Challenges and Perils of Testing Database Manipulation Code

Maxime Gobert[1], Csaba Nagy[2], Henrique Rocha[3], Serge Demeyer[3,4], and Anthony Cleve[1]

[1] Namur Digital Institute, University of Namur, Belgium
[2] Software Institute, Università della Svizzera Italiana, Switzerland
[3] Dept. Computer Science, University of Antwerp, Belgium
[4] Flanders Make vzw, Belgium

Abstract. Software testing enable development teams to maintain the quality of a software system while it evolves. The database manipulation code requires special attention in this context. However, it is often neglected and suffers from software maintenance problems. In this paper, we investigate the current state-of-the-practice in testing database manipulation code. We first analyse the code of 72 projects mined from Libraries.io to get an impression of the test coverage for database code. We confirm that the database is poorly tested: 46% of the projects did not cover with tests half of their database access methods, and 33% of the projects did not cover the database code at all. To understand the difficulties in testing database code, we analysed 532 questions on StackExchange sites and deduced a taxonomy. We found that developers mostly look for insights on general best practices to test database access code. They also have more technical questions related to DB handling, mocking, parallelisation or framework/tool usage. This investigation lays the basis for future research on improving database code testing.
Keywords Testing, Database manipulation code, Empirical study.

1 Introduction

Database manipulation code is usually seen as an outsider in the codebase of an information system. It lies between the programs and the database, so it belongs partially to both, but not entirely to one. It can also involve multiple development teams. For example, in larger systems, a complex database requires a department of database administrators (DBAs) separated from the group of software engineers who maintain the application code. Both groups are in charge of maintaining their own side, but they need to share responsibilities as far as program-database communication is concerned.

However, shared responsibilities come at a price, and the *dual role* of database manipulation code leads to software maintenance problems. Stonebraker *et al.* argue that it is the most significant factor of database or application decay [26]. As they say, evolving requirements result in changes in the schema, which in turn require adjustments in the database manipulation code. Developers tend to minimise their effort to implement modifications, and the application or the database quality suffers the consequences.

M. La Rosa et al. (Eds.): CAiSE 2021, LNCS 12751, pp. 229–244, 2021.
The final publication is available at Springer via https://doi.org/10.1007/978-3-030-79382-1_14.

Several researchers have proposed approaches addressing the evolution of database-centric systems [7–9, 17, 18, 21, 31]. Other authors developed methods specifically designed for testing database applications, with a focus on test case generation [6], test data generation [5], test case prioritisation [11], and regression testing [24]. Despite the undeniable benefits of these methods, *no research work has investigated how developers test database access code in practice* – which could direct researchers where automated assistance is needed the most.

To provide such guidance, we investigated the current state-of-the-practice in testing database manipulation code addressing our research question: *What are the main challenges/problems when testing database manipulation code?*

As a motivational study, we analysed a set of open-source systems that rely on database access technologies and implement automated tests. We mined 6,626 projects from Libraries.io and found automated tests along with database manipulation code in only 332. Out of these, we examined 72 projects, for which we could collect coverage reports of test executions. We observed how tests cover the database access code in these projects. We found that overall *the database manipulation code is poorly tested.* 46% of the projects did not test half of their DB methods, and 33% of the projects did not even test DB communication.

To understand the reasons for such poor coverage, we qualitatively analysed 532 questions from popular StackExchanges websites and *identified the problems that hamper developers when writing tests.* We distilled the results in a comprehensive taxonomy of 83 issues grouped into 7 main categories. For each category, we discuss critical issues and solutions proposed by StackExchange users. Besides our contributions of (i) exploratory evidence of poorly tested database manipulation code and (ii) an extensive taxonomy of database testing problems, (iii) we infer actionable directions both for researchers and practitioners. Our study – being the first of its kind in this field – lays the foundation for future research on improving the maintenance of database code through automated testing.

2 Motivational Study

We first explore how developers test their database manipulation code in practice. Figure 1 depicts an overview of the three main steps we followed during this exploration: (1) we selected a set of open-source projects using databases, (2) we identified which part of their source code was involved in database communication and (3) we analysed how automated tests covered it.

During step ① *Project Selection*, we mined open-source systems from Libraries.io,[5] which we chose because (i) it monitors a broad set of projects (not just libraries), and (ii) it has a large database of dependencies among projects.[6]

We specifically looked for applications using databases and automated testing technologies. Libraries.io provides us with the possibility of searching for such projects through their dependencies. Selected projects had to satisfy four inclusion criteria: (i) *be written in Java*, since we rely on tools that support only Java (*i.e.,* to identify database code and measure test coverage); (ii) *use* JUnit[7]

[5] `https://libraries.io/data` [6] At the time of writing, it has 2.7M unique packages, 33M repositories, and 235M interdependencies between them. [7] `https://junit.org/`

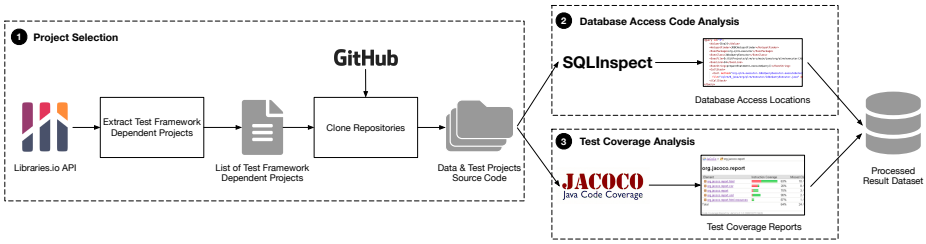

Fig. 1. Overview of the main steps for test coverage analysis of database access code

or TestNG,[8] i.e., the top Java testing frameworks according to the usage statistics of Maven central;[9] (iii) *use database access technologies*, *e.g.,* `java.sql` or `javax.persistence`; (iv) *have executable test suites*, as required by JaCoCo,[10] the test coverage tool we rely on.

We relied on version 1.4.0 of Libraries.io dataset published in December 2018, as it was the most recent at the time of conducting the survey. We cloned 6,626 systems satisfying a search query for Java projects with testing framework dependencies. Then we filtered them looking for imports of database communication libraries. The list of imports can be found in our replication package [1]. At this stage, we identified 905 projects.

In step ② *Database Access Code Analysis*, we identified the part of the source code involved in database communication. We used SQLInspect[11] for this purpose – a static code analyser for Java applications using JDBC, Hibernate, or JPA. This tool looks for locations in the source code where queries are sent to a database, extracts these queries, and analyses them for further inspection (*e.g.,* smell detection). In the remaining of the paper, we call *database access methods* all methods that construct or execute a DB query. We selected SQLInspect because (i) it supports popular database access technologies, (ii) it returns all the database access methods of the project under analysis, and (iii) it relies on a technique reaching a precision of 88% and a recall of 71.5% [18].

From the 905 projects selected at the first stage, SQLInspect identified database access methods in 332 of them. In the other projects, it did not detect database accesses. The reason for this is that SQLInspect looks for SQL, Hibernate, or JPA queries in the source code. An import does not necessarily imply query executions, and other DB communication means can be used (*e.g.,* an object-relational mapping; ORM) too, or the packages may not be used at all.

In step ③ *Test Coverage Analysis*, we looked at the way the DB access methods are *covered by tests*. We used the JaCoCo Maven plugin that can be integrated with a project's tests to collect coverage data at different granularity levels (*e.g.,* method or line). We implemented a script modifying the pom files of the 332 projects to execute tests with JaCoCo. Maven compilation or test execution failures prevented the generation of a test report file for 178 projects. Many projects (82) did not have a pom file or tests at all, despite having a

[8] `https://testng.org/` [9] `https://mvnrepository.com/open-source/testing-frameworks` [10] `https://www.jacoco.org` [11] `https://bitbucket.org/csnagy/sqlinspect`

dependency on a test framework. In the end, we collected test coverage data for 72 systems. Then we processed the reports along with the results of step ②.

Table 1. Overview of the projects

Metric	Min	Q1	Med	Q3	Max
Java LOC (effective)	225	1,476	3,198	12,929	133,331
GitHub Stars	0	0	2	10	9,152
Methods	11	110	278	1,057	15,188
DB Access Methods (in prod. code)	1	2	4	7	80

Table 1 summarises the main characteristics (with minimum, quartiles, median, and maximum values) of the analysed projects. The projects are of various sizes ranging from 225 LOC up to 133 kLOC. The biggest project is Speedment,[12] a Java Stream ORM. The most popular project is MyBatis[13] with 9,152 stars. Regarding database access code, we only considered methods in production code (*i.e.,* test classes excluded). We intentionally did not set a minimum threshold for the projects' size or database methods. Our goal was to see whether they test database access code or not in real-life projects. If the project had only one method communicating with the DB, we wanted to see the tests for it.

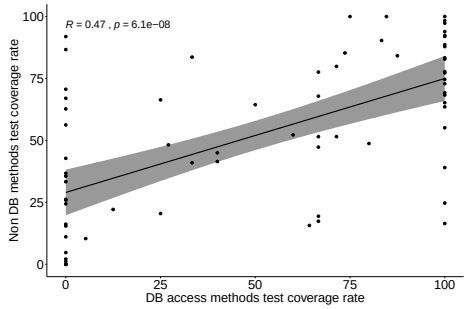

Fig. 2. Non-DB access methods vs DB access methods test coverage rate

Figure 2 shows a scatter plot of all projects and their respective test coverage rates. In total, 24 projects do not test database access communication at all. Also, a significant number of projects with the highest coverage rate have, in fact, full coverage. We found a mean value of 2.8 database methods for projects with full coverage. There are slightly fewer projects (48.6%) in the figure with lower coverage for database methods. However, considering only the projects above the median (*i.e.,* with at least five database methods), there is a bigger difference: 59% of them have a smaller coverage for database methods than for regular methods. Similarly, while 46% of the projects cover less than half of their database methods, this number increases to 53% for projects above the median. Moreover, 33% of the projects do not test the database code at all, and it increases again to 35% for projects with at least five database methods.

[12] `https://github.com/speedment/speedment` [13] `https://github.com/mybatis/mybatis-3`

We assessed the relationship between the test coverage rates of DB access methods vs regular methods using the Kendall correlation, as the Shapiro-Wilk normality test showed a significant deviation from the normal distribution. The result was a moderate positive correlation with a high statistical significance ($\tau = 0.47$, $p < 0.0001$).

In summary, we found a statistically significant correlation between the test coverage of regular and database access methods, but it is a weak-moderate correlation, and there can be important differences between the two. As our closer look at the sample set showed, the coverage of database code is poor in general together with regular methods. But when it comes to more complex database access code, it is even more neglected.

3 Challenges & Problems when Testing DB Access Code

The goal of our main research question is to understand the reasons holding back the developers to consider database access code in their test cases. We decided to study their most common problems on popular question-and-answer (Q&A) websites of the StackExchange network. The outcome of this qualitative study is a hierarchical taxonomy of common issues faced by developers.

3.1 Context and Data Collection

Identification and Extraction of Questions We targeted popular web sites of the StackExchange network for data collection: StackOverflow,[14] SoftwareEngineering[15] and CodeReview.[16] StackOverflow is the largest Q&A website in software engineering, making it a popular target of mining studies. At the time of our analysis, it included over 20M questions and 29M answers for software developers. Questions can be asked about specific programming problems, algorithms, tools used by programmers, and practical problems related to software development. Testing the database access code also falls into these categories. However, the guidelines of StackOverflow say that the best "*questions have a bit of source code in them.*" So more generic questions, not closely related to source code, are often discouraged as out-of-scope or opinion-based. General discussions are preferred on the SoftwareEngineering site of StackExchange. We included this site as we were interested in higher, conceptual-level problems as well; not only those related to the source code. Another valuable source for discussions in the StackExchange network is CodeReview. There, developers can ask for suggestions on a given piece of code. As they often include test code, we considered questions from CodeReview as well.

From these three Q&A websites, we selected our candidate questions according to the following criteria:

(a) *Scope.* We decided to select questions if (i) they explicitly mention testing in their title, and (ii) they use database access terms in their description (*e.g.,* DAO, SQL). For this filtering, we loaded the dumps of StackExchange sites into

[14] `http://stackoverflow.com` [15] `http://softwareengineering.stackexchange.com` [16] `http://codereview.stackexchange.com`

a database. We created full-text indices on both the titles and question bodies. Then we queried them, so the description had to match `(database | (data & access) | sql | dao | pdo) & test` and the title had to match `test`. The full-text search handled normalised text so stemmed words were also considered (*e.g.,* test-*ing*, database-*s*). Notice that StackOverflow has a tagging system for classification. However, the use of these tags is up to the user, who can easily omit them. Besides, the tagging system is different for the three sites considered, which led us to our alternative approach.

(b) *Impact and quality.* Due to the potentially large number of questions and our limited resources, we targeted posts with higher impact and better quality. For this reason, we relied on the scoring system of StackExchange. No up-votes or a negative score indicates problems, *e.g.,* an unclear, or out-of-scope question. Therefore we excluded posts with zero or negative ratings.

We used the StackOverflow dump published by StackExchange in December 2019, and the dumps of SoftwareEngineering and CodeReview published in March 2020. A total number of 1,837 questions matched the criteria: 41 on CodeReview, 174 on SoftwareEngineering and 1,622 on StackOverflow (see Table 2). We did a first manual screening of questions on the different sites. We observed that questions on CodeReview and SofwareEngineering were closer to our scope. Therefore, we decided to select more questions from these sites and to aim at a higher quality. To reach a 99% confidence level with a 5% margin of error, we set a threshold for a minimum score of 1 for CodeReview, 3 for SoftwareEngineering, and 13 for StackOverflow.

Table 2. Overview of the questions selected from StackExchange sites

Source	Candidate Questions	Selected Questions	False Positives
CodeReview	41	41	3
SoftwareEngineering	174	140	25
StackOverflow	1,622	351	86
Total	1,837	532	114

Manual Classification of Database Testing Issues After collecting the 532 questions, we manually inspected them. We followed an open coding process often applied to construct taxonomies or systematic mapping studies [20, 29]. In this approach, participants apply labels to concepts found in the text of artefacts. Then the tags are organised into an overall structure. During the process, labels and categories might be merged and renamed [20]. We performed the classification process in three main rounds. First, we did a trial round with a random set of 100 questions, wherein two of the authors assigned labels to the artefacts. The goal was to see whether we need to apply changes to our selection criteria and to test the classification platform that we implemented for this purpose. After the first round, we implemented a few adjustments to our platform. For the second round, we labeled the remaining questions with the help of two more authors. In the last round, we resolved conflicts where needed.

Each artefact was labeled by two of the four participants, randomly assigned to them. The platform showed the question and its relevant metadata (score, timestamp, tags) along with a link to the original discussion thread for further inspection. We followed a multi-label approach. Each participant could assign multiple labels to the artefact from the list of existing labels in the database. If needed, they could create new tags too. In principle, existing labels should not be shown to participants. But as we expected a high number of tags, showing the existing ones could help us using consistent naming without introducing substantial bias. Indeed, the participants were not aware of the assignments.

After the second round, all 532 questions were labeled by two participants. At this point, one author reviewed all the tags and proposed the merging of those with identical meaning. This merging was discussed among authors and applied to the database. We finally agreed and used identical tags for 147 questions; partially agreed for 77 posts (only a subset of identical labels), and used entirely different tags for 308 questions. The high number of unique tags explains this relatively high number of conflicts (72.37%). Indeed, at this point, the database had 290 different labels. Thus, participants took advantage of the multi-label classification and captured various aspects of questions.

To resolve conflicts, a third tagger was assigned to review each conflicting artefact. This third person was a randomly selected author who took part in the classification but did not label the same question beforehand. The system showed the labels of the previous taggers, and the reviewer could accept or discard them. Minor modifications were also allowed, if necessary.

At the end of this process, one author carefully reviewed all the tags and organised them into categories. This categorisation was then discussed among the authors in multiple rounds. As an outcome, a taxonomy was constructed with 83 database testing issues in 7 main categories. In the rest of this section, we present this taxonomy together with qualitative examples.

3.2 Taxonomy of Database Testing Issues

Usman *et al.* reviewed taxonomies in software engineering and found the hierarchical form the most frequently used classification structure [29]. We adopted this representation as an efficient approach to organise our findings. In this form, there is a parent-child (*is-a*) relationship between categories, and one category has additional subcategories. Categories correspond to issues or problems raised in the question, and subcategories represent subtypes of a problem. Consider, *e.g., Mocking Persistence Layer* as a specialised type of *Mocking*-related issues.

Figure 3 shows the final structure of the taxonomy. There are a total number of 83 leaf issues organised in 7 main (root) categories. For each root category, we show the total number of questions labelled with such problems. The distribution of the corresponding questions over the three sites is also provided. For example, the *Mocking* category had 54 questions including 8 from CodeReview, 17 from StackExchange, and 29 from StackOverflow. Recall that we had a multi-label approach, so one question could represent mixed problems. Thus, a question can belong to more categories in the hierarchical taxonomy.

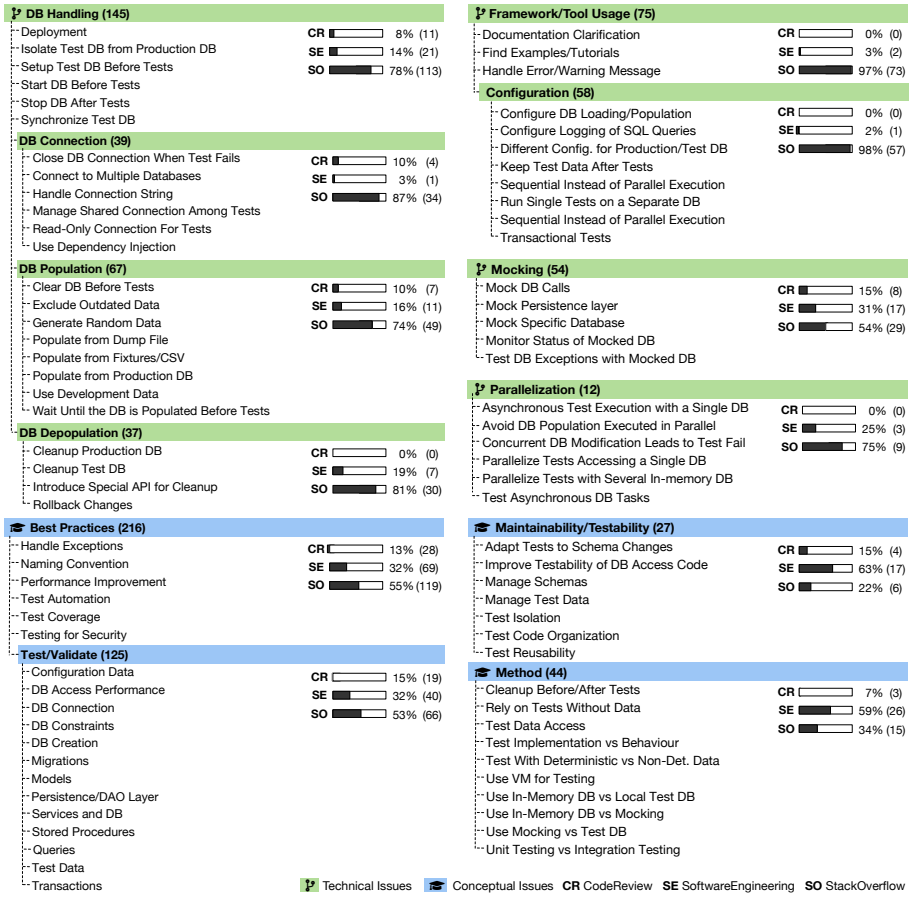

Fig. 3. Taxonomy of issues faced by developers when testing database access code

We observe intriguing technical and conceptual difficulties, and we differentiate between them in Figure 3. We mark the technical problems with and the conceptual ones with . It is interesting to observe the origin of questions for those abstraction levels. Higher-level, conceptual problems mainly originate from SoftwareEngineering, especially for *Maintainability/Testability* or *Method*. Technical problems are closer to the source code and mostly originate from *StackOverflow*, especially for the *Framework/Tool Usage* category. Questions from CodeReview cover both abstraction levels, but most of them relate to the general *Best Practices* category. None of them deals with *Framework/Tool Usage*. Below, we describe and illustrate each main problem category. We cite posts on StackExchange sites with *SE* notation. Due to space limitation, these references can be found in our replication package [1].

DB Handling The most prevalent technical issues are related to the management of the database: we found 145 questions in this category. Indeed, many have problems initialising the database before executing the tests. This includes starting the database, configuring it, and populating it with test data. The test database population was often mentioned as a root cause of performance issues.

These initialisation steps are critical as they have to be performed before test executions. As a developer complained: "*This whole thing takes quite some time (...). Having this run as part of our CI (...) is not a problem, but running locally takes a long time and really prohibits running them before committing code*" *SE1*. Solutions often include performance tweaks, *e.g.,* having an in-memory DB for local tests or ensuring that time-critical parts of the initialisation are executed only when needed. Instead of re-initialising the database before running the tests, an alternative is to clean it up *after* running the tests. This ensures that a failing test does not leave the environment in an inconsistent state *SE2*. Many recommend transactional tests (*i.e.,* wrap tests in transactions) for this purpose, so that a rollback can recover the initial state of the database. An advantage is that it can be significantly cheaper than recreating it. This, however, is not possible for testing DB access that already relies on transactions. As a desired feature, this is also supported by many testing frameworks. Some questions came from situations when the design does not support data deletion *SE3*. Others faced issues keeping a test DB in synch with a production or development DB *SE4*; while many had problems handling the connection to a test DB *SE5*, *SE6*, *SE7*.

Framework/Tool Usage A large number of problems (75) concern the use of a concrete tool or framework. Most of them relate to configuring a framework for a dedicated database in a test/development or production environment *SE8*, *SE9*. These questions have high scores suggesting that many developers suffer from such issues. For example, a question to configure Django *SE9* was voted up 59 times and stared by 16 users. Similarly, developers ask help for different DB initialisation (*e.g.,* running scripts, using dumps or fixtures), or cleanup configurations *SE10*. Interestingly, in some cases, they want to keep the test database after running their tests for debugging purposes *SE11*. Many also ask for guidance to solve a particular error message in the testing framework, *e.g.,* misusing transactional tests *SE12*, or to configure in-memory databases *SE13*.

Mocking Mocks can help by isolating the tests (*i.e.,* cutting off dependencies), and by avoiding the performance drawbacks of databases (*e.g.,* avoiding IO). Many questions indicate that developers need help in mocking the persistence layer. As a first step, an important design decision they have to make is the level at which they implement the mocks. For example, a developer reasoned in a question as follows: "*I could either mock this object at a high level (...), so that there are no calls to the SQL at all (...) Or I could do it at a very low level, by creating a MockSQLQueryFactory that instead of actually querying the database just provides mock data back*" *SE14*. Recommendations depend on the objectives, as an answer says: "*Higher level approaches are more appropriate for unit testing. Lower-level approaches are more appropriate for integration testing.*" Broader questions were also asked about the benefits of mocking *SE15*, or guidelines to mock the data access layer *SE16*, *SE17*. Technical questions tackled, for example, emulating exceptions in a mocked database *SE18*. When mocking is unfeasible, it can indicate poor software design *SE19*. Stored procedures *SE20* and views *SE21* also made mocking impossible in other systems.

Parallelization We observed some (12) technical problems related to parallel test executions. These were closely related, so we grouped them in this category. One of the highest-rated questions was about turning off the parallel execution of tests in *sbt* (a build tool for Scala and Java) *SE22*. The developer complained that a project "*mutates state in a test database concurrently, leading to the test to fail.*" Likewise, asynchronous or lazy calculations led to challenging bug hunts *SE23*. They also asked for advice to make test execution parallel, *e.g.,* to handle a dedicated in-memory database per thread *SE24*.

Best Practices The most frequently used labels were about testing best practices for DB applications. Developers either look for general advice or explicitly want to know about best practices. The highest-rated question has 331 up-votes entitled "*What's the best strategy for unit-testing database-driven applications?*" *SE25*. It generates discussion on mocking vs testing against an actual database. In the answers, mocking is mostly recommended for unit testing, while a copy of the database is favoured for more complex databases. In other cases, a combined approach might be needed: "*Ideally I want to test the data access layer using mocking without the need to connect to a database and then unit test the store procedure in a separate set of tests*" *SE16*. Best practices are also sought for performance improvements *SE26*, *SE1*, *SE27*. In particular, where mocking is not an option, solutions mostly advise the use of in-memory databases to reduce IO operations. Other topics include testing for security vulnerabilities, *e.g.,* looking for static analysers to spot SQL injection attacks *SE28*. Likewise, some questions look for tools to measure test coverage. They want to know, for example, the coverage of executed queries in test cases *SE29*. A majority of these questions were grouped under *Test/Validate*. These are looking for advice on testing or validating a specific code or DB entity. For example, SQL queries embedded in code *SE30*, database migration *SE31* or transactions *SE32*.

Maintainability/Testability Several questions tried to address maintainability problems or the testability of the database access code. In a question, a developer struggled with a system that validated RESTful APIs with SQL queries in its integration tests *SE33*. As he summed up his root problem: "*a small change in the DB structure often results in several man days wasted on updating the SQL and the SQL building logic in the integration tests.*" The developer wanted to wipe out the SQL code from the tests entirely. In the answer, they discouraged him from doing so. They acknowledged that relying on the queries can be a good practice to verify the database state. Instead, it was recommended to improve the maintainability of the tests: (i) by reducing the coupling inside the codebase (one table per module), and (ii) by splitting the tests into smaller pieces. In another question, a developer wanted to reduce the maintenance effort by omitting the tests of the ORM layer. He was, however, afraid of giving up on aiming for a 100% coverage. As he wrote it, "*Our test databases are a bit messy and are never reseted, hence it's impossible to validate any data (and that is out of my control).*" In the answers, they supported him that it is important to balance coverage and prioritise efforts then suggested generating the tests for the ORM layer. Others pointed out that preparing the environment of testing the

database access code is also troublesome. For example, a developer complained: "*The problem I ran into was that I spent a lot of time maintaining the code to set up the test environment more than the tests*" *SE34*. Many questions were also related to the management of changing schema or test data. As a general guideline, a recommendation said: "*I would apply a single rule: keep your test data close to your test. Test is all about maintenance: they should be designed with maintenance in mind, hence, keep it simple*" *SE35*.

🎓 **Method** Many developers were concerned about the problems of their testing method. The most frequent arguments were whether DB-dependent code should be tested via unit or integration tests *SE36*, *SE37*, *SE38*, *SE39*, *SE25*. A regular claim was that "*unit tests should not deal with the database, integration tests deal with the database*" *SE37*. Recommendations target to maximise the isolation of unit tests and decouple the database, *e.g.,* through mocking. In contrast, integration tests aim to test more complex structures by relying on the database. Interesting questions related to populating a database before tests, *e.g.,* whether data should be dynamically generated or pre-populated beforehand *SE40*. A re-occurring discussion was on the use of an in-memory database versus a mocking strategy *SE41*. When performance or decoupling the tests from the database was more critical, the choice was to mock. Otherwise, we could see cases where mocking was not possible (*e.g.,* because of stored procedures or views). The in-memory database was then considered as a good compromise to test the database access. It indeed solves the portability issues of testing against an actual DB and improves the performance. Compared to mocking, the testing can be more extensive, *e.g.,* it enables the tests to validate embedded SQL queries. In some cases, however, the in-memory database differs significantly from the production database. This can be a problem as some DB-specific features cannot be tested, *e.g.,* a special SQL syntax *SE42*.

3.3 Discussion and Implications

Below, we discuss the main observations we made in our investigation, together with actionable directions for researchers and practitioners.

Maintainability of DB Tests A frequent issue was to keep tests in sync with database schema changes, as developers hardly get any support for this task. Many also struggled with isolating tests. Our study is exploratory by nature, and more studies are needed to understand the factors affecting the maintainability of database-related test code. Understanding more from the practices of the developers, and good, maintainable database test code [2, 23] is a promising direction. Alternatively, automated approaches could help in regular tasks of developers. Some approaches aim to identify the system fragments impacted by schema changes [16,17]. Such methods could be extended to the testing context, *e.g.,* to maintain a mapping between schema elements and mocks.

In-Memory DB vs Actual DB vs Mocking We have seen many points in favour and against whether tests should rely on mocking, in-memory databases, or the actual database. In the systems analyzed in our motivational study, we

found that 19 out of the 72 projects (26%) used mocks: 17 had Mockito,[17] and 2 had EasyMock tests.[18] This low number surprised us, as mocking was *the* recommended approach for unit tests to decouple them from the DB. This is in line with the findings of Trautsch and Grabowski [27] who observed only a small amount of unit tests in open source Python projects, especially with mocks. A potential explanation is that it is easier to set up an in-memory database and rely on integration tests; instead of bothering with the implementation of mocks, despite its advantages. In any respect, developers need help in the implementation of database-related tests. They would benefit from automated support in this context. Some authors already explored the generation of tests with mocks [3,19]. The emergence and initial success of such tools (*e.g.,* EasyMock,[18] MockNeat[19]) is encouraging to develop similar approaches.

DB Support in Testing Frameworks In our motivational study, we excluded projects with failing tests. Many failures were due to misconfigured testing environments. The systems either (i) relied on an external database for their tests, or (ii) used in-memory databases, but did not set them up correctly. We observed related problems in our qualitative study: many developers struggle to configure frameworks with multiple database connections. Testing frameworks could better support developers in this task with DB-dedicated features, especially if these are configurable from the build systems. Some frameworks already provide similar functionalities. For example, *Spring Test* has *JdbcTestUtils*,[20] a collection of JDBC-related functions. It also provides support for test fixtures and transactional tests. Another framework, Rails, offers similar features. We observed that the most desired features are related to the initial configuration of databases, and the efficient recovery of the database state between successive tests. Developers' needs remain unexplored in this field, and further research is necessary to improve testing practices as far as DB access is concerned.

4 Threats to Validity

Construct validity In our motivational study, we rely on SQLInspect to identify the database access methods of projects, *i.e.,* methods involved in querying the database. As a static tool, it may miss some DB methods, particularly in case of highly dynamic query construction. For test coverage, we rely on JaCoCo, a state-of-the-art tool used in industry and academia [14]. It might miss execution paths, and its configuration can influence the coverage results (*e.g.,* due to missing classes from the classpath). To avoid this, we executed tests according to Maven standards and excluded projects with failing tests.

Internal validity In our qualitative analysis, the manual classification of StackExchange questions is exposed to subjectiveness. To mitigate this risk, two authors examined each post independently, and a third author resolved conflicts.

External validity Our motivational study is exploratory by nature. It considers various types of projects in terms of application domain, size, and intensity of DB interactions. They are, however, all from Libraris.io and limited to the Java

[17] https://site.mockito.org/ [18] https://easymock.org/
[19] https://github.com/nomemory/mockneat [20] https://docs.spring.io/spring/docs/current/spring-frameworkreference/testing.html

programming language. Projects not considered in our study might lead to other results. In our qualitative study, we extracted questions from three different StackExchange sites, intending to reach a higher level of diversity. We selected higher-ranked questions which are likely to influence more developers. This might introduce a bias towards the posts we selected. In reality, developers might face even more diverse challenges when (not) testing database code.

5 Related Work

Our research work got motivated and inspired by more general studies analysing testing practices and related maintainability issues. Beller *et al.* [4] conducted a large-scale field study on testing practices, monitoring 5 months of activities from 416 software engineers. They observed, among others, that (i) developers rarely run tests in the IDE; (ii) Test-Driven Development is not widely spread among the participants; and (iii) developers usually spend 25% of their time on testing. Gonzalez *et al.* [12] analysed over 80K open-source projects and found that (i) only 17% of those projects included test cases, and (ii) 76% of them did not implement testing patterns that would ease maintainability.

Several researchers have proposed approaches for testing database applications. Deng *et al.* [10] propose a white-box testing approach for web applications. They extract URLs from the application source code to create a path graph, from which they then generate test cases. Ran *et al.* [22] propose a similar framework, but for black-box testing web applications. They use a directed graph of the webpage transitions and database interactions as input for generating test sequences, and for capturing how the database updates along with the test cases. Kapfhammer and Soffa [15] present a test coverage technique that monitors database interactions. They employ instrumentation of the application and test cases to capture when SQL is used. Tuya *et al.* [28] define a criterion to measure SQL query coverage. They argue that SQL queries embedded in code are not taken into account for test design.

In this paper, we collect and classify questions in StackExchange sites, through a multi-tagging approach, itself inspired by previous work in our field. Gonzalez *et al.* [13] propose a 5-way classifier approach that assigns multiple tags to StackOverflow questions. They use a dataset composed of a training set of over 3 million questions and a test set of 20 thousand questions. Vasilescu *et al.* [30] investigate the relationships on StackOverflow questions/answers and GitHub commits. They find a positive correlation indicating that the activity of developers on StackOverflow affects their commit activity on GitHub.

Our qualitative analysis revealed that many StackExchange questions were related to Mocking, a testing technique to simulate dependencies, often used to isolate the component under test. Spadini *et al.* [25] empirically analyse the usage of mocking dependencies on testing. They analyse 4 projects with a total of 2,178 test dependencies and they survey 105 developers on their findings. The results indicate that mocking is often used on dependencies that would have made testing difficult to depend on external resources. Other popular topics we found in StackExchange questions were related to best practices of testing database

code, specially understandability. Alsharif *et al.* [2] study the understandability of auto-generated database tests. They argue that studies focusing on creating database tests do not take into account the human cost to understand such tests.

In summary, the analysis of related research shows that DB access code is sufficiently different from normal code to warrant specialised approaches. Several proposals were made to support database access code testing. Nevertheless, no research work has investigated *how* developers test DB access code *in practice*, nor the main *issues* they face in this context.

6 Conclusion

We present a study of the challenges faced by developers when testing database access code in practice. As a motivational study, we first studied the extent to which database code is covered by tests by analysing 72 open-source Java projects. We found that 46% of those projects did not test half of their database methods and 33% of them did not test the database communication at all.

We then conducted a qualitative study to understand the poor test coverage of database access code. We analysed 532 StackExchange questions related to database code testing and identified a total of 83 issues, classified in a taxonomy of 7 main categories. We found that developers mostly look for insights on general best practices to test DB access code. Concerning technical issues, they ask mostly about DB handling, mocking, parallelisation, or framework/tool usage.

We address an unexplored field of understanding testing practices of database communication and to identify the main difficulties that hamper developers. Our findings can serve as a starting point to direct researchers where practitioners need assistance. They open the door to complementary studies focused on particular categories of issues as well as their link with actual bugs. Further investigation is needed, however, such as the validation of the taxonomy with testing practitioners, or the analysis of the *answers* given to forum questions about database code testing. Immediate feedback of practitioners and answers may contain solutions to the issues we identified in this paper, that could guide researchers towards dedicated techniques and tools to assist developers when testing DB access code.

✎ **Replication package.** We made all data, scripts, and detailed results of our study publicly available in a replication package [1].

Acknowledgements. This work is supported by (a) the F.R.S.-FNRS and FWO-Vlaanderen via the EOS project 30446992 SECO-ASSIST and (b) Flanders Make vzw.

References

1. Repl. pkg., `https://github.com/csnagy/caise2021-db-manipulation-testing`
2. Alsharif, A., et al.: What factors make SQL test cases understandable for testers? a human study of automated test data generation techniques. In: ICSME (2019)
3. Arcuri, A., Fraser, G., Just, R.: Private api access and functional mocking in automated unit test generation. In: Proc. ICST (2017)

4. Beller, M., Gousios, G., Panichella, A., Zaidman, A.: When, how, and why developers (do not) test in their ides. In: Proc. ESEC/FSE (2015)
5. Castelein, J., Aniche, M., Soltani, M., Panichella, A., van Deursen, A.: Search-based test data generation for SQL queries. In: Proc. ICSE (2018)
6. Chays, D., Dan, S., Frankl, P.G., Vokolos, F.I., Weber, E.J.: A framework for testing database applications. In: Proc. ISSTA (2000)
7. Chen, T.H., Shang, W., Hassan, A.E., Nasser, M., Flora, P.: Detecting problems in the database access code of large scale systems. In: Proc. ICSE (2016)
8. Cleve, A., Brogneaux, A., Hainaut, J.: A conceptual approach to database applications evolution. In: Proc. ER (2010)
9. Delplanque, J., Etien, A., Anquetil, N., Ducasse, S.: Recommendations for evolving relational databases. In: Proc. CAiSE (2020)
10. Deng, Y., Frankl, P., Wang, J.: Testing web database applications. SIGSOFT Softw. Eng. Notes **29**(5) (2004)
11. Garg, D., Datta, A.: Test case prioritization due to database changes in web applications. In: Proc. ICST (2012)
12. Gonzalez, D., Santos, J.C.S., Popovich, A., Mirakhorli, M., Nagappan, M.: A large-scale study on the usage of testing patterns that address maintainability attributes: Patterns for ease of modification, diagnoses, and comprehension. In: MSR (2017)
13. González, J.R.C., Romero, J.J.F., Guerrero, M.G., Calderón, F.: Multi-class multi-tag classifier system for stackoverflow questions. In: Proc. ROPEC (2015)
14. Ivanković, M., Petrović, G., Just, R., Fraser, G.: Code coverage at Google. In: Proc. ESEC/FSE (2019)
15. Kapfhammer, G.M., Soffa, M.L.: Database-aware test coverage monitoring. In: 1st India Softw. Eng. Conference (2008)
16. Maule, A., Emmerich, W., Rosenblum, D.: Impact analysis of database schema changes. In: Proc. ICSE'08 (2008)
17. Meurice, L., Nagy, C., Cleve, A.: Detecting and preventing program inconsistencies under database schema evolution. In: Proc. QRS (2016)
18. Meurice, L., Nagy, C., Cleve, A.: Static analysis of dynamic database usage in java systems. In: Proc. CAiSE (2016)
19. Pasternak, B., Tyszberowicz, S., Yehudai, A.: Genutest: A unit test and mock aspect generation tool. In: Hardware and Software: Verification and Testing (2008)
20. Petersen, K., Vakkalanka, S., Kuzniarz, L.: Guidelines for conducting systematic mapping studies in software engineering: An update. IST **64** (2015)
21. Qiu, D., Li, B., Su, Z.: An empirical analysis of the co-evolution of schema and code in database applications. In: Proc. ESEC/FSE (2013)
22. Ran, L., et al.: Building test cases and oracles to automate the testing of web database applications. Information and Software Technology **51**(2) (2009)
23. Riaz, M., Mendes, E., Tempero, E.: Towards maintainability prediction for relational database-driven software applications: Evidence from software practitioners. In: Proc. Advances in Software Engineering (2010)
24. Rosero, R.H., Gómez, O.S., Rafael, G.D.R.: Regression testing of database applications under an incremental software development setting. IEEE Access **5** (2017)
25. Spadini, D., Aniche, M., Bruntink, M., Bacchelli, A.: Mock objects for testing java systems. Empir. Softw. Engineering **24**, 1461–1498 (2019)
26. Stonebraker, M., Deng, D., Brodie, M.L.: Application-database co-evolution: A new design and development paradigm. In: New England Database Day (2017)
27. Trautsch, F., Grabowski, J.: Are there any unit tests? an empirical study on unit testing in open source python projects. In: Proc. ICST (2017)

28. Tuya, J., Suárez-Cabal, M.J., de la Riva, C.: Full predicate coverage for testing SQL database queries. Softw. Testing, Verification and Reliability **20** (2010)
29. Usman, M., Britto, R., Börstler, J., Mendes, E.: Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. Information and Softw. Technology **85** (2017)
30. Vasilescu, B., Filkov, V., Serebrenik, A.: Stackoverflow and github: Associations between software development and crowdsourced knowledge. In: Proc. ICSC (2013)
31. Vassiliadis, P., Zarras, A.V.: Survival in schema evolution: Putting the lives of survivor and dead tables in counterpoint. In: Proc. CAiSE (2017)