The Journal of Systems and Software 147 (2019) 230–245

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Controversy Corner

# Exoneration-based fault localization for SQL predicates

Yun Guo[a], Nan Li[b,*], Jeff Offutt[a], Amihai Motro[a]

[a] Department of Computer Science, George Mason University, 4400 University Dr, Fairfax, Virginia 22030, United States
[b] Research and Development, Medidata Solutions, 350 Hudson Street, 9th Floor, New York 10014, United States

ARTICLE INFO

*Article history:*
Received 13 February 2018
Revised 24 September 2018
Accepted 25 October 2018
Available online 26 October 2018

*Keywords:*
Fault localization
Spectrum-based fault localization
Exoneration-based fault localization
SQL
WHERE clause

ABSTRACT

Spectrum-based fault localization (SFL) techniques automatically localize faults in program entities (statements, predicates, SQL clauses, etc.) by analyzing information collected from test executions. One application of SFL techniques is to find faulty SQL statements in database applications. However, prior techniques treated each SQL statement as one program entity, thus they could not find faulty elements inside SQL statements. Since SQL statements can be complex, identifying the faulty element within a faulty SQL statement is still time-consuming.

In our previous paper, we developed a novel fault localization method based on row-based dynamic slicing and delta debugging techniques that can localize faults in individual clauses within SQL predicates. We call this technique *exoneration-based* fault localization because it can exonerate "innocent" elements and precisely identify the faulty element, whereas previous SFL techniques simply ranked all the elements in an SQL statement based on suspiciousness.

This paper improves the exoneration-based fault localization technique with a new algorithm that considerably reduces the execution time. We also conducted an empirical study that compared nine existing SFL techniques with the exoneration-based technique in localizing faulty clauses in SQL predicates. Results indicate that the new exoneration-based technique surpasses the other techniques both in terms of effectiveness and efficiency.



## 1. Introduction

A *failure* is an unexpected behavior on the part of software and indicates a *fault* in the program (Ammann and Offutt, 2017). By themselves, failures do not provide enough information to locate the root-cause fault. In an attempt to locate faults, spectrum-based automatic fault localization (SFL) techniques trace program execution during testing and analyze program entities. The *program entity* being analyzed for possible faults is a piece of program that can be at various levels of granularity. Researchers have targeted statements (Jones and Harrold, 2005; Abreu et al., 2007), predicates (Liblit et al., 2005; Liu et al., 2006), and clauses (Guo et al., 2017) as program entities. Clearly, techniques that address fine-grained program entities are more useful–locating a faulty statement helps more than locating a faulty block or procedure.

Although many papers have been published on automatically localizing faults in general programs, only a few attempts have been made to find faults in SQL queries. SQL (Structured Query Language) is a declarative language used to access and manipulate data in relational databases. It is widely used in database applications and was ranked as the most in-demand programming language in 2016 (Bouwkamp, 2016). Unlike general programming languages such as JAVA or C#, SQL queries declare what information the answer should contain, but do not specify how to compute it. As a result, debugging SQL can be quite different from debugging general programming languages.

Previous studies have attempted to apply SFL techniques to database applications by treating an entire SQL query or an SQL structure (often of considerable size) as a program entity. In particular, Nguyen et al. (2013) located faults in WHERE *predicates*. A WHERE predicate is a boolean expression that incorporates *clauses*, each comparing the value of a cell to a constant or to another cell. While these methods may suggest that an entire predicate is faulty, they cannot locate the individual clause (or clauses) at fault.

To find faulty clauses in WHERE predicates, we choose clauses as program entities. In our previous paper (Guo et al., 2017), we targeted six *fault classes*[1], which define the categories of faults

* Corresponding author.
*E-mail addresses:* yguo7@gmu.edu (Y. Guo), nli@mdsol.com (N. Li), offutt@gmu.edu (J. Offutt), ami@gmu.edu (A. Motro).

https://doi.org/10.1016/j.jss.2018.10.037
0164-1212/

[1] Monperrus (2014) uses the term "defect class." We use "fault class" to be consistent with the testing literature.

that can be localized by SFL techniques. The concrete fault classes are defined in Section 2.1. We developed an exoneration-based SFL technique to isolate individual faulty clauses within WHERE predicates. This technique first uses row-based dynamic slicing to discover suspicious clauses in WHERE predicates and then removes "innocent" (not faulty) clauses using a technique inspired by delta debugging (Zeller and Hildebrandt, 2002). Thus, our technique can precisely identify which clause is faulty as well as the type of the fault. In contrast, the existing SFL techniques calculate suspiciousness scores for every program entity to indicate its likelihood of being faulty, and then rank them by their suspiciousness scores. The rankings provide guidelines for locating the fault. However, developers still must analyze the ranked list to locate the fault.

Although our original approach was effective, it was not efficient enough to be practically useful. The execution time increased exponentially with the number of columns in the test database, the number of test rows, and the complexity of the faulty query (as approximated by the number of clauses). This paper proposes and evaluates a novel algorithm that is much more efficient. The general idea is to identify and remove equivalent test rows from the exoneration process. The failing test rows are considered to be equivalent if they are caused by the same faulty clauses. Since large databases can contain millions of rows, removing redundant rows from the tests can increase speed. We compared the original exoneration-based technique to the newer version with the efficiency algorithm. The results showed that the newer version is significantly faster than the original version, and as effective.

We conducted a comprehensive empirical study that compared our novel exoneration-based technique with nine existing SFL techniques. Five of the existing techniques are similarity-based (Jones and Harrold, 2005; Abreu et al., 2007; Naish et al., 2011) and the other four are statistics-based (Liblit et al., 2005; Wong et al., 2012; Liu et al., 2005; Zhang et al., 2011). These two categories use coefficient formulas or statistical methods to calculate suspiciousness scores for every program entity. Because they were originally used on program entities such as statements or predicates rather than SQL clauses, we modified them to apply to SQL clauses. Specifically, we analyzed how to compute the suspiciousness score formulas. Our experiments show that the exoneration-based technique outperforms the other approaches in both effectiveness (accuracy of localizing faulty clauses) and efficiency (execution time).

This paper builds on previous work (Guo et al., 2017) that presented an exoneration-based fault localization technique that can effectively find faulty clauses in SQL predicates, and a categorization of fault classes in SQL WHERE conditions. This paper greatly extends the previous conference paper with four significant contributions:

1. A new algorithm that significantly improves the performance of the exoneration-based technique.
2. An experimental comparison of the original and newer exoneration-based fault localization techniques.
3. Modifications to nine existing SFL techniques to apply them to SQL clauses.
4. An experimental comparison of our new exoneration-based fault localization technique with nine other SFL techniques. Some results were consistent with previous results when applied to other program entities such as statements, while other results were different when applied to clauses.

The empirical comparison in this paper is also much larger than in our previous paper (Guo et al., 2017), both in terms of the number of techniques compared and in the size of the subjects. The previous paper compared our old exoneration-based technique with only two similarity-based techniques, whereas this paper uses a new algorithm, and compares it with five similarity-based and four statistics-based techniques. To the best of our knowledge, this empirical study is the first to compare similarity-based, statistics-based, and exoneration-based techniques for localizing faulty SQL clauses. In addition, these experiments are conducted on much larger subjects than previous studies. We used 450 subject queries from five databases including real faults from an industry application, compared with only 180 queries from two databases in the previous study.

The rest of the paper is organized as follows. Section 2 introduces the nine existing SFL techniques and describes how we modified them to apply to SQL clauses. Section 3 reviews our original exoneration-based technique. Section 4 explains the new algorithm that improves the efficiency of the exoneration process. Section 5 presents the empirical study, including the methodology, results, and an analysis. Related work is discussed in Section 6, and Section 7 summarizes the contributions and suggests some future work.

## 2. Spectrum-based fault localization

This section first defines some fundamental concepts used in spectrum-based fault localization in Section 2.1. It then introduces similarity-based and statistics-based techniques, along with a description of how they are applied to clauses in Sections 2.2 and 2.3.

### 2.1. Definitions and an SQL example

Program entities that are covered during test execution are also called *program spectra* (Souza et al., 2016). In general, spectrum-based fault localization techniques collect information during test executions and use that information to identify program entities that appear to be suspicious. When the program entities are statements, the information collected is whether the statements are executed. When the program entities are predicates or clauses, the information collected is whether they evaluated to true or false during execution. In this paper, the program entities to be examined are clauses in SQL predicates. Test cases are rows in a database, thus we use the terms *tests* and *rows* interchangeably in this paper.

An SQL query consists of SELECT, FROM, and WHERE clauses. A SELECT clause isolates the columns to be retrieved. A FROM clause specifies the tables to retrieve the data from. A WHERE clause defines conditions the rows retrieved must satisfy, with a Boolean predicate. That is, a WHERE clause is a predicate prefixed with the WHERE keyword. All WHERE predicates can be transformed to *disjunctive normal form (DNF)*, which is a disjunction of conjunctions of clauses[2] That is, a smaller predicate, a *conjunctive predicate* (CP), connects clauses with AND operators. CPs are connected with OR operators. Each clause is of the form *column opr constant* or *column opr column*, where *opr* is a comparator. This paper focuses on finding faulty clauses in WHERE predicates, because WHERE predicates are more complicated than other elements of SQL queries and therefore are more prone to faults.

To identify which WHERE predicate is faulty, we need at least one failing test case that has an oracle. Test data are sets of database rows $R$, where each row is a test case, denoted $r_i$. Test oracles are based on the original requirements, written by the testers, and embedded in automated tests, usually as assertions, to determine if test cases are passing or failing. Test oracles can be more or less general. Let's call a *general test oracle* to be one that can determine whether any output is correct, and a *specific test oracle* to be one that can only determine correctness for an

[2] For the purpose of this research, we believe CNF (conjunctive normal form) and DNF (disjunctive normal form) are equivalent. We choose DNF because it is more intuitive to us, as it resembles a circuit with switches.

individual test case. General test oracles are more difficult to create than specific test oracles. But when there are hundreds or thousands of test cases, devising one general test oracle can be more cost-effective than writing a specific test oracle for each individual test case. In database query testing, each row is considered as a test case, so it is common to have thousands and even millions of test cases. Creating specific test oracle for so many test cases is unrealistic, so general test oracles are widely used in industry. Thus, this research uses general test oracles.

In this research, the SQL query under test is applied to the test cases (database rows $R$), which results in some rows being returned (included) and others being excluded. The test oracles determine whether a test case (a database row $r_i$) is included or excluded as expected. For example, if the test case passed and the row was not returned, that means the row was expected to be excluded, and was excluded. Or, if the test case failed and the row was returned, that means it was expected to be excluded, but was included. The test oracles categorize the test data into four groups:

- $\mathbf{R_i}$: Rows expected to be included that are included (*included*).
- $\mathbf{R_e}$: Rows expected to be excluded that are excluded (*excluded*).
- $\mathbf{R_s}$: Rows expected to be excluded but are included (*superfluous*).
- $\mathbf{R_a}$: Rows expected to be included but are excluded (*absent*).

Rows $\mathbf{R_i}$ and $\mathbf{R_e}$ are *passing rows* and rows $\mathbf{R_s}$ and $\mathbf{R_a}$ are *failing rows*. If either of the last two groups is not empty, then the SQL is incorrect. Our previous paper (Guo et al., 2017) defined six fault classes for the WHERE clause:

- **E1**: Incorrect constant (e.g., a string was misspelled or a decimal point was misplaced).
- **E2**: Incorrect operator (e.g., $>$ was used instead of $\geq$).
- **E3**: Incorrect column (a different column should have been used).
- **E4**: Missing clauses (that should be present).
- **E5**: Superfluous clauses (that should be removed).
- **E6**: Composite faults with more than one single type.

E1 through E5 are single faults that satisfy complete and disjoint properties. They cover all possible single faults in clauses (thus are *complete*). Clauses have three components: constants, operators, and columns. E1 through E3 represent faults in each of the three components. E4 and E5 are faults where the entire clause is missing or unnecessary. These five fault types also apply to different elements, so do not overlap (thus are *disjoint*). E6 (*composite*) is added to allow for multiple faults in the same query. It can also be used to explain faults that involve incorrect AND or OR operators. For example, if $a$ AND $b$ is incorrectly written as $a$ OR $b$, that can be interpreted as an unnecessary clause $b$ that should be removed (E5) and a missing clause $b$ that should be added with an OR (E4).

Two recent surveys summarize fault localization techniques and their variants (Souza et al., 2016; Wong et al., 2016). Because of the large number of techniques, we could not compare all techniques surveyed. Thus, we focus on recent techniques that appear to have had significant impact. All the similarity-based and statistics-based techniques are similar in how they work. First, a *suspiciousness score* is calculated for each program entity. A higher suspiciousness score indicates that the program entity is more likely to be faulty. The techniques then rank the program entities by their suspiciousness scores and return the ranked result. The SFL techniques are discussed in the rest of this section.

We introduce a small running example to illustrate the SFL techniques. Table 1 displays orders for a product where *Orderid* is the primary key. Consider a request to find orders placed after **year 2009** with price greater than \$100, plus orders shipped to **zip code 10007** with no discount. Now assume that the programmer made a mistake (E1 faults), retrieving instead orders placed after **year 2007** (with price greater than \$100) plus orders shipped to **zip code 10008** (with no discount). Fig. 1 shows the incorrect query. Note that the WHERE predicate is in DNF and incorporates two CPs with a total of four clauses, CP1 (*Year* > 2007 and *Price* > 100), and CP2 (*ZipCode* = 10008 and *Discount* = 0). The goal is to identify the faulty clauses *Year* > 2007 and *ZipCode* = 10008 with SFL techniques. The simple query in Fig. 1 is used as a running example throughout the paper.

```
SELECT orderid
FROM Order
WHERE (year > 2007 AND price > 100)
      # year > 2007 should be year > 2009
      OR (zipcode = 10008 AND discount = 0)
      # zipcode = 10008 should be zipcode = 10007
```

**Fig. 1.** Example incorrect SQL query.

**Table 1**
Example order table.

| *Orderid* | *Year* | *Price* | *Discount* | *ZipCode* |
|---|---|---|---|---|
| 1 | 2008 | 110 | 0 | 22102 |
| 2 | 2014 | 120 | 10 | 22102 |
| 3 | 2013 | 110 | 5 | 20017 |
| 4 | 2006 | 80 | 5 | 20017 |
| 5 | 2014 | 90 | 0 | 10007 |

**Table 2**
Selected similarity-based techniques.

| Order | Name | Suspiciousness formulas |
|---|---|---|
| 1 | Naish2 | $S(c) = c_{ef} - \frac{c_{ep}}{T_p+1}$ |
| 1 | Wong1 | $S(c) = c_{ef}$ |
| 2 | Kulczynski2 | $S(c) = \frac{1}{2} * (\frac{c_{ef}}{T_f} + \frac{c_{ef}}{c_{ef}+c_{ep}})$ |
| 3 | Ochiai | $S(c) = \frac{c_{ef}}{\sqrt{T_f * (c_{ef}+c_{ep})}}$ |
| 4 | Tarantula | $S(c) = \frac{c_{ef}/T_f}{c_{ef}/T_f + c_{ep}/T_p}$ |

## 2.2. Similarity-based SFL

Coefficient formulas are used in statistics to measure the relationship between two variables. For example, the coefficient formula $x = 0.4y$ describes a linear relationship between variables $x$ and $y$. It can also be extended to indicate the *similarity* between variables. Similarity-based techniques use the coefficient formulas to distinguish faulty program entities from correct program entities.

This study investigated five similarity-based techniques from two previous research papers (Xie et al., 2013; Le et al., 2013). In a theoretical study, Xie et al. (2013) investigated 30 similarity-based techniques and concluded that five techniques should be more effective than the others. Xie et al. placed these five techniques into two groups, ER1 and ER5, and showed that the techniques in each group are equivalent. Consequently, for this study we selected a representative from each group: Naish2 from ER1 and Wong1 from ER5. To these we added Tarantula (Jones and Harrold, 2005) and Ochiai (Abreu et al., 2007). In an empirical study, Le et al. (2013) compared the five theoretically superior techniques with Tarantula and Ochiai, finding that Ochiai was the most effective. We also added Kulczynski2 Naish et al. (2011), which the theoretical study showed to be better than Tarantula and Ochiai (though worse than the techniques in ER1 and ER5).

These five techniques are summarized in Table 2. They are ordered according to the effectiveness claimed by Xie et al. (2013):

**Table 3**
Suspiciousness score computation.

| Clauses | | Individual rows | | | | | Suspiciousness |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | Scores |
| 1 | *Year* > 2007 | | | | | | |
| | True | ✓ | ✓ | ✓ | | ✓ | 0.6 |
| | False | | | | ✓ | | 0 |
| 2 | *Price* > 100 | | | | | | |
| | True | ✓ | ✓ | ✓ | | | 0.42 |
| | False | | | | ✓ | ✓ | 0.6 |
| 3 | *ZipCode* = 10008 | | | | | | |
| | True | | | | | | 0 |
| | False | ✓ | ✓ | ✓ | ✓ | ✓ | 0.5 |
| 4 | *Discount* = 0 | | | | | | |
| | True | ✓ | | | | ✓ | 1 |
| | False | | ✓ | ✓ | ✓ | | 0 |
| Test case result | | F | P | P | P | F | |

Naish2 and Wong1 are the most effective, followed by Kulczynski2, Ochiai, and Tarantula (the names are also taken from that paper).

The suspiciousness formulas use four variables. $T_f$ is the total number of failing tests and $T_p$ is the total number of passing tests. For a program entity $c$, $c_{ef}$ is the number of times $c$ is executed by the failing tests and $c_{ep}$ is the number of times it is executed by the passing tests. Although these techniques were originally designed to rank statements, Nguyen et al. (2013) applied Tarantula to SQL predicates and calculated suspiciousness for true and false evaluations separately. This paper adopts the same methodology and applies these techniques to clauses. For a clause $c$, we compute the suspiciousness score's true evaluation $S_t(c)$ and false evaluation $S_f(c)$. In $S_t(c)$, a clause is deemed to be "executed" only if it evaluates to true. $c_{ep}$ is the number of passing tests that resulted in true and $c_{ef}$ is the number of failing tests that resulted in true. In $S_f(c)$, a clause is deemed to be "executed" only if it evaluates to false. $c_{ep}$ is the number of passing tests that resulted in false and $c_{ef}$ is the number of failing tests that resulted in false. We then calculate the final suspiciousness score as the sum $S(c) = S_t(c) + S_f(c)$.

Table 3 illustrates the computation of the suspiciousness score for Tarantula using the running example from Fig. 1. The columns under *Individual Rows* show the five rows from Table 1. For each row, if a clause evaluates to true, then the *True* position is checked; otherwise, the *False* position is checked. The final row shows whether the test case passed or failed. The column *Suspiciousness Scores* is computed from the formulas in Table 2. The total number of test cases is five, with $T_f = 2$ (orders 1 and 5) and $T_p = 3$ (orders 2, 3, and 4). As an example, consider the true evaluations of the clause *Year* > 2007. From the three rows that passed, two evaluated to true (2 and 3); hence, $c_{ep} = 2$. From the two rows that failed, two evaluated to true (1 and 5); hence $c_{ef} = 2$. Therefore, $S_t(c) = (2/2)/(2/2 + 2/3) = 0.6$. Similarly, $S_f(c) = 0$. Altogether, the suspiciousness score is $S(c) = S_t(c) + S_f(c) = 0.6$.

### 2.3. Statistics-based SFL

Statistics-based fault localization applies statistical models to spectrum data and generates suspiciousness rankings. We studied four statistics-based techniques, Crosstab (Wong et al., 2012), Liblit et al. (2005), SOBER (Liu et al., 2005), and Mann-Whitney (Zhang et al., 2011). Crosstab was originally applied to statements. As explained in Section 2.2, Crosstab can also be applied to predicates or clauses. Liblit, SOBER, and Mann-Whitney were originally applied to logical predicates in program decision statements. Since clauses are essentially elementary predicates without logical operator, these techniques can be directly applied to clauses. In addition, they can also be applied to statements. When they were used to identify faulty statements (Zhang et al., 2011; Wong et al., 2012), the predicates were ranked first, and the corresponding statements in top ranked predicates were considered to be suspicious.

#### 2.3.1. Crosstab

Wong et al. (2012) used "crosstab" to refer to a cross tabulation-based analysis. Each statement is associated with a table that contains four variables that indicate the number of times the statement is executed or not executed in passing and failing tests. Based on the crosstab, it proposes a null hypothesis that the execution result is independent of whether the statement was covered. Then it uses the chi-square test to see if the null hypothesis can be rejected. The chi-square statistical model is shown in Eq. 1, where $E_{ef} = \frac{(c_{ef}+c_{ep})*T_f}{T_p+T_f}$, $E_{ep} = \frac{(c_{ef}+c_{ep})*T_p}{T_p+T_f}$, $E_{nf} = \frac{((T_f-c_{ef})+(T_p-c_{ep}))*T_f}{T_p+T_f}$, and $E_{np} = \frac{((T_f-c_{ef})+(T_p-c_{ep}))*T_p}{T_p+T_f}$.

$$\chi^2(c) = (c_{ef} - E_{ef})^2/E_{ef} + (c_{ep} - E_{ep})^2/E_{ep} + (T_p - c_{ep} - E_{np})^2/E_{np} + (T_f - c_{ef} - E_{nf})^2/E_{nf} \quad (1)$$

$\chi^2(c)$ is then compared with the chi-square critical value $\chi^2$ found in chi-square distribution table at a given level of significance. If $\chi^2(c) > \chi^2$, the null hypothesis is rejected. It also means that the execution result depends on the statement coverage. In other words, the statement is *associated* with the fault. To evaluate the degree of association between the statement and the execution result, the suspiciousness score, $\zeta(c)$, is calculated. $\varphi(c)$, calculated in Eq. 2, is then used to compute the suspiciousness $\zeta(c)$ in Eq. 3.

$$\varphi(c) = \frac{c_{ef}/T_f}{c_{ep}/T_p} \quad (2)$$

$$\zeta(c) = \begin{cases} \chi^2(c)/(T_f + T_p) & \text{if } \varphi(c) > 1 \\ 0 & \text{if } \varphi(c) = 1 \\ -\chi^2(c)/(T_f + T_p) & \text{if } \varphi(c) < 1 \end{cases} \quad (3)$$

Applying Crosstab to clauses is similar to applying similarity-based techniques. We calculate suspiciousness scores for true and false evaluations and get the final suspiciousness by summing them: $\zeta(c) = \zeta^t(c) + \zeta^f(c)$. For clause $c =$ *Year* > 2007, we compute $\zeta^t(c) = 0$ and $\zeta^f(c) = -0.16$. The final suspiciousness score $\zeta(c) = \zeta^t(c) + \zeta^f(c) = -0.16$. The suspiciousness score is calculated in a similar way for other clauses.

#### 2.3.2. Liblit

Liblit et al. (2005) assumes that predicates that evaluated only to true in failing tests are more suspicious than other predicates. For a predicate $p$, Liblit calculates a difference (*Increase*($p$) in Eq. 6) between the probability of how likely $p$ can fail tests (*Context*($p$) in Eq. 4) and the probability of how likely $p$ can fail tests when evaluated to true (*Failure*($p$) in Eq. 5). The predicate is suspicious if the difference is large.

$$Context(p) = Pr(Crash|\ p\ observed) \quad (4)$$

$$Failure(p) = Pr(Crash|\ p\ observed\ \mathbf{true}) \quad (5)$$

$$Increase(p) = Failure(p) - Context(p) \quad (6)$$

When applying Liblit to a clause $c$, the definition of *Increase*($c$) remains the same as the definition of *Increase*($p$). As example, for clause $c =$ *Year* > 2007, $Context(c) = Pr(Crash|\ c\ observed) = T_f/(T_f + T_p) = 2/5 = 0.4$; $Failure(c) = Pr(Crash|\ c\ observed\ \mathbf{true}) = \mathbf{c^t_{ef}}/(\mathbf{c^t_{ef}} + \mathbf{c^t_{ep}}) = \mathbf{2/4 = 0.5}$; and $Increase(c) = Failure(p) - Context(p) = 0.1$. Similarly, *Increase*($c$) is calculated for the other three clauses. Then, the clauses are ranked by the *Increase*($c$) to indicate their suspiciousness.

**Table 4**
Evaluation bias.

| Clause | | Individual row | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 |
| 1 | *Year* > 2007 | 1 | 1 | 1 | 0 | 1 |
| 2 | *Price* > 100 | 1 | 1 | 1 | 0 | 0 |
| 3 | *ZipCode* = 10008 | 0 | 0 | 0 | 0 | 0 |
| 4 | *Discount* = 0 | 1 | 0 | 0 | 0 | 0 |
| Test result | | F | P | P | P | F |

#### 2.3.3. SOBER

SOBER (Liu et al., 2005) works by defining what it calls an *evaluation bias*, which estimates the probability that a predicate $p$ will evaluate to true during execution. Let $n_t$ be the number of times a predicate $p$ evaluates to *true* and $n_f$ be the number of times $p$ evaluates to *false* over a set of test executions. The evaluation bias, $\pi(p)$, is given by Eq. 7. SOBER then calculates the distribution of evaluation bias for $p$ on passing tests and failing tests, denoted as $f(X|\theta_p)$ and $f(X|\theta_f)$. If the difference between $f(X|\theta_p)$ and $f(X|\theta_f)$ is large, $p$ is suspicious. Eq. 8 calculates the similarity $L(P)$ between $f(X|\theta_p)$ and $f(X|\theta_f)$ ($Sim$). SOBER then characterizes the distributions $f(X|\theta_p)$ and $f(X|\theta_f)$ for passing and failing test evaluation bias sets, using mean and variance and assuming normal distribution. Eq. 9 computes the suspiciousness score $S(P)$ from $L(P)$. In Eq. 9, $\sigma_p$ is the mean of the passing test evaluation bias set, $m$ is the number of test cases, and $\varphi(Z)$ is the probability density function of N(0,1) for the failing test evaluation bias set.

$$\pi(p) = \frac{n_t}{n_t + n_f} \tag{7}$$

$$L(P) = Sim(f(X|\theta_p), f(X|\theta_f)) \tag{8}$$

$$S(P) = -log(L(P)) = \log(\frac{\sigma_p}{\sqrt{m}\varphi(Z)}) \tag{9}$$

When using SOBER for a clause $c$, if the clause evaluates to *true*, then $n_t$ is 1 and $n_f$ is 0, thus $\pi(c)$ is 1. Similarly, when $c$ evaluates to *false*, $\pi(c)$ is 0. The evaluation bias for the running example in Fig. 1 is shown in Table 4. For clause *Year* > 2007, the passing test evaluation bias set $f(X|\theta_p)$ is {1,1,0} and the failing test evaluation bias set $f(X|\theta_f)$ is {1,1}. Applying Eq. 9 to the clause $c$, we get $S(c) = 0.22$. The suspiciousness score $S(c)$ is calculated in a similar way for each clause.

#### 2.3.4. Mann-Whitney

Zhang et al. (2011) observed that the evaluation bias for predicates may not be distributed normally, so applied the non-parametric statistic tests Wilcoxon and Mann-Whitney to compare the similarity between $f(X|\theta_p)$ and $f(X|\theta_f)$. Because Wilcoxon is used for paired data and the evaluation bias of failing and passing tests are not paired, we used Mann-Whitney in this study.

Mann-Whitney is calculated for clauses in two steps. First, it calculates evaluation bias sets $V_p$ for passing tests $P$ and $V_f$ for failing tests $F$, then ranks $V_p$ and $V_f$. Then it creates ranking sets $R_p$ and $R_f$ that have the rankings for $V_p$ and $V_f$. For the clause *Year* > 2007, $V_f$ is {1, 1} and $V_p$ is {1, 1, 0}. Among the five elements in $V_f$ and $V_p$, the rank of element 0 in $V_p$ is 1 and the rank of each of the other elements is 2.25[3] Mapping the rank-values back to the evaluation bias sets $V_p$ and $V_f$, we get the rank-value sets $R_f = \{2.25, 2.25\}$ and $R_p = \{2.25, 2.25, 1\}$. Second, Mann-Whitney measures the difference between $R_f$ and $R_p$ by enumerating all possible rank-value sets. Let $m$ denote the number of elements in $V_p$ and $n$ denote the number of elements in $V_f$. Mann-Whitney enumerates all possible sets $S_i$ containing $m$ elements from the $m + n$ elements of $V_p$ and $V_f$. The total number of such sets is $K = \binom{n+m}{m}$. Let $K_l$ be the number of sets whose sum of rankings is less than that of $R_p$, and $K_h$ be the number of sets whose sum of rankings is greater than that of $R_p$. The suspiciousness ranking for $p$ is the negative of the minimum of $K_l/K$ and $K_h/K$ in Eq. 10. The larger $R(p)$ indicates the predicate $p$ is more suspicious.

$$R(p) = -min(K_l/K, K_h/K) \tag{10}$$

For clause *Year* > 2007, $m = 2$ and $n = 3$ and $K = \binom{n+m}{m} = 10$. This union of all elements from $V_p$ and $V_f$ is {2.25, 2.25, 2.25, 2.25, 1}. $K_l = 0$ and $K_h = 10$. Thus, $R(p) = -min(K_l/K, K_h/K) = 0$. $R(p)$ can be calculated in a similar fashion for the other clauses.

## 3. Original exoneration-based SFL

Unlike previous SFL techniques, which were applied to general programs, we designed an exoneration-based technique that specifically targets faulty clauses in SQL predicates. This section reviews our original technique. It consists of two steps, creating slices of suspicious clauses in failing rows and exonerating innocent clauses from the suspicious clauses. We implemented this approach in a tool, named *Automated sqL predicaTe fAult localizeR* (ALTAR). We will use ALTAR to refer to the original exoneration-based technique.

### 3.1. Slicing

A *slice* refers to a set of program entities that are relevant to computed values such as test results (Weiser, 1981). ALTAR first creates slices according to binary evaluations of clauses with only failing rows. Each clause is associated with a suspiciousness counter, which is initialized to zero. The failing rows are evaluated against each clause and the suspiciousness is incremented if the clause is identified as suspect. A clause with a positive suspiciousness counter is a *suspicious clause*.

First, users need to provide test oracles to distinguish failing rows from passing rows. Each test oracle should be derived from the requirements as accurately as possible. We write general test oracles so that ALTAR can efficiently process large numbers of test cases (thousands or even millions). The test oracle recognizes rows that should be included by a correct query, and rejects rows that should be excluded. Thus, rows that are included in the query result and that satisfy the test oracle are placed into the included group ($R_i$); rows that are excluded from the result and that do not satisfy the oracle are placed into the excluded group ($R_e$); rows that are included in the result but that do **not** satisfy the oracle are placed into the superfluous group ($R_s$), and rows that were excluded from the result but that **did** satisfy the test oracle are placed into the absent group ($R_a$).

Once the test oracle is created, clauses are sliced as follows. For a correct query, a superfluous row should evaluate to false in at least one clause in each CP. For an incorrect query, the superfluous row evaluates to true for all the clauses in at least one CP. Therefore, in each CP that is "all-true," every clause is suspect. Similarly, for a correct query, an absent row should satisfy all the clauses in at least one CP. For an incorrect query, the absent row evaluates to false in all CPs. Therefore, in each CP, all failing clauses are suspect.

We illustrate finding suspicious clauses with our running example from Fig. 1 and the incorrect predicate ((*Year* > 2007) ∧ (*Price* > 100)) ∨ ((*ZipCode* = 10008) ∧ (*Discount* = 0)). We refer to the four clauses in this predicate as $C_1$, $C_2$, $C_3$, and $C_4$. This predicate consists of two conjunctions: $CP_1 = C_1 \wedge C_2$ and $CP_2 = C_3 \wedge C_4$. Row

[3] The average rank is calculated by adding 1 / (number of elements with the same value) to its original rank. In this example, the average rank for element 1 is $2 + 1/4 = 2.25$.

**Table 5**
Finding suspicious clauses with slicing.

| Clause | | Row eval. | | Suspiciousness |
|---|---|---|---|---|
| | | 1 ($R_s$) | 5 ($R_a$) | Counter |
| $C_1$ | $Year > 2007$ | T | | 1 |
| $C_2$ | $Price > 100$ | T | F | 2 |
| $C_3$ | $ZipCode = 10008$ | | F | 1 |
| $C_4$ | $Discount = 0$ | | | 0 |

$Orderid = 1$ in Table 1 was classified as superfluous ($R_s$), so it should have failed on at least one clause in each CP. It failed on one clause of $CP_2$, but passed both clauses of $CP_1$. Consequently, both clauses of $CP_1$ ($C_1$ and $C_2$) are suspect, and their suspiciousness counters are incremented. Similarly, row $Orderid = 5$ was classified as absent ($R_a$), so it should have passed at least one of the CPs. It failed both: It failed $C_2$ in $CP_1$ and it failed $C_3$ in $CP_2$. Consequently both $C_2$ and $C_3$ are suspect, and their counters are incremented. Table 5 shows the clauses, the evaluation results, and their suspiciousness counters.

### 3.2. Exoneration

The row-based slicing technique reduces the search domain from all clauses to a set of suspicious clauses. However, some of the suspicious clauses identified in the slicing step may be innocent. For example, the innocent clause $C_2$ is identified as a suspicious clause in Table 5. The goal of exoneration is to remove innocent clauses. The exoneration technique of ALTAR is inspired by delta debugging (Zeller and Hildebrandt, 2002). ALTAR mutates (Ammann and Offutt, 2017) the failing rows by replacing column values of a failing row with corresponding values from a passing row, resulting in a *mutant*. The passing row used in the mutant is called a *replacement row*. ALTAR then checks whether the mutant is also a passing row by using the test oracles. If so, the columns of the mutated values are *fault-inducing*. A clause with a fault-inducing column is a *blamed* clause; the other clauses are *exonerated* by decrementing their counters. If the counter of a clause becomes 0, the clause is then considered innocent. As a specific example, consider a superfluous row ($R_s$). We choose an included row ($R_i$) that evaluates to true on the suspicious clauses as the replacement row, and substitute column values of the failing row with the corresponding value from the replacement row to create mutant rows. The goal is to find a mutant that is correctly included ($R_i$). The columns used to generate this mutant are considered to be *fault-inducing*. Similarly, for an absent row ($R_a$), we choose an excluded row ($R_e$) that evaluates to false on the suspicious clauses as the replacement row, and substitute column values of the failing row with the corresponding value from the replacement row to create mutant rows. The goal is to find a mutant that is correctly excluded ($R_e$). The columns used to generate this mutant are considered to be *fault-inducing*.

We demonstrate the process of exonerating superfluous rows with an example. Returning to our running example from Fig. 1, with the predicate $((Year > 2007) \wedge (Price > 100)) \vee ((Zipcode = 10008) \wedge (Discount = 0))$, the $R_s$ row $Orderid = 1$ implicated the clauses $C_1$ and $C_2$. To determine which should be blamed, we choose the row $Orderid = 2$ from group $R_i$ as a replacement because this row evaluates to true on suspicious clauses ($Year > 2007$) and ($Price > 100$). First, we mutate the column *Year* from $C_1$ by substituting the value of *Year* from the replacement row. Then, we use the test oracle to check the mutated row. If the mutated row belongs to group $R_i$, then $C_1$ is the correct suspect and $C_2$ is exonerated. Otherwise, we mutate the column *Price* from $C_2$. If the mutated row belongs to $R_i$, then $C_2$ is the correct suspect and $C_1$ is exonerated. Table 6 shows the original and mutated rows.

**Table 6**
Superfluous row mutants.

| Type | *Orderid* | *Year* | *Price* | *Discount* | *Zipcode* | Group |
|---|---|---|---|---|---|---|
| Original | 1 | 2008 | 110 | 0 | 22,102 | $R_s$ |
| Replacement | 2 | 2014 | 120 | 10 | 22,102 | $R_i$ |
| Mutant | 1 | **2014** | 110 | 0 | 22,102 | $R_i$ |
| Mutant | 1 | 2008 | **120** | 0 | 22,102 | $R_s$ |

**Table 7**
Absent row mutants.

| Type | *Orderid* | *Year* | *Price* | *Discount* | *Zipcode* | Group |
|---|---|---|---|---|---|---|
| Original | 5 | 2014 | 90 | 0 | 10,007 | $R_a$ |
| Replacement | 4 | 2006 | 80 | 5 | 20,017 | $R_e$ |
| Mutant | 5 | 2014 | **80** | 0 | 10,007 | $R_a$ |
| Mutant | 5 | 2014 | 90 | 0 | **20017** | $R_e$ |

```
SELECT Orderid
FROM Order
WHERE  Year > 2010
     #should be Year > 2009
     OR  Zipcode = 10008
     #should be Zipcode = 10007
     OR Discount > 10
```

**Fig. 2.** Another SQL Query.

Column Group indicates the group of the rows, and column Type shows if a row is an original row, a replacement row, or a mutated row. Mutated values are shown in bold font. Since the mutant on *Year* is in group $R_i$, we conclude that $C_1$ is responsible for the failure of row 1 and exonerate $C_2$.

Similarly, to exonerate suspected clauses in absent rows ($R_a$), the goal is to find a mutant in the excluded group $R_e$. The $R_a$ row $Orderid = 5$, which implicated $C_2$ and $C_3$, is suspicious. To determine which is innocent, we choose the row $Orderid = 4$ from group $R_e$ as a replacement because this row evaluates to false on suspicious clauses ($Zipcode = 10008$) and ($Price > 100$). First, we mutate the column *Price* from $C_2$ by substituting the value of *Price* from the replacement row. Then, we use the test oracle to check the mutated row. If the mutated row is excluded ($R_e$), then $C_2$ is suspect and $C_3$ is exonerated. Otherwise, we mutate the column *Zipcode* from $C_3$. If the mutated row belongs to group $R_e$, then $C_3$ is the correct suspect and $C_2$ is exonerated. Table 7 shows the original, replacement, and mutated rows. Since the mutation on $C_3$ is placed in $R_e$, we conclude that $C_3$ is responsible for the failure of row 5 and exonerate $C_2$.

After the exoneration process, the only positive suspiciousness counters are $C_1$ and $C_3$ (the counters for $C_2$ and $C_4$ are zero). The faulty clauses have been identified accurately.

### 3.3. Advanced approach

The basic approach is effective at detecting faults if the failing row is associated with one fault-inducing column. However, it cannot exonerate suspicious clauses when multiple fault-inducing columns are associated with the same failing row. We explain why with an example below. Fig. 2 shows another incorrect SQL query. Assume there is an order ($Orderid = 6$), shown in the first row of Table 8. The column *Group* shows the group to which the row belongs. The column *Type* represents the original rows, replacement rows, rows mutated for one column (Mutant1), rows mutated for two columns (Mutant2), and rows mutated for three columns (Mutant3). The row $Orderid = 6$ is absent ($R_a$) since it does not satisfy the predicate in the incorrect query. The row-based slicing step identifies that the clauses ($Year > 2010$), ($Zipcode = 10008$),

**Table 8**
Absent rows mutation with multiple fault-inducing columns.

| Row# | Type | *Orderid* | *Year* | *Discount* | *Zipcode* | Group |
|---|---|---|---|---|---|---|
| 1 | Original | 6 | 2010 | 0 | 10007 | $R_a$ |
| 2 | Replacement | 4 | 2006 | 5 | 20017 | $R_e$ |
| 3 | Mutant1 | 6 | **2006** | 0 | 10007 | $R_a$ |
| 4 | Mutant1 | 6 | 2010 | **5** | 10007 | $R_a$ |
| 5 | Mutant1 | 6 | 2010 | 0 | **20017** | $R_a$ |
| 6 | Mutant2 | 6 | **2006** | 0 | **20017** | $R_e$ |
| 7 | Mutant2 | 6 | **2006** | **5** | 10007 | $R_a$ |
| 8 | Mutant2 | 6 | 2010 | **5** | **20017** | $R_a$ |
| 9 | Mutant3 | 6 | **2006** | **5** | **20017** | $R_e$ |

and ($Discount > 10$) are suspicious, since they evaluate to false for the $Orderid = 6$ row. To exonerate innocent clauses with the basic approach, we create mutant rows (rows 3–5 in Table 8) by replacing column values with those from the replacement row (row 2 in Table 8). However, none of the mutated rows are excluded rows ($R_e$). The reason is that the $Orderid = 6$ row is associated with two fault-inducing columns (*Year* and *Zipcode*), thus, mutating a single column in the basic approach cannot identify fault-inducing columns. Therefore, we must mutate multiple columns at the same time. In Table 8, rows 6–8 show the rows created by mutating two columns of the three columns, *Year, Discount*, and *Zipcode*. Row 9 shows the mutant row when mutating all the three columns. Row 6 is in $R_e$ but Row 7 and 8 are not, therefore, *Year* and *Zipcode* are fault-inducing columns. Row 9 is also in $R_e$ because the three mutated columns include the two fault-inducing columns. Thus, we do not exhaust all combinations. Instead, we stop when the minimum set of fault-inducing columns is found. To find $k$ fault-inducing columns associated with a failed row, we need to check a total of $\sum_{m=1}^{k}(\binom{n}{1} + \binom{n}{2} \ldots + \binom{n}{m})$ mutated rows. $\binom{n}{m}$ represents all combinations that contain $m$ columns from $n$ columns.

Another limitation of the basic approach is that it cannot detect fault-inducing columns when they are not included in the predicate. For example, a faulty clause $Modified\_date > 2014$ mistakenly used column *Modified_date* instead of column *Created_date*. The basic approach cannot detect that *Created_date* should have been used since it is not included in the predicate and will never be used to create mutants. Therefore, we have to traverse the combinations of all columns in the table regardless of whether they are used in the predicate to find fault-inducing columns.

To solve the two issues in the basic approach above, we extend the basic approach to an advanced approach. The advanced approach iterates the combinations of all columns to address two issues: (1) a row associated with multiple fault-inducing columns, and (2) fault-inducing columns that do not exist in the predicate. Because the combination of all columns include any one combination of columns, the advanced approach covers the basic approach. Next, we present our advanced approach for exonerating suspicious clauses in Algorithms 1 and 2, for both superfluous and absent rows.

**Exonerating suspects implicated by superfluous rows.** Algorithm 1 is used to analyze superfluous rows. Algorithm 1 has four inputs: a superfluous row, *s_row*, a replacement row, *r_row*, a set of suspicious conjunctive predicates, *CPS*, and the tables used in the query, *T*. For a superfluous row, suspicious CPs evaluate to true and all the clauses in each suspicious CP evaluate to true. Thus, all clauses in each suspicious CP are initially marked as suspicious. Each suspicious CP must contain faults, however, some suspicious clauses may be innocent. The goal is to exonerate innocent clauses from each suspicious CP. Thus, for each suspicious CP, we first mutate columns in the existing clauses. For one CP, we create mutants, $MUT_k$, by replacing values of $k$-combinations of $n$ columns in the suspicious row *s_row* from the replacement row *r_row*, where $n$ is the number of columns included in that CP, *num_c*, and $k$ varies from 1 to $n$. $k$-combination mutants have all combinations that contain $k$ columns from $n$ columns. If a mutant is an included row ($R_i$), then fault-inducing columns are found. We stop and exonerate suspicious clauses that do not contain those fault-inducing columns by decreasing their suspiciousness counter by one. We continue the same process for the next CP.

If none of the mutants is in $R_i$, then the predicate does not contain any fault-inducing columns. To find them, we create mutants, $MUT\_ALL_k$, from $k$-combinations of $n$ columns, where $n$ is the number of all the columns included in table *T*, denoted as *num_all_c*, and $k$ varies from 1 to $n$. We go through all the remaining mutants after subtracting $MUT_k$ from $MUT\_ALL_k$, checking if any of them belong to $R_i$. If a mutant is in $R_i$, the mutated columns are fault-inducing in that CP. The suspicious clauses do not contain columns that can be exonerated and their suspiciousness counters are decreased by one. The fault-inducing columns identified should be included as missing clauses in that CP. Thus, the suspiciousness counters for missing clauses that contain the fault-inducing columns are increased by one.

**Exonerating suspects implicated by absent rows.** Algorithm 2 is used to analyze absent rows. Algorithm 2 has

**Algorithm 1** Exoneration algorithm for a superfluous row.

**Require:** A superfluous row *s_row*, a replacement row, *r_row*, a set of suspicious conjunctive predicates, CPS, and the tables, *T*.
**for** each $cp_i \in CPS$ **do**
  $COL$ = all columns included in $cp_i$
  **for** $k = 1 \ldots$ size of $COL$ **do**
    Create mutants, $MUT_k$, by replacing values of k-combinations of $COL$ on *s_row* with *r_row*
    **for** each $mut_j \in MUT_k$ **do**
      **if** $mut_j \in R_i$ **then**
        mark the mutated columns in $mut_j$ as fault-inducing
        jump to next $cp_i$
  $COL\_T$ = all columns in $T$
  **for** $k = 1 \ldots$ size of $COL\_T$ **do**
    Create mutants, $MUT\_ALL_k$, by replacing values of k-combinations of $COL\_T$ on *s_row* with *r_row*
    **for** each $mut_j \in MUT\_ALL_k - MUT_k$ **do**
      **if** $mut_j \in R_i$ **then**
        mark the mutated columns in $mut_j$ as fault-inducing
        jump to next $cp_i$

**Algorithm 2** Exoneration algorithm for an absent row.

**Require:** An absent row *a_row*, a replacement row, *r_row*, a set of suspicious *conjunctive predicates*, *CPS*, and the tables, *T*.
**for** $k = 1 \ldots$ size of CPS **do**
  Create mutants, $MUT$, by replacing values of $k$-combination of *CPS* on *a_row* with *r_row*
  **for** each $mut_j \in MUT_k$ **do**
    **if** $mut_j \in R_e$ **then**
      mark the mutated columns in $mut_j$ as fault-inducing
      stop and exit
$COL\_T$ = all columns in $T$
**for** $k = 1 \ldots$ size of $COL\_T$ **do**
  Create mutants, $MUT\_ALL_k$, by replacing values of $k$-combination of $COL\_T$ on *s_row* with *r_row*
  **for** each $mut_j \in MUT\_ALL_k - MUT_k$ **do**
    **if** $mut_j \in R_e$ **then**
      mark the mutated columns in $mut_j$ as fault-inducing
      stop and exit

four inputs: an absent row, *a_row*, a replacement row, *r_row*, a set of suspicious conjunctive predicates, *CPS*, and the tables used in the query, *T*. For an absent row, every CP is suspicious because every CP evaluates to false. Clauses that evaluate to false in a suspicious CP are suspicious. Unlike superfluous rows where all suspicious CP must contain faults, some suspicious CPs in absent rows may be innocent. Moreover, if a CP is found to contain faults, then all suspicious clauses in that CP must be faulty. The goal is to exonerate innocent CPs as well as all suspicious clauses in the innocent CPs. Therefore, Algorithm 2 iterates over CP combinations, instead of traversing column combinations for each CP as in Algorithm 1. Algorithm 2 creates mutants, $MUT_k$, from $k$-combinations of $n$ CPs, where $n$ is the number of all suspicious CPs, denoted as *num_cp*, and $k$ varies from 1 to $n$. In each mutant, we replace the values of all columns included in suspicious clauses at the same time. If a mutant is in $R_e$, the CPs that have the mutated columns are fault-inducing. Other CPs are innocent. We stop, exonerate innocent CPs and clauses, and exit the program.

If none of the mutants are in $R_e$, then the predicate does not contain any fault-inducing columns. We create mutants, $MUT\_ALL_k$, from $k$-combinations of $n$ columns, where $m$ varies from 1 to the number of columns of $T$, denoted as *num_all_c*, and $k$ varies from 1 to $n$. We go through all the remaining mutants after subtracting $MUT_k$ from $MUT\_ALL_k$, checking if any of them belong to $R_e$. If a mutant is in $R_e$, the mutated columns are fault-inducing. This step is very similar to the step in line 10 to 15 in Algorithm 1. The only difference is that Algorithm 1 checks if the mutated row is in $R_i$, while Algorithm 2 checks if the mutated row is in $R_e$. The suspiciousness counters for the clauses that have fault-inducing columns are increased by one. The fault-inducing columns should be included as missing clauses in a missing CP. Thus, the suspiciousness counters for the missing clauses in the missing CP are increased by one.

## 4. New exoneration-based SFL

Section 4.1 describes the efficiency problem that ALTAR encountered and Section 4.2 presents a significantly more efficient algorithm.

### 4.1. Efficiency problem

The existing SFL techniques described in Sections 2.2 and 2.3 rank all program entities by suspiciousness score. ALTAR applies the exoneration algorithms to remove innocent clauses and returns a result that is more precise than the results returned by other SFL techniques. However, the original exoneration algorithm can be extremely inefficient. Our previous paper (Guo et al., 2017) found that in an extreme case ALTAR took up to 25 minutes to localize faults with 234,244 failing tests. In contrast, Tarantula used around 30 seconds, although it was much less effective.

We use five variables to study the complexity of ALTAR:

1. Number of columns in all the query tables ($c$).
2. Number of suspicious CPs ($b$).
3. Number of suspicious clauses in each suspect CP ($n_i$, $i = 1, \ldots, b$).
4. Number of superfluous rows ($s$).
5. Number of absent rows ($a$).

In addition, we categorize the faults into two types:

1. *IN* Faults: all fault-inducing columns are used in the predicate.
2. *NIN* Faults: some fault-inducing columns are NOT used in the predicate.

For superfluous rows exonerated by Algorithm 1, the *IN* faults are found in lines 2–8, and the complexity is in the range $(\sum_{i=1}^{b}(s \cdot n_i),\ \sum_{i=1}^{b}(s \cdot 2^{n_i}))$. The *NIN* faults are found in lines 9–15, and the complexity is in the range $(\sum_{i=1}^{b}(s \cdot 2^{n_i}),\ s \cdot \sum_{i=1}^{b}(2^{n_i} + 2^{x_i}))$, where $x_i$ is the number of fault-inducing columns associated with the superfluous row in a suspect CP $b_i$. The worst case scenario is that each superfluous row is associated with all $c$ columns in each suspect CP. If that happens, the complexity is $s \cdot b \cdot 2^c$.

For absent rows exoneration in Algorithm 2, the *IN* faults are found in lines 1–6, where the complexity is in the range $(a \cdot b,\ a \cdot 2^b)$. The *NIN* faults are found in lines 7–13, and the complexity is in the range $(a \cdot 2^b,\ a \cdot (2^b + 2^x))$, where $x$ is the number of fault-inducing columns associated with the absent row. The worst case scenario is that each absent row is associated with $c$ fault-inducing columns. If that happens, the complexity is $a \cdot 2^c$.

Although the worst cases are likely to be rare, the potential for exponential running time clearly makes ALTAR impractical.

### 4.2. Redundant test case elimination

From the above analysis we learned that the complexity of Algorithms 1 and 2 is impacted by three factors: First, the *NIN* faults are more expensive than *IN* faults. Second, the complexity increases with the number of fault-inducing columns $x$. Third, the complexity increases with the number of failing rows $s$ and $a$.

The first two factors are associated with the nature of the fault, while the third is related to the testing database size. It is difficult to control the fault since the fault is unknown during fault localization. On the other hand, large databases can have a very large number of failing rows, possibly millions. Thus reducing the number of failing rows has the potential to greatly improve the efficiency. By examining Algorithms 1 and 2, we found a way to optimize them by identifying and eliminating redundant failing rows.

Two failing tests are considered equivalent if they are caused by the same faulty clauses. Therefore, eliminating one of the two rows does not affect the exoneration result. We identify equivalent rows with three conditions:

- S1: They belong to the same group: either superfluous $R_s$ or absent $R_a$.
- S2: The slices created by the two test rows have the same suspicious clauses.
- S3: The fault-inducing columns identified by the two test rows are the same.

We developed Algorithm 3 on top of ALTAR, and call it ALTAR2. Unlike ALTAR, ALTAR2 eliminates redundant failing tests from being processed during exoneration. Algorithm 3 has two general steps. First, it clusters the failing tests based on conditions $S_1$ and $S_2$ (lines 2 through 8). Second, it eliminates redundant tests by evaluating condition $S_3$ (lines 9 through 16). For each cluster $c$, ALTAR2 arbitrarily selects a test $t$. Then Algorithm 1 or Algorithm 2 is applied to exonerate the corresponding suspicious clauses, $SC_t$, and find its fault-inducing columns, $f_t$ (line 12). The algorithm next takes each other test $i$ in the same cluster, $c$, and mutates it with $f_t$. If a mutated test $i$ passes, the test satisfies the condition $S_3$ and can be eliminated from the cluster. The algorithm continues the process until $c$ is empty.

We illustrate the algorithm with the faulty SQL in Fig. 1. Assume a new Order table with four rows, as shown in Table 9. The last two columns in Table 9 show the row's *Group* and *Suspicious Clauses*, as identified by slicing. The first two failing rows, with Orderid 1 and 5, are the same as from Table 1. The other two failing rows have Orderids 6 and 7.

First, we group the rows into clusters based on conditions $S_1$ and $S_2$. The rows with the same *Group* and *Suspicious Clauses* belong to the same cluster. Orderid 1 and 6 should be grouped into

**Algorithm 3** The ALTAR2 algorithm.

**Require:** Failing tests $T$
1: Initialize an empty set $C$ for clusters
2: **for** each $t \in T$ **do**
3: Slice $t$ to get suspicious clauses, $SC_t$
4: **for** each $c \in C$ **do**
5: **if** $((c.group == t.group)$ && $(c.SC == SC_t))$ **then**
6: Add $t$ to $c$
7: **else**
8: Create a new cluster $c$ and add $c$ to $C$
9: **for** each $c \in C$ **do**
10: **while** $c$ is not *empty* **do**
11: Select an arbitrary test $t$ from $c$
12: Exonerate suspicious clauses of $t$ and find its fault-inducing columns $f_t$
13: **for** each other test $i \in C$ **do**
14: Create a mutant, $MUT_i$, by mutating $f_t$ in $i$
15: **if** $MUT_i$ *passes* **then**
16: Delete $i$ from $c$

**Table 9**
New order table.

| *Orderid* | *Year* | *Price* | *Discount* | *ZipCode* | Group | Suspicious Clauses |
|---|---|---|---|---|---|---|
| 1 | 2008 | 110 | 0 | 22,102 | $R_s$ | $C1$, $C2$ |
| 5 | 2014 | 90 | 0 | 10,007 | $R_a$ | $C2$, $C3$ |
| 6 | 2008 | 120 | 0 | 22,105 | $R_s$ | $C1$, $C2$ |
| 7 | 2006 | 135 | 0 | 10,008 | $R_a$ | $C2$, $C3$ |

**Table 10**
Mutated rows.

| *Orderid* | *Year* | *Price* | *Discount* | *ZipCode* | Group |
|---|---|---|---|---|---|
| 1 | **2014** | 110 | 0 | 22,102 | $R_i$ |
| 5 | 2014 | 90 | 0 | **20017** | $R_e$ |
| 6 | **2014** | 120 | 0 | 22,105 | $R_i$ |
| 7 | 2006 | 135 | 0 | **20017** | $R_e$ |

one cluster, and Orderid 5 and 7 should be grouped into another. Next, we pick Orderid 1 from the first cluster and execute the exoneration process to identify its fault-inducing column. The exoneration process described in Table 6 shows that we mutated the row with Orderid 1 twice and found the fault-inducing column, *Year*. We then use the fault-inducing column to mutate the other rows in the same cluster (Orderid 6). Table 10 shows the mutated rows. Since the mutated rows for Orderid 6 is in group $R_i$ (passing), $S_3$ is satisfied. Therefore, we can conclude that row Orderid 6 is equivalent to row Orderid 1 and should be eliminated. Similarly, we apply the same process to the second cluster, which contains Orderid 5 and Orderid 7. The exoneration process identifies the fault-inducing column for the Orderid 5 row to be *Zipcode*, as shown in Table 7. Then, we mutate the *Zipcode* column in row Orderid 7. The mutated row is in group $R_e$ (passing), thus $S_3$ is also satisfied. Row Orderid 7 is equivalent to row Orderid 5 and should be eliminated.

The original ALTAR algorithm needed to exonerate each of the four failing rows, and each exoneration created two mutants because there are two suspicious clauses. Thus, it generated eight mutants in total. The new ALTAR2 algorithm creates clusters for the failing rows, then only needs to exonerate one failing row from each cluster to find fault-inducing columns. Only two mutants are generated in each exoneration. In our example, rows Orderid 1 and Orderid 5 are exonerated, and each is mutated twice. After finding the fault-inducing column in each cluster, ALTAR2 uses it to mutate the remaining rows in the same cluster to determine if they are equivalent to the exonerated row. Since only the fault-inducing column needs to be mutated, rather than all columns involved in the suspicious clause, ALTAR2 only needs to generate one mutant for each remaining row. In our example, rows Orderid 6 and Orderid 7 are mutated only once. This means ALTAR only generates six mutants, 25% fewer than the original ALTAR. When a database has thousands or tens of thousands of test rows, the total execution time can be reduced significantly. For some of our tests, ALTAR took 30 minutes, while ALTAR2 needed less than 10 seconds.

## 5. Experiments

We compared the nine fault localization techniques described in Section 2 with our original technique, ALTAR, and our new technique, ALTAR2. This section presents research questions, subjects, procedure, results, and analysis.

### 5.1. Techniques selected

The eleven techniques we compared fall into three general categories. ALTAR and ALTAR2 are equivalently effective, so we compared their efficiency, but only compared ALTAR2's effectiveness with the other nine techniques.

1. Similarity-based: Naish2, Wong1, Kulczynski2, Ochiai, Tarantula.
2. Statistics-based: Crosstab, Mann-Whitney, SOBER, Liblit.
3. Exoneration-based: ALTAR and ALTAR2.

### 5.2. Objectives

Our experiment addressed four research questions when applying SFL techniques to clauses:

- RQ1: Can the techniques be rank-ordered in terms of effectiveness?
  - RQ1.1: What is the most effective technique overall?
  - RQ1.2: What is the most effective similarity-based technique?
  - RQ1.3: What is the most effective statistics-based technique?
- RQ2: Which is the most efficient technique overall?

Effectiveness is defined in terms of accuracy of finding faulty clauses. Efficiency is defined in terms of execution time. The detailed metrics are described in Section 5.4.

The effectiveness is critical to fault localization techniques. Several research papers have compared the effectiveness of different fault localization techniques as applied to general programs. Xie et al. (2013) theoretically showed that five techniques are the most effective if we assume 100% statement coverage. However, this assumption is often **not true** in practice. Le et al. (2013) studied seven similarity-based techniques with test suites that were less than 100% adequate, and found that Ochiai was the most effective, and more effective than the theoretically best formulas. Our study is different; we are studying predicates and clauses in SQL queries, so statement coverage does not apply. Zhang et al. (2011) compared Liblit, SOBER, and Mann-Whitney and found Mann-Whiteney was the most effective. Wong et al. (2012) compared Crosstab with Liblit and SOBER, and found that Crosstab was more effective. We compared ALTAR with two similarity-based techniques, Tarantula and Ochiai (Guo et al., 2017), and found ALTAR to be more effective. This paper is the first experiment to compare exoneration-based techniques with both similarity-based and statistics-based techniques.

The efficiency of most similarity-based and statistics-based techniques are very close. The techniques differ in coefficient formulas or statistical models used, so the time complexities are very

**Table 11**
Test subject databases.

| Databases | Tables | Columns (Average) | Rows (Total) |
|---|---|---|---|
| AdventureWorks | 68 | 14 | 759,241 |
| DBinventory | 19 | 14 | 234,837 |
| Employees | 6 | 5 | 3,919,015 |
| Mdbal | 127 | 18 | 749,743 |
| Polling_etl | 9 | 10 | 211,681 |

**Table 12**
Real faults.

| Database | E1 | E2 | E3 | E4 | E5 | E6 | Total |
|---|---|---|---|---|---|---|---|
| DBinventory | 6 | 7 | 5 | 9 | 6 | 2 | 35 |
| Mdbal | 8 | 4 | 5 | 3 | 3 | 2 | 25 |
| Polling_etl | 12 | 8 | 5 | 4 | 2 | 3 | 34 |
| Sum | 26 | 19 | 15 | 16 | 11 | 7 | 94 |

similar For example, when Wong et al. compared Crosstab with Tarantula (Wong et al., 2012), the time difference on the most complex program in their study was less than 0.15 seconds.

We are able to explore efficiency more accurately because we use a much larger set of tests than previous studies. The databases we use have millions of rows. Each database row is a test, so we have millions of tests, compared with only hundreds of tests in previous studies. RQ2 also helps us compare the performance of ALTAR2 with ALTAR, a problem identified in our previous study.

### 5.3. Experimental subjects

We selected five subject databases. Two were used in our previous paper (Guo et al., 2017), and three were new. Adventureworks[4] (AW) and employees[5] (EMP) are open source databases. Polling_etl (PEL), Dbinventory (DB), and Mdbal databases are proprietary databases owned by the first two authors' companies. Part of our agreement to use them in an experimental setting is that we are prohibited from disclosing certain details about the databases, especially their contents. The structures and sizes of the subject databases are shown in Table 11. *Columns* is averaged over all the tables in the relevant database.

We defined three sets of queries to examine the scalability of the techniques on queries with different complexities. *Simple queries* have three to five clauses, *moderate queries* have six to eight clauses, and *complex queries* have nine to twelve clauses. We created correct queries to use as subjects to evaluate our technique. For each database, we created five correct queries of each level of complexity. That is, we created 5 (queries) * 3 (levels of complexity) * 5 (databases) for a total of 75 total correct queries. Then, to create incorrect queries, we created six faulty versions of each of the 75 correct versions (one for each fault type E1 through E6). This resulted in a total of 450 incorrect queries. Note that the first five faulty versions (E1–E5) create single faults, whereas E6 creates multiple faults. So in total we have 75 incorrect queries with multiple faults and 375 incorrect queries with single faults. The correct queries served as controls in the experimental study, and the incorrect queries were used for fault localization.

To the best of our knowledge, this is the largest study of localizing faulty clauses in terms of the size of databases, the number the queries, and the complexity of the queries.

For AW and EMP, we obtained correct queries from their tutorial examples and manually constructed faulty variations by modifying the correct versions. For industrial application databases PEL, DB, and Mdbal, we extracted 94 naturally occurring faulty queries from industry applications and manually created the rest. Table 12 shows the number of real faults in each fault class. Polling_etl and Mdbal had more faults of type E1, while DBinventory had more faults of type E4. We cannot conclude which fault class is the most common in general, since it varies with application. However, we observe that there are relatively few composite faults (E6) in all three industrial databases (only 7%). This compares with 75 of 450, or 16.6%, of the faults in our study being composite.

We implemented ALTAR2 and the other ten techniques in a Ruby application, and made it available on github[6].

### 5.4. Procedure and metrics

We ran the experiments on a MacBook Pro-with two Intel i7 cores and 16 GB of RAM. For each faulty query, we ran the eleven techniques, recording the execution time and faults found.

Similarity-based and statistic-based techniques return a ranking of all program entities as the fault localization result. Thus, most prior research measured the effectiveness by the percentage of lines of code examined before reaching the faulty program entity over the total lines of code. We could not adopt this metric because the exoneration-based techniques precisely returns faulty clauses without a ranking.

Instead, we calculated the harmonic mean from information retrieval (Manning et al., 2008) to measure the effectiveness. Two variables are used to calculate the harmonic mean: *Expected* and *Actual*. *Expected* is the set of expected faulty clauses and *Actual* is the set of actual clauses identified by the fault localization technique. *Precision* ($P$) and *recall* ($R$) are calculated based on *Expected* and *Actual*, and $P$ is the proportion of reported clauses that are actually faulty (Eq. 11). $R$ is the proportion of faulty clauses that are actually reported (Eq. 12). We combined the precision and recall with their *harmonic mean* $H$ (Eq. 13). The higher the harmonic mean, the more effective an SFL technique is at localizing faults.

$$P = \frac{|\ Actual \cap Expected\ |}{|\ Actual\ |} \tag{11}$$

$$R = \frac{|\ Actual \cap Expected\ |}{|\ Expected\ |} \tag{12}$$

$$H = \frac{2 \cdot P \cdot R}{P + R} \tag{13}$$

The harmonic mean calculation can also be used to evaluate the effectiveness of rankings computed by similarity-based and statistics-based techniques. The only difference is that the *Actual* set is computed as clauses that have no lower ranking than the faulty clause. For example, for a predicate with a faulty clause $C_3$, where the ranking is $C_2$, $C_4$, $C_3$, $C_1$, *Actual* is $C_2$, $C_4$, $C_3$, because $C_1$ is ranked lower than $C_3$. For this case, $P = \frac{1}{3}$, $R = 1$, and $H = \frac{1}{2}$. However, if a technique assigns the same suspiciousness score to all clauses, then $H = 0$. This would mean the technique was completely ineffective because the ranking is not able to identify which clause is more likely to be faulty.

The exoneration-based techniques can localize multiple faults in one execution. However, similarity-based and statistics-based techniques are designed to identify one fault at one time. When using similarity or statistics-based techniques to find multiple faults, we computed a suspiciousness ranking for all the clauses under test. We first fixed a faulty clause with the highest rank. We then executed the technique again to fix the next faulty clause with the

[4] github.com/lorint/AdventureWorks-for-Postgres.
[5] github.com/datacharmer/test_db.
[6] https://github.com/carolfly86/altar.

**Table 13**
Effectiveness comparison between ALTAR2 and similarity-based SFLs .

| Database | ALTAR2 | | Naish2 | | Wong1 | | Kulczynski2 | | Ochiai | | Tarantula | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Avg | Sdv | Avg | Sdv | Avg | Sdv | Avg | Sdv | Avg | Sdv | Avg | Sdv |
| AW | 0.97 | 0.12 | 0 | 0 | 0 | 0 | 0.51 | 0.33 | 0.47 | 0.32 | 0.52 | 0.35 |
| DB | 0.99 | 0.05 | 0 | 0 | 0 | 0 | 0.38 | 0.30 | 0.36 | 0.24 | 0.37 | 0.26 |
| Emp | 0.98 | 0.13 | 0 | 0 | 0 | 0 | 0.49 | 0.31 | 0.49 | 0.32 | 0.50 | 0.32 |
| Mdbal | 0.95 | 0.15 | 0 | 0 | 0 | 0 | 0.45 | 0.34 | 0.45 | 0.33 | 0.48 | 0.37 |
| PEl | 0.97 | 0.12 | 0 | 0 | 0 | 0 | 0.53 | 0.37 | 0.53 | 0.36 | 0.49 | 0.33 |
| Single Fault | 0.97 | 0.13 | 0 | 0 | 0 | 0 | 0.48 | 0.35 | 0.46 | 0.34 | 0.47 | 0.35 |
| Composite Fault | 0.97 | 0.06 | 0 | 0 | 0 | 0 | 0.49 | 0.25 | 0.48 | 0.25 | 0.49 | 0.24 |
| Overall Avg | 0.97 | 0.12 | 0 | 0 | 0 | 0 | 0.45 | 0.33 | 0.46 | 0.32 | 0.47 | 0.33 |
| Overall Avg (excluding E4) | 0.97 | 0.12 | 0 | 0 | 0 | 0 | 0.58 | 0.31 | 0.55 | 0.30 | 0.57 | 0.31 |

**Table 14**
Effectiveness comparison between ALTAR2 and statistics-based SFLs .

| Database | ALTAR2 | | SOBER | | Liblit | | Mann-Whitney | | Crosstab | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Avg | Sdv | Avg | Sdv | Avg | Sdv | Avg | Sdv | Avg | Sdv |
| AW | 0.97 | 0.12 | 0.51 | 0.34 | 0.48 | 0.34 | 0.25 | 0.30 | 0.46 | 0.34 |
| DB | 0.99 | 0.05 | 0.49 | 0.33 | 0.43 | 0.32 | 0.14 | 0.23 | 0.49 | 0.33 |
| Emp | 0.98 | 0.13 | 0.57 | 0.36 | 0.47 | 0.31 | 0.23 | 0.30 | 0.41 | 0.29 |
| Mdbal | 0.95 | 0.15 | 0.53 | 0.36 | 0.43 | 0.33 | 0.18 | 0.30 | 0.36 | 0.30 |
| PEl | 0.97 | 0.12 | 0.54 | 0.36 | 0.39 | 0.29 | 0.23 | 0.29 | 0.56 | 0.37 |
| Single Fault | 0.97 | 0.13 | 0.53 | 0.37 | 0.44 | 0.33 | 0.27 | 0.29 | 0.46 | 0.35 |
| Composite Fault | 0.97 | 0.06 | 0.50 | 0.24 | 0.46 | 0.25 | 0.32 | 0.25 | 0.41 | 0.25 |
| Overall Avg | 0.97 | 0.12 | 0.53 | 0.35 | 0.44 | 0.32 | 0.21 | 0.29 | 0.45 | 0.33 |
| Overall Avg (excluding E4) | 0.97 | 0.12 | 0.64 | 0.32 | 0.53 | 0.31 | 0.25 | 0.26 | 0.55 | 0.26 |

highest rank in this run. We repeated the process until all the faults were fixed. Other researchers have tried to parallelize debugging (Jones et al., 2007; Högerle et al., 2014). These techniques partition test cases into fault-focusing clusters, and localize faults in parallel using the clusters. They can be used with our exoneration-based techniques as well as other SFL techniques, and would be a valuable future research direction.

Assume $N$ faulty clauses. For a similarity or statistics-based technique, effectiveness is the averaged harmonic mean $\frac{\sum_{i=1}^{N} a^i}{N}$, where $a^i$ is the harmonic mean of the $i$th execution. The efficiency is the total time for $N$ iterations (excluding the time spent on fixing the faults manually) $\sum_{i=1}^{N} t^i$, where $t^i$ is the execution time of the $i$th iteration.

### 5.5. Effectiveness

This section presents the results on effectiveness for each SFL technique and the statistical comparison results. We then analyze the issues that impacted effectiveness for each technique in Sections 5.5.3 through 5.5.6.

#### 5.5.1. Results

Tables 13 and 14 show the effectiveness of the ten techniques. Note that the exoneration-based techniques ALTAR and ALTAR2 have identical effectiveness, so we omit ALTAR data in both tables and use ALTAR2 to represent both exoneration-based techniques. The first column shows the five databases. Table 13 compares ALTAR2 with the five similarity based techniques, and Table 14 compares ALTAR2 with the four statistics-based techniques. For each subject, Tables 13 and 14 show the mean (*Avg*) and standard deviation (*Sdv*) of the effectiveness on the queries for each technique. The *Single Fault* rows show the effectiveness on queries with single faults and the *Multi Fault* rows show the effectiveness on queries with single faults. We can see that the effectiveness for localizing single and multi-faults are not significantly different for all techniques. *Overall Avg* shows the mean of the effectiveness on all the queries in the five databases. The last row, *Overall Avg (excluding E4)*, shows the mean and standard deviation of the effectiveness on all the queries except the queries that have E4 faults. We excluded E4 faults because ALTAR2 was the only technique that found any of the E4 faults. We explore this further in Section 5.5.4.

Table 13 shows that Naish2 and Wong1 were the least effective among the five similarity-based techniques, although they were shown to be theoretically the most effective Xie et al. (2013). Kulczynski2, Ochiai, and Tarantula had similar effectiveness. We check the statistical significance of the differences below. Table 14 shows that SOBER was the most effective statistics-based technique and Mann-Whitney was the least effective. The effectiveness of SOBER, Liblit, and Crosstab are relatively close, so we compare them statistically below.

Tables 13 and 14 show that ALTAR2 was the most effective technique overall. None of the similarity and statistical-based techniques had effectiveness greater than 0.55, whereas the lowest effectiveness score for ALTAR2 was 0.95. The standard deviation of ALTAR2 was also much lower than the other techniques, indicating that ALTAR2 performed consistently over all queries. **We can thus answer RQ1.1: ALTAR2 was the most effective technique at localizing faults in clauses among all 11 techniques.**

#### 5.5.2. Statistical comparison

To complete the answer to RQ1, we use statistical analysis to compare techniques with similar effectiveness. We use the paired one-tailed $t$-test because we had enough queries (450) to assume normal distributions. We did not need to statistically compare ALTAR2 because it was so much more effective than the others. Mann-Whitney, Naish2, and Wong1 were excluded because they were much less effective. We studied the remaining six techniques: Kulczynski2, Ochiai, Tarantula, Crosstab, SOBER, and Liblit. The hypotheses are shown below. X and Y can be any of the six techniques, giving a total of 15 pairs.

**Table 15**
Hypothesis testing results.

| Pairs (X-Y) | p-score | $H_0$ Rejected? | $\mu_d$ |
|---|---|---|---|
| Tarantula-Ochiai | 0.076 | N | – |
| Tarantula-Kulczynski2 | 0.39 | N | – |
| Tarantula-SOBER | 0.0003 | Y | –0.053 |
| Tarantula-Liblit | 0.025 | Y | 0.033 |
| Tarantula-Crosstab | 0.0897 | N | – |
| Ochiai-Kulczynski2 | 0.001 | Y | –0.020 |
| Ochiai-SOBER | 0.0001 | Y | –0.066 |
| Ochiai-Liblit | 0.18 | N | – |
| Ochiai-Crosstab | 0.7927 | N | – |
| Kulczynski2-SOBER | 0.0002 | Y | –0.046 |
| Kulczynski2-Liblit | 0.0043 | Y | 0.039 |
| Kulczynski-Crosstab | 0.2493 | N | – |
| SOBER-Liblit | 0.0001 | Y | 0.086 |
| SOBER-Crosstab | 0.0017 | Y | 0.072 |
| Liblit-Crosstab | 0.524 | N | – |

Null hypothesis ($H_0$):
There is no significant difference between technique X and technique Y in terms of effectiveness

Alternative hypothesis ($H_1$):
There is significant difference between technique X and technique Y in terms of effectiveness

Table 15 shows the p-score for each pair of techniques at the significant level of $\alpha = 0.05$. If p-score is less than $\alpha$ (0.05), then the hypotheses $H_0$ is rejected. The $H_0$ *Rejected?* column uses "*N*" to indicate $H_0$ is not rejected (technique X is not significantly different from Y), and "*Y*" to indicate $H_0$ is rejected (technique X is significantly different from Y). When $H_0$ is rejected, the $\mu_d$ column shows the average of differences between technique X and Y ($\mu_d = avg(x_i - y_i)$). A positive $\mu_d$ value indicates that technique X was more effective, and a negative value means that technique X was less effective. When $H_0$ is not rejected, $\mu_d$ shows "-," meaning "not applicable."

In summary, among the similarity-based SFLs, Kulczynski2 was slightly more effective than Ochiai, but there were no difference in the other pairs. Thus, these three techniques had similar effectiveness. **Therefore, we can answer RQ1.2 that Kulczynski2, Tarantula, and Ochiai were the most effective among the similarity-based SFLs.** Somewhat surprisingly, the techniques that were found to be theoretically the best (Naishi2 and Wong1) (Xie et al., 2013) were the least effective. **We can also answer RQ1.3, that SOBER was the most effective among the statistics-based SFLs, and Liblit and Crosstab were not significantly different in terms of effectiveness.**

The effectiveness of Liblit and Crosstab was close to that of Kulczynski2, Tarantula, and Ochiai. Thus, they are considered to be equivalent in effectiveness.

Now, we can answer RQ1. **The final order of effectiveness for the ten techniques is: ALTAR2** > SOBER > (Kulczynski2 = Tarantula = Ochiai = Liblit = Crosstab) > Mann-Whitney > **(Naish2 = Wong1).**

The following subsections describe types of issues that affected the effectiveness of all the techniques: (1) an issue that is specific to the exoneration-based technique, (2) issues that are common to all similarity-based and statistics-based techniques, (3) issues that are specific to the similarity-based techniques, and (4) issues that are specific to the statistics-based techniques.

*5.5.3. An issue that is specific to the exoneration-based technique*

Exoneration-based techniques might not localize faults accurately when multiple faulty clauses have the same columns. During the exoneration process, ALTAR2 identifies a fault-inducing column and associates it with the clauses that contain that column. If multiple clauses have the same fault-inducing columns, it cannot determine which clause is innocent and may report that all such clauses are faulty. This situation is very rare, thus, the overall effectiveness of ALTAR2 is still very high.

*5.5.4. Issues common to all similarity-based and statistics-based techniques*

Similarity-based and statistics-based techniques can only rank clauses that are included in the predicate under test. Thus, they cannot find missing clauses (fault type E4 in Section 2.1). In contrast, ALTAR2 can accurately report not only the missing clauses but also associated fault-inducing columns. Tables 13 and 14 (the last row) show that the similarity and statistics-based techniques (except Naish2, Wong1, and Mann-Whitney) are about 10% more effective when applied to SQLs that do not have E4 faults.

Another important point that reduces the effectiveness of previous SFL techniques is that they were designed for general programming languages. These SFL techniques work because they identify differences between which program locations were reached by failing and passing tests. However, for SQL analysis, all rows are executed by all clauses in the predicate equally. Thus, the previous SFL techniques do not accurately localize faulty clauses in SQL predicates.

*5.5.5. Issues specific to similarity-based techniques*

Similarity-based techniques rely on a coverage-based assumption, that is, program entities that are executed more frequently in failing tests than in passing tests are more likely to be faulty. However, this assumption does not hold when program entities are clauses. When executing a predicate with a test, all the clauses in the predicate are executed. Thus, the clauses are executed the same number of times in both failing and passing tests. We derive the suspiciousness formulas below to analyze their effectiveness.

Recall that we defined four variables for the similarity-based formulas in Section 2.2, $c_{ef}$, $c_{ep}$, $T_f$, and $T_p$. In addition, for the true evaluation of a clause $c$, $c^t_{ef}$ is the number of times $c$ evaluates to true in a failing test and $c^t_{ep}$ is the number of times $c$ evaluates to true in a passing test. Likewise, for the false evaluation of a clause $c$, $c^f_{ef}$ is the number of times $c$ evaluates to false in a failing test and $c^f_{ep}$ is the number of times $c$ evaluates to false in a passing test. $T_f$ is the total number of failing tests and $T_p$ is the total number of passing tests. $T_f$ and $T_p$ are constants for all clauses, while $c^t_{ef}$, $c^t_{ep}$, $c^f_{ef}$, and $c^f_{ep}$ are variables. The sum of the total number of times when $c$ evaluates to true and false in failing tests is equal to the total number of failing tests. That is, $c^t_{ef} + c^f_{ef} = T_f$. Similarly, the sum of the total number of times $c$ evaluates to true and false in passing tests is equal to the total number of passing tests. That is, $c^t_{ep} + c^f_{ep} = T_p$.

With the above $T_f$ and $T_p$ equations, we derive the formulas for the five similarity-based techniques as follows:

- Naish2:

$$S(c) = S(c)^t + S(c)^f = c^t_{ef} + c^f_{ef} - \frac{c^t_{ep}}{T_p + 1} - \frac{c^f_{ep}}{T_p + 1} = T_f - T_p/(T_p + 1) \quad (14)$$

- Wong1:

$$S(c) = S(c)^t + S(c)^f = c^t_{ef} + c^f_{ef} = T_f \quad (15)$$

- Kulczynski2 :

$$S(c) = \frac{1}{2} * \left(\frac{c^t_{ef}}{T_f} + \frac{c^t_{ef}}{c^t_{ef} + c^t_{ep}} + \frac{c^f_{ef}}{T_f} + \frac{c^f_{ef}}{c^f_{ef} + c^f_{ep}}\right) = \frac{1}{2} * \left(\frac{c^t_{ef} + c^f_{ef}}{T_f} + \frac{c^t_{ef}}{c^t_{ef} + c^t_{ep}} + \frac{c^f_{ef}}{c^f_{ef} + c^f_{ep}}\right) \quad (16)$$

- Ochiai :

$$S(c) = \frac{c^t_{ef}}{\sqrt{(T_f) * (c^t_{ef} + c^t_{ep})}} + \frac{c^f_{ef}}{\sqrt{(T_f) * (c^f_{ef} + c^f_{ep})}} \quad (17)$$

- Tarantula :

$$S(c) = \frac{c^t_{ef}/T_f}{c^t_{ef}/T_f + c^t_{ep}/T_t} + \frac{c^f_{ef}/T_f}{c^f_{ef}/T_f + c^f_{ep}/T_t} \quad (18)$$

Eqs. 14 and 15 show that the derivation results for Naish2 and Wong1 are constants, since $T_f$ and $T_p$ are constants. Thus, Naish2 and Wong1 gave identical ranks to all clauses. Therefore, their effectiveness was 0 for all the queries. Tarantula, Kulczynski2, and Ochiai were able to rank the clauses, because they contain variables $c^t_{ef}$, $c^f_{ef}$, $c^t_{ep}$, and $c^f_{ep}$. However, it is not obvious which formula is more effective. The experiment results also show the effectiveness of these three techniques were not significantly different. In previous studies, Xie et al. (2013) proved that Nashi2 and Wong1 are the theoretically best on statements, with the assumption that the tests covered all statements. But Le et al. (2013) showed that inadequate tests can affect their effectiveness. Nashi2 and Wong1 were very ineffective on clauses in our study, suggesting that program entities affect the effectiveness as well. The SFL techniques designed for statements may not be effective when applying to clauses.

#### 5.5.6. Issues specific to statistics-based techniques

Now we analyze the four statistics-based techniques.

**Crosstab:** Section 2.3 explained that Crosstab is similar to the similarity-based techniques, with the only difference being the statistical model used. The final suspiciousness was calculated for each clause by adding two suspiciousness scores for true and false evaluations. The statistical comparison in Section 5.5.2 shows that the effectiveness of Crosstab was not significantly different from that of Tarantula, Ochiai, and Kulczynski2.

**Liblit:** The assumption of the Liblit model is that program entities that evaluate to true in failing tests are likely to be faulty. We derive the Liblit formulas for a clause $c$ in Eqs. 19–21.

$$\begin{aligned} Context(c) &= Pr(Crash|\, c\, observed) \\ &= T_f/(T_f + T_p) \end{aligned} \quad (19)$$

$$\begin{aligned} Failure(c) &= Pr(Crash|\, c\, observed\, \textbf{true}) \\ &= c^t_{ef}/(c^t_{ef} + c^t_{ep}) \end{aligned} \quad (20)$$

$$\begin{aligned} Increase(c) &= Failure(c) - Context(c) \\ &= c^t_{ef}/(c^t_{ef} + c^t_{ep}) - T_f/(T_f + T_p) \end{aligned} \quad (21)$$

If $Increase(c)$ is large, $c$ is likely to have faults. Since $T_f$ and $T_p$ are constant, $Increase(c)$ relies on $c^t_{ef}$ and $c^t_{ep}$. That means the suspiciousness of $c$ correlates to the number of failing tests and passing tests when $c$ evaluates to **true**. However, this model does not consider what happens when $c$ evaluates to false. Thus, Liblit was not able to identify faulty clauses that evaluate to false.

**SOBER and Mann-Whitney:** Both SOBER and Mann-Whitney compare the distribution of evaluation bias in failing and passing tests. But they calculate the similarity of the distributions differently. Zhang et al. (2011) found that Mann-Whitney was found to be more effective than SOBER. However, our experiments found the opposite.

As explained in Section 2.3, the evaluation bias $\pi(c)$ for a given test row is either 1 or 0. So $V_f$ and $V_p$ consist of all 1s, all 0s, or a mix of 1s and 0s. In Mann-Whitney, if $V_p$ is all 1s or $V_f$ is all 0s, then $K_h$ is 0 because there are no sets with a higher sum of rankings than $V_p$. Similarly, if $V_p$ is all 0s or $V_f$ is all 1s, then $K_l$ is 0 because there are no sets with a lower sum of rankings than $V_p$. In the above four situations, the calculated $R(p)$ must be 0 according to Eq. 10. Such situations are in fact quite common. Among the four clauses in Table 4, three clauses have $R(p) = 0$. If all clauses in a predicate satisfy this condition, then they all get 0 as $R(p)$, which is essentially an ineffective ranking result. We also observed this in our experiment, where Mann-Whitney gave the same ranks to all the clauses in 58% of the subject queries, thus resulting in 58% of 0 effectiveness. In contrast, SOBER was able to rank the clauses. Therefore, Mann-Whitney was less effective than SOBER.

**Table 16**
Efficiency results (in seconds).

| Query types | Time (avg.) | | | | # of rows (avg.) | |
|---|---|---|---|---|---|---|
| | ALTAR2 | ALTAR | MW | Others | Failing | All |
| Simple | 2.77 | 84.72 | 24.34 | 9.29 | 2631 | 808,062 |
| Moderate | 2.22 | 102.78 | 27.74 | 10.83 | 799 | 790,389 |
| Complex | 3.41 | 117.40 | 34.38 | 14.37 | 955 | 455,254 |
| Single Fault | 2.24 | 56.37 | 23.73 | 9.08 | 1305 | 676,941 |
| Composite Faults | 5.88 | 336.53 | 84.48 | 24.67 | 2315 | 721,187 |
| Overall | 2.81 | 97.44 | 64.37 | 11.50 | 1462 | 683,825 |

### 5.6. Efficiency

We now turn to the efficiency of the techniques, first presenting the raw results, then comparing with manual debugging, followed by a discussion.

#### 5.6.1. Results

We found that the five similarity-based techniques had almost the same execution time as three of the statistic-based techniques: Liblit, SOBER, and Crosstab. Over 99% of the time was spent on executing tests and collecting runtime information. The computation of suspiciousness rankings took very little time. Thus, the time difference among these techniques was very small. This is consistent with Wong et al.'s results (Wong et al., 2012). ALTAR2 is the most efficient among all eleven techniques; Mann-Whitney took more time than the other eight similarity- and statistics-based techniques due to the non-parametric statistical model; and ALTAR was the least efficient. Therefore, Table 16 gives time in seconds for all the techniques, combined into four groups, ALTAR2, ALTAR, Mann-Whitney (MW), and the other SFL techniques (Others).

The *Types* column of Table 16 shows the queries in different groups: simple queries, moderate queries, complex queries, queries with single faults, and queries with multiple faults. The *Time (avg.)* column shows the time on average to localize faulty clauses in a query in each group. The *# of Rows (avg.)* column shows the number of failing rows and all rows executed for a query in each group on average. ALTAR2 averaged 2.8 seconds across the 450 queries, which is much faster than the other techniques. **Thus, the answer to RQ2 is that ALTAR2 is the most efficient technique.**

#### 5.6.2. Comparing SFL with manual debugging

We conducted a small study to compare SFL with manual debugging to determine if manual debugging is comparable to the automated SFL techniques in efficiency. We invited a developer who has sufficient background knowledge and working experience with the MDbal database schema to manually debug some queries. He was given the requirements and three faulty queries arbitrarily selected from each of the simple, moderate, and complex query groups. The manual debugging time on average to find the faulty clauses was 52 seconds for simple queries, 9.4 minutes for

moderate queries, and 5.56 minutes for complex queries, whereas the ALTAR2 techniques took less than 5 seconds for all queries. Most of the manual effort was comparing the requirements with the queries. This small study demonstrated that the manual debugging was much slower comparing to ALTAR2, thus we did not conduct further experiments on manual debugging.

#### 5.6.3. Analysis and discussion

Suppose a predicate has $k$ clauses, $n$ test rows, $p$ passing test rows, and $f$ failing rows. For the similarity-based and statistics-based techniques, $k$ clauses are executed against $n$ rows, with the time complexity of $O(n^*k)$. For $k$ clauses, suspiciousness scores are calculated with a similarity coefficient formula or a statistical model in $O(k)$. Then the scores are sorted in $O(k^*log(k))$. The total time complexity is $O(n*k+k+k*log(k))$. When $n$ is much larger than $k$, the complexity is dominated by $n^*k$. Mann-Whitney is more complex because it computes suspiciousness scores for the combinations of $p$ out of $n$, instead of $k$. The total time complexity for Mann-Whitney is $O(n*k+\binom{n}{p}+k*log(k))$. The computation can be substantial when $n$ is large. Thus, the complexity of Mann-Whitney is dominated by $\binom{n}{p}$ when $n$ is large.

ALTAR consists of two steps: slicing and exoneration. ALTAR2 has an additional redundant row elimination step. Comparing to the other nine SFL techniques, ALTAR2 and ALTAR only processes failing rows $f$. Thus, the time complexity of the slicing step is $O(f^*k)$ for ALTAR and ALTAR2. Regarding the time complexity of the exoneration step, the best case and the worst case can be very different. The best case happens when a faulty clause is associated with a single fault-inducing column. In this case, the exoneration process only checks the $s$ columns used in the suspicious clauses. In the worst case, multiple faults are associated with multiple fault-inducing columns, where all columns $c$ in the tables in the predicate are fault-inducing. In this scenario, the exoneration process must check all $\sum_{k=1}^{c}\binom{c}{k}=2^c-1$ combinations. The worst case can only happen when (1) the predicates contain all columns in the tables, and (2) all columns are fault-inducing. It seems likely that this situation would be extremely rare, especially as databases get large. ALTAR processes each failing row for exoneration, thus the time complexity for exoneration step is in the range $(O(f^*s), O(f^*2^c))$. The total complexity of ALTAR is in the range $(O(f*k+f*s), O(f*k+f*2^c))$.

ALTAR2 applies a redundant row elimination step after the exoneration step. We use $e$ to denote the number of the actual rows processed in the exoneration and redundant row elimination. Then the complexity of elimination is in the range of $(f, \sum_{i=1}^{f} i)$. In the best case, it eliminates all remaining failing rows after exonerating the first row. In the worse case, each elimination only removes one failing row. The time complexity of the exoneration process is similar to ALTAR except only $e$ rows are exonerated, and it is in the range $(O(e^*s), O(e^*2^c))$. The total complexity of ALTAR2 is in the range $(O(f*k+f+e*s), O(f*k+e*f+e*2^c))$. The number of actual rows $e$ is only a fraction of the number of failing rows $f$. In our study, the average of $e$ was four and average of $f$ was 1462. Consequently, the complexity of ALTAR2 is much smaller than ALTAR. In our experiments, both ALTAR and ALTAR2 found 91% of the faults in the best-case exoneration scenarios. In the other cases, the impact of $O(e^*2^c)$ on ALTAR2 is not significant, while the impact of $O(f^*2^c)$ on ALTAR is dramatic. Although we can neglect $O(e^*2^c)$ for ALTAR2 and conclude the complexity is dominated by $O(f^*k)$, we cannot neglect $O(f^*2^c)$ for ALTAR.

In summary, the time complexities of ALTAR2, ALTAR, Mann-Whitney, and the others are dominated by $O(f^*k)$, $O(f*k+f*2^c)$, $O(n^*k)$, and $O(\binom{n}{p})$. Since $O(f*k)<O(n*k)<O(\binom{n}{p}))<O(f*k+f*2^c)$, our analysis is consistent with the observed efficiency results.

### 5.7. Threats to validity

An external threat is that the subjects may not be representative. We ameliorated the threat by selecting five databases from different sources, two from open source repositories and three from industry. To increase diversity, we selected some queries from the original database domains and manually constructed additional queries with different complexities.

Another external threat is that the implementation of the ten techniques could have affected the results. We were careful to implement them exactly as described in the papers, and designed tests to ensure they worked as expected.

One construct validity threat is how we created faults. We used a combination of natural faults and manually constructed faulty SQL queries. Using a different source of faults could have led to different results. Our faulty queries include five different individual faults as well as composite faults. Compared to other fault localization empirical studies, we considered more types of faults.

Another internal threat is the measurement of execution time for similarity-based and statistics-based techniques to localize multiple faults. Similarity-based and statistics-based techniques cannot identify multiple faults with one run, and thus need to be repeated several times. We used the "perfect bug detection" assumption (Wong et al., 2016) to calculate the number of restarts needed for fixing all the faults. That is, we assume the faulty clause with the highest ranking can always be identified and fixed in each run, and the fault localization process will then be restarted with one less fault. So the number of times we need to restart the fault localization process is equivalent to the number of faulty clauses. However, this assumption may not hold in reality. Given an inaccurate ranking where the faulty clause is not ranked as the most suspicious, programmers need to spend more time examining and modifying the clauses from the top of the ranking until they arrive at the faulty clause. In other cases, the programmers may not be able to correctly fix the fault. As a result, the fault localization may be restarted more times than the actual number of faulty clauses. The efficiency of similarity-based and statistics-based techniques would be even worse than observed in the experiment. Nevertheless, this would not have affected our result that ALTAR2 is the most efficient technique.

## 6. Related work

Spectrum-based fault localization (SFL) uses dynamic information from test execution. Souza et al. (2016) and Wong et al. (2016) categorized fault localization techniques differently, emphasizing different features of the techniques. Our paper uses Souza et al.'s definition of SFL, in which any technique that uses information from test execution is spectrum-based. Souza et al. described two major categories of SFL: similarity-based and statistic-based. Although they are similar, they use different suspiciousness formulas. Our exoneration-based approach is somewhat similar to the program state-based techniques in Wong et al.'s work (Wong et al., 2016). A program state-based technique is *delta debugging* Zeller and Hildebrandt (2002), which compares program states in passing tests with those in failing tests. Although the exoneration-based approach was inspired by delta debugging, it is very different from the prior techniques that used delta debugging, such as by Jeffrey et al. (2008) and Zhang et al. (2006). They still calculate suspiciousness scores for all program entities, whereas the exoneration-based approach eliminates innocent entities and only reports faulty entities.

We focused on SFL techniques that were frequently compared in prior papers. We did not evaluate artificial intelligence-based or model-based techniques. Some techniques require graphs or

models to be built from source code, which are not applicable to SQL clauses.

Perez et al. (2014) presented a technique to increase the efficiency of code fault localization. They based their technique on Ochiai. Instead of computing the suspiciousness for each line of code, their technique computes the suspiciousness at a coarse level (for example, at a module level), and gradually refines the suspicious coarse-level element into the most suspicious detailed-level software components. Perez et al.'s technique assumes that the underlying fault localization algorithm (Ochiai) is equally effective at any given granularity level. Our experiment showed that Ochiai is *not effective* at the clause level of granularity, and in preliminary trials, we also found no evidence that Ochiai works equally effectively at multiplve levels of granularity.

We compared prior studies on SFL with ours in four aspects, program entities, types of techniques, program domains, and test oracles. Most prior SFL studies used statements as program entities, such as Naish et al. (2009), Le et al. (2013), and Kim and Lee (2014). Liu et al. (2005), Liblit et al. (2005), and Zhang et al. (2011) used predicates as program entities. Sarah et al. (Clark et al., 2011), Nguyen et al. (2013), Saha et al. (2011), and Akhter and Embury (2012) targeted predicates in embedded SQLs. This paper and our prior study (Guo et al., 2017) are the first to go to the detailed level of clauses as program entities. Previous SFL techniques did not describe how test oracles were constructed, so we cannot compare the impact of different types of test oracles. In those papers, the only assumption was that test oracles existed that could correctly distinguish passing and failing test cases. Our approach requires general test oracles, which, although more difficult that specific test oracles, is practical in situations like ours where we have thousands or millions of test cases.

Naish et al. (2009) compared 11 similarity-based techniques on statements assigned with weights. They found the most effective technique varies in the number of faults. Le et al. (2013) empirically compared Tarantula and Ochiai with the theoretically most effective techniques. They found Ochiai to be the most effective. Kim and Lee (2014) compared 32 similarity-based techniques and classified them into three groups, analyzing the characteristics of each group. Zhang et al. (2011) compared similarity-based and statistics-based techniques, finding that non-parametric statistics-based techniques were the most effective. Wong et al. (2012) compared Crosstab with SOBER and Liblit, concluding that Crosstab was more effective. Our study is the first to compare similarity-based, statistics-based, and exoneration-based techniques. Moreover, prior studies showed that many similarity-based techniques are similar in effectiveness, and the theoretically best techniques may not perform as well in practice when statement coverage is not achieved. We leveraged these conclusions by using the techniques that others found to be the most effective in our study.

Most empirical studies used general programs as experiment subjects such as the Siemens suite. A few studies used database and data-centric applications with embedded SQL queries (Clark et al., 2011; Nguyen et al., 2013; Saha et al., 2011; Akhter and Embury, 2012). SFL has also been applied to spreadsheets. Data dependence graphs are constructed based on relationships among the cell data. Hofer et al. (2013) compared Ochiai with a spectrum-enhanced dynamic slicing approach (SENDYS) (Hofer and Wotawa, 2012), and a constraint-based debugging approach (Abreu et al., 2012), finding that Ochiai and SENDYS were more effective. We focus on clauses in SQLs used in data-centric applications. In addition, the subject databases and faulty queries in our experiment are much larger than in previous studies.

Mutation analysis was previously used to localize faults and detect failures in SQL queries. MUtation-baSEd (MUSE) (Moon et al., 2014) mutates the program under test (PUT) to find the faulty statement. Our approach differs from MUSE in that we do not mutate the PUT (in our case PUT is the SQL predicate), but rather, we mutate the test data in failing test cases. Kaminski et al. (2011) showed that using TRF-TIF logic mutation operators can effectively detect failures in SQL queries, but does not address fault localization.

## 7. Conclusion and future work

To find faulty clauses in SQL predicates, we previously developed an exoneration-based technique, ALTAR, and defined the fault classes that our technique can detect (Guo et al., 2017). As opposed to existing SFL techniques, which rank every program entity, our exoneration-based technique can precisely localize the faulty clause and identify the fault type.

This paper significantly extends our previous conference paper (Guo et al., 2017) in four ways. First, we propose a new algorithm to address the efficiency problem of the original exoneration-based technique. Second, we evaluated the new exoneration-based technique, ALTAR2, finding that ALTAR2 was significantly more efficient than all other SFL techniques in finding faulty clauses. Third, we analyzed the applicability of nine similarity-based and statistics-based techniques when program entities are clauses. Fourth, we conducted a considerably larger experiment than in prior studies to compare three major SFL categories, similarity-based, statistics-based, and exoneration-based, with a total of ten techniques. We concluded that ALTAR2 was both the most effective and efficient at finding faults in SQL queries.

In the future, we hope to improve effectiveness and efficiency by exploring parallel debugging techniques. We also plan to use the exoneration-based technique to find faults in JOIN, GROUP BY, and other SQL clauses. We are also working on automatic repair of SQL queries.

## Acknowledgment

This work was partly funded by The Knowledge Foundation (KKS) through the project 20130085: Testing of Critical System Characteristics (TOCSYC).

## References

Abreu, R., Riboira, A., Wotawa, F., 2012. Constraint-based debugging of spreadsheets. In: 15th Iberoamerican Conference on Software Engineering (CIbSE), pp. 1–14.
Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2007. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION. Windsor, UK, pp. 89–98.
Akhter, J.M., Embury, S.M., 2012. Diagnosing faults in embedded queries in database applications. In: Joint EDBT/ICDT Workshops. Berlin, Germany, pp. 239–244.
Ammann, P., Offutt, J., 2017. Introduction to software testing, 2nd Cambridge University Press, Cambridge, UK. ISBN 978–1107172012
Bouwkamp, K., 2016. The 9 most in-demand programming languages of 2016. Online, Coding Dojo Blog. last access: September 2017. http://www.codingdojo.com/blog/9-most-in-demand-programming-languages-of-2016/.
Clark, S.R., Cobb, J., Kapfhammer, G.M., Jones, J.A., Harrold, M.J., 2011. Localizing SQL faults in database applications. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE). Lawrence, KS, USA, pp. 213–222.
Guo, Y., Motro, A., Li, N., 2017. Localizing faults in SQL predicates. In: Tenth International Conference on Software Testing, Verification, and Validation (ICST). Tokyo, Japan, pp. 1–11.
Hofer, B., Riboira, A., Wotawa, F., Abreu, R., Getzner, E., 2013. On the empirical evaluation of fault localization techniques for spreadsheets. In: 16th International Conference on Fundamental Approaches to Software Engineering. Rome, Italy, pp. 68–82.
Hofer, B., Wotawa, F., 2012. Spectrum enhanced dynamic slicing for better fault localization. In: 20th European Conference on Artificial Intelligence. Montpellier, France, pp. 420–425.
Högerle, W., Steimann, F., Marcus, F., 2014. More debugging in parallel. In: Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE). IEEE, Naples, Italy, pp. 133–143.
Jeffrey, D., Gupta, N., Gupta, R., 2008. Fault localization using value replacement. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 167–178.
Jones, J.A., Bowring, J.F., Harrold, M.J., 2007. Debugging in parallel. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA). ACM, London, United Kingdom, pp. 16–26.

Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the Tarantula automatic fault--localization technique. In: 20th IEEE/ACM International Conference on Automated Software Engineering. Long Beach, CA, USA, pp. 273–282.
Kaminski, G., Praphamontripong, U., Ammann, P., Offutt, J., 2011. A logic mutation approach to selective mutation for programs and queries. Inform. Softw. Technol. Els. 53 (10), 1137–1152. Special issue from the Mutation 2009 Workshop.
Kim, J., Lee, E., 2014. Empirical evaluation of existing algorithms of spectrum based fault localization. In: The International Conference on Information Networking 2014 (ICOIN2014). Phuket, Thailand, pp. 346–351.
Le, T.-D.B., Thung, F., Lo, D., 2013. Theory and practice, do they match? A case with spectrum-based fault localization. In: 2013 IEEE International Conference on Software Maintenance, pp. 380–383.
Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I., 2005. Scalable statistical bug isolation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, Chicago, IL, USA, pp. 15–26.
Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S., 2006. Statistical debugging: a hypothesis testing-based approach. IEEE Trans. Softw. Eng. 32, 831–848.
Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P., 2005. Sober: Statistical model-based bug localization. In: 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, Lisbon, Portugal, pp. 286–295.
Manning, C.D., Raghavan, P., Schütze, H., 2008. Introduction to information retrieval. Cambridge University Press.
Monperrus, M., 2014. A critical review of "Automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In: 36th International Conference on Software Engineering. Hyderabad, India, pp. 23–242.
Moon, S., Kim, Y., Kim, M., Yoo, S., 2014. Ask the mutants: Mutating faulty programs for fault localization. In: Seventh International Conference on Software Testing, Verification, and Validation (ICST). Cleveland, USA, pp. 153–162.
Naish, L., Lee, H.J., Ramamohanarao, K., 2009. Spectral debugging with weights and incremental ranking. In: 2009 16th Asia-Pacific Software Engineering Conference. Batu Ferringhi, Penang, Malaysia, Malaysia, pp. 168–175.
Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng.Methodol. 20 (3). 11:1–11:32. doi: 10.1145/2000791.2000795.
Nguyen, H.V., Nguyen, H.A., Nguyen, T.T., Nguyen, T., 2013. Database-aware fault localization for dynamic web applications. In: 29th IEEE International Conference on Software Maintenance (ICSM), pp. 456–459.
Perez, A., Abreu, R., Riboira, A., 2014. A dynamic code coverage approach to maximize fault localization efficiency. J. Syst. Softw. 90 (C), 18–28.
Saha, D., Nanda, M.G., Dhoolia, P., Nandivada, V.K., Sinha, V., Chandra, S., 2011. Fault localization for data-centric programs. In: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE). New York, USA, pp. 157–167.
Souza, H. A., Chaim, M. L., Kon, F., 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. Submitted for publication, last access January 2018. https://arxiv.org/abs/1607.04347.
Weiser, M., 1981. Program slicing. In: 5th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 439–449.
Wong, W.E., Debroy, V., Xu, D., 2012. Towards better fault localization: a crosstab-based statistical approach. IEEE Trans. Syst. Man Cybernet. Part C (Appl. Rev.) 42 (3), 378–396.
Wong, W.E., Go, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42, 707–740.
Xie, X., Chen, T.Y., Kuo, F.-C., Xu, B., 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng .Methodol. 22 (4). 31:1–31:40
Zeller, A., Hildebrandt, R., 2002. Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. 28 (2), 183–200.
Zhang, X., Gupta, N., Gupta, R., 2006. Locating faults through automated predicate switching. In: 28th International Conference on Software Engineering. ACM, New York, NY, USA, pp. 272–281.
Zhang, Z., Chan, W.K., Tse, T.H., Yu, Y.T., Hu, P., 2011. Non-parametric statistical fault localization. J. Syst. Softw. 84 (6), 885–905.

**Yun Guo** is a lead site reliability engineer at Cvent. Her research fields include SQL fault localization and automated program repair. She is passionate about bridging the gap between academia and industry. She received PhD in Computer Science from Volgenau School of Engineering at George Mason University in 2018. Her advisor was Dr. Jeff Offutt and Dr. Amihai Motro. She received her M.S. in Computer Science from George Mason University in 2011. Before that, she received a B.E. in Electronic Engineering from Xi'an Jiaotong University in 2008.

**Dr. Nan Li** is a lead software engineer in test at Medidata Solutions. He leads the research on big data testing, model-based testing, mutation testing, and mobile testing. He has also been developing several tools used for big data testing, model-based testing, and mutation testing. He serves on program committees for ICST, FSE, and MUTATION and will co-chair ICST 2019 industry track. He has been serving as a reviewer for multiple leading journals and conferences including CSUR, TSE, JSS, IST, SoSyM, STVR, ICST, and FSE. He won the 2015 Medidata Innovator Award and holds one patent. Li received a BE in Software Engineering from Beihang University in 2006 and received the PhD in Information Technology with concentration on software engineering from George Mason University in 2014. He is on the web at: https://nli-mdsol.github.io/.

**Dr. Jeff Offutt** is a Professor of Software Engineering at George Mason University. He has published over 175 refereed research papers (h-index of 61), and invented numerous widely used test techniques. Offutt is editor-in-chief of Wiley's journal of Software Testing, Verification and Reliability, co-founded the IEEE International Conference on Software Testing (ICST), and co-authored Introduction to Software Testing. He was awarded GMU's Teaching Excellence Award, Teaching With Technology, in 2013, was a GMU Outstanding Faculty member in 2008, 2009, and 2018, and his 2014 software engineering education paper was chosen by ACM as a notable paper. Current projects include the SPARC educational project, muJava, Testing of Critical System Characteristics (TOCSYC) and PILOT projects at University of Skovde, automatic repair of SQL queries, testing mobile and web applications, test automation, and usable security. Offutt received the PhD in Computer Science from the Georgia Institute of Technology in 1988. He is on the web at https://cs.gmu.edu/~offutt/.

**Dr. Amihai Motro** is a Professor of Computer Science at George Mason University. He holds a B.Sc. degree in mathematics from Tel Aviv University, a M.Sc. degree in computer science from the Hebrew University of Jerusalem, and a PhD degree in computer and information science from the University of Pennsylvania, and was previously on the faculty of the Computer Science Department at the University of Southern California His areas of interests include database management, data integration, cooperative databases, virtual enterprises, information retrieval, and information services. In these areas he published more than 100 papers in major journals and conferences and was the recipient of several research grants.