Quantifying the Performance Impact of SQL Antipatterns on Mobile Applications

Yingjun Lyu, Ali Alotaibi, William G. J. Halfond
Department of Computer Science
University of Southern California
Email: {yingjunl, aalotaib, halfond}@usc.edu

***Abstract*—In mobile applications, local databases have become an important component, providing mobile users with a responsive and secure service for data access and management. However, using local databases comes with a cost. Studies have shown that they are one of the most resource consuming components on mobile devices. Improper usage of the local database can even severely impact the responsiveness of an application. In this paper, we conducted a literature review and a benchmark study to investigate problematic programming practices with respect to database usage. Our results present a comprehensive overview of the current knowledge about these practices, and introduce new knowledge about the impact of these practices on the resource consumption of mobile applications.**

***Index Terms*—SQL antipatterns, mobile applications, database, energy, performance, security**

I. Introduction

Developers provide innovative services via their apps by combining data from a wide assortment of sensors and services. Easy access and management of that data has become essential in many mobile apps. To provide this data management service, many developers incorporate the use of local databases, such as SQLite [1], in their apps. Local databases store information directly on the mobile devices and help apps to provide reliable and responsive service even when the underlying mobile device does not have a reliable connection. The benefits of local databases have led to their widespread use and popularity; a recent study found that they are used in over 50% of all mobile apps [2].

Although local databases offer many benefits to developers, their use comes with potential problems. Many database operations, such as transactions, require file locking, rollback capabilities, and extensive I/O. As a result, these operations can consume a large amount of mobile device resources, such as energy and CPU time. In fact, recent studies have found that local database services are one of the top three resource consuming services on mobile devices [3], [4]. High resource usage can lead to complaints about the app's battery usage and responsiveness, and even lower ratings for the apps [5], [6], [7]. Therefore, developers strive to ensure the good performance of their apps even when they make use of local databases.

Despite a clear motivation to efficiently use database operations, developers face two general problems. First, they lack consolidated information about the good and bad ways to use database operations. Although practitioners and researchers have tried to distill the various improper ways of using database operations into SQL antipatterns, the information about these antipatterns is scattered across different books and research papers. The studies that target the discovery, detection, or repair of SQL antipatterns usually only cover a small set of the wide range of SQL antipatterns. Second, developers lack information about the potential tradeoffs of these antipatterns in terms of their impact on the runtime and energy usage of their mobile apps. This lack of performance information is due to two reasons. First, existing performance studies on local databases focus on the cost of individual local database operations on mobile devices and do not take the SQL antipatterns into consideration (e.g., [8], [9], [2]). Second, existing studies on SQL antipatterns do not cover the performance impact on mobile apps (e.g., [10], [11], [12], [13], [14], [15]). They target web applications and remote databases, which are substantially different from mobile applications, where local databases are frequently used. Furthermore, the energy aspect of antipatterns, which is essential to mobile devices, is not considered in those studies.

Although important, assessing the impact of SQL antipatterns on app performance can be very challenging. This is due to fact that SQL antipatterns can manifest differently in different applications. For the same antipattern, different instances can have a different impact depending on many factors, such as what kind of SQL statements are issued, how large are the underlying database tables, etc. A naive approach that overlooks how local databases are used in mobile application would not be able to comprehensively quantify the performance impact of SQL antipatterns.

To address these issues, we conducted a large scale study to identify the different types of SQL antipatterns and their impact on the performance of mobile apps. We conducted a systematic survey that follows best practices of literature reviews [16] so as to explore the current knowledge about SQL antipatterns. We then carried out a benchmark study to quantify the impact of the discovered SQL antipatterns on the runtime and energy usage of mobile apps. In our benchmark study, we employed a range of dynamic and static program analysis techniques. These techniques allowed us to (1) identify real-world antipattern instances from a large pool of mobile apps, (2) build a representative benchmark suite, and (3) quantify and compare the performance impact of SQL antipatterns in a uniform manner.

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00015

IEEE computer society

Our investigation revealed several interesting observations that provide concrete guidelines for app developers and motivate areas for future research work in the software engineering community. We summarized eleven SQL antipatterns through our literature review. We discovered that eight of them have a significant impact on the runtime and energy consumption of local database operations. As for the rest of the eleven SQL antipatterns, they are security oriented and we discovered that they can be fixed with a minor performance cost.

The rest of the paper is organized as follows: In Section II and Section III, we explain the methodologies of the literature review and benchmark study, respectively. In Section IV, we report our findings. We discuss the threats to the validity of our study in Section V and conclude our paper in Section VI.

II. Literature Review Methodology

In this section, we present our literature review protocol. Based on Kitchenham's [16] systematic review guidelines, we first specify the research questions (Section II-A). We then present the search process, which includes the initial search strategy (Section II-B1), the selection criteria (Section II-B2), and the citation search strategy (Section II-B3).

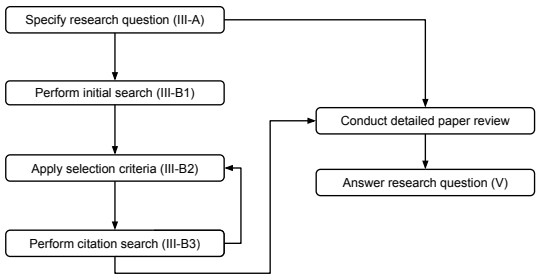

Fig. 1: Literature review process flow

A. Research Questions

The goal of the study is to provide a comprehensive overview of SQL antipatterns covering their definition, impact, detection methods, and repair methods. In this study, we address the following research questions:

RQ 1: What are the SQL antipatterns known to date? The information about SQL antipatterns is very fragmented. The vast majority of related papers often focus on specific problems and only cover a small set of SQL antipatterns. The aim of this research question is to distill the scattered antipatterns from the literature and provide a comprehensive collection of SQL antipatterns.

RQ 2: In what aspects do the SQL antipatterns affect the software quality?

The impact of SQL antipatterns on the software quality can be multidimensional. The purpose of this research question is to qualitatively explore which aspects of software quality each SQL antipattern affects and why it affects them.

RQ 3: What is the current state of research with respect to the detection and optimization of the SQL antipatterns?

The SQL antipatterns target different database programing problems. It is likely that some SQL antipatterns received more research attention than others. The aim of this research question is to explore the methods proposed to detect and fix each of the discovered SQL antipatterns.

B. Literature Search Protocol

The literature search protocol aims to identify related studies that form the basis of the survey. The workflow of the search process is shown in Figure 1. Our search protocol has three major steps: conducting an initial search to collect a primary set of related literature, applying selection criteria to filter the search results, and performing a citation search iteratively to maximize the set of related studies.

1) Initial Search: The goal of of this step is to collect an initial set of the related studies. In this section, we explain our search strategy, which includes the selection of search engines, keywords, and search methods.

To collect an initial set of papers, we started our search with the papers that were published in the related major conferences during the past ten years. Since the studies on SQL antipatterns may involve research in multiple areas, i.e., software engineering, database, and security, the initial set of papers were selected from these fields. The conferences selected are shown in Table I, each of which represents top conferences in its field. As the publications of these conferences are stored in the electronic databases, IEEE Explore and ACM Digital Library, we utilized their searching capability as explained below.

Given the scope of our literature review, we focused on the selected keywords to perform the search on the papers' titles and abstracts. Since each research domain has its own terminology, different keywords were chosen for different research fields. The keywords chosen are listed under the "Keyword" column in Table I. To retrieve all related papers, each research domain in our search string was represented as a disjunction of corresponding keywords provided above. This means that for example, in the security domain, any paper that has "database" or "sql" in its title or abstract would be selected as the initial set.

2) Selection Criteria: Not all of the papers returned by the search engines can help to answer the research questions. Therefore, we used the following inclusion and exclusion criteria to further filter the candidate papers.

Inclusion Criteria. A search result should be included if it: (1) discovers/defines at least one SQL antipattern, (2) explores the impact of at least one SQL antipattern, (3) proposes detection methods, or (4) develops repair mechanisms.

Exclusion Criteria. A search result should be excluded if it: (1) focuses on general coding practices or antipatterns that are not related to databases, (2) discusses problems at the system level, such as the database system, operating system, or file system, or (3) discusses problems at the design level instead of the implementation level.

TABLE I: Initial Conferences

Venue	Research Field	Abbreviation	Keyword
International Conference on Software Engineering	Software Engineering	ICSE	code smell, antipattern, database, sql
Foundations of Software Engineering	Software Engineering	FSE	code smell, antipattern, database, sql
Automated Software Engineering	Software Engineering	ASE	code smell, antipattern, database, sql
Computer and Communications Security	Security	CCS	database, sql
Security and Privacy	Security	S&P	database, sql
Special Interest Group on Management of Data	Database	SIGMOD	database application, developer

Each selected paper obtained from the search went through a manual inspection of the title, keywords, and abstract. The inspection applied the above criteria leading to the inclusion or exclusion of the papers.

3) Forward/Backward Citation Search: The objective of this step is to obtain a more comprehensive list of studies by investigating the citations. After performing the previous two steps, we had an initial set of studies. However, these primary studies only covered the selected conferences in the selected years. To expand the list of studies, we performed an iterative forward and backward citation search. The forward search investigated the works that cite the primary studies. The backward search inspected the articles that are cited by the primary studies. The inspection followed the same selection criteria and protocol defined in Section II-B2. This citation search process leverages the authors' knowledge and expertise in their domain to identify additional closely related work. As shown in Figure 1, we conducted this step iteratively for multiple rounds in order to maximize the list of related studies. The iteration continued until the newly obtained citations highly overlapped with the previous collected studies. In other words, until few new articles that meet the selection criteria could be found.

III. Benchmark Study Methodology

The goal of the benchmark study is to understand the impact of the SQL antipatterns on the resource consumption of mobile applications. In this section, we present the methodology of the study. We specify the research question in Section III-A, explain the design of the benchmark suite in Section III-B, and present the process of measuring the performance impact in Section III-C.

A. Research Question

Using the benchmark suite, we aim to answer the following research question:

RQ 4: How do the SQL antipatterns impact the resource consumption of local database operations in mobile applications? Fixing an antipattern may lead to consuming more or less resources. It is also possible that the resource consumption stays unchanged after the fix. The purpose of this research question is to explore the quantitative performance impact (in terms of runtime and energy) of the discovered SQL antipatterns on mobile applications.

B. Benchmark Design

Benchmarks are one of the most popular tools to compare the performance characteristics of different implementations. In order to write high-performance code, developers often write small benchmark programs to measure the relative performance of one approach against another. A benchmark suite contains a group of benchmarks that share a similar performance testing purpose.

In order to quantify the performance impact of the different SQL antipatterns, we constructed a benchmark suite that was designed to measure the resource consumption of two implementations, the antipattern version and the fixed version, of performing the same database related task. When designing the benchmark suite, we aimed at ensuring that the performance impact we obtained from our study are generalizable to the real-world impact of SQL antipatterns. Achieving this goal is challenging because there are many factors that can affect the costs of database operations, as well as the quantitative performance impact of the SQL antipatterns. The values of these factors can vary across different applications as well. A manually synthesized benchmark, such as the one published in an online blog post [17], may not reflect how developers use local databases in mobile applications.

To overcome this challenge, we based the antipattern version of the implementation on real-world antipattern instances that were collected at a large scale. In order to ensure the generalizability of the study results, our benchmark suite took different performance affecting factors into consideration and simulated their representative values in mobile applications. In the following sections, we explain how the performance affecting factors were chosen and varied (Section III-B1), and how the antipattern version and the fixed version of the implementation were constructed (Section III-B2).

1) Controlling Performance Affecting Factors: In order to isolate the performance impact of the SQL antipatterns, the factors that can affect the performance of database operations must be controlled. Since our study focused on the antipatterns that are in the coding aspects, we varied the factors that can be changed in the application code and controlled the other factors. In our study, the varied factors are the number of SQL statements issued, the SQL statement forms, the underlying table forms, the size of the table, and the two ways of implementation (i.e., the antipattern and its fix). The SQL statement form is defined to be the SQL keywords contained in a SQL statement in order, e.g., *SELECT FROM ORDER BY*, *DELETE FROM*, etc. The table form is defined to be a multiset where the elements are the types of columns in the table, e.g., {*INTEGER, INTEGER, TEXT*}. As for the other factors, including the hardware, the file system, the database management system, the database configuration, etc, these were controlled to be the same across different measurements.

When we varied the values of the performance affecting factors, we aimed at choosing values that are representative of their typical values in mobile applications. However, this is very challenging because the values may differ in individual applications. A naive approach that arbitrarily chooses the varied values from a predetermined range may not reflect how developers use local databases in mobile applications. To overcome the challenge, we made use of static and dynamic analyses, which allowed us to analyze mobile applications at a large scale and to identify representative factor values.

For the two factors, SQL statement forms and table forms, we utilized static analysis to obtain their representative values from the real-world antipattern instances. More specifically, we first implemented a set of static detectors based on the detection algorithms we discovered in our literature review. In case of the absence of a detection algorithm, we came up with a preliminary analysis based on the definition and rationale of the corresponding SQL antipattern. We then ran our detectors on a set of 1,000 marketplace applications from the Google Play app store [18]. These apps were chosen by collecting the top ranked apps from each of the 34 app categories defined in the store. It took around fifteen hours for the detectors to analyze these apps and identify the various SQL antipattern instances. After identifying the real-world instances, we split them into equivalence classes based on their SQL statement forms. We then ranked the equivalence classes based on their sizes. (The sizes represent how frequently the corresponding forms appeared in the detected instances.) For each of the top ranked equivalence classes, we randomly selected one instance. We applied the same process to the table forms. Following this protocol, we selected 52 antipattern instances, whose SQL statement forms and table forms represented at least 72% and on average 90% of all the detected instances.

Similar to the SQL statement forms and table forms, we also need to vary the values for the number of SQL statements issued and the size of the tables. Since the number of columns and the column types have been determined by the table form, the size of the table depends on the number of rows and the size of the columns whose sizes are flexible, e.g., the string/text/varchar types of columns. Instead of naively selecting random values for these numeric factors, we tried to estimate a range of values from real-world apps. We ran the 1,000 apps in our application corpus using a widely-used workload generation tool, PUMA [19]. During the execution, we logged the SQL statements issued along with their execution timestamps. From the logs, we computed empirical values for the number of rows in a table, the size of a string type column, and the number of SQL statements issued. These were estimated by computing the average number of insert statements to a table, the average length of string data values in the insert statements, and the average number of SQL statements issued in a consecutive sequence. Lastly, we chose the average values and varied them by one and two standard deviations.

After identifying the values of the performance affecting factors, the last challenge is to determine the way to combine them. In order to thoroughly test the impact of the antipatterns under different settings, we used the Cartesian product on the sets of factor values. Each tuple in the resulting Cartesian product corresponds to a benchmark in the benchmark suite. A test in the suite consists of the two benchmarks that share the same factor values except for the programming practice, i.e., one of them is the antipattern version while the other one is the fixed version.

2) Constructing the Antipattern and its Fix: To generate the antipattern version of the implemention, we utilized the selected real-world antipattern instances. We minimized the app code so that only the database operations and code constructs that are related to the detected antipattern instances were included in our benchmark. Note that we manually verified the decompiled code to ensure that they were true positives before using them as benchmarks.

To generate the fixed version of the implementation, we followed the strategies proposed in the literature and fixed the antipattern version of the benchmark. For each of the SQL antipatterns, the literature would generally illustrate what kind of database access task a developer wants to perform, how the developer would implement it in an inefficient/insecure manner (i.e., the antipattern) and in an efficient/secure manner (i.e., the fix). We summarized these characteristics of SQL antipatterns and their fixes in Section IV.

C. Performance Evaluation

In the evaluation, we measured and analyzed the energy consumption and runtime of the benchmark code. In the process, we first inserted probes to the source code of the benchmark. We recorded the start and end times of the part of the code that was changed or inserted after fixing the antipattern. We then deployed the app containing the benchmark code on a Samsung Galaxy S5 that was connected to a power meter. When the app was launched, the code was automatically triggered and the energy and runtime measurements were recorded during the execution.

Measuring Energy Consumption and Execution Time: We leveraged the Monsoon power meter [20] to measure energy. This power measurement platform sampled the power consumption of the smartphone at a frequency of 5KHz and synchronized these samples with the standard Unix time. By aligning starting and ending times with the Monsoon measurement samples, we obtained power measurements for each target API invocation. After calculating the individual runtime and energy consumption, their values were summed up as the overall cost.

Conducting Experiment: To reduce the impact of any non-deterministic or uncontrolled behavior, we repeated the measurements fifteen times. We obtained an average relative standard error of 4.4%, which demonstrated the stability of our measurement results. In general, a relative standard error of 25% or greater are subject to high sampling error and should be used with caution [21]. The relative standard error we obtained was significantly smaller than this threshold.

For each measurement, the database was reinitialized to ensure consistency across different measurements. Between each measurement, we added a waiting time to avoid any impact from tail energy behavior and to allow the device to cool down.

Analyzing Data: For each antipattern, we measured the energy consumption and runtime difference between two code versions under various settings. We did not assume the measurement data followed a normal distribution and ran a Mann-Whitney U test (α = .05) to compute the statistical significance of the differences in measurements for both the energy and runtime measurements. The mean and median differences were also computed.

IV. Results

Following the literature review protocol, we identified 57 related studies (these included 56 research papers and 1 book). To answer the research questions, we manually inspected the collected literature.

In the following sections, we list the definitions and rationales for the eleven identified SQL antipatterns. For each of them, we also present and discuss the detection and repair methods, along with the quantitative impact we obtained from the benchmark study.

In Table II, we summarize the performance impacts. A positive number means the resource consumption increases after fixing the antipattern. A negative number means the resource consumption decreases. For each antipattern, we show the mean (absolute value and percentage) and median of the differences in resource consumption. Since we constructed benchmarks for each antipattern using various detected antipattern instances under different settings, in the table, we show the benchmark tests that established statistical significance in the runtime and energy differences. In general, there were less cases that established statistical significance for energy than runtime. This is because there were more fluctuations in the energy measurements.

A. Antipattern: Unbatched-Writes

Definition: A sequence of database writes issued separately instead of being batched in a single transaction [14], [22], [23], [2].

Rationale: If the database writes trigger transactions repetitively, the transaction processing overhead of each write request can lead to significant inefficiency.

1) Techniques: Chen and colleagues [14] targeted the Unbatched-Writes antipattern for applications developed using Object-relational Mapping (ORM) frameworks. These frameworks map standard APIs to SQL statements and allow developers to access the database by manipulating objects. Chen's technique statically detects the database-accessing functions that are invoked inside a loop without being optimized by some ORM batching annotations (e.g., **Batch**).

Tamayo and colleagues [12] developed a technique that can suggest batching opportunities by analyzing the dependencies between multiple SQL statements. If a sequence of data writes does not have certain data dependencies, the technique will suggest submitting all the write statements together with a single database write API.

RAT-TRAP [24] statically detects database writes that happen within loops and that will trigger inefficient autocommit behaviors. RAT-TRAP then uses additional analyses to identify those that are optimizable and rewrites the code.

2) Performance Impact:

a) Repair Strategy: As suggested by Lyu *et al*. [24], we inserted explicit transaction control so as to batch the writes.

b) Result: As shown in Table II, after fixing the antipattern, the runtime and energy were both reduced by over 90% on average with statistical significance. The results suggest that transactions can be very resource-intensive. The more implicit transactions are triggered, the more performance impact this antipattern can make. The average runtime reduction even exceeded 100 ms (a common threshold for human to perceive delay [25], [26]), which means this antipattern can noticeably impact user experience.

B. Antipattern: Not-Merging-Projection-Predicates

Definition: Instead of issuing a single SQL query to read all the needed columns at once, developers issue multiple queries, each of which only reads a subset of the required columns [10], [27]. For example, "*SELECT name FROM User*" and "*SELECT id FROM User*" could have been merged to "*SELECT id, name FROM User*".

Rationale: This antipattern describes the problem of transmitting too many small database requests instead of aggregating them. The processing overhead of each request, e.g., the cost of network round trip (when communicating with remote databases) and query processing, can introduce unnecessary inefficiency.

1) Techniques: Manjhi and colleagues [10] developed static analyses to detect the Not-Merging-Projection-Predicates antipattern. Their analyses check if (1) the second query is executed whenever the first query is executed, and (2) the queries are identical except for the projection predicates. To automate the transformation, they used a source-to-source compiler to merge the queries.

Arzamasova and colleagues [27] analyzed the existing SQL query logs to look for the Not-Merging-Projection-Predicates antipattern. Their log analyzer checks if a sequence of SQL queries has equivalent FROM and WHERE clauses, but a different SELECT clause.

2) Performance Impact:

a) Repair Strategy: To fix this antipattern, we merged the projection predicates and read all the needed columns with one query as suggested by Manjhi *et al*. [10].

b) Result: As shown in Table II, the runtime and energy consumption were reduced 32% and 36% on average. Most of the runtime differences established statistical significance. These results suggest that Not-Merging-Projection-Predicates can introduce performance overhead to local database operations. Although there does not exist a network communication overhead in local databases, the other costs, such as scanning the database tables and finding the target data, can be resource

TABLE II: SQL antipatterns and their performance impacts

Antipattern	Runtime mean (ms)	Runtime median (ms)	Runtime significance	Energy mean (mJ)	Energy median (mJ)	Energy significance
Unbatched-Writes	-263.99 (-94%)	-218.03	108/108	-152.65 (-96%)	-104.57	105/108
Not-Merging-Projection-Predicates	-7.93 (-32%)	-2.73	34/36	-5.04 (36%)	-1.03	18/36
Not-Merging-Selection-Predicates	-18.59 (-66%)	-9.9	14/18	-5.64 (-66%)	-1.63	12/18
Loop-to-Join	-1467.58 (-94%)	-538.13	9/9	-3455.40 (-96%)	-863.33	9/9
Vulnerable-Query	1.43 (18%)	0.40	7/45	0.49 (15%)	0	4/45
Not-Using-Parameterized-Query	-21.70 (-50%)	-11.07	49/54	-31.02 (-55%)	-11.27	48/54
Not-Caching	-8.37 (-7%)	-3.87	24/27	-7.06 (-6%)	-0.53	14/27
Unnecessary-Column-Retrieval	-8.90 (-27%)	-2.93	95/126	-10.42 (-29%)	-2.07	78/126
Unnecessary-Row-Retrieval	-9.41 (-65%)	-2.20	35/45	-10.31 (-70%)	-1.67	28/45
Unbounded-Query	1.57 (7%)	0.80	31/81	1.18 (7%)	0.07	11/81
Readable-Password	0.84 (0.5%)	0.67	9/18	0.10 (0.1%)	0	2/18

expensive. By merging the projection predicates and retrieving the necessary columns at once, the aforementioned costs can be reduced.

C. Antipattern: Not-Merging-Selection-Predicates

Definition: Instead of issuing a single SQL query to read all the needed rows at once, developers issue multiple select queries, each of which only reads a subset of the required rows [10], [27].

Rationale: The rationale behind this antipattern is similar to the Not-Merging-Projection-Predicates antipattern, as they both describe the problem of not aggregating the queries and introduce performance overheads.

1) Techniques: To detect this antipattern, Manjhi and colleagues [10] developed static analyses to check if (1) the second query is executed whenever the first query is executed, and (2) the queries are identical, except one selection clause. To automate the transformation, they used a source-to-source compiler to merge the queries.

Arzamasova and colleagues [27] also covered the Not-Merging-Selection-Predicates antipattern in their query log analyzer. The analyzer identifies the SQL queries that have equal SELECT and FROM clauses, but a different WHERE clause.

2) Performance Impact:

a) Repair Strategy: As suggested by Arzamasova *et al*. [27], we merged the selection predicates and retrieved all the needed rows with one query. The columns used in the merged selection predicates were added to the projection attributes. For example, "*SELECT id FROM ScaleTable WHERE Name = 'Optimism'*" and "*SELECT id FROM ScaleTable WHERE Name = 'Caring'*" were merged to "*SELECT id, Name FROM ScaleTable WHERE Name in ('Optimism', 'Caring')*". To ensure that the functionality of the program remained unchanged, the returned rows were split in the application code according to the values in the *Name* column.

b) Result: The runtime and energy consumption were both reduced 66% after fixing the antipattern. However, not all of the performance differences established statistical significance. We even observed two cases where the resource consumption increased after the fix. When looking in depth at those cases, we found that in the antipattern version, the cost of the queries was low. After merging them in the fixed version, the performance gain was lower than or similar to the overhead of splitting the data. Nevertheless, in most of the cases, there was a big performance reduction after fixing the antipattern. This was because multiple costly queries were combined. These results demonstrated that the Not-Merging-Selection-Predicates antipattern can have a negative impact on the resource consumption of local database operations, but this impact depends on the cost of the queries and the overhead of splitting the results.

D. Antipattern: Loop-to-Join

Definition: Instead of joining two tables and querying from the joint table, developers first query from one table to get multiple values and then for each value (using a loop structure) query from another table [28], [10], [29].

Rationale: This antipattern causes unnecessary frequent database requests. The processing overhead of each request can introduce unnecessary inefficiency.

1) Techniques: Manjhi and colleagues [10] proposed a detection analysis for the Loop-to-Join antipattern. The analysis checks if: (1) the loop iterates using the result of a previous query, (2) the loop issues a query in each iteration, and (3) the previous query is executed whenever the loop executes. Once the pattern is identified, they used the work done by Kim [30] to replace the small queries by a merged query.

Cheung [28] and Emani [31] both proposed techniques to optimize the Loop-to-Join antipattern for applications developed using ORM. Their techniques automatically transform imperative code fragments (written using ORM and nested loops) into SQL queries that use joins.

2) Performance Impact:

a) Repair Strategy: To fix this antipattern, we joined the two tables that were used in the first query and second query, and then queried from the joint table as suggested by the literature [10], [28], [31].

b) Result: As we can see in Table II, the runtime and energy consumption were reduced 94% and 96% on average after fixing the antipattern. The statistical test established statistical significance for all of the results. As the runtime difference was over 100 ms, this antipattern is very likely to noticeably impact user experience. The reason behind this big performance impact is that fixing the antipattern can reduce the number of queries issued from 1+N (N stands for the number

of rows retrieved from the first table) to 1. By looking in depth at the results, we found that when the size of the retrieved data exceeded a certain limit, the time required to iterate over the retrieved rows increased significantly. In Android, the retrieved data is stored in a `Cursor` object [32]. It has a buffer of 2 MB in size by default to store the retrieved data. When a row that is not in the buffer is requested and the buffer is full, the memory needs to be freed and the buffer will be refreshed. In the antipattern version, the buffer refreshing overhead was triggered when the application code iterated over the data that was retrieved from the first table. In the fixed version, since the result iteration was avoided by joining the tables, the buffer refreshing overhead was avoided as well. In summary, fixing the antipattern can reduce the number of database operations, reduce the amount of data transferred, and reduce the time that is required to iterate over the result. These ultimately led to significant performance improvement in terms of runtime and energy.

E. Antipattern: Vulnerable-Query

Definition: A vulnerable input is not properly sanitized and is concatenated to a SQL query [33], [34], [35], [36]. For example, if a string derived from an untrusted source, such as user input, is concatenated to a query using string operations, the resulting query would be considered a vulnerable query.

Rationale: This antipattern is the root cause of the SQL injection vulnerability. SQL injection refers to a type of attack in which data that derives from untrusted input sources is included in an SQL query in such a way that part of the vulnerable input is treated as SQL code. This allows the attacker to manipulate the syntax of the SQL query, threatening the security of the application and its underlying database.

1) Techniques: There is a wide range of techniques in the security domain that tackle this antipattern. These techniques include, but are not limited to static code checkers [37], [38], combined static and dynamic analyses [39], [40], taint based approaches [41], [42], [43], etc. These techniques have been well-summarized by existing surveys [44], [45], [46].

2) Performance Impact:

a) Repair Strategy: To fix this antipattern, we sanitized the vulnerable input using the encoding functions provided by the OWASP Enterprise Security API (ESAPI) Toolkits [47]. The encoding functions use the proper escaping scheme for the database and sanitize the provided input. The DBMS will not confuse the sanitized input with the SQL code written by the developer, thus avoiding any possible SQL injection vulnerabilities. Sanitizing the input is not the only way to tackle SQL injection. Another typical way is to used parameterized queries. In Section IV-F, we also evaluated how parameterization would impact the performance.

b) Result: Our experiment results showed that sanitizing the vulnerable input can increase the runtime and energy consumption by 18% and 16%, respectively. But as we can see in Table II, the absolute cost was relatively low. The performance increases did not establish statistical significance for most of the cases as well. When we investigated into the ones with significant differences, we discovered that calling the sanitization API for the first time had a relatively high cost (10 ms on average). This was introduced by the initialization process of the sanitizer. Overall, the experiment results suggest that developers can improve the security of their applications with a relatively low performance cost if they sanitize the vulnerable input.

F. Antipattern: Not-Using-Parameterized-Query

Definition: A parameterized query (also called a prepared statement) takes the form of a template that contains the logical purpose of the query. It leaves certain values unspecified, called parameters or placeholders. The actual values of those parameters are bound at runtime. This antipattern arises when a query could have been parameterized but it is not [48], [49], [23], [36].

Rationale: By parameterizing the queries, the DBMS can parse, compile, and perform query optimization on the SQL statement template. The processed template can be used for the future execution with input parameters. From a security perspective, parameterized queries are very useful against SQL injection because the parameter values are syntactically bound to the positions of string literals and are therefore not interpreted as commands. From a performance perspective, parameterized queries can reduce parsing time because the preparation on the query is done only once no matter how many times the query is executed. Therefore, not parameterizing the query would lose these benefits.

1) Techniques: TAPS [48], [49] is an automated technique for query parameterization. It analyzes the parsed structure of the SQL statements to identify data arguments for the parameterized query. It then traverses the program backwards to the program statements that generate these arguments, and substitutes the arguments with placeholders (i.e., the symbol "?").

2) Performance Impact:

a) Repair Strategy: To fix this antipattern, we parameterized the queries and reused the precompiled template as suggested by Karwin [36].

b) Result: As shown in Table II, the runtime and energy of the database operations were reduced 50% and 55% on average after fixing the antipattern. Statistical significance was established for most of the cases. These results suggest that using parameterized queries is not only a more secure way, but also a more resource-efficient way to interact with local databases.

G. Antipattern: Not-Caching

Definition: Multiple syntactically equivalent or partially equivalent queries are issued to retrieve data from the database [13], [50], [29], [51], [22]. These queries can be exactly the same or they share a common subexpression.

Rationale: Not caching query results often leads to redundant computation being performed. If the database contents have not been altered between the execution of a group of syntactically equivalent queries, caching their results can help to improve performance.

1) Techniques: CacheOptimizer [13] is a technique that helps developers optimize the configuration of caching frameworks for web applications that are implemented using Hibernate (a Java ORM framework). CacheOptimizer leverages existing web logs to discover the optimal cache configuration so that the caching framework of Hibernate can cache the query results properly.

Yan and colleagues [29] proposed a technique to detect the Not-Caching antipattern in web applications that are developed using ORM. ORM frameworks allow users to construct queries by chaining multiple function calls (e.g., where, join), with each chain translated into a SQL query. To detect the equivalent or partially equivalent queries, the proposed technique analyzes the query call chains using static analysis: if the same ORM query function is used in two different ORM query function chains, the corresponding queries will share a common expression.

Sqlcache [51] is a compiler optimization technique that automatically adds sound SQL caching to Web applications. Sqlcache computes conditions for irrelevance [52] between read queries and write queries. If relevant, Sqlcache instruments the application where the write query with cache invalidations automatically.

As mentioned previously in the Unbatched-Writes section, the technique by Tamayo and colleagues [12] analyzes the dependencies between multiple SQL statements. In addition to finding batching opportunities, the technique also looks for caching opportunities by checking if multiple queries always return the same result in their dependency graph.

2) Performance Impact:

a) Repair Strategy: Since the existing repair methods usually rely on adding a cache layer, of which software developers may not have control over, we experimented with a strategy that can fix this antipattern by code refactoring. We avoided issuing repeated queries by reusing the `Cursor` object [32]. This object is returned by the select query API and it stores the query results.

b) Result: The experiment results showed that fixing this antipattern can reduce the runtime and energy of the database operations by 7% and 6%, respectively. When we investigated into the savings, we found that the resource consumption of retrieving data reduced significantly, but the time required to iterate over the retrieved data (i.e., the cursor), remained unchanged. This is because caching the data helps to reduce the amount of data transferred from the database to the application, but does not help to reduce of amount of data that the application needs to iterate over. Therefore, the buffer refreshing overhead would still occur. Nevertheless, the absolute energy and runtime savings were still relatively high. Many of them established statistical significance as well, suggesting the benefits of fixing this antipattern.

H. Antipattern: Unnecessary-Column-Retrieval

Definition: Developers issue a query that reads more columns than needed [15], [22], [29], [36]. For instance, the query "*SELECT * FROM User*" retrieves all the columns, but in the application, a subset of the retrieved columns is never used.

Rationale: The more columns the SQL query fetches, the more data must travel between the application and the database. While the data transfer comes with a cost, retrieving columns that are not needed in subsequent computation can be harmful to performance and scalability.

1) Techniques: Chen and colleagues [15] proposed an approach that combines static and dynamic analysis to detect this Unnecessary-Column-Retrieval antipattern. The static analysis part of the approach identifies and instruments database-accessing functions. The dynamic analysis part obtains the code execution traces and the SQL queries. Then, the approach discovers instances of this antipattern by examining the data access mismatches between the needed columns and the requested columns.

Yan and colleagues [29] proposed a detector using only static analysis. The detector identifies the columns retrieved by each query and checks if there are subsequent uses of the retrieved columns. If there are no such uses, it means the Unnecessary-Column-Retrieval antipattern occurs in the application.

2) Performance Impact:

a) Repair Strategy: As suggested by Yan *et al.* [29], we modified the query so that it only retrieved the needed columns.

b) Result: The experiment results showed that after fixing the antipattern, the runtime and energy consumption of the database operations can be reduced 27% and 29% on average, respectively. The differences established statistical significance for most of the runtime results. We investigated into the cases that did not show a statistically significant impact. We found that the unnecessarily retrieved column had an integer type, whose size was small. As for the cases that had a significant impact, we found that fixing the antipattern can reduce the time of retrieving data since less data needed to be transferred. This demonstrates that even though the data does not need to travel over the network, there is still a significant cost for retrieving data from a local database. Moreover, we discovered that retrieving less columns could also save the resources consumed by iterating over the results. Although fixing the antipattern did not reduce the number of retrieved rows, it reduced the total amount of retrieved data as less columns were fetched. As a consequence, the memory pressure was alleviated and the buffer refreshing rate was decreased. These ultimately led to less resource consumption.

I. Antipattern: Unnecessary-Row-Retrieval

Definition: Developers issue a query that reads more rows than needed [23], [53]. When table rows are retrieved from the database, not all of them may be needed by the application. This antipattern arises when the retrieved rows are filtered by the application logic and only a subset of the rows are actually used by the application.

Rationale: Similar to the Unnecessary-Column-Retrieval antipattern, redundant row retrieval is a waste of resources

and can harm performance. The filtering conditions could have been specified in the selection clause of the query so as to reduce the amount of data transfer.

1) Techniques: Chaudhuri and colleagues [23] proposed a cost estimation tool to alleviate this antipattern. They first used dynamic profiling to compute the number of rows consumed by the application and the total number of rows returned by the query. If the query returns many rows (say N) and the application consumes only k rows ($k \ll N$), then it may be beneficial to pass a query hint to the database server requesting that the plan should be optimized for returning the top k rows quickly. Based on the profiling information, Chaudhuri developed a Fast-k analysis tool that can estimate how the cost of the query varies with k. Such information can be used by developers to decide if it is appropriate to rewrite their query to use the hint.

Dugan and colleagues [53] did not propose specific techniques for automated detection and optimization. They analyzed the antipattern using software performance engineering modeling techniques and compute several metrics, such as the number of disk I/O operations. Using the same methodology, they also examined several solutions to the antipattern using a lower bound and upper bound in the query to limit the number of retrieved rows.

DBridge [31] includes an optimization mechanism for unnecessary row retrieval. It identifies filters on query results expressed using conditional imperative constructs, such as if, and pushes them into the query as a selection.

2) Performance Impact:

a) Repair Strategy: As suggested by Emani *et al*. [31], we modified the query so that it only retrieved the needed rows.

b) Result: The experiment results showed that by fixing the antipattern, the runtime and energy can be reduced 65% and 70% on average, respectively. The statistical test established statistical significance for most of the runtime results. The ones that did not show a significant improvement were from an instance with statement form *SELECT DISTINCT*. In that instance, since only the distinct rows were retrieved, the antipattern version only retrieved two more rows than the fixed version, resulting in a negligible performance difference. When we looked in depth at the results that showed a significant difference, we found that fixing this antipattern can not only reduce the time needed to retrieve the data but also reduce the time needed to iterate over the retrieved result. This is because there was less data transferred and less rows needed to be iterated over. Similar to fixing the Unnecessary-Column-Retrieval antipattern, when the total amount of data to read became larger and exceeded the threshold, optimizing the Unnecessary-Row-Retrieval antipattern can significantly reduce the time needed to iterate over the result; because it can avoid or alleviate the buffer refreshing overhead.

J. Antipattern: Unbounded-Query

Definition: A query that may return an unbounded number of records is issued and there exists a subsequent computation over the returned records [29], [22], [34], [54].

Rationale: From a performance perspective, if the application directly renders the results of the query (whose size can be potentially very large), the responsiveness of the application can be affected because of the long rendering process [29]. From a security perspective, attackers can carry out a second-order denial-of-service attack by tainting the database table with a large number of records [34]. Then if an unbounded query is issued to retrieve data from the table, and the number of executions of a loop is controlled by the query result, CPU exhaustion may happen.

1) Techniques: The work by Yan and colleagues [29] examines if a query is bounded by checking if (1) the query always returns a single value (e.g., a COUNT query); (2) the query always returns a single record (e.g., retrieving using a unique identifier); or (3) the query uses a LIMIT keyword bounding the number of returned records.

Olivo and colleagues [34] verified if this antipattern occurs by statically checking if (1) a database attribute can be tainted by a user input, (2) a query uses the tainted database attribute in its selection clause, and (3) the number of executions of a loop is controlled by the tainted query result.

Panorama [54] suggests a variety of refactorings that can fix this antipattern, such as pagination, asynchronous loading, etc. It identifies opportunities for applying such refactoring in web applications, and automatically suggests patches.

2) Performance Impact:

a) Repair Strategy: For this antipattern, carrying out a repair can be challenging. This is because changing a query from unbounded to bounded can alter the semantics and even the display of the program. Depending on the functionality of the application, developers can choose to use the SQL keyword *LIMIT* to simply limit the number of retrieved rows. A sophisticated change at the design level, such as employing pagination [54], can also be applied. Although there can be a variety of repair strategies, there is one check that developers can always insert to the code before issuing the unbounded queries, which is to check the number of rows that are going to be retrieved and iterated over. If this number is under certain threshold, the original unbounded query can be issued anyway. Otherwise, a fixed version of the code would be executed. Since there does not exist a universal repair strategy for this antipattern, in our experiment, we measured the performance overhead of checking the number of retrieved rows.

b) Result: Our experiment results showed that the runtime and energy consumption were both increased 7% on average after adding the bound check. The absolute runtime and energy increase were 1.57 ms and 1.18 mJ, respectively. Comparing to the performance impact of other antipatterns, this overhead was relatively low. The low cost was due to the fact that the bound check did not require the application to load all the necessary rows of data from the database to the memory. Developers can check the number of retrieved rows using the `COUNT()` function in SQL, which returns a single number. By checking the number of retrieving rows in advance, developers can improve the security of their applications with a low

performance cost. Developers can choose to apply a fix to this antipattern on top of the bound check so as to further improve the performance and security.

K. Antipattern: Readable-Password

Definition: Sensitive information, such as a user password, is stored in plain text in the database [36]. This antipattern arises if an application stores the password by specifying the password as a string literal in an insert or update statement. When authenticating the password, this antipattern can also appear if the application compares the user's input to the password string stored in the database. For example, "*SELECT * FROM Accounts WHERE account_id = '123'AND password = 'opensesame'*".

Rationale: This programming practice results in security vulnerabilities because if attackers can read the SQL statement, they can see the plain text password. Hackers can steal a password by searching SQL statement logs, or reading data from database backup files.

1) Techniques: We have not identified any technique that targets this antipattern.

2) Performance Impact:

a) Repair Strategy: To fix this antipattern, we encrypted the sensitive information before storing it to the database as suggested by Karwin [36].

b) Result: The experiments showed that the runtime and energy were increased 0.5% and 0.1% on average after fixing the antipattern. The absolute runtime increased 0.84 ms and the energy consumption increased 0.1 mJ. These costs are relatively low given 100 ms the threshold for humans to perceive delay [25], [26]. These results mean that fixing this antipattern by encryption comes at a low performance cost.

V. Threats to Validity

A potential threat is that our benchmark study was based on SQLite, which could have different performance characteristics than other database management systems. However, SQLite is the default and dominant local database service in mobile devices [2]. Therefore, our experiment results are generalizable to most mobile apps.

We collected the energy and runtime measurement results from a single device, a Samsung Galaxy S5 running Android 5.0. Other devices and operating systems may have different runtime and energy characteristics. To mitigate this threat, we repeated the measurements on a Nexus 5 running Android 6.0.1 and a Google Pixel running Android 7.1.2. We observed that although the absolute values may differ from the results we obtained from the S5, the percentage differences were consistent.

We used an automated workload generator to generate the workload, which may not be representative of real user workload. However, this potential bias was not likely to undermine our conclusion as our experiment was based on a large number of applications and SQL statement executions. In addition, our benchmark suite only required an estimated range of values for the varied factors instead of an accurate number.

We evaluated the performance impact on a synthesized benchmark suite, which may not reflect the realistic impact of the SQL antipattern on real apps. We made several efforts to mitigate this threat. First, the code in the benchmark suite was replicated from SQL antipattern instances detected in real apps. Second, we chose representative values of the performance affecting factors to be used in the benchmark suite. For example, the selected table forms and statement forms represented 90% of all instances detected in the real apps.

We fixed the antipattern instances manually. If the repair was carried out incorrectly and altered the functionality of the original program, it may affect the validity of the performance results we obtained. To ensure that the code functionality remained consistent before and after the fix, we added additional checks to the benchmark code. If the detected SQL antipattern instance involved database reads, we checked if the read data that was used by the application remained consistent before and after the fix. If database writes were involved, we verified the corresponding database table and checked if the content and size remained the same. Note that these checks were not part of the measurements.

It is worth noting that this paper focuses on the performance impact of SQL antipatterns. The repair strategies we used may affect some other aspects of the software, such as readability and maintainability. Our study presents the performance cost of SQL antipatterns to developers and they can consider whether it is worth fixing the problem.

VI. Conclusions

In this paper, we presented a consolidated overview of the current knowledge about SQL antipatterns and demonstrated their performance impact on local database operations in mobile applications. We identified eleven SQL antipatterns through our literature review. We discovered that eight of them have a significant impact on the runtime and energy consumption of local database operations. Out of these eight SQL antipatterns, two of them (Unbatched-Writes and Loop-to-Join) are particularly impactful and can noticeably affect user experience. For the three security oriented SQL antipatterns, we discovered that the repair only introduced a minor performance cost. In terms of the detection and repair techniques, we found that there exist automated detection or repair approaches for ten out of the eleven SQL antipatterns. Overall, our study provides interesting insights that give concrete guidelines for app developers to improve their apps and motivate areas for future research work in the software engineering community. The benchmark suite and the raw measurement data is available from https://github.com/USC-SQL/SQLABenchmark.

VII. Acknowledgments

This work was supported, in part, by US NSF grant 1619455 and ONR grant 14-17-1-2896.

REFERENCES

[1] Google, "Android SQLite Documentation," https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html, 2017.

[2] Y. Lyu, J. Gui, M. Wan, and W. G. J. Halfond, "An empirical study of local database usage in android applications," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 444–455.

[3] D. Li, S. Hao, J. Gui, and W. G. Halfond, "An empirical study of the energy consumption of android applications," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, September 2014.

[4] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: an empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014.

[5] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What Do Mobile App Users Complain About?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, May 2015.

[6] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond, "Truth in advertising: The hidden cost of mobile ads for software developers," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, May 2015, to Appear.

[7] J. Gui, M. Nagappan, and W. G. Halfond, "What Aspects of Mobile Ads Do Users Care About? An Empirical Study of Mobile In-app Ad Reviews," *arXiv preprint arXiv:1702.07681*, 2017.

[8] J.-M. Kim and J.-S. Kim, "Androbench: Benchmarking the storage performance of android-based mobile devices," in *Frontiers in Computer Education*. Springer, 2012, pp. 667–674.

[9] O. Kennedy, J. Ajay, G. Challen, and L. Ziarek, "Pocket data: The need for tpc-mobile," in *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2015, pp. 8–25.

[10] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic, "Holistic query transformations for dynamic web applications," in *2009 IEEE 25th International Conference on Data Engineering*, March 2009, pp. 1175–1178.

[11] A. Cheung, S. Madden, and A. Solar-Lezama, "Sloth: Being lazy is a virtue (when issuing database queries)," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 931–942. [Online]. Available: http://doi.acm.org/10.1145/2588555.2593672

[12] J. M. Tamayo, A. Aiken, N. Bronson, and M. Sagiv, "Understanding the behavior of database operations under program control," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 983–996. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384688

[13] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 666–677.

[14] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1001–1012. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568259

[15] T. H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Transactions on Software Engineering*, vol. 42, no. 12, pp. 1148–1161, Dec 2016.

[16] B. Kitchenham, "Procedures for performing systematic reviews," vol. 33, 08 2004.

[17] J. Feinstein, "Squeezing Performance from SQLite: Insertions," https://medium.com/@JasonWyatt/squeezing-performance-from-sqlite-insertions-971aff98eef2, April 2017.

[18] "https://play.google.com/store/apps."

[19] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large scale dynamic analysis of mobile apps," in *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2014.

[20] Monsoon Solutions, Inc, "Monsoon Power Monitor," http://www.msoon.com/LabEquipment/PowerMonitor, 2017.

[21] S. Horn, *Guide to Standard Errors for Cross Section Estimates*. Melbourne Institute of Applied Economic and Social Research, 2004.

[22] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "How not to structure your database-backed web applications: a study of performance bugs in the wild," 2018.

[23] S. Chaudhuri, V. Narasayya, and M. Syamala, "Bridging the application and dbms profiling divide for database application developers," in *VLDB*. Very Large Data Bases Endowment Inc., September 2007.

[24] Y. Lyu, D. Li, and W. G. J. Halfond, "Remove rats from your code: Automated optimization of resource inefficient database writes for mobile applications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 310–321. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213865

[25] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ser. AFIPS '68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 267–277. [Online]. Available: http://doi.acm.org/10.1145/1476589.1476628

[26] B. A. Myers, "The importance of percent-done progress indicators for computer-human interfaces," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '85. New York, NY, USA: ACM, 1985, pp. 11–17. [Online]. Available: http://doi.acm.org/10.1145/317456.317459

[27] N. Arzamasova, M. Schler, and K. Bhm, "Cleaning antipatterns in an sql query log," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 3, pp. 421–434, March 2018.

[28] A. Cheung, A. Solar-Lezama, and S. Madden, "Optimizing database-backed applications with query synthesis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462180

[29] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding database performance inefficiencies in real-world web applications," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17. New York, NY, USA: ACM, 2017, pp. 1299–1308. [Online]. Available: http://doi.acm.org/10.1145/3132847.3132954

[30] W. Kim, "On optimizing an sql-like nested query," *ACM Trans. Database Syst.*, vol. 7, no. 3, pp. 443–469, Sep. 1982. [Online]. Available: http://doi.acm.org/10.1145/319732.319745

[31] K. V. Emani, T. Deshpande, K. Ramachandra, and S. Sudarshan, "Dbridge: Translating imperative code to sql," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 1663–1666. [Online]. Available: http://doi.acm.org/10.1145/3035918.3058747

[32] A. Developer, "Android Cursor Documentation," https://developer.android.com/reference/android/database/Cursor, March 2019.

[33] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "Candid: Preventing sql injection attacks using dynamic candidate evaluations," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 12–24. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315249

[34] O. Olivo, I. Dillig, and C. Lin, "Detecting and exploiting second order denial-of-service vulnerabilities in web applications," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 616–628. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813680

[35] S. Son, K. S. McKinley, and V. Shmatikov, "Diglossia: detecting code injection attacks with precision and efficiency," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1181–1192. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516696

[36] B. Karwin, *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*, 1st ed. Pragmatic Bookshelf, 2010.

[37] C. Gould, Z. Su, and P. Devanbu, "Jdbc checker: A static analysis tool for sql/jdbc applications," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 697–698. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999476

[38] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, Sep. 2007. [Online]. Available: http://doi.acm.org/10.1145/1276933.1276935

[39] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 372–382. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111070

[40] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent sql injection attacks," in *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, ser. SEM '05. New York, NY, USA: ACM, 2005, pp. 106–113. [Online]. Available: http://doi.acm.org/10.1145/1108473.1108496

[41] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 40–52. [Online]. Available: http://doi.acm.org/10.1145/988672.988679

[42] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251416

[43] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *Proceedings of the 21st Annual Computer Security Applications Conference*, ser. ACSAC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 303–311. [Online]. Available: https://doi.org/10.1109/CSAC.2005.21

[44] W. G. Halfond, J. Viegas, A. Orso *et al.*, "A classification of sql-injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, vol. 1. IEEE, 2006, pp. 13–15.

[45] D. A. Kindy and A. K. Pathan, "A survey on sql injection: Vulnerabilities, attacks, and prevention techniques," in *2011 IEEE 15th International Symposium on Consumer Electronics (ISCE)*, June 2011, pp. 468–471.

[46] R. Johari and P. Sharma, "A survey on web application vulnerabilities (sqlia, xss) exploitation and security engine for sql injection," in *2012 International Conference on Communication Systems and Network Technologies*, May 2012, pp. 453–458.

[47] OWASP, "Category:OWASP Enterprise Security API," https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API, August 2018.

[48] P. Bisht, A. P. Sistla, and V. N. Venkatakrishnan, "Taps: Automatically preparing safe sql queries," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 645–647. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866384

[49] P. Bisht, A. P. Sistla, and V. Venkatakrishnan, "Automatically preparing safe sql queries," in *International Conference on Financial Cryptography and Data Security*. Springer, 2010, pp. 272–288.

[50] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner, "Efficient exploitation of similar subexpressions for query processing," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 533–544. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247540

[51] Z. Scully and A. Chlipala, "A program optimization for automatic database result caching," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 271–284, 2017.

[52] J. A. Blakeley, N. Coburn, and P.-V. Larson, "Updating derived relations: Detecting irrelevant and autonomously computable updates," *ACM Trans. Database Syst.*, vol. 14, no. 3, pp. 369–400, Sep. 1989. [Online]. Available: http://doi.acm.org/10.1145/68012.68015

[53] R. F. Dugan, Jr., E. P. Glinert, and A. Shokoufandeh, "The sisyphus database retrieval software performance antipattern," in *Proceedings of the 3rd International Workshop on Software and Performance*, ser. WOSP '02. New York, NY, USA: ACM, 2002, pp. 10–16. [Online]. Available: http://doi.acm.org/10.1145/584369.584372

[54] J. Yang, C. Yan, C. Wan, S. Lu, and A. Cheung, "View-centric performance optimization for database-backed web applications," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 994–1004. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00104