MODELING AND ANALYSIS OF SQL QUERIES IN PHP SYSTEMS

by

David Anderson

April, 2018

Director of Thesis: Mark Hills, PhD

Major Department: Computer Science

PHP is a common language used for creating dynamic websites. These websites often include the use of databases to store data, with embedded SQL queries constructed within the PHP code and executed through the use of database access libraries. One of these libraries is the original MySQL library that, despite not being supported in current versions of PHP, is still widely used in existing PHP code. As a first step towards developing program comprehension and transformation tools for PHP systems that use this library, this research presents a query modeling tool that models embedded SQL queries in PHP systems and an empirical study conducted through analysis of these models. A main focus of this study was to establish common patterns developers use to construct SQL queries and to extract information about their occurrences in actual PHP systems. Using these patterns, the parts of queries that are generally static, and the parts that are often computed at runtime were extracted. For dynamically computed query parts, we also extracted data about which PHP language features are used to construct them. Finally, information about which clauses most often differ based on control flow was extracted as well as counts for how often each SQL query type and SQL clause is used in practice. We believe this information is useful for future work on building program understanding and transformation tools to renovate PHP code using database libraries.

MODELING AND ANALYSIS OF SQL QUERIES IN PHP SYSTEMS

A Thesis

Presented to The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Software Engineering

by

David Anderson

April, 2018

Copyright David Anderson, 2018

MODELING AND ANALYSIS OF SQL QUERIES IN PHP SYSTEMS

by

David Anderson

APPROVED BY:

DIRECTOR OF THESIS:	Mark Hills, PhD
COMMITTEE MEMBER:	Junhua Ding, PhD
COMMITTEE MEMBER:	Nasseh Tabrizi, PhD
CHAIR OF THE DEPARTMENT OF COMPUTER SCIENCE:	Venkat Gudivada, PhD
DEAN OF THE GRADUATE SCHOOL:	Paul J. Gemperline, PhD

Table of Contents

LIST OF TABLES vi

LIST OF FIGURES vii

1 INTRODUCTION 1

Research Contribution: 2

2 TOOLS AND TECHNOLOGIES 4

3 SQL MODELING 6

3.1 Overview 6

3.2 Model Building 8

3.3 Yield Extraction 12

3.4 Parsing 14

4 ANALYSIS OF SQL MODELS 17

4.1 The Corpus 18

4.2 Query Construction Patterns 20

4.2.1 QCP0: Literal Query Strings 20

4.2.2 QCP1: Dynamic Parameters 22

4.2.3 QCP2: Dynamic 23

4.2.4 QCP3a: Multiple Yields, Same Parsed Query 26

4.2.5 QCP3b: Multiple Yields, Same Query Type 28
4.2.6 QCP3c: Multiple Yields, Different Query Types 29
4.2.7 QCP4: Function Queries 31
4.2.8 Query Construction Pattern Results 33
4.3 Query Fragment Categories 35
4.4 Model Yield Comparison 37
4.5 SQL Clause Usage 45
5 RELATED WORK 48
6 FUTURE WORK AND CONCLUSION 52
6.1 Future Work 52
6.2 Conclusion 54
BIBLIOGRAPHY 55

LIST OF TABLES

4.1 The Corpus. 19
4.2 QCP Counts by System. 34
4.3 Fragment Category Counts by System. 36
4.4 Clause Comparison Counts for Pattern QCP3b. 42
4.5 Query Type Counts by System. 43
4.6 Clause Counts for each Query Type. 44

LIST OF FIGURES

3.1 Overview: Extracting Query Models and Parsing Modeled Queries. . . 6

3.2 Model Building Example: PHP Code. 8

3.3 Query Fragment Constructors in PHP AiR. 8

3.4 FragmentRel Definition in PHP AiR. 11

3.5 Model Building Example: Dot Graph. 13

3.6 Model Building Example: Yields. 13

3.7 Model Building Example: Query Strings. 14

3.8 Model Building Example: SQL ASTs. 14

3.9 Dot Graph for a Query with Dynamic Query Text. 15

3.10 Dynamic Query Text: Yield, Query String, and AST. 16

4.1 QCP0: Example 1. 21

4.2 QCP0: Yield, Query String, and AST. 21

4.3 QCP0: Example 2. 22

4.4 QCP1: Example 1. 23

4.5 QCP1: Yield, Query String, and AST. 24

4.6 QCP2: Example 1. 25

4.7 QCP2: Yield, Query String, and AST. 25

4.8 QCP3a: Example 1. 27

4.9 QCP3a: Yields, Query String, and AST. 27

4.10 QCP3b: Yields, Query Strings, and ASTs. 30
4.11 QCP3c: Yields, Query Strings, and ASTs. 32
4.12 QCP4: Example 1. 33
4.13 QCP4: Yield, Query String, and AST. 33
4.14 YieldInfo Rascal Definition. 38
4.15 ClauseComp Construction: State Machine Diagram. 39
4.16 YieldInfo example: SQL Strings. 40
4.17 YieldInfo example: YieldInfo. 40

6.1 Proposed Transformation Approach. 53

Chapter 1

Introduction

PHP is a server-side scripting language used for building dynamic web applications. PHP is widely used, ranking fifth in popularity on GitHub in terms of pull requests.[1] In dynamic web applications, the data displayed to the user often depends on information stored in relational databases. Furthermore, user inputs are often sent to a database where records are created or modified. To support database interaction, PHP provides a number of database access libraries such as MySQL,[2] MySQLi[3] and PDO.[4] When using database libraries, queries are built using a mixture of static SQL text and dynamic inputs. These queries are then sent to be executed by the database using an API call (such as `mysql_query` in the MySQL library). In this research, we examine PHP systems using the original MySQL library. This library was deprecated in PHP 5.5.0 and completely removed in PHP 7. Despite this, many systems, including some still in active use/development, use this library. The end goal of this research is to create program comprehension and transformation tools for renovating PHP systems to use safer, more modern database access libraries. A major roadblock to creating such tools is the dynamic nature of the PHP language that makes it difficult to reason about statically [1], [2]. Therefore, this research focuses on answering

[1] https://octoverse.github.com/
[2] http://php.net/manual/en/book.mysql.php
[3] http://php.net/manual/en/book.mysqli.php
[4] http://php.net/manual/en/book.pdo.php

the following research questions to provide a foundation for future work on program comprehension and transformation tools.

- **RQ1:** What patterns do developers commonly use to construct database queries?
- **RQ2:** Which parts of queries are generally static? Where do dynamic parts usually occur in queries?
- **RQ3:** Which PHP language features do the data from dynamic query parts come from?
- **RQ4:** In cases where different control flow paths lead to different queries, which clauses are the same across yields, and which are different?
- **RQ5:** Which SQL clauses are most often used in practice? Which are hardly used?

The rest of this thesis is organized as follows. In Chapter 2, we give a brief overview of the tools and technologies used in this project, as well as links to the repositories for our query modeling and analysis tool. In Chapter 3, our query modeling and analysis tool is described in detail. In Chapter 4, we present the process of applying our query modeling tool on a corpus of open source systems, the empirical data obtained from these systems, and what these results mean in terms of the 5 previously discussed research questions. In Chapter 5, we consider other research projects related to analysis of embedded SQL queries. Finally, in Chapter 6, we provide areas for future work and conclusion.

Research Contribution: The empirical study discussed in this work presents quantitative data on how developers construct database queries in PHP systems, which PHP language features they use, and which types of queries and query clauses are used most often in practice. This analysis is designed to enable future work on building program understanding and transformation tools to renovate existing PHP

code using database libraries. To ensure repeatability of this analysis, all results, percentages, and LaTeX tables shown in this work can be generated using the functions found in our repository, which is publicly available and can be downloaded using the links provided in Chapter 2.

Chapter 2

Tools and Technologies

The analysis tool developed for this research is written in Rascal [3], a meta-programming language designed to cover the entire meta-programming domain, with main focuses including program analysis and transformation. The PHP AiR (PHP Analysis in Rascal) framework [4], [5] is used as the foundation on which our analysis tool is built. The goal of this framework is to enable empirical software engineering and program analysis of PHP systems. This framework makes use of Rascal's high-level data types such as sets, maps, relations, and ADTs, and language features such as pattern matching and source code locations, which we use heavily in our own scripts. This framework also provides support for parsing PHP code from versions 5.2 through 7.2, which is enabled through a fork of an open source PHP parser. The repositories for PHP AiR and the PHP parser are freely available and can be found at `https://github.com/cwi-swat/php-analysis` and `https://github.com/cwi-swat/PHP-Parser` respectively. Our SQL modeling tool builds on the PHP AiR framework to enable analysis of SQL queries in PHP systems.

The main repository for our SQL analysis tool can be found at `https://github.com/ecu-pase-lab/mysql-query-construction-analysis`. Our modified SQL parser can be found at `https://github.com/ecu-pase-lab/sql-parser`. Detailed descriptions of these tools and how they were used to extract empirical data can be

found in Chapters 3 and 4, respectively.

Chapter 3

SQL Modeling

3.1 Overview

Figure 3.1 provides an overview of the process used to build a model of a specific `mysql_query` call, extract all possible yields of the model, and parse the yields into MySQL Abstract Syntax Trees (ASTs), which captures the structure of the queries. First, the PHP scripts are input into our PHP parser which outputs a Rascal term of type `System`. A `System` includes the AST for each PHP script, as well as information such as the name and version number of the system being analyzed. Next, the

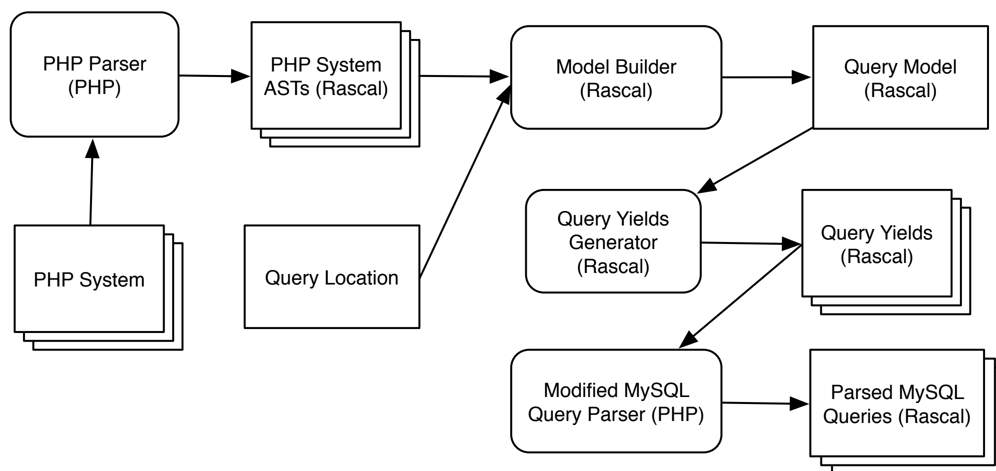

Figure 3.1: Overview: Extracting Query Models and Parsing Modeled Queries.

`mysql_query` call location, giving the position of the call in a specific script, and the `System` are input into our Query Model Builder. This outputs a value of type `SQLModel`. A `SQLModel` value represents all possible SQL queries that could be passed into the call at the provided location. This model differentiates between static and dynamic parts of the query. For each name in the query, the model provides links to the corresponding definitions for the name, with information on each edge showing under what conditions each part of a query is present. Once the models are built, they are input into our Query Yields Generator, which generates a set of all possible queries that the model for a particular call can yield. The yields are then converted to strings with "query hole" tokens representing the dynamic parts of the query. These query strings with hole placeholders are then input to our modified MySQL query parser which parses the query strings and returns SQL ASTs (as Rascal terms) for each parsed string. The final result is a set of parsed queries representing all possible queries that could be input to a particular call to `mysql_query`. In the rest of this chapter, each of these steps are discussed in more detail. In Section 3.2, the model building process is described. In Section 3.3, the process of extracting yields from query models is described. Finally, in Section 3.4, the process for parsing query strings with dynamic holes is explained. This chapter follows an example from lines 64-89 of *html/main.inc.php* in system MyPHPSchool (see Section 4.1 for a description of the corpus). Figure 3.2 shows the PHP code (with comments removed for clarity) that serves as the basis for this example. In each of the following sections, the output of that stage of the analysis on this example is shown.

```
1 function check_auth () {
2         global $cookie_user, $cookie_enc, $db, $base_url,
             $sbase_url, $cookie_type;
3
4         if($cookie_user  && $cookie_enc && cookie_type){
5
6                 $cookie_user = clean_var($cookie_user);
7                 $cookie_type = clean_var($cookie_type);
8
9                 if($cookie_type == "staff"){
10                        $type = "staff";
11                } elseif($cookie_type == "student") {
12                        $type = "student";
13                } else {
14                        die("INVALID USER TYPE SPECIFIED!");
15                }
16
17                $sql = "SELECT * FROM user_$type WHERE
                     username = '$cookie_user'";
18                $result = mysql_query($sql, $db);
19        ...
```

Figure 3.2: Model Building Example: PHP Code.

```
literalFragment(str literalFragment)
compositeFragment(list[QueryFragment] fragments)
concatFragment(QueryFragment l, QueryFragment r)
nameFragment(Name name)
dynamicFragment(Expr fragmentExpr)
globalFragment(Name name)
inputParamFragment(Name name)
unknownFragment()
```

Figure 3.3: Query Fragment Constructors in PHP AiR.

3.2 Model Building

Calls to `mysql_query` accept a PHP expression as a parameter. This expression should evaluate to a SQL query string. This can either be a static string, or, more commonly, a mixture of static SQL text and dynamic inputs from variables and

other expressions. Furthermore, the assignments to these dynamic query fragments may differ based on control flow, resulting in multiple distinct SQL queries that can be input to a particular query call. In our tool, we model SQL queries as a relation over `QueryFragment` Rascal terms that represent each possible expression that can contribute to the query at a particular call to `mysql_query`. Figure 3.3 shows the Rascal definition for `QueryFragment`. A `QueryFragment` can be either a static string (`literalFragment`), string interpolation with embedded PHP expressions (`compositeFragment`), a string concatenation of multiple query fragments (`concatFragment`), the use of a name (`nameFragment`), the use of a function/method parameter (`inputParamFragment`), the use of a global name (`globalFragment`), another PHP expression that we have not modeled explicitly (`dynamicFragment`), or a fragment with no definitions (`unknownFragment`). As a first step to translating a SQL expression into a query fragment, we first perform a number of simplifications to the expression. This includes simulation of library functions, replacing constants with their values, and performing concatenation of string literals. For example, after simplification, the PHP string concatenation: `$x = "select *". "from houses". "where floor = 'carpet'";` would result in the the fragment: `literalFragment("select * from houses where floor = 'carpet")` rather than: `concatFragment(literalFragment ("select *"), concatFragment(literalFragment("from houses"), literalFragment ("where floor = 'carpet'")))`.

Algorithm 1 shows the process for building a `SQLModel` representing all possible SQL queries that could be passed to a particular query call. The inputs to the system are the system AST (*sys*) for the system containing the specified query call and the location of the desired query call to be modeled (*callLoc*). The procedure also makes used of a system's `QCPSystemInfo`, which contains cached analysis results such as control flow graphs and use/def information, which records assignments to (definitions

Algorithm 1: Extracting a Model of a SQL Query.

Input : *sys*, a PHP system, mapping from file locations to abstract syntax trees
Input : *callLoc*, a location indicating the query call to be analyzed
Output: *res*, a query model

```
1  inputCFG ← buildCFG4Loc (callLoc)
2  inputNode ← the CFG node from inputCFG representing the call at callLoc
3  d ← definitions (inputCFG)
4  u ← uses (inputCFG, d)
5  slicedCFG ← basicSlice (inputCFG, inputNode, usedNames (u, inputNode),
     d, u)
6  startingFragment ← expr2qf (inputNode)
7  fragmentRel ← {}
8  while fragmentRel continues to change do
9  |  nodesToExpand ← (inputNode.l × startingFragment) ∪ fragmentRel⟨3, 4⟩
10 |  foreach (nodeLabel × nodeFragment) ∈ nodesToExpand do
11 |  |  add expandFragment (nodeLabel, nodeFragment, d, u) to fragmentRel
12 |  end
13 end
14 fragmentRel ← addEdgeInfo (fragmentRel, slicedCFG)
15 res ← the model, including fragmentRel, startingFragment, and callLoc
```

of) a name and uses of specific definitions. This is a performance improvement that is not shown in Algorithm 1. The procedure begins by building an intraprocedural control-flow graph for the script, function, or method containing the query call at *callLoc* and assigns it as *inputCFG* (line 1). Next, the CFG node representing the call at *callLoc* is assigned to *inputNode* (line 2). Next, the definitions contained in the *inputCFG* are computed (line 3) along with the uses of those definitions (line 4). Using the *inputCFG*, *inputNode*, and the definitions/uses, a backwards, intraprocedural slice, starting at the query call, is computed that throws away any CFG nodes that do not affect the value of the query being passed into the call at *callLoc* (line 5).

The rest of Algorithm 1 involves the computation of the model's `FragmentRel`. The Rascal definition of `FragmentRel` can be found in Figure 3.4. This relation

```
alias FragmentRel = rel[Lab sourceLabel, QueryFragment
   sourceFragment, Name name, Lab targetLabel, QueryFragment
   targetFragment, EdgeInfo edgeInfo];
```

Figure 3.4: FragmentRel Definition in PHP AiR.

maps a name in a given CFG node (*Name name*), the unique label of that CFG node (*Lab sourceLab*) and the `QueryFragment` representing that name (*QueryFragment sourceFragment*) to the label of another CFG node (*Lab targetLabel*) and the *QueryFragment* corresponding to that CFG node (*QueryFragment targetFragment*). Optional information about under what conditions the name given by *name* can take on the value given by *targetFragment* is provided in the *EdgeInfo* field as a set of predicates. The `FragmentRel` can be viewed as a directed graph with each graph node being a (*Lab*, *QueryFragment*) pair and each edge connecting a name used in the `QueryFragment` to one of its definitions, with optional *EdgeInfo* describing the conditions under which the definition in *targetFragment* is assigned to the name in *sourceFragment*.

In the first step of computing the `FragmentRel` the *inputNode* is converted to a `QueryFragment` which is assigned to the *startingFragment* field of the `SQLModel` (line 6). This corresponds to the expression used as the parameter to the query call. From here, for each name in the *startingFragment*, *expandFragment* uses the def-use information to return mappings to other fragments that define these names. Names used in these new fragments are then mapped to new fragments that define them. This fixpoint process continues until new mappings are no longer added to `FragmentRel`. For names where no definitions could be found, a mapping to an `unknownFragment` is added. Names that map to formal parameters or global declarations expand to `inputParamFragment`s and `globalFragment`s, respectively.

The final step to computing the `SQLModel` is to add information to each edge

in the `FragmentRel` that describes which conditions must hold for each edge to be reached. This is achieved in *addEdgeInfo*, where the *slicedCFG* is used to decorate each edge in the `FragmentRel` with predicates (Line 14). Finally, the `FragmentRel` (with `EdgeInfo`), the query call location, and the starting fragment are returned (line 15). This final result represents all possible queries that can be passed into the query call. Figure 3.5 shows a dot graph representation of the model built for the code example in Figure 3.2. For the set of edge conditions, each comma can be viewed as an OR. We do not attempt to filter out tautologies in the predicate sets (i.e. `$cookie_type == ‘staff’`, `!($cookie_type == ’staff’)`. In this example, the `$sql` name is the *startingFragment* of the model. This name is mapped to a *compositeFragment* that provides the definition to this name, under the conditions labeled on the edge. In this *compositeFragment*, two new names are introduced: `$type` and `$cookie_user`. `$type` is then mapped to two possible literal fragments: “student” and “staff”. This corresponds to the two possible assignments in the if-statement shown in the lines 9-12 from the code example in Figure 3.2.

3.3 Yield Extraction

The procedure described in Algorithm 1 returns a `SQLModel` representing all possible queries that can be passed into a particular query call. From here, the first step to generating actual queries from the model is to compute the *yields* of the model. In Rascal, a `SQLYield` is defined as a list of `SQLPiece` instances. Each `SQLPiece` is either a `staticPiece`, for literal query fragments in the model, a `namePiece`, for name fragments, or a `dynamicPiece`, for dynamic fragments that are not names. Each yield corresponds to a path through the `FragmentRel` of the model starting at the *startingFragment*. Paths that lead to cycles due to names that are defined in terms

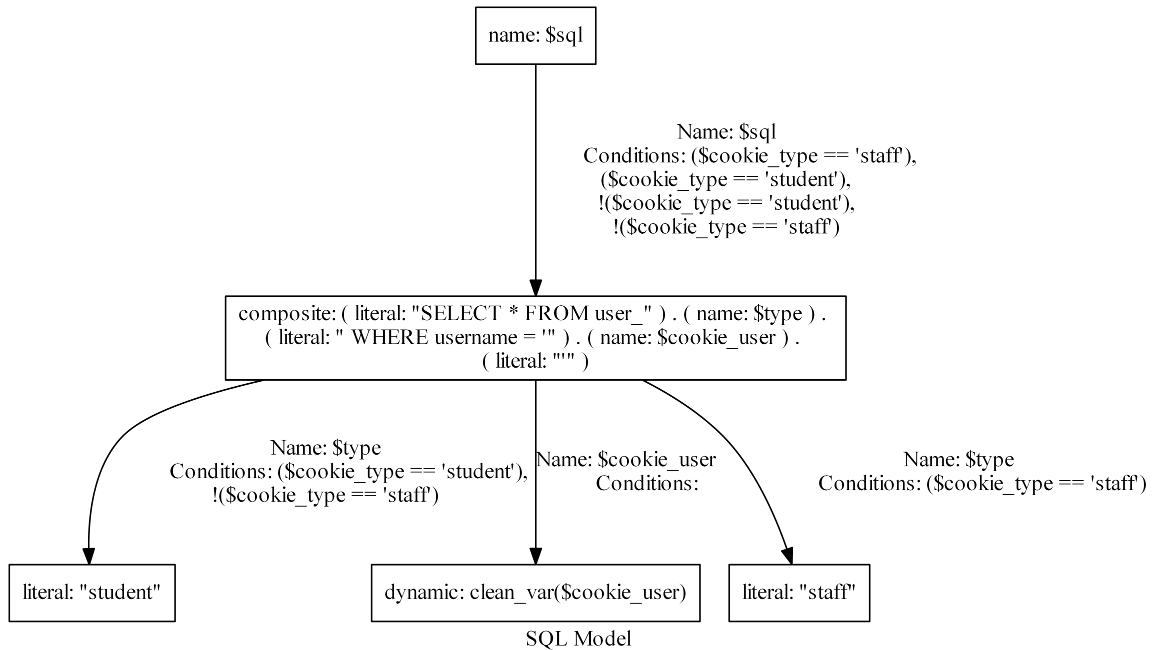

Figure 3.5: Model Building Example: Dot Graph.

```
[staticPiece("SELECT * FROM user_student WHERE username = \'"
   ),namePiece("cookie_user"),staticPiece("\'")]
[staticPiece("SELECT * FROM user_staff WHERE username = \'"),
   namePiece("cookie_user"),staticPiece("\'")]
```

Figure 3.6: Model Building Example: Yields.

of themselves are cut off. This leaves these names unexpanded, which could lead to a potential source of incompleteness where, in these cyclic cases, some possible queries could be missed. Finally, infeasible yields are removed using the condition information on each edge. Figure 3.6 shows the two yields computed for the model from Figure 3.5. The first yield corresponds to the program path where `$cookie_type == "student"`, and `$type` is mapped to the literal fragment “student”. The second yield corresponds to the program path where `$cookie_type == "staff"` (meaning `$cookie_type == "student"` is false), and `$type` is mapped to the literal fragment “staff”. Since the name `$cookie_user` is mapped to a dynamic fragment, it is unexpanded, and remains

```
SELECT * FROM user_student WHERE username = '?0'
SELECT * FROM user_staff WHERE username = '?0'
```

Figure 3.7: Model Building Example: Query Strings.

```
selectQuery([star()],[name(table("user_staff"))],where(
   condition(simpleComparison("username","=","?0"))),
   noGroupBy(),noHaving(),noOrderBy(),noLimit(),[])
selectQuery([star()],[name(table("user_student"))],where(
   condition(simpleComparison("username","=","?0"))),
   noGroupBy(),noHaving(),noOrderBy(),noLimit(),[])
```

Figure 3.8: Model Building Example: SQL ASTs.

in each yield as an undefined name piece. This set of yields corresponds to all feasible paths through the `SQLModel` that result in actual queries that could reach the query call.

3.4 Parsing

In Section 3.3, the process for extracting yields from a `SQLModel` was shown. The next step is to convert these yields into SQL strings. To convert `SQLYield`s to strings, we defined the following conversion. If a `SQLPiece` is a `staticPiece`, the literal query fragment is added to the string. For `namePiece`s and `dynamicPiece`s, a `?` followed by an integer (e.g. `?0`, `?1`, `?2`, ...) is added to the string. This is referred to as a "query hole symbol" and represents a part of the query where dynamic information will be produced at runtime. This can be seen in Figure 3.7, where the yields from Figure 3.6 have been converted to SQL strings. In this example, there are two SQL strings corresponding to the two yields generated from the model. In both yields, the `namePiece: $cookie_user` is replaced with the query hole symbol `?0`. The difference in table name in the `FROM` clause is also reflected in the two generated strings, but is not replaced with a query hole, since in both yields, `$cookie_type` can be resolved

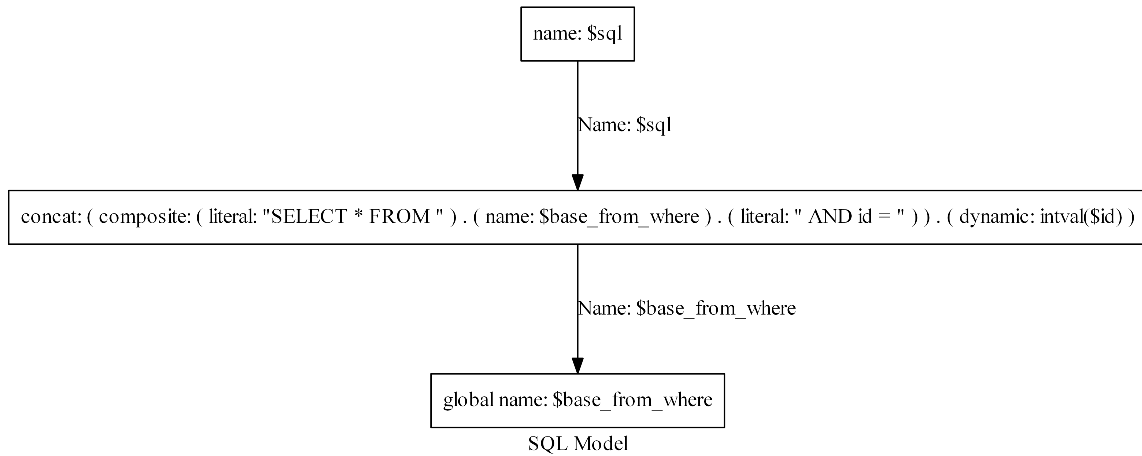

Figure 3.9: Dot Graph for a Query with Dynamic Query Text.

to a static string.

The final step is to parse the generated query strings into SQL ASTs. For this, we used a custom fork of the MySQL parser used by phpMyAdmin. This parser focuses on the MySQL dialect of SQL and has been used by phpMyAdmin since version 4.5. This parser required modifications in order to work with queries that contain query hole symbols, since it was originally designed to only parse completely static queries. We also added a PHP script to the parser that takes the parsed SQL data structure used by the parser as input, and outputs a Rascal value for the resulting SQL AST. On the Rascal side, we have a module that invokes the parser, reads in the string output by the parser, and converts this string to a Rascal term of type `SQLQuery`. The `SQLQuery` type defines an abstract syntax of the MySQL dialect of SQL where a constructor for each query type is defined, and each constructor contains fields for each clause that could be provided in the related type of query. Figure 3.8 shows the result from parsing the two yields from Figure 3.7. These SQL ASTs provide a structured way to compare different model yields and extract empirical information about SQL queries.

1. `[staticPiece("SELECT * FROM "),namePiece("base_from_where"), staticPiece(" AND id = "),dynamicPiece()]`

2. `SELECT * FROM ?0 AND id = ?1`

3. `partialStatement(connectiveWithoutWhere("SELECT"))`

Figure 3.10: Dynamic Query Text: Yield, Query String, and AST.

One problem with parsing partial SQL queries comes in cases where actual query text is contained in a query hole. An example of this from system AddressBook can be found in Figure 3.9. In this case, both the `FROM` clause of the `SELECT` query and part of the `WHERE` clause are contained in the global name `$base_from_where`. For now, we handle these cases on a case-by-case basis and include some checks prior to parsing to identify one of these predefined cases (such as the one seen here, where a logical connective such as `AND` is encountered but no `WHERE` token can be found. The yield, query string, and SQL AST for this example can be found in Figure 3.10. The AST for this example indicates that this query could not be parsed fully, since a connective was encountered but not a `WHERE`. At this point, the only information extracted from these cases is the query type, if it can be determined. Since the original parser was not designed to work with cases like this, handling them is a difficult problem. More powerful options for handling these cases and extracting more information from them are discussed as part of our future work in Chapter 6.

Chapter 4

Analysis of SQL Models

Chapter 3 described the tool that we built for modeling and analysis of SQL queries in PHP systems. This tool was built to facilitate program analysis of embedded SQL queries in PHP systems with the ultimate goal of answering the following research questions:

- **RQ1:** What patterns do developers commonly use to construct database queries?
- **RQ2:** Which parts of queries are generally static? Where do dynamic parts usually occur in queries?
- **RQ3:** Which PHP language features do the data from dynamic query parts come from?
- **RQ4:** In cases where different control flow paths lead to different queries, which clauses are the same across yields, and which are different?
- **RQ5:** Which SQL clauses are most often used in practice? Which are hardly used?

In this chapter, we describe the results of our empirical study and relate the results to these 5 research questions. In Section 4.1, our corpus of 22 systems is described. Section 4.2 shows the Query Construction Patterns (QCPs) that we observed were used most frequently in our corpus and the counts of each pattern (**RQ1** and **RQ2**). Section 4.3 address **RQ3** by examining the PHP language features that are used in the dynamic parts of SQL queries. In Section 4.4, **RQ4** and **RQ2** are addressed by

taking a further look at how queries differ based on control flow. **RQ5** is addressed in Section 4.5, where we take a look at which SQL clauses developers use frequently and which are used more rarely.

4.1 The Corpus

22 systems were selected as the corpus for this research. Four of the systems: Schoolmate, geccBBLite, FAQ Forge, and WebChess, were selected due to their inclusion in previous experiments [6], [7]. All other systems were selected from GitHub and SourceForge with the criteria that 1) each system should use the original MySQL API and 2) the systems chosen should vary in size, age, and problem domain (e.g. School, Medical, Bulletin Board, etc.). The corpus includes AddressBook, an application for managing contacts; CPG and LinPHA, photo gallery tools; FAQ Forge, a tool for managing lists of Frequently Asked Questions; the bulletin board applications Fire-Soft-Board, geccBBLite, and UseBB; inoERP, an enterprise resource management system; MantisBT, a tool for bug tracking; the school management applications MyPHPSchool, SchoolERP, and Schoolmate; OcoMon, a computer help-desk system; OMS, an organization management system; OpenClinic, a medical records system; OrangeHRM, a human resources management application; PHPAgenda, an agenda management tool; PHPFusion, a content management system; SugarCE, a customer relationship management application; Timeclock, a tool for managing employee work hours; web2project, a project management system; and WebChess, an online chess game. Table 4.1 contains information about the corpus including the system names, versions, file counts, and SLOC. In total, the corpus contains 9615 files and 1,600,472 total lines of PHP source code, measured using the CLOC tool. For the original 4 systems picked from previous research, the versions used in those experiments were

System	Version	File Count	SLOC
AddressBook	8.2.5.2	239	30,296
CPG	1.5.46	359	128,985
FAQ Forge	1.3.2	17	1,040
Fire-Soft-Board	2.0.5	271	47,464
geccBBlite	0.1	11	304
inoERP	0.5.1	2,025	309,592
LinPHA	1.3.4	223	64,167
MantisBT	2.10.0	1,251	155,641
MyPHPSchool	0.3.1	70	8,230
OcoMon	2.0RC6	398	77,879
OMS	1.0.1	16	2,234
OpenClinic	0.8.2	170	16,613
OrangeHRM	4.0	2,560	271,733
PHPAgenda	2.2.12	60	9,680
PHPFusion	7.02.07	470	40,442
SchoolERP	1.0	657	296,053
Schoolmate	1.5.4	63	6,554
SugarCE	6.5.26	2,872	624,987
Timeclock	1.04	63	17,446
UseBB	1.0.16	84	12,650
web2project	3.3	584	99,944
WebChess	0.9.0	24	3,525

The File Count includes files with either a .php or an .inc extension, while SLOC includes source lines from these files. In total, there are 22 systems consisting of 9,615 files with 1,600,472 total lines of source.

Table 4.1: The Corpus.

used. For all other systems, the newest version (at the time of download) was used.

Due to performance issues in the analysis that are being solved as part of our current work, two of the systems in the corpus were not included in the results of this empirical study. These two systems are OcoMon and SchoolERP.

4.2 Query Construction Patterns

Understanding common patterns developers use to create database queries is an important step towards developing program understanding and program transformation tools for PHP systems that use databases. In this section, we describe the Query Construction Patterns (QCPs) that we identified through analysis of the corpus described in Section 4.1. This set of patterns represents our work towards answering **RQ1**. **RQ2** is also addressed in this section as some query construction patterns are defined based on the location of dynamic query parts in the SQL ASTs. The rest of this section is organized by pattern. For each QCP, in addition to a description of the pattern and how it is recognized, examples are given at both the script level (PHP code) and the model, yield, and SQL AST level.

4.2.1 QCP0: Literal Query Strings

The QCP0 pattern is the most basic of the patterns. In this pattern, the parameter passed into `mysql_query` is a static SQL string without any dynamic inputs. The following code snippet from line 149 of *Registration.php* in Schoolmate provides an example of QCP0:

```
1 $query = mysql_query("SELECT studentid,fname,lname FROM
     students ORDER BY fname ASC")
```

Since the query parameter is a static string with no dynamic inputs, the SQL model for this call consists of a single `literalFragment`. This is shown in Figure 4.1.

When run through the Yield Generator, the single `literalFragment` in this model corresponds to a single static yield. Figure 4.2 shows 1) this yield, 2) the corresponding query string, and 3) the SQL AST output by the parser. Notice that since the yield for this model contains only a single `staticPiece`, the query string and SQL

literal: "SELECT studentid,fname,lname FROM students ORDER BY fname ASC"

SQL Model

Figure 4.1: QCP0: Example 1.

1. `{[staticPiece("SELECT studentid,fname,lname FROM students ORDER BY fname ASC")]}`

2. `{"SELECT studentid,fname,lname FROM students ORDER BY fname ASC"}`

3. `{selectQuery([name(column("studentid")),name(column("fname")),name(column("lname"))],[name(table("students"))],noWhere(),noGroupBy(),noHaving(),orderBy({<name(column("fname")),"ASC">}),noLimit(),[])}`

Figure 4.2: QCP0: Yield, Query String, and AST.

AST contain no query holes. While the example in Figure 4.1 shows a static query defined directly in the call to `mysql_query`, QCP0 also encompasses cases where a static SQL string is assigned to a variable, then that variable is used in a `mysql_query` call. The following example from lines 53 and 54 of *lib/topics.inc* in FaqForge shows an example of this case:

```
1 $q  = "SELECT * FROM Faq WHERE parent_id = 0 ORDER BY
    list_order";
2 $result0 = mysql_query ($q, $dbLink);
```

In this case, the model for this query consists of two fragments: a `nameFragment` for the variable `$q` and a `literalFragment` for the SQL string assigned to `$q`. This is shown in Figure 4.3. When the yields are generated, however, they are of the same form as in Figure 4.2 with a single yield, query string, and SQL AST. Recognizing either case of QCP0 is done by 1) checking whether the yield set of the

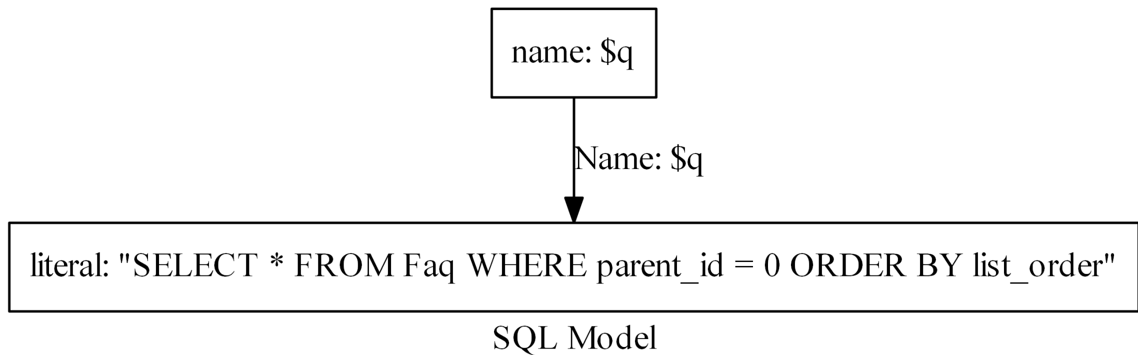

Figure 4.3: QCP0: Example 2.

model is a singleton set and 2) checking that the yield of the model contains a single `staticPiece`.

4.2.2 QCP1: Dynamic Parameters

In pattern QCP1, the query is a mixture of static SQL text and dynamic inputs, where each dynamic part is a parameter to SQL constructs such as `SET` or `WHERE`. This is in contrast to cases where the dynamic parts are not limited to just parameters, but can represent longer fragments of query text, potentially including some schema elements (these are not allowed in QCP1; they are covered by QCP2). The following code example from lines 26 and 27 of */html/newsadmin.php* in MyPHPSchool shows an example of a QCP1 query in PHP code:

```
1 $sql = "UPDATE news SET subject = '$new_subject', lead = '
    $new_lead', content = '$new_content', type = '$new_type'
    WHERE id = '$id'";
2 $result = mysql_query($sql, $db);
```

In this example, the dynamic inputs `$new_subject`, `$new_lead`, `$new_content`, `$new_type`, and `$id` are used as parameters to the `SET` operations in the `UPDATE` query. No dynamic parts of the query contain schema elements or other query text. Another

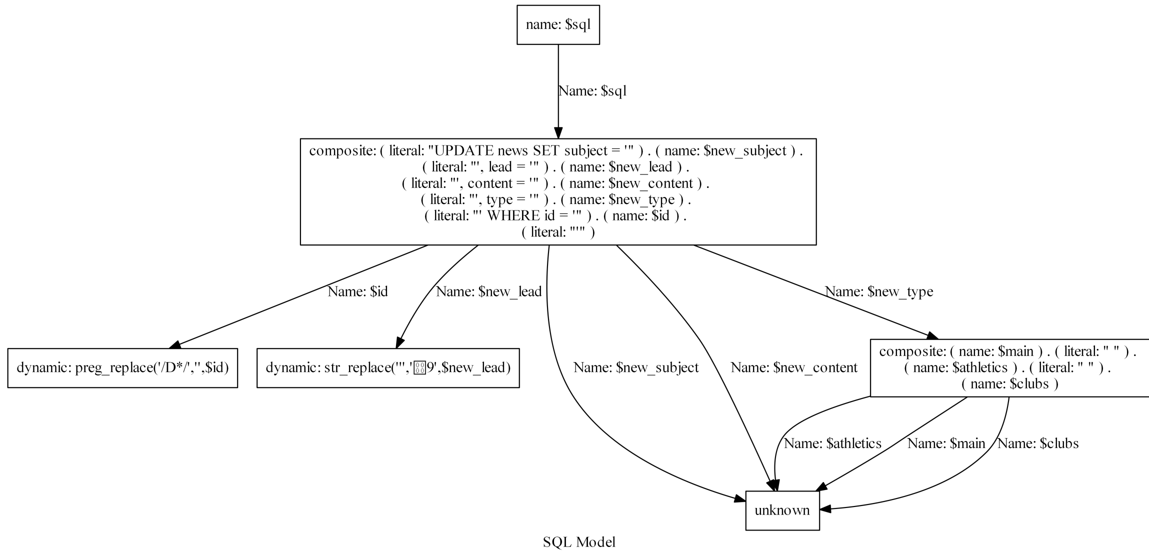

Figure 4.4: QCP1: Example 1.

characteristic of QCP1 models is that, as with QCP0, only a single yield is generated. In other words, the model does not take on different paths based on control flow, and none of the edges of the model have conditions. Figure 4.4 shows a dot graph representation of the QCP1 query from the previous code example and Figure 4.5 shows the yield, query string, and SQL AST corresponding to this model. Unlike QCP0, which had no name or dynamic pieces in its yield, the yield for this model contains name pieces which show up as hole tokens (`?0`, `?1`,...,`?6`) in the query string and the parsed SQL AST. Recognizing QCP1 involves first checking whether the yields set of the model is a singleton set. If it is, then the location of each query hole in the SQL AST is checked. If all query holes are located in the query as parameters, then the query being examined is a QCP1 query.

4.2.3 QCP2: Dynamic

Like Pattern QCP1, QCP2 is made of a mixture of static SQL and dynamic inputs from PHP constructs. Also like QCP1, the query has a single yield and does not

1. ```
{[staticPiece("UPDATE news SET subject = \'"),namePiece("
    new_subject"),staticPiece("\', lead = \'"),namePiece("
    new_lead"),staticPiece("\', content = \'"),namePiece("
    new_content"),staticPiece("\', type = \'"),namePiece("main
    "),staticPiece(" "),namePiece("athletics"),staticPiece(" "
    ),namePiece("clubs"),staticPiece("\' WHERE id = \'"),
    namePiece("id"),staticPiece("\'")]}
```

2. ```
{"UPDATE news SET subject = \'?0\', lead = \'?1\', content =
    \'?2\', type = \'?3 ?4 ?5\' WHERE id = \'?6\'"}
```

3. ```
{updateQuery([name(table("news"))],[setOp("subject","?0"),
    setOp("lead","?1"),setOp("content","?2"),setOp("type","
    \'?3 ?4 ?5\'")],where(condition(simpleComparison("id","=",
    "?6"))),noOrderBy(),noLimit())}
```

Figure 4.5: QCP1: Yield, Query String, and AST.

differ based on control flow. Unlike QCP1, in pattern QCP2, at least one dynamic part of the query takes on the role of a schema element or part of query text. An example of this can be found in the following code example from lines 413 and 414 of *admin.php* in OMs:

```
1 $sql = "select * from memberships where org_id = $org_id
    ORDER BY $order";
2 $result = mysql_query($sql,$conn) or die(mysql_error());
```

The query in this example has two dynamic parts: `$org_id`, which is a parameter to the `WHERE` clause, and `$order`, which specifies the column name for the `ORDER BY` clause. Since this second dynamic part is not a parameter, this query is classified as QCP2. Figure 4.6 shows the dot graph representation of this query. Figure 4.7 shows the corresponding yield, query string, and SQL AST. Recognizing QCP2 is similar to recognizing QCP1. First, it is checked whether the yield set is singleton, and if it is, the locations of all holes in the SQL AST are checked. If at least one hole is not
```

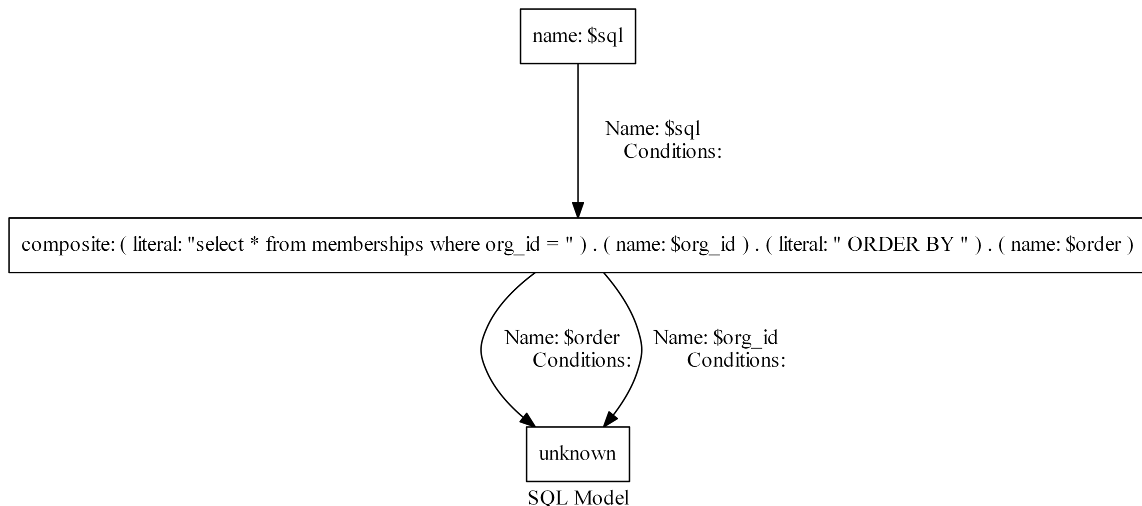

Figure 4.6: QCP2: Example 1.

1. `{[staticPiece("select * from memberships where org_id = "), namePiece("org_id"),staticPiece(" ORDER BY "),namePiece("order")]}`

2. `{"select * from memberships where org_id = ?0 ORDER BY ?1"}`

3. `{selectQuery([star()],[name(table("memberships"))],where(condition(simpleComparison("org_id","=","?0"))),noGroupBy(),noHaving(),orderBy({<name(column("?1")),"ASC">}),noLimit(),[])}`

Figure 4.7: QCP2: Yield, Query String, and AST.

used as a parameter, this query is classified as a QCP2 query.

Recall the case discussed in Section 3.4, where entire pieces of SQL text are contained in a variable. These cases also fall under QCP2, since a dynamic part of the query is used to contain a part of a query other than a parameter. Recognizing these cases simply involves checking if the AST output by the parser is of type *partialStatement*.

4.2.4 QCP3a: Multiple Yields, Same Parsed Query

QCP3a is the first pattern where the number of yields is greater than 1. However, in QCP3a, every yield in the set of yields leads to the same query string and the same SQL AST. This means that the yields of the model differ in the source of dynamic information but not the actual structure of the query. The following code example from lines 205-364 (with irrelevant lines replaced with ellipses) of */admin/officeedit.php* in Timeclock shows an example QCP3a query call:

```
1 $post_officeid = $_POST['post_officeid'];
2 ...
3 while ($row=mysql_fetch_array($result)) {
4           $post_officeid = "".$row['officeid']."";
5 }
6 ...
7 $query = "select * from ".$db_prefix."groups where officeid =
      ('".$post_officeid."') order by groupname";
8 $result = mysql_query($query);
```

Notice that the value of the `$post_officeid` variable has two possible sources based on control flow. First, if the `while` loop is executed, the value comes from the `$row` array. For the case where the loop is not executed, the value comes from the `$_POST` superglobal. This is shown in Figure 4.8 where `$post_officeid` has two edges in the model. Note that condition labels for the edges in this model have been removed, for readability. This is also shown in Figure 4.9 where there are two yields of the model, one with `namePiece("row")` and one with `namePiece("_POST")`. When these yields are converted to query strings, however, they both result in the same query string, which when parsed, results in a single SQL AST. Figure 4.9 also shows this. It should be noted that, while there is another dynamic input, `$dbprefix` in

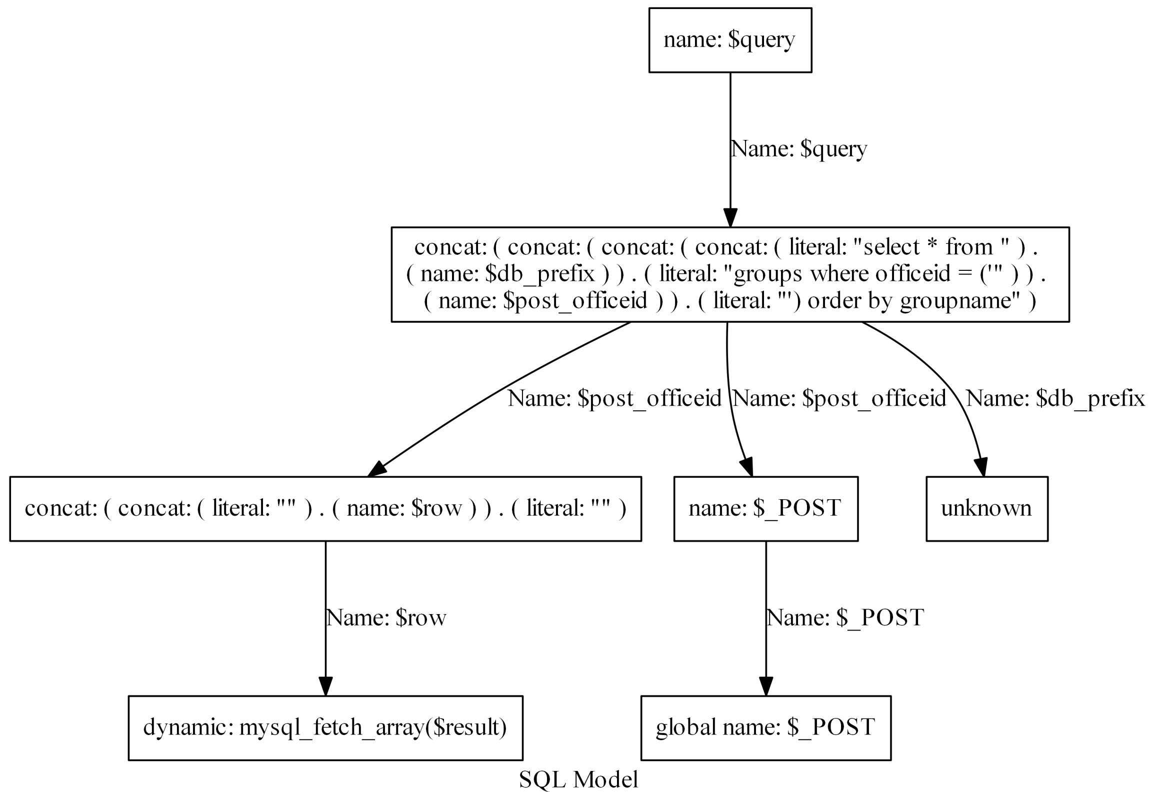

Figure 4.8: QCP3a: Example 1.

1. ```
   {[staticPiece("select * from "),namePiece("db_prefix"),
       staticPiece("groups where officeid = (\'"),namePiece("row"
       ),staticPiece("\') order by groupname")],
    [staticPiece("select * from "),namePiece("db_prefix"),
       staticPiece("groups where officeid = (\'"),namePiece("
       _POST"),staticPiece("\') order by groupname")]}
   ```

2. ```
   {"select * from ?0groups where officeid = (\'?1\') order by
       groupname"}
   ```

3. ```
   {selectQuery([star()],[name(table("?0groups"))],where(
       condition(simpleComparison("officeid","=","?1"))),
       noGroupBy(),noHaving(),orderBy({<name(column("groupname"))
       ,"ASC">}),noLimit(),[])}
   ```

Figure 4.9: QCP3a: Yields, Query String, and AST.
   ```

this query, its value does not change based on control flow. This also shows another QCP3 property: we do not differentiate between holes being parameters, schema elements, and query text for this pattern. This is because, since QCP3 queries have multiple yields, it is possible that for one yield, the query has schema element holes, and for the other, it does not. This is not a problem for QCP3a, since all yields lead to the same SQL AST, but for other QCP3 sub-patterns, this issue could arise (see Section 4.2.5 and Section 4.2.6). Recognizing QCP3a involves first checking that the size of the yields set of the model is greater than one. If it is, the size of the SQL AST set is checked. If it is singleton, then this is a QCP3a query, since all yields in the yields set lead to a single SQL AST.

4.2.5 QCP3b: Multiple Yields, Same Query Type

Like QCP3a, the number of yields for QCP3b models is greater than one. Unlike QCP3a, in QCP3b the structure of the query differs across the yields. This is in contrast to QCP3a, where only the sources of dynamic data differ but the structure of the query remains static. Due to the difference in query structure, QCP3b queries have multiple distinct SQL ASTs. Another constraint on QCP3b is that all SQL ASTs must have the same query type (i.e. there cannot be a case where one AST is an `INSERT` query, and another is an `UPDATE` query). The following code example from lines 383-391 of */leftmain.php* in TimeClock shows example PHP code for a QCP3b query call:

```
1 if (strtolower($ip_logging) == "yes") {
2     $query = "insert into ".$db_prefix."info (fullname, `
        inout`, timestamp, notes, ipaddress) values ('".
        $fullname."', '".$inout."', '".$tz_stamp."', '".$notes
        ."', '".$connecting_ip."')";
```

```
3 } else {
4     $query = "insert into ".$db_prefix."info (fullname, `
         inout`, timestamp, notes) values ('".$fullname."', '".
         $inout."', '".$tz_stamp."', '".$notes."')";
5 }
6
7 $result = mysql_query($query);
```

The PHP code that sets the values of each of the variables in the `VALUES` clauses is omitted, for space. In this example, the `INSERT` query differs based on the value of the `strtolower($ip_logging) == "yes"` condition. In each of the two branches, the `INTO` and `VALUES` clauses contain different numbers of column names and values, respectively. This is a difference in query structure, which makes this case fall under QCP3b. The dot graph for this example was omitted due to the large size of the image file associated with it. Figure 4.10 shows the two yields, two SQL strings, and two SQL ASTs corresponding to this example. The first SQL AST is for the case where `strtolower($ip_logging) == "yes"` is true, and an extra column and value is added for the logged IP address. The second SQL AST corresponds to the case where that condition evaluated to false, and the extra column and value are not included in the query. QCP3b is recognized by first checking that the size of the yields set of the model is greater than one, and the size of the SQL AST set is greater than one. If both are true, then it is asserted that all the ASTs are of the same type.

4.2.6 QCP3c: Multiple Yields, Different Query Types

In the QCP3c pattern, the model has multiple yields of differing query types. The following PHP code from lines 187-196 (with comments removed) of *phpweather.php* in

1.
```
[staticPiece("insert into "),namePiece("db_prefix"),
   staticPiece("info (fullname, ‘inout‘, timestamp, notes)
   values (\’"),namePiece("fullname"),staticPiece("\’, \’"),
   namePiece("_POST"),staticPiece("\’, \’"),namePiece("
   tz_stamp"),staticPiece("\’, \n                          \’"),
   namePiece("notes"),staticPiece("\’)")]
[staticPiece("insert into "),namePiece("db_prefix"),
   staticPiece("info (fullname, ‘inout‘, timestamp, notes,
   ipaddress) values (\’"),namePiece("fullname"),staticPiece(
   "\’, \’"),namePiece("_POST"),staticPiece("\’, \n
                         \’"),namePiece("tz_stamp"),
   staticPiece("\’, \’"),namePiece("notes"),staticPiece("\’,
   \’"),namePiece("connecting_ip"),staticPiece("\’)")]
```

2.
```
insert into ?0info (fullname, ‘inout‘, timestamp, notes)
   values (’?1’, ’?2’, ’?3’,’?4’)
insert into ?0info (fullname, ‘inout‘, timestamp, notes,
   ipaddress) values (’?1’, ’?2’,’?3’, ’?4’, ’?5’)
```

3.
```
insertQuery(into(name(table("?0info")),["fullname","inout","
   timestamp","notes","ipaddress"]),[["?1","?2","?3","?4","?5
   "]],[],noQuery(),[])
insertQuery(into(name(table("?0info")),["fullname","inout","
   timestamp","notes"]),[["?1","?2","?3","?4"]],[],noQuery()
   ,[])
```

Figure 4.10: QCP3b: Yields, Query Strings, and ASTs.

Timeclock shows an example of QCP3c. Notice that the entire query differs based on the `$new` variable, and the different paths hold different query types: an `INSERT` and an `UPDATE` query. Note that the code that sets the values of the variables embedded in the queries is not shown.

```
1 if ($new) {
2         $query = "INSERT INTO ".$db_prefix."metars SET
             station = '$station', " ."metar = '$metar',
             timestamp = '$date'";
```

```
3 } else {
4         $query = "UPDATE ".$db_prefix."metars SET metar = '
            $metar', " ."timestamp = '$date' WHERE station = '
            $station'";
5 }
6 mysql_query($query);
```

Figure 4.11 shows example yields, SQL strings, and SQL ASTs for this example. In this model, there are multiple `UPDATE` yields and multiple `INSERT` yields. The collection of yields of this model was shortened for space.

4.2.7 QCP4: Function Queries

QCP4 is the most dynamic of the query construction patterns. In this pattern, the entire query is dynamic. Most commonly, this happens when a query comes from a parameter of a PHP function or method. The following code example from lines 25 and 26 of */inoerp/tparty/extenstions/social_login/hybridauth/examples/signin_signup/application.config.php* in system inoERP shows an example of QCP4.

```
1 function mysql_query_excute( $sql ){
2         $result = mysql_query($sql);
3         ...
```

Figure 4.12 shows the dot graph for the model of this example. In this example, the name `$sql` can be traced back to the function parameter name, leaving the entire query as dynamic, which makes it a QCP4 query. Figure 4.13 gives the yield, query string, and SQL AST for this example. The yield consists of a single name piece, which, when converted to a query string, results in "?0", since the whole query is a hole. Since this does not represent a full query that can be recognized by the parser, it

1. [staticPiece("UPDATE "),namePiece("db_prefix"),staticPiece(" metars SET metar = \'\', timestamp = \'"),namePiece("date"),staticPiece("\' WHERE station = \'"),namePiece("station"),staticPiece("\'")]
   [staticPiece("UPDATE "),namePiece("db_prefix"),staticPiece(" metars SET metar = \'"),dynamicPiece(),staticPiece("Z "), namePiece("metar"),staticPiece(" "),dynamicPiece(), staticPiece("\', timestamp = \'"),namePiece("date"), staticPiece("\' WHERE station = \'"),namePiece("station"), staticPiece("\'")]
   [staticPiece("INSERT INTO "),namePiece("db_prefix"), staticPiece("metars SET station = \'"),namePiece("station"),staticPiece("\', metar = \'\', timestamp = \'"), namePiece("date"),staticPiece("\'")]
   [staticPiece("INSERT INTO "),namePiece("db_prefix"), staticPiece("metars SET station = \'"),namePiece("station"),staticPiece("\', metar = \'"),namePiece("metar"), staticPiece(" "),dynamicPiece(),staticPiece("\', timestamp = \'"),namePiece("date"),staticPiece("\'")]

2. UPDATE ?0metars SET metar = '', timestamp = '?1' WHERE station = '?2'
   UPDATE ?0metars SET metar = '?1Z ?2 ?3', timestamp = '?4' WHERE station = '?5'
   INSERT INTO ?0metars SET station = '?1', metar = '', timestamp = '?2'
   INSERT INTO ?0metars SET station = '?1', metar = '?2 ?3', timestamp = '?4'

3. updateQuery([name(table("?0metars"))],[setOp("metar","\'\'"), setOp("timestamp","?1")],where(condition(simpleComparison("station","=","?2"))),noOrderBy(),noLimit())
   updateQuery([name(table("?0metars"))],[setOp("metar","\'?1Z ?2 ?3\'"),setOp("timestamp","?4")],where(condition(simpleComparison("station","=","?5"))),noOrderBy(),noLimit ())
   insertQuery(into(name(table("?0metars")),[]),[],[setOp(" station","?1"),setOp("metar","\'\'"),setOp("timestamp","?2 ")],noQuery(),[])
   insertQuery(into(name(table("?0metars")),[]),[],[setOp(" station","?1"),setOp("metar","\'?2 ?3\'"),setOp("timestamp ","?4")],noQuery(),[])

Figure 4.11: QCP3c: Yields, Query Strings, and ASTs.

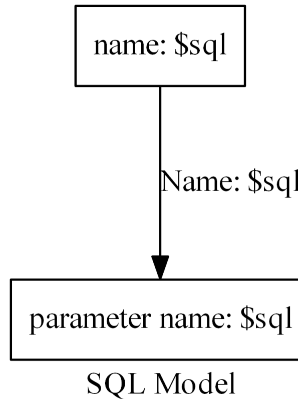

Figure 4.12: QCP4: Example 1.

1. `[namePiece("sql")]`

2. `?0`

3. `partialStatement(unknownStatementType())`

Figure 4.13: QCP4: Yield, Query String, and AST.

returns a `partialStatement`, where `unknownStatementType` indicates that it could not determine what type of SQL statement the query represents. For QCP4 queries, we do not examine the SQL ASTs as they do not contain any relevant information.

4.2.8 Query Construction Pattern Results

Table 4.2 shows the counts of each Query Construction pattern in the corpus. In addition to the Query Construction Patterns we discussed in the previous sections, this table includes figures for other query types (O) which are queries that are not `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements. The reason these are discarded is that our tool does not support them—many (e.g. `ALTER TABLE`) are used to modify the database schema, while we are focusing on statements used to query and modify

System	Version	0	1	2	3a	3b	3c	4	O	P	U
AddressBook	8.2.5.2	0	21	60	2	10	2	1	6	0	0
cpg	1.5.46	0	0	10	0	0	0	6	5	0	0
FAQ Forge	1.3.2	4	21	0	1	7	0	0	0	0	0
Fire-Soft-Board	2.0.5	0	0	0	0	0	0	2	1	0	0
geccBBlite	0.1	1	6	0	0	2	0	0	1	0	0
inoERP	0.5.1	0	0	0	0	0	0	2	0	0	0
LinPHA	1.3.4	0	0	3	0	0	0	2	8	0	0
mantisbt	2.10.0	0	0	0	0	0	0	1	0	0	0
MyPHPSchool	0.3.1	13	63	1	0	8	0	0	0	0	0
OMS	1.0.1	4	80	2	0	0	0	0	1	0	0
OpenClinic	0.8.2	0	3	0	0	0	1	0	4	2	0
orangehrm	4.0	22	4	3	0	0	0	1	11	0	0
PHPAgenda	2.2.12	0	0	16	0	0	0	3	2	0	0
PHPFusion	7.02.07	0	0	7	0	0	0	2	0	0	0
Schoolmate	1.5.4	76	192	0	15	11	0	0	0	0	0
SugarCE	6.5.26	0	0	0	1	0	0	2	1	0	0
Timeclock	1.04	2	3	279	11	18	1	0	52	0	0
UseBB	1.0.16	0	0	0	0	0	0	1	0	0	0
web2project	3.3	0	0	0	0	0	0	1	0	0	0
WebChess	0.9.0	0	69	0	6	18	0	0	0	0	0
totals	-	122	462	381	36	74	4	24	92	2	0

Counts for each QCP in each System. QCP names are abbreviated. Numbered patterns are replaced by their number. O stands for other query type. P stands for parse error. U stands for models that match no patterns.

Table 4.2: QCP Counts by System.

the data in the existing database schema. Figures for parse errors (P) and queries that fall under no pattern (U) are also included in Table 4.2. From this data, QCP1 is the most frequent with 38.6% of models falling under this pattern. QCP2 is the next most frequent with 31.8%. QCP0 is next with 10.2%. Other query types (O), QCP3b, QCP3a, QCP4, and QCP3c follow next with 7.7%, 6.2%, 3.0%, 2.0%, and

0.33%, respectively. Finally, 0.16% of models have query strings that result in parse errors (due to 2 cases of `USE` statements that are not handled by the parser), and 0% of models fall under no pattern.

While these results indicate that dynamic parameters (QCP1) occur a similar number of times as dynamic queries (QCP2), it should be noted that an overwhelming majority of QCP2 queries came from a single system, Timeclock. The reason for this can be seen in the Timeclock example in Section 4.2.4 (and other Timeclock examples in subsequent sections). Nearly every call in this system appends `$db_prefix`, a globally-defined configuration variable, to every table name, causing them to be classified as QCP2. We believe this to be an outlier, as QCP2 queries do not have nearly as high occurrences in the rest of the corpus, and no other system in the corpus uses this practice of appending a prefix to every table name. Standard practice in PHP is to instead define such a prefix as a constant. If that had been done here, this constant would be replaced as part of the string simplifications mentioned in Chapter 3, making that part of the query static. If Timeclock is removed from the corpus, the top 3 patterns shift to 55.2% of queries falling under QCP1, 14.4% falling under QCP0, and 12.2% falling under QCP2. We believe these results are promising for future program understanding and transformation efforts, since with Timeclock removed, nearly 70% of queries are either literal, or dynamic in parameter values.

4.3 Query Fragment Categories

Recall that **RQ3** asks the question "Which PHP language features do the data from dynamic query parts come from?". To answer this, we defined an analysis in Rascal that categorizes model fragments into the following categories: literals, local variables, properties of local variables, computed local names, global variables, global proper-

System	L	LV	LP	LC	GV	GP	GC	PN	PP	PC	C
AddressBook	506	374	1	0	74	0	0	31	0	0	258
cpg	27	29	1	0	2	0	0	3	0	0	26
FAQ Forge	84	88	0	0	0	0	0	29	0	0	51
Fire-Soft-Board	2	4	0	0	0	0	0	2	0	0	1
geccBBlite	24	31	0	0	3	0	0	4	0	0	21
inoERP	0	2	0	0	0	0	0	2	0	0	0
LinPHA	10	21	0	0	3	0	0	1	0	0	6
mantisbt	0	1	0	0	0	0	0	1	0	0	0
MyPHPSchool	278	268	0	0	56	0	0	0	0	0	152
OMS	173	246	0	0	22	0	0	0	0	0	186
OpenClinic	35	15	0	0	0	0	0	3	0	0	17
orangehrm	22	14	2	0	0	0	0	5	0	0	18
PHPAgenda	56	45	0	0	0	0	0	23	0	0	42
PHPFusion	22	11	0	0	6	0	0	4	0	0	11
Schoolmate	484	316	0	0	120	0	0	39	0	0	362
SugarCE	3	4	0	0	0	0	0	2	0	0	2
Timeclock	1492	1297	0	0	152	0	0	1	0	0	615
UseBB	0	1	0	0	0	0	0	1	0	0	0
web2project	0	1	0	0	0	0	0	1	0	0	0
WebChess	240	201	0	0	87	0	0	1	0	0	72
total	3458	2969	4	0	525	0	0	153	0	0	1840

Counts of each Fragment Category in the corpus. The table headings for each fragment category have the following abbreviations: L for literals, LV for local variables, LP for properties of local variables, LC for computed local names, GV for global variables, GP for properties of global variables, GC for computed global names, PN for parameters, PP for properties of parameters, PC for computed property names and C for computed fragments that are not names

Table 4.3: Fragment Category Counts by System.

ties, computed global properties, parameter names, parameter properties, computed parameter properties, and computed dynamic fragments that are not names. Computed names refer to cases where the name that is associated with a model fragment is given as an expression. An example of this is the use of a variable variable (such as

`$$x`) to refer to the value of a name. Table 4.3 gives an overview of the counts of each fragment category in the corpus. In total, 38.6% of model fragments are literal query text values (L), 33.2% of model fragments come from local variables (LV), 20.6% come from computed values that are not names (C), 5.9% come from global names (GV), 1.7% come from function/method parameters (PN), and 0.04% come from properties of local variables. None of the model fragments in the corpus fall under computed global names (GC), computed local names (LC), properties of global variables (GP), computed property names (PC) or properties of parameters (PP).

4.4 Model Yield Comparison

As discussed in Section 4.2, in QCP3 queries, there are multiple model yields that differ based on control flow. In this section, we further examine these cases to provide an answer to **RQ4**. For this, we created an ADT in Rascal called `YieldInfo` which organizes the SQL ASTs of a model into a single structure. Figure 4.14 shows the Rascal definition of type `YieldInfo`. A `YieldInfo` instance is either `sameType`, for QCP3a and QCP3b queries, or `differentTypes` for QCP3c queries. For `differentTypes`, a `sameType` instance is created for each type. For `sameType`, a `ClauseInfo` instance is stored that contains information about the clauses of that type of query. The Rascal definition of `ClauseInfo` can also be found in Figure 4.14, however, only the constructor for select queries is shown. The `ClauseInfo` for select/insert/update/delete queries contains a `ClauseComp` instance for each clause of its respective query type. The Rascal definition for `ClauseComp` can also be found in Figure 4.14. The `ClauseComp` type represents how a particular clause differs between yields of a SQL model. The *none* constructor represents the case where a clause does not exist in any yield of the model. The *same* constructor corresponds to the case where a

```
data YieldInfo = sameType(ClauseInfo clauseInfo)
              | differentTypes(set[ClauseInfo] clauseInfos);

data ClauseInfo = selectClauses(ClauseComp select, ClauseComp
    from, ClauseComp where, ClauseComp groupBy, ClauseComp
   having, ClauseComp orderBy, ClauseComp limit, ClauseComp
   joins)

data ClauseComp = different(set[&T] clauses)
               | some(set[&T] clauses)
               | same(&T clause)
               | none();
```

Figure 4.14: YieldInfo Rascal Definition.

clause is static among all yields of the query. For cases where some yields contain a particular clause and others do not, the *some* constructor is used. For cases where every yield contains a particular clause, but the clause differs across yields, the *different* constructor is used.

To build the `YieldInfo` for a SQL model, the root node of each SQL AST of the model is first compared. If they are all the same, the `sameType` constructor is used. Otherwise, the `differentTypes` constructor is used. For the case of the latter, a `sameType` instance is built for each type. For a `sameType` model of a select/insert/update/delete query, a `ClauseComp` is built for each clause of the query. The process for building a `ClauseComp` is best best described using a state machine diagram, which can be found in Figure 4.15. The clause comparison algorithm iterates through the SQL ASTs for each yield of the model and examines the desired clause. A state transition occurs each time a new AST is examined. Once all the SQL AST's for the model have been examined, the final state is the result of the clause comparison. The starting state for the clause comparison is the *none* state. If the algorithm is currently processing the clause from the first AST, it transitions to the *same* state. Otherwise, it checks if the current AST contains the desired clause. If it does, the

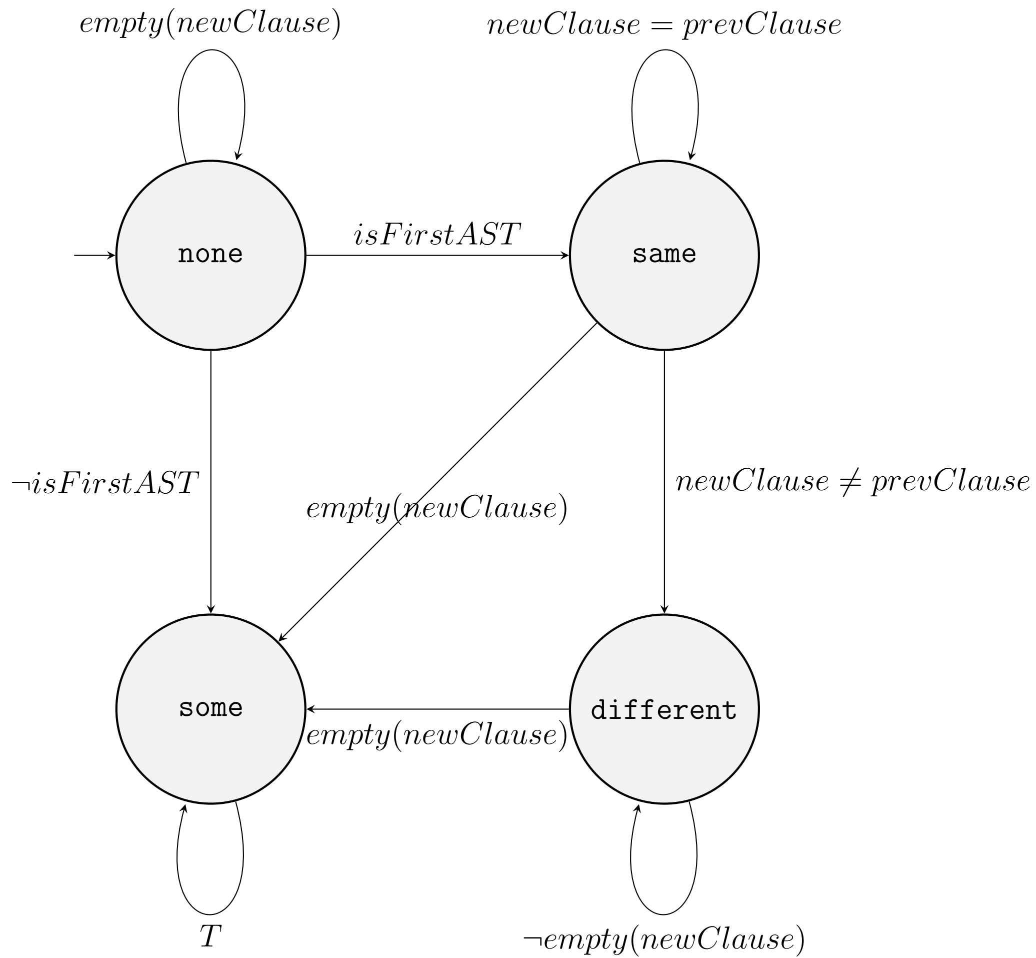

Figure 4.15: ClauseComp Construction: State Machine Diagram.

`ClauseComp` transitions to the *some* state, since this means the previous AST did not contain the desired clause, but the current one does. Otherwise, the `ClauseComp` remains in the *none* state. From the *same* state, if the current AST does not contain the desired clause, the `ClauseComp` transitions to the *some* state, since the previous AST contained the desired clause but the current one does not. If the current AST does contain the desired clause, it is checked against the value of the clause in the previous AST. If they are the same, the `ClauseComp` remains in the *same* state. If they are different, the `ClauseComp` transitions to the *different* state. From the *different*

```
SELECT COUNT(*) FROM schoolattendance WHERE studentid = ?0
   AND type = 'tardy' AND ?1 <= sattenddate AND sattenddate
   <= ?2

SELECT COUNT(*) FROM schoolattendance WHERE studentid = ''
   AND type = 'tardy' AND ?0 <= sattenddate AND sattenddate
   <= ?1
```

Figure 4.16: YieldInfo example: SQL Strings.

```
YieldInfo: sameType(selectClauses(
    same([call("count")]),
    same([name(table("schoolattendance"))]),
    different({
        where(and(
            condition(simpleComparison("sattenddate","\<=","
               ?2")),
            and(
              condition(simpleComparison("?1","\<=","
                 sattenddate")),
              and(
                condition(simpleComparison("studentid","=","
                   ?0")),
                condition(simpleComparison("type","=","tardy"
                   )))))),
        where(and(
            condition(simpleComparison("sattenddate","\<=","
               ?1")),
            and(
              condition(simpleComparison("?0","\<=","
                 sattenddate")),
              and(
                condition(simpleComparison("studentid","=",""
                   )),
                condition(simpleComparison("type","=","tardy"
                   ))))))
      }),
    none(),
    none(),
    none(),
    none(),
    none()))
```

Figure 4.17: YieldInfo example: YieldInfo.

state, if the current AST contains the desired clause, the `ClauseComp` remains in the *different* state. Otherwise, it transitions to the *some* state. The *some* state is a "trap" state that always transitions to itself, regardless of the ASTs that are processed after. Figure 4.17 shows an example `YieldInfo` for a `SELECT` query with 2 ASTs. The SQL strings for this example can be found in Figure 4.16. Notice that with this structure, it is easy to see that the table being selected from and columns being selected are the same across both ASTs, since these are wrapped by *same* nodes. The `WHERE` clause is different between the two ASTs for this example, since it is represented by a *different* node with the two possible `WHERE` clauses. No other clauses occur in these ASTs since they are represented by *none* nodes.

Table 4.4 shows empirical data extracted from the `YieldInfo` for all QCP3b query models in the corpus. This comparison information only applies to QCP3b because in QCP3a, every node is a *same* or *none* node, since all yields lead to a single SQL AST. In QCP3c, there are multiple `YieldInfo` instances for each query type, and since this pattern does not come up frequently in practice, we did not implement a way to compare clauses across query types. The `YieldInfo` for all other query construction patterns is used to extract query type counts and clause counts in Section 4.5.

In QCP3b `SELECT` queries, the difference between ASTs happens most commonly in the `WHERE` clause where 19/35 cases have a *different* node on the `WHERE` clause. One single case has a *some* node in the `WHERE` clause. This is followed by 12/35 cases having a difference in the `ORDER BY` clause. 2/35 QCP3b `SELECT` queries differ in the `FROM` clause, 1/35 differs in the columns being selected, and 1 differs in the `LIMIT` clause. All other clauses either do not differ in any of the cases, or do not occur. For QCP3b `INSERT` queries, 18/22 cases differ in the `VALUES` clause, followed by 4/22 differing in the `SELECT` subquery statement, and 2/22 differing in the `INTO` clause. All other clauses do not occur in QCP3b `INSERT` query models. In `UPDATE` QCP3b

Query Type	Clauses	Same	Different	Some	None
select	select	34	1	0	0
	from	33	2	0	0
	where	15	19	1	0
	groupBy	0	0	0	35
	having	0	0	0	35
	orderBy	8	12	0	15
	limit	0	1	0	34
	joins	1	0	0	34
insert	into	20	2	0	0
	values	0	18	0	4
	setOps	0	0	0	22
	select	0	4	0	18
	onDuplicateSetOps	0	0	0	22
update	tables	10	2	0	0
	setOps	2	10	0	0
	where	2	10	0	0
	orderBy	0	0	0	12
	limit	1	0	0	11
delete	from	4	0	0	0
	using	0	0	0	4
	where	0	4	0	0
	orderBy	0	0	0	4
	limit	0	0	0	4

Table 4.4: Clause Comparison Counts for Pattern QCP3b.

queries, 10/12 differ in the `SET` operations and 10/12 differ in the `WHERE` clause. 2/12 cases differ in the tables being updated. All other clauses either do not differ in any cases or do not occur. For `DELETE` QCP3b query models, 4/4 differ in the `WHERE` clause. All other clauses either do not differ in any cases or do not occur.

While this information gives an early indication of the clauses where queries com-

System	SELECT	INSERT	UPDATE	DELETE	PARTIAL	OTHER
AddressBook	48	10	15	7	19	7
cpg	4	2	4	0	6	5
FAQ Forge	22	3	5	3	0	0
Fire-Soft-Board	0	0	0	0	2	1
geccBBlite	7	2	0	0	0	1
inoERP	0	0	0	0	2	0
LinPHA	3	0	0	0	2	8
mantisbt	0	0	0	0	1	0
MyPHPSchool	49	17	5	14	0	0
OMS	67	7	9	3	0	1
OpenClinic	0	0	3	0	1	7
orangehrm	3	16	2	8	1	11
PHPAgenda	8	1	3	4	3	2
PHPFusion	3	2	0	2	2	0
Schoolmate	215	16	30	33	0	0
SugarCE	0	0	0	0	2	2
Timeclock	268	18	22	7	0	52
UseBB	0	0	0	0	1	0
web2project	0	0	0	0	1	0
WebChess	55	11	15	12	0	0
totals	752	105	113	93	43	97

Table 4.5: Query Type Counts by System.

monly differ based on control flow, additional data is needed, since the number of QCP3b queries in the corpus is relatively low (only 73 cases). This is discussed more as part of our future work in Section 6.1.

Query Type	Clauses	Counts
select	select	752
	from	752
	where	611
	groupBy	1
	having	0
	orderBy	221
	limit	11
	joins	12
	total queries	752
insert	into	105
	values	95
	setOps	5
	select	5
	onDuplicateSetOps	0
	total queries	105
update	tables	113
	setOps	112
	where	108
	orderBy	0
	limit	15
	total queries	113
delete	from	93
	using	0
	where	88
	orderBy	0
	limit	13
	total queries	93

Table 4.6: Clause Counts for each Query Type.

4.5 SQL Clause Usage

To answer **RQ5**, which asks "Which SQL clauses are most often used in practice? Which are hardly used?", we defined functions that extract empirical information about the counts of each query clause and the counts of each query type. This information is extracted from the `YieldInfo` discussed in Section 4.4. Unlike in Section 4.4, where only QCP3b models were examined, the `YieldInfo` for all models is examined. This is first done by extracting the *same*, *some*, *different*, and *none* counts for each clause in each query type, and putting them in a map of type `map[str, map[str, tuple[int, int, int, int]]`. This map maps each query type to a map of clauses as keys and a tuple of *same*, *some*, *different*, and *none* counts as values. The counts for a particular query type are determined by picking a random clause in the value mapped to the key for that query type, and adding up the *same*, *some*, *different*, and *none* counts for that clause. To get the counts of each clause, this map is converted to a map of type `map[str, map[str, int]]` where each query type is mapped to a map of clauses and their counts. This conversion is achieved by adding up the *same*, *some*, and *different* counts for each clause in the original map (i.e., only counting cases where a clause occurs in at least one yield of the model).

To answer **RQ5**, we first explored a related question: "Which query types are used most frequently in practice?" Table 4.5 shows the counts of each query type, grouped by system. From this data, it can be seen that `SELECT` statements make up 62.5% of queries in the corpus. This is followed by `UPDATE` statements at 9.4%, `INSERT` statements at 8.7%, other statement types at 8.1%, `DELETE` statements at 7.7%, and partial statements at 3.6%. Since this data also extracts type counts from QCP3C queries, the data for other statement types is slightly different than the data shown in Section 4.2.8. Just like in our pattern classifications, Timeclock seems to be

an outlier, with most "other" statement types coming from this system. These come from a high amount of `ALTER` statements, which are part of a database upgrade script that is not part of the regular code that will execute, and `REPLACE` statements, which are MySQL-specific extensions to `INSERT` statements. These statements do not occur as frequently in any of the other systems. With this system removed, the percent of "other" statements falls to 5.4%.

Table 4.6 shows the counts of each clause in each query type. For `SELECT` statements, the `FROM` clause and tables to be selected occur in 100% of queries. 81.2% of `SELECT` queries contain `WHERE` clauses, which is expected as most of the time, records need to be filtered based on some conditions. This is followed by `ORDER BY` with 29.4%. Next, 1.6% of `SELECT` statements contain joins, and 1.5% contain a `LIMIT` clause. Finally, 0.13% of `SELECT` statements use the `GROUP BY` clause and `HAVING` does not occur in the corpus.

For `INSERT` statements, 100% of queries include the `INTO` clause. This is followed closely by the `VALUES` clause which occurs in 90.5% of `INSERT` statements. Next, `SET` operations occur in 4.8% of `INSERT` queries. This indicates that the `INSERT INTO table...VALUES...` statement variant is used much more frequently than the `INSERT INTO table SET...` variant. 4.8% of `INSERT` statements include a `SELECT` subquery that selects columns from another table to provide the data to be inserted into. Finally, `ON DUPLICATE SET` operations do not occur in the corpus.

100% of `UPDATE` statements include tables to be updated. 99% contain `SET` operations. 95.6% contain a `WHERE` clause, which is expected since, in most cases, specific records need to be updated rather than an entire table. 13.2% of `UPDATE` statements include a `LIMIT` clause, and `ORDER BY` does not occur in `UPDATE` statements in the corpus.

For `DELETE` statements, 100% contain tables to be deleted from. 94.6% contain

`WHERE` clauses, which is expected since, in most cases, deleting all records from a table is not desired. Next, 13.9% of `DELETE` queries contain a `LIMIT` clause. Finally, `ORDER BY` and `USING` are not included in any `DELETE` statements in the corpus.

Chapter 5

Related Work

This research builds off of and includes some of our previously published work. This includes our work in [8], where we developed an initial set of query construction patterns and identified those patterns using a less precise analysis on a smaller corpus of systems. We also extracted some initial empirical results that indicated that dynamic portions of queries are most often used as parameters, and that cases where the query text is dynamic completely come up much less frequently. The work in [8] was inspired by the earlier work of Ioana Rucareanu [6] on identifying MySQL query patterns. Our recent work has focused on expanding these patterns, creating a more flexible and precise analysis, and examining the use of these patterns in a larger collection of PHP systems. In [9], we described our SQL modeling and analysis tool, included an overview of the process of extracting SQL models from PHP code, and described the process of parsing partial SQL queries with dynamic holes.

The research by Nagy et al. [10] presents an analysis that is similar to our own. In this work, an analysis is described that provides a set of program locations where a specified query can be executed. This work also contains support for dynamic query fragments that are replaced with "joker" nodes that indicate that this fragment of the query cannot be parsed. Similarly to our own approach, the authors use a modified MySQL parser to handle queries with dynamic inputs. The motivation behind this

work is different from our own. Our analysis focuses on supporting empirical research, program transformation tools, and program understanding tools. The analysis in Nagy et al. is designed to allow developers to determine where the program concepts that correspond to a particular query can be found in the code.

The work by Meurice et al. [11] is also closely related to our analysis. In this paper, the authors describe a static analysis for recovering SQL query strings from Java systems that use the JDBC, Hibernate (a popular ORM), and JPA (a Java standard for object persistence) libraries. The authors' approach involves first finding database access points by searching for JDBC/Hibernate/JPA method calls. Next, each call is sent to an analysis tailored for the specific library. For JDBC a local String expression recovery is performed and, if needed, a call graph analysis is performed to handle inter-procedural cases. The process for Hibernate and JPL goes beyond simple string recovery, since these libraries include support for operating on mapped entity classes rather than executing SQL queries directly. Once the analysis for each particular library recovers complete SQL strings, the authors use a SQL parser along with the database schema to determine which schema elements are executed at a particular location. Our approach does not relate calls to schema elements, but this is something we could add in the future. Their approach also does not account for unresolved dynamic query fragments, which leads to some cases where executed queries that use these features are missed by the analysis.

Other work has been done involving the use of static and dynamic analysis to extract information about database queries in software systems. "Query smells" in Java programs were examined by Nagy and Cleve [12]. In this paper, they present a static code smell detector that analyzes the Java code along with the database schema and data in the database to detect potential faults or inefficient usage of the database. The authors were able to implement detectors for SQL antipatterns

that had been described in the book *SQL Antipatterns* [13] as common mistakes made during database programming. Cleve and Hainaut used aspect-based tracing to extract information about program behavior and database structure [14]. In this work, the authors generate traces to capture query executions and the results returned by query executions and use this information to facilitate software and database reverse engineering.

Noughi et al. [15] used tracing to extract information on query executions in order to determine which queries are executed during a given program execution scenario. The authors further analyze this information to both determine the dependencies between successive queries and which schema elements are accessed during a specific program execution scenario. The authors also describe the implementation of this approach into DAViS [16], a plugin of the database design and evolution tool DB-MAIN, and a proof-of-concept application of the tool to a e-learning application. Alafi et al. also used tracing through a tool called WAFA [17], which uses the TXL programming language [18]. In this approach, PHP web applications are instrumented to extract information about SQL statements, page access, server environment variables, cookies, and session management functions.

Ngo and Tan [19] present a static analysis for extracting database interactions from PHP web applications. Their approach involved first creating a reduced Control Flow Graph (called an Interaction Flow Graph, or IFG) of the PHP system by removing all nodes that do not affect the execution of database interactions in the system. All paths through the IFG that contain database query execution nodes are then symbolically executed to extract all possible database interactions in the program. The work of Gould et al. [20] involved a string analysis that produces a finite state automaton that provides a conservative estimate for all the possible queries that could be input to a location in Java code that invokes database queries, called a *hotspot*.

This automaton is then transformed using a context-free reachability algorithm to remove type errors using the database schema.

Additional approaches have explored detecting errors in queries, assessing the quality of code that invokes and constructs queries, and finding security vulnerabilities in systems with embedded database queries. van den Brink et al. examined the quality of embedded SQL in host code [21]. In the authors' approach, strings that begin with SQL statement keywords such as `SELECT`, `INSERT`, `DROP`, etc. are collected. Next, each string is used in a backward flow analysis to identify the variables in which the query string is assigned. Then, a forward flow is used to reconstruct all pieces of the query. These extracted queries, and their relationships to the host program, are then used to assess the quality of the use of embedded SQL statements. In [22], Wassermann et al. explore dynamically constructed database queries in Java applications. In this work, the authors created a static analysis to check for type errors in SQL strings that could lead to SQL runtime exceptions.

Chapter 6

Future Work and Conclusion

6.1 Future Work

As discussed in Section 3.4, cases where parts of query text occur in query holes are currently represented by `partialStatement` instances. This approach was intended to be a placeholder, and exploring options for extracting more information from these cases is part of our future work. One approach we have thought of is using parser error recovery techniques to handle these cases, while modifying the abstract syntax definition in Rascal to have a better representation of these statements with query text holes.

Another area for future work is to expand our corpus to additional systems. This will allow us to further validate the patterns and empirical data that we have reported here, and possibly identify new patterns and trends in the construction of database queries in PHP systems. As discussed in Section 4.4, expanding the corpus will also allow us to get a larger number of query models that differ based on control flow (QCP3b), which will let us get more accurate data about the clauses in which these cases differ. We also recently adapted our analysis to work better with two additional systems in the corpus that were left out due to performance problems (OcoMon and SchoolERP), as discussed in Section 4.1, but did not have enough time to rerun the analysis for these.

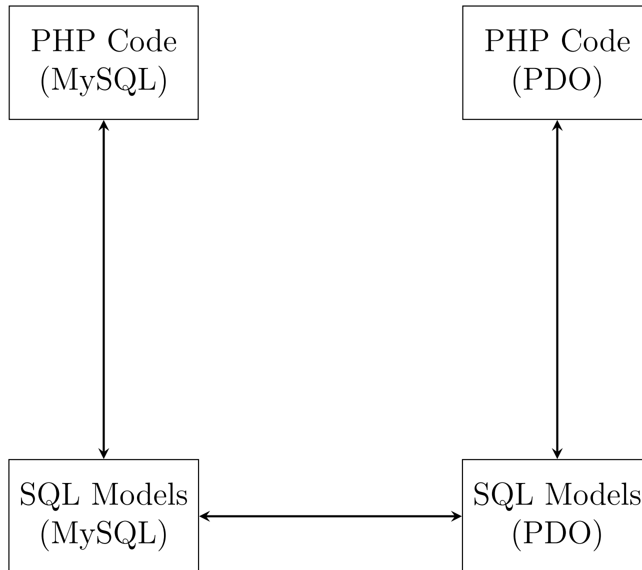

Figure 6.1: Proposed Transformation Approach.

Expanding the `YieldInfo` to be more descriptive is also a possible area for future work. In our approach, we extracted information about query clauses, and assigned them *same*, *some*, *different*, and *none* nodes. A similar approach could also be applied to conditions within `WHERE` clauses to extract information about what kinds of conditions developers include in their database queries. Other information such as which SQL functions are used could also be explored.

Another possible area for future work is expanding the modeling tool to work with other database libraries (MySQLi, PDO, etc). This could lead to more accurate results on how embedded queries are constructed in PHP systems, since the analysis would no longer be restricted to systems that use the original MySQL library. Patterns between libraries could also be compared to see if the specific library being used has any affect on how developers construct embedded database queries.

Finally, the empirical data from this work was conducted to provide a foundation for future work on building program understanding and transformation tools to renovate PHP code to use more modern database libraries. Being able to model other

libraries (as discussed in the previous paragraph) is an important first step to this. Figure 6.1 gives an overview of our proposed future approach for transforming systems that use the original MySQL library to use the PDO library. In this approach, we propose first expanding our modeling tool to model embedded queries executed using the PDO library. Next, semantics-preserving transformations between MySQL SQL models and PDO SQL models will be defined. Finally, a model-to-text transformation will be used to transform PDO SQL Models to PHP code using the PDO library.

6.2 Conclusion

In this research, we presented our tool for modeling embedded queries in PHP systems. After extracting models from open source systems, we were able to identify Query Construction Patterns that developers commonly use to create database queries. Data for PHP language features used in dynamic query parts, counts of the use of each SQL statement and clause in practice, and initial figures on how SQL statements differ based on control flow were also extracted. We believe this information lays a solid foundation for future empirical studies, as well as for the construction of tools for understanding and renovating PHP systems with embedded database queries.

BIBLIOGRAPHY

[1] M. Hills, P. Klint, and J. J. Vinju, "An Empirical Study of PHP Feature Usage: A Static Analysis Perspective," in *Proceedings of ISSTA 2013*. ACM, 2013, pp. 325–335.

[2] M. Hills, "Evolution of Dynamic Feature Usage in PHP," in *Proceedings of SANER 2015*. IEEE, 2015, pp. 525–529.

[3] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of SCAM 2009*. IEEE, 2009, pp. 168–177.

[4] M. Hills and P. Klint, "PHP AiR: Analyzing PHP Systems with Rascal," in *Proceedings of CSMR-WCRE 2014*. IEEE, 2014, pp. 454–457.

[5] M. Hills, P. Klint, and J. J. Vinu, "Enabling PHP Software Engineering Research in Rascal," *Science of Computer Programming*, vol. 134, pp. 37–46, 2017.

[6] I. Rucareanu, "PHP: Securing Against SQL Injection," Master's thesis, University of Amsterdam, 2013.

[7] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks," in *Proceedings of ICSE 2009*. IEEE, 2009, pp. 199–209.

[8] D. Anderson and M. Hills, "Query Construction Patterns in PHP," in *Proceedings of SANER 2017*. IEEE, 2017, pp. 452–456.

[9] ——, "Supporting Analysis of SQL Queries in PHP Air," in *Proceedings of SCAM 2017*. IEEE, 2017, pp. 153–158.

[10] C. Nagy, L. Meurice, and A. Cleve, "Where Was This SQL Query Executed? A Static Concept Location Approach," in *Proceedings of SANER 2015*. IEEE, 2015, pp. 580–584.

[11] L. Meurice, C. Nagy, and A. Cleve, "Static Analysis of Dynamic Database Usage in Java Systems," in *Proceedings of CAiSE 2016*, ser. LNCS, vol. 9694. Springer, 2016, pp. 491–506.

[12] C. Nagy and A. Cleve, "A Static Code Smell Detector for SQL Queries Embedded in Java Code," in *Proceedings of SCAM 2017*, 2017, pp. 147–152.

[13] B. Karwin, "SQL antipatterns; avoiding the pitfalls of database programming," *Scitech Book News*, 2010.

[14] A. Cleve and J. Hainaut, "Dynamic Analysis of SQL Statements for Data-Intensive Applications Reverse Engineering," in *Proceedings of WCRE 2008*. IEEE, 2008, pp. 192–196.

[15] N. Noughi, M. Mori, L. Meurice, and A. Cleve, "Understanding the Database Manipulation Behavior of Programs," in *Proceedings of ICPC 2014*. ACM, 2014, pp. 64–67.

[16] L. Meurice, "Visualizing SQL execution traces for program comprehension," Master's thesis, University of Namur, 2013.

[17] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "WAFA: Fine-grained Dynamic Analysis of Web Applications," in *Proceedings of WSE 2009*. IEEE, 2009, pp. 141–150.

[18] J. R. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[19] M. N. Ngo and H. B. K. Tan, "Applying static analysis for automated extraction of database interactions in web applications," *Information & Software Technology*, vol. 50, no. 3, pp. 160–175, 2008.

[20] C. Gould, Z. Su, and P. T. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," in *Proceedings of ICSE 2004*. IEEE, 2004, pp. 645–654.

[21] H. van den Brink, R. van der Leek, and J. Visser, "Quality Assessment for Embedded SQL," in *Proceedings of SCAM 2007*. IEEE, 2007, pp. 163–170.

[22] G. Wassermann, C. Gould, Z. Su, and P. T. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," *ACM TOSEM*, vol. 16, no. 4, 2007.