Database-Access Performance Antipatterns in Database-Backed Web Applications

Shudi Shao*, Zhengyi Qiu*, Xiao Yu*, Wei Yang‡, Guoliang Jin*, Tao Xie§, Xintao Wu¶
* North Carolina State University, {sshao, zqiu2, xyu10, guoliang_jin}@ncsu.edu
‡ University of Texas at Dallas, wei.yang@utdallas.edu
§ Peking University, taoxie@pku.edu.cn
¶ University of Arkansas, xintaowu@uark.edu

***Abstract*—Database-backed web applications are prone to performance bugs related to database accesses. While much work has been conducted on database-access antipatterns with some recent work focusing on performance impact, there still lacks a comprehensive view of database-access performance antipatterns in database-backed web applications. To date, no existing work systematically reports known antipatterns in the literature, and no existing work has studied database-access performance bugs in major types of web applications that access databases differently.**

To address this issue, we first summarize all known database-access performance antipatterns found through our literature survey, and we report all of them in this paper. We further collect database-access performance bugs from web applications that access databases through language-provided SQL interfaces, which have been largely ignored by recent work, to check how extensively the known antipatterns can cover these bugs. For bugs not covered by the known antipatterns, we extract new database-access performance antipatterns based on real-world performance bugs from such web applications. Our study in total reports 24 known and 10 new database-access performance antipatterns. Our results can guide future work to develop effective tool support for different types of web applications.

***Index Terms*—performance antipatterns, performance bugs, database-backed web applications, characteristic study**

I. Introduction

Databases are an integral component for server-side web applications to store and process data produced or consumed by end users. Performance of these database-backed web applications is critical, as studies have shown that around half of the users expect to load a page from the server within two seconds [11] and will leave a page that takes longer than three seconds to load [12], directly affecting user experience and business profits. Sometimes, performance bugs may affect millions of users. For example, Healthcare.gov encountered significant performance issues after its initial deployment, and it had to be upgraded to process a high volume of user requests that were not anticipated [39].

Despite high criticality of performance bugs related to database accesses, it is challenging to develop database-backed web applications that interact with backend databases efficiently. As critical performance information and constraints of database accesses are hidden behind the database-access interface, developers do not necessarily have good knowledge on the performance implication of various database accesses and the interaction between database accesses and application code at development time. Therefore, database-access-related performance bugs in database-backed web applications are challenging to be avoided or identified.

To handle performance bugs especially those related to database accesses, researchers have conducted empirical studies to understand their common characteristics or abstract their belonging antipatterns. Some studies focus on performance bugs in general server and client applications [28], [37], Android applications [31], and client-side JavaScript programs [44]. Since these performance bugs being studied are not specific to server-side database-backed web applications, the gained insights cannot be directly generalized to web-application performance bugs related to database accesses. Specific to database-backed web applications, while database-access antipatterns have been studied for a long period of time, it is only recently that previous work [17], [32] started to focus specifically on antipatterns with performance impact that can lead to performance bugs.

However, the previous work still fails to provide a comprehensive understanding of database-access performance antipatterns in web applications.

- To summarize known database-access performance antipatterns, researchers [17], [32] conducted literature surveys, but they did not comprehensively cover or report antipatterns that exist in the literature. Chen et al. [17] surveyed five papers [18], [20], [21], [48], [50] and reported 17 antipatterns that can be used for performance-aware refactoring. While the five surveyed papers all have a focus on performance issues related to database accesses in web applications, Chen et al. did not include many more related papers and books that exist in the literature. Lyu et al. [32] surveyed 56 research papers and 1 book with database-access antipatterns, not all of which are related to performance, and they reported 8 performance antipatterns and 3 security antipatterns. While Lyu et al. surveyed more papers and books, they unfortunately did not report all performance antipatterns from the surveyed literature. Their paper does not describe the reason why some antipatterns were excluded. One guess could be that their focus was on how mobile applications use local databases and they excluded performance antipatterns that were not found under this context.
- To extract new database-access performance antipatterns,

researchers studied real-world performance bugs in bug-tracking systems [50] and/or performance issues found by static analysis or dynamic profiling tools [17], [48], [50], but they did not cover comprehensive types of database-backed web applications. Recent characteristic studies focused on performance bugs in web applications developed using different Object Relational Mapping (ORM) frameworks, including ActiveRecord for Ruby on Rails [48], [50] and Eloquent for PHP [17]. However, there are still a lot of web applications that access databases by directly constructing queries with language-provided SQL interfaces in application code but not through ORM frameworks, and we refer to these applications as *direct-accessing web applications*. Notably, among the top 15 most popular websites [6], *MediaWiki* and *WordPress* are open sourced, and neither of them uses an ORM framework. It is not clear whether known performance antipatterns in the literature can cover comprehensively performance bugs in direct-accessing web applications.

To complement the current state of the art, in this paper, we first conduct a literature review to summarize the current knowledge about database-access antipatterns that can lead to performance bugs, and our goal is to cover and report known antipatterns in the literature as comprehensively as possible. After gathering known antipatterns, we conduct a characteristic study of performance bugs in direct-accessing web applications to investigate whether there are new database-access performance antipatterns not covered by the existing research literature.

Specifically, we address the following three major research questions in this paper:

- **RQ1:** What are the known database-access performance antipatterns in the research literature?
- **RQ2:** How extensively can known antipatterns cover the root causes of performance bugs in direct-accessing web applications?
- **RQ3:** Are there any unknown database-access performance antipatterns based on performance bugs in direct-accessing web applications?

To answer **RQ1**, we follow the literature-survey methodology laid out by Lyu et al. [32]. Since the list of publications studied by Lyu et al. is not available in their paper or on their GitHub site, where the benchmark suite and raw measurement data are available at the time of checking, we repeat the literature search process. In total, from our surveyed research publications, we identify 24 performance antipatterns, which are substantially more than those identified via the previous two literature surveys [17], [32]. By listing all the studied papers and reporting all the known antipatterns, future research does not have to repeat the same literature-survey process again and can reuse our results as the starting point.

For **RQ2** and **RQ3**, we study 140 database-access related performance bugs collected from the bug-tracking systems of seven direct-accessing web applications, i.e., *BugZilla*, *DNN*, *Joomla!*, *MediaWiki*, *WordPress*, *Moodle*, and *Odoo*. We make our subject collection based on popularity of applications, availability of bug-tracking systems, and sufficiency of information for bug understanding. Our selection covers four different popular web-application languages, i.e., PHP, C#, Python, and Perl.

To answer **RQ2**, we use the 24 known performance antipatterns identified while answering **RQ1** to match with the root causes of our collected performance bugs. Among the 140 collected performance bugs, 107 of them match with known performance antipatterns.

To answer **RQ3**, we further extract 10 new performance antipatterns from the 33 performance bugs identified while answering **RQ2**, which cannot be covered by known performance antipatterns. Among these 10 new antipatterns, only 1 antipattern is related to query construction and thus specific to direct-accessing web applications, and the other 9 are all applicable to database-backed web applications in general. Our results confirm the necessity of studying direct-accessing web applications to provide a more comprehensive coverage on real-world database-access performance antipatterns.

Overall, our study not only summarizes existing knowledge on performance antipatterns related to database accesses in web applications, but also complements existing studies by focusing on performance bugs in direct-accessing web applications. Our new findings include new database-access performance antipatterns that are general to all database-backed web applications, and our results also reveal unique characteristics and challenges that are specific to direct-accessing database-backed web applications.

II. Methodology

In this section, we describe our methodology on (1) how we survey the existing literature for summarizing known database-access performance antipatterns to answer **RQ1** and (2) how we collect and analyze performance-bug reports from popular direct-accessing database-backed web applications to answer **RQ2** and **RQ3**.

A. Literature Search Protocol

We follow the literature search protocol (as described by Lyu et al. [32]) developed based on Kitchenham's systematic review guidelines [30]. Specifically, we start with 8 papers [14], [17], [18], [20], [21], [32], [48], [50] as our initial set and conduct forward/backward citation search with the initial set. During the process, we include papers that discover or define at least one antipattern related to database accesses and add them to our set. We stop when the forward/backward citation search results in only papers that are already included.

In total, we obtain 47 papers and 1 book, while Lyu et al. obtained 56 papers and 1 book [32]. As we include selected papers published in only recent high-profile software-engineering or database conferences as our initial set of papers, while Lyu et al. did keyword search to get their initial set from not only software-engineering and database conferences but also security conferences, our collection could have fewer papers focusing on security antipatterns, and thus

Table I. Subjects and bugs in our study

Subject	Abbreviation	Stars	Commits	Contributors	Language	Number of Bugs
BugZilla	BZ	323	9483	76	Perl	51
DNN	DN	712	15332	136	C#	26
Joomla!	JM	1969	96463	538	PHP	15
MediaWiki	MW	3514	32219	600	PHP	7
Moodle	MD	3011	97654	500	PHP	24
WordPress	WP	13866	41440	60	PHP	12
Odoo	OD	17524	133567	1041	Python	5
Sum						**140**

the smaller paper number. We do not communicate with Lyu et al. to get their publication list for comparison. However, given our focus on performance antipatterns, the exclusion of security-antipattern papers should have little to none impact on the number of publications with performance antipatterns collected from the literature; the impact is partially reflected by the small difference on paper numbers.

From the 47 papers and 1 book, 27 papers and the 1 book contain at least one performance antipattern. They are used to summarize known database-access performance antipatterns in the literature, and we refer to all of them as appropriate when presenting known antipatterns.

B. Bug Collection and Bug-Report Examination

Based on application popularity and bug-tracking system availability, we select seven open-source direct-accessing web applications to answer **RQ2** and **RQ3**: *BugZilla* [1], *DNN* [2], *Joomla!* [3], *MediaWiki* [8], *Moodle* [4], *WordPress* [9], and *Odoo* [5]. *BugZilla* is a bug-management system heavily used by Mozilla and other open-source projects. *DNN* is a popular content-management system, and its customers include large companies such as Bank of America, Canon, and BP. *Joomla!* and *MediaWiki* are also popular content-management systems, and they are used as subjects in a related study on database applications [40]. *Moodle* is a learning management system, which is widely used in schools, including North Carolina State University where several authors of this paper are affiliated with. *WordPress* is currently the most dominant content-management system on the market [7]. *Odoo* is a popular all-in-one business application. Table I shows the numbers of stars, commits, and contributors on GitHub. Our studied web applications are implemented using four different programming languages, including Perl, C#, PHP, and Python, and they all directly construct queries in application code without using ORM frameworks.

To collect performance bugs related to database accesses, we first search the bug-tracking systems of the selected web applications with keywords "performance," "timeout," and "slow" in order to retrieve performance bugs. After obtaining an initial set of performance bugs, we filter out bug reports that are not relevant to database accesses. Specifically, we keep only the reports whose description and comments contain database-related keywords, e.g., "database," "query," and "schema." Further, we include only bugs that have been closed with fixes. Following this process, our final performance-bug set contains 140 bug reports. In Table I, the last column shows the number of bug reports that we collect for each application. In comparison, a previous study [50] focusing on performance antipatterns in Ruby-on-Rails web applications studied 140 performance bugs reported in the bug-tracking systems of 12 applications and 64 performance issues identified through profiling. We focus on only performance bugs collected from bug-tracking systems, and our number of real-world performance bugs under study is comparable.

A bug report typically contains some bug description, followed by some discussions and comments on possible causes and fixes, and some intermediate fixes and the final committed fix. To answer **RQ2** and **RQ3**, our study centers around these preceding parts to understand the root causes and fix strategies of our collected bugs.

Every bug report is manually inspected and discussed by at least three authors to ensure the objectivity of our conclusions. We determine the root cause of each bug by examining each bug report to understand what particular reasons in program code, schemas, or database behaviors cause the performance bugs, and we determine the fix strategy of each bug by reviewing the patch submitter's description of the fix and inspecting the code in the patch to look for changes in the program code, queries, or schema.

For performance bugs whose root cause and fix strategy match known performance antipatterns, we label these performance bugs accordingly. For those without any matched known antipattern, we come up with new performance antipatterns to describe the root causes and fix strategies.

III. **RQ1:** Performance Antipatterns in the Literature

As discussed in Section II-A, we identify 27 research papers and 1 book that discover or discuss at least one database-access performance antipattern through our literature search process. From these publications, we summarize 24 database-access performance antipatterns. Table II lists all the 24 performance antipatterns, and describes the root cause and fix strategy of each antipattern. All the 27 research papers and 1 book are cited under the "Origin" column when appropriate. Due to space limit, we do not provide examples for these antipatterns, and the readers can refer to the papers under the "Origin" column if more detailed explanations and examples are needed.

Table II. Known database-access performance antipatterns summarized from the literature

ID	Name	Root cause	Fix strategy	Origins
AP-01	Inefficient queries	Issuing queries where semantically equivalent but more performant alternatives exist.	Using the more performant alternatives.	[14], [17], [29], [35], [50]
AP-02	Moving computation to the DBMS	Computing with the results of multiple queries, where the computation can also be done by the DBMS, and the network round-trip cost is larger than the query processing cost in the DBMS.	Moving the computation to the DBMS to save the network round-trip cost.	[17], [32], [48], [50]
AP-03	Moving computation to the server	Computing some results by the DBMS, which unfortunately is less performant compared with computing by the server, despite the increase in round-trip cost.	Moving the computation to the server despite extra round-trip cost.	[15], [50]
AP-04	Loop-invariant queries	Queries issued repeatedly in a loop always load the same database contents and hence are unnecessary.	Moving the query out of the loop and storing the queried results to intermediate objects.	[17], [50]
AP-05	Dead-store queries	The results of multiple queries are loaded to the same object, but the object is not used between some of these reloads.	Removing queries whose results are not used.	[17], [50]
AP-06	Queries with known results	Issuing queries whose results can be determined by examining the queries and program contexts without actually being executed.	Replacing the queries with the known results.	[17], [50]
AP-07	Redundant condition check	Queries issued inside condition checks and branches are identical and return the same results.	Storing the queried results to intermediate objects and using them in both the condition checks and branches.	[17]
AP-08	Not caching	Issuing multiple queries that are syntactically equivalent or of the same template without caching the query results.	Adding caching either using a new cache layer or storing the query results in static objects.	[19], [32], [43], [48], [53]
AP-09	Inefficient lazy loading	Issuing one query to retrieve N objects from one table, and N other queries to retrieve information related to the N objects from another table.	Issuing one query with a join clause of the two tables.	[17], [24], [25], [32], [34], [50]
AP-10	Not merging selection predicates	Issuing multiple `SELECT` queries where each loads only a subset of the needed rows.	Loading all needed rows in one query.	[13], [32], [34], [41]
AP-11	Not merging projection predicates	Issuing multiple `SELECT` queries where each loads only a subset of the needed columns.	Loading all needed columns in one query.	[13], [17], [32], [34]
AP-12	Inefficient eager loading	Eagerly loading associated objects that are too large.	Delaying the loading of the associated objects.	[17], [23], [50]
AP-13	Inefficient updating	Issuing N separate queries to update N database records.	Batching the N update queries into a single query.	[10], [17], [32], [33], [46], [50]
AP-14	Unnecessary column retrieval	Retrieving more columns than needed.	Retrieving only the columns that are needed.	[16], [17], [29], [32], [48], [50]
AP-15	Unnecessary row retrieval	Retrieving more rows than needed.	Only retrieving the rows that are needed.	[14], [16], [27], [29], [32]
AP-16	Unnecessary whole queries	The results of certain queries are completely unused.	Removing the queries.	[10], [18], [20], [21]
AP-17	Inefficient rendering	When a view file renders a set of objects, inefficient APIs are used.	Using more performant APIs for view rendering.	[50]
AP-18	Missing fields	Fields that are costly to be derived from other fields are not stored directly in database tables.	Storing the fields in database tables directly.	[50]
AP-19	Missing indexes	Appropriate indexes are not included in table schema.	Adding the necessary indexes.	[29], [35], [45], [50]
AP-20	Table denormalization	Issuing queries with fixed join predicates.	Storing the pre-joined, i.e., denormalized, table based on the fixed join predicates in the DBMS.	[48]
AP-21	Partial evaluation of projections	Issuing queries that mostly use a subset of stored fields in a table and the mostly unused fields are much larger in data size.	Partitioning the table column-wise into a table for frequently queried small fields and another for less queried large fields in the DBMS.	[48]
AP-22	Partial evaluation of selections	Issuing queries whose selection predicates contain constant values.	Storing table rows matching the predicates with constant values in the DBMS as a separate table.	[48]
AP-23	Unbounded queries	Queries returning an unbounded number of records to be displayed.	Pagination, i.e., splitting and displaying records on different pages.	[17], [32], [49], [48], [51], [52]
AP-24	Functionality trade-offs	Developers introducing new functionalities that are too costly.	Removing the costly new functionalities.	[49]–[51], [52]

Although the 24 database-access performance antipatterns that we summarize are collectively covered by four papers in our initial paper set via the literature search [17], [32], [48], [50], none of them individually covers all of the 24 antipatterns. We also have to resolve some differences on categorization and reporting strategies, antipattern naming, and antipattern understanding to come up with the listed database-access performance antipatterns. In the remainder of this section, we discuss these issues related to the four papers that can collectively cover all the 24 antipatterns.

A. Categorization and Reporting Strategies

Yang et al. [50] categorized the root causes of performance antipatterns into several high-level categories; this categoriza-

tion is followed by Chen et al. [17]. In our 24 performance antipatterns, [AP-01] to [AP-03] can be categorized as inefficient computation, [AP-04] to [AP-08] can be categorized as unnecessary computation, [AP-09] to [AP-13] can be categorized as inefficient data accessing, [AP-14] to [AP-16] can be categorized as unnecessary data retrieval, [AP-17] is on its own, [AP-18] to [AP-22] can be categorized as database-design problems, and [AP-23] and [AP-24] can be categorized as application-design trade-offs. Under the context of ORM-based web applications, Yang et al. [50] considered [AP-01] to [AP-17] all as ORM API misuses at a higher level, but this categorization does not apply to direct-accessing applications, as they do not simply use APIs to access databases but have to construct queries in the applications.

Regarding [AP-01] to [AP-03], which are all categorized into the same high-level category, inefficient computation, while Chen et al. [17] reported this high-level category, they did not report the three performance antipatterns. Instead, they reported seven specific performance rules that are related to ORM APIs. Since these rules are not applicable to direct-accessing web applications, we report the three general performance antipatterns. For performance rules that are specific to languages or frameworks, we consider them as special cases of general performance antipatterns instantiated on specific applications, languages, or frameworks. Except the study by Chen et al. [17], the same strategy is followed by all the other three studies [32], [48], [50] among the four that collectively cover all the 24 antipatterns.

On the other hand, we also acknowledge that there is value in summarizing specific performance rules. The reason is that [AP-01] (inefficient queries) is very broad. For ORM-based web applications, many different ORM APIs can be misused and result in performance inefficiencies, and [AP-01] indeed can be further specialized based on different ORM-API misuses. Since we are interested in summarizing performance antipatterns that are applicable to both ORM-based web applications and direct-accessing web applications, we include only the general antipatterns.

B. Resolving Naming and Understanding Differences

The surveyed publications could sometimes use different names for the same database-access performance antipattern, and we choose the most intuitive name if this case occurs. Otherwise, we inherit the names without changing them. To this end, most of our antipattern names are inherited from the four study papers in our initial search set.

Specifically, from the study by Yang et al. [50], we inherit the names of [AP-01] to [AP-06], [AP-09], [AP-12], [AP-13], [AP-17] to [AP-19], and [AP-21]; from the study by Chen et al. [17], we inherit the names of [AP-07]; from the study by Lyu et al. [32], we inherit the names of [AP-08], [AP-10], [AP-11], [AP-14], [AP-15], [AP-20]; and from the study by Yan et al. [48], we inherit the names of [AP-22] to [AP-24].

Some database-access performance antipatterns have other names that are equally good as the ones that we choose and are worth mentioning. Specifically, [AP-09] (inefficient lazy loading) is also known as loop to join [32] or N+1 problems; [AP-13] (inefficient updating) is also known as unbatched writes [32]; and [AP-23] (unbounded queries) is also known as content display trade-offs [50].

We introduce the name of antipattern [AP-16] for cases that belong to unnecessary data retrieval but are not [AP-14] (unnecessary column retrieval) or [AP-15] (unnecessary row retrieval). While the study by Lyu et al. [32] included names for [AP-14] and [AP-15], it does not include an appropriate name for [AP-16].

Chen et al. [17] reported a new antipattern, mid-result misuse, which happens when there are multiple queries on the same object, and each query retrieves different columns. In our case, we consider it as a special case of [AP-11] not merging projection predicates. Since Chen et al. did not include the papers that originally discussed [AP-11] in their study, they reported mid-result misuse as a new antipattern.

Table III. Numbers of performance bugs matching the 24 known antipatterns in applications selected by us and Yang et al. [50]

ID	BZ	DN	JM	MW	WP	MD	OD	**Sum**	[50]
AP-01	2	1	2	0	3	1	0	9	12
AP-02	2	2	1	0	0	0	1	6	4
AP-03	4	1	0	0	1	1	0	7	2
AP-04	0	0	0	0	0	1	0	1	5
AP-05	0	0	0	0	0	0	0	0	5
AP-06	0	1	1	0	0	0	0	2	7
AP-07	0	0	0	0	0	0	0	0	0
AP-08	5	2	1	0	0	2	0	10	0
AP-09	0	0	0	0	0	0	0	0	27
AP-10	16	0	0	0	0	0	0	16	0
AP-11	0	0	0	0	0	0	0	0	0
AP-12	0	0	0	0	0	0	0	0	1
AP-13	1	0	1	0	0	1	0	3	1
AP-14	3	0	1	0	0	0	0	4	4
AP-15	2	2	0	1	1	1	2	9	1
AP-16	3	0	1	0	3	2	0	9	3
AP-17	0	0	0	0	0	0	0	0	0
AP-18	1	2	0	0	0	0	0	3	5
AP-19	3	13	1	0	1	6	0	24	30
AP-20	0	0	0	0	0	0	0	0	0
AP-21	1	0	0	0	0	0	0	1	0
AP-22	0	0	0	0	0	0	0	0	0
AP-23	2	0	0	1	0	0	0	3	14
AP-24	0	0	0	0	0	0	0	0	19
Sum	45	24	9	2	9	15	3	107	140

IV. **RQ2:** Coverage of Known Antipatterns on Database-Access Performance Bugs from Direct-Accessing Web Applications

Following the methodology described in Section II-B, we collect 140 database-access performance bugs from seven direct-accessing web applications. After studying their root causes and fix strategies, we can match 107 of them with the 24 known database-access performance antipatterns described in Section III.

Table III shows the numbers of database-access performance bugs matching known antipatterns in each of our selected applications. We also show the total numbers of performance bugs matching each known antipattern from the previous study

Table IV. New antipatterns found in our studied performance bugs from direct-accessing web applications

ID	Name	BZ	DN	JM	MW	WP	MD	OD	Sum
AP-25	Existing indexes not leveraged	1	0	3	0	2	3	0	9
AP-26	Non-optimal force index	0	0	0	3	0	0	0	3
AP-27	Changing subqueries to join operations	0	1	0	0	0	4	1	6
AP-28	Changing join operations to subqueries	1	1	1	1	0	1	0	5
AP-29	Joining unused tables	1	0	1	0	0	1	1	4
AP-30	Unnecessary locks	1	0	0	1	0	0	0	2
AP-31	Subquery returning duplicated rows	1	0	0	0	0	0	0	1
AP-32	Conditions containing subsuming clauses	0	0	0	0	1	0	0	1
AP-33	Unnecessary `where` clause when all conditions are selected	1	0	0	0	0	0	0	1
AP-34	Unnecessary query construction	0	0	1	0	0	0	0	1
Sum		6	2	6	5	3	9	2	33

focusing on ORM-based web applications using Ruby-on-Rails [50]. Since we focus on performance bugs collected from bug-tracking systems, we exclude the problematic actions that they identified through profiling. Thus, although antipattern [AP-17] was reported by Yang et al. [50], the corresponding number in the last column of Table III is 0, as this antipattern does not appear in performance bugs collected from bug-tracking systems. Other antipatterns with their corresponding numbers as 0 in the last column were not reported by Yang et al. [50]. Antipatterns [AP-14] to [AP-16] were reported as a single antipattern, unnecessary data retrieval, by Yang et al. [50], and we further categorize their studied bugs falling into unnecessary data retrieval with a finer granularity.

In total, 107 of our studied bugs can be categorized with 15 out of the 24 known antipatterns, while performance bugs studied by Yang et al. [50] can be categorized with 16 out of the 24 known antipatterns. In this sense, these two studies are of similar representative on covering known database-access performance antipatterns. However, 33 out of our studied bugs cannot be categorized into known antipatterns, revealing that there are new antipatterns not covered by the current research literature. This result shows that studying performance bugs in direct-accessing database-backed web applications indeed can improve the coverage of real-world database-access performance antipatterns.

Both studies cover [AP-01] to [AP-04], [AP-06], [AP-13] to [AP-16], [AP-18], [AP-19], and [AP-23]. The number of studied performance bugs matching [AP-19] (missing indexes) is the highest in both studies. This result shows that it is challenging for developers to pick the optimal indexes at the database design phase regardless of whether the web applications are direct-accessing or ORM-based. Among the 12 antipatterns covered by both studies, the number differences on [AP-15], [AP-16], and [AP-23] are larger than five. [AP-14] to [AP-16] are the three antipatterns falling into the same high-level category, unnecessary data retrieval, and our study has 14 more bugs matching [AP-14] to [AP-16] combined. On [AP-23] (unbounded queries), the study by Yang et al. [50] has 11 more bugs than ours. We find number differences on the other nine shared antipatterns not significant.

Seven antipatterns appear in the study by Yang et al. [50] or in our study but not both. Specifically, antipatterns appear in only our studied bugs are [AP-08] (not caching), [AP-10] (not merging project predicates), and [AP-21] (partial evaluation of projections), while antipatterns appear in only their studied bugs are [AP-05] (dead-store queries), [AP-09] (inefficient lazy loading), [AP-12] (inefficient eager loading), and [AP-24] (functionality trade-offs). Other than [AP-12] and [AP-21] where the number difference is one in both cases, the number differences for the other antipatterns are at least five. [AP-09] and [AP-10] are related, where both have N queries that can be merged into a single query, while [AP-09] further merges the one query, which returns N objects and leads to N queries, with the N queries into a single query. For the differences on [AP-05], [AP-08], and [AP-24], we find it difficult to come up with definite explanations.

Both studies do not have bugs matching five database-access performance antipatterns: [AP-07] (redundant condition check), [AP-11] (not merging projection predicates), [AP-17] (inefficient rendering), [AP-20] (table denormalization), and [AP-22] (partial evaluation of selection). Among these five antipatterns, the first two are generally applicable to both direct-accessing web applications and ORM-based web applications, although both studies do not have matching bugs; the third one was reported by Yang et al. [50] in problematic actions identified through profiling; and the last two are less likely to match performance bugs from bug-tracking systems than the first three as the last two involve high-level database-design changes.

V. New Performance Antipatterns

From the 33 bugs that do not match with known antipatterns found in the literature, we derive 10 new database-access performance antipatterns. Table IV shows the overall results. Below, we first describe each new database-access performance antipattern in detail. For each antipattern, we first describe the root cause and fix strategy and then present real-world performance-bug examples. After that, we conclude this section with a discussion.

A. Details of the New Performance Antipatterns

[AP-25] Existing indexes not leveraged

Root cause: Due to unawareness of existing indexes or mismatches among queries and table schema definitions, the queries fail to leverage existing indexes.

```
# Before fix
query = "SELECT comment_date_gmt FROM comments WHERE ...
     ORDER BY comment_date DESC LIMIT 1;"

# After fix
query = "SELECT comment_date_gmt FROM comments WHERE ...
     ORDER BY comment_date_gmt DESC LIMIT 1;"

# Affected table schema
CREATE TABLE comment(
  ...
  (no KEY on comment_date)
  KEY comment_date_gmt (comment_date_gmt)
)
```

Fig. 1. *WordPress* bug #4366. Before and after code snippets are shown.

```
# The affected query
query = "SELECT ... FROM '#__content' AS a LEFT JOIN ...
     LEFT JOIN '#__associations' AS asso ON asso.id =
     a.id ..."

- CREATE TABLE IF NOT EXISTS '#__associations'
-      ('id' varchar(50) NOT NULL, ...)
+ CREATE TABLE IF NOT EXISTS '#__associations'
+      ('id' INT(11) NOT NULL, ...)
```

Fig. 2. *Joomla!* bug #29845. Patch diff results are shown.

Fix Strategy: Change the queries or table schema definitions so that existing indexes can be taken advantage of.

Examples: Figure 1 shows *WordPress* #4366, where the buggy code orders the results by the unindexed `comment_date`, and the fix is to order the results by `commend_data_gmt` that is already indexed.

Figure 2 shows *Joomla!* #29845, where the problem is more subtle. The problem is mismatched column types between two tables: the join operation joins the `id` column in table `#__associations` with a column in another table, where the type of column `id` is string, but the type of the column in the other table is integer. When the database engine performs the join operation, it has to do an extra type conversion on the two columns with different data types. After casting, the index cannot be leveraged, causing significant performance slowdown. The patch changes the type of column `id` from `varchar` to `INT`.

[AP-26] Non-optimal force index

Root cause: Developers construct queries with a force index, which unfortunately is not optimal.

Fix Strategy: Removing the force index and using the optimal index.

Examples: Figure 3 shows *MediaWiki* #59285. With the force index on `page_random`, the query will check the where conditions starting with the indexed column `page_random` and then continue to check other conditions for each record. However, there are a lot of rows satisfying the condition `page_random >= 0`. So the size of rows to be checked with other conditions will still be large. The fix removes the force index on `page_random`. After patching, the database engine will start with checking the indexed column

```
# Before fix
query = "SELECT ... FROM page FORCE INDEX (page_random)
     WHERE page_namespace = $page AND page_is_redirect =
     '0' AND page_random >= 0 ORDER BY page_random LIMIT
     1;"

# After fix
query = "SELECT ... FROM page WHERE page_namespace =
     $page AND page_is_redirect = '0' AND page_random >=
     0 ORDER BY page_random LIMIT 1;"

# Affected table schema
CREATE TABLE page(
  ...
  KEY page_namespace (page_namespace)
  KEY page_random (page_random)
)
```

Fig. 3. *MediaWiki* bug #59285. Before and after code snippets are shown.

```
$sql = "SELECT gi.id FROM {grade_items} gi
- WHERE ... AND gi.categoryid IN (
- SELECT gc.id FROM {grade_categories} gc
- WHERE gc.path LIKE ?)"
+ JOIN {grade_categories} gc ON gi.categoryid = gc.id
+ WHERE ... AND gi.courseid = ?
+ AND gc.path LIKE ?"
```

Fig. 4. *Moodle* bug #42065. Patch diff results are shown.

`page_namespace` and find a small number of rows satisfying the condition on `page_namespace`, leading to speedup on query execution.

[AP-27] Changing subqueries to join operations

Root cause: When a query can be implemented with subqueries or join operations, there are cases where using join operations is more efficient. This situation can happen when the size of the joined tables is not large.

Fix Strategy: Changing subqueries to join operations.

Examples: Figure 4 shows *Moodle* #42065, where the fix changes the way how the `categoryid` field of the `grade_items` table is matched with the `id` field of the `grade_categories` table from an `IN` clause with a subquery to a `JOIN` operation. Since the size of each table is not large, the cost of joining those tables is smaller than the subquery, which will need to create and destroy temporary storage for subquery results.

[AP-28] Changing join operations to subqueries

Root cause: When a query can be implemented with subqueries or join operations, there are cases where using subqueries is more efficient. This situation can happen when the size of the joined tables is large.

Fix Strategy: Changing join operations to subqueries.

Examples: Figure 5 shows *Moodle* #32340. In order to fix this bug, the patch changes the way how the `id` field of the `course` table is matched with the `courseid` field of the `event` table from a `JOIN` operation to an `EXISTS` clause with a subquery. The fix improves performance as doing a subquery on the `event` table takes less time than joining multiple tables that are very large.

```
$sql = "SELECT c.*, ... FROM ... JOIN {course} c
- JOIN {event} e ON e.courseid = c.id
+ WHERE EXISTS (SELECT 1 FROM
+ {event} e WHERE e.courseid = c.id)
```

Fig. 5. *Moodle* bug #32340. Patch diff results are shown.

```
# Construct the join clause
- $join .= "LEFT JOIN bugs_activity actcheck");

# Construct the where conditions
...
my @list;
foreach my $f (@chfield) {
   if(...){
      push(@list, "actcheck.fieldid = " . get_field($f));
   } else {
      ...
   }
}

if(@list) {
+ $join .= "LEFT JOIN bugs_activity actcheck");
   foreach my $l (@list){
      $where .= $l;
   }
}

$where .= ...
...

$query .= $join . $where
```

Fig. 6. *BugZilla* bug #226284. Patch diff results are shown.

[AP-29] Joining unused tables

Root cause: Queries join tables that are not used.

Fix Strategy: Removing the tables being joined but not used by queries.

Examples: Figure 6 shows *BugZilla* #226284. For each value in the `@chfield` list, if it satisfies certain conditions, the value will be compared with the `actcheck.fieldid` field, where `actcheck` is an alias for the `bugs_activity` table. However, if no value in the `@chfield` list satisfies the conditions, it is unnecessary to join the `bugs_activity` table. The buggy code joins the `bugs_activity` table at the beginning, while the fix joins the table only if it will really be checked against with.

[AP-30] Unnecessary locks

Root cause: Tables get locked unnecessarily while executing some queries.

Fix Strategy: Remove the unnecessary locks in the queries.

Examples: Figure 7 shows *BugZilla* #301020, where the queries update some fields in the `components` table with a `WRITE` lock. Since the information of a component's `initialowner` and `initialqacontact` actually correspond to a user's username, which is stored in the `profiles` table. So, a `READ` lock needs to be added on the `profiles` table. No information stored in the `products` table is needed by these queries. Therefore, the `READ` lock on the `products` table is unnecessary and removed by the patch.

```
# Lock tables
- $dbh->bz_lock_tables('components WRITE', 'products
     READ', 'profiles READ');
+ $dbh->bz_lock_tables('components WRITE', 'profiles
     READ');

# Execute queries
$dbh->do("UPDATE components SET name = ? WHERE id = ?",
     $name, $component_id);
$dbh->do("UPDATE components SET initialowner = ? WHERE
     id = ?", $assignee_id, $component_id);
$dbh->do("UPDATE components SET description = ? WHERE id
     = ?", $description, $component_id);
$dbh->do("UPDATE components SET initialqacontact = ?
     WHERE id = ?", $qacontact_id, $component_id);

# Unlock tables
$dbh->bz_unlock_tables();
```

Fig. 7. *BugZilla* bug #301020. Patch diff results are shown.

```
# Before fix
query = "SELECT ... FROM ... LEFT JOIN bugs LEFT JOIN
     (SELECT bug_id FROM comments WHERE ...) as c ON
     bugs.bug_id = c.bug_id ... WHERE ...;"

# After fix
query = "SELECT ... FROM ... LEFT JOIN bugs LEFT JOIN
     (SELECT DISTINCT bug_id FROM comments WHERE ...) as
     c ON bugs.bug_id = c.bug_id ... WHERE ...;"
```

Fig. 8. *BugZilla* bug #818007. Before and after code snippets are shown.

[AP-31] Subquery returning duplicated rows

Root cause: Subqueries in a query could return duplicated rows, leading to unnecessary computation in the query.

Fix Strategy: Removing duplicated rows from the results of subqueries.

Examples: Figure 8 shows *BugZilla* #818007, where the results of a `SELECT` subquery will be used as a table to be joined in the query. The problem is that the result of the subquery contains a large number of duplicates. After adding a `DISTINCT` keyword to remove the duplicates, the subquery dramatically reduces the amount of data returned, reducing the time needed for the query to join the result table of the subquery.

[AP-32] Conditions containing subsuming clauses

Root cause: A query could contain condition clauses where one may subsume another, leading to unnecessary computation in the query.

Fix Strategy: Removing the condition clauses that are subsumed by others.

Examples: Figure 9 shows *WordPress* #17152. The comparison with the whole `$search_term` string for string search in the text is not necessary, because if all substrings of `$search_term` have been searched in the text, there is no need to search the whole `$search_term` string in the text. The fix removes the condition clause that compares with the whole `$search_term` string; this clause is subsumed by condition clauses that compare with its substrings, and thus is unnecessary.

```
foreach( (array) $q['search_terms'] as $term ) {
   $search .= "$searchand" . "(($wpdb->posts.post_title
        LIKE '%{$term}%') OR ($wpdb->posts.post_content
        LIKE '%{$term}%'))";
   $searchand = ' AND ';
}
- $search .= " OR ($wpdb->posts.post_title LIKE
     '%{$search_term}%') OR ($wpdb->posts.post_content
     LIKE '%{$search_term}%')";
```

Fig. 9. *WordPress* bug #17152. Patch diff results are shown.

```
+ if ($params->param('bug_status')) {
+  my @bug_statuses = $params->param('bug_status');
+  if (scalar(@bug_statuses) ==
     scalar(@::legal_bug_status)) {
+     $params->delete('bug_status');
+  }
+ }
+
+ if ($params->param('resolution')) {
+  my @resolution = $params->param('resolution');
+  if (scalar(@resolution) ==
     scalar(@::legal_resolution)) {
+     $params->delete('resolution');
+  }
+ }
+

foreach my $field ($params->param()) {
   push(@where, join("OR" . @params->param($field)));
}
```

Fig. 10. *BugZilla* bug #173571. Patch diff results are shown.

[AP-33] Unnecessary `where` *clause when all conditions are selected*

Root cause: A query could contain `where` condition clauses that compare a field with all its possible values, making all these clauses unnecessary.

Fix Strategy: Removing the `where` condition clauses that cover all possible values.

Examples: Figure 10 shows *BugZilla* #173571. In this example, the query will build a clause into the `where` condition for each user-selected value. When all possible legal values of a column are selected, there is no need to include those clauses into the `where` condition. The fix removes the condition clauses on `bug_stats` and `resolution` to save the time for checking these conditions.

[AP-34] Unnecessary query construction

Root cause: A query is constructed but not sent to the DBMS for execution.

Fix Strategy: Remove the unnecessary query-construction code logic.

Examples: Figure 11 shows *Joomla!* #23164. Originally, the application first constructs a query, and then tries to load data from cache. If there is a cache hit, the query will not be executed. The fix instead gets cache results first (doing so is fast), and constructs the query only if there is a cache miss.

B. Discussion

The 10 new database-access performance antipatterns unveiled by our study show the benefits of going beyond ORM-based web applications and studying performance bugs in direct-accessing web applications.

```
# Before fix
# Construct the query
$query .= "SELECT ...";
$query .= "FROM ...";
$query .= "LEFT JOIN ... ON ..."
$query .= "WHERE ..."

#Check if cache is empty
$modules = $cache->get($cacheid);
if (null === $modules){
   $modules = $db->loadObjectList();
}

# After fix
# Check if cache is empty
if(!($modules = $cache->get($cacheid)){
   # Construct the query
   $query .= "SELECT ...";
   $query .= "FROM ...";
   $query .= "LEFT JOIN ... ON ..."
   $query .= "WHERE ..."
   $modules = $db->loadObjectList();
}
```

Fig. 11. *Joomla!* bug #23164. Before and after code snippets are shown.

On the generality of the 10 new database-access performance antipatterns, we believe that all the 10 new antipatterns are applicable to other direct-accessing web applications. Although the numbers of some antipatterns are small, e.g., [AP-31] to [AP-34] each have only one matching performance bug in our study, the necessary program and query features are all general to all direct-accessing web applications, and similar mistakes can happen. Therefore, we consider these antipatterns as new ones despite the small numbers of studied bugs falling into some of the antipatterns.

Among the 10 new antipatterns, the first nine, i.e., [AP-25] to [AP-33], are also applicable to ORM-based web applications, as the underlying root causes are not specific to direct-accessing web applications and the necessary features involved in each antipattern are available in ORM frameworks. Since ORM-based web applications do not construct queries directly, [AP-34] (unnecessary query construction) is not directly applicable to ORM-based web applications. However, an analogy exists, i.e., ORM frameworks could perform unnecessary query construction while translating ORM-API calls to queries.

Among the 33 performance bugs matching new performance antipatterns, 12 of them match antipatterns [AP-25] or [AP-26], both of which are related to database indexes. We have discussed a similar finding in Section IV that the number of performance bugs matching [AP-19] missing indexes is the largest both in our study and the study by Yang et al. [50]. Our results on direct-accessing web applications show that the challenges of using database indexes effectively go beyond the phase of database table design but also lie in how to select and use appropriate indexes.

Similar to the known performance antipatterns, where duos of opposite antipatterns exist, e.g., [AP-02] (moving computation to the DMBS) vs. [AP-03] (moving computation to

the server), and [AP-09] (inefficient lazy loading) vs. [AP-12] (inefficient eager loading), one duo of opposite exists in our new antipatterns, i.e., [AP-27] (changing subqueries to join operations) vs. [AP-28] (changing join operations to subqueries). This case serves as another example that one-fits-all design does not exist for accessing database efficiently in web applications.

Although only one antipattern can be considered as unique for direct-accessing web applications, i.e., [AP-34] (unnecessary query construction), detecting performance bugs in direct-accessing web applications using antipatterns shared with ORM-based web applications may encounter extra challenges due to the difference between ORM-based web applications and direct-accessing web applications. For example, it will be necessary to implement string analysis [26] or other necessary techniques to handle dynamically generated queries in direct-accessing web applications. We leave for future work the design, development, and evaluation of bug detection techniques that leverage the 34 database-access performance antipatterns.

VI. Related Work

Earlier in this paper, we have discussed related work on database-access antipatterns and performance-bug studies. Our work is unique in that we target at reporting all known database-access performance antipatterns that can be found in the literature and complementing existing studies by focusing on direct-accessing web applications. We next discuss other related work.

Various previous approaches focus on detecting antipatterns. Some of them do not focus on performance antipatterns but on functionality antipatterns [13], [20], [29], and some of them focus on performance antipatterns in ORM-based applications [18], [21], [51]. Our results suggest that there are multiple performance antipatterns that are currently missed by existing work on ORM-based web applications, and direct-accessing web applications deserve more research attention.

Both static approaches [21], [28], [31], [37], [44], [48], [50], [51] and dynamic approaches [36], [38], [47] have been explored to detect different types of performance issues. It is challenging to develop static checkers for applications studied in our work, as currently there is a lack of mature implementations that can statically analyze dynamically generated queries involving multiple programming languages. Future work can leverage lessons from our study to develop performance bug detection tools for direct-accessing web applications.

There are multiple pieces of work focusing on improving the performance of database-backed applications [15], [19], [22], [23], [34], [42]. Although some of the key ideas are similar to the fix strategies seen in performance-bug patches, such as reducing database accesses and optimizing query execution efficiency, these database-side optimization approaches cannot optimize away the performance inefficiencies of related bugs in our study, as existing approaches are usually limited to a single round-trip between application and database, while many database-access antipatterns involve multiple round-trips. Future work can leverage database-access antipatterns summarized in our study to develop more comprehensive and powerful optimization approaches.

VII. Threats to Validity

The validity of our study results may be subject to multiple threats. Below we describe potential threats and our ways to address them.

The first threat is the likely incompleteness of our surveyed publications. We alleviate this threat by following the state-of-the-art literature-survey methodology, and we cross-check our results with the previous literature surveys on the surveyed publications and the reported antipatterns.

The second threat is the likely lack of representativeness of the studied applications. To minimize this threat, we choose popular open-source applications with a significant user base. Although *BugZilla* has the largest number of studied bugs, it does not bias our results, because the overall distribution of categorized bugs spreads across different web applications and antipatterns. So the characteristics of our studied bugs can likely be generalized to other database-backed web applications.

The third threat is that we may miss relevant bug reports during our search for performance bugs. We mitigate this threat by using keyword search together with bug categories and tags. We also search bug descriptions and comments in addition to bug report summaries, as developers tend to use common terms in the description and comments.

The fourth threat is related to our manual inspection of the collected bug reports. The manual inspection is independently performed and verified by at least three authors to alleviate this threat. If there are different opinions on a bug report, we discuss the bug report together to reach a consensus.

VIII. Conclusion

In this paper, we have presented a comprehensive empirical study that characterizes performance antipatterns related to database accesses in web applications. From our literature survey, we have summarized and reported a total of 24 known performance antipatterns, and the comprehensiveness of our results makes it a great reference for future work on database-access performance antipatterns. Based on real-world performance bugs from direct-accessing web applications, we have found 10 new database-access performance antipatterns that are not previously reported in the research literature. Our study results can guide future research in combating performance bugs related to database accesses in web applications.

IX. Acknowledgments

This work was supported in part by US NSF grant no. CCF-2008056, CNS-1564274, CCF-1816615. Tao Xie is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, and is the corresponding author.

REFERENCES

[1] "Bugzilla," https://bugzilla.mozilla.org/.

[2] "Dnn platform issue tracker," https://dnntracker.atlassian.net.

[3] "Joomla! developer network," https://developer.joomla.org/.

[4] "Moodle tracker," https://tracker.moodle.org/.

[5] "Odoo issues," https://github.com/odoo/odoo/issues/.

[6] "Top 15 Most Popular Websites — January 2020," http://www.ebizmba.com/articles/most-popular-websites.

[7] "Usage statistics and market share of WordPress for websites," https://w3techs.com/technologies/details/cm-wordpress/all/all.

[8] "Wikimedia phabricator," https://phabricator.wikimedia.org/.

[9] "Wordpress trac," https://core.trac.wordpress.org/.

[10] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang, "Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications: An experience report," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–12. [Online]. Available: https://doi.org/10.1145/2901739.2901774

[11] Akamai Technologies, Inc, "Akamai Reveals 2 Seconds as the New Threshold of Acceptability for eCommerce Web Page Response Times," https://www.akamai.com/us/en/about/news/press/2009-press/akamai-reveals-2-seconds-as-the-new-threshold-of-acceptability-for-ecommerce-web-page-response-times.jsp, 2009.

[12] ——, "Akamai Online Retail Performance Report: Milliseconds Are Critical," https://www.akamai.com/us/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp, 2017.

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

[14] B. Asmare Muse, M. Masudur Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "On the prevalence, impact, and evolution of SQL code smells in data-intensive systems," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3379597.3387467

[15] I. T. Bowman and K. Salem, "Optimization of query streams using semantic prefetching," *ACM Trans. Database Syst.*, vol. 30, no. 4, pp. 1056–1101, Dec. 2005. [Online]. Available: http://doi.acm.org/10.1145/1114244.1114250

[16] S. Chaudhuri, V. Narasayya, and M. Syamala, "Bridging the application and dbms profiling divide for database application developers," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, p. 1252–1262.

[17] B. Chen, Z. M. J. Jiang, P. Matos, and M. Lacaria, "An industrial experience report on performance-aware refactoring on a database-centric web application," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 653–664. [Online]. Available: https://doi.org/10.1109/ASE.2019.00066

[18] T. 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.

[19] 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*, ser. FSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 666–677. [Online]. Available: https://doi.org/10.1145/2950290.2950303

[20] ——, "Detecting problems in the database access code of large scale systems: An industrial experience report," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 71–80. [Online]. Available: https://doi.org/10.1145/2889160.2889228

[21] 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 '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1001–1012. [Online]. Available: https://doi.org/10.1145/2568225.2568259

[22] A. Cheung, S. Madden, O. Arden, and A. C. Myers, "Automatic partitioning of database applications," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1471–1482, Jul. 2012. [Online]. Available: http://dx.doi.org/10.14778/2350229.2350262

[23] 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

[24] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers, "Using program analysis to improve database applications," *IEEE Data Engineering Bulletin*, vol. 37, no. 1, pp. 48–59, 2014.

[25] 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: Association for Computing Machinery, 2013, p. 3–14. [Online]. Available: https://doi.org/10.1145/2491956.2462180

[26] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS '03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 1–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1760267.1760269

[27] R. F. Dugan, 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: Association for Computing Machinery, 2002, p. 10–16. [Online]. Available: https://doi.org/10.1145/584369.584372

[28] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 77–88. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254075

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

[30] B. Kitchenham, "Procedures for performing systematic reviews," Joint Technical Report, Computer Science Department, 2004, Keele University (TR/SE-0401) and National ICT Australia Ltd. (0400011T.1), 2004.

[31] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE '14. New York, NY, USA: ACM, 2014, pp. 1013–1024. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568229

[32] Y. Lyu, A. Alotaibi, and W. G. J. Halfond, "Quantifying the performance impact of SQL antipatterns on mobile applications," in *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '19. IEEE, 2019, pp. 53–64. [Online]. Available: https://doi.org/10.1109/ICSME.2019.00015

[33] 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 '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 310–321. [Online]. Available: https://doi.org/10.1145/3213846.3213865

[34] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic, "Holistic query transformations for dynamic web applications," in *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ser. ICDE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1175–1178. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2009.194

[35] C. Nagy and A. Cleve, "SQLInspect: A static analyzer to inspect database usage in Java applications," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 93–96. [Online]. Available: https://doi.org/10.1145/3183440.3183496

[36] K. Nguyen and G. Xu, "Cachetor: Detecting cacheable data to remove bloat," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '13. New

York, NY, USA: ACM, 2013, pp. 268–278. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491416

[37] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 237–246. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487134

[38] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 562–571. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486862

[39] U. S. G. A. Office, " HEALTHCARE.GOV CMS Has Taken Steps to Address Problems, but Needs to Further Implement Systems Development Best Practices," http://www.gao.gov/assets/670/668834.pdf.

[40] D. Qiu, B. Li, and Z. Su, "An empirical analysis of the co-evolution of schema and code in database applications," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '13. New York, NY, USA: ACM, 2013, pp. 125–135. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491431

[41] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan, "Program transformations for asynchronous and batched query submission," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 2, pp. 531–544, 2015.

[42] K. Ramachandra and S. Sudarshan, "Holistic optimization by prefetching query results," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 133–144. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213852

[43] Z. Scully and A. Chlipala, "A program optimization for automatic database result caching," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 271–284. [Online]. Available: https://doi.org/10.1145/3009837.3009891

[44] M. Selakovic and M. Pradel, "Performance issues and optimizations in JavaScript: An empirical study," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884829

[45] T. Sharma, M. Fragkoulis, S. Rizou, M. Bruntink, and D. Spinellis, "Smelly relations: Measuring and understanding database schema quality," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 55–64. [Online]. Available: https://doi.org/10.1145/3183519.3183529

[46] 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: Association for Computing Machinery, 2012, p. 983–996. [Online]. Available: https://doi.org/10.1145/2384616.2384688

[47] X. Xiao, S. Han, D. Zhang, and T. Xie, "Context-sensitive delta inference for identifying workload-dependent performance bottlenecks," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA '13. New York, NY, USA: ACM, 2013, pp. 90–100. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483784

[48] 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

[49] ——, "View-driven optimization of database-backed web applications," in *10th BiAnnualennial Conference on Innovative Data Systems Research*, ser. CIDR '20. www.cidrdb.org, 2020.

[50] 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," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 800–810. [Online]. Available: https://doi.org/10.1145/3180155.3180194

[51] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung, "Powerstation: Automatically detecting and fixing inefficiencies of database-backed web applications in IDE," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 884–887. [Online]. Available: https://doi.org/10.1145/3236024.3264589

[52] 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. IEEE Press, 2019, p. 994–1004. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00104

[53] 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: Association for Computing Machinery, 2007, p. 533–544. [Online]. Available: https://doi.org/10.1145/1247480.1247540