Information Systems 111 (2023) 102105

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Best practices of testing database manipulation code

Maxime Gobert [a], Csaba Nagy [b,*], Henrique Rocha [c], Serge Demeyer [d,e], Anthony Cleve [a]

[a] Namur Digital Institute, University of Namur, Namur, Belgium
[b] Software Institute, Università della Svizzera italiana, Lugano, Switzerland
[c] Loyola University Maryland, Baltimore, MD, USA
[d] Department of Computer Science, University of Antwerp, Antwerp, Belgium
[e] Flanders Make vzw, Belgium

ARTICLE INFO

Article history:
Received 7 March 2022
Accepted 20 July 2022
Available online 30 July 2022
Recommended by M. Weidlich

Keywords:
Testing
Database manipulation code
Empirical study

ABSTRACT

Software testing enables 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 study the current state-of-the-practice in testing database manipulation code. We first analysed the tests of 72 open-source projects to gain insight into the coverage of database access code. The database was poorly tested: 46% of the projects did not cover with tests half of their database access methods, and 33% did not cover the database code at all. This poor coverage motivated us to study developers' challenges and best practices. (i) First, we analysed 532 questions on Stack Exchange sites and deduced a *taxonomy of issues*. Developers mostly looked for general best practices to test database access code. Their technical questions were related to database management, mocking, parallelisation, or framework/tool usage. (ii) Next, we examined the answers to these questions. We manually labelled 598 answers to 255 questions. We distinguished 363 solutions and organised them in a *taxonomy of best practices*. Most of the suggestions considered the testing environment and recommended various tools or configurations. The second largest category was database management, where many addressed database initialisation and clean-up between tests. Other categories pertained to code structure or design, concepts, performance, processes, test characteristics, test code, and mocking. We illustrate the two taxonomies through intriguing examples.

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 software engineers who maintain the application code. Both teams are in charge of maintaining their own side, but they need to share responsibilities as far as program-database communication is concerned.

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 [1]. 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.

Several researchers have proposed approaches addressing the evolution of database-centred systems [2–8]. Other authors have developed methods specifically designed for testing database applications, with a focus on test case generation [9], test data generation [10], test case prioritisation [11], and regression testing [12].

Despite the undeniable benefits of these methods, no studies have investigated how developers test database access code *in practice*—which could direct researchers where automated assistance is needed the most.

With the aim to fill this gap, in our previous work, we presented an empirical study on the challenges and perils of testing database manipulation code [13]. First, we mined open-source systems from Libraries.io and analysed 6626 projects that relied on database access technologies. We found automated tests and database manipulation code in only 332 of these projects. We

* Corresponding author.
E-mail addresses: maxime.gobert@unamur.be (M. Gobert), csaba.nagy@usi.ch (C. Nagy), henrique.rocha@gmail.com (H. Rocha), serge.demeyer@uantwerpen.be (S. Demeyer), anthony.cleve@unamur.be (A. Cleve).

https://doi.org/10.1016/j.is.2022.102105

further examined the 72 projects for which we could execute the tests and collect coverage reports. The results indicated that the *database manipulation code was poorly tested*: 46% of the projects did not test half of their DB methods, and 33% did not test DB communication.

Such a poor test coverage motivated us to understand the reasons holding back the developers from testing database access code [13]. We qualitatively analysed questions from popular Stack Exchange websites and identified the *problems* that hampered developers in writing tests. We distilled the results in a *taxonomy of issues* with 83 different problems grouped into 7 main categories. We found that developers mostly looked for general best practices.

In this paper, we extend our previous work [13] by looking at the *solutions* proposed by the developers. We further analysed the questions mined from Stack Exchange websites, focusing on the best practices. We manually labelled the top three highest-ranked answers of each question and built a *taxonomy of best practices*. Overall, we examined 598 answers to 255 questions and listed 363 different practices in the taxonomy.

The category in the taxonomy with the highest number of tags and questions was related to the testing environment, *e.g.,* proposed various tools and configurations. The second most exhaustive category was database management, *e.g.,* initialising or cleaning up a database between tests. Other categories included code structure or design guidelines, concepts, performance, processes, test characteristics, test code, and mocking.

This paper makes the following contributions:

- *Motivational Study*: An empirical study on the coverage of database access code in the tests of open-source projects [13].
- *Study I*: A qualitative assessment of developers' challenges when testing database access code and the resulting *taxonomy of issues* [13].
- *Study II*: An investigation of developers' best practices when testing database access code and the resulting *taxonomy of best practices*.
- *Dataset*: The dataset of the three studies is publicly available as an online appendix [14].

Paper structure. Section 2 presents our motivational study on database manipulation code coverage in open-source systems. Section 3 provides the first study about the challenges and problems when testing database access code. Section 4 presents the second study about the best practices proposed by developers. In Section 5, we discuss observations we made in the two studies, together with directions for researchers and practitioners. In Section 6, we examine the threats that could affect the validity of the studies. In Section 7, we discuss the related literature. Finally, we conclude the paper and outline future work in Section 8.

2. Motivational study: Test coverage of database access code in open-source systems

We first explore how developers test their database manipulation code in practice. Fig. 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.[1] We chose it because they monitor a broad set of projects (not just libraries), and maintain an extensive database of dependencies among projects.[2] 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*[3] or TestNG,[4] *i.e.,* the top Java testing frameworks according to the usage statistics of Maven central[5]; (iii) *use database access technologies*, *e.g.,* `java.sql` or `javax.persistence`; (iv) *have executable test suites*, as required by JaCoCo,[6] the test coverage tool we rely on.

We relied on version 1.4.0 of the Libraries.io dataset published in December 2018, which was the most recent release at the time of conducting the survey. We cloned 6626 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 [14]. At this stage, we identified 905 candidate projects.

In step ② *Database Access Code Analysis*, we identified the part of the source code involved in database communication. We used SQLInspect[7] for this purpose—a static code analyser for Java applications using JDBC, Hibernate, or JPA [15]. 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% [4].

SQLInspect identified database access methods in 332 of the 905 projects selected at the first stage. It did not detect database accesses in the other projects. The reason 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), or the packages may not be used at all.

In step ③ *Test Coverage Analysis*, we looked at how tests cover the DB access methods. We used the JaCoCo Maven plugin that can be integrated with the tests of a project to collect coverage data at different granularity levels.

We implemented a script modifying the pom files of the 332 projects to execute tests with JaCoCo. Maven compilation or test execution failures prevented generating a test report file for 178 projects. For example, many projects (82) did not have a pom file or tests, despite 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 summarises the main characteristics (with the minimum, quartiles, median, and maximum values) of the analysed projects. The projects are of various sizes ranging from 225 LOC to 133 kLOC. The biggest project is Speedment,[8] a Java Stream ORM. The most popular project is MyBatis[9] with 9152 stars.

[1] https://libraries.io/data.

[2] At the time of writing, Libraries.io had 2.7M unique packages, 33M repositories, and 235M interdependencies between them.

[3] https://junit.org/.

[4] https://testng.org/.

[5] https://mvnrepository.com/open-source/testing-frameworks.

[6] https://www.jacoco.org.

[7] https://bitbucket.org/csnagy/sqlinspect.

[8] https://github.com/speedment/speedment.

[9] https://github.com/mybatis/mybatis-3.

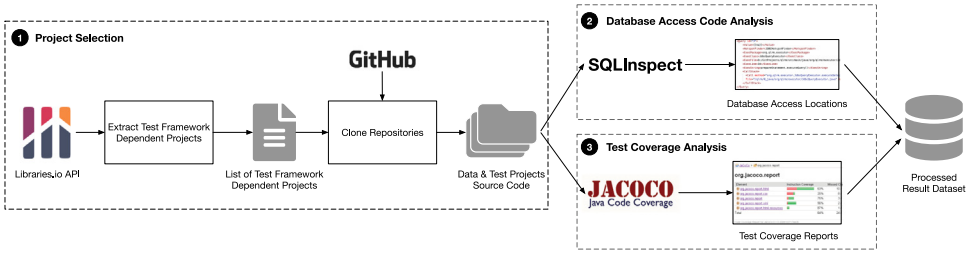

Fig. 1. Overview of the main steps for test coverage analysis of database access code.

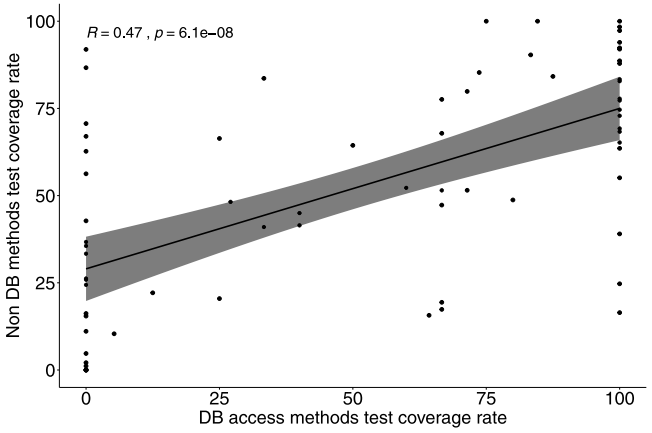

Fig. 2. Test coverage rates of Non-DB access methods vs DB access methods.

Table 1
Overview of the projects (minimum, quartiles, median, and maximum values).

Metric	Min	Q1	Med	Q3	Max
Java LOC (effective)	225	1476	3198	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

Regarding database access code, we only considered methods in production code, *i.e.*, we excluded test classes. We intentionally did not set a minimum threshold for the projects' size or database methods. Our goal was to see whether database access code is tested or not in real-life projects. If the project had only one method communicating with the DB, we wanted to see its tests.

Fig. 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. A significant number of projects with the highest coverage rate had, 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 more significant difference: 59% have a smaller coverage for database methods than 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 rises again to 35% for projects with at least five database methods.

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 substantial 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 it is even more neglected when it comes to more complex database access code.

3. Study I: Challenges & problems when testing DB access code

Our first study aims to understand the difficulties of developers when considering database access code in their tests. We seek to answer the following research question (RQ):

RQ$_1$: *What are the main challenges/problems when testing database manipulation code?*

We studied developers' most common problems on popular question-and-answer (Q&A) websites of the Stack Exchange network. The outcome of this qualitative study is a taxonomy of common issues faced by developers.

3.1. Context and data collection

We describe the method of building the taxonomy. We first present the data collection phase, then discuss the main steps of the manual labelling process.

3.1.1. Identification and extraction of questions

We targeted popular websites of the Stack Exchange network for data collection: Stack Overflow,[10] Software Engineering[11] and Code Review.[12] Stack Overflow is the largest Q&A website in software engineering, making it a popular target of mining studies. It included over 20M questions and 29M answers for software developers at the time of our analysis. 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 say that "*the best Stack Overflow questions have a bit of source code in them*".[13] More generic questions, not closely related to source code, are often discouraged as out-of-scope or opinion-based. General discussions are preferred on Software Engineering. 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 Stack Exchange network is Code Review. There, developers can ask for suggestions on a given piece of code. As they often include test code, we considered questions from Code Review as well.

[10] http://stackoverflow.com.
[11] http://softwareengineering.stackexchange.com.
[12] http://codereview.stackexchange.com.
[13] https://stackoverflow.com/help/on-topic.

Table 2
Overview of the questions selected from Stack Exchange sites.

Source	Candidate questions	Selected questions	False positives
Code Review	41	41	3
Software Engineering	174	140	25
Stack Overflow	1622	351	86
Total	1837	532	114

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). We loaded the dumps of Stack Exchange sites into a database for this filtering. 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 Stack Overflow has a tagging system for classification. However, using 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 limited resources, we targeted posts with higher impact and better quality. For this reason, we relied on the scoring system of Stack Exchange. No up-votes or a negative score may indicate problems, *e.g.,* an unclear or out-of-scope question. Therefore, we excluded posts with zero or negative ratings.

We used the Stack Overflow dump published by Stack Exchange in December 2019 and the dumps of Software Engineering and Code Review published in March 2020. A total number of 1837 questions matched the criteria: 41 on Code Review, 174 on Software Engineering and 1622 on Stack Overflow (see Table 2).

We did a first manual screening of questions on the different sites. We observed that Code Review and Software Engineering questions were closer to our scope. Therefore, we selected more questions from these sites and aimed for 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 Code Review, 3 for Software Engineering, and 13 for Stack Overflow.

3.1.2. 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 [16,17]. 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 [16].

We performed the classification process in three rounds. First, we carried out a trial round with a random set of 100 questions, wherein two of the authors assigned labels to the artefacts. We wanted to test the classification platform and see whether we needed to apply changes to our selection criteria. After this trial round, we implemented a few adjustments to our platform. Then, we labelled the remaining questions in a second round by involving four authors. Each artefact was labelled by two authors, randomly assigned to them. In the last round, we resolved conflicts where needed.

We implemented a labelling platform for this purpose. It showed the question, its relevant metadata (score, timestamp, tags) and 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 existing list in the database. If needed, they could also create new tags. 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 use consistent naming without introducing substantial bias. Indeed, the participants were not aware of the assignments.

After the second round, all 532 questions were labelled by two participants. At this point, one author reviewed all the tags and proposed merging those with identical meanings. 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 the 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. We present this taxonomy with qualitative examples in the rest of this section.

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 [17]. 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.

Fig. 3 shows the final structure of the taxonomy. There are a total number of 83 leaf issues organised in 7 main (root) categories. We indicate the total number of questions labelled with related problems for each root category. The distribution of the corresponding questions over the three sites is also provided. For example, the *Mocking* category had 54 questions, including 8 from Code Review, 17 from Stack Exchange, and 29 from Stack Overflow. 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.

We observe intriguing technical and conceptual difficulties, differentiating between them in Fig. 3 as follows. 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 Software Engineering, especially for *Maintainability/Testability* or *Method*. Technical problems are closer to the source code and mostly originate from Stack Overflow, especially for the *Framework/Tool Usage* category. Questions from Code Review 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.

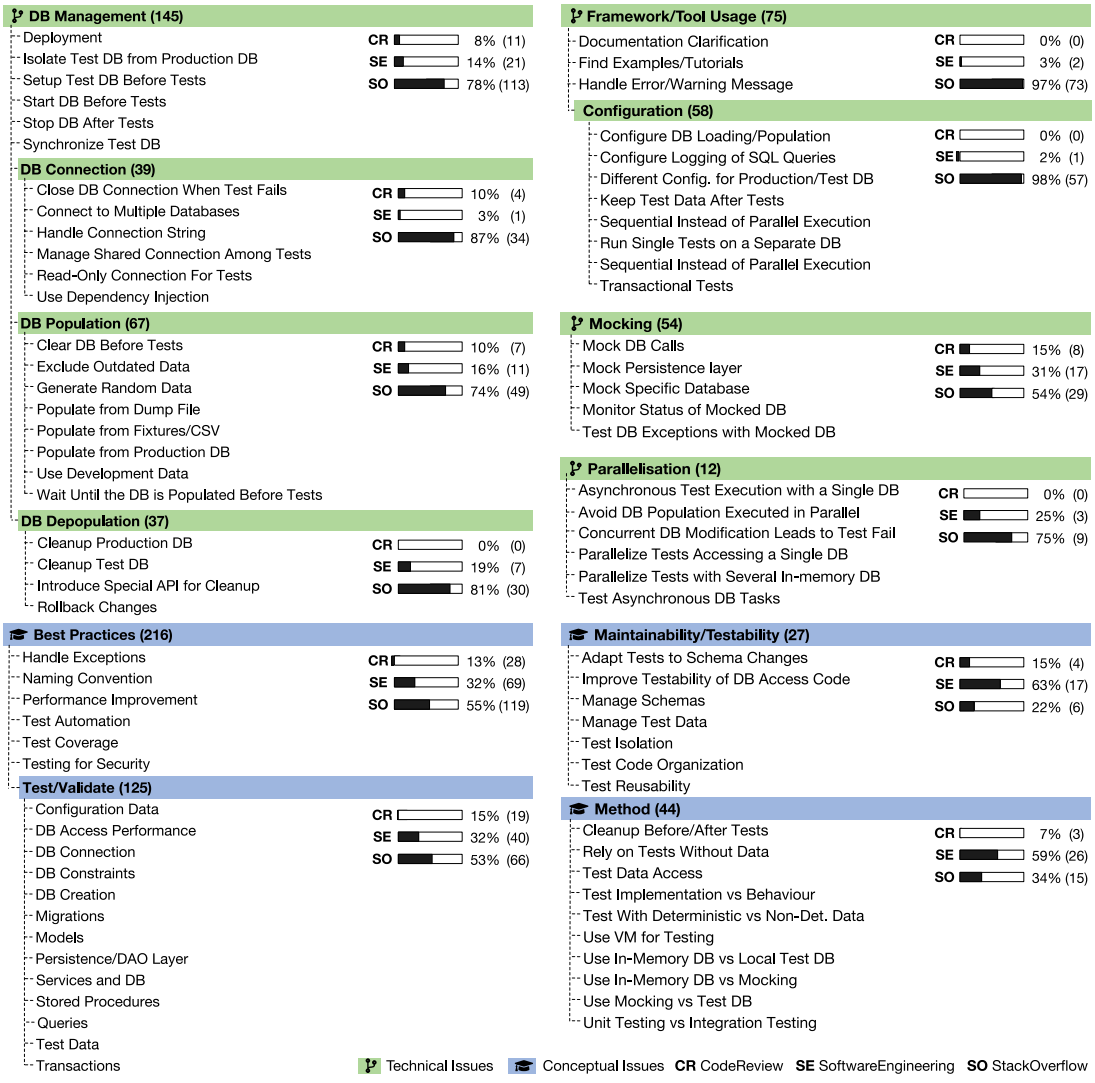

Fig. 3. Taxonomy of issues faced by developers when testing database access code.

3.2.1. DB management

The most prevalent technical issues are related to database management: 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 must 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).[14]

Questions also came from situations when the design did not support data deletion (SE2). Others faced issues keeping a test DB in synch with a production or development DB (SE3), while many had problems handling the connection to a test DB (SE4, SE5, SE6).

3.2.2. Framework/tool usage

Many problems (75) concern using a concrete tool or framework. Most of them relate to configuring a framework for a dedicated database in a test/development or production environment (SE7,SE8). These questions have high scores suggesting that many developers suffer from such issues. A question to configure Django (SE8) 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 (SE9). Interestingly, in some cases, they want to keep the test database after running their tests for debugging purposes (SE10). Many also ask for guidance to solve a particular error message in the testing framework, *e.g.,* misusing transactional tests (SE11) or configuring in-memory databases (SE12).

3.2.3. Mocking

Mocks can help by isolating the tests (*i.e.,* cutting off dependencies) and 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*" (SE13). 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*".

[14] We cite posts on Stack Exchange sites with *SE* notation. These references can be found in Table A.5 of our appendix.

Broader questions were also about the benefits of mocking (SE14) or guidelines to mock the data access layer (SE15, SE16). Technical questions tackled, for example, emulating exceptions in a mocked database (SE17). When mocking is unfeasible, it can indicate poor software design (SE18). Stored procedures (SE19) and views (SE20) made mocking impossible in other systems.

3.2.4. Parallelisation

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) (SE21). 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 (SE22). They also asked for advice to parallel test execution, *e.g.,* to handle a dedicated in-memory database per thread (SE23).

3.2.5. 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?*" (SE24). It generates discussion on mocking vs testing against an actual database. In the answers, mocking is mainly 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*" (SE15).

Best practices are also sought for performance improvements (SE25, SE26, .SE27) In particular, where mocking is not an option, solutions mainly advise using 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, *e.g.,* SQL queries embedded in code (SE30), database migration (SE31) or transactions (SE32).

3.2.6. 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 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 balancing coverage and prioritising efforts is important, then suggested generating the tests for the ORM layer.

Other questions 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).

3.2.7. 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, SE40) . 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 were related to populating a database before tests, *e.g.,* whether data should be dynamically generated or pre-populated beforehand (SE41).

A recurring discussion was on using an in-memory database versus a mocking strategy (SE42). 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 impossible (*e.g.,* because of stored procedures or views). The in-memory database was considered 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 (SE43).

4. Study II: Best practices when testing DB access code

In the previous study, we investigated the challenges of testing database access code. Here, we study the solutions proposed by developers. We seek to answer the following research question:

RQ_2: *What are the best practices when testing database manipulation code?*

We assessed RQ_2 by studying answers to the StackExchange sites' questions in our previous study. The outcome is a hierarchical taxonomy of best practices recommended by the developers.

4.1. Method

We conduct this research with an open coding process similar to the labelling in our first study (see Section 3.1.2). First, we describe this process by presenting our dataset preparation, and then we discuss the details of the manual labelling.

Table 3
Overview of the selected answers and their scores.

Site	Questions	Answers	Scores			
			Min	Max	Avg	Median
Code Review	38	50	0	19	3.22	2
Software Engineering	115	243	0	182	5.70	3
Stack Overflow	265	691	0	545	21.10	10
Total	418	984	0	545	16.39	6

4.1.1. Dataset preparation

We took RQ_1's dataset of questions about database access code testing from Stack Overflow, Software Engineering, and Code Review. We loaded the same data dump versions into a database to remain consistent with our previous research question. We exported all the answers to the 418 questions labelled previously (532 questions excluding 114 false positives). Again, we filtered the answers with negative scores as they usually suffered from quality issues, *i.e.,* they could be wrong, incomplete, or irrelevant to the question. Next, we picked the top three highest-rated answers for each question. When the accepted answer was not among the top three, we included it as a fourth answer.

Table 3 presents an overview of the answers to the questions after the selection process. The table shows, for each Stack Exchange site, the number of questions, answers, and statistics (min, max, average, and median) of their scores. Overall, we had 984 answers to 418 questions in our database. The highest-rated answer had 545 upvotes. It responded to a Stack Overflow question about "*MySQL—force not to use cache for testing speed of query*" (SE44). Also interesting to notice an answer to a Software Engineering question, which had 182 likes. The question with the title "*Shouldn't unit tests use my own methods?*" addressed the unit testing of Data Access Objects' methods (SE45).

4.1.2. Manual labelling

We used a similar open coding process to RQ_1 and manually labelled the answers. However, we had some significant differences, which we explain below.

First, we needed to adapt the labelling platform to the answers, as it was designed for labelling questions. When someone started tagging, the platform randomly assigned a question to the user and showed it with the highest-rated answers. The platform also displayed the metadata of the question (score, time, URL) and its answers (score, URL, and whether it was accepted). It also presented the problem categories assigned to the question in the previous round.

We could assign multiple labels to each problem category for each question, covering all answers. For example, if a question had been labelled in RQ_1 as "*Best Practices* > *Test/Validate* > *Queries*" and "*Mocking*", one could assign a "*Don't mock the connection*" tag to the mocking category and another "*Avoid In-Memory DB as it might not be fully compatible*" tag to the query validation.

This is a significant difference from RQ_1, where we assigned problem labels directly to the question.

Second, we added a feature to highlight relevant sentences in questions or answers. We needed this feature as the answers had many exciting ideas, guidelines, or takeaway messages, which could easily get lost in the longer texts and code examples. After our first trial round, we also anticipated that highlights would be helpful when reviewing each others' tags. We highlighted 835 text fragments, in the end, 3.27 on average per question. The highlighted sentences are available with our tagging in the replication package [14].

Table 4
Total and labelled questions per main issue categories.

Issue category	Questions		Answers	
	Total	Labelled	Total	Labelled
Best practices	216	216	510	510
DB management	145	57	351	142
Framework/Tool usage	75	34	185	75
Maintainability/Testability	27	16	50	32
Method	44	21	106	56
Mocking	54	34	123	75
Parallelisation	12	3	29	7
Total	418	255	984	598

Third, we simplified the process, and each question was first tagged by an author then reviewed by another one. In RQ_1, two authors labelled each question independently, and then a third one reviewed it. We altered the process because we experienced many conflicts due to the large number of tags, and the reviewing typically meant merging the two authors' tags. In RQ_2, the second tagger had an explicit reviewer role instead of an independent tagger.

All five authors participated in the labelling. Like in the first study, we conducted the process in three main rounds. First, we had a trial round of 27 questions, adding the highlighting feature and minor fixes to the platform. Then, we performed the first labelling round, followed by the reviewing round. After each round, we held discussions among all authors, shared our experience, and renamed or merged tags where needed (*e.g.,* because of their identical meaning).

We did not label *all* the questions from RQ_1 because of limited time constraints. Instead, we focused on *Best Practices*, the most extensive and significant problem category (see Fig. 3).

Table 4 presents an overview of the labelled questions and their answers per each main issue category. A question could belong to multiple problem categories in RQ_1. Thus, "Total" represents the union of the questions, and its figures do not necessarily equal the sum of all categories. We labelled 598 answers of 255 questions, 61% of all the answers and 100% of the *Best Practices* category. Among these, we found 4 questions without answers, and 18 with irrelevant answers, *e.g.,* they were unrelated to database manipulation code.

Finally, one author carefully reviewed all the tags and organised them into a hierarchical taxonomy. This categorisation was then discussed among the authors in multiple rounds. The final taxonomy had 363 tags in 9 main categories. We present this taxonomy through qualitative examples in Section 4.2.

4.2. Taxonomy of database testing best practices

Fig. 4 shows the main categories in the taxonomy of database testing best practices. The taxonomy follows the same hierarchical structure as we described in Section 3.2. We had a total number of 363 tags at the end of the manual labelling process. Each tag represents a solution to a problem fitting into contexts like "*The developers recommend...*" or "*The solution to this problem is...*". We organised the tags into 9 root categories following a similar classification to the taxonomy of issues. The figure presents the number of questions (❓ icon) and the number of tags (🏷 icon) for the root categories. The lower-level tags, that are not in the figure, are listed in the replication package [14]. In the rest of the section, we describe each category through intriguing examples.

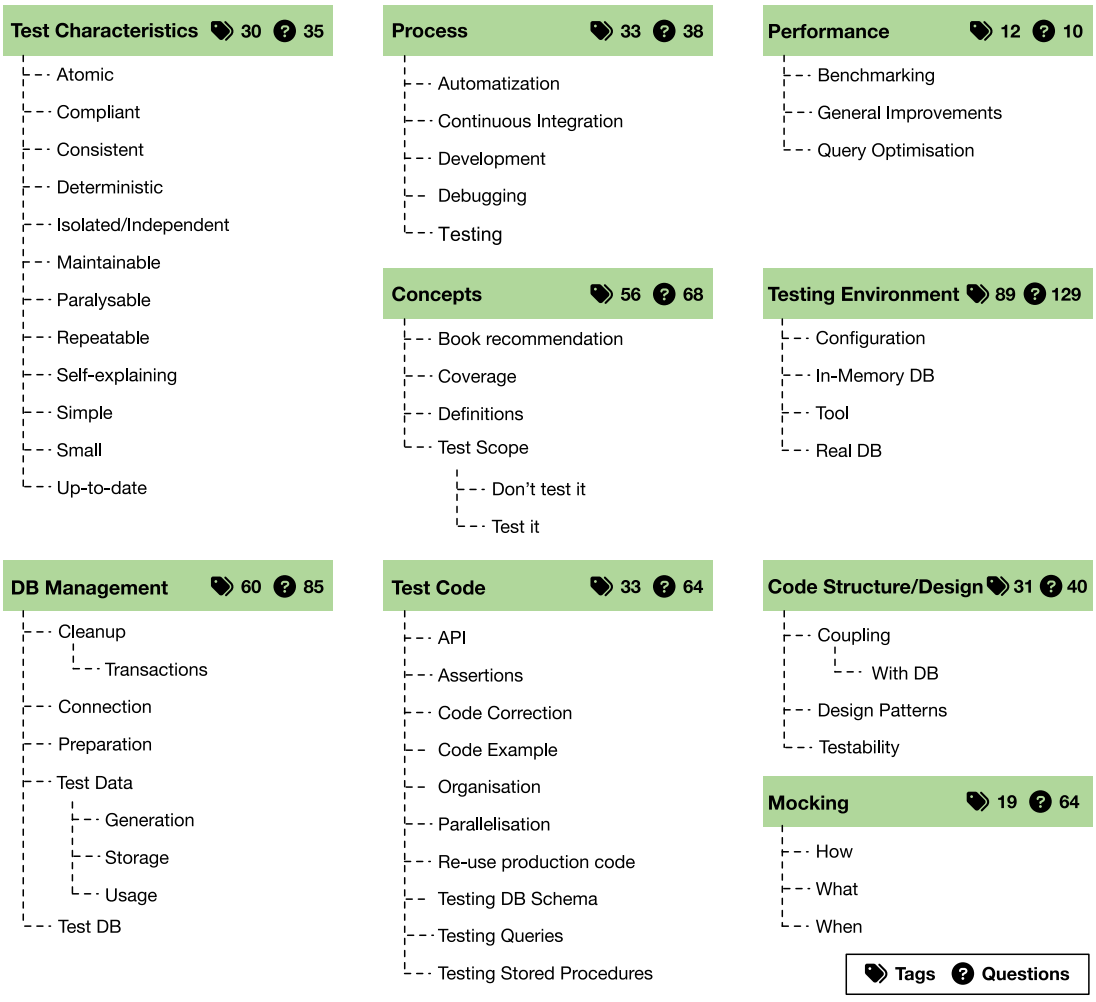

Fig. 4. Taxonomy of best practices proposed by developers when testing database access code.

4.2.1. Test characteristics

Many answers expressed that sound tests should adhere to specific characteristics and principles. Suggestions are mainly general, such as the *FIRST (Fast, Independent, Repeatable, Self Validating, Thorough)* principle (SE46), or that tests should be *atomic*, *small*, *simple*, *consistent* and *in compliance with requirements* (SE47). Many highlighted that tests should be *independent/isolated*. For example, an answer says that "*if the tests can not run independently, then they are not unit tests*" (SE48). This is especially important for database-centred applications where it can be tempting to write tests relying on a database state from a previous test, which would be against the rule of isolation. It can also affect *repeatability* and further complicate test failures. An answer points it out, "*relying on the order of your tests indicates that you are persisting state across tests. This is smelly*" (SE49). Leaving the database in a consistent state requires extra effort. Developers have various suggestions, *e.g.,* transactions, in-memory databases, mocks, fakes, stubs. We have more specific tags for these in the following taxonomy categories. It is also interesting to note the *up-to-date* test property, highlighting the importance of syncing tests with changes applied to the production database (SE50).

4.2.2. Code structure/design

This category contains recommendations targeting source code structure or design. We grouped tags into three subcategories: *Coupling*, *Design Patterns*, and *Testability*. Although they are interrelated, we differentiated them based on the primary aim of the suggestion.

Coupling was a primary concern for testing: 15 questions had answers falling under this category. We identified tags such as *Design with loose coupling for better mocking*, *Decouple tests from implementation*. An answer noted: "*Make sure you design them [service classes] with loose coupling in mind so you can mock out each dependency*" (SE51).

We separated a subgroup where they specifically targeted coupling *With DB*. Common recommendations were to *Keep logic out of the database*, have a *Separate data access layer*, and *Decouple the data layer*. As a developer said, "*I would strongly suggest decoupling it [the data access layer] from both the web and from the DB*" (SE52). Others added, "*if your objects are tightly coupled to your data layer, it is difficult to do proper unit testing*"; (SE53) "*The test knows too much about intimate details of the implementation*" (SE54). Developers also proposed keeping logic out of the database (SE19, SE55). An answer stressed that "*too much business logic is making its way into databases these days*" (SE56).

Recommendations also aimed for general *testability* improvements, such as a *Design with single responsibility* and *Break the code down to smaller testable units* — for stored procedures too (SE57).

Many suggestions mentioned *Design Patterns*. For example, developers recommended *dependency injection* in the answers to 14

questions. They employed it to automatically inject a connection to a test database instead of the production database.

Developers said, "*using dependency injection, have the unit tests select a different database than what the production (or test, or local) builds use*" (SE58); "*I've found that dependency injection is the design pattern that most helps make my code testable*" (SE59). A design recommendation was also to have a dedicated *base class* or *template interface* for tests involving database access code, *e.g.,* simplifying database initialisation, cleaning up, or mocking (SE60).

4.2.3. Concepts

We found valuable discussion threads about definitions, various aspects of tests, and coverage. We grouped them under the *Concepts* category. In particular, 34 questions were tagged as *Explanation of levels of tests: unit, interaction, integration, and acceptance tests*, the second most used tag.

Many answers state the differences between unit and integration testing when databases are involved. A common argument is that unit tests should test their units in isolation; hence, they should mock, fake, or stub the database. In contrast, integration tests consist of an actual test database with test data reset in a known state before and after each test.

An answer summarises it as follows: "*A unit test deals with a part of the code which is granular enough to be able to narrow the search of a bug if the unit test fails. There is no long polling here. No REST calls. No AJAX. No database access. REST, access to files, database calls, and all those operations which are exterior to the tested code are mocked, i.e. a mock or a stub is created for everything your tested unit needs. […] Once you have unit tests covering the critical parts of the application, you can start assembling the parts. Interfaces between different components of a system are good places for mistakes, so the integration of components needs to be tested as well. This is what integration tests are about*" (SE61).

We found interesting discussions about *Coverage*. They generally agreed that border and corner cases must be covered (SE62). They also argued the importance of testing data access code: "*If you don't test your database operations, how do you know that your data access component works?*" (SE63)

The *Scope* subcategory collects answers about *What to* or *What not to* test. Developers suggested *not to test* (i) anything that can't fail (SE64), (ii) third-party code (SE65) and (iii) code without logic (SE66). For example, an answer said, "*there are many purists who say that you shouldn't test technologies such as EF and NHibernate. They are right, they're already very stringently tested and […] it's often pointless to spend vast amounts of time testing what you don't own*" (SE65). Another argued, "*if it's really thin and there's no interesting code there, don't bother unit testing it. Don't be afraid to not unit test something if there's no real code there*" (SE66). The data access layer has a special role in this case, as noted by a developer: "*Some people […] say you should only test code which has conditional logic (IF statements etc.), which may or may not include your DAL [Data Access Layer]. Some people (often those doing TDD [Test-Driven Development]) will say you should test everything, including the DAL, and aim for 100% code coverage*" (SE67).

What to test was more sparse with suggestions such as *Check both entities and queries*, *One test for each type of output resultset (one row, multiple rows, empty resultset)*, *Test DAO[Data Access Object]/Repository normally if it performs any logic*. Developers suggested tests for pre- and postconditions (SE68), query correctness (SE69, SE70, SE71), the connection string (SE5), database schema (SE15), UI (SE72), and CRUD operations. The importance of the latter one was highlighted in an answer as follows: "*To really test your service layer, I think your layer needs to go down to DLLs and the database and write at least CRUD test*" (SE66).

We also found book recommendations, *e.g., Growing Object-Oriented Software, Guided by Tests* (SE73, SE74), *The Art of Unit Testing* (SE75 SE76), and *xUnit Test Patterns* (SE76).

4.2.4. Database management

The second most prevalent category was *Database Management*. The same problem category was also significant in the taxonomy of issues. We divided it into subcategories, *i.e.,* the preparation or clean up of a test database, the generation, storage, and usage of test data, or handling the connection to the database.

The tags revolve around reaching a known state before each test execution (*e.g.,* SE77, SE78). Indeed, the recommendation to *Clean up before each test (known state)* was the most prevalent, with 20 occurrences. We have seen mainly two best practices as follows. (i) *Using an actual database*, loading the schema and test data before each test, then cleaning the database before the next test case. There are various tuning practices. For example, one can optimise database initialisation by loading the schema once then populating only with necessary minimal data. Then clean up only the modified records if there were any. Many also proposed in-memory databases for performance reasons or simply because they are easily destroyable after each test run. (ii) The other thread was about *transactions*, *i.e.,* load the schema and test data, run the tests in transactions, then rollup changes. We tagged these as *Use transaction scopes (which revert the state of DB after each test)*.

Interestingly, many frameworks provided support for these techniques. We counted answers recommending *setUp() & tearDown() methods* for database initialisation and clean up in unit tests. *Spring* also supports test execution through *transaction management*.

Developers had intriguing arguments for these approaches as follows. "*Just remember: At the start of the test, everything is created, at the end of the test everything is destroyed*" (SE79). "*I suggest either connecting to an empty DB and filling with data in the test set-up phase, then either emptying it or deleting it in the test clean up phase or creating a copy of a constant DB, connecting to it in the test set-up phase, then deleting in the test clean up phase. It is important to do this per test, so that the tests are truly independent*" (SE55). "*Using setUp() and tearDown() to get a consistent state for your data before running your tests is (imho) a fine way to write DB-driven unit tests*" (SE80).

We also grouped generic DB-related recommendations in the *Test DB* subcategory. For instance, an answer advised avoiding a different type of DB than in production to prevent cross-platform issues (SE81).

Finally, the *Test data* subcategory consists of recommendations to generate, store, and use testing data. For example, *Use the ORM [Object-Relational Mapping] to initialise test data* (SE65) or *Generate random but valid data entries* (SE82).

4.2.5. Mocking

We separated a *Mocking* category in the taxonomy due to the prevalence of these tags. The *Use Mocking* tag was assigned to the most questions; we encountered it 46 times. A recurring discussion of *Concepts* argued unit tests should imply mocking. An answer stated it as follows: "*If you test class B, which is a client of A, then usually you mock the entire A object with something else, […]. Likewise, when you write a unit test for class C, which is a client of B, you would mock something that takes the role of B*" (SE5).

Answers proposed *what* to mock, *e.g.,* "*You shouldn't mock calls to the database because that would defeat the purpose. What you SHOULD mock are, for example, calls to your DAO from, say, a service layer. Mocking allows you to test methods in isolation*". (SE83). Others elaborated on *how* to mock and presented complete code examples (SE84). Mocking frameworks (*e.g.,* Easymock, Mockito, Moq, Rhino Mocks) were often recommended. Many threads also discussed the differences between mocks, stubs and fake objects (SE85, SE86, SE27).

Interestingly, a few answers hinted that caution is needed in using mocks (SE87, SE88). An answer argues that "*I'd try to*

use them [mocking] sparingly in unit tests since by using them you actually try to test the function implementation and not the adherence to its interface" (SE88).

4.2.6. Performance

We separated a group for performance tuning or optimisation recommendations. These were often context-specific or fine-tuning alternatives. For example, an answer (SE81) recommended using *tmpfs* (*i.e.,* run the database on an in-memory filesystem) when a *per se* in-memory database was not a viable option (*e.g.,* because of a different SQL dialect). Another answer suggested using prepared statements in loops (SE82) or deferring garbage collection for the tests (SE27). A recurring recommendation was using a lightweight database (in-memory) to optimise performance (SE89, SE90).

4.2.7. Process

Process category groups tags about automatisation, continuous integration, test failure debugging, or the testing process in general (*e.g.,* SE78 SE50). The most recurring discussions revolved around integration and unit tests. In particular, they suggested separating integration from unit tests (SE5), highlighted the importance of automated UI tests (SE91), or proposed testing the persistence layer manually (SE74).

Recommendations were also related to debugging, *e.g.,* logging failing or slow queries (SE92). They also suggested integration testing in certain situations (SE93, SE31). As an example, an answer says, "*there's no way to unit test Spring Data JPA [Java Persistence API] repositories reasonably for a simple reason: it's way too cumbersome to mock all the parts of the JPA API we invoke to bootstrap the repositories. Unit tests don't make too much sense here anyway, as you're usually not writing any implementation code yourself […] so that integration testing is the most reasonable approach*" (SE94).

4.2.8. Test code

Recommendations also focused on the source code of the tests, *e.g.,* recommended specific APIs. We group these under the *Test Code* category. The answers include configuration fixes, general how-tos of APIs, code examples, or asserts in tests.

For example, developers often stress the importance of *single asserts*. An answer says, "*The tests are far more granular, each test verifies one property […] single asserts are good*" (SE65). Another developer argues that "*there should only be one reason for a test to fail*" (SE95).

We learned that many testing frameworks actually provide APIs to support database testing. In particular, we found API recommendations for *Android*, *Entity Framework*, *Django*, *LINQ*, *Spring Framework*, and *NUnit*.

4.2.9. Testing environment

We separated a group of *Testing Environment* recommendations. This group has the highest number of tags and questions due to the several tools named in the answers. It consists of advice about the application or build frameworks, configurations, database management system, or various tool proposals.

Many answers recommended in-memory databases, *e.g.,* H2, HSQLBD, HyperSQL, Ephermal PG, and SQLite. These are not exclusively relational. For example, a thread talks about using MongoDB in memory mode (SE12).

Tool recommendation includes also mocking libraries (*e.g.,* Easymock, Mockito, Moq, or Spring Data Mock). Database configuration concerned various technologies such as Laravel (SE96), JUnit (SE97), Spring Boot (SE98), Django (SE99, SE100). These were mainly related to configuring a test database, fixtures, or migrations. Flyway and Liquibase were also notable tools in this context (SE81). They were mentioned to manage (track, version, and deploy) database schema changes.

We also found tool recommendations for vulnerability testing, *e.g.,* SQL injections (SE101, SE102), and tools for virtualisation or container environments (SE81).

Finally, it is interesting to notice that 24 answers proposed DBUnit,[15] a testing framework for database-centred applications (SE103, SE89, SE77, SE81).

An answer remarks that "*the DbUnit framework (a testing framework allowing to put a database in a known state and to perform assertion against its content) has a page listing database testing best practices that, to my experience, are true*" (SE78).

5. Discussion and implications

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

Maintainability of database tests. Test maintenance was a frequent issue. A developer aptly outlined, "*if it is hard to maintain, you're doing it wrong*" (SE39). Many answers recommended following sound characteristics or principles. However, their implementation guidelines were often unclear. A common challenge was to isolate tests. Best practices suggested mocking in unit tests and a separate testing database for integration tests. They preferred in-memory databases for this purpose. A well-designed source code where the database access code is loosely coupled to other parts also played a crucial role in maintainability. Many struggled to keep tests in sync with database schema changes. Indeed, developers hardly get any support for this task.

Our study is exploratory by nature. 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 [18,19] 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 [5,20]. Such methods could be extended to the testing context, *e.g.,* to maintain a mapping between schema elements and mocks.

In-memory database vs. actual database vs. mocking. We have seen many arguments for and against mocking, in-memory databases, or the actual database. In our motivational study, we found that 19 out of the 72 projects (26%) used mocks: 17 had Mockito,[16] and 2 had EasyMock tests.[17] This low number surprised us, as mocking was *the* recommended approach for unit tests to decouple them from the database. This is in line with the findings of Trautsch and Grabowski [21], 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.

We also found positive examples when manually inspected top-starred projects from the motivational study.

For instance, MyBatis,[18] a popular project with 17k stars and 11.4k forks, had a 74% of its database access methods covered with tests. It is a persistence framework, hence the high coverage. They use mocking for unit testing[19] and a test database with test

15 https://www.dbunit.org/.
16 https://site.mockito.org/.
17 https://easymock.org/.
18 https://github.com/mybatis/mybatis-3.
19 https://tinyurl.com/2dm37hv5.

data in scripts for integration testing.[20] The test database is initialised before executing all tests, using the *@BeforeAll* annotation. Finally, each test ends with a rollback function to get back to the initial state of the database.[21]

As another example, AxonFramework,[22] a framework for building event-driven microsystems, has 2.6k stars and 699 forks. They use dependency injection in the Spring framework to mock the data source of tests.[23] Their tests flush the database before each test using the *@BeforeEach* annotation.[24]

In any respect, developers need help in the implementation of database-related tests. They use frameworks' features when available, but they would benefit from more automated support in this context. Researchers have already explored generating tests with mocks [22,23]. Such tools' emergence and initial success (*e.g.,* EasyMock,[17] MockNeat[25]) encourages similar approaches.

Database 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 struggled to configure frameworks with multiple database connections. Consequently, our most extensive category among the solutions was the testing environment. Almost half of the questions and a third of the tags were related to this category.

Testing frameworks could provide more support to developers with database-dedicated features. Especially if these are configurable from the build systems. Some frameworks already offer similar functionalities. For example, *Spring Test* has *JdbcTestUtils*,[26] a collection of JDBC-related functions. It also supports test fixtures and transactional tests. Rails and Django offer similar features.

Answers also mentioned dedicated tools to support databases in unit or integration tests. For example, DBUnit is a JUnit extension targeted at database-driven projects, "*an excellent way to avoid the myriad of problems that can occur when one test case corrupts the database and causes subsequent tests to fail or exacerbate the damage*".[15] PHPUnit's database extension has similar features.[27]

We observed that the most desired features pertained to the initial configuration of databases and the efficient recovery of the database state between successive tests. The high demand and many problems related to such features indicate that developers' needs remain unexplored in this field. Moreover, only a few answers tackled trending technologies such as clouds and virtualisation or docker containers. Such technologies could offer robust solutions, but they appear to be unexploited.

Further research is necessary to improve testing practices as far as database access is concerned. As an answer notes, "*the database is the bread and butter of most business*" (SE104).

6. Threats to validity

In this section, we discuss threats to the validity of our motivational study and two research questions.

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 the case of highly dynamic query construction.

For test coverage, we rely on JaCoCo, a state-of-the-art tool used in industry and academia [24]. It might miss execution paths, and its configuration can influence the coverage results (*e.g.,* 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 Stack Exchange questions is exposed to subjectiveness. To mitigate this risk, two authors examined each post independently, and a third author resolved conflicts in the first study. In the second study, one author assessed a question first, and then a second author reviewed it. Already assigned tags could influence a reviewer. However, the statistics confirmed that reviewers did not simply accept the taggings but also removed or added new tags when needed. The questions had 2.51 tags on average in the first round and 3.25 after the reviews. The total number of tags has also increased 21% from 300 to 365.

External validity. Our motivational study is exploratory by nature. It considers various types of projects in terms of their application domain, size, and intensity of DB interactions. They are, however, all from Libraries.io and limited to the Java programming language. Projects not considered in our study might lead to other results.

In our first qualitative study, we extracted questions from three different Stack Exchange sites, intending to reach a higher level of diversity. We selected higher-ranked questions which are likely to influence more developers. Similarly, we labelled only the top three answers in the second study. This might introduce a bias towards the posts we selected. In reality, developers might face even more diverse challenges when (not) testing database code.

7. Related work

In this section, we overview the related work of our study. First, we present empirical studies inspiring our research. Then, we discuss approaches to support testing database applications and present studies mining Stack Overflow.

7.1. Empirical studies on software testing

Our research got motivated and inspired by more general studies analysing testing practices and maintainability issues.

In this context, Kochhar et al. [25] investigated the adoption of testing in open source projects. They studied more than 20 thousand projects and explored the correlation of test cases with project development characteristics, including project size, development team size, number of bugs, number of bug reporters, and the underlying programming languages.

Greiler et al. [26] conducted a qualitative study about testing practices of plug-in based applications. They interviewed 25 senior practitioners and surveyed more than 150 professionals. As an outcome, they provide an overview of testing practices. They identified obstacles limiting the adoption of automated tests and proposed recommendations and areas for future research.

Beller et al. [27] conducted a large-scale field study on testing practices, monitoring five months of activities from 416 software

20 https://tinyurl.com/39vkk8j2.
21 https://tinyurl.com/2p9ftswk.
22 https://github.com/AxonFramework/AxonFramework.
23 https://tinyurl.com/yckjv4h5.
24 https://tinyurl.com/5bs4nbzp.
25 https://github.com/nomemory/mockneat.
26 https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html.
27 https://phpunit.de/manual/6.5/en/database.html.

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. [28] analysed over 80k open-source projects. They found that only 17% of those projects included test cases, and 76% did not implement testing patterns that would ease maintainability.

Trautsch and Grabowski [21] analysed more than 70k revisions of 10 Python projects. They observed that most projects had minimal unit tests, resulting in poor test coverage. They also showed that developers tended to overestimate the coverage of their tests and that mocks did not significantly influence the number of unit tests.

Our qualitative analysis revealed that many Stack Exchange questions concerned mocking, a testing technique often used to isolate the component under test. Spadini et al. [29] empirically analysed the usage of mocking dependencies on testing. Their goal was to understand how and why developers used mocking. They explored four projects with 2178 test dependencies and surveyed 105 developers. Their results indicate that mocking is often used on dependencies, complicating tests dependent on external resources.

Alsharif et al. [18] studied the understandability of auto-generated database tests. They argued that studies on creating database tests did not consider the human cost to understand such tests. They used five database test generators and asked participants to explain the results. The authors highlighted two main findings: (i) the values in *insert* statements affected understandability, and (ii) using *null* values with integrity constraints may confuse human subjects on the outcome of tests.

7.2. Support for testing database applications

Several researchers have proposed approaches to *support* testing database applications.

In this regard, Deng et al. [30] proposed a white-box testing approach for web applications. They extracted URLs from the application source code to create a path graph and generate test cases.

Ran et al. [31] proposed a similar framework for black-box testing of web applications. They used a directed graph of web page transitions and database interactions to generate test sequences and capture how the database gets updated with the test cases.

Marcozzi et al. [32] proposed an approach to symbolic execution of SQL statements integrated with the traditional symbolic execution of the application source code. Their approach handled interdependent interactions between the application and the database. They also presented a symbolic execution algorithm for a subset of Java and SQL, implemented as a testing tool for generating test cases.

Another important aspect of testing database applications is *specialised coverage* since standard coverage techniques appear unsuitable for preserving all database constraints.

In this sense, Kapfhammer and Soffa [33] presented a test coverage technique to monitor interactions with database elements. They employed instrumentation of the application and test cases to capture SQL statement usage. Then they collected database-aware coverage reports of a test suite. Their coverage results also considered database interactions from the test cases and the program methods. They used six database-centric applications as case studies and observed a testing time increase from 13% to 54% as a drawback.

Tuya et al. [34] presented an approach to measure SQL query coverage. They argued that SQL queries embedded in code are not considered for test design, although queries implement an important part of the business logic. Their approach identified test data requirements for SQL statements and expressed them as a set of predicate rules. They demonstrated it on an open-source ERP application as a case study.

7.3. Mining stack exchange discussions

We collected and classified questions in Stack Exchange sites through a multi-label approach, inspired by previous work in our field.

Vasilescu et al. [35] investigated relationships between StackOverflow questions/answers and GitHub commits. They argued that developers could find suitable technical solutions in StackOverflow, affecting their commit productivity on GitHub. Their study showed a positive correlation indicating that developers' activity on StackOverflow affected their commit activity on GitHub.

Finally, Gonzalez et al. [36] proposed a five-way classifier approach assigning multiple tags to StackOverflow questions. They used a dataset of over 3 million questions.

7.4. Summary

The analysis of related research shows that database access code is sufficiently different from regular code to warrant specialised approaches. Several research works proposed approaches to *support* testing database access code. Nevertheless, no research has investigated *how* developers test database access code *in practice*, the main *issues* they face in this context, and the *best practices* recommended by the developer community.

Our previous work [13] tried to fill this gap by analysing how tests in open-source systems cover the database access code and investigating the challenges of testing database access code. In this extension, we study the best practices of the developers.

8. Conclusion

We studied developers' challenges and best practices in testing database access code. In our first motivational study, we analysed 72 open-source Java projects and investigated how their tests cover database access code. 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 two qualitative studies. (i) First, we analysed 532 StackExchange questions about database code testing and identified 83 issues, classified in a taxonomy of 7 main categories. We found that developers mostly look for general best practices to test DB access code. Concerning technical issues, they ask mostly about DB handling, mocking, parallelisation, or framework/tool usage. (ii) Next, we examined the answers to the questions. We distinguished 363 best practices and organised them with 9 main categories in a taxonomy. Most of the tags and questions were related to the testing environment and proposed various tools or configurations. The second most significant category was about database management best practices for initialising and cleaning a test database. The remaining categories were code structure or design, concepts, performance, processes, test characteristics, test code, and mocking.

We addressed an unexplored field of testing database communication and identified developers' main difficulties and best practices. Further investigation is necessary, however, such as the validation of the two taxonomies with testing practitioners. Feedback from practitioners could guide researchers towards dedicated techniques and tools to assist developers when testing DB access code.

Table A.5
StackExchange posts.

Id	Title	URL
SE1	Integration Testing best practices	http://www.stackoverflow.com/questions/1328730
SE2	TDD: "Test Only" Methods	http://www.stackoverflow.com/questions/2295965
SE3	Good approach/Strategy to keep integration...	https://softwareengineering.stackexchange.com/questions/302458
SE4	Managing database connections for unit tests	https://codereview.stackexchange.com/questions/201711
SE5	Unit-Tests and databases: At which point do...	https://softwareengineering.stackexchange.com/questions/206539
SE6	In JUnit 5, how to run code before all tests	http://www.stackoverflow.com/questions/43282798
SE7	Spring integration tests with profile	http://www.stackoverflow.com/questions/20551681
SE8	Different db for testing in Django?	http://www.stackoverflow.com/questions/4650509
SE9	How to run Django tests on Heroku	http://www.stackoverflow.com/questions/13705328
SE10	Django test to use existing database	http://www.stackoverflow.com/questions/6250353
SE11	How to suppress...	http://www.stackoverflow.com/questions/44080733
SE12	In-memory MongoDB for test?	http://www.stackoverflow.com/questions/13607732
SE13	Should mock objects for tests be created at...	https://softwareengineering.stackexchange.com/questions/216072
SE14	What's the idea behind mocking data access...	https://softwareengineering.stackexchange.com/questions/262686
SE15	Ways of unit testing data access layer	http://www.stackoverflow.com/questions/15000908
SE16	How to Mock Test Data for complicated...	https://softwareengineering.stackexchange.com/questions/405456
SE17	How to throw a SqlException when needed for...	http://www.stackoverflow.com/questions/1386962
SE18	How to write unit tests without mocking data	https://softwareengineering.stackexchange.com/questions/193614
SE19	Unit testing Systems with Logic Tightly...	https://softwareengineering.stackexchange.com/questions/356087
SE20	Rethinking testing strategy	https://softwareengineering.stackexchange.com/questions/212887
SE21	How to turn off parallel execution of tests...	http://www.stackoverflow.com/questions/11899723
SE22	Unit testing Room and LiveData	http://www.stackoverflow.com/questions/44270688
SE23	How to run tests in parallel in Django?	http://www.stackoverflow.com/questions/5303819
SE24	What's the best strategy for unit-testing...	http://www.stackoverflow.com/questions/145131
SE25	How do you handle testing applications that...	http://www.stackoverflow.com/questions/2393428
SE26	Integration Testing best practices	http://www.stackoverflow.com/questions/1328730
SE27	Rails 3.0.7 -> How do you get your tests to...	http://www.stackoverflow.com/questions/6087329
SE28	How can I automatically test my site for SQL...	http://www.stackoverflow.com/questions/9685884
SE29	Django: is there a way to count SQL queries...	http://www.stackoverflow.com/questions/1254170
SE30	Do you test your SQL/HQL/Criteria?	https://softwareengineering.stackexchange.com/questions/33182
SE31	How do I test database migrations?	http://www.stackoverflow.com/questions/2332400
SE32	How to show SQL query log generated by a...	http://www.stackoverflow.com/questions/6884408
SE33	SQL queries in integration tests	https://softwareengineering.stackexchange.com/questions/326003
SE34	Integrating Automated Web Testing Into Build...	http://www.stackoverflow.com/questions/1240057
SE35	What is a good method of storing test data...	https://softwareengineering.stackexchange.com/questions/238971
SE36	Do the terms "unit test" and "integration"...	https://softwareengineering.stackexchange.com/questions/302559
SE37	Databases and Unit/Integration Testing	https://softwareengineering.stackexchange.com/questions/101273
SE38	How do you unit test business applications?	http://www.stackoverflow.com/questions/38598
SE39	Unit Test vs Integration Test in Web...	http://www.stackoverflow.com/questions/15292751
SE40	What's the best strategy for unit-testing...	http://www.stackoverflow.com/questions/145131
SE41	Should on each test create and nuke a...	https://softwareengineering.stackexchange.com/questions/394145
SE42	Testing—In-Memory DB vs. Mocking	https://softwareengineering.stackexchange.com/questions/358491
SE43	Unit testing a service to return items from...	https://codereview.stackexchange.com/questions/98301
SE44	MySQL—force not to use cache for testing...	http://www.stackoverflow.com/questions/181894
SE45	Shouldn't unit tests use my own methods?	https://softwareengineering.stackexchange.com/questions/330304
SE46	How to test data based on SQL queries?	https://softwareengineering.stackexchange.com/questions/315178
SE47	Testing my VB.NET code?	https://softwareengineering.stackexchange.com/questions/159943
SE48	Should each unit test be able to be run...	https://softwareengineering.stackexchange.com/questions/64306
SE49	Is it bad form to count on the order of your...	http://www.stackoverflow.com/questions/497699
SE50	My first model test in PHPUnit	https://codereview.stackexchange.com/questions/59662
SE51	How do I unit test a WCF service?	http://www.stackoverflow.com/questions/37375
SE52	unit/integration testing web service proxy...	https://softwareengineering.stackexchange.com/questions/167906
SE53	How to unit test an object with database...	http://www.stackoverflow.com/questions/30710
SE54	Unit test for a method that adds tweets to a...	https://codereview.stackexchange.com/questions/128287
SE55	Unit/Integration Testing my DAL	https://softwareengineering.stackexchange.com/questions/133448
SE56	Is Unit Testing your SQL taking TDD Too far?	http://www.stackoverflow.com/questions/730488
SE57	Am I Unit Testing or Integration Testing my...	https://softwareengineering.stackexchange.com/questions/81801
SE58	How to test the data access layer?	https://softwareengineering.stackexchange.com/questions/219362
SE59	Is a class that is hard to unit test badly...	http://www.stackoverflow.com/questions/2658859
SE60	Basic Unit Test of Application Service,...	https://codereview.stackexchange.com/questions/234960
SE61	How to create unit/integration tests for my...	https://softwareengineering.stackexchange.com/questions/214529
SE62	SQLite Database inserting + Unit tests in...	https://codereview.stackexchange.com/questions/132742
SE63	Unit Testing—What not to test	http://www.stackoverflow.com/questions/1316848
SE64	Unit Testing—What not to test	http://www.stackoverflow.com/questions/1316848
SE65	How are people unit testing with Entity...	http://www.stackoverflow.com/questions/22690877
SE66	How do I unit test a WCF service?	http://www.stackoverflow.com/questions/37375
SE67	Should I Unit Test Data Access Layer? Is...	http://www.stackoverflow.com/questions/3333120
SE68	How to add rigor to my testing?	https://softwareengineering.stackexchange.com/questions/270422
SE69	How to write unit tests for database calls	http://www.stackoverflow.com/questions/1217736
SE70	What kind of unit tests should be written...	https://softwareengineering.stackexchange.com/questions/336880
SE71	Unit testing with MongoDB	http://www.stackoverflow.com/questions/7413985

(*continued on next page*)

Table A.5 (*continued*).

Id	Title	URL
SE72	Beginning Automated Testing	http://www.stackoverflow.com/questions/12907080
SE73	Does TDD include integration tests?	http://www.stackoverflow.com/questions/18988040
SE74	Do we need test data or can we rely on unit...	https://softwareengineering.stackexchange.com/questions/113441
SE75	Never written much unit tests, how can I...	https://softwareengineering.stackexchange.com/questions/128859
SE76	Removing the "integration test scam" -...	https://softwareengineering.stackexchange.com/questions/135011
SE77	How to test the data access layer?	https://softwareengineering.stackexchange.com/questions/219362
SE78	How to do database unit testing?	http://www.stackoverflow.com/questions/3772093
SE79	Should you hard code your data across all...	https://softwareengineering.stackexchange.com/questions/212678
SE80	Phpunit testing with database	http://www.stackoverflow.com/questions/4585345
SE81	How to simulate a DB for testing (Java)?	http://www.stackoverflow.com/questions/928760
SE82	Creating many random test database entries	https://codereview.stackexchange.com/questions/14411
SE83	Why do we write mock objects when writing...	https://softwareengineering.stackexchange.com/questions/61366
SE84	How to test DAO methods using Mockito?	http://www.stackoverflow.com/questions/28388204
SE85	How to create unit/integration tests for my...	https://softwareengineering.stackexchange.com/questions/214529
SE86	Testing properties with private setters	https://softwareengineering.stackexchange.com/questions/317121
SE87	Unit Testing with massive lookup tables?	https://softwareengineering.stackexchange.com/questions/287735
SE88	Where is the line between unit testing...	https://softwareengineering.stackexchange.com/questions/322909
SE89	How do I unit test jdbc code in java?	http://www.stackoverflow.com/questions/266370
SE90	How to Test Web Code?	http://www.stackoverflow.com/questions/2913
SE91	Unit testing database application with...	http://www.stackoverflow.com/questions/2609204
SE92	Best practices for database testing with...	http://www.stackoverflow.com/questions/3697815
SE93	Unit-testing an adapter	https://codereview.stackexchange.com/questions/38906
SE94	How to test Spring Data repositories?	http://www.stackoverflow.com/questions/23435937
SE95	Is this good practice with unit-testing?	https://codereview.stackexchange.com/questions/37584
SE96	Laravel 5 : Use different database for...	http://www.stackoverflow.com/questions/35227226
SE97	JUnit tests always rollback the transactions	http://www.stackoverflow.com/questions/9817388
SE98	Initialising a database before Spring Boot...	http://www.stackoverflow.com/questions/38262430
SE99	How can I specify a database for Django...	http://www.stackoverflow.com/questions/4606756
SE100	How to create table during Django tests with...	http://www.stackoverflow.com/questions/7020966
SE101	Do you have any SQL Injection Testing "Ammo"?	http://www.stackoverflow.com/questions/274659
SE102	Testing for security vulnerabilities in web...	http://www.stackoverflow.com/questions/2351315
SE103	Best way to test SQL queries	http://www.stackoverflow.com/questions/754527
SE104	How do you unit test your T-SQL	http://www.stackoverflow.com/questions/2765212

✎ *Replication package.* All data, scripts, and detailed results of our study publicly are available in a replication package [14].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research was supported by (i) the F.R.S.-FNRS (Fonds de la Recherche Scientifique) and FWO-Vlaanderen EOS project SECO-ASSIST (30446992), (ii) the F.R.S.-FNRS and SNF (Swiss National Science Foundation) PDR project INSTINCT (35270712), and (iii) Flanders Make vzw.

Appendix A. StackExchange References

See Table A.5.

References

[1] M. Stonebraker, D. Deng, M.L. Brodie, Application-database co-evolution: A new design and development paradigm, in: New England Database Day, 2017.

[2] A. Cleve, A. Brogneaux, J. Hainaut, A conceptual approach to database applications evolution, in: Proceedings of the 29th International Conference on Conceptual Modeling (ER 2010), Springer Berlin Heidelberg, 2010, pp. 132–145, http://dx.doi.org/10.1007/978-3-642-16373-9_10.

[3] D. Qiu, B. Li, Z. Su, An empirical analysis of the co-evolution of schema and code in database applications, in: Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013), ACM, 2013, pp. 125–135, http://dx.doi.org/10.1145/2491411.2491431.

[4] L. Meurice, C. Nagy, A. Cleve, Static analysis of dynamic database usage in Java systems, in: Proceedings of the 28th International Conference on Advanced Information Systems Engineering (CAiSE 2016), Springer, 2016, pp. 491–506, http://dx.doi.org/10.1007/978-3-319-39696-5_30.

[5] L. Meurice, C. Nagy, A. Cleve, 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), 2016, pp. 262–273, http://dx.doi.org/10.1109/QRS.2016.38.

[6] T.-H. Chen, W. Shang, A.E. Hassan, M. Nasser, P. Flora, Detecting problems in the database access code of large scale systems, in: Proceedings of the 38th International Conference on Software Engineering (ICSE 2016), ACM, 2016, pp. 71–80, http://dx.doi.org/10.1145/2889160.2889228.

[7] P. Vassiliadis, A.V. Zarras, Survival in schema evolution: Putting the lives of survivor and dead tables in counterpoint, in: Proceedings of the 29th International Conference on Advanced Information Systems Engineering (CAiSE 2017), Springer International Publishing, 2017, pp. 333–347, http://dx.doi.org/10.1007/978-3-319-59536-8_21.

[8] J. Delplanque, A. Etien, N. Anquetil, S. Ducasse, Recommendations for evolving relational databases, in: Proceedings of the 32nd International Conference on Advanced Information Systems Engineering (CAiSE 2020), Springer International Publishing, 2020, pp. 498–514, http://dx.doi.org/10.1007/978-3-030-49435-3_31.

[9] D. Chays, S. Dan, P.G. Frankl, F.I. Vokolos, E.J. Weber, A framework for testing database applications, in: Proceedings of the 2000 International Symposium on Software Testing and Analysis (ISSTA 2000), ACM, 2000, pp. 147–157, http://dx.doi.org/10.1145/347324.348954.

[10] J. Castelein, M. Aniche, M. Soltani, A. Panichella, A. van Deursen, Search-based test data generation for SQL queries, in: Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), ACM, 2018, pp. 1220–1230, http://dx.doi.org/10.1145/3180155.3180202.

[11] D. Garg, A. Datta, Test case prioritization due to database changes in web applications, in: Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST 2012), IEEE, 2012, pp. 726–730, http://dx.doi.org/10.1109/ICST.2012.163.

[12] R.H. Rosero, O.S. Gómez, G.D.R. Rafael, Regression testing of database applications under an incremental software development setting, IEEE Access 5 (2017) 18419–18428, http://dx.doi.org/10.1109/ACCESS.2017.2749502.

[13] M. Gobert, C. Nagy, H. Rocha, S. Demeyer, A. Cleve, Challenges and perils of testing database manipulation code, in: M. La Rosa, S. Sadiq, E. Teniente (Eds.), Advanced Information Systems Engineering, Springer International Publishing, 2021, pp. 229–245, http://dx.doi.org/10.1007/978-3-030-79382-1_14.

[14] M. Gobert, C. Nagy, H. Rocha, S. Demeyer, A. Cleve, Replication package, 2022, https://github.com/csnagy/infosys2022-db-manipulation-testing, Accessed: March, 2022.

[15] C. Nagy, A. Cleve, Sqlinspect: a static analyzer to inspect database usage in Java applications, in: Proceedings of the 40th International Conference on

Software Engineering: Companion Proceeedings (ICSE 2018), ACM, 2018, pp. 93–96, http://dx.doi.org/10.1145/3183440.3183496.
[16] K. Petersen, S. Vakkalanka, L. Kuzniarz, Guidelines for conducting systematic mapping studies in software engineering: An update, Inf. Softw. Technol. 64 (2015) 1–18, http://dx.doi.org/10.1016/j.infsof.2015.03.007.
[17] M. Usman, R. Britto, J. Börstler, E. Mendes, Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method, Inf. Softw. Technol. 85 (2017) 43–59, http://dx.doi.org/10.1016/j.infsof.2017.01.006.
[18] A. Alsharif, G.M. Kapfhammer, P. McMinn, What factors make SQL test cases understandable for testers? A human study of automated test data generation techniques, in: Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME 2019), 2019, pp. 437–448, http://dx.doi.org/10.1109/ICSME.2019.00076.
[19] M. Riaz, E. Mendes, E. Tempero, Towards maintainability prediction for relational database-driven software applications: Evidence from software practitioners, in: Proceedings of the 2010 International Conference on Advances in Software Engineering (ASEA 2010), Springer, 2010, pp. 110–119, http://dx.doi.org/10.1007/978-3-642-17578-7_12.
[20] A. Maule, W. Emmerich, D.S. Rosenblum, Impact analysis of database schema changes, in: Proceedings of the 30th ACM/IEEE International Conference on Software Engineering (ICSE 2008), ACM, 2008, pp. 451–460, http://dx.doi.org/10.1145/1368088.1368150.
[21] F. Trautsch, J. Grabowski, Are there any unit tests? An empirical study on unit testing in open source python projects, in: Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017), IEEE, 2017, pp. 207–218, http://dx.doi.org/10.1109/ICST.2017.26.
[22] B. Pasternak, S. Tyszberowicz, A. Yehudai, GenUTest: a unit test and mock aspect generation tool, Int. J. Softw. Tools Technol. Transf. 11 (4) (2009) 273, http://dx.doi.org/10.1007/s10009-009-0115-4.
[23] A. Arcuri, G. Fraser, R. Just, Private API access and functional mocking in automated unit test generation, in: Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017), 2017, pp. 126–137, http://dx.doi.org/10.1109/ICST.2017.19.
[24] M. Ivanković, G. Petrović, R. Just, G. Fraser, Code coverage at google, in: Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019), ACM, 2019, pp. 955–963, http://dx.doi.org/10.1145/3338906.3340459.
[25] P.S. Kochhar, T.F. Bissyandé, D. Lo, L. Jiang, An empirical study of adoption of software testing in open source projects, in: Proceedings of the 13th International Conference on Quality Software (QSIC 2013), 2013, pp. 103–112, http://dx.doi.org/10.1109/QSIC.2013.57.
[26] M. Greiler, A. van Deursen, M.D. Storey, Test confessions: A study of testing practices for plug-in systems, in: Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), IEEE Computer Society, 2012, pp. 244–254, http://dx.doi.org/10.1109/ICSE.2012.6227189.
[27] M. Beller, G. Gousios, A. Panichella, A. Zaidman, When, how, and why developers (do not) test in their IDEs, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), ACM, 2015, pp. 179–190, http://dx.doi.org/10.1145/2786805.2786843.
[28] D. Gonzalez, J.C. Santos, A. Popovich, M. Mirakhorli, M. Nagappan, A large-scale study on the usage of testing patterns that address maintainability attributes: Patterns for ease of modification, diagnoses, and comprehension, in: Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017), 2017, pp. 391–401, http://dx.doi.org/10.1109/MSR.2017.8.
[29] D. Spadini, M. Aniche, M. Bruntink, A. Bacchelli, Mock objects for testing Java systems, Empir. Softw. Eng. 24 (3) (2019) 1461–1498, http://dx.doi.org/10.1007/s10664-018-9663-0.
[30] Y. Deng, P. Frankl, J. Wang, Testing web database applications, SIGSOFT Softw. Eng. Notes 29 (5) (2004) 1–10, http://dx.doi.org/10.1145/1022494.1022528.
[31] L. Ran, C. Dyreson, A. Andrews, R. Bryce, C. Mallery, Building test cases and oracles to automate the testing of web database applications, Inf. Softw. Technol. 51 (2) (2009) 460–477, http://dx.doi.org/10.1016/j.infsof.2008.05.016.
[32] M. Marcozzi, W. Vanhoof, J.-L. Hainaut, Relational symbolic execution of SQL code for unit testing of database programs, Science of Computer Programming 105 (2015) 44–72, http://dx.doi.org/10.1016/j.scico.2015.03.005.
[33] G.M. Kapfhammer, M.L. Soffa, Database-aware test coverage monitoring, in: Proceedings of the 1st India Software Engineering Conference (ISEC 2008), ACM, 2008, pp. 77–86, http://dx.doi.org/10.1145/1342211.1342228.
[34] J. Tuya, M.J. Suárez-Cabal, C. de la Riva, Full predicate coverage for testing SQL database queries, Softw. Test. Verif. Reliab. 20 (2010) 237–288, http://dx.doi.org/10.1002/stvr.424.
[35] B. Vasilescu, V. Filkov, A. Serebrenik, StackOverflow and GitHub: Associations between software development and crowdsourced knowledge, in: Proceedings of the 2013 International Conference on Social Computing (ICSC 2013), 2013, pp. 188–195, http://dx.doi.org/10.1109/SocialCom.2013.35.
[36] J.R. Cedeño González, J.J. Flores Romero, M.G. Guerrero, F. Calderón, Multi-class multi-tag classifier system for StackOverflow questions, in: Proceedings of the 2015 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC 2015), 2015, pp. 1–6, http://dx.doi.org/10.1109/ROPEC.2015.7395121.