Understanding Database Usage in PHP Systems: Current and Future Work

Mark Hills
East Carolina University, Greenville, NC, USA
mhills@cs.ecu.edu

***Abstract*—Most PHP applications use databases to store application data. To manipulate this data, developers create queries in their programs using database APIs. These queries are constructed in source code using a combination of static query text and dynamically computed values, including values based on user input. In this abstract, we briefly explain our ongoing work on understanding how developers use PHP language features and query libraries to create database queries, and on how query languages are used in source code.**

I. Introduction

Relational databases play a central role in many dynamic websites. These web applications use data access APIs to interact with databases from different vendors, such as MySQL[1] and PostgreSQL.[2] For PHP, commonly-used data access APIs for MySQL include the original MySQL library[3] and the MySQL Improved (MySQLi) library.[4] Other languages also have data access APIs, such as JDBC[5] in Java. These libraries generally accept database commands, referred to here as just *queries*, as strings containing expressions written using the Structured Query Language (SQL), a standard query language commonly used by relational databases. These strings are built using a combination of string literals and programming language expressions: the literals represent static parts of the query, provided in advance by the developer, while expressions represent dynamic information, such as data entered into a form by a user of the website, optional parts of a query selected along different control flow paths, or (more generally) values computed using application code at runtime.

To help developers understand how database query APIs are used in actual PHP code, we are building static models of database queries and query construction operations, and then using these models to identify idiomatic patterns for building database queries. This abstract describes our current results, and future plans, around this research topic. Section II describes our research method, while Section III presents our initial results and current corpus. Section IV discusses future plans for this research. The work presented here includes, and builds on, earlier published work [1], [2].

[1]https://www.mysql.com
[2]https://www.postgresql.org/
[3]http://php.net/manual/en/book.mysql.php
[4]http://php.net/manual/en/book.mysqli.php
[5]http://www.oracle.com/technetwork/java/javase/jdbc/index.html

II. Research Method

PHP AiR [3], [4], a framework for PHP Analysis in Rascal, is used to perform the analysis used to identify the query APIs and query building operations found in PHP systems. It is also used to compute the reported results to ease reproducibility. The PHP AiR framework is written in Rascal [5], a meta-programming language focused on program analysis and transformation, and makes heavy use of Rascal language features such as pattern matching, source locations (indicating regions in source code), and data types such as relations, lists, and maps. PHP AiR adds support for parsing PHP code, building control flow graphs and analyses over these graphs, and simplifying PHP expressions using rules about PHP constants, operations over strings, and commonly-used library functions. The PHP AiR framework[6] and libraries specifically for query analysis[7] are both available on GitHub under an open-source license.

Figure 1 provides an overview of the process used to build a model and extract and parse the possible queries used in a database library call at a specific program point. The first step in the process is to parse the PHP scripts that make up the system being analyzed. This is done using our fork[8] of an open-source PHP parser[9], written in PHP. This generates a value of Rascal type `System`, which represents the parsed system and includes abstract syntax trees (ASTs) for each of the parsed PHP scripts.

Using these ASTs, and the location of a call to a database query API function such as `mysql_query` (part of the original MySQL API), the Model Builder statically extracts a model representing the actual SQL queries that could be passed to this function. This model differentiates between static and dynamic parts of the queries, provides links from names used in the query to the values that these names represent, and includes information about the conditions under which certain parts of a query are present. To get each of the possible queries represented by the model, the Query Yields Generator builds a representation—here called a *yield*—of each query, with each yield reflecting a specific query executed based on a specific program path from the start of the surrounding context (e.g., the function containing the API call) to the API call. As with

[6]https://github.com/cwi-swat/php-analysis
[7]https://github.com/ecu-pase-lab/mysql-query-construction-analysis
[8]https://github.com/cwi-swat/PHP-Parser
[9]https://github.com/nikic/PHP-Parser/

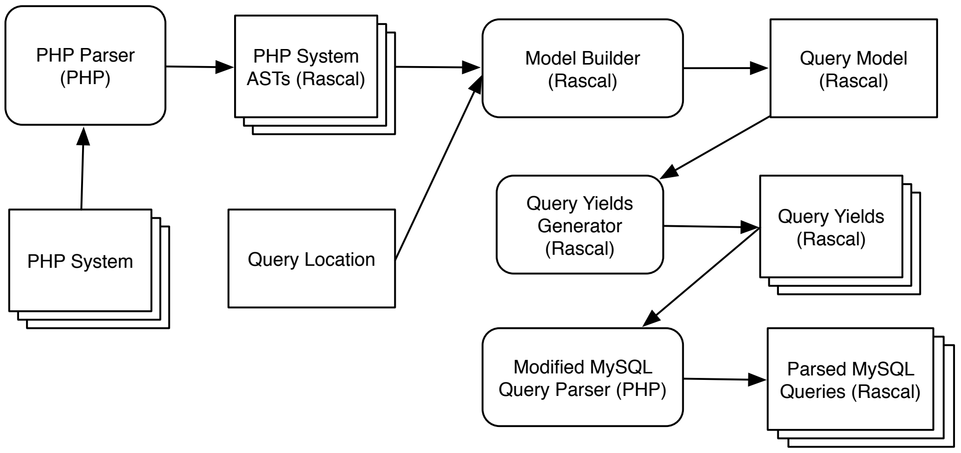

Fig. 1. Overview: Extracting Query Models and Parsing Modeled Queries.

the model, each yield includes a combination of static and dynamic pieces.

To parse each of these queries, we use our fork[10] of the open-source MySQL parser[11] released as part of phpMyAdmin,[12] a web-based tool for managing MySQL databases. This parser has been modified to properly parse partial queries, with special symbols used to represent query text provided dynamically. The result of running the parser on each yield is to generate a Rascal AST for each query. At this time only queries using the MySQL dialect of SQL are supported.

III. Initial Results and Expanded Corpus

Our initial work [1] analyzed 10 systems that use the MySQL API. In this work, we identified Query Construction Patterns (QCPs) using a combination of manual inspection and static analysis. This work focused only on the original MySQL library, performed a limited analysis of the dynamic parts of queries using regular expression matching over query code, and did not describe which features of SQL were used in the queries or how these queries differed from one another.

Our current work is expanding this earlier work in all these areas, including a focus on a more diverse collection of database APIs. We are also now using a larger corpus of existing systems. To build this corpus, we first identified the top 1000 most starred PHP projects as of April 2018 using an extract from GHTorrent [6] available through Google's BigQuery service.[13] A project is classified as a PHP project if at least half the code, measured in bytes, is written in PHP. This corpus currently includes 78 systems that use either the MySQL or MySQLi API, with 91,839 PHP files and 10,657,061 lines of PHP code. We plan to extend this further to include the PDO library,[14] which provides an object-based abstraction for database operations, and Doctrine,[15] a popular object-relational mapper (ORM) for PHP applications.

[10]https://github.com/ecu-pase-lab/sql-parser

[11]https://github.com/phpmyadmin/sql-parser

[12]https://www.phpmyadmin.net/

[13]https://cloud.google.com/bigquery/

[14]https://www.php.net/manual/en/book.pdo.php

IV. Future Work

Building on these initial results, our future work involves both looking more deeply at common database query construction idioms, and leveraging these idioms to build more precise code transformations tools. For the first, we would like to use the new corpus to expand the number of studied systems. We also plan to further improve the existing analysis, and look in more detail at the dynamic portions of queries, better categorizing where they are used inside the queries and where the information needed to compute these dynamic query portions comes from in the program (e.g., from local computations vs. from function parameters or global arrays). For the second, we want to build code transformation tools for evolving existing systems to use newer, more modern database APIs, using the information about these idioms to guide tool development (e.g., which cases are the most important to precisely handle first). We also believe this will be helpful in building code comprehension tools, which can inform developers of the links between queries and calls to database APIs and also provide models of these queries that can improve developer understanding of existing code.

Acknowledgments

This research is supported in part by NSF grants CCF-1262933 and CCF-1560037, and is based on work performed by David Anderson as part of his MS thesis.

References

[1] D. Anderson and M. Hills, "Query Construction Patterns in PHP," in *Proceedings of SANER 2017*. IEEE, 2017, pp. 452–456.
[2] ——, "Supporting Analysis of SQL Queries in PHP AiR," in *Proceedings of SCAM 2017*. IEEE, 2017, pp. 153–158.
[3] M. Hills and P. Klint, "PHP AiR: Analyzing PHP Systems with Rascal," in *Proceedings of CSMR-WCRE 2014*. IEEE, 2014, pp. 454–457.

[15]https://www.doctrine-project.org/

[4] 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.

[5] 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.

[6] G. Gousios, “The GHTorrent dataset and tool suite,” in *Proceedings of MSR 2013*. IEEE, 2013, pp. 233–236.