A CHARACTERISTIC STUDY OF PERFORMANCE BUGS IN APPLICATION-DATABASE INTERACTIONS

BY

MENGQI GU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Associate Professor Tao Xie

ABSTRACT

Environmental interactions (e.g., file I/O, network communication, database querying) are common bottlenecks of software applications. These interactions are also prone to performance bugs because developers may not understand the performance implication of the information sent to or from the environment (e.g., a database query sent to a database or a result set returned from the database). As a result, the performance bugs can further magnify the bottlenecks. Understanding the characteristics of these performance bugs is crucial for developers and testers to better address performance problems. Such understanding also provides guidance for researchers and tool vendors to develop effective tool support. However, there has been no study for understanding such characteristics in real-world software.

To fill this gap, in this thesis, we present the first empirical study of bug reports for database-related performance bugs collected from popular real-world open-source projects (i.e., *BugZilla*, *DNN*, *Joomla!*, *MediaWiki*, *WordPress*, *Simple Machines*, and *Roundcube*). We study common optimization opportunities, types of database-related performance bugs, and difficulties of fixing these bugs. Among the studied bug reports, we identify nine common bug types and seven common fix strategies. We also observe that bugs of certain types require more effort to diagnose and fix. Furthermore, we identify various opportunities for tool support to identify and diagnose these bugs.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would especially thank my adviser, Associate Professor Tao Xie, for his support and guidance. I would also acknowledge the contribution from two teammates that I have been working with on this project. Xiao (Lester) Yu and Wei Yang were the initial members when this project started. They structured the preliminary conceptual framework of this study. Xiao and Wei also contributed during the bug report investigation. I would like to Xiao and Wei for their help and support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

A large number of popularly used software applications need to store and process data produced or consumed by users. For example, web applications typically store data such as user information and usage data. Software applications commonly use databases as the back-end of data processing and storage. Such software applications are referred to as database applications.

Performance of database applications is critical for their success because it can directly affect the user experience. Web database applications can lose users if the applications cannot efficiently load web pages. Online transaction systems may not be able to process enough transactions if their databases suffer from performance problems. These performance problems can also crash the applications and affect the daily lives of thousands or millions of users. For example, after the initial deployment of the Obamacare website Healthcare.gov, consumers suffered "widespread performance issues when trying to create accounts and enroll in health plans," according to the official report released by the U.S. Government Accountability Office [1]. The report also indicated that various critical software defects were identified from the main system, whose core is a transactional database.

Performance bottlenecks of database applications can be attributed to two main aspects. First, the database management systems (DBMS) itself can be decisive to the overall application performance. The database research community has been addressing the performance issues in DBMSs for many years, e.g., by improving storage systems [19] and leveraging GPU powers [10, 18]. On the other hand, database administrators are also equipped with the practical knowledge of tuning DBMS instances to run in optimal configurations.

Second, the application-database interactions can be another major source of performance bottlenecks. For example, query instances and database result-set instances are commonly transferred over the network. As the size of an instance or the number of these instances increases, increased network overheads are incurred. This problem is exacerbated when the result-set instances are large: it may require more time for the database to prepare and for the application to process the results.

By investigating bug reports, we observe an alarming fact that many performance problems in database applications are caused by performance bugs of the second aspect: application database interactions. As a result, these bugs further increase the bottlenecks of the interactions. Following a previous convention [9], performance bugs are defined as software defects where certain changes of source code can significantly speed up software. Although the previous convention also specifies that these changes should preserve functionality, our observation indicates that for a non-trivial number of performance bug reports, the accepted fixes do modify functionality. Performance bugs can be introduced when developers are implementing interactions (e.g., queries, schema designs) between an application and a DBMS. We refer to these kinds of bugs as database-related performance bugs.

Understanding the characteristics of database-related performance bugs can greatly benefit developers, testers, researchers, and tool vendors. First, developers can understand what types of application-database interactions are more prone to database-related performance bugs. This understanding can provide guidance for developers to avoid, diagnose, and fix these bugs. Second, testers can understand what types of input are more likely to trigger database-related performance bugs, and thus they can more effectively test against such bugs. Third, researchers and tool vendors can gain a deep understanding of the bug characteristics to produce new solutions of tool support to detect or fix database-related performance bugs.

However, there has been no characteristic study of database-related performance bugs. Worse still, findings from previous studies [9, 14] of general performance bugs are not applicable to database-related performance bugs. Database applications use a Data Manipulation Language such as SQL together with application programming languages to interact with databases. Studying such interactions requires studying multiple programming languages together. Existing techniques and studies focus on applications written in one single programming language. Furthermore, critical performance-related information from the database side can be hidden behind the database interface. These characteristics of database applications are unique and may not be observed among general performance bugs.

To address this issue of lacking a characteristic study, we present the first empirical study of database-related performance bugs in open-source database applications. This study covers 183 database-related performance bugs collected from seven popular real-world open-source database applications: *BugZilla*, *DNN*, *Joomla!*, *MediaWiki*, *WordPress*, *Simple Machines*, and *Roundcube*. Note that the number of real-world performance bugs (183) in our study is comparable with the number of real-world bugs studied in other state-of-the-art empirical studies (e.g., in the study published in a PLDI 2012 paper [9], Jin *et al.* studied 109 real world general performance bugs), and studying each bug report consumes many man hours. Given the common belief that optimization opportunities may commonly exist in the design and implementation of application-database interactions, we address three main research questions through our study:

- **RQ1:** What are common categories of database-related performance bugs?
- **RQ2:** What are common fix strategies of database-related performance bugs?
- **RQ3:** How much time elapses and how many developers are involved in the discussion before a category of database-related performance bugs is fixed?

Identifying common types or fix strategies of database-related performance bugs can provide guidance for detecting and fixing database-related performance bugs. The elapsed time and involved developers can suggest the difficulty of fixing a category of database-related performance bugs. It can also suggest the priority of allocating bug-fixing efforts and resources.

This thesis makes the following major contributions:

- We conduct the first empirical study of bug reports for database-related performance bugs collected from popular real-world open-source database applications.
- We identify common categories of database-related performance bugs along with fix strategies and fix difficulties for these bugs. Such finding can provide guidance for avoiding and diagnosing database-related performance bugs, and for research in tool support for detecting and fixing these bugs.

CHAPTER 2

MOTIVATING EXAMPLES

Real-world database-related performance bugs are very diverse in their causes. Such diversity reflects the intrinsic complexity in the implementation of application-database interactions. As missing indexes from database schema is well known to cause slow queries, there are other causes that are more complex and more difficult to diagnose.

On the application side, the implementation of application logic should take into account the potential performance impact from the application's database interactions. Figure 2.1 shows a simplified code snippet extracted from the patch for *BugZilla* #286625 (*BugZilla* is implemented in Perl, with MySQL as its database back-end). The original code contains a performance bug that executes queries inside a nested loop to obtain the status and resolution for each bug entity (to avoid confusion, we use "bug entity" to refer to a bug managed and stored in the *BugZilla* database). This implementation is functionally correct according to the application logic. However, the implementation can submit too many query instances to the back-end database, causing significant overheads in processing query instances and transferring data.

The fix for this bug is to reduce the number of query instances by obtaining and storing the status and resolution information for all the bug entities beforehand in one query. Later, each loop iteration reads the stored information instead of querying the database repeatedly. After the fix, this code snippet "take[s] minutes instead of hours or days," according to the discussion in the bug report. Note that the developers mis-diagnosed the problem at first and it took years before this bug was finally fixed.

On the database side, an improper combination of schema design and query instances can result in significant performance loss. Such problems can reflect a certain level of information hiding, which is rooted from developers' not knowing how a database handles the application-database interactions

```
# Before fix
for (my $day = $start + 1; $day <= $end; $day++)
   for my \$bug (@bugs)
      my $status = $dbh->selectrow_array($sth_bug, undef, $bug,
          'bugStatus');
      my $resolution = $dbh->selectrow_array($sth_bug, undef, $bug,
          'resolution');

# After fix
my (%bugStatus, %bugResolution);
%bugResolution = {$dbh->selectcol_arrayref('SELECT bugId,
    resolution FROM bugs', {Columns=>[1,2]}) };
%bugStatus = {$dbh->selectcol_arrayref('SELECT bugId, bugStatus
    FROM bugs', {Columns=>[1,2]})};
for (my $day = $start + 1; $day <= $end; $day++)
   for my \$bug (@bugs)
      use_data_structure(%bugStatus, %bugResolution);
```

Figure 2.1: *BugZilla* #288625 (Simplified)

```
# Before fix
CREATE TABLE IF NOT EXISTS '#__associations' ('id' varchar(50) NOT
    NULL, ...)

# After fix
CREATE TABLE IF NOT EXISTS '#__associations' ('id' INT(11) NOT
    NULL, ...)
```

Figure 2.2: *Joomla!* #29845 (Simplified)

internally. Figure 2.2 shows a patch from *Joomla!* #29845. Developers are able to identify a slow query that contains a join operation. This operation joins column ‘id’ in table ‘_ _associations’ with a column in another table, where the type of column ‘id’ is string, but the type of other column is integer. When the database performs the join operation, it has to do an extra type conversion on the two columns with different data types, causing significant performance loss. This extra computation was initially unnoticed by the developers. The fix is simply changing the column type from “`varchar`” to “`INT`,” but it brings from 85 to 166 times of speedup according to the tests in the bug report.

In general, database-related performance bugs can have a huge impact on performance. Fixing these bugs may require non-trivial changes to the query, database schema, and the application logic itself. Furthermore, the mechanism of information hiding makes it hard for developers to conduct effective and efficient diagnosis without understanding these bugs' characteristics. Understanding such characteristics can provide guidance for detecting and fixing database-related performance bugs.

CHAPTER 3

METHODOLOGY

In this chapter, we describe the subjects used for our study and the process that we adopt to collect and analyze bug reports.

3.1 Subjects

We select seven popular open-source database applications on the Internet: *BugZilla*, *DNN*, *Joomla!*, *MediaWiki*, *WordPress*, *Simple Machines*, and *Roundcube* (shown in Table 4.1). *Joomla!* and *MediaWiki* are content-management systems and are used as subjects in a previous study [16]. *Joomla!* is also very popular and won the *Open Source Award* in 2011 [16]. *DNN* and *WordPress* are also popular content-management systems. *DNN*'s customers include large companies such as Bank of America, Canon, and BP while *WordPress* is used by the New York Times, CNN, and Ebay. *Simple Machines* is an online community system with 3,659,864 total posts as of July 2016 [5]. *BugZilla* is a bug-management system heavily used by Mozilla. *Roundcube* is a browser-based email client. We classify these applications as database applications because their features heavily rely on databases.

To identify database-related performance bugs, we first search bug repositories of the subjects with the keywords "timeout", "slow", and "performance" in order to retrieve performance bugs. If a repository explicitly has a specific category of performance bugs, we directly use such category without a keyword search. Across all the subjects, only *Simple Machines* has a performance category in its repository. After obtaining an initial set of performance bug reports, we filter out bug reports that are irrelevant to application-database interactions. Specifically, we keep only the reports whose description and comments contain the keywords "database", "query", or "schema". We observe that developers tend to use acronyms while describing a database or a query. Therefore, we also include keywords "db" and "sql" in our search. Note that our search is case-insensitive.

We include every bug report that we identify as database-related performance bugs. Our final subject set contains 183 bug reports. In Table 4.1, column "Versions of Study" shows the range of affected versions involved in the studied bug reports; column "Length of Study" shows the range of time shown by the reported and fixed dates in the studied bug reports; column "Version Control Availability" shows whether a public version control system is available for us to extract revisions for bug fixes. The last column shows the total number of bug reports that we study. We observe that the number of bug reports tends not to distribute evenly across the entire lifetime. For example, project *BugZilla* was initially started 15 years ago. However, over half of the bug reports were created from 2005 and 2009. One possible explanation can be that the *BugZilla* developers put more emphasis on performance during that time period.

3.2 Analysis Process

A bug report typically contains a bug description, followed by multiple comments on possible causes and fixes, and the committed fix. Our study centers around these parts to investigate the research questions. Depending on whether or not the version control system of an open-source application is public, a committed fix may or may not be available for our inspection. If such information is not publicly available, we then heavily rely on the bug description and comments in order to infer the concrete situation in the application as well as the potential fix. More specifically, there are in total 4 bug reports from *DNN* and 17 bug reports from *Simple Machines* for which we are unable to locate the corresponding source code for the bug and the fix. However, 16 of these bug reports contain the problematic query code and fixes in the bug descriptions. Note that every bug report is inspected and discussed by at least two authors to ensure objectivity of the conclusions.

To address *RQ1*, our analysis centers around queries as the starting point. Database-related performance bugs may manifest directly from query executions, and correlate with various attributes of queries. To categorize those performance bugs, we start from investigating what attributes of queries are related to the reported performance problems. We manually investigate each bug report, look for what attributes have been observed regarding queries, and then conclude what particular reasons in the program code, schemas, or the database behaviors cause the performance problems. We use the observable attributes of queries and the concluded reasons to label all collected bug reports. Such labeling naturally leads to a multi-level categorization system, which starts from observable attributes to possible reasons.

To address *RQ2*, we focus on the types of changes, which include the place of changes (program code, queries, or schemas), and the detailed changes in the three parts. To identify how a bug report was fixed, we first manually inspect the bug description and developer comments to locate the patch that contains the final fix. We review the patch submitter's description of the fix and then inspect the code in the patch to look for changes in the program code, queries, or the schema. We leverage the patch submitter's description to help us understand the code. By analyzing the relations between change types and bug categorizations, we are able to learn how those performance bugs are fixed.

To address *RQ3*, we collect the following metrics:

- The amount of time between the creation and closing of a bug report (measured in days).
- The number of developers involved in the discussion prior to the fix.

CHAPTER 4

RESULTS AND ANALYSIS

We start this chapter with our terminology for application-database interactions and their optimization opportunities. Such terminology lays a foundation for our later-presented taxonomy. Our empirically based terminology and taxonomy are grounded from our empirical observations on the studies subjects. Finally, we present the detailed study results centered around such terminology and taxonomy.

4.1 Terminology

Application-database interactions. Typical interactions between a software application and its back-end database involve a set of query instances and their result-set instances. The application constructs and sends each query instance to the database, and the database executes the query instance and sends back its result-set instance to the application for further processing. As depicted by Figure 4.1, to implement a specific feature, an application (**APP**) and its database (**DB**) may have multiple round trips, each of which consists of a query instance (i.e., Q_0, Q_1, or Q_n), and its result-set instance (i.e., R_0, R_1, or R_n).

Performance-affecting attributes. Based on our empirical observations of the studies subjects, we model the cost $C(Q)$ of an application interacting with its database as follows:

$$C(Q) = \sum_{q \in Q} [C_{DB}(q, R(q)) + \sum_{r \in R(q)} C_{APP}(r)]$$

With a set of query instances Q that implement a specific application feature, the overall cost is composed by the costs of the database's executing

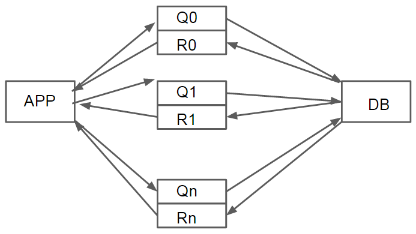

Figure 4.1: Interactions between Application and Database

each query instance (represented by C_{DB}) and the application's processing each data entry in the result-set instance (represented by C_{APP}).

The database cost C_{DB} is parameterized by each query instance q and its result-set instance $R(q)$, since the database needs to both execute the query and transfer the query result. The application cost C_{APP} is parameterized by each data entry r in result-set instance $R(q)$ from the database.

This conceptual equation essentially shows several attributes that may affect the performance of application-database interactions: the cardinality of Q, the characteristics of individual instances in Q, the cardinality of $R(q)$, and the characteristics of individual entries in $R(q)$. In addition, as the number of query instances increases, the latency of transferring the query instances over the network also increases. Similarly, an increased number of result-set instances can also result in a higher latency due to transmitting more data over the network. In the subsequent sections, we provide a categorization that studies these attributes in detail.

Optimization opportunities. We refer to optimization opportunities as possible changes in performance-affecting attributes for reducing overall costs. Optimization opportunities essentially suggest possible fixes for database-related performance bugs. For instance, different query instances may be combined into one query instance to reduce the overall cost. To understand what optimization opportunities can exist in database-related performance bugs, we empirically construct a taxonomy using a bottom-up, iterative style while studying the bug reports.

Table 4.1: Subjects

Application	Description	Versions of Study	Length of Study	Version Control Availability	# of Studied Bugs
BugZilla	Bug management system	2.13 - 4.5	2000 - 2015	Y	62
DNN	Content management system	5.3.0 - 7.3.0	2012 - 2015	Y	39
Joomla!	Content management system	1.6 - 2.5	2010 - 2015	Y	18
Roundcube	Web mail	0.x - 1.1.2	2005 - 2015	Y	5
MediaWiki	Content management system	1.11.x - 1.21.3	2005 - 2015	Y	15
Simple Machines (SMF)	Community forum	2.0	2007 - 2015	Y	17
WordPress	Content management system	2.9 - 3.5.1	2007 - 2015	Y	27
Total					183

Table 4.2: Bug Classification

High-level Category	Query Attribute	Bug Type
Application-Query	Number of Queries (QN)	Unnecessary Queries (UnQ)
		Repetitive Queries (ReQ)
		Iterative Queries (ItQ)
	Size of Query Results (QD)	Unnecessary Queried Data (UnD)
		Large Result Set (LaR)
Query-Database	Expected Indexes (IN)	Missing Indexes (MiI)
		Indexes Not Leveraged (NlI)
	Performance-Sensitive Operations (OP)	Unnecessary Operations (UnO)
		Expensive Operations (ExO)

4.2 RQ1: Bug Categorization

Table 4.2 shows the detailed statistics of our findings on bug categorization. We present the bug categorization in a multilevel category. In particular, we first categorize the bugs into two High-level Categories depending on whether the symptoms of performance problem occur during the application-database interactions or during the databases' execution of query instances. We name the two High-level Categories as *Application-Query* and *Query-Database* (shown in the left column of Table 4.2). For each High-level Category, we further categorize the bugs based on the different metrics that we observe from the problematic query instances. We refer to such metrics as Query Attributes (shown in the center column of Table 4.2). Finally, we categorize the bugs in each Query Attribute based on one or more bug types (shown in the right column of Table 4.2). The bug types are identified by observing how the code is changed from the proposed patch(es) for each bug report. Using this multi-level hierarchy, we can associate a detailed bug type with a generally observable attribute and a symptom of performance problem.

In Table 4.3, each row represents a Query Attribute or a bug type. Each column in the middle section corresponds to a subject. Column "Total" shows the total number of bug reports in each category, and column "CV" shows the coefficient of variation of the classified bugs across different subjects in the corresponding category. Coefficient of variation (CV) measures the dispersion of a frequency distribution, in this case, the distribution of the number of bug reports across the seven subjects for each bug type. CV is independent of the unit of the measurement and can be used to compare the dispersion of frequency distributions with different mean values. In particular, a lower value of coefficient of variation indicates more-evenly distributed occurrences. The CV for an evenly distributed frequency distribution is 0 while the value of CV cannot exceed $\sqrt{n-1}$, where n is the number of data points in the distribution (in our case n is equal to seven). Generally, a CV greater than 1 is considered high but the threshold varies for different applications. In our case, there is one bug category that has exceptionally high CV. 17 out of 19 bugs in this category are reported for *BugZilla*. Bug reporting and fixing usually involve human factors. A lower value of CV increases the confidence that the specific bug type is more generalizable across subjects.

Note that there are 11 bug reports being classified to dual categories because of the complexity of the reported bugs. Therefore, the total number of bugs in Table 4.3 is 194, whereas the total number of bug reports in Table 4.1 is 183.

4.2.1 Application-Query Category

Performance bugs in the *Application-Query* High-level Category commonly causes symptoms of performance problem in application-database interactions. There exist two observable metrics to measure the severity of the performance slowdown for this symptom. In particular, bugs in the *Application-Query* High-level Category can be associated to Query Attributes: the number of query instances and size of query result-set instances. These bugs are commonly introduced when developers are not able to construct or issue query instances in an efficient manner under certain workloads. We identify five bug types in this category (shown in Table 4.2). We introduce them based on the related query attributes as follows.

Query Attribute: Number of Queries (Total: 51, CV: 1.42). The cost of issuing and executing one query instance includes the cost to construct the query in the database application through database driver libraries, transmitting the query instance to the database across the network, processing the query instance and fetching the result in the database, and finally transmitting the result-set instance back to the database application across the network. A large number of query instances may lead to poor performance due to the latency of transferring query instances and result-set instances back and forth over the network, in addition to the latency of executing the query instances on the database side. From the studied bug reports, we identify three bug types belonging to this Query Attribute, as shown below.

Unnecessary Queries (Total: 15, CV: 0.95). In the application logic, some particular queries do not need to be executed when the program execution follows some paths. However, the developers may misplace such queries and these queries are executed. If the executions of such queries happen to be expensive, the overheads can be significant.

Table 4.3: Classified Bug Reports (B:*BugZilla*, D:*DNN*, J:*Joomla!*, M:*MediaWiki*, SMF:*Simple Machines*, W:*WordPress*, R:*Roundcube*)

Classification	**B**	**D**	**J**	**M**	**S**	**W**	**R**	**Total**	**CV**
Expected Indexes								**45**	**0.57**
IN-MiI	5	13	2	2	8	5	0	35	0.81
IN-NlI	1	0	3	4	1	1	0	10	0.97
Performance-Sensitive Operations								**47**	**0.69**
OP-ExO	12	6	3	3	6	6	0	36	0.67
OP-UnO	4	2	3	0	0	2	0	11	0.95
Size of Query Results								**26**	**0.68**
QD-LaR	3	3	0	3	1	3	0	13	0.72
QD-UnD	6	1	3	0	0	2	1	13	1.05
Number of Queries								**51**	**1.42**
QN-UnQ	6	1	2	0	0	4	2	15	0.95
QN-ReQ	9	5	1	0	0	2	0	17	1.30
QN-ItQ	17	0	1	0	0	1	0	19	2.15
Others	5	9	3	3	1	2	2	**25**	-

Figure 4.2 shows the patch for *BugZilla* #528918. In the original version, the method `_check_field` contains an expensive query. It turns out that the invocation of this method can be bypassed under some conditions: "If the field being passed to match() is `ID_FIELD`, then this field is safe, and there is no need to validate it."

```
# Before fix
$class->_check_field($field, 'match');
# After fix.
$class->_check_field($field, 'match') unless $field eq
    $class->ID_FIELD;
```

Figure 4.2: *BugZilla* #528918 (Simplified)

Repetitive Queries (Total: 17, CV: 1.30). It is possible that result-set instances may be identical across multiple executions of different query instances. Such query instances are repetitive and their issuing can be avoided by caching the result value in the database application. Figure 4.3 shows a simplified version of the fix for *Joomla!* #20675. The author of the fix comments that "I noticed that about half of the queries were just the same query executed again and again ... added in a static variable that keeps track of the different `userIds` called and saves their results, thereby saving Joomla from repeating those queries over and over."

```
# Before fix.
$query->select($recursive ? 'b.id' : 'a.id');
$query->from('#__user_usergroup_map AS map');
    $query->where('map.user_id = '.(int) $userId);
$db->setQuery($query);
$result = $db->loadResultArray();
# After fix.
static $results = array();
$storeId = $userId . ':' .(int) $recursive;
if(!isset($results[$storeId]))
   $query->select($recursive ? 'b.id' : 'a.id');
   $query->from('#__user_usergroup_map AS map');
   $query->where('map.user_id = '.(int) $userId);
   $db->setQuery($query);
   $result = $db->loadResultArray();
   $results[$storeId] = $result;
```

Figure 4.3: *Joomla!* #20675 (Simplified)

```
# Before fix.
sub new{... $self->{'attachments'} =
   Bugzilla::Attachment::query($self->{bug_id});...}
# After fix.
sub attachments () {
  my ($self) = @_;
  return $self->{'attachments'} if exists $self->{'attachments'};
  $self->{'attachments'} =
     Bugzilla::Attachment::query($self->{bug_id});
  return $self->{'attachments'}; }
```

Figure 4.4: *BugZilla* #282145 (Simplified)

Iterative Queries (Total: 19, CV: 2.15). We observe that it is common for developers to execute queries within loops, whose total numbers of iterations may be input dependent. *BugZilla* #286625 mentioned in Chapter 2 Figure 2.1 is an example for Iterative Queries. Note that for the problem of *Iterative Queries*, the issued queries can return different results, so they cannot be simply cached.

Query Attribute: Size of Query Results (Total: 26, CV: 0.68). The size of query result-set instances can also have an impact on the performance of application-database interactions. Typically, as the size of a result-set instance increases, the performance overhead of fetching the data and transmitting the result-set instance data increases as well. We observe the two common bug types for this Query Attribute.

Unnecessary Queried Data (Total: 13, CV: 1.05). One common mistake that developers make is querying for more data than actually needed (e.g., `SELECT *` but not using all the column data). *BugZilla* #282145 (Figure 4.4) is an example. The developers initially put the initialization code of all fields of a bug entity in the constructor. However, not all fields are actually used every time in reality, making such all-field initialization a waste of database resources. In the patch, the developers introduce a method `attachments` to retrieve and cache the data for each field on-demand.

Large Result Set (Total: 13, CV: 0.72). Although it is possible that some of the data returned in result-set instances are unnecessary, we observe that in some cases the performance problem is introduced because the application logic requires using a large amount of data. The size of such result-set instances cannot be reduced without affecting functionality. Figure 4.5 shows

```
my $limit = $self->_params->{limit};
# Code added in the fix to specify a max upper bound.
my $max = Bugzilla->params->{'max_search_results'};
if (!$self->{allow_unlimited} && (!$limit || $limit > $max))
   $limit = $max;
```

Figure 4.5: *BugZilla* #632717 (Simplified)

an example of *BugZilla* #632717. A developer reports that "One frequently-reported problem is that you can do searches that return so many bugs that the search never actually completes". The final fix "set a maximum upper bound for how many results a search can return".

4.2.2 Query-Database Category

The symptoms of the performance bottlenecks for bugs in the *Query-Database* Category mainly come from the database's executions of query instances. The performance of a database's execution of query instances can be affected by the design of queries and database schemas. We observe that the design mismatch between queries and database behaviors commonly exists in all studied projects. In particular, bug reports with the *Query-Database* Query Attribute occur more frequently than those with the *Application-Query* Query Attribute. Specifically, *Missing Indexes* and *Expensive Operations* in queries are most commonly reported bug types.

Query Attribute: Expected Indexes (Total:45, CV: 0.57). Indexing plays an important role to speed up a database's processing of query instances. For example, *MySQL* provides a documentation [2] on how the DBMS leverages indexes to improve performance. In addition, one metric that developers commonly use to debug performance problems in database applications is what indexes are used in the corresponding databases. For the performance bugs in this Query Attribute, we observe three major bug types.

Missing Indexes (Total: 35, CV: 0.81). One common cause of performance bottlenecks of query processing in the database is that the developers forget to add certain indexes. However, the developers may not be aware of what the most effective index is. Worse still, a badly-created index may degrade performance. If a badly-created index is not leveraged during database ex-

```
# Before fix.
$query->where('(a.id = '.(int)$asset.($recursive ? 'OR
   a.parent_id=0':'').')');
# After fix.
$query->where('a.id = '.(int)$asset);
...
if ($recursive && empty($result))
  $query = $db->getQuery(true);
  $query->select('rules');
  $query->from('#__assets');
  $query->where('parent_id=0');
  $db->setQuery($query);
  $result = $db->loadResultArray();
```

Figure 4.6: *Joomla!* #25617 (Simplified)

ecution, the index can significantly slow down the performance of `INSERT`, `UPDATE`, and `DELETE` operations.

Indexes Not Leveraged (Total: 10, CV:0.97). Even if necessary indexes are in place for the database, an improperly designed query instance may cause the existing indexes not to be fully leveraged. Figure 4.6 shows an example from *Joomla!* #25617. The query condition shown in Line 1 is supposed to include both query criteria in one query. According to the developer comments, combining these two query criteria can prevent the query execution from using the index on column `id`, leading to bad performance when the database table grows large. The fix splits the original query condition into two different queries to solve this problem.

Query Attribute: Performance-Sensitive Operations (Total: 47, CV: 0.69). When constructing a query instance, a developer may specify certain operations inside the query string. In certain cases, the specified operations can incur performance overhead when the database is processing the query instances. For example, the clause `ORDER BY` in SQL instructs databases to sort queried data. As the size of data increases, the performance overhead of such operations can become significant. We identify two major bug types for the Query Attribute Performance-Sensitive Operation.

Unnecessary Operations (Total: 11, CV: 0.95). Bugs of this bug type are introduced by developers when specifying unnecessary operations in the query string when constructing the query. Figure 4.7 describes *WordPress* #12557, where an unnecessary `SQL_CALC_FOUND_ROWS` keyword instructs the

```
# This if condition is removed in the patch
if ( !empty($limits) )
# New if condition added in the patch
if ( !$q['no_found_rows'] && !empty($limits) )
   $found_rows = 'SQL_CALC_FOUND_ROWS';
```

Figure 4.7: *WordPress* #12557 (Simplified)

database to calculate the total number of rows satisfying a `SELECT` query without considering its `LIMIT` clause that is used to limit the size of result set. The fix adds an option to bypass the `SQL_CALC_FOUND_ROWS` operation.

Expensive Operations (Total: 36, CV: 0.67). Although some operations specified in a query string can be unnecessary, in some cases such operations are not avoidable unless the functionality is sacrificed. Such operations may be expensive and can cause performance bottlenecks. *Joomla!* #29845 mentioned in Chapter 2 (Figure 2.2) is an example of Expensive Operations.

4.2.3 Optimization Opportunities and Bug Types

Conceptually, optimization opportunities can be viewed as abstractions to the nine bug types. Relationship between R and APP suggests that the data returned and its usage may contain performance bugs. In particular, these performance bugs tend to belong to categories QD-UnD and QD-LaR. Relationship between APP and Q suggests that some query instances may be unnecessary based on the application logic. Such unnecessary query instances trigger performance bugs of category QN-UnQ. Relationship between R and R indicates that many returned result-set instances may be redundant. Since each result-set instance is associated with a query instance, many query instances should also be redundant as well. This optimization opportunity can be associated with a bug of category QN-ReQ. Relationship between Q and Q suggests that redundancies may exist between query instances. Although these query instances may not be associated with redundant result-set instances, these query instances may also be combined to a single query instance. Particularly, these query instances are likely issued iteratively: QN-ItQ. Relationship between Q and DB indicates that certain features in a single query instance may trigger performance problems in the database execution. More specifically, the query instance may contain unnecessary or expensive features (OP-ExO, OP-UnO) or the query may not fully use the existing index on database (IN-MiI, IN-NlI).

4.3 RQ2: Fix Strategies

In this section, we present our finding on categorization of fix strategies. We aim to study whether there exist common fix strategies for each bug type presented in RQ1. To identify and categorize fix strategies, we start with reasoning about the performance purpose of the fix for each of the four Query Attributes. For Number of Queries, the fix purpose should be reducing the number of query instances. For Size of Query Results, the fix purpose should be reducing the queried data. For Expected Indexes, the fix purpose should be modifying the database schema or existing indexes. For Performance-Sensitive Operations, the fix purpose should be optimizing the problematic query. When implementing a fix for each fix purpose, there may be different ways to change the code. We investigate the changes to the code, queries, or the database schema that occur in the fix for each bug report. Then, we group similar fixes together to obtain several fix strategies for each fix purpose. More specifically, we identify seven fix strategies: Bypassing Queries, Consolidating Queries, Limiting Results, Decomposing Queries, Performant Alternatives, Query Elimination, and Changing Indexes for the four fix purposes. Table 4.4 shows the fix strategies for each fix purpose.

4.3.1 Fix-Strategy Classification

Executing Fewer Queries. The fix strategy of Bypassing Queries (ByQ) introduces guard conditions and caching, in order to bypass the issuing of certain query instances. The fix strategy of Consolidating Queries (CoQ) modifies the application code and the query string to combine repetitive query instances. One common scenario is removing a loop, and batching the query issued within the loop with a single loop-free query that performs the same functionality. Figure 4.2 shows an example for ByQ while Figure 2.1 shows an example for CoQ.

Table 4.4: Fix Classification

Fix Purpose	**Fix strategy**	**Description**
Executing Fewer Queries	Bypassing Queries (ByQ)	Add conditions to bypass some queries.
	Consolidating Queries (CoQ)	Use fewer queries to load/modify data.
Querying Less Data	Limiting Results (LiR)	Modify query to reduce the amount of returned data.
Optimizing Queries	Decomposing Queries (DeQ)	Decompose a single slow query into several faster queries.
	Performant Alternatives (PeA)	Use alternative and more performant query operations.
	Query Elimination (ElQ)	Remove unneeded query parts or queries.
Optimizing Schema	Changing Indexes (ChI)	Add or remove indexes.

Table 4.5: Fix Strategies by Bug Classification

Classification	**ByQ**	**DeQ**	**CoQ**	**LiR**	**PeA**	**ElQ**	**ChI**	**Others**	**Mode**	**C-Index**
IN-MiI	-	-	-	2	-	-	32	1	ChI	91.42%
IN-NlI	-	2	-	1	1	3	2	1	ElQ	30.00%
OP-ExO	2	4	-	1	17	4	-	8	PeA	47.22%
OP-UnO	1	-	-	-	-	10	-	-	ElQ	90.91%
QN-ReQ	10	-	4	-	-	-	-	1	ByQ	66.67%
QN-ItQ	1	1	16	-	-	-	-	1	CoQ	84.21%
QN-UnQ	15	-	1	-	-	-	-	1	ByQ	88.23%
QD-LaR	-	2	-	7	-	-	-	4	LiR	53.85%
QD-UnD	-	2	-	8	-	-	-	3	LiR	61.54%
Others	1	-	1	-	2	-	1	20	-	-

Querying Less Data. For performance bugs with bug type Limiting Results (LiR), a corresponding fix strategy is to modify the functionality and query for less data. Typically, such fix strategy is achieved by adding or modifying the `LIMIT` clause in the original query. Figure 4.5 shows an example for LiR.

Optimizing Queries. Some complex subqueries can increase the performance cost of certain query operations. For example, `MySQL` experienced a case of poor performance [4] caused by the `IN` subquery, which would induce a high intermediate cost. We observe one common fix strategy, Decomposing Queries (DeQ), which avoids such subqueries by decomposing the original query string. *BugZilla* #819432 is such a case with some detailed discussions on the effect of decomposing queries. Although decomposing queries increases the number of query executions, the performance gain can still be much higher than the overhead where this strategy is applicable. For other expensive operations specified in a query string, the fix strategy of Performant Alternatives (PeA) substitutes these operations with efficient ones. In some cases, using a different type of `JOIN` clause can improve the query performance. For the unnecessary operations specified in a query string, the fix strategy of Query Elimination (ElQ) eliminates the unneeded and/or slow query parts or queries. Figure 4.7 shows an example of ElQ.

Optimizing Schemas. The fix strategy of Changing Indexes (ChI) modifies the existing indexes on the database so that the indexes can be fully leveraged when processing query instances.

4.3.2 Analysis

In this section, we study whether there are common fix strategies for each category of database-related performance bug.

Table 4.5 shows the statistics of the fix strategies for each bug type. Each row represents the data for each root cause. Specifically, each cell indicates, for the corresponding row (bug type), how many bug reports are fixed by the corresponding column (fix strategy). We refer to Mode as the most common fix strategy for a bug category while C-Index presents the percentage of bugs that are fixed by the Mode for that category.

The Mode of most bug types agrees with our expectation. Among the nine bug types, eight of them have C-Index greater than 50%, while five of them has C-Index greater then 70%. Therefore, we conclude that most of the bugs for each bug type are fixed by a common fix strategy, which is the corresponding Mode. The existence of a common fix strategy for a bug type may also suggest that it could be easier to develop tool support to fix bugs of that particular bug type.

The C-Index of bug type Indexes Not Leveraged(IN-NlI) is extremely lower than that of other bug types. A closer look at the bug reports reveals that the reason for preventing a query from leveraging existing indexes may vary greatly, and thus the corresponding fixes can also be very different from each other. For example, the query in *Joomla!* #25617 (Figure 4.6) has an `OR` statement, which prevents the existing index from being leveraged. The fix was to decompose the problematic query. The index in *Simple Machines* #3602 had an incorrect data type specified. As a result, some of the queries were unable to leverage that index. The fix was to modify the index. Due to this special property of IN-NlI, the fixes for this root cause spread evenly across may fix strategies.

4.3.3 Optimization Opportunities and Fix Strategies

Optimization opportunities can also be viewed as abstractions to the fixing strategies. For relationship between R and APP, the general fixing strategy of this optimization opportunity is to remove the performance bottlenecks caused by redundant data: LiR. For relationship between APP and Q, the common fixing strategy to optimize this optimization opportunity is to bypass these unnecessary query instances: ByQ. For relationship between R and R, common fix strategies aim to remove the redundancy by bypassing or consolidating these query instances: ByQ, CoQ. For relationship between Q and Q, a common fix is to combine these query instances: CoQ. For relationship between Q and DB, fixing these bugs requires changing the database index or removing the redundant features in query instances: ChI, ElQ.

4.3.4 Lessons Learned

In this section, we discuss lessons learned from studying how the developers fix the database-related performance bugs.

Diagnosing database-related performance bugs. We observe that profiling is widely used when diagnosing a database related performance bug. In particular, profiling can show the severity of the problem and provide information to aid developers while debugging. Furthermore, without profiling information, a developer typically hypothesizes the bug to be in one of the common bug types. But such hypothesis may not necessarily be correct. In *BugZilla* #851267, a developer comments that the database-related performance bug may be of bug type Large Result Set (QD-LaR) and suggests a fix of Limiting Results (LiR). In a later comment, another developer comments that "I did some profiling and there are major bottlenecks in the Voting code. I think I fixed them all in my patch". This comment reflects a number of places to be optimized, instead of a single one.

Related to the severity of a problem, we also observe that the level of acceptable performance can vary across different developers. In *BugZilla* #528918, a developer submits a patch that has 20% improvement: "Commenting this call out (see attached patch), takes us from 144ms to 110ms, or about 20%." However, another developer disagrees: "you're talking about the difference between 144ms and 110ms, a totally insignificant number to a Bugzilla user". We also observe that developers may not treat performance bugs as real bugs if the perceived performance problems are not severe enough, as commented by a developer in *BugZilla* #286625: "Let's take it on the 3.4 branch as it fixes a bug. And the huge perf problem can also be seen as a bug." Note that the first "bug" refers to another functional bug fixed by the proposed fix for the performance bug as a side benefit. In addition, some developers may not consider being slow as "broken": "nothing is broken here, just slow."

Fixing database-related performance bugs. Profiling is also frequently used by developers to demonstrate the performance boost after the fix. After the patch for *BugZilla* #851267, the developer posts the profiling results before and after fix and concludes that "numbers reported by Devel::NYTProf are divided by 2 with my patch. With Devel::NYTProf disabled, page.cgi now takes 0.9 second to load instead of 3.0 seconds. That's a huge win". In fact, developers may explicitly ask for profiling results when deciding whether or not to accept a fix for database-related performance bugs: "Could you show some statistics for how this helps performance on your sites?".

We also observe that when deciding whether a fix should be accepted, developers consider many factors other than the performance boost of the fix. One important factor is security. Developers may not accept a fix that may cause security issues, even it can boost the performance substantially. For *BugZilla* #528918, one developer submits a patch that can boost the performance by 20%. However, this patch is not accepted and another developer comments "No, definitely not, this regresses a major security fix–a severe SQL injection in the WebService". Another factor is the design quality and the software architecture. Developers tend to be careful in not compromising the design quality when fixing performance bugs (would be willing to do that only when the benefits, i.e., performance improvement, are substantial): "No, I'm against putting it too far up in the call chain, because that adds complexity. If there's a *significant* performance impact, I'd consider it, but for minor performance differences I don't think the complexity would be justified". Another developer also explicitly mentions that "The most important thing is profiling and good, maintainable architecture. That will lead to good performance".

Tool support for diagnosing and fixing database-related performance bugs. We observe that there exist several tools that are commonly used by developers when diagnosing and fixing database-related performance bugs. Slow Query Log[1] is often used to detect queries that are slow. The developers can specify a time limit and then the log can report queries that take more than the specified time to execute. EXPLAIN[2] is also commonly used to obtain information about how the database executes the query. Typically, developers use EXPLAIN to observe whether a particular index is leveraged during the execution of the query. EXPLAIN can help developer debug and fix database-related performance bugs that have query attribute Expected Indexes (IN).

Although these tools provide useful information to the developers regarding the cause of the performance bugs and the potential fixes, these tools cannot

[1]`http://dev.mysql.com/doc/refman/5.7/en/slow-query-log.html`
[2]`http://dev.mysql.com/doc/refman/5.7/en/explain.html`

Table 4.6: Elapsed Time and Involved Developers for each Bug Type

Bug Type	Avg Time	Avg Time Closed	Avg # Devs
IN-MiI	222.69	194.65	4.31
IN-NlI	156.10	232.00	4.10
OP-ExO	316.94	238.33	4.58
OP-UnO	39.55	17.43	3.00
QD-LaR	213.77	345.17	5.38
QD-UnD	205.85	301.71	4.46
QN-UnQ	253.90	100.01	3.69
QN-ReQ	273.00	18.75	3.06
QN-ItQ	394.26	758.00	3.47

automate the bug detection and fixing process. Developers still need to discuss with each other and manually diagnose or fix the performance bug. Therefore, there are still opportunities for new techniques along with tool supports that can improve the efficiency of bug detection and bug fixing.

To demonstrate an example benefit of our study, we implement a simple rule checker to statically detect the occurrences of database-related performance bugs caused by QN-ItQ. Our checker performs static code checking on PHP code to detect queries that are issued inside a loop. We apply the checker on the source code of *Joomla!* and are able to identify nine potential Iterative Queries bugs. In practice, this checker can be used during development to notify developers of potential bugs in their code. We describe the implementation and evaluation details on our project website [3].

4.4 RQ3: Time and Developers

In this section, we report the results for RQ3.

4.4.1 Elapsed Time

The time elapsed before fixing a bug in a bug report is measured in days and is calculated from the date when the bug report is created to the date when the bug report is closed. Depending on the used bug tracking system, a bug report may be (1) closed with an explicit closing date, (2) closed but without an explicit closing date, or (3) not closed. To determine how much time elapses before a bug is fixed, we consider only closed bug reports. If a closed bug report contains an explicit closing date, we use that date to calculate the time of fixing. Otherwise, we use the date of the last comment in the developer's discussions related to bug fixing as the closing date. We report the calculated time for both types of bug reports. For our seven subjects, *DNN*, *Joomla!*, *WordPress*, and *Roundcube* have explicit indication of closing date in their bug reports. Note that 13 out of the 183 bug reports in our subject are not closed. Please refer to the project website [3] for more information on these bug reports.

We observe that developers may post some comments not relevant to the bug fixing a long time after the bug was fixed. To identify the last comment, we filter out these irrelevant comments. Typically, comments fall into the following categories:

- Some other bugs marked as duplicate of the current bug.
- The details of the current bug added to the release note when a new update is released.
- The current bug mentioned in other discussion.

The consequence of this design decision is that the measurements for some bug reports may not reflect the actual time elapsed prior to the fix. In the case of no closed date specified, deciding which developer comment suggests that the bug report is closed may be a subjective judgment. To ensure objectivity, we enforce the design decision to determine the time spent for all the bug reports without a clearly marked closed date.

In Table 4.6, column *Avg Time* shows the average time elapsed for each bug type, while column *Avg Time Closed* shows the average time elapsed for bug reports with explicit closed date only. The values show that bugs of type Iterative Queries (QN-ItQ) tend to have a larger average value while bugs of type Unnecessary Operations (OP-UnO) have a smaller average value.

Longer time elapsed before fixing Iterative Queries (QN-ItQ) bugs may be that these bugs are harder to diagnose. For example, the developers in *BugZilla* #286625 incorrectly diagnosed the performance bug as Missing Index (IN-MiI) and spent a large amount of time fixing the bug by changing index. After a few years, the developers finally discovered the real cause: "As usual, what is ACTUALLY slow is not what you might think", and fixed the performance bug: “This makes –regenerate take minutes instead of hours or days” (regenerate is an option that can be passed in while executing the code). We also observe that some of the Iterative Queries (QN-ItQ) bugs are of low priority. The comments suggest that the developers might not have worked on these bugs in a timely manner because of the low priority. Although the time elapsed before fixing is used as a metric for bug fixing difficulty in an earlier empirical study [11], a longer time elapsed may also be caused by a bug report having lower priority. Some additional analysis may be needed to use the time elapsed as a metric for fixing difficulty.

4.4.2 Number of Developers Involved

In Table 4.6, column *Avg # Devs* shows the average number of developers that took part in the discussion for each bug report across all bug types. The average values of the number of developers across all bug types are very close to each other: the majority of average values lie between 3 and 5.

We hypothesize that the reason why the numbers of developers for each bug type are very similar is that there is a certain fixed number of regular developers and that the number of regular developers is very close to the average number of discussion-participating developers whom we observe. For example, in the bug reports that we study, these regular developers could be the developers who are mainly responsible for implementing the database interactions. Each of the regular developers took part in the discussion of a majority number of the bug reports. For each bug report, it is usually reported by an arbitrary user or developer and discussed by a few of the regular developers. The number of regular developers that took part in the discussion may vary for each bug type, but may not vary significantly.

To validate our hypothesis, we investigate the number of regular developers for each project. For each project, five or six regular developers commented in from 82.14% to 92.31% of the collected bug reports for that project. Due to the small number of regular developers, if a bug that is harder to fix, it may attract only one or two more regular developers into the discussion. Therefore, the resulting average numbers of developers in the discussion may be very similar across all bug types and the differences in the average numbers may be only around one developer. The detailed statistics of regular developers and some other statistics related to RQ3 can be found on the project website [3].

4.5 Threats To Validity

The validity of our study results may be subject to several threats. The first is the representativeness of our selected database applications. To minimize this threat, we select seven popular real-world open-source database applications, which cover different categories. The second threat is that we may miss relevant bug reports during our search for database-related performance bugs. We mitigate this problem by using keyword search together with bug categories and tags. We also search the bug description and comments in addition to the bug report summary since developers tend to use common terms in the description and comments. The third threat is related to our manual inspection of the collected bugs reports. The manual inspection is independently performed by at least two authors to alleviate this issue. If the two authors have different opinions on a bug report, multiple inspectors discuss the bug report to reach an agreement. We follow this process from a previous empirical study on performance bugs [9]. For bug reports without publicly available source control information, there are only five bug reports (one for *Simple Machines*, four for *DNN*) for which we cannot locate the source code for the bugs. The percentage of such bug reports is very small given that we study 183 bug reports in total.

CHAPTER 5

RELATED WORK

Performance Empirical Study. There are multiple pieces of work on studying real-world performance bugs. Jin *et al.* [9] studied 109 performance bugs that were randomly sampled from five open-source software suites. In particular, they studied the root causes of performance bugs, how the bugs were introduced, how the bugs manifested, and how the bugs were fixed. In addition, they proposed a rule-checking-based detection approach for common performance bugs.

Nistor *et al.* [14] studied the detecting and fixing process of performance bugs. They particularly investigated the difficulties in fixing performance bugs versus fixing non-performance bugs. They also investigated how performance bugs are discovered by developers.

Liu *et al.* [11] studied 70 performance bugs collected from popular Android applications. They categorized these performance bugs by common causes. They also investigated the debugging and fixing effort, and proposed an approach that could detect two categories of these bugs.

Performance Bug Detection. Xiao *et al.* [20] proposed a technique to detect performance bottlenecks that are input-dependent. In particular, the proposed approach detects loops with slow operations that are sensitive to the input workloads. Nistor *et al.* [15] developed Toddler, an approach to detect loops causing repetitive memory-access patterns. These repetitive accesses are likely to be unnecessary, and may be optimized. Nguyen and Xu [13] proposed a run-time profiling approach to detect operations that keep producing identical values. Such identical values expose caching opportunities for removing memory bloats.

Performance of Database Applications. Manjhi *et al.* [12] proposed an approach to transform database queries to reduce latency in web applications. To this end, they proposed two transformations: merging and nonblocking, to reduce and re-schedule queries. They evaluated their approach on three benchmarks and identified various optimization opportunities.

Bowman and Salem [6] proposed the Scalpel system to optimize database queries by prediction. In particular, the Scalpel system monitors queries in a query stream to detect optimization opportunities. Scalpel then chooses an optimal strategy to re-write the query. The modified query has a lower latency and a lower cost of query evaluation.

Ramachandra and Sudarshan [17] improved the existing approaches on prefetching query results by introducing an approach to identify the earliest program location where a query result could be prefetched. Prefetching at the earliest program location can boost the benefit of prefetching since the query result would likely be available when it is actually needed. The proposed approach performs an inter-procedural data-flow analysis to identify such program locations, and rewrites the queries to perform prefetching.

Cheung *et al.* [8] proposed Sloth, an approach to reduce round-trip latency for database queries by extending traditional lazy evaluation. As the application executes, Sloth delays the execution of queries until their results are needed. These delayed queries are kept in a query store, and executed later in a batch, thus reducing round-trip latency.

Cheung *et al.* [7] proposed Pyxis, an approach to automatically partition database application code to extract stored procedures. Invoking stored procedures can reduce query latency since these procedures are running on the database server. Pyxis leverages both static and dynamic analyses to collect relevant application data. The data is used to formulate a linear program, whose objective is to minimize latency. The solved linear program produces a partitioning of the original application code.

CHAPTER 6

CONCLUSION

In this thesis, we have conducted an empirical study of 183 bug reports for database-related performance bugs collected from seven real-world open-source projects: *BugZilla*, *DNN*, *Joomla!*, *MediaWiki*, *WordPress*, *Simple Machines*, and *Roundcube*. We have studied the common optimization opportunities, types of database-related performance bugs, time elapsed before fixing these bugs, and number of developers involved in the discussions prior to the fixes.

In particular, we have identified nine common bug types and seven common fix strategies. We have studied the characteristics of each bug type and fix strategy. Our findings can provide guidance to avoid and diagnose database-related performance bugs, and to develop new bug-detection techniques. More information of our study such as evaluation results is available on our project website [3].

REFERENCES

[1] 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`.

[2] How MySQL Uses Indexes. `https://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html`.

[3] Project Website. `https://sites.google.com/site/databaseperformancestudy`.

[4] Restrictions on Subqueries. `http://dev.mysql.com/doc/refman/5.5/en/subquery-restrictions.html`.

[5] Simple Machines Member Statistics. `http://www.simplemachines.org/community/index.php?action=stats`.

[6] I. T. Bowman and K. Salem. Optimization of query streams using semantic prefetching. *ACM Trans. Database Syst.*, 30(4):1056–1101, Dec. 2005.

[7] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *Proc. VLDB Endow.*, 5(11):1471–1482, July 2012.

[8] 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*, SIGMOD '14, pages 931–942, New York, NY, USA, 2014. ACM.

[9] 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*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[10] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM.

[11] 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*, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.

[12] 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*, ICDE '09, pages 1175–1178, Washington, DC, USA, 2009. IEEE Computer Society.

[13] 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*, ESEC/FSE 2013, pages 268–278, New York, NY, USA, 2013. ACM.

[14] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 237–246, Piscataway, NJ, USA, 2013. IEEE Press.

[15] 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*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[16] 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*, ESEC/FSE 2013, pages 125–135, New York, NY, USA, 2013. ACM.

[17] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 133–144, New York, NY, USA, 2012. ACM.

[18] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[19] L. Woods, J. Teubner, and G. Alonso. Less watts, more performance: an intelligent storage engine for data appliances. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1073–1076, New York, NY, USA, 2013. ACM.

[20] 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*, ISSTA 2013, pages 90–100, New York, NY, USA, 2013. ACM.