TOWARDS AUTOMATICALLY LOCALIZING AND REPAIRING SQL FAULTS

by

Yun Guo

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
(Computer Science)

Committee:

Towards Automatically Localizing and Repairing SQL Faults

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Yun Guo
Master of Science
George Mason University, 2011
Bachelor of Science
Xi'an JiaoTong University, 2008

Co-Director: Jeff Offutt, Professor
Co-Director: Amihai Motro, Professor
Department of Computer Science

Summer 2018
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my husband Nan Li. I dedicate this dissertation to my father Jin Guo and my mother Xiaoxia Ma. I dedicate this dissertation to my father in law Yukui Li and my mother in law Dehua Lv.

Acknowledgments

I would like to thank my husband Nan Li for his unconditional support. I would like to thank Professor Amihai Motro and Professor Jeff Offutt who made this possible.

Table of Contents

List of Tables

List of Figures

Abstract

TOWARDS AUTOMATICALLY LOCALIZING AND REPAIRING SQL FAULTS

Yun Guo, PhD

George Mason University, 2018

Dissertation Co-Director: Jeff Offutt

Dissertation Co-Director: Amihai Motro

As the standard database language, SQL statements can be complex and expensive to debug by hand. Automated fault localization and repair techniques have the potential to reduce cost significantly. I propose a novel fault localization and repair technique to repair faulty SQL statements. It targets faults in two common SQL constructs, JOIN and WHERE. It identifies the fault location and type precisely, and then creates a patch to fix the fault. I implemented this technique in a tool, and evaluated it on five medium to large-scale databases using 825 faulty queries with various complexity and faulty types. Experimental results showed that this technique can identify and repair WHERE faults more precisely than previous techniques, and also repair JOIN faults that previous techniques could not. Moreover, patches generated by this approach are more acceptable to engineers, and the tool is much faster.

Chapter 1: Introduction

1.1 Introduction

Database management systems (DBMS) were first created in the 1970s to help people manage information. With the advances of software technologies, almost all modern applications use database system to persist and analyze information. SQL (Structured Query Language) is the standard language used to retrieve and manipulate data in DBMSs. It was ranked as the most in-demand programming language in 2016 [1].

The volume of data stored in today's databases is increasing. A JPMorgan Chase Institute survey [2] reported that the data volume increased 42% in one year. It predicted that "more data will be produced in the next two years than has been produced from the dawn of civilization through today." As one of the largest database users in the world [3], AT&T hosts 1.9 trillion phone call records and more than 323 terabytes of data. The number is still increasing daily. As the volume of data increases, SQL evolved with DBMS in two directions: it has become more frequently used and more complex. Large amount of data often means large number of end users and busy system. The largest e-commerce company Alibaba processed 325,000 transactions per second at peak [4]. Each transaction requires multiple queries to retrieve or update their MySQL databases. In addition to simple data retrieval tasks, business analyst perform data analysis tasks to make business decisions and predict future performance. The data analysis tasks involve bringing data from various resources, transforming the data to complex structures, and loading the data into other data stores. These operations often consist of large number of complicated queries. I surveyed a software-as-service company of about 2500 employees with 143 project repositories, more than 20 terabytes of data, and 75 databases in production. They have in total 834,796 SQL scripts, each containing multiple queries. I randomly examined 143 queries from the

834,796 SQL scripts and found 89 of them to be complex queries (a complex query joins more than 6 tables and more than 8 clauses in the WHERE condition).

Debugging large number of complex queries is laborious and resource intensive. I investigated the efficiency of manual debugging. I invited a developer with domain knowledge to debug three queries. It took him 59 seconds to debug a simple query (with 3 clauses), 5 minutes 3 seconds for a medium query (with 6 clauses), and 9 minutes and 17 seconds for complex query (with 10 clauses). This experiment is presented in Chapter 9. Imagine a company with 500 developers with hourly wage of $40 and each developer spends 20 minutes a day to debug SQL faults. The total amount spend on debugging SQL queries is $6,667 daily. If this process can be automated, it has the potential to reduce cost significantly.

Although many papers have been published on automatically repairing general program code [5–9], only one attempt has been made to debug and repair SQL queries [10]. As a declarative language, SQL differs from procedural programming languages in two ways: (1) SQL is designed to process tabular data sets; and (2) SQL allows engineers to specify information to be retrieved (e.g., the table and column names), but not the algorithm for retrieving the information. Thus, debugging and repairing SQL queries is very different from general program debugging. The automated fault localization and program repairing techniques used in general programming languages are often ineffective when applied to SQL queries.

The human cost of manual debugging and ineffectiveness of existing techniques has motivated this research to find an effective and efficient fault localization and repair technique specifically targeting SQL faults.

1.2 Problem Statement

This research addresses the problem of localizing and repairing faults in two fundamental constructs of SQL queries: WHERE and JOIN clauses. Section 1.2.1 describes the research problem in this research: automatic fault localization and repair in SQL queries.

Section 1.2.2 shows a simple example to illustrate the research problem. Section 1.2.3 and section 1.2.4 explains the two problem in detail.

1.2.1 Problem Description

In database applications containing many complex SQL queries with many clauses, manual debugging SQL queries is laborious. My study found that given a faulty WHERE predicate with ten clauses, manual fault localization took almost ten minutes. After the fault is identified, developers need to spend more time to analyze and fix the fault. Applying automated fault localization and repair techniques in SQL queries can greatly reduce the effort. As SQL queries are used to process tabular data sets, a *test case* is a database row, and the *test oracle* evaluates whether the test data should be included in the result.

Researchers have studied how to automatically identify faults in database applications [11–13]. However, those papers consider the entire SQL statement as one line of code, indicating that the entire SQL statement contains errors. This is the first research to look for faults in individual *components* of SQL statements such as clauses. The state-of-the-art SQL query repair technique was proposed by Gopinath et al. [10]. It adopted a decision tree-based approach to generate patches from test suites and test oracles. However, the patches generated by this approach can not be accepted because it completely rewrites the query with unrelated clauses instead of fixing the faults. In addition, to the best of my knowledge, there is no fully automated technique that streamlines the entire process from fault localization to repairing SQL queries.

Problem Statement:

Currently, debugging and repairing complex SQL queries has three significant problems. First, existing fault localization techniques are not very effective or efficient at finding faulty entities (clauses) in SQL queries. Second, the existing methods to automatically fix the faulty queries are limited by quality and inefficiency. Third, there is no technique that integrates automatic fault localization and automatic fault repair.

Table 1.1: Order Table O

CustId	*OrderId*	*Year*	*Price*	*Discount*	*ZipCode*
1	1	2008	110	0	22102
1	2	2014	120	10	22102
2	3	2013	110	5	20017
6	4	2006	80	5	20017
2	5	2014	90	0	10007

Table 1.2: Customer Table C

CustId	*Name*
1	Linda
2	David
3	Andrew

1.2.2 Example

This section shows an example that is used to illustrate the SQL query fault localization and repair problem. Table 1.1 lists product orders purchased by customers and Table 1.2 provides additional information on the customers. The request is to retrieve orders placed after 2009 and priced greater than \$100, or orders shipped to zip code 10007 with no discount. Now assume the programmer misinterpreted this request, retrieving instead orders placed after 2007 (with price greater than \$100) plus orders shipped to zipcode 10008 (with no discount). Figure 1.1 shows the incorrect query, with comments, starting with "#", that give the correct version.

```
SELECT *
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
```

Figure 1.1: Faulty Query 1

The orders that satisfy the request are OrderId 2, 3, and 5, whereas the orders that are returned by the incorrect query are OrderId 1, 2, and 3. Table 1.3 compares these answers. An outcome P (Pass) indicates a row that was expected and retrieved (true positive) or not expected and not retrieved (true negative). An outcome F (Fail) indicates an expected row that was not retrieved (false negative) or an unexpected row that was retrieved (false positive).

Table 1.3: Answer Comparison

Orderid	Outcome	Type
1	F	false positive (superfluous)
2	P	true positive
3	P	true positive
4	P	true negative
5	F	false negative (absent)

1.2.3 The SQL Query Fault Localization Problem

Given an incorrect query as shown in Figure 1.1 and a set of test data as in Table 1.1, assume there are test oracles that can determine whether each test data passes or fails as shown in Table 1.3. The goal of the query fault localization problem is to identify the faulty clauses $Year > 2007$ and $Zipcode = 10008$.

I define the query fault localization problem formally as following.

Given the below information:

1. An incorrect query Q

2. A set of test data R with each row denoted r_i; when Q is applied to R, it determines whether the row r_i is included or excluded in the result (for example, $Q(r_i) = Included$ or $Q(r_i) = Excluded$)

3. A test oracle O that can be used to evaluate the test result of each row (for example, $O(r_i) = Passing$ or $O(r_i) = Failing$)

The fault localization technique uses the above information to localize faulty components in JOIN or WHERE clauses in query Q.

Previous research for localizing faults in data centric applications treated the entire SQL query or the WHERE predicate as a program entity. While these methods may discover that an entire predicate is faulty, they do not locate the individual clauses that are faulty. Moreover, these techniques apply statement coverage-based fault localization technique. That is, they are based on the assumption that the entities executed by more failed tests are more likely to contain faults. Although this assumption has been useful in localizing faults in general programming statements, it does NOT hold in SQL clause fault localization. Test rows are executed equally by all clauses. This research address the problem of how to automate the SQL query fault localization to identify faulty clauses effectively and efficiently. Section 1.3.1 will discuss the approaches and hypotheses for solving the problem.

1.2.4 The SQL Query Repair Problem

Automatic program repair technique attempts to fix the faulty code by generating a patch, which is a replacement for part of the program (in this case, part of the SQL query). Most automated program repair techniques are based on test suites [14], that is, a patch is considered as a “correct” fix when all test suites have passed. In this research, the query repair problem follows the same rule and tries to generate a query that is “correct” with respect to the given test suite.

However, correctness is not sufficient for evaluating the quality of a patch. Monperrus proposed an additional evaluation criterion, acceptability [14]. A patch may pass the test suite but contains nonsensical code. This patch is not acceptable because it fails to preserve the logic in the requirement.

In addition, SQL query repair uses database rows as test cases. When databases can contain large number of rows, the repair efficiency may be severely impacted.

One previous research effort used decision tree-based technique to generate patches for an SQL-like language [10]. Although a good start, this research was limited by the quality and efficiency. That research uses a decision tree learning algorithm to derive the classification rule from test data instead of fully localizing faults and repairing the faults. It generated patches that failed to reflect the correct logic, thus were not acceptable to engineers. Specifically, the patches often include undesired clauses or miss desired clauses. In addition, the decision tree algorithm is resource intensive. It took 2.6 hours to find a patch in a database of 3,039,969 rows.

This research addresses the problem of automatically repairing SQL faults. Moreover, it aims to solve the acceptability and efficiency problem.

1.3 Approaches and Hypotheses

My approach to solve the research problem includes two components: First, I propose a novel technique to accurately localize the faults, second, I use the fault localization result to automatically fix the faulty query. I introduce the two components in the following subsections.

1.3.1 SQL Query Fault Localization

The general process of localizing the faulty components in a program is to analyze the failing tests' execution traces and identify the abnormality in the execution traces. Since SQL queries consist of boolean conditions, the abnormality may be identified by examining the boolean evaluation results of the failing test cases. If I can compute the boolean evaluation result of each component for the failing tests and compare them with the expected evaluation results, I may be able to identify the faulty components.

Based on this motivation, I propose to localize faults in WHERE clauses in two steps. First, I discover suspect clauses in the predicate with *row-based dynamic slicing*, and then exonerate unjustly suspected clauses with *delta debugging*. These two techniques and how I apply them are explained briefly below.

Program slicing finds statements that are relevant to the values of given variables, deleting the parts of the program that are irrelevant to those values, and thus reducing the search domain from the entire program to a particular *program slice* [15]. Discovering slices by examining only the source code is referred to as *static slicing*, whereas discovering slices by analyzing specific execution traces of the program is referred to as *dynamic slicing*. I used row-based dynamic slicing to identify suspect clauses: I executed each test row against the query and recorded the boolean result of each individual clause. Then I counted the number of failing rows of each clause as the suspiciousness counter.

Delta debugging is a methodology for isolating failure-inducing inputs [16] to identify the clause most likely to have the fault. Inspired by this technique, I mutate column values of failing rows and replace them with the corresponding values from passing rows. If the mutated row passes, then the clause containing this column is considered to be at fault. As the result, only the clauses containing the fault-inducing columns are returned as suspicious clauses. The rest of the clauses are exonerated.

To localize JOIN clause faults, I categorize JOIN clause faults into two groups: JOIN condition faults and JOIN type faults. Both types of faults can be identified by analyzing failing test rows and the faulty queries. Essentially, JOIN conditions and JOIN types are transformed into boolean conditions and the boolean conditions are evaluated against failing rows. By analyzing whether the failing rows should or should not satisfy the boolean condition, I can determine whether the related component is faulty.

To evaluate the proposed SQL query fault localization technique, I form the following hypothesis:

SQL Query Fault Localization Hypotheses:

The SQL query fault localization technique is effective and efficient at repairing WHERE clause and JOIN clause faults.

The effectiveness is defined in terms of accuracy of finding faulty faults. The efficiency is defined in terms of execution time. The detailed metrics are described in Section 9.4. This research conducted experimental evaluations to verify the hypotheses by comparing the SQL query fault localization technique with nine well-known existing fault localization techniques. The techniques are carefully selected from three categories: theoretically proved optimal techniques [17], superior techniques in an empirical study [18], and the frequently used techniques [19,20]. The experiments are designed to address following research questions:

- RQ1: Which technique is the most effective?
- RQ2: Which technique is the most efficient?

1.3.2 SQL Query Repair

The fault localization approach proposed in Section 1.3.1 precisely identifies the faulty components. The repair step can greatly benefit from the fault localization result. As a test suite based repair technique, it will mutate the faulty components to generate a patch that does not produce any failing rows.

I use the test oracles to group test cases into two groups: rows that should be included and rows that should be excluded. Then, I compute the statistics for rows in these two groups such as the minimum value, maximum values, number of distinct values, the number of null values, etc. The statistics are used to repair the faulty components. To repair WHERE clauses, I developed a set of repair rules that target the most commonly used data types and the 11 SQL comparison operators. The statistics are checked against the repair rules to generate a patch. Similarly, to repair JOIN faults, I check the null values statistics of failing rows and the statistics of passing rows against a set of repair rules.

To evaluate the proposed SQL query fault repair technique, I form the following hypothesis:

SQL Query Repair Hypotheses:

The SQL query repair technique is effective and efficient at repairing WHERE clause and JOIN clause faults.

I compared my query repair technique with the only existing SQL query repair technique: the DT approach [10]. I evaluated effectiveness from two aspects: correctness and acceptability. The patch generated is evaluated against all test rows, if all test rows pass then the patch is considered as a *correct* repair. The patch is also compared with the correct query. The patches that are more similar to a correct query are more *acceptable*. The efficiency is defined in terms of the time spent to repair the fault. Two research questions were raised and answered in the experiment:

- RQ1 (*Effectiveness*): Is the SQL query fault repair technique more effective than the DT approach for all fault classes?
    - RQ1a (*Correctness*): Can the patches generated by the SQL query fault repair technique pass all test rows for more queries than the DT approach?
    - RQ1b (*Acceptability*): Are the patches generated by the SQL query fault repair technique more acceptable to users than the DT approach?
- RQ2 (*Efficiency*): Is the SQL query fault repair technique more efficient than the baseline in terms of execution time?

1.4 Structure of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides background on two related research topics: automatic fault localization and program repair. Because this research focuses on the JOIN and WHERE clauses in SQL queries, Chapter 2 also introduces the basic structures of JOIN and WHERE clauses with examples. Next, it discusses issues in existing techniques.

Chapter 3 reviews research papers that target automatic fault localization and program repair problems, and compares my technique with these papers.

Chapter 4, 5, 6, and 7 describes the fault localization and repairing techniques. Chapter 4 introduces the fault localization technique for WHERE predicate faults. It defines WHERE fault types, studies how to apply existing SFL techniques to WHERE faults localization, and presented my fault localization algorithms–ALTAR and ALTAR2. The main content of this chapter has been developed into two papers: one is published in the tenth International Conference on Software Testing, Verification, and Validation (ICST 2017) [21] and the other one is undergoing revision for Journal of Systems and Software [22]. After I identified faulty clauses, in Chapter 5 I present how to repair faults in WHERE predicates. Similarly, I explain JOIN clause fault localization in Chapter 6 and then demonstrate how to repair JOIN clause faults in Chapter 7. The content in Chapter 5, 6, and 7 is published in the 18th International Conference on Software Quality, and Security (QRS 2018) [23].

To evaluate the proposed techniques, I implemented them into a tool. Chapter 8 presents the implementation detail. I designed experiments to evaluate the fault localization technique and fault repair techniques separately. Chapter 9 illustrates the experiments for evaluating the fault localization techniques. It describes the experiment subjects, comparison techniques, and the procedure and metrics. Next, it shows the experimental results followed by analysis and interpretations of the result. Similarly, Chapter 10 illustrates the experiments for evaluating the fault repair techniques.

Finally, the dissertation summarizes the contributions and concludes future research directions.

Chapter 2: Backgroud

2.1 Automatic Fault Localization and Program Repair

Program repair consists of four steps: *failure detection*, *fault diagnosis*, *fault localization*, and *repair inference* [24]. A *failure* represents unexpected behavior of the software and indicates that the program contains a *fault* [25]. A failure can be revealed to the tester with proper test oracles [26]. Fault diagnosis is part of fault localization. Fault diagnosis observes internal program state and fault localization finds the exact location of the fault. Repair inference attempts to fix the fault.

With advances in automating test design, test generation, and test execution, the first step has been largely automated. However, fault diagnosis, localization, and repair are still usually done by hand. This manual process is time-consuming, labor-intensive, and technically challenging. To reduce costs, researchers have been targeting the last three steps: developing ways to automatically diagnose, locate [15, 19, 20, 30, 33], and repair software faults [5–7].

Fault localization is the process of searching the program code to find suspicious entities such as blocks, statements, and predicates that are related to the program failure. To avoid manually searching complicated and large programs, automatic fault localization (AFL) techniques are applied to analyze the program entities and eliminate "innocent" program entities. This allows programmers to only examine a refined set of "suspicious" entities. The precision of the automatic fault localization technique is an important criteria. The more precise the fault localization result is, the fewer entities need to be manually reviewed.

Automatic program repair (APR) attempts to find a fix to a program failure without human intervention. Unlike AFL, the search space of APR may be infinite. To guide the search, APR techniques usually require a report containing the suspicious program entities that may be generated by AFL or gathered from manual debugging, a test suite and test oracles to distinguish failure behaviors from expected behaviors. Manual repair is technically challenging because the engineer must deeply understands the code. It is also resource intensive because each repair attempt can take significant time and effort to fix and verify. Automated program repair, on the other hand, relies on a very limited "understanding" of the program (that is, whether a set of tests passes or not), but repair attempts are completely automated and therefore very inexpensive. Thus, automated program repair can try many more possible corrections than manual repair can, giving it the potential to save significant resources. Many APR techniques adopt a generate-validate approach: a modified version of the program is generated and is validated against the test suite and test oracle. If the patch fails then another is generated and validated. This process repeats until a passing patch is generated.

2.2 SQL Queries

Structured Query Language (SQL), the standard database language, was ranked as the most in-demand programming language in 2016 [1]. Applications use SQL to persist and retrieve data. Because SQL is a declarative language, programming and debugging are quite different from procedural programming languages.

SQL queries retrieve data from tables with clauses such as SELECT, FROM, JOIN, and WHERE. A SELECT clause specifies the columns from which the data are retrieved. A FROM clause specifies the source tables. A JOIN clause concatenates rows from multiple source tables to produce a larger table. A WHERE clause defines a predicate (condition) that the data retrieved must satisfy. This section briefly reviews the JOIN and WHERE clauses with examples.

2.2.1 The JOIN Clause

SQL uses five JOIN types to connect two tables in different ways: CROSS JOIN, INNER JOIN, FULL JOIN, LEFT JOIN, and RIGHT JOIN. A JOIN clause can connect more than two tables using different JOIN types. The left table in the JOIN keyword is referred to as LT and the right table as RT. A *condition* (starting with the ON keyword) can be used to specify how the rows in LT and RT should be matched.

CROSS JOIN is a Cartesian product of LT and RT. It combines every row of LT with every row of RT. Table 2.1 is the cross join of tables O (Table 1.1) and C (Table 1.2).

Table 2.1: CROSS JOIN Result Table

CustId	*OrderId*	*Year*	*Price*	*Discount*	*ZipCode*	*CustId*	*Name*
1	1	2008	110	0	22102	1	Linda
1	2	2014	120	10	22102	1	Linda
2	3	2013	110	5	20017	1	Linda
6	4	2006	80	5	20017	1	Linda
2	5	2014	90	0	10007	1	Linda
1	1	2008	110	0	22102	2	David
1	2	2014	120	10	22102	2	David
2	3	2013	110	5	20017	2	David
6	4	2006	80	5	20017	2	David
2	5	2014	90	0	10007	2	David
1	1	2008	110	0	22102	3	Andrew
1	2	2014	120	10	22102	3	Andrew
2	3	2013	110	5	20017	3	Andrew
6	4	2006	80	5	20017	3	Andrew
2	5	2014	90	0	10008	3	Andrew

INNER JOIN is a special case of CROSS JOIN because it specifies a JOIN condition. Only rows that satisfy the JOIN condition are retrieved. The columns used for matching are the JOIN *keys*. Figure 2.1 shows an SQL query that retrieves the customers listed in table Customer C with the orders they placed, as listed in table Order O. It combines the Order O and Customer C tables using INNER JOIN. The join condition $O.CustId = C.CustId$ specifies the orders should be matched to the customer with the same *CustId*. In this example the join keys are $O.CustId$ and $C.CustId$. The result is shown in Table 2.2.

FULL JOIN, LEFT JOIN, and RIGHT JOIN are similar to INNER JOIN except that rows of either table that do not satisfy the join condition are still preserved in the result, by "matching" them with rows of *nulls*. LEFT JOIN preserves only unmatched rows of LT, whereas RIGHT JOIN preserves only unmatched rows of RT. FULL JOIN preserves both

```
SELECT *
FROM Order O
INNER JOIN Customer C
ON O.CustId = C.CustId
```

Figure 2.1: Query with JOIN Clause

Table 2.2: INNER JOIN Result Table

CustId	*OrderId*	*Year*	*Price*	*Discount*	*ZipCode*	*CustId*	*Name*
1	1	2008	110	0	22102	1	Linda
1	2	2014	120	10	22102	1	Linda
2	3	2013	110	5	20017	2	David
2	5	2014	90	0	10008	2	David

unmatched rows of LT and RT. Table 2.3 shows the result of a full join. The tables for LEFT and RIGHT JOIN are similar: The LEFT JOIN result would exclude the last row of Table 2.3 (an unmatched row of RT) and the RIGHT JOIN result would exclude the fourth row (an unmatched row of LT).

Table 2.3: FULL JOIN Result Table

CustId	*OrderId*	*Year*	*Price*	*Discount*	*ZipCode*	*CustId*	*Name*
1	1	2008	110	0	22102	1	Linda
1	2	2014	120	10	22102	1	Linda
2	3	2013	110	5	20017	2	David
6	4	2006	80	5	20017	null	null
2	5	2014	90	0	10008	2	David
null	null	null	null	null	null	3	Andrew

2.2.2 The WHERE Clause

WHERE clauses consist of the WHERE keyword followed by a Boolean predicate. This predicate connects Boolean clauses with OR, AND, or NOT. Each clause is of the form *column opr constant* or *column opr column*, where *opr* is a comparator.

Figure 2.2 shows a query using WHERE clause to retrieve orders placed after 2009 and price greater than \$100, or orders shipped to zip code 10008 with no discount. Table 2.4 shows the query result.

```
SELECT  *
FROM Order O
WHERE (Year>2009 AND Price>100)
    OR (Zipcode=10008 AND Discount=0)
```

Figure 2.2: Query with WHERE Clause

Table 2.4: WHERE Query Result Table

CustId	*OrderId*	*Year*	*Price*	*Discount*	*ZipCode*
1	2	2014	120	10	22102
2	3	2013	110	5	20017
2	5	2014	90	0	10008

I transform the predicate to a *monotone* Boolean function by eliminating NOT operators. For example, NOT ($Year > 2008$) is transformed to $Year \leq 2007$, then to *disjunctive normal form (DNF)*, which is a disjunction of conjunctive clauses. The DNF can be further reduced to an equivalent *irredundant disjunctive normal form* (IDNF), which is unique for monotone Boolean functions [27]. The final predicate consists of *conjunctive predicates* (CP). A CP is a list of clauses connected by AND operators. The last two lines in Figure 2.2 shows an IDNF predicate with two CPs, each of which contains two clauses, CP1 ($Year > 2007$ AND $Price > 100$) and CP2 ($ZipCode = 10008$ AND $Discount = 0$).

2.3 Issues with Existing Techniques

2.3.1 Issues with Existing SQL Query Fault Localization Techniques

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. For example, Nguyen et al. [12] located faults in WHERE predicates. While these methods could discover that an entire predicate is faulty, they do not locate the individual clauses that are faulty.

Moreover, the approach used by Nguyen et al. [12] and other techniques [11, 13] are based on statement coverage. Specifically, they are based on the assumption that entities executed by more failed tests are more likely to contain faults. Although this assumption has been useful in localizing faults in general programming statements, it does NOT hold in SQL clause fault localization. Test rows are executed equally by all clauses.

I illustrate the limitations of the existing techniques for finding faults in clauses with a small example. Nguyen et al. [12] adopted the Tarantula [19] metrics in their fault localization technique. The Tarantula metric is shown in Equation 2.1. It calculates a suspiciousness score $S(c)$ of a program entity c by combining two proportions: (1) The number of test cases for which the program behaved correctly and in which the particular statement was executed ($Passed(c)$), relative to the *total* number of test cases for which the program behaved correctly (*TotalPassed*); and (2) the number of test cases for which the program behaved incorrectly and in which the particular statement was executed ($Failed(c)$), relative to the *total* number of test cases for which the program behaved incorrectly (*TotalFailed*). In my research, the WHERE predicate is the program, each clause is a program entity, and database rows serve as test cases.

$$S(c) = \frac{\frac{Failed(c)}{TotalFailed}}{\frac{Failed(c)}{TotalFailed} + \frac{Passed(c)}{TotalPassed}} \tag{2.1}$$

Following the approach used by Nguyen et al. [12], I separate each clause into a "true case" and a "false case" and associate a suspiciousness counter with each. The suspiciousness score of each part indicates the likelihood of an error when the clause evaluates to *true* or to *false*. Table 2.5 illustrates the computation of these suspiciousness scores. The column *Clause* lists the four clauses and their *true* and *false* evaluations. The column *Individual Row* shows the five rows, identified by *Orderid*. For each row, if a clause is evaluated to *true*, then the *true* position is checked; otherwise, the *false* position is checked. The final row indicates the test result for each row. The column *Suspiciousness Score* shows the suspiciousness score of each clause.

As an example, consider the *true* evaluations of the clause $Year > 2007$. *TotalPassed* = 3 (orders 2, 3 and 4) and *TotalFailed* = 2 (orders 1 and 5). From the three rows that passed, two were executed (2 and 3), and from the two rows that failed, two were executed (1 and 5). Altogether the suspiciousness score is $\frac{2}{2}/(\frac{2}{2}+\frac{2}{3}) = 0.6$.

Table 2.5: Suspiciousness Score Computation

Clause	Individual Row					Suspiciousness
	1	2	3	4	5	Score
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 Result	F	P	P	P	F	

This attempt is clearly flawed. The suspiciousness scores calculated for *true* and *false* evaluations are not meaningful. Even if the suspiciousness of a clause is presented with the average of its *true* and *false* scores, the results still fail to correctly prioritize the clauses by their suspiciousness. For example, the incorrect clauses $Year > 2007$ and $Zipcode = 10008$ receive suspiciousness scores $(0.6 + 0)/2 = 0.3$ and $(0 + 0.5)/2 = 0.25$, which are lower than the scores for the correct clauses $Price > 100$ and $Discount = 0$, which are $(0.42 + 0.6)/2 = 0.51$ and $(1 + 0)/2 = 0.5$. The Tarantula metric is based on the premise that a statement that is executed by more failed tests than passed tests is more likely to contain faults. However, in this case all clauses in a WHERE predicate are evaluated for each test row.

The example demonstrates that the Tarantula metric is not effective for localizing faults in SQL predicates. To address this issue, I propose a novel SQL fault localization technique and describe it in Chapters 4 and 6

2.3.2 Issues with Existing SQL Query Repair Techniques

Only one paper [10] presented results on automatic repair in data-centric applications with an SQL-like language, called ABAP. This research uses a decision tree learning algorithm to generate a completely new query rather than fixing the faulty components. Although a start, the generated patches are limited by quality and efficiency.

The decision tree learning algorithm is widely used in the knowledge discovery field. It learns from the labeled data, and induces a classification function that can correctly split the data by their labels. Gopinath et al. [10] apply the decision tree algorithm to automatically repair WHERE predicate faults in an SQL-like language. I call this "the DT approach." The DT approach adopts a decision tree algorithm called the ID3 [28] algorithm. With the test oracle, test rows are labeled as included or excluded to indicate whether the row should be included by the correct query. The learning process works as follows. Columns are used to generate classification rules that predicts rows into included and excluded groups. ID3 picks a column with the highest predictiveness accuracy. If any predicted group is not accurate (for example, an included row is predicted to be in excluded group), another column will be chosen to further separate this group. The process is recursively executed until either all predicted groups are accurate or all columns have been used.

I illustrate the DT approach with an example below showing that it does not analyze the faulty query but relies only on test rows and oracles. Table 2.6 shows a labeled data set with five rows. The first and second columns are two attributes, and the last column labels each row with "included" (+) or "excluded" (−). Based on the attribute values in Table 2.6, the DT approach derives classification rules to split the rows into "included" and "excluded" subsets as in Figure 2.3. Non-leaf nodes are attributes (columns), branches are classification rules, and leaf nodes are the labels of the split subsets. Combining the branches and non-leaf nodes leading to the "included" node results in the patch $Year > 2007$ AND $Price > 100$.

Table 2.6: Labeled Data Set

Year	*Price*	*Label*
2008	110	+
2014	120	+
2007	110	-
2005	80	-
2012	100	-

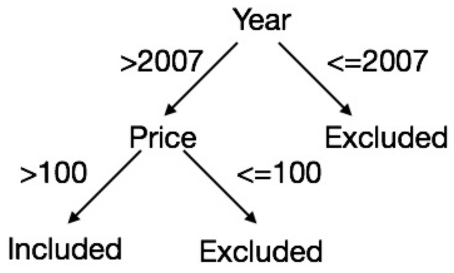

Figure 2.3: Decision Tree Example

This technique has two major problems. First, the generated patch could be very different from the desired correct query, that is, the patch has low acceptability. Note that the DT approach is independent of the original query. Rather than identifying fault types and localizing faults in the original query, the technique directly derives a new query from the labeled data. This means, for a given set of labeled data, the DT approach always generates the same patch. However, distinct queries can produce the same labeled data. For example, a query with the condition $Year\ IN\ (2008, 2014)$ produces the same data in Table 2.6 as the query $Year > 2007$ AND $Price > 100$. Assume the DT approach generates a patch $Year > 2007$ AND $Price > 100$ when the correct query is $Year\ IN\ (2008, 2014)$. The patch cannot be accepted as it does not correctly fix the fault.

The "overfitting" problem is another cause for low acceptability. The DT approach may attempt to generate the classification rule that fits the given labeled data tightly, but is less general. For example, it may use the most selective attribute to generate a tree with many leaves; or it may use many unnecessary clauses to generate a deep tree with many levels. The overfitting problem may result in a "nonsensical" patch.

The second major problem with the DT approach is that it is computationally intensive. It enumerates all attributes and analyzes all possible splitting values to find the best classification rule. When applying DT as part of my experiment, I observed that it can take over 50 minutes to generate a patch on a table with 5 columns and 127,348 rows.

To address these two problems, I propose a novel approach that can efficiently and precisely fix the faults. These are described in Chapters 5 and 7.

Chapter 3: Related Work

In this chapter, I review related work in two areas: fault localization techniques and automatic program repair techniques.

3.1 Fault Localization

A recent survey [29] classified fault localization techniques in eight categories:

1) Slice-based	5) Machine learning-based
2) Spectrum-based	6) Data mining-based
3) Statistics-based	7) Model-based
4) Program state-based	8) Miscellaneous

My fault localization method is related to the slice-based, spectrum-based, and program state-based categories, which are discussed below.

3.1.1 Slice-based Fault Localization

Program slicing was initially introduced by Weiser [15]. It finds statements that are relevant to the values of given variables, deleting the parts of the program that are irrelevant to those values, and thus reducing the search domain from the entire program to a particular *program slice*. Discovering slices by examining the source code only is referred to as *static slicing*, whereas discovering slices by analyzing specific execution traces of the program is referred to as *dynamic slicing*. The execution slicing is a variation of dynamic slicing. It counts the statements actually executed without consideration of a specification. Agrawal et al. [30] presented an execution slicing approach for C programs. It constructs slices based on the executions of failed and successful test cases. My fault localization method identifies program slices (suspect clauses), with a specification (evaluations of clauses over rows). Thus, my method is in the dynamic slice-based category.

3.1.2 Spectrum-based Fault Localization

A program entity is a part of a program. It can be at different granularities, such as a statement, a block, a predicate, or a clause. A *program spectrum* is the detailed execution information of a program entity [31]. For example, it can be the coverage of a statement or the boolean evaluation of a predicate. Spectrum-based fault localization (SFL) techniques use the program spectrum information to analyze whether the associated entities are more likely to be faulty. These techniques have been studied extensively because they have a lower overhead comparing to other techniques. According to Souza et al. [32], there are two major categories of SFL techniques: similarity-based and statistic-based. *Similarity-based* techniques use the coefficient formulas while *statistic-based* techniques applies statistical models. In fact, similarity-based and statistics-based techniques are very similar. The major difference is the suspiciousness formulas used. They use the suspiciousness formulas to calculate the suspiciousness score for each program entity and then rank them by suspiciousness score. The key difference between my method and other spectrum-based methods is that mine finds suspect clauses and exonerates innocent clauses, instead of calculating rankings for all the clauses. The exoneration is especially useful when multiple faults exist. In the ranking approach, developers must fix the top ranked fault, then execute the fault localization technique again to obtain a new suspiciousness ranking, then fix the new top ranked fault. This process repeats until all faults are correctly fixed. In contrast, my approach directly locates all the suspicious clauses, which enables developers to fix all faults at once. I studied nine SFL techniques [19,20,33–38] and will present how to apply them in SQL fault localization in Section 4.2. I also compared my fault localization technique with the nine techniques through comprehensive experiments in Chapter 9.

3.1.3 Program State-based Fault Localization

Program state-based techniques identify suspect statements by investigating program states. One program state-based technique is *delta debugging* [16], in which failure-inducing inputs are identified by comparing program states of a passing test execution with that of a failing test execution. Delta debugging has been used in combination with dynamic slicing [39] and the cause transition method [40]. Jeffery et al. [41] and Zhang et al. [42] applied delta debugging to general program statements and predicates, respectively. My method uses delta debugging differently in three aspects. First, I alter the column values in failed rows, whereas Jeffery et al.'s technique [41] replaces the values of program variables and Zhang et al.'s technique [42] changes predicates rather than values. Second, to look for faults that have missing columns and composite faults, my method replaces the values of different combinations of columns, whereas their methods do not identify specific types of faults or replace values with different combinations. Third, I exonerate innocent clauses that are not related to the fault-inducing columns, whereas their methods calculate suspiciousness rankings for all statements.

3.1.4 Miscellaneous

In addition to the above three fault localization techniques, my approach is also related to mutation-based fault localization and database application fault localization. Unlike the other fault localization technique, research works in these two categories are relatively new and thin.

Mutation-based Fault Localization: Spectrum-based fault localization relies heavily on the availability of program spectrum data. When spectrum data are limited, the accuracy of the statistical model could be badly skewed. MUtation-baSEd (MUSE) [43] attempts to alleviate this problem by mutating the Program Under Test (PUT). The intuition is that mutating an already faulty statement is more likely to make failed tests pass than mutating a correct statement. Thus, by evaluating the failing and passing test cases before and after mutating PUT, MUSE is able to identify faulty statements. My approach differs from MUSE in that I do not mutate PUT (in my case PUT is the SQL predicate), but rather, I mutate the test data in failing test cases.

Database Application Fault Localization: Clark et al. [11], Nguyen et al. [12], and Saha et al. [13] apply fault localization techniques to database and data-centric applications. Clark et al.'s technique [11] counts executed embedded SQL and regular statements for test rows and computes suspiciousness scores. It studied applications in which SQL queries could be constructed dynamically, so a query could have different SQL statements. The technique proposed by Nguyen et al. [12] not only captures executions of SQL statements, but also evaluates the result of predicates of WHERE clauses using row-based slicing method. It treats an entire predicate as one program entity, whereas I consider each clause in the predicate as one entity. Saha et al. [13] apply a field-row sensitive slicing algorithm to generate execution traces. A key-based slicing approach is used to further remove irrelevant statements. Then, it computes sequential and semantic differences between correct and incorrect slices to identify faulty statements. Overall, none of these methods can be used to localize faults in individual components (clauses) in SQL predicates. They are based on the coverage assumption, that is, a program entity executed by more failing test cases than passing test cases are more likely to be faulty. However, this assumption does not hold in SQL fault localization, because all clauses are equally executed by all tests. Moreover, these methods can only reveal the existence of a fault, whereas my method can further identify specific types of faults.

It is worth mentioning that detecting faults in queries and suggesting possible corrections has also been studied in the database community. The detection is often done automatically at query time (that is, there is no testing phase): An error is suspected when a query returns an empty answer or an answer that is unusually small. It is then followed up automatically with additional queries in an attempt to pinpoint erroneous presuppositions of the programmer regarding the database structure and the stored data [44, 45].

As described in mutation analysis, a test set is *adequate* if it can cause all faulty versions of the program to fail [46]. Test set adequacy is crucial to my fault localization technique. In my experiment, I used large databases with millions of rows and manually created additional data sets to improve test case adequacy. This process can be improved by using automatic database test case generation techniques. For example, Vemasani et al. [47] proposed a sound and complete test database generation algorithm to distinguish conjunctive queries with equalities. Li et al. [48] suggested an approach to generate test data for data-centric applications from the original database and the requirements. These techniques can generate small yet effective test data sets, which can further improve the efficiency and effectiveness of the fault localization technique.

3.2 Automatic Program Repair

The most closely related work to my work in automatic program repair was Gopinath et al.'s DT approach [10], which I will quantitatively compare with my query repair technique in Chapter 10. Here I present qualitative comparisons with that work and others.

3.2.1 Fault Classes

There are five basic components in SQL queries: the SELECT clause defines the columns to be retrieved; the FROM clause specifies the source tables; the JOIN clause concatenates rows from the source tables to produce a larger table; the WHERE clause defines a predicate (condition) that the data retrieved must satisfy; and the GROUP-BY clause aggregates the rows. I classify five fault classes based on each of the five components.

As the most fundamental element in an SQL query, faults in FROM clause are trivial to identify and rarely committed into source code repositories. I investigated 116 real SQL faults from industry applications, and found no faulty FROM clause. Thus I do not consider faults in FROM clauses. SELECT clause faults indicate columns are incorrectly retrieved. Repairing faults in SELECT clause can benefit from Brodsky et al.'s research work on regression database [49]. They proposed to augment traditional relational database tables with an unknown attribute: the *learned attributes*. The linear least squares regression technique is used to derive the relationship between the learned attribute and the existing attributes. This technique can be applied to fix incorrectly retrieved columns in SELECT clauses. Gopinath et al. investigated repairing faults in GROUP-BY clauses [10]. The aggregation functions combine multiple rows and compute a single value such as average, sum, or count. As the test oracle only specifies the expected value of the aggregated result, it is difficult to identify the failing rows. Gopinath et al. proposed a novel approach to address this issue with a semi-supervised learning algorithm for inducing the failing rows from the aggregation result.

To complement the regression database approach and DT approach, I propose a novel fault localization and repairing approach focusing on JOIN faults and WHERE clause faults. The DT approach claims that it can repair JOIN condition faults and WHERE clause faults. However, my experiments showed that it is only feasible for small databases and the generated patches are not acceptable to developers. My approach is more efficient than the DT approach and the generated patch queries are highly effective and efficient.

3.2.2 Fault Localization

It is common for automatic program repair techniques to adopt fault localization methods to minimize the repair. Genprog [5] calculates a weighted path to narrow down the repair search space to code segments visited in failing test cases. SemFix [7] relies on a ranked suspicious report with the Tarantula [19] fault localization technique. BugFix [6] uses the Value Replacement-based fault localization technique [41].

I apply the exoneration-based fault localization technique to individual components in SQL query. It is more precise than the previous ranking-based fault localization techniques [19, 20], which rank clauses by the likelihood of containing fault.

The DT approach does not include a fault localization step, but rather directly derives a new patch by learning from the test data. Although the new patch may satisfy the test data, it has low acceptability because the change often fails to meet the requirements.

3.2.3 Fault Repair

Many test suite-based behavior repair techniques, including GenProg [5] and Debroy and Wong's [8], use search and mutation-based approaches. Template based approaches [6, 9] learn from previous fault fixes, and leverage the learned patterns to generate new patches.

Gopinath et al. [10] analyzed the search and mutation-based approach and concluded that it is less efficient than the DT approach. Therefore, I did not compare the search and mutation-based approach in my experiments.

The *DT* approach is the only prior research that repairs SQL faults. The DT approach relies only on test data and test oracles. Instead of repairing the faulty elements, it generates a new query that satisfies the test oracles. My repair technique is built on the fault localization result. It investigates the test data distribution statistics and failing row types to generate patches with only minor changes.

3.2.4 Evaluation Criteria

Gopinath et al. evaluated their approach on seven queries with 274 to 90,346 test rows. My experiments were evaluated on 825 queries with different complexities and fault classes, and the test rows ranged from 211,681 to almost four million in a database. Moreover, Gopinath et al. only used correctness and efficiency as the metrics, while I also evaluate acceptability. It is a known issue that automatically generated patches may contain "nonsensical" code [9, 24]. Thus, it is important to assess the patch quality beyond the correctness. Kim et al. [9] related the concept of patch acceptability to human understandability and preference. They evaluated the acceptability by surveying people who are not original developers. Monperrus [24] argued that this evaluation approach is not reliable due to the reviewers' lack of domain knowledge. I avoid this issue by obtaining the correct SQL query, and using a similarity formula to measure the difference between the correct query and the patch.

Chapter 4: Localizing WHERE Predicate Faults

4.1 WHERE Fault Types

To determine that a WHERE predicate is faulty, I need at least one failing test case that has an oracle. The test data is a set of database rows R, with each row denoted r_i. The test oracle evaluates whether a test row should be included in the result. A test oracle can be the SQL query asserts to be true if a given row should be included. Given the test data and oracles, rows can be divided 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.*

I 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 and only three components: constants, operators, and columns. E1 through E3 represent faults in each of the three components, which cover all possibilities that a component is wrong. E4 and E5 are faults where the entire clause is missing or unnecessary, which cover the faults related to existence of a clause. These five fault types also apply to different elements, so do not overlap (thus are *disjoint*). E6 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).

4.2 Applying Existing SFL Techniques

To the best of my knowledge, there are no previous research papers on localizing faulty clauses in WHERE predicates. A few papers have attempted to localize faults in database applications using spectrum-based fault localization (SFL) techniques [11–13]. Unlike my approach, which considers the clauses as program entities, they treat the entire SQL statement or the entire WHERE predicate as a program entity. To investigate the effectiveness of these SFL techniques when applied to clauses, I studied nine well-known techniques (Tarantula [19], Ochiai [20], Naish2 [33], Wong1 [34], Kulczynski2 [33], Crosstab [35], Liblit [36], SOBER [37], and Mann-Whitney [38]) and how they can be applied at clause level. I adopt the categorization method by Souza et al. [32] and discuss them in two groups: similarity-based and statistic-based. 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.

4.2.1 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 variable x and y. It can also be extended to indicate the *similarity* between variables. Similarity-based techniques use coefficient formulas to distinguish faulty program entities from correct program entities.

My study investigated five similarity-based techniques from two previous research papers [17,18]. In a theoretical study, Xie et al. [17] 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, I selected a representative from each group: Naish2 from ER1 and Wong1 from ER5. To these I added Tarantula [19] and Ochiai [20]. In an empirical study, Le et al. [18] compared the five theoretically superior techniques with Tarantula and Ochiai, finding that Ochiai was the most effective. I also added Kulczynski2 [33], 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 4.1. They are ordered according to the effectiveness claimed by Xie et al. [17]: 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. [12] applied Tarantula to SQL predicates and calculated suspiciousness for true and false evaluations separately. I adopted the same methodology and applied these techniques to clauses. For a clause c, I compute the suspiciousness score's true evaluation

Table 4.1: 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	$S(c) = \frac{c_{ef}/T_f}{c_{ef}/T_f+c_{ep}/T_p}$

$S_t(c)$ and false evaluation $S_f(c)$. $S_t(c)$ represents the suspiciousness of a clause when the clause is evaluated to true. Thus, in $S_t(c)$, a clause is deemed to be "related" 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. Similarly, in $S_f(c)$, a clause is deemed to be "related" 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. I then calculate the final suspiciousness score as the sum $S(c) = S_t(c) + S_f(c)$.

Section 1.2.3 demonstrated how to apply Tarantula in locating faulty clauses. The other methods are similar. The only difference is the suspiciousness formulas used.

4.2.2 Statistics-based SFL

Statistics-based fault localization applies statistical models to spectrum data and generates suspiciousness rankings. I studied four statistics-based techniques, Crosstab [35], Liblit [36], SOBER [37], and Mann-Whitney [38]. Crosstab was originally applied to statements. As explained in Section 4.2.1, 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 operators, 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 [35,38], the predicates were ranked first, and the corresponding statements in top ranked predicates were considered to be suspicious.

Crosstab

Wong et al. [35] 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 Equation 4.1, 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 $E_{np} = \frac{((T_f-c_{ef})+(T_p-c_{ep}))*T_p}{T_p+T_f}$.

$$\begin{aligned}\chi^2(c) = \ & (c_{ef} - E_{ef})^2/E_{ef} + (c_{ep} - E_{ep})^2/E_{ep} \\ & + (T_p - c_{ep} - E_{np})^2/E_{np} + (T_f - c_{ef} - E_{nf})^2/E_{nf}\end{aligned} \tag{4.1}$$

$\chi^2(c)$ is then compared with the chi-square critical value χ^2 found in the 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 Equation 4.2, is then used to compute the suspiciousness $\zeta(c)$ in Equation 4.3.

$$\varphi(c) = \frac{c_{ef}/T_f}{c_{ep}/T_p} \tag{4.2}$$

$$\zeta(c) = \begin{cases} \chi^2(c)/(T_f + T_p) & \text{if } \varphi(c) > 1 \\ 0 & \text{if } \varphi(c) = 1 \\ -\chi^2(c)/(T_f + T_p) & \text{if } \varphi(c) < 1 \end{cases} \tag{4.3}$$

Applying Crosstab to clauses is similar to similarity-based techniques. I calculate suspiciousness scores for true and false evaluations and get the final suspiciousness by summing them: $\zeta(c) = \zeta^t(c) + \zeta^t(c)$.

Liblit

Liblit [36] 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 Equation 4.6, between the probability of how likely p can fail tests, $Context(p)$ in Equation 4.4, and the probability of how likely p can fail tests when evaluated to true, $Failure(p)$ in Equation 4.5. The predicate with the larger difference is more suspicious. When applying Liblit to clause c, the definition of $Increase(c)$ remains the same as $Increase(p)$.

$$Context(p) = Pr(Crash|\, p \; observed) \tag{4.4}$$

$$Failure(p) = Pr(Crash|\, p \; observed \; \mathbf{true}) \tag{4.5}$$

$$Increase(p) = Failure(p) - Context(p) \tag{4.6}$$

SOBER

SOBER [37] 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*

Table 4.2: 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

over a set of test executions. The evaluation bias, $\pi(p)$, is given by Equation 4.7, 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. Equation 4.8 calculates the similarity $L(P)$ between $f(X|\theta_p)$ and $f(X|\theta_f)$ (Sim), and Equation 4.9 computes the suspiciousness score $S(P)$ from $L(P)$.

$$\pi(p) = \frac{n_t}{n_t + n_f} \tag{4.7}$$

$$L(P) = Sim(f(X|\theta_p), f(X|\theta_f)) \tag{4.8}$$

$$S(P) = -log(L(P) \tag{4.9}$$

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 Figure 1.1 is shown in Table 4.2. For clause $Year > 2007$, the passing test evaluation bias set is {1,1,0} and the failing test evaluation bias set is {1,1}. 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 assuming normal distribution.

Mann-Whitney

Zhang et al. [38] 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, I 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[1]. Mapping the rank-values back to the evaluation bias sets V_p and V_f, I 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 Equation 4.10. The larger $R(p)$ indicates the predicate p is more suspicious.

$$R(p) = -min(K_l/K, K_h/K) \tag{4.10}$$

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.

[1]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 $2 + 1/4 = 2.25$.

4.3 ALTAR Algorithm

Unlike previous SFL techniques, which were applied to general programs, I designed an exoneration-based technique that specifically targets faulty clauses in SQL predicates. I implemented this approach in a tool, named *Automated sqL predicaTe fAult localizeR* (ALTAR). I will use ALTAR to refer to this exoneration-based technique.

4.3.1 Basic Approach

This section reviews the basic approach. It consists of two steps, creating slices of suspect clauses in failing rows and exonerating innocent clauses from the suspect clauses.

Slicing

A *slice* refers to a set of program entities that are relevant to some computed values such as test results [15]. 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 *suspect clause*.

Specifically, clauses are sliced as follows. ALTAR checks each failing row to determine if it is superfluous (R_s) or absent (R_a). 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 the failing clauses are suspect.

I illustrate finding suspect clauses with the running example from Figure 1.1 and the incorrect predicate $((Year > 2007) \wedge (Price > 100)) \vee ((ZipCode = 10008) \wedge (Discount = 0))$. I 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 $Orderid$ = 1 in Table 1.1 was

Table 4.3: Finding Suspicious Clauses with Slicing

Clause		Row Eval. 1 (R_s)	5 (R_a)	Suspiciousness 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

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 4.3 shows the clauses, the evaluation results, and their suspiciousness counters.

Exoneration

The row-based slicing technique reduces the search domain from all clauses to a set of suspect clauses. However, some of the suspect clauses identified in the slicing step may be innocent. For example, the innocent clause C_2 is identified as a suspect clause in Table 4.3. The goal of exoneration is to remove innocent clauses. The exoneration technique of ALTAR is inspired by delta debugging [16]. ALTAR replaces column values of a failing row with corresponding values from a passing row. In other words, ALTAR *mutates* [25] the failing row, resulting in a *mutant*. The passing row used in the mutant is called a *replacement row*. ALTAR then checks whether the mutant becomes a passing row. Specifically, for a superfluous row (R_s), the goal is to find a mutant that is correctly included (R_i); for an absent row (R_a), the goal is to find a mutant that is correctly excluded (R_e). 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 is 0, this clause becomes innocent.

Table 4.4: Superfluous Row Mutants

Type	*Orderid*	*Year*	*Price*	*Discount*	*Zipcode*	Group
Original	1	2008	110	0	22102	R_s
Replacement	2	2014	120	10	22102	R_i
Mutant	1	**2014**	110	0	22102	R_i
Mutant	1	2008	**120**	0	22102	R_s

I demonstrate the process of exonerating superfluous rows with an example. Returning to our running example from Figure 1.1, 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, I choose the row $Orderid = 2$ from group R_i as a replacement. First, I mutate the column *Year* from C_1 by substituting the value of *Year* from the replacement row. If the mutated row belongs to group R_i, then C_1 is the correct suspect and C_2 is exonerated. Otherwise, I 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 4.4 shows the original and mutated rows. 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, I 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, I choose the row $Orderid = 4$ from group R_e as a replacement. First, I mutate the column *Price* from C_2 by substituting the value of *Price* from the replacement row. If the mutated row is excluded (R_e), then C_2 is suspect and C_3 is exonerated. Otherwise, I 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 4.5 shows the original, replacement, and mutated rows. Since the mutation on C_3 is placed in R_e, I 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

Table 4.5: Absent Row Mutants

Type	*Orderid*	*Year*	*Price*	*Discount*	*Zipcode*	Group
Original	5	2014	90	0	10007	R_a
Replacement	4	2006	80	5	20017	R_e
Mutant	5	2014	**80**	0	10007	R_a
Mutant	5	2014	90	0	**20017**	R_e

(the counters for C_2 and C_4 are zero). The faulty clauses have been identified accurately.

4.3.2 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 suspect clauses when multiple fault-inducing columns are associated with the same failing row. I explain why with the incorrect SQL query in Figure 4.1 . Assume there is a row ($Orderid = 6$) shown in the first row of Table 4.6. 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 ($Year > 2010$), ($Zipcode = 10008$), and ($Discount > 10$) are suspicious, since they evaluate to false for the $Orderid = 6$ row. To exonerate innocent clauses with the basic approach, I create mutant rows (rows 3–5 in Table 4.6) by replacing column values with those from the replacement row (row 2 in Table 4.6). 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, I must mutate multiple columns at the same time. In Table 4.6, 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

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

Figure 4.1: Faulty Query 2

Table 4.6: Absent Rows Mutation with Multiple Fault-inducing Columns

Row#	Type	*Orderid*	*Year*	*Discount*	*Zipcode*	Group
1	Original	6	2010	0	10007	4
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

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, I do not exhaust all combinations. Instead, I stop when the minimum set of fault-inducing columns is found. To find k fault-inducing columns associated with a failed row, I need to check a total of $\sum_{m=1}^{k}(\binom{n}{1} + \binom{n}{2}... + \binom{n}{m})$ 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, I 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, I 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, I present the advanced approach for exonerating suspect 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, I first mutate columns in the existing clauses. For one CP, I 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. I stop and exonerate suspicious clauses that do not contain those fault-inducing columns by decreasing their suspiciousness counter by one. I 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, I 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. I 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.

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, $T$.
1:  for each $cp_i \in CPS$ do
2:      $COL$ = all columns included in $cp_i$
3:      for $k = 1...$ size of $COL$ do
4:          Create mutants, $MUT_k$, by replacing values of k-combinations of $COL$ on $s\_row$ with $r\_row$
5:          for each $mut_j \in MUT_k$ do
6:              if $mut_j \in R_i$ then
7:                  mark the mutated columns in $mut_j$ as fault-inducing
8:                  jump to next $cp_i$
9:      $COL\_T$ = all columns in $T$
10:     for $k = 1...$ size of $COL\_T$ do
11:         Create mutants, $MUT\_ALL_k$, by replacing values of k-combinations of $COL\_T$ on $s\_row$ with $r\_row$
12:         for each $mut_j \in MUT\_ALL_k - MUT_k$ do
13:             if $mut_j \in R_i$ then
14:                 mark the mutated columns in $mut_j$ as fault-inducing
15:                 jump to next $cp_i$
```

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

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, $T$.
 1: for $k = 1...$ size of CPS do
 2:     Create mutants, $MUT$, by replacing values of $k$-combination of $CPS$ on $a\_row$ with
        $r\_row$
 3:     for each $mut_j \in MUT_k$ do
 4:         if $mut_j \in R_e$ then
 5:             mark the mutated columns in $mut_j$ as fault-inducing
 6:             stop and exit
 7: $COL\_T$ = all columns in $T$
 8: for $k = 1...$ size of $COL\_T$ do
 9:     Create mutants, $MUT\_ALL_k$, by replacing values of $k$-combination of $COL\_T$ on
        $s\_row$ with $r\_row$
10:     for each $mut_j \in MUT\_ALL_k - MUT_k$ do
11:         if $mut_j \in R_e$ then
12:             mark the mutated columns in $mut_j$ as fault-inducing
13:             stop and exit
```

and k varies from 1 to n. In each mutant, I 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. I 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. I 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. I 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.

4.4 ALTAR2 Algorithm

Section 4.4.1 describes the efficiency problem that ALTAR encountered and Section 4.4.2 presents a significantly more efficient algorithm.

4.4.1 Efficiency Problem

The existing SFL techniques described in Sections 4.2.1 and 4.2.2 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 ALTAR algorithm can be extremely inefficient. I found that in an extreme case ALTAR took up to 25 minutes to localize faults with 234,244 failing tests [21]. In contrast, Tarantula used around 30 seconds, although it was much less effective.

I use five variables to study the complexity of ALTAR:

1. Number of columns in all the query tables (c)
2. Number of suspect CPs (b)
3. Number of suspect 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, I 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 in the number of columns clearly makes ALTAR impractical.

4.4.2 Redundant Test Case Elimination

From the above analysis I learned that the complexity of Algorithms 1 and 2 are 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 size of the test database. 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, I 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. I 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

I 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 suspect 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.

I illustrate the algorithm with the faulty SQL in Figure 1.1. Assume a new Order table with four rows, as shown in Table 4.7. The last two columns in Table 4.7 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 1.1. The other two failing rows have Orderids 6 and 7.

First, I 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 one cluster, and Orderid 5 and 7 should be grouped into another. Next, I pick Orderid 1 from the first cluster and execute the exoneration process to identify its fault-inducing column. The exoneration process described in Table 4.4 shows that I mutated the row with Orderid 1 twice and found the fault-inducing column, $Year$. I then use the fault-inducing column to mutate the other rows in the same cluster (Orderid 6). Table 4.8 shows the mutated rows. Since the mutated rows for Orderid 6 is in group R_i (passing), S_3 is satisfied. Therefore, I can conclude that row Orderid 6 is equivalent to row Orderid 1 and should be eliminated. Similarly, I 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 4.5. Then, I mutate the *Zipcode* column in row Orderid 7. The mutated row is in group R_e (passing), thus S_3 is

Table 4.7: New Order Table

Orderid	*Year*	*Price*	*Discount*	*ZipCode*	Group	Suspicious Clauses
1	2008	110	0	22102	R_s	$C1$, $C2$
5	2014	90	0	10007	R_a	$C2$, $C3$
6	2008	120	0	22105	R_s	$C1$, $C2$
7	2006	135	0	10008	R_a	$C2$, $C3$

Table 4.8: Mutated Rows

Orderid	*Year*	*Price*	*Discount*	*ZipCode*	Group
1	**2014**	110	0	22102	R_i
5	2014	90	0	**20017**	R_e
6	**2014**	120	0	22105	R_i
7	2006	135	0	**20017**	R_e

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 the 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 the example, rows Orderid 6 and Orderid 7 are mutated only once. This means ALTAR2 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 tests in the experimental study (Section 9.6), ALTAR took 30 minutes, while ALTAR2 completed in less than 10 seconds.

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 suspect 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 suspect 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 $c$
```

Chapter 5: Repairing WHERE Predicates

The fault localization technique introduced in Chapter 4 identifies the fault inducing column by mutating the value of fault inducing column in failing rows. When the value of the fault inducing column is mutated, the failing row becomes a passing row. It indicates the column must be used in the clause. Otherwise mutating the value of the column would not have changed the row from failing to passing. The repair approach leverages the fault localization results to generate patches. So the ideas is to create a new clause with the fault inducing column to *replace* the faulty clause. Assume the format of a faulty clause is $Column\ Operator\ Constant$, where $Column$ is the identified fault inducing column. The interpretation is $Operator$ and $Constant$ could be faulty, whereas $Column$, the fault inducing column, should be used in the correct clause. Thus, it focuses on replacing $Operator$ and $Constant$.

The algorithm replaces the old clause with a new clause in two steps. First, it finds data sets (INC and EXC) that are relevant to the faulty clause (i.e., influenced by the clause) and computes statistics (e.g., min and max) on those data sets. Second, the algorithm evaluates the relevant data sets and their statistics against a list of clause replacement rules (Table 5.1). Each replacement rule takes the two sets of data (INC and EXC), and defines a function that can correctly separate them. The function is used to create a clause that replaces the existing clause with a potential repair. The repair algorithm constructs a new clause from the right side of the rule when the rule on the left side is satisfied. When more than one rule can be satisfied, the evaluation stops when the first satisfying rule is found. This process is repeated for every fault.

I illustrate the relevant data sets and explain the rules in Table 5.1 with a detailed example. Assume a clause *cls* that contains a fault inducing column *col* is faulty. The data relevant to *cls* are the values of the column *col* that are influenced by *cls*. The relevant data

Table 5.1: Clause Repair Replacement Rules

1	$min(INC) > max(EXC)$	$\Longrightarrow$	$col \geq min(INC)$
2	$max(INC) < min(EXC)$	$\Longrightarrow$	$col \leq max(INC)$
3	$max(INC) < max(EXC)$ $\wedge\ min(INC) > min(EXC)$	$\Longrightarrow$	col BETWEEN $min(INC)$ AND $max(INC)$

4	$max(EXC) < max(INC)$
	 $\wedge\ min(INC) > min(EXC)$
		$\Longrightarrow$	IF $distinct\text{-}num(INC) \leq 10$
			 THEN col IN (INC)
			 IF $distinct\text{-}num(EXC) \leq 10$
			 THEN col NOT IN (EXC)
			 ELSE remove cls

5	$distinct\text{-}num(INC) = 1$	$\Longrightarrow$	$col = max(INC)$
6	$distinct\text{-}num(EXC) = 1$	$\Longrightarrow$	$col \neq max(EXC)$
7	$\forall r(r \in INC\ \wedge r = null)$	$\Longrightarrow$	col IS null
8	$\forall r(r \in EXC\ \wedge r = null)$	$\Longrightarrow$	col IS NOT null
9	for a string type, find the largest common substring among all values in the INC, s	$\Longrightarrow$	col LIKE '%s%'
10	for a string type, find the largest common substring among all values in the EXC, s	$\Longrightarrow$	col NOT LIKE '%s%'

contains two subsets, an included data set INC (from R_i and R_a) and an excluded data set EXC (from R_s and R_e). Assume the CP that contains cls is cp. I use $cp(r)$ and $cls(r)$ to denote the boolean evaluation of row r on the CP cp and the clause cls. In R_i, the data that are relevant to cls are rows that are included because of cls and cp, that is, $cls(r) == True$ and $cp(r) == True$. Since cp contains cls, $cp(r) == True$ implies $cls(r) == True$. In R_a, I need to find rows that are excluded because of cls, that is, $cls(r) == False$. Likewise, in R_s, the relevant data are rows that evaluate to true on cp, that is, $cp(r) == True$. In R_e, I need to find rows that are excluded because of cls, that is, $cls(r) == False$. Then I project the rows identified above to the fault inducing column, col, resulting in the relevant data set INC and EXC, as shown below. The $\pi_{col}\{s\}$ denotes the values of col in a row set s.

$$\begin{aligned} INC = \pi_{col}\{r|(r \in R_i \wedge cp(r) == True) \\ \vee(r \in R_a \wedge cls(r) == False)\} \end{aligned} \tag{5.1}$$

$$EXC = \pi_{col}\{r|(r \in R_s \wedge cp(r) == True)$$
$$\vee(r \in R_e \wedge cls(r) == False)\} \tag{5.2}$$

I compute statistics from INC and EXC and evaluate them against the clause repair replacement rules in Table 5.1. The rules in Table 5.1 are based on the most commonly used data types (string, numeric, datetime, unique identifier or uuid, and boolean) and the 11 SQL comparison operators ($=$, $\neq$, $\geq$, $\leq$, IN, $NOT\ IN$, $BETWEEN$, $IS\ NULL$, $IS\ NOT\ NULL$, $LIKE$, $NOT\ LIKE$)[1]. Although these rules cover the majority of possibilities, they do omit a few rarely used SQL operators and data types. A commercial tool, of course, would need to be comprehensive.

For example, assume a clause $Year > 2007$ was mistakenly written as $Year > 2008$. The fault localization algorithm identifies that $Year$ is the fault inducing column and $Year > 2008$ is faulty. Applying Equations 5.1 and 5.2 to the rows in Table 1.1 results in $INC = \{2008, 2014, 2013\}$ and $EXC = \{2006\}$. Rule 1, $min(INC) > max(EXC)$, is satisfied, thus $col \geq min(INC)$ is used to replace the clause. A new clause $Year >= 2008$ is generated, which is correct on this data set.

This approach is similar to the DT approach in that it also separates the test data into included and excluded groups and derives the classification rule based on the grouped data. However, while the DT approach iterates over all possible columns and then splits values to derive the classification rule, this approach is based on the fault localization result, and only needs to examine the clause containing the fault inducing column and relevant data sets of the fault inducing column. It not only greatly reduces the complexity but also limits the repair attempt to faulty clauses.

[1] $<$ is included in $\leq$ and $>$ is included in $\geq$, so I don't have separate rules for $<$ and $>$.

Chapter 6: Localizing JOIN Clause Faults

6.1 JOIN Fault Types

As described in Section 2.2.1, there are tow components in JOIN clauses: JOIN type and JOIN condition. I category JOIN faults into five groups as below. The first three are JOIN condition faults, the fourth is a JOIN type fault, and the last is composite faults.

J1: Incorrect JOIN condition (e.g., the clause $O.OrderId = C.CustId$ was used, when $O.CustId = C.CustId$ should have been used)

J2: Missing JOIN condition

J3: Unwanted JOIN condition

J4: Incorrect JOIN type (e.g., LEFT JOIN was used when a RIGHT JOIN should have been)

J5: Composite faults with more than one fault

Note that CROSS JOIN is a special JOIN type. I consider INNER JOIN as a CROSS JOIN with an additional JOIN condition. Consequently, when a CROSS JOIN is used instead of an INNER JOIN, it is a J2 fault (not J4), and when an INNER JOIN is used instead of a CROSS JOIN, it is a J3 fault (not J4). Similarly, when a CROSS JOIN is incorrectly used instead of LEFT JOIN or RIGHT JOIN, it is considered as a composite fault that includes a J2 fault, which should convert the CROSS JOIN to INNER JOIN, and a J4 fault, which then converts the INNER JOIN to LEFT JOIN or RIGHT JOIN. Thus, the J4 faults actually are used to refer to incorrect use among the four join types: INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN.

Also, I do not specify unwanted or missing JOIN type faults. This is because the unwanted and missing JOIN types are always followed by unwanted JOIN conditions and missing JOIN conditions. Thus, missing JOIN type and unwanted JOIN type faults are considered to be in group J2 or J3.

All JOIN clauses can be transformed to CROSS JOIN with a WHERE clauses. For example, the query in Figure 2.1 is equivalent to the query shown in Figure 6.1. Theoretically, the exoneration-based technique introduced in Chapter 4 can be applied to localizing faults in the WHERE clause of the transformed CROSS JOIN query. I did not apply exoneration-based technique on JOIN clauses for one major reason: Computing CROSS JOIN is extremely expensive. The result of CROSS JOIN on two tables with 1000 rows is a table with 1,000,000 ($1000 * 1000 = 1,000,000$) rows. In industry databases, tables often contain tens of thousands rows, so such transformation is essentially impractical. Instead, I developed an analysis-based fault localization technique. It analyzes the failing rows and compares the result with expected result to reveal the faulty components. I will introduce this technique Section 6.2 and Section 6.3.

```
SELECT  *
FROM Order O
CROSS JOIN Customer C
WHERE O.CustId IS NOT NULL AND
      C.CustId IS NOT NULL AND
      O.CustId = C.CustId
```

Figure 6.1: Transformed CROSS JOIN Query

6.2 Join Condition Fault Localization

A faulty JOIN condition can be determined when this scenario occurs: an absent row does *not* satisfy the JOIN condition.

Assume Figure 2.1 showed a correct query. This query yields the result in Table 2.2. Now assume a faulty query in which the JOIN condition $C.CustId = O.CustId$ at line 4

Table 6.1: Result of a Faulty JOIN Condition

CustId	*OrderId*	*Year*	*Price*	*Discount*	*ZipCode*	*CustId*	*Name*
1	1	2008	110	0	22102	1	Linda
1	2	2014	120	10	22102	2	David
2	3	2013	110	5	20017	3	Andrew

was mistakenly written as $C.CustId = O.OrderId$. The result of this incorrect query is shown in Table 6.1. Comparing Table 6.1 with the correct result Table 2.2 shows that the second and third rows in Table 6.1 are superfluous, and the second through fourth rows in Table 2.2 are absent. The two absent rows do not satisfy the erroneous JOIN condition. For this reason, I can determine the JOIN condition is incorrect.

6.3 Join Type Fault Localization

To distinguish among INNER JOIN, FULL JOIN, LEFT JOIN, and RIGHT JOIN, I look for failing rows that do not have a match in the other table (*unmatched rows*). A correct FULL JOIN should not result in any superfluous unmatched rows from either LT or RT. A correct INNER JOIN should not result in any absent unmatched rows from either LT or RT. A correct LEFT JOIN should not result in any superfluous unmatched rows from LT or absent unmatched rows from RT. A correct RIGHT JOIN should not result in any absent unmatched rows from LT or superfluous unmatched rows from RT. Therefore, for a JOIN clause, when I find that the query result does not conform to the analysis above, I can conclude that the JOIN type is wrong.

For example, assume an INNER JOIN was mistakenly used instead of a FULL JOIN, resulting in Table 2.2 instead of Table 2.3. The third row in Table 2.3 is an unmatched row from LT and the last row in Table 2.3 is an unmatched row from RT. Both rows are absent. I can conclude that the use of INNER JOIN is incorrect. If the query includes multiple JOIN operators, I examine each JOIN individually. Thus my technique can detect multiple faults in multiple JOIN clauses.

Chapter 7: Repairing JOIN Clause

Based on the fault localization result of the technique introduced in Chapter 6, I developed a repair technique to fix the JOIN clause faults. I describe repairing JOIN conditions and repairing JOIN types separately in this chapter.

7.1 JOIN Condition Repair

A JOIN condition describes matching relationships between two tables. Thus, to repair a faulty JOIN condition, I must identify matching column pairs from tables as the candidate JOIN keys. To avoid evaluating each possible column pair against the entire test data set, I first filter out infeasible column pairs by examining their statistical similarities, then the remaining column pairs are evaluated against the entire data set. The repair process has five steps.

1. Create *clusters* of columns, where columns of the same data type are grouped into the same cluster.

2. For each column in the same cluster, analyze the statistics of the included test data, including the minimum, maximum, average, and number of distinct values.

3. Group columns in a cluster into sub-clusters where all statistics of the columns are equal. The columns in such sub-clusters are candidate JOIN keys.

4. Test each pair of columns from all candidate JOIN keys against the included data set. If the values of these two columns in the test data match, they are the desired JOIN keys. Then I create JOIN conditions with the desired JOIN keys.

5. If no JOIN key can be found, then JOIN conditions are not expected. I conclude that the JOIN type should be CROSS JOIN.

When using this algorithm to generate a patch for a faulty JOIN condition, redundant JOIN conditions may appear when joining more than two tables. Assume the JOIN conditions for TableA a, TableB b, and TableC c are faulty, and the expected correct JOIN condition is $a.id = b.id$ AND $b.id = c.id$. Since we have $a.id = b.id = c.id$, the final patch generated by the above algorithm could include three JOIN conditions, $a.id = b.id$, $b.id = c.id$, and $a.id = c.id$. The last condition is redundant, since the first two JOIN conditions are enough to join the three tables. I build an acyclic undirected graph to avoid this problem. Each column is a node, and a JOIN condition between two columns is an edge. To avoid joining columns in the same table, columns in the same table containing identical values are merged into a single node. After obtaining the initial JOIN condition set by the five-step process, I apply a pruning process to eliminate redundant JOIN conditions. For each JOIN condition, I add an edge if it will not create a cycle. The JOIN conditions in the resulting acyclic graph are returned as the repair for the JOIN condition fault.

7.2 JOIN Type Repair

As described in Section 6.3, if a failing row is an unmatched row from LT or RT, I can determine that the JOIN type is incorrect. I derive repair solutions by examining combinations of the JOIN type and the failing row types, as shown in Table 7.1. The header row lists four types of JOIN types. The first column lists five failing row types: a superfluous row r_s that is an unmatched row from RT, a superfluous row r_s that is an unmatched row from LT, an absent row r_a that is an unmatched row from RT, an absent row r_a that is an unmatched row from LT, and any two of the four types above. Given a faulty JOIN type and the failing row type, the corresponding cell shows the correct JOIN type.

Consider the second column in Table 7.1, in which the faulty JOIN type is INNER JOIN. In this case, all rows in the incorrect result are matched, so it is infeasible to have any superfluous unmatched rows r_s from LT or RT. I use ✗ to denote infeasible failing row types. A failing unmatched row must be an absent unmatched row from LT or RT. If all failing rows are unmatched rows from RT and are absent, then the correct type should be

Table 7.1: JOIN Type Repair

Row Type \ Faulty JOIN Type	*INNER*	*LEFT*	*RIGHT*	*FULL*
unmatched r_s from RT	✗	✗	INNER	LEFT
unmatched r_s from LT	✗	INNER	✗	RIGHT
unmatched r_a from RT	RIGHT	FULL	✗	✗
unmatched r_a from LT	LEFT	✗	FULL	✗
Two failing row types	FULL	RIGHT	LEFT	INNER

RIGHT JOIN. Likewise, if all of the failing rows are unmatched rows from LT and are absent rows, then the correct type should be LEFT JOIN. When both failing row types exist, the faulty INNER JOIN should be changed to FULL JOIN. I explain this situation using Table 2.2 and Table 2.3. Assume the INNER JOIN was mistakenly used instead of a FULL JOIN, resulting in Table 2.2 instead of Table 2.3. Comparing the faulty result in Table 2.2 with the expected result in Table 2.3, I observe two types of failing rows. The fourth row of Table 2.3 is an absent unmatched row from LT, and the last row of Table 2.3 is an absent unmatched row from RT. Having unmatched rows from both LT and RT matches the characteristic of FULL JOIN as shown in the fifth row of the first column in Table 7.1, thus, FULL JOIN should replace INNER JOIN. Similar to INNER JOIN, the last three columns in Table 7.1 show how I fix the faulty LEFT JOIN, RIGHT JOIN, and FULL JOIN.

Chapter 8: Implementation

8.1 Tool Architecture

I used Ruby to implement the fault localization and repairing techniques in an automated system, ALARM[1]. ALARM contains 52 class files and total 27,846 lines of code. Figure 8.1 shows ALARM's architecture. ALARM requires three inputs: a faulty SQL query, test rows, and a test oracle.

ALARM first localizes and repairs JOIN clause faults. Next, ALARM localizes and repairs WHERE clause faults. Finally, ALARM generates a detailed report containing four attributes: the number of failing rows, the location of the fault and the fault inducing columns for WHERE faults, the patch, and whether the patch passed all the test rows.

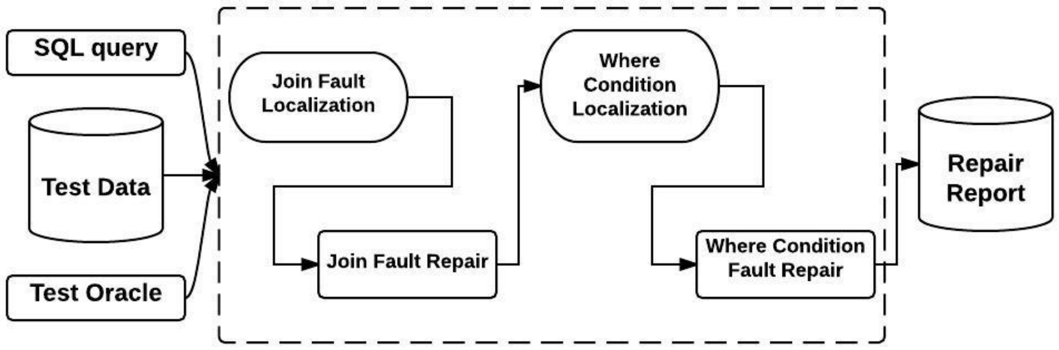

Figure 8.1: The Architecture of ALARM

[1]source code available at github repository github.com/carolfly86/altar

8.2 Implementation Enhancement

To assure good performance I took two measures. First, when exonerating suspect clauses, I need to check k-combinations of mutated rows. Instead of checking one mutated row at a time in Ruby arrays, I process all the mutated rows in a database table at once. With appropriate indexes used to accelerate the search process, the execution was 600 times faster than using Ruby arrays, for a set having 10,000 mutated rows.

Second, I encode the list of mutated columns of each mutated row as a binary number, instead of character strings. Every bit of the binary number is mapped to a column, indicating whether the column is mutated. Since I could have a total of 2^c mutated rows (c is the number of all columns), this binary encoding optimization can be efficient for both storage and searching.

Chapter 9: Fault Localization Experiments

This chapter presents the fault localization experiments. The comparison techniques, research questions, subjects, procedure, results, and analysis are discussed.

9.1 Comparison Techniques

I introduced the fault localization techniques in previous chapters. Chapter 4 discussed the WHERE fault localization techniques: the ALTAR algorithm and the improved ALTAR2 algorithm, Chapter 6 presented the JOIN fault localization technique which is referred to as JFL.

To evaluate these fault localization techniques, I compared them with the nine spectrum-based fault localization (SFL) techniques described in Section 4.2. I implemented these nine techniques by collecting the number of failing and passing test row, and then using them to calculate the suspicious formulas as described in previous papers. The twelve techniques I compared fall into four general categories:

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

9.2 Objectives

ALTAR, ALTAR2, and JFL target specific fault classes, either WHERE or JOIN faults. The similarity and statistics-based techniques are generic enough to be applied to both

JOIN and WHERE fault classes. The experiments compare the techniques in WHERE and JOIN clauses separately. The exoneration-based techniques (ALTAR and ALTAR2) are compared with the nine SFL techniques from similarity-based and statistics-based categories for WHERE fault localization. The analysis-based technique JFL is compared with the nine techniques for JOIN fault localization. I evaluated the experimental results by examining below four research questions:

- RQ1: Which is the most effective technique in WHERE clause fault localization?
- RQ2: Which is the most efficient technique in WHERE clause fault localization?
- RQ3: Which is the most effective technique in JOIN clause fault localization?
- RQ4: Which is the most efficient technique in JOIN clause fault localization?

The effectiveness is defined in terms of accuracy of finding faulty faults. The efficiency is defined in terms of execution time. The detailed metrics are described in Section 9.4.

Other researchers have studied the effectiveness of the SFL techniques as applied to general programs. In the category of similarity-based techniques, Xie et al. [17] theoretically proved that five techniques are the most effective under the assumption of 100% statement coverage. However, this assumption is often **not true** in practice. Le et al. [18] 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. My study is different; I am studying predicates and clauses in SQL queries, so statement coverage does not apply. In the category of statistics-based techniques, Zhang et al. [38] compared Liblit, SOBER, and Mann-Whitney and found Mann-Whiteney was the most effective. Wong et al. [35] compared Crosstab with Liblit and SOBER, and found that Crosstab was more effective. This research is the first experiment to compare exoneration-based and analysis-based techniques with both similarity-based and statistics-based techniques.

The efficiency related research questions (RQ2 and RQ4) are not extensively studied in

Table 9.1: Test Subject Databases

Databases	Tables	Columns (Average)	Rows (Total)
AdventureWorks	68	14	759,241
DBinventory	19	14	3,039,869
Employees	6	5	3,919,015
Mdbal	127	18	749,743
Polling_etl	9	10	211,681

previous papers. This is because 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 similar For example, when Wong et al. compared Crosstab with Tarantula [35], the time difference on the most complex program in their study was less than 0.15 seconds. I was able to explore efficiency more accurately because I used a much larger set of tests than previous studies. The databases I used have millions of rows. Each database row is a test, so I have millions of tests, compared with only hundreds of tests in previous studies. RQ2 also compares the performance of ALTAR2 with ALTAR to show that ALTAR2 is more efficient.

9.3 Experimental Subjects

I selected five subject databases. Adventureworks[1] (AW) and Employees[2] (EMP) are open source databases. Polling_etl (PLE), Dbinventory (DB), and Mdbal databases are proprietary databases owned by industry companies. Part of the agreement to use them in an experimental setting is that I am prohibited from disclosing certain details about the databases, especially their contents. The structures and sizes of the subject databases are shown in Table 9.1. *Columns* is averaged over all the tables in the relevant database.

I created three sets of queries to examine the scalability of the techniques on queries with different complexities. *Simple queries* join 2–3 tables and have 3–5 clauses in the WHERE condition, *moderate queries* join 4–5 tables and have 6–8 clauses in the WHERE condition,

[1]github.com/lorint/AdventureWorks-for-Postgres
[2]github.com/datacharmer/test_db

and *complex queries* join 6–8 tables and have 9–12 clauses in the WHERE condition. I constructed five correct queries of each size for each database. I then created eleven faulty versions of each correct query, one for each fault type (E1–E6 and J1–J5). To sum up, I had:

- 5 (databases) * 3 (query complexities) * 5 (correct queries) * 6 (faulty queries) = 450 faulty queries in WHERE conditions and
- 5 (databases) * 3 (query complexities) * 5 (correct queries) * 5 (faulty queries) = 375 faulty queries in JOIN clauses.
- Total 450 + 375 = 825 faulty queries.

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

For AW and EMP, I obtained correct queries from their tutorial examples and manually constructed faulty variations by modifying the correct versions. For PLE, DB, and Mdbal, I extracted 116 naturally occurring faulty queries from industry applications and manually created the rest. Table 9.2 shows the number of real faults by fault classes in the three industry databases: PLE, DB, and Mdbal.

In the three subject databases, the overall number of JOIN faults are less than WHERE faults. Among the JOIN faults, J4 and J5 type faults are the least and J1 faults are the most common. Among the WHERE faults, PLE and Mdbal had more E1 type faults, while DB had more E4 type faults. I cannot conclude which fault class is the most common in general, since it varies with application. However, I observe that there are relatively few composite faults (E6) in all three applications.

9.4 Procedure and Metrics

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

Table 9.2: Real Faults

Database	E1	E2	E3	E4	E5	E6	Total_E	J1	J2	J3	J4	J5	Total_J	Total
PLE	12	8	5	4	2	3	34	1	1	2	1	0	4	38
DB	6	7	5	9	6	2	35	4	1	2	1	1	9	44
Mdbal	8	4	5	3	3	2	25	2	3	2	0	1	7	32
Sum	26	19	15	16	11	7	94	7	5	6	2	2	22	116

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. I could not adopt this metric because the exoneration-based and analysis-based techniques precisely returns faulty components without a ranking.

Instead, I calculated the harmonic mean from information retrieval [50] 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 (Equation 9.1). R is the proportion of faulty clauses that are actually reported (Equation 9.2). I combined the precision and recall with their *harmonic mean* H (Equation 9.3). The higher the harmonic mean, the more effective an SFL technique is at localizing faults.

$$P = \frac{\mid Actual \cap Expected \mid}{\mid Actual \mid} \tag{9.1}$$

$$R = \frac{\mid Actual \cap Expected \mid}{\mid Expected \mid} \tag{9.2}$$

$$H = \frac{2 \cdot P \cdot R}{P + R} \tag{9.3}$$

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, I computed a suspiciousness ranking for all the clauses under test. I first fixed a faulty clause with the highest rank. I then executed the technique again to fix the next faulty clause with the highest rank in this run. I repeated the process until all the faults were fixed.

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 ith execution. The efficiency is the total time for N iterations $\sum_{i=1}^{N} t^i$ (excluding the time spent on fixing the faults manually), where t^i is the execution time of the ith iteration.

9.5 Effectiveness

This section presents the results on effectiveness for each SFL technique and the statistical comparison results. I then analyze the issues that impacted effectiveness for each technique in Sections 9.5.3 through 9.5.7.

Table 9.3: Effectiveness Comparison between SQLFL(including ALTAR2 and JFL) and Similarity-based SFLs

Database	Fault Class	SQLFL		Naish2		Wong1		Kulczynski2		Ochiai		Tarantula	
		Avg	Sdv	Avg	Sdv	Avg	Sdv	Avg	Sdv	Avg	Sdv	Avg	Sdv
AW	WHERE	0.97	0.12	0	0	0	0	0.51	0.33	0.47	0.32	0.52	0.35
	JOIN	0.81	0.13	0	0	0	0	0.37	0.36	0.37	0.36	0.37	0.36
DB	WHERE	0.99	0.05	0	0	0	0	0.38	0.30	0.36	0.24	0.37	0.26
	JOIN	0.80	0.09	0	0	0	0	0.34	0.35	0.34	0.35	0.34	0.35
Emp	WHERE	0.98	0.13	0	0	0	0	0.49	0.31	0.49	0.32	0.50	0.32
	JOIN	0.82	0.10	0	0	0	0	0.22	0.34	0.22	0.34	0.20	0.29
Mdbal	WHERE	0.95	0.15	0	0	0	0	0.45	0.34	0.45	0.33	0.48	0.37
	JOIN	0.81	0.12	0	0	0	0	0.45	0.41	0.45	0.41	0.45	0.41
PLE	WHERE	0.97	0.12	0	0	0	0	0.53	0.37	0.53	0.36	0.49	0.33
	JOIN	0.81	0.10	0	0	0	0	0.17	0.29	0.17	0.29	0.17	0.29
Overall	WHERE	0.97	0.12	0	0	0	0	0.45	0.33	0.46	0.32	0.47	0.33
	JOIN	0.80	0.11	0	0	0	0	0.30	0.36	0.29	0.36	0.29	0.35

Table 9.4: Effectiveness Comparison between SQLFL(including ALTAR2 and JFL) and Statistics-based SFLs

Database	Fault Class	SQLFL		SOBER		Liblit		Mann-Whitney		Crosstab	
		Avg	Sdv	Avg	Sdv	Avg	Sdv	Avg	Sdv	Avg	Sdv
AW	WHERE	0.97	0.12	0.51	0.34	0.48	0.34	0.25	0.30	0.46	0.34
	JOIN	0.81	0.13	0.19	0.32	0.32	0.32	0.07	0.20	0.27	0.31
DB	WHERE	0.99	0.05	0.49	0.33	0.43	0.32	0.14	0.23	0.49	0.33
	JOIN	0.80	0.09	0.15	0.31	0.32	0.33	0.04	0.17	0.26	0.30
Emp	WHERE	0.98	0.13	0.57	0.36	0.47	0.31	0.23	0.30	0.41	0.29
	JOIN	0.82	0.10	0.09	0.20	0.20	0.28	0.11	0.30	0.18	0.29
Mdbal	WHERE	0.95	0.15	0.53	0.36	0.43	0.33	0.18	0.30	0.36	0.30
	JOIN	0.81	0.12	0.28	0.40	0.27	0.26	0.01	0.06	0.27	0.28
PLE	WHERE	0.97	0.12	0.54	0.36	0.39	0.29	0.23	0.29	0.56	0.37
	JOIN	0.81	0.10	0.08	0.16	0.12	0.24	0.07	0.18	0.14	0.22
Overall	WHERE	0.97	0.12	0.53	0.35	0.44	0.32	0.21	0.29	0.45	0.33
	JOIN	0.80	0.11	0.16	0.30	0.25	0.29	0.09	0.23	0.23	0.29

9.5.1 Results

Tables 9.3 and 9.4 show the effectiveness of the techniques. *SQLFL* (SQL Fault Localization) denotes my fault localization technique. It locates WHERE clause faults using exoneration-based techniques (ALTAR or ALTAR2) and identifies JOIN clause faults using JFL. Note that the exoneration-based techniques ALTAR and ALTAR2 have identical effectiveness, so I omit ALTAR data in both tables and use ALTAR2 to represent both exoneration-based techniques. The first column shows the five databases. The second column indicates the fault class is either WHERE or JOIN clause. Table 9.3 compares SQLFL with the five similarity based techniques, and Table 9.4 compares SQLFL with the four statistics-based techniques. For each subject, Tables 9.3 and 9.4 show the mean (*Avg*) and standard deviation (*Sdv*) of the effectiveness on the queries for each technique. *Overall Avg* shows the mean of the effectiveness on all the queries in the five databases.

Tables 9.3 and 9.4 show that SQLFL was the most effective technique . Although the similarity and statistic based techniques can be applied to both JOIN and WHERE faults, the effectiveness in both fault classes are very low. Table 9.3 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 [17]. These two techniques gave the same suspiciousness score to all clauses, which means they were unable to rank the clauses by suspiciousness. So the resulting effectiveness was 0. I will explain this issue in Section 9.5.6. Kulczynski2, Ochiai, and Tarantula had similar effectiveness. Table 9.4 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. I will examine the statistical significance of the differences in Section 9.5.2.

Tables 9.3 and 9.4 show that SQLFL 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 SQLFL was 0.77. The standard deviation of SQLFL was also much lower than the other techniques, indicating that they performed consistently over all queries. **The answers to RQ1 and RQ3 are clear: SQLFL was the most effective**

technique. Specifically, ALTAR2 was the most effective technique at localizing WHERE faults and JFL is the most effective at localizing JOIN faults.

9.5.2 Statistical Comparison

I use statistical analysis to compare techniques with similar effectiveness. The paired two-tailed t-test is used because I had enough queries (825) to assume normal distributions. I did not need to statistically compare SQLFL because it was so much more effective than the others. Mann-Whitney, Naish2, and Wong1 were excluded because they were much less effective. I 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.

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 9.5 shows the p-score for each pair of techniques at the significant level of $\alpha = 0.05$. If p-score is less than α (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 μ_d column shows the average of differences between technique X and Y ($\mu_d = avg(x_i - y_i)$). A positive μ_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, μ_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

Table 9.5: Hypothesis Testing Results

Pairs (X-Y)	p-score	H_0 Rejected?	μ_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	-

similar effectiveness. Therefore, I can conclude that Kulczynski2, Tarantula, Ochiai were the most effective, whereas the theoretically best techniques were the least effective. Among the statistics-based SFLs, SOBER was the most effective and Liblit and Crosstab were not significantly different. Thus, I can conclude that SOBER was the most effective technique among the four statistic-based techniques, although it was less effective than SQLFL. The effectiveness of Liblit and Crosstab were close to that of Kulczynski2, Tarantula, and Ochiai. Thus, they are considered to be equivalent in effectiveness. Mann-Whitney was less effective than Liblit, Crosstab, Kulczynski2, Tarantula, and Ochiai; but it was more effective than Naish2 and Wong1. The final order of effectiveness for the ten techniques is shown in Figure 9.1.

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) an issue that is specific to the analysis-based technique, (3) issues that are common to all similarity-based and statistics-based techniques, (4) issues that are specific to the similarity-based techniques, and (5) issues that are specific to the statistics-based techniques.

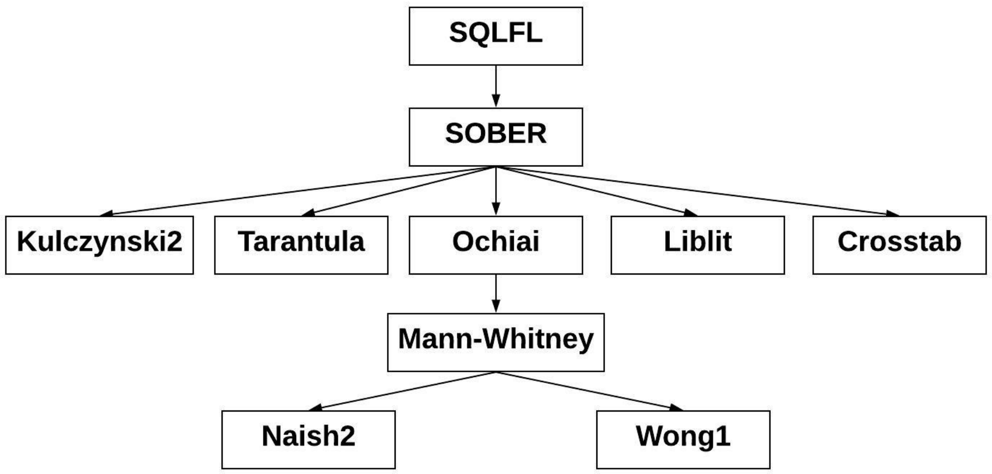

Figure 9.1: Effectiveness Ranking of The Ten Techniques

9.5.3 An Issue Specific to Exoneration-based Technique

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

9.5.4 An Issue Specific to Analysis-based Technique

The analysis-based technique (JFL) checks existing clauses in join conditions against failing rows to determine suspicious join conditions. However, it can determine the join conditions contains faults It can not determine if a clause is missing (fault type J2 in Section 6.1).

9.5.5 Issues Common to all Similarity-Based and Statistics-Based Techniques

Similarity-based and statistics-based techniques are coverage based fault localization technique. They are generic enough to be applied to both WHERE and JOIN faults. However, the effectiveness of these techniques is low due to following reasons.

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 4.1 or fault type J2 in Section 6.1). In contrast, ALTAR2 can accurately report not only the missing clauses but also associated fault-inducing columns. The average effectiveness of similarity and statistics-based techniques (except Naish2, Wong1, and Mann-Whitney) increased by about 10% when excluding SQLs with E4 faults from the experiment subjects. In addition, they cannot identify JOIN type faults (fault type J4 in Section 6.1). Similarity-based and statistics-based techniques can not interpret the semantic meanings of JOIN type keywords, thus they cannot distinguish the different JOIN types.

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 fault localization, 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.

9.5.6 Issues Specific to Similarity-based Techniques

In this section, I derive the suspiciousness formulas for each similarity-based technique to analyze their effectiveness.

Recall that I defined four variables for the similarity-based formulas in Section 4.2.1, 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, $c_{ef}^{t} + c_{ef}^{f} = T_f$. 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, $c_{ep}^{t} + c_{ep}^{f} = T_p$.

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

- Naish2:

$$\begin{aligned} S(c) = S(c)^t + S(c)^f &= c_{ef}^{t} + c_{ef}^{f} - \frac{c_{ep}^{t}}{T_p + 1} - \frac{c_{ep}^{f}}{T_p + 1} \\ &= T_f - T_p/(T_p + 1) \end{aligned} \tag{9.4}$$

- Wong1:

$$S(c) = S(c)^t + S(c)^f = c_{ef}^{t} + c_{ef}^{f} = T_f \tag{9.5}$$

- Kulczynski2 :

$$\begin{aligned} S(c) &= \frac{1}{2} * (\frac{c_{ef}^{t}}{T_f} + \frac{c_{ef}^{t}}{c_{ef}^{t} + c_{ep}^{t}} + \frac{c_{ef}^{f}}{T_f} + \frac{c_{ef}^{f}}{c_{ef}^{f} + c_{ep}^{f}}) \\ &= \frac{1}{2} * (\frac{c_{ef}^{t} + c_{ef}^{f}}{T_f} + \frac{c_{ef}^{t}}{c_{ef}^{t} + c_{ep}^{t}} + \frac{c_{ef}^{f}}{c_{ef}^{f} + c_{ep}^{f}}) \end{aligned} \tag{9.6}$$

- Ochiai :

$$S(c) = \frac{c^t_{ef}}{\sqrt{(T_f) * (c^t_{ef} + c^t_{ep})}} + \frac{c^f_{ef}}{\sqrt{(T_f) * (c^f_{ef} + c^f_{ep})}} \tag{9.7}$$

- Tarantula :

$$S(c) = \frac{c^t_{ef}/T_f}{c^t_{ef}/T_f + c^t_{ep}/T_t} + \frac{c^f_{ef}/T_f}{c^f_{ef}/T_f + c^f_{ep}/T_t} \tag{9.8}$$

Equations 9.4 and 9.5 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. [17] proved that Nashi2 and Wong1 are the theoretically best on statements, with the assumption that the tests covered all statements. But Le et al. [18] 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.

9.5.7 Issues Specific to Statistics-based Techniques

Now I analyze the four statistics-based techniques.

Crosstab: Section 4.2.2 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 9.5.2 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. I derive the Liblit formulas for a clause c in Equations 9.9, 9.10, and 9.11.

$$\begin{aligned} Context(c) =& Pr(Crash|\, c\, observed) \\ =& T_f/(T_f + T_p) \end{aligned} \tag{9.9}$$

$$\begin{aligned} Failure(c) =& Pr(Crash|\, c\, observed\, \mathbf{true}) \\ =& c^t_{ef}/(c^t_{ef} + c^t_{ep}) \end{aligned} \tag{9.10}$$

$$\begin{aligned} Increase(c) =& Failure(c) - Context(c) \\ =& c^t_{ef}/(c^t_{ef} + c^t_{ep}) - T_f/(T_f + T_p) \end{aligned} \tag{9.11}$$

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. [38] found that Mann-Whitney was found to be more effective than SOBER. However, my experiments found the opposite.

As explained in Section 4.2.2, 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 Equation 4.10. Such situations are in fact quite common. Among the four clauses in Table 4.2, 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. I also observed this in the experiment, for WHERE faults, Mann-Whitney gave the same ranks to all the clauses in 58% of the subject queries, thus resulting in 58% with 0 effectiveness; and for JOIN faults, it gave the same ranks to all the JOIN conditions in 84% of the subject queries, thus resulting in 84% of 0 effectiveness. In contrast, SOBER was able to rank the clauses. Therefore, Mann-Whitney was less effective than SOBER.

9.6 Efficiency

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

9.6.1 Results

I found that the five similarity-based techniques had almost the same execution time as three of the statistic-based techniques: Liblit, SOBER, and Crosstab. For these eight techniques, over 99% of the time was spent on executing tests and collecting runtime information, and the computation of suspiciousness rankings took less 1% of the time. Because they are executed on the same test data sets, the execution time for those techniques are the same. This is consistent with Wong et al.'s results [35]. These eight SFL techniques are referred to as *Others*. Mann-Whitney took more time than the other eight similarity- and statistics-based techniques due to the non-parametric statistical model. Therefore, Table 9.6 and

Table 9.6: WHERE Faults Localization 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
Multi Faults	5.88	336.53	84.48	24.67	2315	721,187
Overall	2.81	97.44	64.37	11.50	1462	683,825

Table 9.7: JOIN Faults Localization Efficiency Results (in Seconds)

Query Types	Time (avg.)			# of Rows (avg.)	
	JFL	MW	Others	Failing	All
Simple	19.71	29.45	29.44	1,112,005	1,138,432
Moderate	11.86	7.88	5.26	89,729	94,093
Complex	47.03	8.93	8.93	57,290	79,273
Single Fault	22.98	9.34	9.34	259,807	269,083
Multi Faults	32.73	16.14	15.61	454,731	473,849
Overall	26.49	15.47	14.58	422,785	440,290

Table 9.7 give time in seconds for all the techniques, combined into five groups, ALTAR2, ALTAR, JFL, Mann-Whitney (MW), and the other SFL techniques (Others).

The *Types* column of Table 9.6 and Table 9.7 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. As shown in Table 9.6, ALTAR2 took 2.8 seconds for all 450 queries on average, much faster than the other techniques. Thus, the answer to RQ3 is that ALTAR2 was the most efficient technique. ALTAR took 97.44 seconds on average, so it is the least efficient. In Table 9.7, *Others* are the most efficient. The overall average execution time is only 14.58 seconds for all queries. I will analyze the experimental results in detail in Section 9.6.3.

9.6.2 Comparing SFL with Manual Debugging

I conducted a study to compare automatic fault localization techniques with manual debugging to determine if manual debugging is comparable in terms of efficiency. I invited a developer who has 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 ALTAR2 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 than the automatic SFL techniques, thus I did not conduct further experiments on manual debugging.

9.6.3 Analysis and Discussion

WHERE Faults

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 suspect 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, and even rarer 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. I 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 my study, the average of e was four and average of f was 1462. Consequently, the complexity of ALTAR2 is much smaller than ALTAR. In the 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 $O(e * 2^c)$ for ALTAR2 can be neglected and conclude the complexity is dominated by $O(f * k)$, $O(f * 2^c)$ for ALTAR cannot be neglected.

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 $O(f*k) < O(n*k) < O(\binom{n}{p}) < O(f*k+f*2^c)$, my analysis is consistent with the observed efficiency results.

JOIN faults

Table 9.7 shows that *JFL* is the slowest among all techniques. However, the reason is that *MW* and *Others* are not able to detect JOIN type faults, so they are only executed for JOIN condition faults. *JFL* checks for both JOIN type faults and JOIN condition faults, thus the execution time is more than the other techniques. In additions, JOIN type faults often produces large number of rows with *null* values. Querying tables with null-valued rows are often slow because those rows can not be indexed. Both factors contribute to the JFL technique being slow. And the impact of *null* values rows is significant. I observed that querying a table with total 264,345 rows and 153,853 *null* values rows took 285 seconds. The same query completed in less than 1 seconds when all the *null* values are updated to empty strings. Note that the average execution time of JOIN faults for *MW* technique is much smaller comparing that of WHERE faults. When *MW* computes $R(p) = 0$ for all components it essentially fails to rank the components and the fault localization process will terminate prematurely. In this situation, the execution time is short because the complicated computation is not yet executed. *MW* failed to rank 84% of the subject queries in JOIN faults comparing to 58% subject queries in WHERE faults. As shown in Table 9.4, the overall average effectiveness is only 9% for JOIN faults and 21% for WHERE faults. So the average execution time of *MW* in JOIN faults is smaller.

9.7 Threats to Validity

An external threat is that the subjects may not be representative. I ameliorated the threat by selecting five databases from different sources, two from open source repositories and three from industry. To increase diversity, I 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. I was careful to implement them exactly as described in these papers, and designed tests to ensure they worked as expected.

One construct validity threat is how I created faults. I used a combination of natural faults and manually constructed faulty SQL queries. Using a different source of faults could have led to different results. The faulty queries used in the experiment include five different individual faults as well as composite faults. Compared to other fault localization empirical studies, I 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. I used the "perfect bug detection" assumption [51] to calculate the number of repeats needed to fix all the faults. That is, I 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 I 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 the core result that ALTAR2 was the most efficient technique.

Chapter 10: Fault Repair Experiments

I evaluated my automatic faulty query fix technique (ALARM) by conducting a set of comprehensive experiments. I used Gopinath et al.'s DT approach [10] introduced in Section 1.2.4 as the experimental baseline. The experiment result is analyzed from exploring two research questions. I illustrate the experimental procedure and measurements, followed by an analysis of the results and discussion of the threats to validity.

10.1 Experimental Design

I evaluated two aspects of effectiveness, correctness and acceptability, as well as the efficiency.

- RQ1 (*Effectiveness*): Is ALARM more effective than the baseline for every fault class?
  - RQ1a (*Correctness*): Can ALARM generate patches that pass all test rows for more queries than the baseline?
  - RQ1b (*Acceptability*): Are the ALARM patches more acceptable to users than the baseline?
- RQ2 (*Efficiency*): Is ALARM more efficient than the baseline in terms of execution time?

In the experiments, a patch is always evaluated against all rows to ensure that the patch does not introduce new fault. The effectiveness is evaluated from two aspects: *correctness* measures the percentage of faulty queries fixed over total number of faulty queries. and *acceptability* measures the similarity of the patch to a correct query. Similarity is defined in Section 10.3. The efficiency is measured in terms of time to generate a patch, as measured on a MacBook Pro with 2 Intel i7 cores and 16 GB RAM.

10.2 Experimental Subjects

In this experiment, I used the same experiment subjects introduced in Section 9.3. In total, there are 825 faulty queries (450 faulty queries in WHERE conditions + 375 faulty queries in JOIN clauses) from five subject databases. The details of the subject queries and database schema are presented in Section 9.3.

My subjects, both the databases and the queries, are larger than Gopinath et al.'s [10], which only used seven subject code segments and about 170,000 test rows in total.

10.3 Experimental Procedure

For each faulty query, I generated one patch with the DT approach and one patch with ALARM, then compared and analyzed the results.

I consider a patch to be correct if all test rows pass. Note that this is not the same as fully satisfying the specification, that is, correctness is defined relative to the tests.

Previous studies have found that this relative correctness means repair techniques sometimes generate nonsensical patches [9]. To accommodate this, patches are also evaluated for acceptability. Kim et al. [9] attempted to evaluate acceptability by interviewing developers. However, Monperrus [24] criticized this evaluation method due to the participants' bias and lack of domain knowledge. To avoid this problem in the experiment, I obtained the correct queries from source code history when available, and generated them by hand when not. I then measured structural differences between each patched query and the correct query. Patches that are more similar to the correct query are considered to be more acceptable.

As introduced Section 2.2.2, each predicate can be reduced to disjunctive normal form (DNF). Because the IDNF is unique for monotone boolean functions [27], there is only one IDNF for a given query. If the IDNF of a patch is identical to that of the correct query, then the similarity is perfect. Predicates in IDNF are treated as unordered trees, where each CP is a subtree and each clause is a node. Figure 10.1 shows the INDF of the WHERE condition in Figure 1.1. The root node is a place holder for joining two branches (CPs).

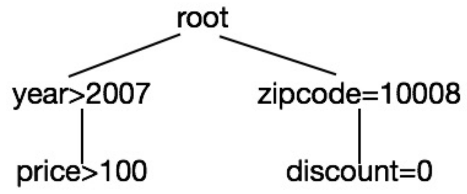

Figure 10.1: Correct Query IDNF Tree Structure

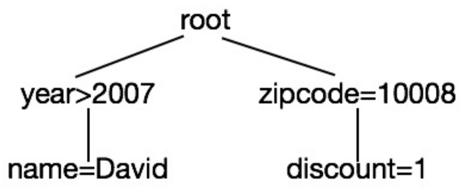

Figure 10.2: Incorrect Query IDNF Tree Structure

X-Diff algorithm [52] was originally used to compare XML structures. I adopted X-Diff to compare the IDNF conditions. X-Diff converts XML files into unordered tree structures, then computes the edit distance from one tree to another tree. The edit distance between two trees is the minimum number of node edit operations required to transform one tree to another. The three possible edit operations are inserting a node, editing a node, and deleting a node. In my application, the edit operation is only allowed when the source and target nodes contain the same column. I demonstrate the edit distance calculation using the following example. Assume Figure 10.2 is the IDNF for an incorrect query. Three edit operations are required to transform the incorrect query IDNF in Figure 10.2 to the correct query IDNF in Figure 10.1: (1) **remove** the clause $name = David$; (2) **insert** the clause $price > 100$; (3) **edit** clause $discount = 1$ to $discount = 0$. Thus, the edit distance between the two IDNF tree structures is 3. The maximum edit distance between a condition with m clauses and a condition with n clauses is when I must remove all clauses in one query, then insert them into the other query, or $m + n$. I use Equation 10.1 to calculate the *similarity*. If a patch can be transformed into the correct condition with fewer operations than another patch, it is more similar and is deemed to be more acceptable to the developer.

$$similarity = 1 - \frac{edit\_distance}{max\_edit\_distance} \qquad (10.1)$$

I consider the JOIN conditions and JOIN types as single branch trees since they are connected with AND operators. Equation 10.1 is also used to measure similarity of JOIN fault patches.

10.4 Experimental Results and Analysis

10.4.1 Effectiveness

This section presents the effectiveness data on the subjects and analyzes correctness and acceptability.

Correctness

If a patch can pass all tests, it is considered a *correct* patch. Table 10.1 shows averaged percentages of correct patches generated by ALARM and DT. Faulty queries are grouped by complexity; simple (S), moderate (M), and complex (C), and then by JOIN faults and WHERE faults. The $ALARM$ and DT columns show the average correct query percentages for queries with the same complexity and fault class for each database.

The DT approach does not distinguish JOIN types, LEFT JOIN, RIGHT JOIN, or FULL JOIN. Thus, it cannot localize or repair *J4* faults. For JOIN condition faults, the DT approach can't generate a patch before exhausting 100GB of disk space (efficiency is discussed in Section 10.4.2). These are marked as "N/A" in Table 10.1. ALARM is able to precisely find the JOIN fault location and determine the fault type, thus, the correctness is very high. For the WHERE faults, both ALARM and the DT approach were effective. I compared the correctness of ALARM and DT with two-tailed t-test at 0.05 significance level. The t-value is 0.817, the effect size is 0.446 and the p-value is 0.414. It indicates there is no significant difference between the two methods at 0.05 significance level. The effectiveness of both approaches was consistent across the different query groups. The answer to *RQ1a* is that ALARM is capable of fixing JOIN fault classes when the DT approach is infeasible and the correctness for WHERE faults is comparable to DT approach.

ALARM fails to generate correct patches for WHERE faults when it cannot identify the correct fault inducing column in the fault localization step. Assume there is a *W3* faulty clause that uses $C.OrderId$ instead of $C.CustId$. At the fault localization phase, ALARM may incorrectly determine that $C.OrderId$ is the fault inducing column. Because

Table 10.1: Correctness Comparison

Database	Complexity	Fault Classes	ALARM	DT
EMP	S	JOIN	1.00	N/A
		WHERE	0.83	0.91
	M	JOIN	1.00	N/A
		WHERE	0.80	0.86
	C	JOIN	0.89	N/A
		WHERE	0.93	0.87
AW	S	JOIN	0.96	N/A
		WHERE	0.90	0.86
	M	JOIN	1.00	N/A
		WHERE	0.79	0.84
	C	JOIN	0.81	N/A
		WHERE	1.00	0.83
PLE	S	JOIN	1.00	N/A
		WHERE	0.90	0.85
	M	JOIN	0.97	N/A
		WHERE	0.80	0.87
	C	JOIN	0.98	N/A
		WHERE	1.00	0.81
DB	S	JOIN	1.00	N/A
		WHERE	0.97	0.81
	M	JOIN	1.00	N/A
		WHERE	0.93	0.82
	C	JOIN	1.00	N/A
		WHERE	0.93	0.81
Mdbal	S	JOIN	1.00	N/A
		WHERE	0.97	0.83
	M	JOIN	1.00	N/A
		WHERE	0.86	0.82
	C	JOIN	1.00	N/A
		WHERE	0.62	0.86
Overall		JOIN	0.97	N/A
		WHERE	0.88	0.86

Table 10.2: Acceptability Comparison

Database	Query Type	Fault Class	ALARM	DT
EMP	S	JOIN	1.00	N/A
		WHERE	0.77	0.12
	M	JOIN	1.00	N/A
		WHERE	0.87	0.08
	C	JOIN	1.00	N/A
		WHERE	0.90	0.05
AW	S	JOIN	0.87	N/A
		WHERE	0.75	0
	M	JOIN	0.97	N/A
		WHERE	0.86	0
	C	JOIN	0.93	N/A
		WHERE	0.90	0.07
PLE	S	JOIN	0.98	N/A
		WHERE	0.81	0.09
	M	JOIN	0.96	N/A
		WHERE	0.85	0.02
	C	JOIN	0.91	N/A
		WHERE	0.92	0.04
DB	S	JOIN	0.93	N/A
		WHERE	0.80	0.05
	M	JOIN	0.85	N/A
		WHERE	0.87	0.07
	C	JOIN	0.89	N/A
		WHERE	0.92	0.04
Mdbal	S	JOIN	0.85	N/A
		WHERE	0.83	0.06
	M	JOIN	0.80	N/A
		WHERE	0.89	0.07
	C	JOIN	0.79	N/A
		WHERE	0.89	0.18
Overall		JOIN	0.92	N/A
		WHERE	0.85	0.06

the repair algorithm depends on the identified fault-inducing column to generate a new clause and replace the existing faulty clause, choosing $C.OrderId$ as the fault-inducing column in the repair phase would not fix the faulty clause.

Acceptability

I measure patches' acceptability with similarity (Equation 10.1). Table 10.2 shows the averaged similarities of the patches generated by ALARM and the DT approach in all the queries grouped by database, query complexity, and fault type.

Because ALARM can precisely locate the fault location and type, it can usually fix only the faulty clause without impacting other correct clauses. Thus, the similarity of ALARM is very high, with a perfect 1.0 in several cases..

On the other hand, the DT approach is impacted by the "overfitting" problem. It often selects columns that have more distinct values so that it can construct classification functions that correctly predict every test row. When databases have many columns with many distinct values, such as name, id, and date of birth, the DT patches often contain such columns even when not relevant. This problem is more significant with complicated queries that reference many columns. The complex queries have the lower similarities across the five databases in Table 10.2. In particular, the AW database tables have more columns than the other two, so the patches generated by the DT approach have the lowest similarities.

Below is an example with the correct predicate, the faulty predicate, the patch predicate generated by ALARM, and the patch predicate generated by the DT approach. It shows that the ALARM-generated patch only changes the faulty clause, while the DT-generated patch uses irrelevant columns. Thus, the answer to *RQ1b* is that ALARM-generated patches are more acceptable than the DT approach.

- Correct predicate:

  $(required = t$ AND $questioncode > 7)$ OR

  $questionposition = 1$ OR

  $externalsystemid < 70$

- Faulty predicate:

  $(required = t$ AND $\boldsymbol{questioncode > 8})$ OR

  $questionposition = 1$ OR

  $externalsystemid < 70$

- Patch generated by ALARM:

  $(required = t$ AND $\boldsymbol{questioncode \geq 8})$ OR

  $questionposition = 1$ OR

$externalsystemid < 70$

- Patch generated by the DT approach:

  ($questions\_label$ = 'AccessTokens') OR

  ($questions\_surveyid > 9.50$ AND $questions\_label$ = 'Lead Score') OR

  ($questions\_label$ = 'Notes') OR

  ($questions\_status$ = 'DELETED' AND $questions\_label$ = 'Quick Notes')

10.4.2 Efficiency

Table 10.3 shows the average execution times of ALARM and the DT approach in seconds. Since ALARM and the DT approach are based on test suites, the execution time is relative to the test suite size, which was the same for both techniques. As shown in Table 10.3, fixing queries in the largest database *EMP* and *DB* took the longest time, and fixing queries in the smallest database *PLE* took the least time.

Query complexity also affects the execution time. The execution time of ALARM is proportional to $number\_of\_failing\_rows * number\_of\_clauses$. For the DT approach, the execution time is proportional to $total\_number\_of\_rows * number\_of\_columns$, where the $number\_of\_columns$ is proportional to the query complexity. Table 10.3 shows the execution time increases as the query complexity increases in the smaller databases, *PLE* and *AW*. However, in the large database, *EMP*, the number of rows was nearly four million, so the execution time is dominated by the number of rows instead of the query complexity.

For the WHERE faults, the overfitting problem discussed in Section 10.4.1 also impacted the efficiency of the DT approach. The DT approach generates decision trees whose sizes are relative to the size of the database and the complexity of the queries. For the JOIN faults, the DT approach constructs a cross product, which is infeasible for large test databases. For example, a simple query in the smallest database (Polling_Etl) has three tables. The cross product of three tables with 10,000 rows each could result in a table with 1,000,000,000,000 (one trillion!) rows.

Table 10.3: Execution Time Comparison (Seconds)

Database	Query Type	Fault Class	ALARM	DT
EMP	S	JOIN	278.46	N/A
		WHERE	29.86	6927.8
	M	JOIN	35.65	N/A
		WHERE	141.23	18,384.0
	C	JOIN	11.94	N/A
		WHERE	46.80	3044.4
AW	S	JOIN	8.35	N/A
		WHERE	1.93	63.0
	M	JOIN	11.4	N/A
		WHERE	1.27	75.2
	C	JOIN	41.43	N/A
		WHERE	12.03	149.2
PLE	S	JOIN	0.43	N/A
		WHERE	5.33	5.6
	M	JOIN	0.58	N/A
		WHERE	0.90	10.6
	C	JOIN	1.04	N/A
		WHERE	0.80	102.4
DB	S	JOIN	2.76	N/A
		WHERE	40.83	7352.3
	M	JOIN	89.80	N/A
		WHERE	66.17	14,589.2
	C	JOIN	472.52	N/A
		WHERE	138.03	24,374.1
Mdbal	S	JOIN	0.43	N/A
		WHERE	17.30	963.0
	M	JOIN	0.58	N/A
		WHERE	10.14	514.0
	C	JOIN	1.04	N/A
		WHERE	19.71	105.8
Overall		JOIN	63.76	N/A
		WHERE	35.49	5100.7

Table 10.4: Execution Time Percentage in Each Step

Step Name	ALARM	DT
data pre-processing	27%	1%
fault localization	62%	N/A
fault repairing	11%	99%

Overall, the execution time of ALARM is only a fraction of the DT approach. I can conclude that the answer to *RQ2* is that ALARM is much more efficient than the DT approach.

To further investigate how the two techniques allocate the time, I break down the execution time for each approach by step and show the result in Table 10.4. In the data preprocessing step, both approaches label the test data and split them into the different groups. In addition, ALARM calculates statistical data from database tables. The most expensive step for ALARM is the fault localization, which involves checking all failing rows, mutating them to locate the faulty clause, and finding fault inducing columns. Since the DT approach does not localize faults, this step is marked as "N/A." In fault repairing step, ALARM leverages the fault localization information, the fault repair step is much faster. Note that the statistical computations are done in the pre-processing step, thus saving time in the repair step. However, DT approach spend almost all of the time in the fault repairing step. It iterates all columns and check each possible splitting value to derive the classification rule.

10.5 Threats to Validity

An external threat common to all software engineering studies, is that the subjects may not be representative. I ameliorated the threat by selecting databases from open source projects and commercial software, and constructing queries with different complexities using various fault classes. I used 116 real faulty queries. Moreover, the number of columns and rows in my subject databases are much larger than in previous studies. The two largest databases use a lot of numeric and date data types, and relatively few string data types, which influences which rules come into play the most. I see no reason this would affect the conclusions, however.

Another external threat is that the implementation of the DT approach may affect the results. I was careful to implement the DT approach exactly as described in their papers, and tested the software to ensure it worked as expected.

An internal threat is that we used a Python library[1] to implement the DT approach. The Python library is the most well-maintained and widely used in the data science community. In contrast, I tested another Ruby decision tree library[2], and found the it to be much slower.

[1]https://svaante.github.io/decision-tree-id3/
[2]https://github.com/igrigorik/decisiontree

Chapter 11: Conclusion and Future Work

This dissertation presents a novel SQL fault localization and repairing technique that targets two major classes of SQL faults: faults in JOIN clauses and faults in WHERE predicates. It accurately identifies the fault location and then uses statistical information about test suites to efficiently repair faults with minimal changes. I implemented this technique into a tool, *ALARM*, and compared it with nine state-of-the-art fault localization techniques and the only existing SQL repair technique *DT*. I did experiments on 825 faulty queries to evaluate my fault localization and fault repair techniques. The fault localization experimental results show that ALARM is more effective and efficient than all nine of the state-of-the-art fault localization techniques. The fault repair experimental results show that ALARM is capable of fixing JOIN fault classes when the DT approach is infeasible. And for WHERE predicate faults, although the number of passing patches generated by *ALARM* and *DT* are comparable, *ALARM*'s patches are more acceptable. In addition, *ALARM* is highly efficient and scales well with the number of test rows.

Future work will be in two additional directions. First, I will improve the fault localization and repairing effectiveness. I observed that some faults are not correctly repaired by *ALARM* because the queries contained multiple faults, and others because the fault localization was incorrect. I plan to improve the fault localization and repairing algorithm to correctly fix such faults. Second, I will study fault localization and repair for more classes of SQL faults. JOIN and WHERE clauses are the common components in SQL query, and I will adapt my techniques to faults in other SQL elements, such as GROUP BY aggregation operations and SELECT target lists. I am also exploring repairs to additional data types, such as XML and hash sets.

Bibliography

Bibliography

[1] K. Bouwkamp, "The 9 most in-demand programming languages of 2016," Online, Coding Dojo Blog, January 2016, http://www.codingdojo.com/blog/9-most-in-demand-programming-languages-of-2016/, last access: September 2017.

[2] JPMorgan Chase Institute, "Weathering volatility, big data on the financial ups and downs of u.s. individuals," Online, May 2015, https://www.jpmorganchase.com/corporate/institute/document/54918-jpmc-institute-report-2015-aw5.pdf, last access: March 2018.

[3] "Top 10 largest databases in the world," Online, March 2010, https://www.comparebusinessproducts.com/fyi/10-largest-databases-in-the-world, last access: May 2018.

[4] B. Adam Routh, "Alibaba records $25.3 billion in singles day sales," Online, November 2017, https://www.nasdaq.com/article/alibaba-records-253-billion-in-singles-day-sales-cm876020, last access: April 2018.

[5] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *31st International Conference on Software Engineering (ICSE)*, Vancouver, BC, Canada, 2009, pp. 364–374.

[6] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "Bugfix: A learning-based tool to assist developers in fixing bugs," in *17th International Conference on Program Comprehension.* Vancouver, British Columbia, Canada: IEEE Press, 2009, pp. 70–79.

[7] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *International Conference on Software Engineering (ICSE)*, San Francisco, USA, 2013, pp. 772–781.

[8] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *3rd International Conference on Software Testing, Verification, and Validation*, Paris, France, 2010, pp. 65–74.

[9] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *IEEE/ACM 2013 International Conference on Software Engineering*, San Francisco, CA, USA, 2013, pp. 802–811.

[10] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra, "Data-guided repair of selection statements," in *36th International Conference on Software Engineering.* Hyderabad, India: ACM, 2014, pp. 243–253.

[11] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold, "Localizing SQL faults in database applications," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lawrence, KS, USA, November 2011.

[12] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. Nguyen, "Database-aware fault localization for dynamic web applications," in *29th IEEE International Conference on Software Maintenance (ICSM)*, Eindhoven, Netherlands, December 2013, pp. 456–459.

[13] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra, "Fault localization for data-centric programs," in *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, Szeged, Hungary, September 2011, pp. 157–167.

[14] M. Monperrus, "Automatic Software Repair: a Bibliography," *ACM Computing Surveys*, vol. 51, pp. 17:1–17:24, Janunary 2018.

[15] M. Weiser, "Program slicing," in *5th International Conference on Software Engineering*, San Diego, California, USA, 1981, pp. 439–449.

[16] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[17] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodolgy*, vol. 22, no. 4, pp. 31:1–31:40, Oct. 2013.

[18] T.-D. B. Le, F. Thung, and D. Lo, "Theory and practice, do they match? A case with spectrum-based fault localization," in *2013 IEEE International Conference on Software Maintenance*, Sept 2013, pp. 380–383.

[19] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, CA, USA, 2005, pp. 273–282.

[20] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, Windsor, UK, 2007, pp. 89–98.

[21] Y. Guo, A. Motro, and N. Li, "Localizing faults in SQL predicates," in *Tenth International Conference on Software Testing, Verification, and Validation (ICST)*, Tokyo, Japan, March 2017.

[22] Y. Guo, N. Li, J. Offutt, and A. Motro, "Exoneration-based fault localization for SQL predicates," *Journal of Systems and Software*, under review.

[23] ——, "Automatically repairing SQL faults," in *18th IInternational Conference on Software Quality, and Security (QRS)*, Lisbon, Portugal, July 2018.

[24] M. Monperrus, "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, 2014, pp. 23–242.

[25] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge, UK: Cambridge University Press, 2017, ISBN 978-1107172012.

[26] N. Li and J. Offutt, "Test oracle strategies for model-based testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, April 2017.

[27] L. M. Karahanjan and A. A. Sapozhenko, "On some operations of partial monotone boolean function simplifying," in *Fundamentals of Computation Theory, International Conference (FCT)*, Kazan, USSR, June 1987, pp. 231–233.

[28] J. R. Quinlan, *Induction of Decision Trees.* Hingham, MA, USA: Kluwer Academic Publishers, 1986.

[29] W. E. Wong, R. Go, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transaction on Software Engineering*, vol. 42, pp. 707–740, August 2016.

[30] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Sixth International Symposium on Software Reliability Engineering (SRE)*, Toulouse, France, October 1995, pp. 143–151.

[31] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, verification, and Reliability*, vol. 10, pp. 171–194, 2000.

[32] H. A. Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *The Computing Research Repository (CoRR)*, vol. abs/1607.04347, 2016.

[33] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on Software Engineering Methodology*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

[34] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *31st International Computer Software and Applications Conference.* Beijing, China: IEEE Computer Society, 2007, pp. 449–456.

[35] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 3, pp. 378–396, May 2012.

[36] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation.* Chicago, IL, USA: ACM, 2005, pp. 15–26.

[37] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "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.* Lisbon, Portugal: ACM, 2005, pp. 286–295.

[38] Z. Zhang, W. K. Chan, T. H. Tse, Y. T. Yu, and P. Hu, "Non-parametric statistical fault localization," *Journal of Systems and Software*, vol. 84, no. 6, pp. 885–905, Jun. 2011.

[39] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *20th IEEE/ACM International Conference on Automated Software Engineering.* Long Beach, CA, USA: ACM, 2005, pp. 263–272.

[40] A. Zeller, "Isolating cause-effect chains from computer programs," in *10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Charleston, South Carolina, USA, 2002, pp. 1–10.

[41] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *International Symposium on Software Testing and Analysis (ISSTA)*, July 2008, pp. 167–178.

[42] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *28th International Conference on Software Engineering.* New York, NY, USA: ACM, 2006, pp. 272–281.

[43] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Seventh International Conference on Software Testing, Verification, and Validation (ICST)*, Cleveland, USA, March 2014, pp. 153–162.

[44] A. Motro, "Seave: A mechanism for verifying user presuppositions in query systems," *ACM Trans. Inf. Syst.*, vol. 4, no. 4, pp. 312–330, Dec. 1986.

[45] ——, "Flex: a tolerant and cooperative user interface to databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 2, pp. 231–246, 1990.

[46] R. A. Demillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," pp. 34–41, 1978.

[47] P. Vemasani, A. Brodsky, and P. Ammann, "Generating test data to distinguish conjunctive queries with equalities," in *7th International Conference on Software Testing, Verification and Validation Workshops*, Cleveland, Ohio, USA, 2014, pp. 216–221.

[48] N. Li, Y. Lei, H. R. Khan, J. Liu, and Y. Guo, "Applying combinatorial test data generation to big data applications," in *31st IEEE/ACM International Conference on Automated Software Engineering.* Singapore, Singapore: ACM, 2016, pp. 637–647.

[49] A. Brodsky, C. Domeniconi, and D. Etter, "Regression databases: Probabilistic querying using sparse learning sets," in *5th International Conference on Machine Learning and Applications (ICMLA'06)*, Orlando, FL, USA, 2006, pp. 123–128.

[50] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[51] W. E. Wong, R. Go, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transaction on Software Engineering*, vol. 42, pp. 707–740, August 2016.

[52] Y. Wang, D. DeWitt, and J.-Y. Cai, “X-diff: An effective change detection algorithm for XML documents,” in *19th International Conference on Data Engineering*, Bangalore, India, April 2003, pp. 519– 530.

Curriculum Vitae

Yun Guo is a Ph.D. candidate in the Department of Computer Science of Volgenau School of Engineering at George Mason University. She received her M.S. in Computer Sciences from George Mason University in 2011. Before that, she received a B.E. in Electronic Engineer from Xi'an JiaoTong University of China in 2008. Her research interests include software testing, database language, and virtual enterprise. Her advisors are Dr. Jeff Offutt and Dr. Amihai Motro.