each system's every 500^{th} commits. Another approach would have been to select only a few projects and study every commit of each subject system. However, our research is exploratory and, therefore, we emphasize the generalizability of our results and conclusions. A similar choice was made in other studies as well [6,32].

We used the SATD detection tool by Liu et al. [26]. This tool uses a machine learning-based detection approach that combines the decisions of multiple Naive-Bayes classifiers into a composite classifier using a majority vote. During the tool's training phase, the source codes comments are represented using vector space modeling (VSM), and features are selected from VSM using information gain. The details of their approach are discussed in [16]. The tool has a Java API as well as an Eclipse plugin to support developers. Given a source code comment, the tool returns a boolean indicating whether it is a SATD comment or not. We chose this detection tool because it has the highest accuracy (average F_1 score of 73.7%) among different approaches, and the rest of the approaches were not realized as a tool to the best of our knowledge.

SATD detection was carried out in two phases. In the first phase, we extracted the comments of each snapshot of all the projects using srcML.[4] SrcML initially converts the source code into XML format. The comments can then be identified by running XPath queries. In the second phase, we run the SATD detection on the identified comments. The output of the SATD detection tool is binary: it classifies the comment as SATD-related or not.

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

4.2.3 Identifying Data-Access SATD

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

4.2.4 Study Dataset

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

4.3 Data analysis

4.3.1 RQ1: How prevalent are SATDs in data-intensive systems?

To answer RQ1, we computed the total number of data-access SATD comments and non-data-access SATDs for both SQL and NoSQL subject systems. We collected the number of SATDs for each snapshot of the subject systems' change history. We used violin plots to show how the prevalence of SATDs change as systems evolve and compared data-access and regular SATDs as well as SQL and NoSQL systems.

4.3.2 RQ2: How long do SATDs persist in data-intensive systems?

To answer this research question, we analyzed the persistence of SATDs using survival analysis. There are two cases when we automatically check if SATDs in File X are addressed between two versions A and B. Case 1: if an SATD comment in File X is similar between version A and B, we consider it as "not fixed" at version B. Case 2: if the comment found in version A is missing in version B, we consider it "fixed" at version B. We choose the number of commits over time in days for the survival analysis because different projects have different activities in time. As we discussed before (see Section 3.2), the number of commits suits better than time for our purpose to reflect the projects' activity [7].

We used the Kaplan-Meir curve to visualize the survival of subject SATDs. The Kaplan-Meir curve shows the survival probability $S(t)$ of a given SATD at a time t. We define the addressing of a SATD as our event of interest. The occurrence of this event determines the survival probability. SATDs that persisted up to the latest versions and those removed with the source files are flagged as censored (see Section 3.2).

4.3.3 RQ3: What is the composition of data-access SATD?

To answer this research question, we first identified unique data-access SATD comments in our dataset. We built an LDA topic model on the dataset to generate the strata needed for stratified sampling. Finally, We conducted a manual analysis on the sample SATD comments. We provide a detailed description in the following paragraphs.

Build LDA model: We then applied common NLP preprocessing techniques. In particular, we removed punctuation, common English stop words, and the words "todo" and "fixme," as they are very common in most comments.

Then, we applied lemmatization and stemming using the Python NLTK library. The final output of this pre-processing is a tokenized comment.

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

Stratified sampling: We prepared a dataset for manual analysis using stratified sampling from each LDA topic group. The dataset contains 183 data-access SATD comments for SQL systems and 178 data-access SATD comments for NoSQL systems. We used stratified random sampling to pick representative samples from each LDA topic group or cluster.

Manual analysis: The manual analysis was conducted using deductive coding to assign themes to the comments. We started with the themes identified by Bavota and Russo [7] and extended them with themes specific to data access. To have a common interpretation of the labels among authors, we conducted iterative sessions to label sample SATDs and resolve conflicts. After that, the first author labeled all the 361 SATDs, which were then divided into three sets and reviewed by three more authors to ensure that at least one additional person checks each label. Finally, all conflicts were resolved through face-to-face discussions.

During the labeling process, we found some comments that were identified as SATD comments by the detector tool but were not related to technical debt. Recall that Liu et al. reported an average F_1 score of 73.7% for the tool [26]. A common reason was that they contained one of the keywords (*e.g.,* "fix"), but the developer meant it for a different purpose (*e.g.,* "*// import release fix into the release branch*"[5]). We found 105 instances (29%) of these comments and marked them as *false positives*.

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

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

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

For multi-label comments, we cannot identify one specific category. Hence we exclude them for RQ3 but keep them for the evolution-related research question RQ4. After we excluded false positives, unclear comments and multi-label comments, the final dataset contained a total number of 240 data-access SATD comments.

4.3.4 RQ4: What are the circumstances behind the introduction and removal of data-access SATD?

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

Identify SATD introducing and removing commits: Using this labeled data from RQ3, we extracted the commits that introduced the comments and commits that removed them from the change history of the subject systems. We used the PyDriller repository mining framework [45] for our analysis. PyDriller is used to analyze both local and remote repositories and extract information related to their change history. We looked for the *SATD introducing commit* given the path of a file by looking at the change history starting from the beginning to the end and looking for the first occurrence of the SATD under study. Similarly, we looked for the *SATD removal commit*, the commit in which a SATD is removed from the system, by looking for the first commit in which the SATD is no longer present given that the SATD occurred in the previous versions. To check if the SATD is removed together with the hosting class, we also keep track of the commit where the hosting class is removed (if it is removed).

When are data-access SATDs introduced or removed? For our purpose, the number of commits is better than the absolute time at reflecting software evolution (see Section 3.2). Hence, we measure introduction time and removal time in terms of number of commits.

We define *introduction time* (t'_i) as the number of commits that occurred before and including the first occurrence of the SATD under study.

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

Since the total number of commits varies across the projects, we normalize the *introduction time* and *removal time* with the total number of commits for each subject system (see Equations 2 and 3). We use a similar normalization for the removal time. For example, a SATD introduction time of 20% for a project with 1,000 commits means the SATD was introduced in the 200^{th} commit from the beginning. The smaller the value, the closer the introduction of SATD to the early stages of the project evolution and vice versa.

$$\textit{Introduction time} = \frac{t'_i \cdot 100}{\textit{Total number of commits}}, \tag{2}$$

$$\textit{Removal time} = \frac{t'_r \cdot 100}{\textit{Total number of commits}} \tag{3}$$

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

Why are data-access SATDs introduced or removed? To investigate why data-access SATDs are introduced, we collected the commit messages of SATD introduction and removal commits, then manually categorized their goal or purpose. We use similar categories to Tufano et al. [47]: *bug fixing*, *enhancement*, *new feature*, and *refactoring*. In our case, we added *merging* and *multiple goals* to account for merging commits and commits whose messages have more than one goal. In this way, the commit goal can be mapped to more than one of the categories from Tufano et al.

Bug fixing commits mention that the commit was made to fix an existing bug or issue. *Enhancement* commits aim at enhancing existing or already implemented features. Commits with the goal *new feature* describe their goal as introducing or supporting a new functionality. Commits that mention refactoring operations are categorized under *refactoring*. Finally, commits made for merging pull requests and branches are categorized under *merging*. We labeled SATD-removing commits similarly.

5 Study Result

In this section, we present the results of our study. The raw results and corresponding analysis are reported for each research question.

5.1 RQ1: How prevalent are SATDs in data-intensive systems?

In this subsection, we present the prevalence of SATD in SQL and NoSQL systems.

We also show how data-access and regular SATDs evolve in multiple snapshots of the systems.

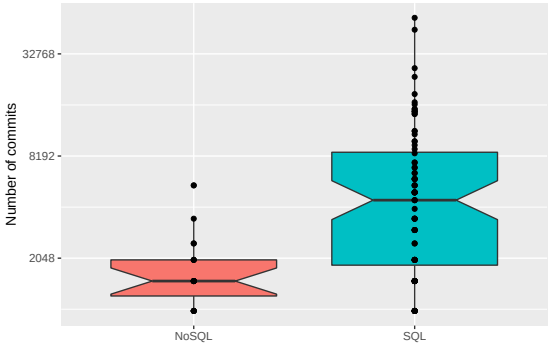


Fig. 3 Distribution of the number of commits in SQL and NoSQL subject systems. The y-axis is on a log scale.

Fig. 3 shows the distribution of the number of commits for SQL and NoSQL systems.

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

The quantile analysis of the distribution of commits shows that 25% of the projects have less than 1,501 commits, 50% of the projects less than 3,001, and 75% of the projects less than 6,751 commits. We grouped the projects based on the quantiles into three for the purpose

Table 1 Project groups

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

Table 2 Summary of the distribution of data-access and regular SATDs over the number of commits in Group 1 subject systems

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

Table 3 Summary of the distribution of data-access and regular SATDs over the number of commits in Group 2 subject systems

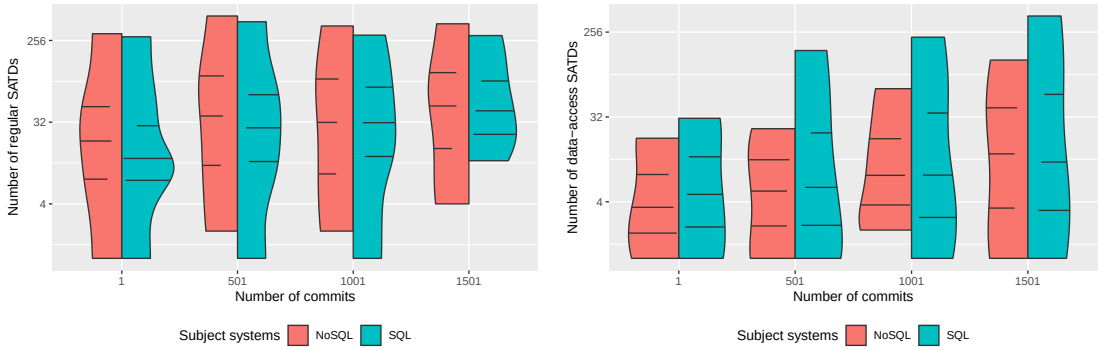
		Data-access SATD						Regular SATD					
Commit	System	Min	25%	Mean	**Median**	75%	Max	Min	25%	Mean	**Median**	75%	Max
1	NoSQL	0	0	0.57	**0**	0	4	4	5	21.14	**10**	29	66
	SQL	0	0	6.89	**0**	0.25	203	1	7	64.61	**24**	86.25	580
1001	NoSQL	0	0	7.00	**4**	12	21	2	7.5	30.86	**15**	46.5	91
	SQL	0	0	8.14	**0**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

of visualization. Table 1 presents a summary of the systems in each group. For example, all projects with a maximum of 1,500 commits are included in $Group_1$, including 12 NoSQL and 21 SQL subject systems.

Tables 2, 3 and 4 show the summary of the distribution of SATDs in our subject systems by the project groups. The distribution was computed over the snapshots of the subject systems. Fig. 4a shows the distribution of regular SATDs in $Group_1$. We observe that the number of regular SATDs increases for SQL systems as the number of commits increases. For NoSQL systems, an increase in the SATDs is observed, moving from 1 to 501 and 1,001 to 1,501. The median of regular SATDs in NoSQL systems (23.5, 38.5, 71) is higher than in SQL systems (12, 28.5, 40) for snapshots at commits 1 and 501 and 1,501, respectively. The highest number of

Table 4 Summary of the distribution of data-access and regular SATDs over the number of commits in Group 3 SQL subject systems

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



(a) Regular SATD (b) Data-access SATD

Fig. 4 Prevalence of regular and data-access SATD in $Group_1$. The horizontal lines in this and subsequent violin plots show the 25%, median, and 75% quantiles respectively from bottom to top.

regular SATDs (477) was observed at the 501^{st} commit of a NoSQL system, *Bboss*.[6] Bboss is a framework that provides API support for developing enterprise and mobile applications.

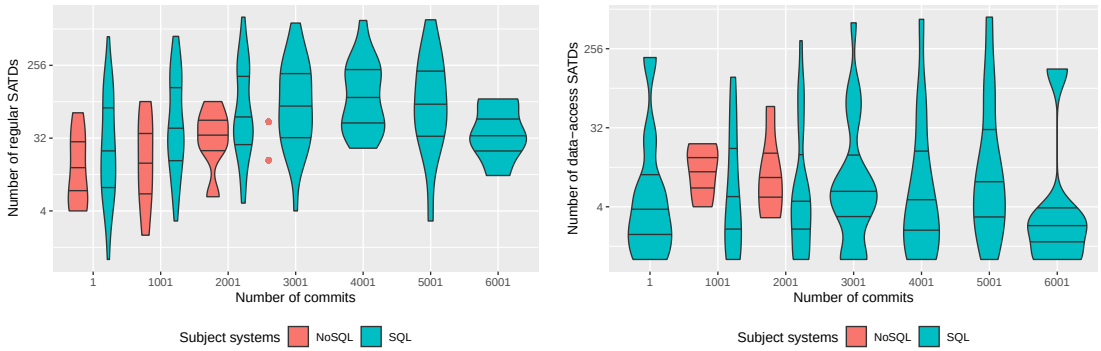
Fig. 4b shows the distribution of data-access SATDs in $Group_1$. The number of data-access SATDs in $Group_1$ increases with the number of commits. The median data-access SATD for SQL systems is 0, 1.5, 4, and 5.5 for commits 1, 501, 1,001 and 1,501. For NoSQL systems, the median is 0, 1, 4, and 22 for commits 1, 501, 1,001 and 1,501, respectively. We can see that the median of data-access SATDs is roughly similar between SQL and NoSQL subject systems except for commit 1,501, where we observe a large difference in magnitude between SQL and NoSQL subject systems. The highest number of data-access SATDs (380) was observed at commit 1,501 by the SQL subject system *Blaze-persistence*.[7] Blaze-persistence is a criteria API provider project for applications that rely on JPA for data persistence.

Fig. 5a shows the distribution of regular SATDs in $Group_2$. We observe an increasing trend in the number of regular SATDs for both SQL and NoSQL systems. SQL systems have a higher median number of regular SATD in all snapshots. The median number of regular SATD of SQL systems is 24, 40, 51 and 87 for commits 1, 1,001, 2,001 and 3,001, respectively. For NoSQL systems, the median is 10, 15, 26 and 34, respectively. The maximum number of regular SATD (1,015) was registered in an SQL system, *Jena-sparql-api*,[8] at commit 2,001. *Jena-sparql-api* provides a SPARQL processing stack for building Semantic Web applications.

[6] https://github.com/bbossgroups/bboss

[7] https://github.com/Blazebit/blaze-persistence

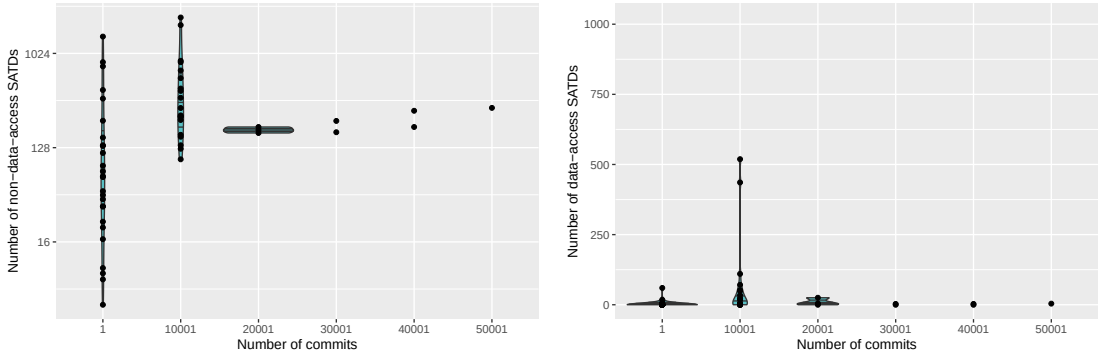
[8] https://github.com/SmartDataAnalytics/jena-sparql-api



(a) Regular SATD (b) Data-access SATD

Fig. 5 Prevalence of regular and data-access SATD in $Group_2$.

We observe a similar trend of increase in the number of data-access SATDs on $Group_2$, as shown in Fig. 5b. The median number of data-access SATD in NoSQL systems is 0, 4, and 7 for commits 1, 1,001, and 2,001. SQL systems have a median number of data-access SATD 0, 0, and 2, respectively. The largest data-access SATD (588) was registered at commit 5,001 by SQL system *Threadfix*,[9] a software vulnerability management application.



(a) Regular SATD (b) Data-access SATD

Fig. 6 Prevalence of regular and data-access SATD in $Group_3$.

Fig. 6 shows the distribution of regular and data-access SATD in $Group_3$. We only have SQL systems in $Group_3$. After commit 10,001, we have two projects where we observe SATD, and only one project, WordPress-Android,[10] remains after commit 20,001. The violin plot is not needed for such cases. Fig. 6a shows that the number of regular SATDs rises between commit 1 (median=67.5) and commit 10,001 (307), then decreases at 20,001 (189). The most significant regular SATDs (2,263) were observed at version 10,001 in *ControlSystemStudio*,[11] a repository of

[9] https://github.com/denimgroup/threadfix
[10] https://github.com/wordpress-mobile/WordPress-Android
[11] https://github.com/ControlSystemStudio/cs-studio

applications to operate large-scale industrial control systems. In Fig. 6b, we can see an increasing median number of data-access SATDs (0, 18) at commits 1 and 10,001.

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

5.2 RQ2: How long do SATDs persist in data-intensive systems?

We conducted a survival analysis to see the persistence of data-access SATDs. In particular, we plot the Kaplan-Meier curve for both SQL and NoSQL systems.

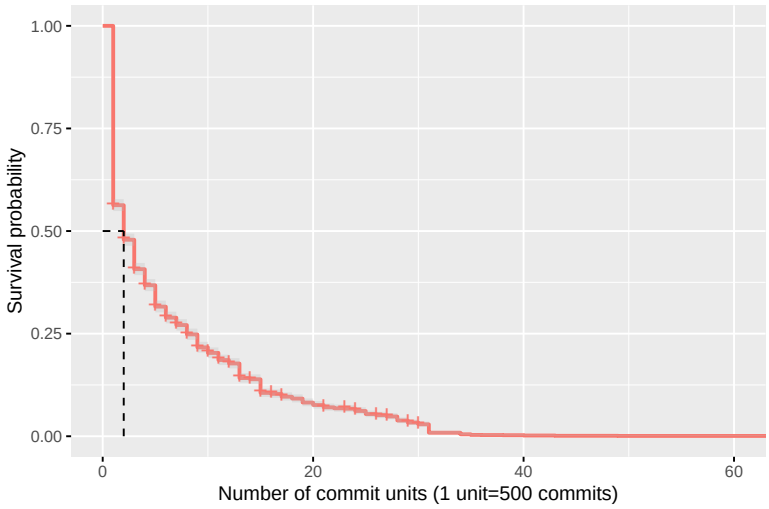


Fig. 7 Kaplan–Meier survival curve for data-access SATDs in SQL subject systems. The x-axis is the number of commits. The censoring time and confidence intervals are marked on the plot. The Logrank test's p-value is indicated.

Fig. 7 shows the survival probability of data-access SATDs in SQL projects. The median survival is 1,000 commits. Given the average value of 500 commit time span of 535 days for SQL subject systems, described in Subsection 3.3, the average median survival time is 2.93 years. The steeper slope before 10,000 commits has two potential explanations. One possibility is that several data-access SATDs are fixed/censored at the early stages of the projects. Alternatively, several subject systems have a small number of commits. The distribution of the total number of commits (median=3,729, mean=7,005, skewness=3.07) of SQL subject systems is right-skewed. Hence, the steep slope is not likely due to small project activities. The number of "data-access SATD fixed" events is 3,914, with the remaining 608 being censored. This shows that many SATD comments are introduced and fixed at the early stages of the projects.

Fig. 8 shows the survival probability of data-access SATDs in NoSQL subject systems. The median survival time of NoSQL data-access SATDs is 1,000 commits (2.3 years using an average 500 commit time span for NoSQL subject systems as described in Subsection 3.3). The number of events is 391 out of 441, with the remaining data being censored. The number of commits of

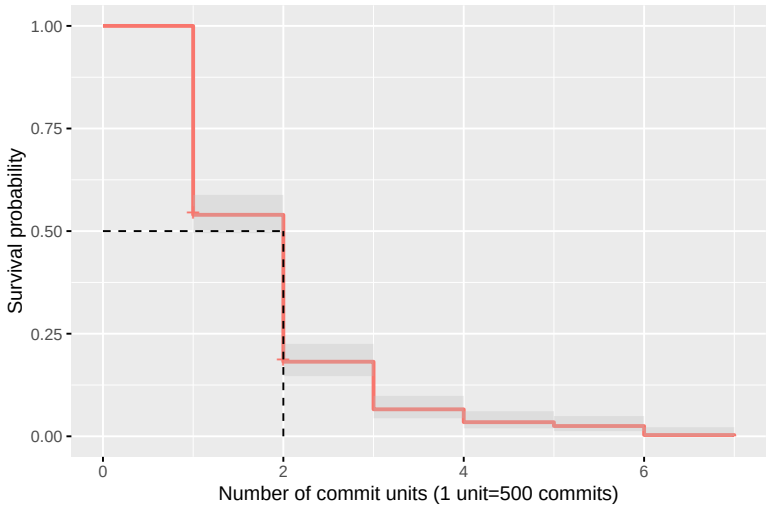


Fig. 8 Kaplan–Meier survival curve for data-access SATDs in NoSQL subject systems. The x-axis represents the number of commits. The censoring time and confidence interval are marked on the plot. The Logrank test's p-value is indicated.

NoSQL projects has a right-skewed distribution (median=1,927, mean=2,114, skewness=2.16). The smaller median survival value aligns with the smaller median number of commits of NoSQL subject systems.

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

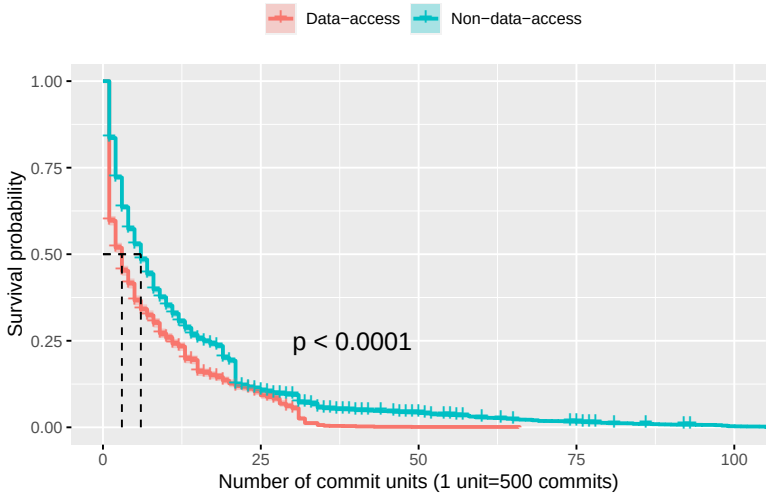


Fig. 9 Kaplan–Meier survival curve for SQL subject systems by grouping them into data-access and regular SATD comments. The x-axis represents the number of commits. The censoring time is marked on the plot.

Fig. 9 compares the survival curves of data-access and regular SATD comments in SQL systems. This comparison provides an insight into the prioritization of addressing technical debt. Data-access comments have a lower survival curve compared to their regular counterparts. We run the Log-Rank test to compare the survival curves statistically. The p-value of the log-rank test is $< 2e-16$. Hence, we can reject the null hypothesis that there is no difference between the survival curves of data-access and regular SATD comments. Data-access SATDs tend to get more priority in addressing compared to regular SATDs.

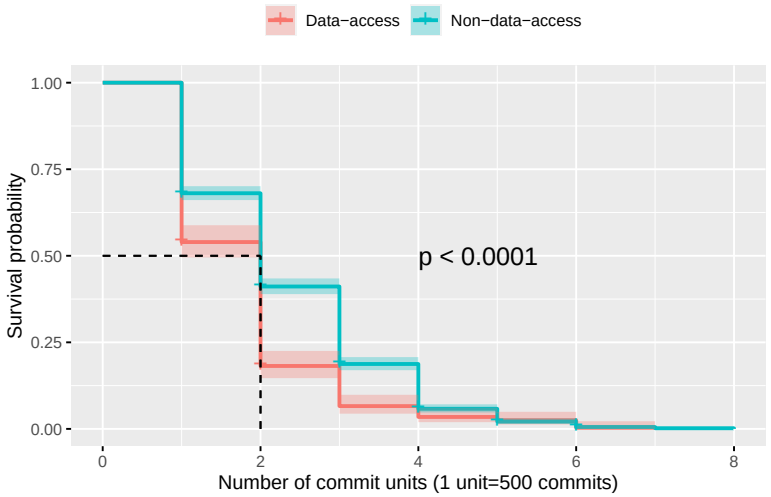


Fig. 10 Kaplan–Meier survival curve for NoSQL subject systems by grouping them into data-access and regular SATD comments. The x-axis represents the number of commits. The censoring time is marked on the plot.

Similarly, Fig. 10 shows for NoSQL subject systems that data-access SATDs tend to get fixed quicker than regular SATDs. The Log-Rank test's p-value was $< 2e-16$. Hence, we can reject the null hypothesis that there is no difference in survival curves.

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

5.3 RQ3: What is the composition of data-access SATD?

In this section, we describe the result of our manual classification of SATD comments in the data-access classes. Fig. 11 shows the taxonomy we extended from the work of Bavota et al. [7]. In particular, we added a new high-level category called *data-access debt* and provided more specialized categories for code debt, test debt and documentation debt. While our primary focus is on the newly added categories, especially on the data-access debt categories, we also provide a brief description of the original categories [7] for completeness.

5.3.1 Distribution of manually categorized data-access SATDs

We have manually classified 361 data-access SATD comments that represent our entire dataset with 95% confidence. We did not have enough information from the comments and the source code in some cases. We labeled such comments as *unclear*. Excluding 105 *false positives*, 4 multi-label and 12 unclear comments, we ended up with 240 data-access SATD comments. Table 5 shows the distribution of the final labels in the sample dataset. The comments under each category were presented separately for SQL and NoSQL subject systems. We mark SATDs related to database accesses with a database icon (🗄) and regular SATDs with a file icon (🗋). The categories are sorted according to the *total number of comments*.

Table 5 shows that a large portion of the comments belongs to sub-categories of *code debt*, *requirement debt* and *defect debt*. This is a similar observation with Bavota et al. [7]. We can also see that *data-access debts* are also found in smaller quantities compared to the traditional SATDs. The most considerable *data access debt* is *data access test debt*, followed by *query construction*.

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

Next, we describe the composition of SATDs categorized in Fig. 11 in the following paragraphs. We start with the SATDs identified by Bavota et al. [7], then we move to the newly added categories.

Table 5 Distribution of categories in the manually classified dataset

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

Code debt: *Code debt* includes "*problems found in the source code which can affect negatively the legibility of the code making it more difficult to be maintained*" [5]. It is divided into *low internal quality* and *workaround* categories. SATD comments that mention code quality issues related to program comprehension are categorized as *low internal quality*.

For example, a comment from the *low internal quality* category in *Blaze-Persistence*[12] says:

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

Comments justified by the developers as a workaround to address specific requirements are categorized under *workaround*. For example, quick fixes that mention a hack or workaround belong to this category. We extended *workaround* SATDs with a *workaround on hold* category. An "on-hold" SATD comment describes a problem that can be fixed once an issue referenced in the comment is addressed [27].

We found a specific case of an "on-hold" SATD when the issue holding back the developers was due to synchronization problems with the database schema. We dedicated the *workaround on hold due to database schema* category for similar SATDs. As an example, the comment in *OpenL Tablets*[13] says:

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

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

Defects are further divided into *known defects to fix* and *partially fixed defects*. An example of a *partially fixed defect* can be seen in *Snowstorm*:[14]

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

We found two specific cases when the issue was due to a *known defect of external library*; thus, we introduced a sub-category for these cases. *Low external quality* SATD comments describe problems with a high probability of becoming a bug or defect [7], as they may affect user experience.

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

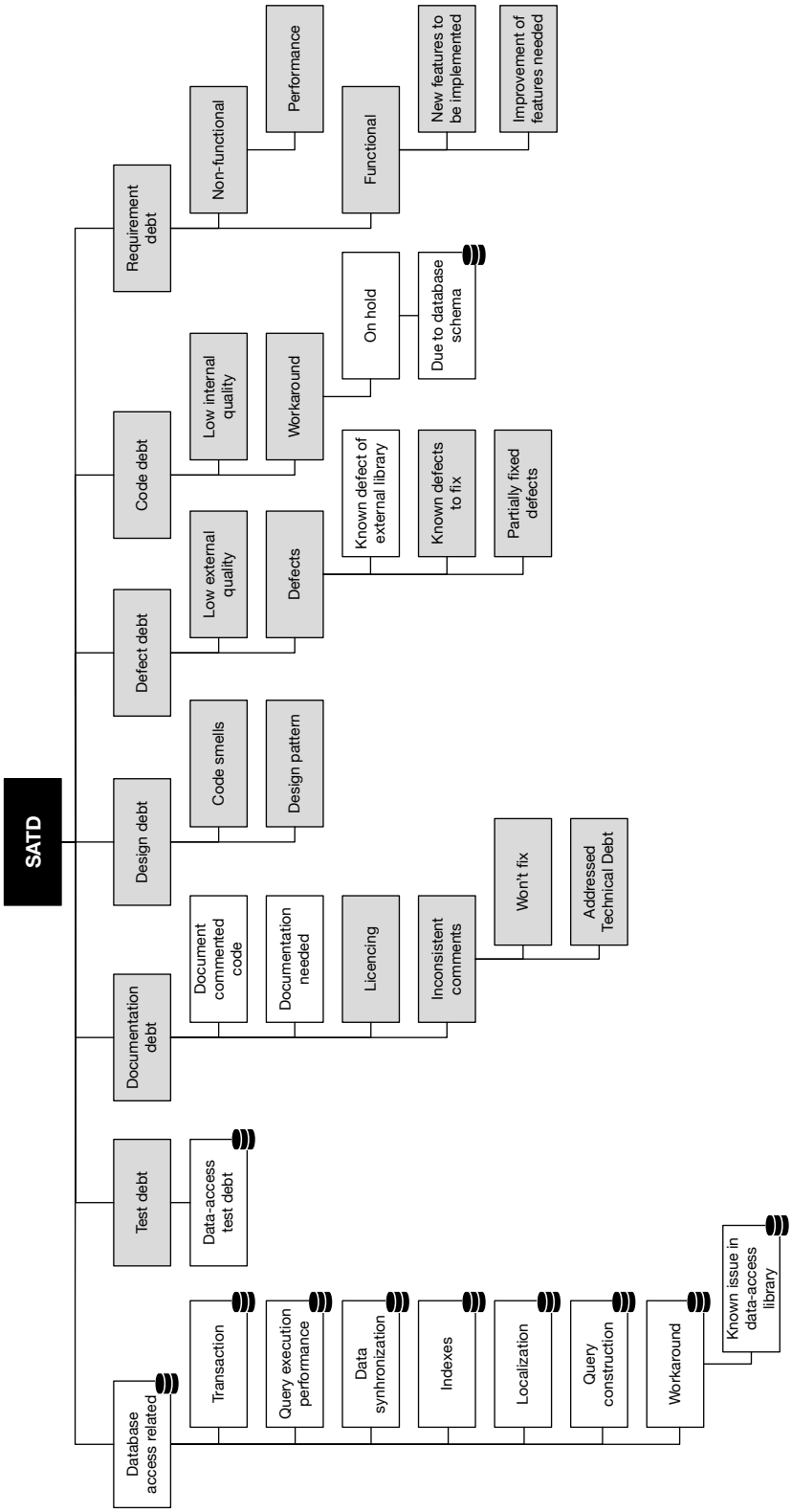


Fig. 11 SATD classification hierarchy extended from Bavota et al. [7]. White boxes are newly added categories to existing categories (gray boxes). Boxes marked with a database icon (🗄) are categories closely related to database accesses.

Design debt: SATDs related to *code smells* or *design patterns* are grouped in this category. Comments that discuss the violation of object-oriented design or mention refactoring as a solution are categorized under *code smells*. Comments suggesting the usage of a design pattern are classified under *design patterns*.

Documentation debt: This type of SATD can be identified in comments by looking for "*missing, inadequate, or incomplete documentation of any type*" [5]. Comments referring to issues already addressed are also categorized under documentation debt. This might happen when developers forget to update the documentation or comments after some source code changes. *Documentation debt* is divided into *inconsistent comments* and *licensing* categories. *Inconsistent comments* are further divided into *addressed technical debt* and *won't fix* categories [7].

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

Requirement debt: Comments that describe the need for new features to be implemented are categorized under *requirement debt*. Bavota et al. [7] further classified these to *functional* and *non-functional* requirement debt. *Functional* requirement debt includes the *new feature to be implemented* and *improvement to features needed* categories.

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

Test debt: *Test debt* affects the quality of testing activities [5]. These comments are typically found in testing classes and indicate low quality of testing code, *e.g.,* in terms of readability or the appropriateness of test cases and testing conditions.

We identified several *test debt* comments in the test code related to data accesses. We grouped these under the *data access test debt* category. Examples of these are related to the testing of database access operations such as transactions and query syntax. For example, a comment in *Sqlg*[15] says:

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

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

5.3.2 Database access related SATDs

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

Query execution performance: We found SATD comments dealing with issues about the execution performance of database queries. For example, a comment in *GnuCash Android*[16] says:

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

[15] Sqlg, `https://bit.ly/3wxbAqW`
[16] GnuCash Android, `https://bit.ly/37H1PeV`

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

Transactions: We identified comments about code that deal with transactions or rollback operations. An example of this type of debt was found in *Sqlg*:[17]

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

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

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

The following comment in *Foxtrot*[18] explains a workaround by directing the developer to an issue of Hazelcast, a key-value store implementation.

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

Data synchronization: These SATD comments describe a synchronization issue between the application and the database. An example comment can be found in *UPortal*:[19]

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

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

Indexes: Comments about issues related to indexes in the database are grouped under this category. For example, the following comment in *Sqlg*[20] describes the need for support for indexes.

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

Localization: We found comments about localization issues in the database, *i.e.,* problems with character sets or collation. The following comment in *Robolectric*[21] highlights the need for creating a collator as part of registering a localized collator.

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

Query construction: We found comments that mentioned issues about the construction of database queries. The following comment in *Carbon-apimgt*[22] notes a pending task to filter results by the status of the APIs.

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

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

[17] Sqlg, `https://bit.ly/3pRCOnK`
[18] Foxtrot, `https://bit.ly/3urWcey`
[19] UPortal, `https://bit.ly/3qY5OX2`
[20] Sqlg, `https://bit.ly/3aIqEcc`
[21] Robolectric, `https://bit.ly/3umvXpD`
[22] Carbon-apimgt, `https://bit.ly/2NvDZvQ`

Summary: We identified in data-access classes a large variety of SATD categories from the taxonomy of Bavota et al. [7]. Low internal quality code debt has the highest prevalence among data-access SATDs in both SQL and NoSQL subject systems. Most of the data-access SATDs have a higher prevalence in SQL subject systems compared to NoSQL subject systems. Besides the categories in the taxonomy of Bavota et al. [7], we found several SATDs pertaining to data access operations such as query construction, data synchronization, index management and transactions.

5.4 RQ4: What are the circumstances behind the introduction and removal of data-access SATD?

In this subsection, we present our result and analysis concerning the circumstances behind the introduction and removal of data-access SATDs. We first discuss when data-access SATDs are introduced and removed. Then we discuss the reasons motivating their introduction and removal.

5.4.1 When are data-access SATDs introduced?

To answer this question, we analyze the change history of the data-access SATDs labeled in RQ3 and identify the commits introducing the comments. The commit at which the comment first appears in change history is referred to as the *data-access SATD introducing commit*.

We measure the *introduction time* as the number of commits it takes for SATDs to manifest in the subject systems. As for the survival analysis, we use the number of commits to measure time as it is more reflective of software development activities. Similarly, we define *removal time* as the number of commits between the SATD comment's introduction and removal. Since each system's number of commits varies, we normalized the introduction time and removal time using Equation 2 (see Section 5.4).

Fig. 12 shows the overall distribution of data-access SATD introduction time. The distribution is right-skewed with the median introduction time (72.53%) and mean (64.14%). This indicates that most of the data-access SATD introducing commits did not happen at the beginning of the change history. This also confirms our observation of the survival analysis in RQ2. Data-access SATDs seem to be introduced at later stages in the change history. We also identified SATDs committed in the most recent snapshots of the subject systems (introduction time=100%).

Fig. 13 shows the distribution of introduction time for SQL and NoSQL systems. For both SQL and NoSQL data-access SATDs, introduction time is right-skewed. The notches of the SQL and NoSQL overlap, which means that the difference in the median is not significant. SQL data-access SATDs have a slightly higher median (80.31%) than in NoSQL systems (71.67%).

Table 6 shows the number of comments, mean, and median introduction time for all data-access SATD categories. The categories are ordered by the median introduction time from highest to lowest. *Low external quality* and *design patterns* data-access SATDs are introduced in the latest stages of change history among all the categories. On the other extreme, most of the database access related SATDs tend to be introduced at the early stages of change history. Compared to regular SATDs, most of the database access related SATDs are introduced earlier. *Transactions*, *indexes*, and *data-access test debt* tend to be introduced at later stages. *Addressed technical debt* comments tend to be introduced at the very beginning of the subject systems' development.

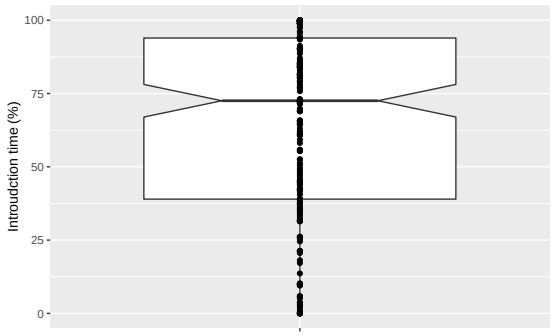


Fig. 12 Distribution of data-access SATD introduction time

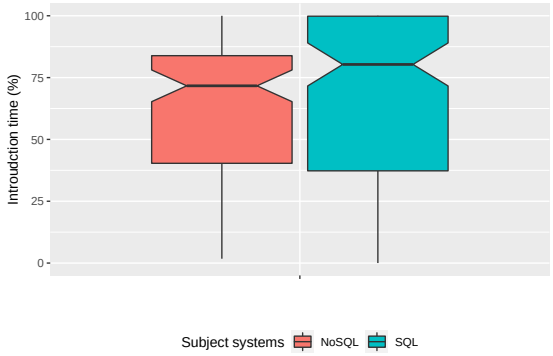


Fig. 13 Distribution of data-access SATD introduction time in SQL and NoSQL subject systems

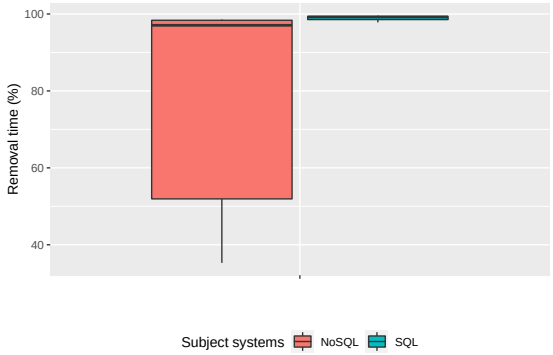


Fig. 14 Distribution of data-access SATD removal time in SQL and NoSQL subject systems

5.4.2 When are data-access SATDs removed?

We found 12 data-access SATDs that were removed at different stages of the change history. Fig. 14 shows the distribution of data-access SATD removal time for SQL and NoSQL subject systems. Both SQL and NoSQL SATDs were removed at the latter stages close to the most recent versions. The median removal time is 99.58% for SQL and 98.58% for NoSQL data-access SATDs.

Table 6 Data-access SATD introduction time for SATD categories

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

Table 7 Distribution of data-access SATD removal time among the data-access categories

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

Table 7 shows the distribution of data-access SATD removal time grouped by categories. We did not have any removed comments from the *database access related* SATD category. *Improvement of features needed* comments tend to be introduced at later stages of change history with the highest median removal time of 99.58%. On the other hand, *document commented code* comments were introduced in the middle stages of the change history (median=47.32%).

5.4.3 Why are data-access SATDs introduced and removed?

We now focus on the potential reasons for data-access SATDs' introduction and removal. We manually labeled the data-access SATDs' introducing/removing commit messages to classify their purposes. We classified the goal of the commit messages as *bug fixing*, *enhancement*, *new feature*, *refactoring*, and *merging*. Some commit messages described *multiple goals*, and some comments were labeled *unclear* as they did not contain enough information in the commit message for categorization.

Table 8 Data-access introducing commit goals in NoSQL and SQL subject systems

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

Table 8 summarizes the various goals of data-access SATDs' introductions. Considering NoSQL data-access SATDs, *refactoring* is the most associated reason with 38 instances (33.04%). It is followed by *new feature* with 30 cases (26.09%) and 22 *enhancements* (19.13%). For SQL, *bug fixing* was the most often mentioned reason in comments with 45 instances (34.88%). It is followed by *refactoring* with 38 cases (29.46%) and 30 *new features* (23.26%). Overall, *bug fixing* and *refactoring* are the main reasons behind the introduction of data-access SATDs.

Table 9 Data-access SATD introducing commit goals grouped by data-access SATD categories

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

Table 9 shows the introduction goals grouped by data-access debt categories.

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

We present the removal goals of SATD categories in Table 10. *Low internal quality* is associated with *enhancement* (60%) and *new feature* (40%). The remaining SATD categories have 6 instances combined.

Table 11 summarizes the goals of the removals of data-access SATDs. Several comments were removed for feature *enhancements* and *new features*. *Bug fixing* commits also contribute to the reduction of data-access SATD. Both SQL and NoSQL systems follow a similar distribution of commit goals.

Table 10 Data-access SATD removing commit goals grouped by data-access SATD categories

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

Table 11 Data-access SATD removing commit goals for SQL and NoSQL subject systems

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

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

6 Discussion

The goal of this study is to explore SATDs in data-intensive systems. In particular, we investigated the prevalence, persistence, composition of SATDs, and introduction and removal circumstances. The results show that SATDs are prevalent in data-intensive systems, and their prevalence increases as systems evolve. This pattern is similar to traditional software systems. Bavota and Russo [7] showed that SATDs are prevalent and increase as new ones are introduced during software evolution. This indicates that in both traditional and data-intensive systems, developers tend to introduce new SATDs instead of addressing existing ones. In addition, our results show that the prevalence of SATDs is different between SQL and NoSQL data-intensive systems. Given that NoSQL persistence systems are getting higher preference due to the advantages they offer in terms of schema flexibility and scalability and our result showing more prevalent SATDs in some NoSQL-based systems, our findings motivate further investigation of the impact of the persistence technologies on SATD.

Our results regarding the persistence of SATDs in data-intensive systems are similar to traditional systems. Bavota and Russo [7] found that the median survival time of SATDs to be

1000 commits for traditional software systems. We also find similar median survival times for both SQL and NoSQL subject systems. On the other hand, Maldonado et al. [28] reported that SATDs persist up to 173 days on average using five open-source traditional software systems. This implies that SATDs in data-intensive systems have even higher persistence (more than two years on average in our case). We also found that a significant number of SATDs persisted in all versions without getting addressed. Since the longer the SATD stays in the system, the higher the cost of repaying, practitioners should incorporate fixing technical debts as part of their workflow. This result highlights the importance of research work in SATDs in terms of providing tool support, raising awareness of the costs of technical debts, and providing processes and frameworks for monitoring technical debt.

The state-of-the-art SATD detection systems do not differentiate between different types of SATDs. One reason for this could be the lack of information on the specific types of SATDs. In this direction, Bavota and Russo [7] provided a taxonomy of SATDs, including design debt, code debt, defect debt, requirement debt, and test debt. While they addressed most of the software development workflow, they did not cover data-access debts since the subject systems were not data-intensive. We extended their taxonomy, incorporating 11 new database access-specific SATDs generalizing their taxonomy to data-intensive systems. This taxonomy can be utilized for proposing SATD detection approaches that provide more information than their mere existence. This, in turn, helps practitioners in their effort to manage technical debts and future researchers to investigate the impacts of specific types of SATDs on software quality. We find that low internal quality code debts were the most prevalent SATDs among our subject systems. Code debts are also found to be dominant SATDs in traditional software systems [7]. Hence, future research efforts should focus more on code debts as they are more prevalent SATDs in software systems. Data-access SATDs are also important in the context of data-intensive systems.

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

7 Threats to validity

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

Another threat to construct validity is the precision of the SATD detector tool. The 73.7% F_1 score shows that the tool could introduce a significant number of false positives. Indeed, we conducted a manual analysis and identified a considerable number of false positives. However, The SATD detector is a state-of-the-art tool whose base approach was also used in other studies (*e.g.,* [7]). Improving the accuracy of the SATD detector is out of the scope of the paper. However, the conclusions from this paper are carefully formulated and need to be interpreted taking into account the imprecise nature of the tool.

There might be cases when SATD comments are removed without code changes in effect. This may mean that the SATD admitted earlier is no longer viewed as technical debt by the developers, or they may not be interested in keeping track of that SATD [56]. Such cases are not actual removals of SATDs. Zampetti et al. [56] conducted an empirical study on Java open-source systems and observed that such cases are not frequent ($< 10\%$) in most cases and the maximum being 17%.

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

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

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

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

Threats to reliability validity: Reliability validity concerns factors that cause an error in data collection and analysis. To minimize potential threats to reliability, we analyzed open-source projects available on GitHub and provided a replication package that contains our dataset and analysis scripts [31].

8 Conclusion and future work

Technical debt represents the costs associated with favoring short-term low-quality solutions rather than appropriate solutions that take more time. Developers use SATD comments to track technical debt and reserve it for future fixes. We conducted a large-scale empirical study on data-access SATD using 102 open-source data-intensive systems. In particular, we extracted SATD comments from multiple snapshots of each subject system and explored the prevalence and persistence of data-access SATD. We conducted a manual analysis on representative data-access SATDs to categorize them under the type of SATD refereed. We further analyzed the data-access SATDs to understand the circumstances behind introducing/removing such debt.

Results show that data-access SATDs are introduced as software gets more mature, and many instances of SATDs persisted for a more extended time. Bug fixing and refactoring are the main

reasons behind the introduction of data-access SATDs followed by feature enhancements and new features. The observed SATD removal activities are not associated with refactoring, which implies that the removals are merely parts of bug fixing or feature enhancement activities.

SATDs in general and data-access SATDs, in particular, are critical to data-intensive systems as they determine the quality of the subject systems in terms of robustness and efficiency of data-access operations. Supporting more functionalities and maintaining code quality at the same time is a general problem to any software system. Having the right balance would help maintain software quality and reduce technical debt costs in the long run.

Our exploratory study could be extended in different ways. One extension could be to validate the newly identified database access-related SATDs and evaluate how developers prioritize such SATDs. Another extension of this study could be to investigate the impact of data-access SATDs on software quality. This could help demonstrate the harmfulness of technical debt to practitioners, which is particularly important in the context of data-intensive systems.

9 Declarations

9.1 Funding and Conflicts of interests

The authors did not receive support from any organization for the submitted work.The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Al-Barak, M., Bahsoon, R.: Database design debts through examining schema evolution. In: 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD), pp. 17–23 (2016). DOI 10.1109/MTD.2016.9
2. Albarak, M., Bahsoon, R.: Prioritizing technical debt in database normalization using portfolio theory and data quality metrics. In: Proceedings of the 2018 International Conference on Technical Debt, TechDebt '18, p. 31–40. Association for Computing Machinery (2018). DOI 10.1145/3194164.3194170
3. Alfayez, R., Alwehaibi, W., Winn, R., Venson, E., Boehm, B.: A systematic literature review of technical debt prioritization. In: Proceedings of the 3rd International Conference on Technical Debt, TechDebt '20, p. 1–10. Association for Computing Machinery (2020). DOI 10.1145/3387906.3388630
4. Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C.: Identification and management of technical debt. Inf. Softw. Technol. **70**(C), 100–121 (2016). DOI 10.1016/j.infsof.2015.10.008
5. Alves, N.S.R., Ribeiro, L.F., Caires, V., Mendes, T.S., Spínola, R.O.: Towards an ontology of terms on technical debt. In: 2014 Sixth International Workshop on Managing Technical Debt, pp. 1–7 (2014). DOI 10.1109/MTD.2014.9
6. Aniche, M., Bavota, G., Treude, C., Gerosa, M.A., van Deursen, A.: Code smells for model-view-controller architectures. Empirical Software Engineering **23**(4), 2121–2157 (2018)
7. Bavota, G., Russo, B.: A large-scale empirical study on self-admitted technical debt. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 315–326 (2016)
8. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. the Journal of machine Learning research **3**, 993–1022 (2003)
9. Chang, J., Gerrish, S., Wang, C., Boyd-graber, J., Blei, D.: Reading tea leaves: How humans interpret topic models. In: Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, A. Culotta (eds.) Advances in Neural Information Processing Systems, vol. 22. Curran Associates, Inc. (2009)
10. Cleve, A., Mens, T., Hainaut, J.: Data-intensive system evolution. Computer **43**(8), 110–112 (2010). DOI 10.1109/MC.2010.227
11. Cunningham, W.: The wycash portfolio management system. In: Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum), OOPSLA '92, p. 29–30. Association for Computing Machinery (1992). DOI 10.1145/157709.157715
12. Foidl, H., Felderer, M., Biffl, S.: Technical debt in data-intensive software systems. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 338–341 (2019). DOI 10.1109/SEAA.2019.00058

13. de Freitas Farias, M.A., de Mendonça Neto, M.G., da Silva, A.B., Spínola, R.O.: A contextualized vocabulary model for identifying technical debt on code comments. In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), pp. 25–32. IEEE (2015)
14. de Freitas Farias, M.A., Santos, J.A., Kalinowski, M., Mendonça, M., Spínola, R.O.: Investigating the identification of technical debt through code comment analysis. In: International Conference on Enterprise Information Systems, pp. 284–309. Springer (2016)
15. Gokhale, M., Cohen, J., Yoo, A., Miller, W.M., Jacob, A., Ulmer, C., Pearce, R.: Hardware technologies for high-performance data-intensive computing. Computer **41**(4), 60–68 (2008)
16. Huang, Q., Shihab, E., Xia, X., Lo, D., Li, S.: Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering **23**(1), 418–451 (2018)
17. Hummel, O., Eichelberger, H., Giloj, A., Werle, D., Schmid, K.: A collection of software engineering challenges for big data system development. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 362–369 (2018). DOI 10.1109/SEAA.2018.00066
18. Inc, G.: Search (2019). URL `https://developer.github.com/v3/search/`
19. Johannes, D., Khomh, F., Antoniol, G.: A large-scale empirical study of code smells in javascript projects. Software Quality Journal pp. 1–44 (2019)
20. Kamei, Y., Maldonado, E.d.S., Shihab, E., Ubayashi, N.: Using analytics to quantify interest of self-admitted technical debt. In: QuASoQ/TDA@ APSEC, pp. 68–71 (2016)
21. Kaplan, E.L., Meier, P.: Nonparametric estimation from incomplete observations. Journal of the American statistical association **53**(282), 457–481 (1958)
22. Kuutila, M., Mäntylä, M., Farooq, U., Claes, M.: Time pressure in software engineering: A systematic review. Information and Software Technology **121**, 106257 (2020). DOI https://doi.org/10.1016/j.infsof.2020.106257
23. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. J. Syst. Softw. **101**(C), 193–220 (2015). DOI 10.1016/j.jss.2014.12.027
24. Lim, E., Taksande, N., Seaman, C.: A balancing act: What software practitioners have to say about technical debt. IEEE Software **29**(6), 22–27 (2012). DOI 10.1109/MS.2012.130
25. Lin, D., Neamtiu, I.: Collateral evolution of applications and databases. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, pp. 31–40. ACM (2009). DOI 10.1145/1595808.1595817
26. Liu, Z., Huang, Q., Xia, X., Shihab, E., Lo, D., Li, S.: Satd detector: A text-mining-based self-admitted technical debt detection tool. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, pp. 9–12 (2018)
27. Maipradit, R., Treude, C., Hata, H., Matsumoto, K.: Wait for it: identifying "on-hold" self-admitted technical debt. Empirical Software Engineering **25**(5), 3770–3798 (2020)
28. Maldonado, E.d.S., Abdalkareem, R., Shihab, E., Serebrenik, A.: An empirical study on the removal of self-admitted technical debt. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 238–248. IEEE (2017)
29. Meurice, L., Nagy, C., Cleve, A.: Detecting and preventing program inconsistencies under database schema evolution. In: Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS 2016), pp. 262–273. IEEE (2016). DOI 10.1109/QRS.2016.38
30. Miller Jr, R.G.: Survival analysis, vol. 66. John Wiley & Sons (2011)
31. Muse, B.A., Nagy, C., Khomh, F., Cleve, A., Antoniol, G.: Replication package for: FIXME: Synchronize with Database. An Empirical Study of Data Access Self- Admitted Technical Debt (2022). DOI 10.5281/zenodo.5825671. URL `https://doi.org/10.5281/zenodo.5825671`
32. Muse, B.A., Rahman, M.M., Nagy, C., Cleve, A., Khomh, F., Antoniol, G.: 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 '20, p. 327–338. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3379597.3387467
33. Nagy, C., Cleve, A.: SQLInspect: A static analyzer to inspect database usage in Java applications. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, pp. 93–96. ACM (2018)
34. Papadimitriou, C.H., Raghavan, P., Tamaki, H., Vempala, S.: Latent semantic indexing: A probabilistic analysis. Journal of Computer and System Sciences **61**(2), 217–235 (2000)
35. Park, B., Rao, D.L., Gudivada, V.N.: Dangers of bias in data-intensive information systems. In: P. Deshpande, A. Abraham, B. Iyer, K. Ma (eds.) Next Generation Information Processing System, pp. 259–271. Springer Singapore, Singapore (2021)
36. Potdar, A., Shihab, E.: An exploratory study on self-admitted technical debt. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 91–100. IEEE (2014)
37. Ramasubbu, N., Kemerer, C.F.: Technical debt and the reliability of enterprise software systems: A competing risks analysis. Management Science **62**(5), 1487–1510 (2016). DOI 10.1287/mnsc.2015.2196
38. Rios, N., de Mendonça Neto, M.G., Spínola, R.O.: A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. Information and Software Technology **102**, 117 – 145 (2018). DOI https://doi.org/10.1016/j.infsof.2018.05.010

39. Röder, M., Both, A., Hinneburg, A.: Exploring the space of topic coherence measures. In: Proceedings of the eighth ACM international conference on Web search and data mining, pp. 399–408 (2015)
40. Sadalage, P.J., Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley (2014)
41. Scherzinger, S., Klettke, M.: Managing schema evolution in nosql data stores. In: Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013) (2013)
42. Scherzinger, S., Sidortschuck, S.: An empirical study on the design and evolution of NoSQL database schemas. In: G. Dobbie, U. Frank, G. Kappel, S.W. Liddle, H.C. Mayr (eds.) Conceptual Modeling, pp. 441–455. Springer International Publishing, Cham (2020)
43. Sierra, G., Shihab, E., Kamei, Y.: A survey of self-admitted technical debt. Journal of Systems and Software **152**, 70–82 (2019)
44. da Silva Maldonado, E., Shihab, E., Tsantalis, N.: Using natural language processing to automatically detect self-admitted technical debt. IEEE Transactions on Software Engineering **43**(11), 1044–1062 (2017)
45. Spadini, D., Aniche, M., Bacchelli, A.: Pydriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, p. 908–911. Association for Computing Machinery (2018). DOI 10.1145/3236024.3264598
46. Stonebraker, M., Deng, D., Brodie, M.L.: Application-database co-evolution: A new design and development paradigm. In: New England Database Day (2017)
47. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: When and why your code starts to smell bad. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 403–414. IEEE (2015)
48. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering **43**(11), 1063–1088 (2017)
49. Vassiliadis, P.: Profiles of schema evolution in free open source software projects. In: Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 1–12 (2021)
50. Weber, J.H., Cleve, A., Meurice, L., Ruiz, F.J.B.: Managing technical debt in database schemas of critical software. In: 2014 Sixth International Workshop on Managing Technical Debt, pp. 43–46 (2014). DOI 10.1109/MTD.2014.17
51. Wehaibi, S., Shihab, E., Guerrouj, L.: Examining the impact of self-admitted technical debt on software quality. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 179–188. IEEE (2016)
52. Xavier, L., Ferreira, F., Brito, R., Valente, M.T.: Beyond the code: Mining self-admitted technical debt in issue tracker systems. In: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, p. 137–146. Association for Computing Machinery (2020). DOI 10.1145/3379597.3387459
53. Yan, M., Xia, X., Shihab, E., Lo, D., Yin, J., Yang, X.: Automating change-level self-admitted technical debt determination. IEEE Transactions on Software Engineering **45**(12), 1211–1229 (2018)
54. Yu, Z., Fahid, F.M., Tu, H., Menzies, T.: Identifying self-admitted technical debts with jitterbug: A two-step approach. arXiv preprint arXiv:2002.11049 (2020)
55. Zampetti, F., Noiseux, C., Antoniol, G., Khomh, F., Di Penta, M.: Recommending when design technical debt should be self-admitted. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 216–226. IEEE (2017)
56. Zampetti, F., Serebrenik, A., Di Penta, M.: Was self-admitted technical debt removal a real removal? an in-depth perspective. In: Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, p. 526–536. Association for Computing Machinery (2018). DOI 10.1145/3196398.3196423
57. Zampetti, F., Serebrenik, A., Di Penta, M.: Automatically learning patterns for self-admitted technical debt removal. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 355–366. IEEE (2020)
58. Zhao, W., Chen, J.J., Perkins, R., Liu, Z., Ge, W., Ding, Y., Zou, W.: A heuristic approach to determine an appropriate number of topics in topic modeling. In: BMC bioinformatics, vol. 16, pp. 1–10. Springer (2015)