Localizing SQL Faults in Database Applications

Sarah R. Clark*, Jake Cobb*, Gregory M. Kapfhammer†, James A. Jones‡, and Mary Jean Harrold*
*Georgia Institute of Technology, {sclark|jcobb|harrold}@cc.gatech.edu
†Allegheny College, gkapfham@allegheny.edu
‡University of California, Irvine, jajones@ics.uci.edu

***Abstract*—This paper presents a new fault-localization technique designed for applications that interact with a relational database. The technique uses dynamic information specific to the application's database, such as Structured Query Language (SQL) commands, to provide a fault-location diagnosis. By creating statement-SQL tuples and calculating their suspiciousness, the presented method lets the developer identify the database commands and the program statements likely to cause the failures. The technique also calculates suspiciousness for statement-attribute tuples and uses this information to identify SQL fragments that are statistically likely to be responsible for the suspiciousness of that SQL command. The paper reports the results of two empirical studies. The first study compares existing and database-aware fault-localization methods, and reveals the strengths and limitations of prior techniques, while also highlighting the effectiveness of the new approach. The second study demonstrates the benefits of using database information to improve understanding and reduce manual debugging effort.**

I. Introduction

Real-world software applications have large and complicated code bases. As such, finding the faults that cause the programs to fail, through the process of *fault localization*, can be both time consuming and difficult. To reduce the time and effort required to locate the faults, researchers have developed techniques, such as those that use statistical methods and coverage information, to automate the fault localization. These methods typically select a program entity, such as a statement or a branch in the source code, and utilize coverage information recorded from program executions, along with statistical methods, to infer or correlate this entity with program failure (e.g., [2], [6], [9], [11], [13], [14], [17]).

Studies performed using these fault-localization techniques demonstrate their effectiveness in reducing the code that developers must inspect to locate the fault (e.g., [2], [10], [14]). Yet, the programs on which these studies have been performed are relatively small and written in a single language, and thus, do not consider other important components of the system with which the program interacts. Consequently, studies have not evaluated the effectiveness of these existing methods for systems that interact with complex external components.

One such external component is a database—an essential part of many software applications [7]. Brooks and colleagues [4] report that the most common types of errors in three real-world industrial systems result from data-access and handling, including interactions between the application and the database. They also report that another common type of error relates to the way data is passed among subsystems or between the system and the database.[1]

[1]Private correspondence with the authors.

The most common type of database is a relational database,[2] with which applications interact using the Structured Query Language (SQL) to select, update, insert, and delete data [7]. Not only can SQL commands contain control and data dependencies, but the results of their executions often determine further calculations and execution paths of the program. As such, these results can potentially propagate the effects of an SQL fault to other parts of the application. Existing methods are limited in the information that they can provide to the developer because they do not address applications' use of relational databases.

Database-specific faults include problems with the structure of the database (i.e., *schema faults*), incorrect values stored in the database (i.e., *data faults*), and problems with the SQL commands used to query and modify the database (i.e., *SQL faults*). Although the overall goal of our work is to develop techniques capable of localizing all these types of faults, our first technique, and the one presented in this paper, is a new approach to fault-localization that targets SQL faults. Our database-aware method monitors SQL commands executed by database applications and uses the observed database interactions to assist in debugging the program in two ways. First, the technique helps developers discover the specific SQL calls that are made at runtime, which lets developers study and reason about the contents of the commands. Second, the technique informs the statistical calculation of the database commands and attributes that are most likely to be the cause of program failures, thus, giving the developers guidance as to where to focus their attention during testing and debugging.

Our technique creates *statement-SQL tuples* by combining executed SQL command information with program locations (i.e., statements) at which these commands are executed. The method then computes the suspiciousness of each program statement and each statement-SQL tuple, and provides a ranking of them both. If multiple SQL commands are executed at a particular location, our technique further decomposes the commands to produce *statement-attribute tuples* and assigns a suspiciousness to each to help the developer identify which part of the SQL query is most associated with program failure.

The main benefit of our method is that, unlike previous approaches to fault-localization, it uses database-specific information to rank SQL commands in the program along with their associated attributes. Thus, the technique can help developers locate faults in a database application more precisely than

[2]A survey of developers that appeared in InfoWorld in September 2003 reported that 89.2% of the respondents indicated that they use relational databases [1].

existing approaches that consider only program statements. Another benefit of our technique is that it provides a link between program statements and the SQL commands that they executed. The developer can use this information to discover the SQL commands that were generated by the program, find errors within these commands, and discover the places in the program where they were invoked. A third benefit of our technique is that it serves as a starting point for additional research on fault localization for both database applications and programs with other kinds of external components, such as configuration files and network communication. Thus, it can help in addressing the complexities of real-world software.

This paper also presents the results of empirical studies that evaluate our technique when applied to three database applications. The first study investigates how well our database-aware fault-localization approach performs compared to an existing statistical fault-localization technique. The study reveals that, although the existing technique performs well on faults related to the Java code, it is not always as effective for database-specific faults. The second study examines the qualitative value of the additional database-specific information that the new method provides to the developer. This study shows that, depending on the database application's structure, this additional information can be extremely valuable.

The main contributions of this paper are:

- The first database-aware fault-localization technique, which includes information from SQL calls made by the application, and thus, uses database-related information in computing suspiciousness for fault localization.
- A prototype database-aware fault-localization system that provides the developer with a ranked list of suspicious entities as well as details about the SQL commands executed by the database application.
- Empirical studies that compare our new technique with an existing fault-localization method, and illustrate (1) the limitations of these existing approaches in providing information related to faults in the database commands and (2) the improvement over existing fault-localization techniques that can be achieved with database-aware approaches. Our results show that at least one such technique—the one we present in this paper—provides up to 94.6% improvement in fault-localization effectiveness over existing non-database-aware techniques. Such results highlight the need for research in this area.

II. Relational Databases and Motivating Example

This section presents background on relational databases as well as an example illustrating the limitations of applying existing fault-localization techniques to database applications.

A. Relational Databases

The most common type of database is the relational database [7]. Figure 1 shows a simple relational database consisting of one table, SALE, that records product sales. Database tables have columns or attributes to identify the stored data and rows that contain collections of related data. The SALE table

SALE

MERCHANTID	CUSTOMERID	PRODUCT	PRICE
1	1	Cat Food	$9.99
2	1	Soda	$1.00
2	3	Cheese	$3.99
3	2	Hammer	$5.00
3	3	Nails	$0.50
3	4	Drill	$30.00

Fig. 1: A populated database table.

TABLE I: Sample configuration.

`userType:`	Merchant
`Attributes:`	PRODUCT, PRICE
`WHERE Clause:`	MERCHANTID >= `#userID`
`userType:`	Customer
`Attributes:`	PRODUCT, PRICE
`WHERE Clause:`	CUSTOMERID= `#userID`

contains attributes MERCHANTID, CUSTOMERID, PRODUCT, and PRICE. The second row in the table is $\langle 2, 1, \text{Soda}, \$1.00 \rangle$ where 2 is the MERCHANTID, 1 is the CUSTOMERID, Soda is the PRODUCT, and $1.00 is the PRICE. Applications use SQL to access, modify, add to, and delete data in the database. SQL provides four main commands for interaction with the database: `SELECT` retrieves data from the database; `UPDATE` modifies existing data in the database; `INSERT` adds new data to the database; and `DELETE` removes data from the database.

B. Motivating Example

The first column in Figure 2 contains a code snippet, method `printProductsSold`, that uses the database table in Figure 1 to display sales information about products sold: MERCHANTID, CUSTOMERID, PRODUCT, and PRICE. Each time a product is sold, a record is added to the database table to reflect the sale. The method then prints a list of the products sold. A user of the system can be either a *Customer* or a *Merchant*. The user classification and the user's customer or merchant number are determined at login. The method `printProductsSold` receives information for the user requesting the data in `userType` and `userID`. For this code snippet, the attribute list and the `WHERE` clause for the SQL query are stored externally. Method calls `conf.getAttributes` and `conf.getWhere`, defined elsewhere in the application, retrieve the attribute list and `WHERE` clause, respectively, from the external configuration for the specified user type and ID. Method call `printResultSet(rs)`, also defined elsewhere in the application, prints its parameter.

Table I shows the initial configuration for the example code snippet in Figure 2. In the table, each user type has its own set of data: a list of attributes and a `WHERE` clause. The expression `#userID` in each `WHERE` clause refers to the `userID` passed into the method calls that get data from the configuration. The merchant configuration has a fault in the `WHERE` clause, which reads ">=" instead of "=".

The second through eighth columns of Figure 2 provide information about the test cases: three have *Merchant* as the user type and four have *Customer* as the user type. The top of each column identifies the user, the bullets in the columns indicate which statements were executed by the test case, and

Program Entities		Test Cases							
	`printProductsSold(String UserType, String userID) {`	`Merchant, 1`	`Merchant, 2`	`Merchant, 3`	`Customer, 1`	`Customer, 2`	`Customer, 3`	`Customer, 4`	**Suspiciousness**
1	`String Attributes = conf.getAttributes(userType, userID);`	•	•	•	•	•	•	•	0.53
2	`String whereClause = conf.getWhere(userType, userID);`	•	•	•	•	•	•	•	0.53
3	`String SQL="SELECT "+Attributes+" FROM Sale WHERE" + whereClause;`	•	•	•	•	•	•	•	0.53
4	`PreparedStatement ps = new PreparedStatement();`	•	•	•	•	•	•	•	0.53
5	`ResultSet rs = ps.executeQuery(SQL);`	•	•	•	•	•	•	•	0.53
6	`printResultSet(rs);`	•	•	•	•	•	•	•	0.53
	`}`								
	Pass/Fail Status	F	F	P	P	P	P	P	

Fig. 2: Example program, test cases, and suspiciousness scores.

the bottom of the column shows the execution results with *P* representing passed and *F* representing failed. For example, the first test case, which uses the inputs `Merchant` and `1`, executes all of the statements and fails.

Existing fault-localization techniques that use coverage information about program entities (e.g., statements or predicates) utilize a statistical metric to associate each entity with a suspiciousness score. One example is the Ochiai metric [2]

$$Suspiciousness(s) = \frac{failed(s)}{\sqrt{totalFailed \cdot (failed(s) + passed(s))}} \quad (1)$$

where *failed(s)* is the number of failing test cases that execute s, *totalFailed* is the total number of failing test cases, and *passed(s)* is the number of passing test cases that execute s. Other metrics for computing suspiciousness are also based on passing and failing test cases and the coverage provided by them (e.g., [6], [11], [13], [14], [17]).

The last column in Figure 2 shows the suspiciousness computed using the Ochiai metric [2] for the example code snippet and test suite. Although the example has a fault in the SQL query at statement 5, no information is provided by the fault-localization technique to help developers identify the faulty query. In fact, considering the suspiciousness values given, even statement 5 has the same suspiciousness as the other statements. This limitation of existing fault-localization methods—the lack of suspiciousness score for the database-specific entities—motivated the development of our new database-aware fault-localization technique.

III. Database-Aware Fault Localization

This section defines the terms used for database-aware fault localization, describes the challenges we encountered while developing our technique, and presents a new database-aware fault-localization method.

A. Database Interactions

SQL provides four main commands for interaction with the databases: `SELECT`, `UPDATE`, `INSERT`, and `DELETE`. For our purposes, we give a name to a statement in the application that invokes the database driver and executes these commands: database-interaction point. A *database-interaction point* is a location in the source code where control and data transfer from the application to the database system and back again.

One unique aspect of database applications is the querying and modification process. (We discuss the implications of these database-application design choices for fault-localization in Section IV-F.) There are two steps in this process: building the SQL command and executing the SQL command. For an application to *build an SQL command*, there are three main approaches: (1) applications can construct SQL commands within the code, such as by creating or modifying a string; (2) an external source, such as a configuration file or user input, can provide the SQL command to the application; or (3) part of the command can be built in the code and other parts of the command, such as attributes, can be received from an outside source. For an application to *execute an SQL command*, there are two approaches: (1) applications may use many database-interaction points to execute the SQL commands; or (2) applications use one (or few) centralized database interaction point. These SQL execution patterns are not mutually exclusive: an application may contain more of one type or use a more even mixture of both types.

B. Database Fault-localization Entities

Existing fault-localization techniques select a program entity, such as a statement, a branch, or a predicate, on which to perform the fault localization. The first challenge we encountered in developing our database-aware fault-localization technique was to identify an appropriate database entity that would be most useful for fault localization.

Our first approach was to localize on database attributes. Using existing statistical metrics, such as Tarantula [10], [11] and Ochiai [2], we calculated the suspiciousness for each attribute used in the SQL queries made by the application. Although ranking attributes by their suspiciousness proved to be quite accurate, the technique was not able to relate these attributes to the application. Thus, this method could not report sufficient information to the developers to locate the fault.

Our second approach aimed to address the limitations of the attribute-ranking approach by relating the attributes back to the application code. To accomplish this relationship between attributes and the application, we created statement-attribute tuples, which link each statement with the attributes that it

executes. We calculated the suspiciousness of the statement-attribute tuples using the same statistical metrics we used in the first approach. Again, the rankings were effective in identifying the faulty attribute as suspicious, but two new difficulties arose. First, depending on the application, many statement-attribute tuples received the same suspiciousness score. Second, when the SQL command was built dynamically or retrieved from a file, it was not obvious which SQL command had been executed. Without the SQL command providing the context in which the attributes were interacting, it was often difficult to know how to proceed with the faulty statement-attribute tuple.

To provide this context, our third approach constructs statement-SQL tuples. Knowing the SQL commands that execute at each statement and computing their suspiciousness scores was successful. However, we lost the information about which attribute in the command was most suspicious. Thus, to regain the precision lost in localizing on entire SQL commands rather than just faulty attributes, we created, and present in the next section, a technique to leverage the strengths of both the statement-SQL and the statement-attribute approaches.

C. Our Integrated Technique

Our database-aware fault-localization technique has two main goals for SQL faults: (1) to localize on the faulty statement-SQL tuple or statement-attribute tuple, and (2) to provide additional information about the SQL commands executed by the test suite. Figure 3 gives a dataflow diagram[3] that represents our database-aware fault-localization technique. The technique accepts as input an application A, a test suite T for A, and a database used by A. The technique outputs, to the developer, the SQL Execution Data and the Suspiciousness Rankings of statements, statement-SQL tuples, and statement-attribute tuples for multi-SQL statements.

Instrument Application instruments A to produce $\hat{A}$, an instrumented version of the application. $\hat{A}$ performs the same sequence of operations as A, but also records statement-coverage information, the SQL commands executed, and the statement that executed each SQL command. *Run Test Cases* executes $\hat{A}$ with test suite T, and records and outputs the Statement Coverage and the Executed SQL command information. *Identify Statement-SQL & Statement-Attribute Tuples* produces Statement-SQL Tuples and Multi-SQL Statement-Attribute Tuples. Statement-SQL tuples are defined as the set of $\langle s, c\rangle$ such that, for each test case $t \in T$, the execution produces S_t, which contains the coverage information of each statement s during execution of t, C_t, which is a set of SQL command data tuples $\langle s, c\rangle$ where c is an SQL command that was executed and s is the statement that initiated the SQL call, and p_t, which indicates whether t passed or failed. Multi-SQL Statement-Attribute Tuples are produced as the set SA_t for each test case $t \in T$ by parsing attributes from the SQL commands c from each statement-SQL tuple $\langle s, c\rangle$ only when statement s has executed multiple unique SQL commands.

[3]In the *dataflow diagram*, labeled edges represent data and rounded boxes represent the processing of the data.

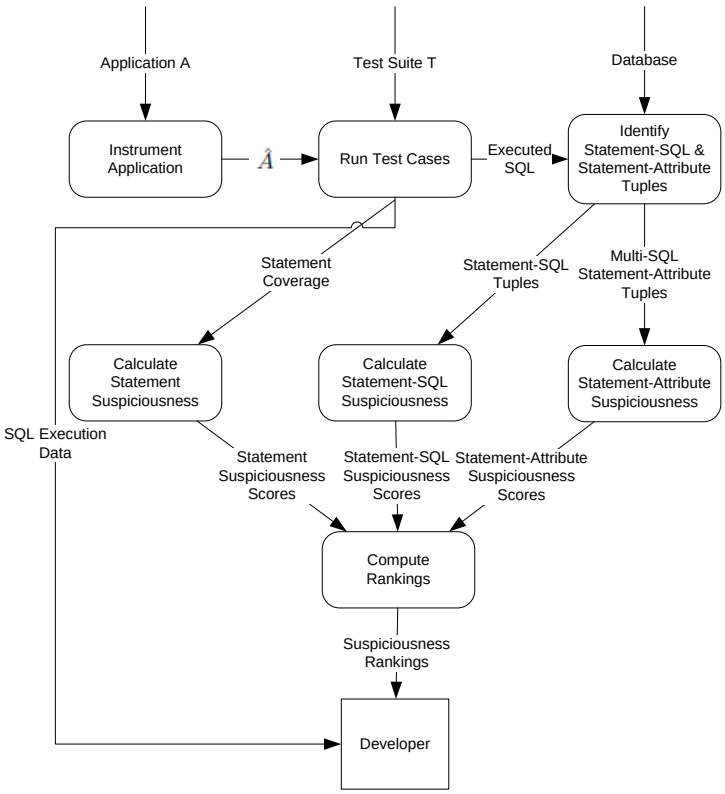

Fig. 3: Database-aware fault-localization technique.

In the example in Figure 2, statement 5 is a multi-SQL statement because two unique SQL commands are executed:
`SELECT PRODUCT, PRICE FROM SALE WHERE MERCHANTID = ?`
`SELECT PRODUCT, PRICE FROM SALE WHERE CUSTOMERID = ?`
These commands expand to the four statement-attribute tuples: $\langle$5, PRODUCT$\rangle$, $\langle$5, PRICE$\rangle$, $\langle$5, CUSTOMERID$\rangle$, and $\langle$5, MERCHANTID$\rangle$.

Calculate Statement Suspiciousness, Calculate Statement-SQL Suspiciousness, and *Calculate Statement-Attribute Suspiciousness* produce Statement Suspiciousness Scores, Statement-SQL Suspiciousness Scores, and Statement-Attribute Suspiciousness Scores, respectively, by employing M, the suspiciousness metric. M can be any of the existing coverage-based suspiciousness metrics, such as Tarantula [10], [11] or Ochiai [2]. In this case, an entity e may be a statement s, a statement-SQL tuple $\langle s, c\rangle$, or a statement-attribute tuple $\langle s, a\rangle$. Our technique records the number of passing and failing test cases that cover each entity, and applies M to obtain the suspiciousness score. To illustrate, consider the example in Figure 4. When suspiciousness is computed using the integrated database-aware fault-localization technique with the Ochiai metric, the suspiciousness of the faulty query at statement 5 (i.e., SELECT ... MERCHANTID >= ?) is 0.82. Figure **??** shows the result of applying Ochiai to the statement-attributes tuples, and Figure **??** reveals the outcome of using the result to inspect a suspicious SQL command.

Program Entities		Test Cases							
	`printProductsSold(String UserType, String userID) {`	`Merchant, 1`	`Merchant, 2`	`Merchant, 3`	`Customer, 1`	`Customer, 2`	`Customer, 3`	`Customer, 4`	**Suspiciousness**
1	`String Attributes = conf.getAttributes(userType, userID);`	●	●	●	●	●	●	●	0.53
2	`String whereClause = conf.getWhere(userType, userID);`	●	●	●	●	●	●	●	0.53
3	`String SQL="SELECT "+Attributes+" FROM Sale WHERE" + whereClause;`	●	●	●	●	●	●	●	0.53
4	`PreparedStatement ps = new PreparedStatement();`	●	●	●	●	●	●	●	0.53
5	`ResultSet rs = ps.executeQuery(SQL);`	●	●	●	●	●	●	●	0.53
	⟨5, SELECT PRODUCT, PRICE FROM SALE WHERE MERCHANTID>=?⟩	●	●	●					**0.82**
	⟨5, SELECT PRODUCT, PRICE FROM SALE WHERE CUSTOMERID=?⟩				●	●	●	●	0.00
6	`printResultSet(rs);`	●	●	●	●	●	●	●	0.53
	}								
	Pass/Fail Status	F	F	P	P	P	P	P	

Fig. 4: Example program, test cases, and suspiciousness scores from Figure 2 with SQL command tuples included.

Attribute	Pass	Fail	Suspiciousness
PRODUCT	5	2	0.53
PRICE	5	2	0.53
CUSTOMERID	4	0	0.00
MERCHANTID	1	2	0.82

(a) Attribute suspiciousness at statement 5.

Suspiciousness	SQL Subclause
	SELECT
0.53	PRODUCT,
0.53	PRICE
	FROM SALE WHERE
0.82	MERCHANTID>=?

(b) Suspicious SQL command attributes.

Fig. 5: Expansion of attributes from SQL commands at statement 5 in Figure 4.

In the final step, *Compute Rankings* creates a single list by combining Statement Suspiciousness Scores and Statement-SQL Suspiciousness Scores. This step then sorts the list by suspiciousness scores from greatest to least, and outputs the Suspiciousness Rankings. This ranked list of statements, statement-SQL tuples, and statement-attribute tuples, along with the executed SQL-command information, is then presented to the developer for use in locating the fault. In the example in Figure 4, after sorting, the statement-SQL tuple ⟨5, SELECT . . . MERCHANTID >= ?⟩ and the statement-attribute tuple ⟨5, MERCHANTID⟩, with suspiciousness scores of 0.82, appear as the top-ranked entities. This SQL command and attribute are, in fact, the sources of the fault in this example.

In our example in Figure 2, the existing technique was unable to localize the faulty statement. However, our database-aware technique was able to pinpoint the database-interaction point, the faulty SQL command, and the attribute involved in the faulty subclause of that command. Although the SQL command in this example is simple, for illustrative purposes, the additional information provided by our technique can be especially useful when the SQL commands are obfuscated or large and complex, as they often are in practice.

IV. EMPIRICAL STUDIES

To evaluate our technique, we implemented it and performed two empirical studies. This section discusses the implementation, describes the subjects, overviews the empirical setup, and presents the results of the studies.

A. Implementation

We implemented our technique in Java for use on Java database applications. We modified COBERTURA[4] to provide per-test-case coverage because the original implementation supports only aggregate test-suite coverage. We used P6SPY[4] to capture executed SQL commands. To enable association of SQL commands with database-interaction points in the program, and to identify statement-SQL tuples, we modified P6SPY to capture statements that dispatch SQL commands.

To identify statement-attribute tuples, we used a modified version of the UNITY parser [15] to create a tool that parses the output of our modified P6SPY. Our modifications to the UNITY parser let the tool parse additional constructs and syntactic variances that occur across SQL dialects.

To compute the suspiciousness of program entities, our tool collects statement coverage and statement-SQL coverage, and identifies all statements that execute multiple unique SQL commands to create the appropriate statement-attribute tuples. The tool then uses this coverage information, along with the results of the JUnit test cases, to compute the suspiciousness of every observed entity, using the Ochiai metric [2]. The tool sorts the program statements, statement-SQL, and statement-attribute tuples by suspiciousness to provide a ranking. Finally, we use scripts that we created to execute the mutated versions, process the data, and assess the effectiveness of the technique.

B. Subjects

For our studies, we used three programs: ITRUST, JWHOIS-SERVER, and MESSAGESWITCH. Table II shows the number of lines of Java code, the number of database tables referenced by the application, and the number of database interaction points in each application. The first subject, ITRUST,[5] which

[4]http://cobertura.sourceforge.net/, http://www.p6spy.com/

[5]http://agile.csc.ncsu.edu/iTrust, http://www.mysql.com

TABLE II: Subject details.

Subject	Java LOC	DB Tables	DBI Points
iTrust	25517	30	157
JWhoisServer	6684	10	2
MessageSwitch	3672	15	16

TABLE III: Mutants for each subject program.

Subject	Code Mutants	SQL Mutants
iTrust	100	125
JWhoisServer	50	40
MessageSwitch	25	10

was created as a class project at N. C. State University for teaching testing techniques, is a medical application that caters to patients and the medical staff. iTrust uses a MySQL[5] database and is predominately written in Java. iTrust's database queries are mostly static: the majority come from pre-written `String` literals and are not constructed at runtime. For our studies, we considered only the Java portion of the application. We ran 802 of the JUnit test cases that accompany the application, excluding those that tested the HTTP interface.

The second subject, JWhoisServer,[6] is an open source WHOIS server implemented in Java that includes configuration files, Velocity templates,[7] and permits a variety of database dialects to be used. In our studies, we configured JWhoisServer to use HSQLDB.[6] Unlike iTrust, JWhoisServer's database commands are constructed dynamically at runtime from data stored in the configuration files. We used the 79 JUnit test cases provided by the application.

The third subject, MessageSwitch, is a proprietary application used by a transaction-processing company. The application is implemented in Java and uses an Oracle[8] database. MessageSwitch also includes configuration files and Spring templates.[7] MessageSwitch provides its database commands in configuration files, completely external to the Java code. To evaluate our technique, we ran the 80 JUnit test cases used by the company to test the application.

For each subject, we created single-fault mutants, and used them as faulty versions of the subject. We created two types of mutants: (1) *Code mutants* contain code faults in the application; and (2) *SQL mutants* contain SQL faults in the application. Table III shows the number of mutants for each subject by mutant type. We manually created the mutants because existing mutation tools are unable to process our subject programs. We constructed each mutant by applying one of the mutation operators [16], [18] to the original program. Examples of such mutations can be found in Section IV-E.

C. Empirical Setup

For each mutant of the program, we

1) Identified the faulty entities for each technique,
2) Instrumented the program to gather SQL-command and statement coverage,
3) Ran the test suite and recorded the coverage,
4) Identified the multi-SQL database interaction points,
5) Parsed the SQL commands to obtain statement-attribute tuples for the multi-SQL statements,
6) Calculated the suspiciousness of all monitored entities,
7) Ranked the entities according to decreasing suspiciousness scores, and
8) Assessed the effectiveness of the approach.

To facilitate the evaluation, we performed the identification of the faulty entities in Step 1 differently depending on the type of mutation in the program: for code mutants, we identified the faults by program statements, and for SQL mutants, we identified the faults by statement-SQL tuples.

D. Study 1: Quantitative Evaluation

The goal of this study is to compare the effectiveness of statement-based fault-localization techniques that do not provide special treatment of database interactions with our database-aware fault-localization technique. To evaluate the effectiveness of a proposed technique, previous work in fault localization (e.g., [10], [14]) uses a ranking system as well as the percentage of code a developer would not need to inspect by utilizing the ranked results. We emulated this evaluation approach for this study. We used the Ochiai metric to compute suspiciousness values and rankings for each entity. We defined the faulty entities for statement-based fault-localization: for code mutants, we used the statement containing the mutated code; for SQL mutants, we used either the mutated statement, when the mutation occurred within the application code (e.g. a `String` literal containing SQL), or the database-interaction point where the faulty SQL command was executed, when the mutation was external to the source code. We defined the faulty entities for database-aware fault-localization: for code mutants, we again used the mutated statement; for SQL mutants, we used the faulty statement-SQL tuple or statement-attribute tuple consisting of the database-interaction point and the faulty SQL command or attribute, respectively.

Table IV shows our results—the percentage of faults that can be found without examining 99% and 90% of the code. The data is separated by subject and then by fault type. "SQL" refers to SQL faults, "Code" refers to code faults, and "All" refers to the combination of both the SQL and the code faults. The data is also separated by the technique used to calculate the rankings: Stmt 99% refers to the statement-based technique and DB 99% refers to the database-aware technique.

We found that for iTrust, the statement-based approach did well at localizing both SQL and code faults. Specifically, 97.6% of the code faults, 94.1% of SQL faults, and 96.6% of all faults were found within the top 1% of the statement rankings. Statement-based fault-localization also performed well on MessageSwitch for both types of faults: 26.3% of the code faults, 50.0% of the SQL faults, and 32.0% of all faults were found within the top 1% of the statement rankings. However, the approach did not do as well for the SQL faults in JWhoisServer, only 17.4% of the code faults, 0.0% of the SQL faults, and 6.7% of all faults were found within the top 1% of the statement rankings.

[6]http://jwhoisserver.net/, http://hsqldb.org/
[7]http://velocity.apache.org/, http://www.springsource.org/
[8]http://www.oracle.com/products/database

TABLE IV: Study 1 Results.

Subject	Fault Type	Stmt 99%	DB 99%	Stmt 90%	DB 90%
iTrust	SQL	94.1%	94.1%	100%	100%
	Code	97.6%	97.6%	97.6%	100%
	All	96.6%	96.6%	98.3%	100%
JWhoisServer	SQL	0.0%	94.6%	86.5%	100%
	Code	17.4%	13.0%	60.7%	60.9%
	All	6.7%	63.3%	76.7%	85.0%
MessageSwitch	SQL	50.0%	66.7%	100%	100%
	Code	26.3%	26.3%	68.4%	68.4%
	All	32.0%	36.0%	76.0%	76.0%

By comparing the results of the statement-based approach with the results using our database-aware technique, we can see that our database-aware technique does considerably better for JWhoisServer, especially for the SQL faults. The iTrust results for both the statement-based and the database-aware techniques are very similar. Thus, there was little room for improvement, and our technique succeeded in not degrading the effectiveness. For iTrust, 97.6% of the code faults, 94.1% of the SQL faults, and 96.6% of all faults were found within the top 1% of the entity rankings. For JWhoisServer, 13.0% of the code faults, 94.6% of the SQL faults, and 63.3% of all faults were found within the top 1% of the entity rankings. For MessageSwitch, 26.3% of the code faults, 66.7% of the SQL faults, and 76.0% of all faults were found within the top 1% of the entity rankings.

E. Study 2: Qualitative Case Study

The goal of this study is to evaluate the additional benefits of our technique that are difficult to quantify numerically. We realize that not all developers will debug by stepping through the application one entity at a time in ranked order. Developers may use the rankings to identify sections of code that seem suspicious, and then investigate further from that point. Nevertheless, after a section of code that includes a database interaction point has been identified as suspicious, our database-aware technique provides additional information that can be useful to the developer. For this case study, we assume the developer has located a suspicious section of code. Each of our subjects builds its SQL commands in a slightly different manner so we selected one mutant of each subject to examine. For each mutant, we provide a code sample to illustrate the suspicious section of the application, a description of the mutation, and a discussion of the additional information provided to the developer by our technique.

1) iTrust*:* The majority of SQL commands in iTrust are static strings embedded in the source code. iTrust typically prepares and executes these commands at or close to the location of the string. For example:

```
ps = conn.prepareStatement(
  "UPDATE GlobalVariables SET Value=? " +
  "WHERE Value='Timeout'");
ps.setInt(1, mins);
int numUpdated = ps.executeUpdate();
```

In this example, the mutation is an attribute replacement; the clause `Value='Timeout'` was originally `Name='Timeout'`, where `Value` and `Name` are both attributes of the `GlobalVariables` table. Because the SQL command is in a `String` literal just before the database-interaction point—the `ps.executeUpdate` call—the developer would be able to identify the SQL command quickly when the database-interaction point is suspicious. Our technique provides some additional assistance by revealing the literal values used in executions of the SQL command:

```
UPDATE GlobalVariables SET Value=21 WHERE Value='Timeout'
UPDATE GlobalVariables SET Value=5 WHERE Value='Timeout'
```

These values can provide clues beyond those given by the structure of the SQL command itself. In this particular example, `Value` is being assigned integer values but is compared with a textual string.

2) JWhoisServer*:* JWhoisServer builds SQL commands dynamically from command fragments contained in non-standard configuration files and `String` literals. All SQL commands in the application are executed by one of two database-interaction points. The majority of the SQL commands are executed by the following method:[9]

```
private final synchronized
ResultSet execPST(PreparedStatement pst)
    throws SQLException {
  ResultSet res = pst.executeQuery();
  return res;
}
```

The `execPST` method takes the SQL command to execute as the parameter `pst`. It is apparent that the suspiciousness of this database-interaction point alone does not easily lead the developer to the faulty SQL command as it would in iTrust. Furthermore, the SQL commands are constructed in a more dynamic manner as illustrated in this method:

```
protected final String getWherePart() {
  Vector<String> qv = this.getQfield();
  final String qf = this.getQfield().get(0);
  StringBuilder ret = new StringBuilder(
    "WHERE "+qf+" <= ? "
    +"AND inetnumend >= ? "
    +"AND "+this.bytelengthField+" = ? ");
  if (this.getWhereaddition().length() > 0) {
    if(!this.getWhereaddition().startsWith(" ")) {
      ret.append(" ");
    }
    ret.append(this.getWhereaddition());
  }
  ret.append("ORDER BY "+qf+" ASC, inetnumend ASC");
  return ret.toString();
}
```

In the method `getWherePart`, pieces of the `WHERE` clause, including use of the `inetnumend` attribute, are contained in `String` literals, and the rest are input through method calls and member fields. The calls to `getQfield` and `getWhereaddition` and the access of `bytelengthField` all reference data from an external configuration file. A sample from one such file is:

[9]Logging code has been omitted from all JWhoisServer examples.

```
db.inetnum.table=inetnum
db.inetnum.objectlookup=inetnum;inet
db.inetnum.qfield=inetnumstart
db.inetnum.key=descr
db.inetnum.bytelength=bytelength
db.inetnum.display=netname AS network;
    bytelength;inetnumstart;inetnumend;descr;source
db.inetnum.recurse.person=admin_c;tech_c
```

Many SQL mutations for JWHOISSERVER involve changes to configuration files. In this example, the value of `db.inetnum.key` changed from `netname` to `descr`. For this application, identifying the database-interaction point as suspicious is not particularly useful to the developer: it takes many steps to work back to where the `PreparedStatement` was created and even then it is not immediately apparent which tables, attributes, and values were used in the SQL command. Our technique provides additional helpful information: it supplies the full SQL commands executed at the database-interaction point, ties each command to the test case that executed the command, and indicates the correlation of each command with the failure of test cases in the test suite. The database-interaction point alone is not ranked highly because it is executed by both passing and failing test cases. In contrast, our technique identifies the database-interaction point and mutated SQL command with high suspiciousness:

```
dbpool.java:631
select descr, netname as network, bytelength,
  inetnumstart, inetnumend, source from inetnum
  where inetnumstart <= ? and inetnumend >= ?
  and bytelength = ?
  order by inetnumstart asc, inetnumend asc
Suspiciousness: 0.9128709291752769
```

For this application, our technique aides the developer both through a more precise ranking and by directly presenting the command information. Even if existing techniques identified the database-interaction point in JWHOISSERVER as highly suspicious, which they do not, the developer would be left with more manual effort in locating the relevant command.

3) MESSAGESWITCH*:* MESSAGESWITCH utilizes static SQL commands, which it retrieves from external configuration files. MESSAGESWITCH is similar to ITRUST in that its SQL commands are not constructed dynamically, and similar to JWHOISSERVER in that the commands come from external files rather than being defined directly in the source code. MESSAGESWITCH is a proprietary application, so we are unable to show the actual contents of the source or configuration files. Instead, the following examples emulate the structure, database-access strategies, and SQL command patterns of the application. Database interaction in MESSAGESWITCH tends to follow the pattern shown below:

```
public CourseGrade selectStudentGradeByCourse(
    long studentId, String course) {
  CourseGrade grade = null;
  grade = _template.queryForObject(
    getSqlByKey(SQL_SELECT_STUDENT_GRADE_BY_COURSE),
            _studentMapper, new Object[]
            { studentId, course, course });
  return grade;
}
```

The static SQL command to execute is identified by the key `SQL_SELECT_STUDENT_GRADE_BY_COURSE`. In this case, the developer must first locate the definition of the key to find the value, which corresponds to a configuration file entry containing the SQL command. The key references the following SQL command:

```
<sql key="selectStudentGradeByCourse">
SELECT  student_id FROM (
  SELECT sg.student_id
  FROM ${config.schema}.v_grade_professor_xref gpx
  INNER JOIN ${config.schema}.grade sg
    ON sg.student_id = gpx.student_id
<!-- 25 lines omitted -->
  WHERE p.short_name = ? AND u.course = ?
  AND u.is_active = 1 AND gt.short_name = 'letter'
  )
</sql>
```

An SQL mutation might change the comparison of `gt.short_name` from `'letter'` to `'audit'`. Given only the database-interaction point, the developer would need to locate both the key and the external configuration file that defines the command. Even though a developer familiar with the application would know to perform these lookups when a database-interaction point is implicated in test-case failure, our technique reduces this effort by supplying the executed SQL command directly to the developer.

F. Discussion

The results of the first study demonstrate that, in some cases, a statistical, statement-based (i.e., non-database-aware) approach to fault-localization is effective at locating database-related faults. However, there are also cases in which the statement-based approach is insufficient and fault-localization is considerably more effective with a database-aware approach. The statement-based technique on JWHOISSERVER was able to correctly place *none* of the SQL faults within the top 1% of the code examined, whereas the database-aware technique was able to find 94.6% of those SQL faults. This increase in effectiveness in the localization of the fault is likely to translate to savings in the time a developer requires to find and identify the fault responsible for software failures. In the case of ITRUST, the statement-based technique was already able to correctly place 94% of the SQL faults in the top 1% of the code examined, and this number does not change when using the database-aware extensions.

We investigated why the statement-based approach was effective for ITRUST and MESSAGESWITCH but not for JWHOISSERVER. We found that it performed well for ITRUST and MESSAGESWITCH because their database-interaction points showed little dynamic behavior. That is, the majority of database-interaction points in ITRUST executed only one distinct SQL command—the structure of the command remained fixed and only the literal values changed. In this case, the suspiciousness score assigned to a statement-SQL tuple will necessarily be the same as the score of the statement itself. If we were to rank statement-attribute tuples, they would also be tied with the statement itself. However, a clear benefit of our approach is that in this case, it automatically avoids the cost of evaluating statement-attribute tuples.

In contrast, JWHOISSERVER is dynamic in that it contains only two database-interaction points through which all SQL commands are executed. Because the statement-based approach makes no distinction about which SQL command is being executed, it is ineffective at localizing to the database-interaction point. Even if the database-interaction point were identified, the developer would be left to manually determine which SQL commands were executed at that point and which among those commands was faulty. However, our approach effectively identifies the database-interaction point and the suspicious SQL commands, and gives additional information about the attributes used in the faulty command.

Although for the MESSAGESWITCH application, the technique achieves results similar to ITRUST in terms of ranking, the second study demonstrates that there is a qualitative difference between the statement-based approach and our database-aware approach. Unlike ITRUST, MESSAGESWITCH retrieves its SQL commands from external files rather than embedding them in `String` literals close to the database-interaction point. Similarly, the executed SQL command information provided by our technique is extremely useful for JWHOISSERVER because the SQL commands are created dynamically and from an external source.

G. Threats to Validity

Threats to external validity arise when the results of the study are unable to be generalized to other situations. In this study, we evaluated our new technique using only three applications and mutation faults, and thus, we are unable to definitively state that our findings will hold for programs in general. We addressed these limitations by using three different types of applications, and by creating many mutants of the applications that contain many different types of faults.

Threats to internal validity occur when factors affect the dependent variables without the researchers' knowledge. Although it is possible that implementation flaws affected the results, we have reasons for confidence in their correctness: we built the implementation on a mature infrastructure that has been used in previous studies (e.g., [10], [11], [17]), and the results from the statement-based approach matches those from the database-aware approach, which demonstrates consistency.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. The metric we used to evaluate fault-localization effectiveness assumes the developer will inspect the program, entity by entity, in the prescribed order until reaching the fault, and she will be able to recognize that it is faulty. Although this may not be a realistic debugging process, the metric has been used in many fault-localization studies (e.g., [10], [14]) as a reasonable approximation of the relative effectiveness of the fault-localization technique. Moreover, we also performed a more qualitative evaluation of the benefits of debugging with and without the information provided by our new database-aware fault-localization technique.

V. RELATED WORK

Although there has been no previous research on database-aware fault-localization, there has been research in the related areas of automated fault localization and testing and analysis of database applications. This section reviews this research and relates it to our presented technique.

The Tarantula [10] approach applies a statistical metric to program entities (e.g., statements and predicates) based on coverage of those entities by passing and failing test cases. The output of this metric is a suspiciousness score for each entity and is interpreted as the likelihood that the entity is responsible for the failure (e.g., [2], [10], [11], [13]). These techniques have been referred to as spectra-based, coverage-based, or statistical fault-localization techniques. Our database-aware fault-localization technique extends these approaches to incorporate entities related to the database that do not necessarily appear directly in the program, and considers multiple-entity types at once. Thus, it is able to more accurately assign suspiciousness to entities related to the database than the previous techniques.

Several existing fault-localization approaches select a subset of the program that may be responsible for the failure. Delta Debugging [20] attempts to isolate the cause of a failure resulting from program changes by repeatedly running the program with a subset of the changes applied. Dynamic slicing [3] uses an execution trace to select program entities that influence the value of a variable at a particular point in the program's execution. Our approach is more lightweight than Delta Debugging because it requires only one execution of the test suite, and it does not require knowledge of the program point from which the dynamic slice should be initiated. Moreover, our approach is expressly targeted toward database applications, which are commonly used in practice.[2]

Although there is no existing work on fault localization for database applications, the literature contains a wide variety of testing and analysis techniques designed for database environments; Reference [12] provides an extensive review. Our technique is complementary to these approaches—just as strong testing techniques and standards improve fault-localization, other database-aware testing techniques can help to strengthen fault localization for database applications.

SQL mutation testing [18], [19] applies a set of syntactic mutation operators to existing SQL queries to produce slightly modified versions. The resulting modifications are commonly used to assess adequacy of a test-suite. Other work extends the set of mutation operators to incorporate additional contextual information, such as use in Java programs [21] or by building a conceptual data model [5]. Instead of focusing on SQL mutation more generally, our empirical studies use a subset of the mutation operators to seed SQL faults into the applications. Because the SQL mutation operators in References [18], [19] consider only the `SELECT` command, we extended these operators to also handle `UPDATE`, `INSERT`, and `DELETE`.

Gould and colleagues [8] propose a method for statically type-checking SQL queries generated by a Java program. They perform static analysis to conservatively approximate

the queries a program may generate and the types of the attributes and expressions in these queries. The results are then checked for mismatched types. In contrast, our method uses a dynamic analysis that operates on the SQL commands that are executed by a test suite and leverages the results from each test case. Even though our approach does not explicitly consider type information, it may still reveal faults resulting from runtime type errors.

VI. Conclusions and Future Work

This paper presents the first database-aware fault-localization technique—a statistical fault-localization method that accounts for an application's interactions with its database. Our new approach was motivated by a study we performed on three database applications in which we found that existing techniques provide little help to the developer for locating faults within the application-database interactions.

Our integrated technique uses SQL commands executed by the test suite to compute sets of statement-SQL and statement-attribute tuples. The technique computes suspiciousness scores for the statement-SQL tuples, statement-attribute tuples, and statements, then presents the developer with a ranked list and a set of executed SQL commands to assist during debugging.

The paper also describes two empirical studies on three applications that evaluated the effectiveness of our new technique compared to existing approaches. The first study shows that, for our subject programs, our new technique was able to improve the effectiveness of finding SQL faults as much as 94.6% over the existing technique. The second study demonstrates some of the qualitative benefits, such as a potentially more efficient debugging process, provided by our technique.

The results are encouraging and demonstrate the improvement that can be achieved with our new database-aware technique. There are, however, many areas of future work that can be explored. We performed our studies on three projects. To fully evaluate our technique, and guide additional research, we must identify other suitable subjects for our research.

We also investigated attribute-specific faults and localized on statement-attribute tuples. We plan to expand the class of faults we address to include multiple-attribute faults and faults within other parts of the SQL command, such as the comparative operator. Additionally, we are working on techniques that can localize data and schema faults, which are not captured by analyzing SQL commands.

Finally, many database applications use stored procedures to execute their SQL code rather than building or loading the SQL within their code. Stored procedures can contain their own parameters, logic, and stored procedure calls. Providing a fault-localization technique to help developers localize faults within their stored procedures could be a valuable contribution. We are investigating how to instrument stored procedures to obtain coverage information, and use this information to enhance our fault-localization method. Ultimately, combining our new database-aware technique with both these additional features, and existing testing and analysis methods with database awareness, will yield a comprehensive framework for testing and debugging database applications.

Acknowledgements

This research was supported in part by NSF award CCF-1116943 and a Google Faculty Research Award to UC Irvine, by NSF awards CCF-0725202 and CCF-0541048, an IBM Software Quality Innovation Award, and a grant from InComm to Georgia Tech. The anonymous reviewers provided many helpful suggestions that improved the paper's presentation.

References

[1] InfoWorld. http://xml.coverpages.org/xmlPapers200309.html.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proc. of the 2nd Testing: Academic and Industrial Conference, Practice and Research Techniques*, 2007.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. of the 8th Conference on Programming Language Design and Implementation*, 1990.

[4] P. Brooks, B. Robinson, and A. M. Memon. An initial characterization of industrial graphical user interface systems. In *Proc. of the 2nd International Conference on Software Testing, Verification and Validation*, 2009.

[5] W. K. Chan, S.C. Cheung, and T.H. Tse. Fault-based testing of database application programs with conceptual data model. In *Proc. of the 5th International Conference on Quality Software*, 2005.

[6] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *Proc. of the 19th European Conference on Object-Oriented Programming*.

[7] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, fifth edition, 2007.

[8] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. of the 26th International Conference on Software Engineering*, 2004.

[9] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proc. of the 22nd International Conference on Automated Software Engineering*, 2007.

[10] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proc. of the 20th International Conference on Automated Software Engineering*, 2005.

[11] J. A. Jones, J. Stasko, and M. J. Harrold. Visualization of test information to assist fault localization. In *Proc. of the 24th International Conference on Software Engineering*, 2002.

[12] G. M. Kapfhammer. *A comprehensive framework for testing database-centric software applications*. PhD thesis, University of Pittsburgh, 2007.

[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. of the 26th Conference on Programming Language Design and Implementation*, 2005.

[14] C. Liu, X. Yan, L. F., J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proc. of 10th European Software Engineering Conference and 13th Symposium on the Foundations on Software Engineering*, 2005.

[15] T. Mason and R. Lawrence. Dynamic database integration in a JDBC driver. In *Proc. of the 7th International Conference on Enterprise Information Systems*, 2005.

[16] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *Transactions on Software Engineering and Methodology*, 5(2), 1996.

[17] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proc. of the 31st International Conference on Software Engineering*, 2009.

[18] J. Tuya, M. J. Surez-Cabal, and C. de la Riva. Mutating database queries. *Information and Software Technology*, 49(4), 2007.

[19] M.J. Tuya, J. Suarez-Cabal and C. de la Riva. SQLMutation: A tool to generate mutants of SQL database queries. In *Proc. of the 2nd Workshop on Mutation Analysis*, 2006.

[20] A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. of the 10th Symposium on Foundations of Software Engineering*, 2002.

[21] C. Zhou and P. Frankl. Mutation testing for Java database applications. In *Proc. of the 2nd International Conference on Software Testing Verification and Validation*, 2009.