Abstract—Localizing and fixing software faults is an important maintenance task. In a dynamic Web application, localizing the faults is challenging due to its dynamic nature and the interactions between the application and databases. The faults could occur in the statements in the host program or inside the queries that are sent from the application to be executed in the database engines. This paper presents SQLook, a novel database-aware fault localization method that is able to locate output faults in PHP statements of a dynamic Web application as well as in SQL queries. In SQLook, a PHP interpreter is instrumented to execute an SQL query and to monitor the evaluation of those SQL predicates to determine if they affect the output process of individual data records. It performs row-based slicing across PHP statements and SQL queries to record the entities that are involved in the output of each data row. Our empirical evaluation shows that SQLook can achieve higher accuracy than the state-of-the-art database-aware fault localization approach.

Keywords—Fault Localization; Database-aware; Dynamic Web

I. INTRODUCTION

A dynamic Web application is often written in a language such as PHP or ASP that communicates with the databases to retrieve data, processes and displays them on the client-side browsers. In such a program, there exist program statements that are responsible for interacting with the databases, which are called database-interaction points. Before an interaction point, the program constructs a string query from literals, variables, functions’ returned values, etc. That query is written in a query language supported by the database (e.g. SQL).

As in other types of application, dynamic Web applications have failures as well. It was reported that there are common failures in a dynamic Web application that are caused by the interaction and passing of data between the application and the database [2]. Many researchers developed automated fault localization methods for traditional, non-database applications [1], [4], [5] and for data-centric applications (i.e. single-language database programs) [3], [6]. However, little attention has been paid to database-aware fault localization, i.e. taking into account the interaction between the applications and databases via queries, in dynamic Web applications.

Clark et al. [2] introduce an approach for database-aware fault localization in dynamic Web applications. It uses Tarantula [4] to assign each statement a suspiciousness score computed based on the percentage of passing/failing test cases executing that statement. However, it has a key restriction on its effectiveness. It requires users to provide passing/failing test cases that must produce different unique queries at runtime, i.e. produce different unique SQL structures in which one structure is exercised by all failing test cases. This is too strict since all passing/failing tests often create SQL queries with the same structure and only literal values vary for each query.

This paper presents SQLook, a novel method for database-aware fault localization in PHP-based Web applications with SQL support. We focus on the output errors caused by incorrect queries with erroneous WHERE clauses or by the manipulation of the queries’ result in PHP. We use a row-based test case technique in which instead of using the entire output of data records as a test case, we leverage the presence/absence of individual data rows and their expected values to create more test cases, called row-based test cases. Specifically, an input of a PHP program and a present/absent data row in the output forms a row-based test case. If a presence/absence is expected, the test case is a passing one, and vice versa. Instead of passing the control to the database engine to run a command as in Clark et al. [2], we instrument into a PHP interpreter the code to execute the SQL query and to monitor the evaluation of the predicates of a WHERE clause to determine if they affect the output of individual data records. Based on whether a WHERE part is exercised frequently by passing/failing tests, it is assigned with a Tarantula suspiciousness score [4]. Since our row-based test cases are for individual data rows, to compute the scores for PHP statements, SQLook performs row-based slicing across PHP statements and SQL parts to record the PHP statements that are exercised in the output of a row. It returns a ranked list of suspicious PHP entities and SQL queries.

II. MOTIVATION

Web applications often use data stored in databases, such as in the table shown in Table I. The Users database table contains four rows (data records). The five columns of the table indicate five attributes associated with each record. For example, the first record has the information on a user with the attributes $\text{ID}=1$, $\text{Name}=Alice$, $\text{Age}=20$, $\text{Gender}=Female$, and $\text{Country}=USA$. This data can then be accessed or updated from the Web application through SQL queries. For instance, the SQL query “SELECT Name FROM Users WHERE Age = 20” returns a result set containing the names of all users whose age is 20.

As a Web application interacts with a database through SQL queries, database-specific failures can occur. Figure 1 shows an example of a PHP function that produces incorrect output
TABLE I
THE USERS DATABASE TABLE

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>Gender</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>20</td>
<td>Female</td>
<td>USA</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>20</td>
<td>Male</td>
<td>Canada</td>
</tr>
<tr>
<td>3</td>
<td>Carol</td>
<td>25</td>
<td>Female</td>
<td>Canada</td>
</tr>
<tr>
<td>4</td>
<td>Daniel</td>
<td>30</td>
<td>Male</td>
<td>USA</td>
</tr>
</tbody>
</table>

A PHP function with an SQL query error on line 3:

```php
function displaySearchResults($age, $gender, $country) {
    $sql = "SELECT Name FROM Users WHERE Age >= $age AND Gender = "$gender OR Country <> "$country ";"
    $result = mysql_query($sql);
    while($row = mysql_fetch_array($result)) {
        echo $row['Name'] . '<br />';}
}
```

Test Cases Sus.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Sus.</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0.5</td>
</tr>
<tr>
<td>P</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Fig. 1. An PHP function with an SQL query error

values due to an error in its SQL query. The purpose of the function is to display the names of the users from the Users table (Table I) that satisfy a searching criteria (by age, gender, or country). First, the connection to the database is established (lines 1-2, Figure 1). The $sql variable (line 3) contains the SQL query that retrieves the users’ names for a given search input. This query is then sent to the database server to be executed via the PHP function mysql_query (line 4), and the returned result set is stored in the variable $result. Finally, the code on lines 5-7 is used to loop through the records in the result set and display the corresponding names of the users found via the search. Note that this function contains an SQL fault: on line 3, the operator in the last predicate Country <> ’$country’ of the SQL query should be ‘=’ instead of ‘<>’.

Due to that error in the SQL query, the function does not display the expected results. Figure 2 shows the actual and expected output values given the search input $age=25, $gender=’Female’, $country=’USA’. In the correct SQL query, the three-predicate condition determining the values in the returned result set is: Age >= 25 AND Gender = ’Female’ OR Country = ’USA’. Since the actual query has a fault in the last predicate, there is a mismatch between its actual and expected outputs. As seen, Alice’s and Daniel’s names are expected to be found in the result, but are not displayed. Meanwhile, Bob’s name is included in the actual output although it must not.

Given such mismatch, it is not obvious which part of the program accounts for the error. In this example, the fault lies at a predicate of an SQL query, whereas the Web application is written in PHP. Moreover, the actual execution of the SQL query occurs at the database server, which is separate from the main PHP program where the SQL query is created. Thus, the process of localizing this type of fault needs to be database-aware, namely taking into account the interaction between the Web application and the database via queries.

The state-of-the-art tool in Clark et al. [2] for database-aware fault localization in Web applications monitors SQL queries generated at runtime and computes the suspiciousness scores for those queries and their associated attributes, as well as the statements in the main program. Similar to other statistical fault localization methods (e.g. Tarantula [4], Ochiai [1]), the idea in computing such scores is that if a program entity is exercised by more failing tests than passing ones, it more likely contributes to the failure and assigned with a higher score.

Compared with the previous fault localization approaches, Clark et al.’s method [2] is database-aware in that it considers SQL queries or SQL attributes as program entities and also computes their suspiciousness scores. For example, Figure 3 illustrates the score computation for the program entities in Figure 1 including SQL queries. Line 4+ shows the query executed by the PHP statement mysql_query on line 4 at runtime, with the question marks indicating literal values (numbers/strings). If the mysql_query statement executes multiple unique SQL queries (each with a different set of attributes) in different runs, the scores of individual queries will be computed.

That computation is based on the idea that if some of unique SQL queries are executed by more failing test cases than passing ones, they will have higher scores. This strategy is useful when there are multiple unique SQL queries that expose different behaviors in passing and failing test cases. However, in practice, a PHP mysql_query statement often executes only one SQL query, in which the set of attributes is fixed, and the concrete SQL queries in different executions have the same structure and vary only at the literal values. For example, the unique query in this example (line 4+ of Figure 3) is “SELECT Name FROM Users WHERE Age >= ? AND Gender = ? OR Country <> ?”, with the set of attributes (Name, Age, Gender, Country).
Since there is one unique SQL query executed by the PHP `mysql_query` statement, the coverage of the SQL query in the passing and failing test cases is the same as the `mysql_query` statement, and therefore, its suspiciousness score does not provide further information about the location of the fault. Let us describe the score computation to illustrate this limitation. In Figure 3, the bullets indicate the program entities that are exercised by a given test case. At the bottom row, the letters `P` and `F` specify a passing and failing test case, respectively. Column `Sus.` shows the suspiciousness score `S(e)` for a program entity `e` using the Tarantula ([4]) metric:

$$S(e) = \frac{\text{Passed}(e)}{\text{TotalPassed}} + \frac{\text{Failed}(e)}{\text{TotalFailed}}$$

where `Passed(e)` is the number of passing test cases that execute `e`, `Failed(e)` is the number of failing test cases that execute `e`, and `TotalPassed` and `TotalFailed` are the respective total numbers of passing and failing test cases.

Although there is a fault in the SQL query, its suspiciousness score (0.5) is the same as the PHP `mysql_query` statement that executes it. In fact, all the program entities always have the same score even when a different suspiciousness metric is used, since the same set of program entities is executed in every passing or failing test case (regardless of the test suite). From those suspiciousness scores, no further information on the fault is gained. This limitation motivated us to develop SQLook, a method to better localize database-specific faults.

III. LOCALIZING FAULTY SQL QUERIES

In a PHP Web application, program faults may be found in regular PHP statements or those that interact with the database, called database-interaction points (e.g., line 4 of Figure 1). The goal of this step is to localize these faults, specifically to decide if a database-interaction point contains a fault(s) at its `WHERE` clause or not. To do that, SQLook uses Tarantula [4] to compute the suspiciousness scores for all entities in PHP and in SQL `WHERE` clauses. To avoid the issues as in Clark et al. [2]'s approach, we have following key design strategies:

1. **Row-based test cases.** Instead of viewing the input and entire output from the database as one test case, we analyze individual data records in the actual and expected outputs to create row-based test cases. For example, in Figure 2, we have 4 row-based test cases: 1) the input `$age=25`, `$gender='Female'`, `$country='USA'` and the output of Alice’s record, 2) that input and the absence of Bob’s, 3) that input and the output of Carol’s, and 4) that input and the output of Daniel’s.

2. **Monitoring the execution of PHP and SQL entities via instrumentation.** The suspiciousness scores are given to PHP statements and SQL `WHERE` clause(s). Instead of passing the control to the database engine to execute an SQL command as in [2], SQLook instruments into an PHP interpreter the code to execute the SQL command (Figure 4) and to observe the evaluation of the `WHERE` clause with respect to every row-based test case. Because the `WHERE` clause’s value decides if the output of a row-based test case is present or not, SQLook needs to record if that clause is evaluated to True or False.

![Fig. 4. Instrumented PHP interpreter to monitor the execution of SQL queries](image-url)

Specifically, the operations that fulfill an SQL `SELECT` query consist of the following: (a) retrieving data from the database table(s) specified by the `FROM` part of the SQL query, (b) extracting the data records that satisfy the criteria specified by the `WHERE` condition of the SQL query, and (c) projecting the set of columns (attributes) given in the `SELECT` part of query into the final result set. As an example, the SQL query `SELECT Name FROM Users WHERE Age >= 25 AND Gender = 'Female' OR Country <> 'USA'` retrieves the names of all the users from the ‘Users’ table that meet the condition `Age >= 25 AND Gender = 'Female' OR Country <> 'USA'`. Our instrumented interpreter re-implements these three operations in four steps (see the instrumented code in Figure 4). Detailed instrumentation execution is in Figure 5.

From that, it knows which part (True or False) of a `WHERE` clause is exercised in a row-based test case. Thus, the suspiciousness metric can now be applied not only to PHP code, but also to the deeper level of the two parts of a `WHERE` clause.

3. **Row-based Slicing across PHP and SQL.** Since SQLook uses row-based test cases, it needs to record the PHP statements that are exercised in the execution of such a test case, i.e., PHP statements that are involved in the output of a data record (i.e., a row). To do that, during the monitoring in step (2), it computes the forward slice corresponding to each data row across the PHP statements and the two parts of `WHERE`.

4. **Computing Suspiciousness Scores.** Based on the monitoring results, SQLook computes the suspiciousness scores for all program entities, including the `WHERE` conditions of SQL queries and other statements in the PHP program. Figure 6 illustrates the score computation for the example in Figure 1. A test case is marked as Passed(P) if the presence (or absence)
of the corresponding record in the actual output is as expected; otherwise, it is marked as Failed(F). For example, Alice does not appear in the actual output, which is not expected; thus record 1 is a Failed test case. The only Passed test case is record 3, where Carol is output as expected. The bullets for the statements indicate whether the entities are included in the slice for the corresponding test case, which is established when SQLook monitors the execution trace of row-based test cases.

As seen in Figure 6, the False part of the WHERE expression on line 4b has a high suspiciousness score (1.0), indicating that the program state when the WHERE of the SQL query is evaluated to False is likely incorrect (i.e., the result set does not contain the corresponding record whereas the record is expected to be included). Also, the suspiciousness score of the WHERE expression corresponding to the False case is higher than the score of any other program entity, which suggests that the predicates in the WHERE clause of the SQL query are most likely to contain an error. In this example, the fault is located at the last predicate of the WHERE condition (the operator in Country <> 'USA' should be '=' instead of '<>'). Thus, the scores are useful in localizing faulty SQL queries.

IV. EMPIRICAL EVALUATION

In this experiment, for each subject system, we manually seeded two types of database-related faults: (1) SQL faults in the WHERE clause of SQL queries, and (2) PHP faults that affect the output data retrieved from a database query. For (1), we mutated the operators of the predicates in the WHERE clauses, while for (2), we used the same mutation strategy as in the experiment in Clark et al. [2]. Each mutant program has a single fault. Then, we created a failed test case that was resulted from the fault in either an SQL query or the PHP code. Table II shows the result. Column Mutants shows the number of created mutants. Column SQL faults% Rank shows the percentage of statements in the execution trace that a fixer need not examine by using SQLook’s ranked list of suspicious statements. For example, for PHP faults, s/he does not have to examine 86-98% of the statements in the execution trace. Since the faulty SQL query is ranked at the top by SQLook, column SQL faults% Rank shows 100% for all three systems.

Comparison. Since SQLook uses information about individual rows in the test case, SQLook is able to rank the likelihood of faulty entities with one test case only. In contrast, the state-of-the-art approach, Clark et al. [2], was designed to require multiple test cases to localize faults. For comparison, we took 106 randomly sampled SQL queries and manually examined if a query involves a unique set of attributes and varies only at the literal values. To do that, we first divided the queries into groups according to the numbers of their predicates. The number of sampled queries in each group is proportional to its size. In Table III, the first columns show the numbers of SQL queries with the corresponding numbers of predicates. Columns Checked and Unique show the numbers of queries that were checked and have unique sets of attributes, respectively. As seen, among 106 random query samples, 101 of them involve a unique set of attributes. Clark et al. [2] could not give those SQL statements higher suspicious scores even with multiple test cases, thus, could not locate those 101 faults. In contrast, SQLook was able to rank all of them at the highest.

V. CONCLUSIONS

SQLook is able to detect output faults in PHP statements and in SQL queries. The key solutions include (1) an instrumented PHP interpreter to monitor the execution of a query and the evaluation of predicates to see if they affect the output of records, and (2) row-based slicing across PHP and SQL to record entities in the output process of each row. Our empirical evaluation shows that SQLook can achieve high accuracy.

ACKNOWLEDGMENT

This project is funded in part by US National Science Foundation (NSF) CCF-1018600 and CNS-1223828 awards.

REFERENCES