Test Case Generation for Database Applications

by

Arup Kumar Sarker

A thesis submitted in partial fulfilment of the requirements
for the degree of
Doctor of Philosophy

in the
Centre for Computing and Engineering Software Systems
School of Software and Electrical Engineering
Faculty of Science, Engineering and Technology

2015
Abstract

SQL statements are widely used in the database application and are essential part of database testing. In development stage of a database application, SQL statements may be used incorrectly because of syntactic changes; for example, use of wrong relational operators or join operators. These syntactic changes of SQL statements are termed mutants of the original SQL statements. In order to determine the correctness of SQL statements, mutation testing can play an important role by checking the correctness of test cases. Test cases (i.e. dataset) generated through a proper guideline can lead to find faults in SQL statements. An appropriate guideline can enhance the effectiveness of the test cases. A number of researches have been conducted to generate test cases targeting different types of testing for database application. Among various types of database application testing, one of the popular testing strategy is SQL mutation testing. Recently, many researchers have proposed several guidelines to generate test cases for different SQL mutation operators. But as there is a wide range of SQL mutation operators, these guidelines do not cover all types of mutation operators.

In this thesis, I propose a number of guidelines to generate test datasets for checking the faults in the SQL statement. These test datasets can distinguish the results of different mutants from its original SQL statements. The guidelines are proposed to address three mutation operators, namely absolute value (ABS), identifier column replacement (IRC), and identifier constant replacement (IRT) mutation operators. These mutation operators are applied on a number of selected SQL statements from NIST to generate all possible types of mutant. Then test datasets are generated according to the corresponding guidelines and these test datasets are executed against the original SQL statements and those mutants to verify the results.

There are six types of mutants generated by ABS mutation operator. But in some cases for a particular type of mutant there are several variants. Altogether there are 11 variants. For each of these variants, I propose one guideline to generate appropriate test cases. In order to verify the effective of the guidelines, 19 original SQL statements are selected
from NIST test suite and 68 ABS mutants are generated. Among them 67 are non-equivalent mutants. The test dataset generated through the proposed guidelines kill all 67 non-equivalent mutants. Similarly, for IRC mutation operator there are twenty four types of mutants with altogether 44 variants. For each of these variants, one guideline is proposed. To verify the effectiveness of the guidelines 41 original SQL statements are selected and IRC mutation operator produces 249 IRC mutants. Among them 23 mutants are equivalent. The test dataset generated through the proposed guidelines kill all 226 non-equivalent mutants. Finally, for IRT mutation operator there are sixteen types of mutants with altogether 32 variants. Same as above, one guideline is proposed for each of these variants. About 37 original SQL statements are selected and in total 174 IRT mutants are generated. Among them there are 9 equivalent mutants. Test dataset generated through those guidelines kill all 165 non-equivalent mutants.
Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Man Fai Lau for his continuous support and excellent guidance for my completion of PhD and his wonderful patience and kindness as a mentor.

I also would like to sincerely thank to Prof Tsong Yueh Chen and my associate supervisor Dr. Antony Tang for their wonderful cooperation to achieve scholarship in Swinburne University and Technology. Many thanks to Prof Yun Yang, Dr. Rajesh Vasa, Dr. Irene Moser for their constructive suggestions.

I would also like to thank my friends/colleagues Dr. Shimul Kumar Nath, Dr. Tanay Paul, Dr. Shibli Saleheen, Dr. Saiful Islam, Dr. Ashad Kabir, Dr. ASM Kayes, Dr. Samiran Mahmood, Dr. Tanjila Kanij, Dr Imrul Hassan, Dr Robert Setiadi, Mahmuda Ashrafi, and Sheikh Sadi for their valuable discussions and help on many occasions.

I would also like to express my gratitude to my parents Sunil Chandra Sarker and Kamala Rani Sarker, my brother Shourav Kumer Sarker, my friends Shaon Debnath and Kowshik Brata Paul for their continual advices, suggestions, and encouragements through this journey.

Last, but not least, my deepest gratitude goes to my wife, Mitu Dhali who sacrificed her own career ambition to make the PhD journey smooth for me. I would never have finished this journey without her supports and motivation.
Declaration of Authorship

I, Arup Kumar Sarker, certify that all materials contained in this thesis are my own and was conducted under the supervision of Dr. Man Fai Lau. All references of other people are properly cited.

Signed: 

Date:
Contents

Abstract iii

Acknowledgements v

Declaration of Authorship vii

List of Figures xiii

List of Tables xv

Notation and Terminology xvii

1 Introduction 1

1.1 Database testing ........................................... 1

1.1.1 Black-box Testing ..................................... 3

1.1.2 White-box Testing ..................................... 3

1.2 Research Problem and Motivation ..................... 4

1.3 Related Work ............................................... 5

1.4 The Approach of This Thesis ............................ 6

1.5 Contributions ............................................. 7

1.6 Thesis Organization .................................... 7

2 Literature Review 9
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Mutation Testing in Traditional Programs</td>
<td>9</td>
</tr>
<tr>
<td>2.2 SQL Mutation Operators</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Test Case Generation for Database Application</td>
<td>19</td>
</tr>
<tr>
<td>2.4 Test Case Generation for SQL</td>
<td>20</td>
</tr>
<tr>
<td>2.4.1 Constraint based Test Case Generation</td>
<td>20</td>
</tr>
<tr>
<td>2.4.2 Coverage Based Test Case Generation</td>
<td>20</td>
</tr>
<tr>
<td>2.4.3 Mutation based Test Case Generation</td>
<td>21</td>
</tr>
<tr>
<td>2.5 Summary</td>
<td>22</td>
</tr>
<tr>
<td>3 Test Case Generation for ABS Operator</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Motivation</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Absolute Value (ABS) Operator</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Six Types of ABS Mutants</td>
<td>25</td>
</tr>
<tr>
<td>3.4 Test Case Generation Guidelines</td>
<td>28</td>
</tr>
<tr>
<td>3.5 Experimental Study</td>
<td>40</td>
</tr>
<tr>
<td>3.5.1 Subject SQL Statements</td>
<td>40</td>
</tr>
<tr>
<td>3.5.2 Mutant Generation</td>
<td>41</td>
</tr>
<tr>
<td>3.5.3 Test Dataset Generation Procedure</td>
<td>41</td>
</tr>
<tr>
<td>3.5.4 Experimental Process and Data Collection</td>
<td>42</td>
</tr>
<tr>
<td>3.5.5 Results Analysis</td>
<td>42</td>
</tr>
<tr>
<td>3.6 Threats and Validity</td>
<td>43</td>
</tr>
<tr>
<td>3.7 Summary</td>
<td>44</td>
</tr>
<tr>
<td>4 Test Case Generation for IRC Operator</td>
<td>45</td>
</tr>
<tr>
<td>4.1 Identifier Column Replacement (IRC) Operator</td>
<td>45</td>
</tr>
<tr>
<td>4.2 Twenty Four Types of IRC Mutants</td>
<td>46</td>
</tr>
<tr>
<td>4.2.1 Original SQL without Aggregate Function and WHERE Clause</td>
<td>50</td>
</tr>
<tr>
<td>4.2.2 Original SQL with Aggregate Function but without WHERE clause</td>
<td>50</td>
</tr>
<tr>
<td>4.2.3 Original SQL with WHERE Clause but without Aggregate Function</td>
<td>51</td>
</tr>
<tr>
<td>4.2.3.1 IRC in SELECT Clause</td>
<td>51</td>
</tr>
<tr>
<td>4.2.3.2 IRC in WHERE Clause</td>
<td>53</td>
</tr>
</tbody>
</table>
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.4</td>
<td>Original SQL with \texttt{WHERE} Clause and Aggregate Functions</td>
<td>54</td>
</tr>
<tr>
<td>4.2.4.1</td>
<td>\texttt{IRC} in Aggregate Function in \texttt{SELECT} Clause</td>
<td>54</td>
</tr>
<tr>
<td>4.2.4.2</td>
<td>\texttt{IRC} in \texttt{WHERE} Clause</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Test Case Generation Guidelines</td>
<td>56</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Guidelines for Mutants from SQL without Aggregate Function and \texttt{WHERE} clause</td>
<td>57</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Guidelines for Mutants from SQL with Aggregate Function but without \texttt{WHERE} clause</td>
<td>58</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Guidelines for Mutants From SQL without Aggregate Function but with \texttt{WHERE} clause</td>
<td>62</td>
</tr>
<tr>
<td>4.3.3.1</td>
<td>\texttt{IRC} in the \texttt{SELECT} clause</td>
<td>63</td>
</tr>
<tr>
<td>4.3.3.2</td>
<td>\texttt{IRC} in the \texttt{WHERE} clause</td>
<td>69</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Guideline for Mutants from SQL with both Aggregate Function and \texttt{WHERE} Clause</td>
<td>73</td>
</tr>
<tr>
<td>4.3.4.1</td>
<td>\texttt{IRC} in Aggregate Functions of the \texttt{SELECT} Clause</td>
<td>74</td>
</tr>
<tr>
<td>4.3.4.2</td>
<td>\texttt{IRC} in the \texttt{WHERE} Clause</td>
<td>95</td>
</tr>
<tr>
<td>4.4</td>
<td>Experimental Study</td>
<td>123</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Subject SQL Statements</td>
<td>123</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Mutant Generation</td>
<td>124</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Test Dataset Generation and Data Collection</td>
<td>124</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Results Analysis</td>
<td>125</td>
</tr>
<tr>
<td>4.5</td>
<td>Threads and Validity</td>
<td>126</td>
</tr>
<tr>
<td>4.6</td>
<td>Summary</td>
<td>127</td>
</tr>
<tr>
<td>5</td>
<td>Test Case Generation for \texttt{IRT} Operator</td>
<td>129</td>
</tr>
<tr>
<td>5.1</td>
<td>Identifier Constant Replacement (\texttt{IRT}) Operator</td>
<td>129</td>
</tr>
<tr>
<td>5.2</td>
<td>Sixteen Types of \texttt{IRT} Mutants</td>
<td>130</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Original SQL without Aggregate Function and \texttt{WHERE} clause</td>
<td>131</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Original SQL with Aggregate Function but without \texttt{WHERE} clause</td>
<td>134</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Original SQL with \texttt{WHERE} clause but no Aggregate Function</td>
<td>135</td>
</tr>
</tbody>
</table>
5.2.3.1 IRT in the SELECT clause ........................................... 135
5.2.3.2 IRT in the WHERE clause ........................................... 137
5.2.4 Original SQL with WHERE clause and Aggregate Function ........... 137
  5.2.4.1 IRT in Aggregate Function in SELECT clause ................... 138
  5.2.4.2 IRT in the WHERE clause ........................................... 139
5.3 Test Case Generation Guidelines ........................................ 140
  5.3.1 Guidelines for Mutants from SQL without Aggregate Function and
       WHERE clause .......................................................... 140
  5.3.2 Guidelines for Mutants from SQL with Aggregate Function but with- 
       out WHERE clause ...................................................... 142
  5.3.3 Guidelines for Mutants from SQL without Aggregate Function but 
       with WHERE clause ..................................................... 148
    5.3.3.1 IRT in the SELECT clause ...................................... 148
    5.3.3.2 IRT in the WHERE clause ........................................... 153
  5.3.4 Guidelines for Mutants from SQL with both Aggregate Function and 
       WHERE clause .......................................................... 156
    5.3.4.1 IRT in Aggregate Function of the SELECT clause .......... 156
    5.3.4.2 IRT in the WHERE clause ........................................... 174
5.4 Experimental Study ....................................................... 200
  5.4.1 Subject SQL Statements ............................................. 200
  5.4.2 Mutant Generation .................................................... 200
  5.4.3 Test Dataset Generation and Data Collection ....................... 201
  5.4.4 Results Analysis .................................................... 202
5.5 Threads and Validity .................................................... 202
5.6 Summary ................................................................. 203
6 Conclusion ................................................................. 205
  6.1 Summary and Contributions ........................................... 205
  6.2 Future Works ............................................................ 206

Bibliography ............................................................... 209
List of Figures

2.1  SQLMutation web interface ........................................ 12
2.2  SQLRules Standalone java application .......................... 13
3.1  Table Schema of STAFF ........................................... 24
3.2  Flow Diagram for generating ABS Mutants .................... 26
4.1  Table Schema of WORKS ......................................... 124
5.1  Table Schema VTABLE ............................................ 200
List of Tables

2.1 Example of Code Mutation ................................................. 10
2.2 SQL Mutation Operators by Tuya[1] ................................. 11
2.3 Mutation Operators by Dhuhan [2] ................................. 13
2.4 Mutation Operators by Cabeca [3] ................................. 15
2.5 Mutation Operators by Chan and Tse [4] ......................... 18

3.1 Different Types of \textit{ABS()} Mutants .......................... 27
3.2 Test Case Generation Guidelines for \textit{ABS} mutants .......... 29
3.3 Number of Records satisfying “SELECT salary FROM STAFF WHERE \textit{condition}” with \{39, 40, 41, −39, −40, −41\} in the “grade” Field .... 34
3.4 Number of records returned by “SELECT AGGR(grade) FROM STAFF WHERE \textit{condition}” ............................................. 38
3.5 Number of Mutants for each Type Against Original SQL .... 42
3.6 Results of killed Mutants according to Guidelines ......... 43

4.1 Possible Types of Original SQL Statement ..................... 47
4.2 Different Types of \textit{IRC} Mutant .................................. 48
4.3 Examples of Various \textit{IRC} Mutants .............................. 49
4.4 Dataset for IRC-m5 Mutants in STAFF ......................... 64
4.5 Result showing the SQL Statements with \textit{COUNT(DISTINCT ...)} ............................... 76
4.6 Dataset that can kill the Mutants .............................. 97
4.7 Dataset that can kill the Mutants ............................ 100
4.8 Dataset that can kill the Mutants ................................................... 103
4.9 Dataset that can kill the Mutants ................................................... 107
4.10 Test Case Generation Guidelines for IRC Mutants ............................. 109
4.11 Number of Mutants for IRC ......................................................... 125
4.12 Number of Killed and Alive Mutants for IRC (only shows the available
mutants) ......................................................................................... 126

5.1 Possible Types of Original SQL Statements ...................................... 131
5.2 Different Types of IRT Mutant ........................................................ 132
5.3 Examples of Various IRT Mutants ................................................... 133
5.4 Dataset for IRT-m5 Mutants in STAFF ............................................. 149
5.5 Dataset that can kill the Mutants ..................................................... 176
5.6 Dataset that can kill the Mutants ..................................................... 179
5.7 Dataset that can kill the Mutants ..................................................... 182
5.8 Dataset that can kill the Mutants ..................................................... 185
5.9 Test Case Generation Guidelines for IRT Mutants ............................. 187
5.10 Number of Mutants for IRT ............................................................ 201
5.11 Number of Killed and Alive Mutants for IRT (Only shows the available
mutants) ......................................................................................... 202
Notation and Terminology

ABS  Absolute Value mutation operator
IRC  Identifier Column Replacement mutation operator
IRT  Identifier Constant Replacement mutation operator
BVA  Boundary Value Analysis
f1,f2, . . .  The fields in a database table
C1, C2, . . .  Numeric constants
S1, S2, . . .  String Constants
SQL  Structured Query Language
Dedicated to my parents and my wife
Chapter 1

Introduction

One of the major area of database application testing is SQL mutation testing. Many re-
searchers have provided various definitions of mutation operators and test case generation
guidelines. For instance, Tuya and his colleagues [1] provide a range of SQL mutation
operators for SQL statements. Several other studies [6, 7] address test case generation
guidelines for some of these mutation operators. However, they did not cover all the possi-
bile mutants that can be generated from SQL statements. For example, a single mutation
operator may have many complex scenarios to be considered for mutation operators, how-
ever, no such mutation operator has been introduced in the literature to the best of my
knowledge. Hence, it is necessary to introduce guidelines for those complex scenarios that
have not been covered in earlier research. This thesis therefore aims to propose test case
generation guidelines for complex scenarios in relation to previously unexamined mutation
operators.

1.1 Database testing

A database is a collection of information stored in an organized way to provide efficient
retrieval and manipulation. The collected information could be in different formats (elec-
tronic, printed, graphic, audio, statistical, combinations). Based on these different data
formats, databases are classified into two main categories - physical (e.g. paper/print) and
electronic databases (e.g. a digital library, online dictionary, wikipedia etc.). In the current digital era, electronic databases are widely used among the organizations. The data is organized in database through tables, views etc. To organize these electronic data, a computer program is used which is often referred to as database application. Database applications basically helps in the efficient collection, storage, manipulation and retrieval of those data. It is also used to create, modify, and remove database objects (e.g. a database table, view or schema). Online libraries, eBay, FaceBook etc., are common examples of database applications.

There is a wide range of database applications which are used in many places for various purposes. Chan and Chung [8] classified them into two major categories- (1) application that are solely built in Data Manipulation Language (DML) and the language supported by Database Management System (DBMS), (2) applications that are build in both DML and general purpose programming language such as C, C++, Java, C#, PHP, and ASP.NET [9]. Database is the information storage for an organization and database application is the interface for storing and accessing the data to and from the database. Database is a valuable asset for an organization and the database application is most often mission-critical in nature. Database applications may be changed over time, old applications may be thrown away and new applications may be taken over, however, a database is never thrown away; it is always preserved and migrated carefully. Nevertheless, the error free database application is very crucial as the business process is greatly dependent on the output of the database applications. As a consequence, software engineers always perform quality assurance activities before deploying a database application. Like other software artifacts, software testing is a popular means of quality assurance activities for database applications. In database application testing, the user inputs are considered as test cases and selecting the correct test cases is one of the challenging task in testing those applications. There are two main types of testing used in database testing- black-box testing and white-box testing.
1.1.1 Black-box Testing

Black-box testing focuses on the input and output without knowing the details of the internal code of a database application. In this testing programmers and testers are independent to each other. As most of the time it is performed from user perspective, it reveals faults related to software functionalities, specifications or requirements. There are many techniques [10] of black-box testing such as equivalence partitioning, boundary value analysis, comparison testing, orthogonal array testing. These techniques are also applicable for a database application. More specifically, a black-box testing can perform the following actions for a database application- a) mapping of data (including metadata), b) verifying incoming data, c) verifying outgoing data from query functions, and d) various techniques such as effect graphing technique, e) equivalence partitioning and f) boundary-value-analysis.

1.1.2 White-box Testing

White-box testing is a traditional test case design method [10] that covers several testing features such as statement testing, branch testing, condition testing, path testing etc. In white-box testing testers test detail of the program code and ensure that (a) all independent paths have been covered at least once, (b) all logical decisions have been exercised, (c) all loops are executed, and (d) validation of internal data structure is exercised. Chan and Cheung [8] show that traditional white-box testing can also be applied in database testing involving the following criteria: i) it involves testing of database triggers and logical views which are going to support database refactoring, ii) it performs module testing of database functions, triggers, views, SQL queries etc, iii) it validates database tables, data models, database schema etc, iv) it checks rules of referential integrity, v) it selects default table values to check on database consistency, and vi) the techniques used in white box testing are condition coverage, decision coverage, statement coverage, cyclomatic complexity.

There are many testing techniques of white-box testing. Embedded testing is one of them. At the beginning embedded testing was used for conventional program. Later, many researches[11, 12] apply embedded testing on database applications specially for testing
Chapter 1. Introduction

embedded SQL statement. Embedded SQL is a method of combining the programming language with Structured Query Language (SQL). This embedded SQL is written inline with the programming source code of the host language such as C, C++, java, C#, and ASP.NET. An embedded SQL processor parses this embedded SQL and passes it to a database engine to perform the specific database task (e.g. data manipulation, create, update, delete, and many other functions).

In mutation testing, an original program is modified in a controlled way to generate mutants by adding small syntactic changes to the original program [13–15]. Test cases are applied on these mutants to check whether the outputs of the original program are different from the mutated program. If the outputs are different, it is said that the test case kills the mutant. A modified program, which is called a mutated program, or simply a mutant of the original program. The small syntactic changes are usually perceived as the mutation operators [16, 17]. In case the outputs of the original program and one of its mutant are all the same for every input in the input domain, the mutant is said to be equivalent to the original program. Mutation testing can be applied in conventional programming [18,19] and database application [20]. In database application, mutation testing can be performed on SQL statements [1,2,5]. This thesis specifically focuses on SQL mutation testing.

1.2 Research Problem and Motivation

Nowadays the uses of data or information is getting very popular. It is being used in most industries such as software firms, banks, insurance companies, education, research and development, manufacturing, road and transportation. A company needs a proper database application to store, manage and retrieve its data. And SQL statements play the most essential and sophisticated role in database application by creating database objects (e.g. table, schema), populating data (INSERT), modifying data (UPDATE), and retrieving data (SELECT). Among them retrieving data (SELECT SQL statement) is the mostly frequent statement used in database application [21].
During the development of SQL statements, different types of errors may occur such as semantic error and syntactic error [22]. A database engine detects syntactic errors easily. But semantic error are not picked by the database engine. For example, a SQL statement is “SELECT staffname FROM STAFF WHERE staffid = 200 AND staffid = 500” (assuming staffid is primary key). Although this query is syntactically correct, we can easily understand that a staff member does not have two different staffids, as a result this statement will always return null. Again, some syntactic changes such as use of wrong relational operator (e.g. $>$, $<$, $\geq$, $\leq$, $=$, $\neq$ ) or join operator may occur in SQL statement which are syntactically correct (i.e. no database error) but may lead to an incorrect output. Some earlier studies shows that among the SQL developers, the percentage of using incorrect SQL statements is from 12% to 75% [23, 24]. Although, to the best of my knowledge, there is no recent statistics of the error rate of SQL statements, it is understandable that a substantial amount of errors still exist in implementing the SQL statement. This costs a lot of time and money to the industry. Hence, ensuring the correctness of SQL statement in developing database applications is very important.

Realizing the necessity of SQL statement testing, researchers [20, 25, 26] focus different testing areas such as test case generation, test data preparation, and test execution and output verification [8]. Among them test case generation is one of the most important and complex issue. In this thesis, I am investigating test case generation guidelines for special SQL mutation operators so as to ensure that the test cases generated can kill all related mutants.

1.3 Related Work

In this section, I briefly mention some related works on testing SQL. A more detailed discussions of these works can be found in Chapter 2. Various research studies in testing SQL statements focused on SQL coverage testing [1, 12, 27–31]. Researchers also propose test case generation criteria in different studies [32–41]. They develop algorithm and tools for database application in different languages to track symbolic constraints.
They also use stored procedure to generate database instances for SQL statements embedded in programs. However, they have not considered mutation testing. Other researchers [1–7, 28, 29, 42–44] consider SQL mutation in database testing. Among them most prominently, Shetal et al. [6, 7] address some guidelines for test case generation in SQL mutation testing. They propose guidelines for particular types of mutation operators for SQL queries: join/outer join mutation operator, comparison mutation operator, and aggregation mutation operator. Besides these there are a number of mutation operators [1] which they did not address in their studies i.e. absolute value mutation operator, identifier column replacement, and identifier constant replacement mutation operators.

1.4 The Approach of This Thesis

In this thesis, I adopt a fault-based approach to develop test case generation guidelines for three different mutation operators - absolute value (ABS), identifier column replacement (IRC) and identifier constant replacement (IRT). The test case generation guidelines for these mutation operators have not been considered by previous research.

- **ABS**: There are altogether six different types of ABS mutants with 11 different variants. For each of these variants, I propose a test case generation guideline. In many cases the guideline suggests to use some positive and negative values in a dataset. For some other cases guideline proposes to use boundary value analysis (BVA) and sometimes it states to use positive-negative values together in one dataset or a different dataset with or without BVA values.

- **IRC**: Similar to ABS, IRC mutation operator is applied on SQL statements to generate twenty-four different types of mutants with altogether 44 variants. For each of these variants the test case generation guideline is proposed. For most of the scenarios the strategy requires BVA with some different arrangements of the dataset. Furthermore, in some cases it needs to apply BVA and reverse-BVA\(^1\).

\(^1\)Reverse-BVA is discussed in Chapter 4
IRT: Similar to IRC, IRT mutation operators are used in different types of SQL queries in different positions to generate sixteen types of mutants with altogether 32 variants. For each of these variants, a test case generation guideline is proposed. Depending on the characteristics of original SQL statements and its mutants; the guideline suggests to apply BVA, reverse-BVA, and different distinct values to generate test cases. In some cases some special arithmetic calculations are applied to generate the mutants.

1.5 Contributions

The contributions made in this thesis are given as follows:

- I illustrate the importance of using ABS, IRC and IRT mutation operators in SQL statements. I demonstrate how these mutation operators can be used in different positions of the SQL statement to generate different types of mutants and why the test generation guideline is important for these mutation operators.

- I provide the guidelines to generate test cases for ABS, IRC and IRT mutants to find faults in SQL statements generated from these three mutation operators.

- I conduct a number of experiments using the SQL statements found in NIST[79] conformance test suite to demonstrate the effectiveness of the proposed test generation guidelines.

1.6 Thesis Organization

Chapter-2 reviews the literature of test case generation for database application and mutation testing. I also find the existing gap in the test case generation guidelines for detecting SQL related mutants.

Chapter-3 defines all types of mutant that can be created by ABS mutation operator. For each type of mutant, test generation guidelines are proposed. An experiment
is conducted to measure the effectiveness of the test case generation guidelines for detecting ABS mutants.

Chapter-4 defines all types of mutant that can be created by IRC mutation operator. For each type of mutant, test generation guidelines are proposed. An experiment is conducted to measure the effectiveness of the test case generation guidelines for detecting IRC mutants.

Chapter-5 defines all types of mutant that can be created by IRT mutation operator. For each type of mutant, test generation guidelines are proposed. An experiment is conducted to measure the effectiveness of the test case generation guidelines for detecting IRT mutants.

Chapter-6 concludes the thesis with related future works.
2.1 Mutation Testing in Traditional Programs

Mutation testing is a popular technique for the fault based technique that is first introduced by Richard Lipton [13] in 1971. However, DeMillo and colleagues [14] publish the first major task on mutation testing in 1978. In mutation testing, an original program is modified in a controlled way to generate mutants by adding small syntactic changes to the original program. Test cases are applied on these mutants to check whether the outputs of the original program are different from the mutated program. If the outputs are different, it is said that the test case kills the mutant. A modified program, which is called a mutated program, or simply a mutant of the original program. The small syntactic changes are usually perceived as the mutation operators. In case the outputs of the original program and one of its mutant are all the same for every input in the input domain, the mutant is said to be equivalent to the original program. Table 2.1 shows a real example (a code segment of a C program) of an original program and one of its mutant. In this example, the logical OR operator “||” in the if condition, the original program has been replaced by the logical AND operator ‘&&’. for the test case where a=1 and b=0, the original program p returns 1 whereas the mutant \( \tilde{p} \) returns 0. As a result this test case distinguishes the mutant from its original program.
Table 2.1: Example of Code Mutation

<table>
<thead>
<tr>
<th>Original Program ( p )</th>
<th>Mutated Program ( \hat{p} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (( a \lor b )) {</td>
<td>if (( a \land b )) {</td>
</tr>
<tr>
<td>( c = 1; )</td>
<td>( c = 1; )</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>else{</td>
<td>else{</td>
</tr>
<tr>
<td>( c = 0; )</td>
<td>( c = 0; )</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>return ( c );</td>
<td>return ( c );</td>
</tr>
</tbody>
</table>

Mutation testing has been applied in many programming languages such as C [18, 19, 45–48], JAVA [49–56], C# [57–61], FORTRAN [62–66] etc. Agrawal and his colleagues [45] provide a large set of mutation operators for ANSI C programming language. Delamaro et al. [67] extend the concept of mutation testing at the inter-procedural level providing mechanism for the evaluation of integration testing. Offut and his colleagues [16] suggest a set of mutation operators for integration testing although their main objectives are to compare selective mutation with non-selective mutation. DeMillo his colleagues [17] propose a mutation testing system named *Mothra*, which uses 22 mutation operators for FORTRAN language.

### 2.2 SQL Mutation Operators

There are many studies in the mutation operator of SQL. In the following, I review these studies one at a time.

Tuya and his colleagues [1] propose a set of mutation operators for SQL queries that cover a wide spectrum of SQL features. They classify mutation operators in four categories namely, (1) mutations for the main SQL clause (SC), (2) mutations for the operators that are present in conditions and expressions (OR), (3) mutations related to the handling of NULL values (NL), and (4) replacement of identifiers (IR). These four major categories are divided into 19 sub-categories. Table 2.2 illustrates a short description of each of these mutation operators. They also implement a tool, SQL Mutation [29] to generate SQL mutants automatically. This tool takes database schema and SQL query as input (shown in Figure 2.1) and produces SQL mutants using their proposed mutation operators. This tool is publicly available on the web and accessible in two different interfaces- a Web
Table 2.2: SQL Mutation Operators by Tuya\[1\]

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SC Category</strong></td>
<td></td>
</tr>
<tr>
<td>SEL (Select clause)</td>
<td>Replacement of Select/ Select Distinct</td>
</tr>
<tr>
<td>JOI (Join clause)</td>
<td>Replacement of JOIN type keywords</td>
</tr>
<tr>
<td>SUB (Sub query predicates)</td>
<td>Replacement of predicates in eRp(Q)</td>
</tr>
<tr>
<td>GRU (Groupings)</td>
<td>Removal of group-by-expressions</td>
</tr>
<tr>
<td>AGR (Aggregate functions)</td>
<td>Replacement of aggregate functions</td>
</tr>
<tr>
<td>UNI (Query concatenation)</td>
<td>Replacement of UNION keywords and removal of participating queries in UNION</td>
</tr>
<tr>
<td>ORD (Ordering of the result set)</td>
<td>Change in direction and removal of order-by-expressions</td>
</tr>
<tr>
<td><strong>OR Category</strong></td>
<td></td>
</tr>
<tr>
<td>ROR (Relational Op. Replacement)</td>
<td>Replacement of relational operators</td>
</tr>
<tr>
<td>LCR (Logical Connector Replacement)</td>
<td>Replacement of logical operators</td>
</tr>
<tr>
<td>UOI (Unary Operator Insertion)</td>
<td>Replacement in arithmetic expression or reference to a number e by e, e+1, e-1</td>
</tr>
<tr>
<td>ABS (Absolute Value Insertion)</td>
<td>Replacement in arithmetic expression or reference to a number e by abs(e) and -abs(e)</td>
</tr>
<tr>
<td>AOR (Arithmetic Op. Replacement)</td>
<td>Replacement of arithmetic operators</td>
</tr>
<tr>
<td>BTW (Between Predicate)</td>
<td>Replacement in conditions like a BETWEEN x AND y</td>
</tr>
<tr>
<td>LKE (Like predicate)</td>
<td>Replacement and removal of wildcard characters in a LIKES</td>
</tr>
<tr>
<td><strong>NL Category</strong></td>
<td></td>
</tr>
<tr>
<td>NLF (Null check predicates)</td>
<td>Replacement of NULL keyword</td>
</tr>
<tr>
<td>NLS (Null in select list)</td>
<td>Replacement each column reference c in the select-list by a function ifnull (c, r)², that substitute a value c by r (r is replacement value outside of the domain c.)</td>
</tr>
<tr>
<td>NLI (Include Nulls)</td>
<td>Forces the true value of the condition when value is null</td>
</tr>
<tr>
<td>NLO (other nulls)</td>
<td>Completes the other combinations of nulls</td>
</tr>
<tr>
<td><strong>IR Category</strong></td>
<td></td>
</tr>
<tr>
<td>IRC (column replacement)</td>
<td>Replacement of column references by other column references, constants, parameters</td>
</tr>
<tr>
<td>IRT (constant replacement)</td>
<td>Replacement of constant by other constants, columns and parameters</td>
</tr>
<tr>
<td>IRP (parameter replacement)</td>
<td>Replacement of query parameter reference by other parameters, constants and columns</td>
</tr>
<tr>
<td>IRH (hidden column replacement)</td>
<td>Replacement of column attribute reference by other columns</td>
</tr>
</tbody>
</table>
application to interactively generate the mutants and a Web service that allows it to be integrated with other applications developed using different platforms. They also create a standalone JAVA application named 'SQLRules' (shown in Figure 2.2). They execute those created mutants against a given test dataset to determine the mutation score. The higher percentage of mutation score ensures the adequacy of test dataset. In their study they emphasize on defining the mutation operators. In my study I assert on developing guidelines to generate test dataset because if I can make a proper test dataset rather than
random dataset, I will be able to ensure higher rate of mutation score.

Dhuhan [2] suggest a wider variety of mutation operators, which cover various features of SQL queries. He points out several functions of SQL queries that Tuya and his colleagues [29] do not emphasize. These SQL features include insertion, creation of table, searching range, scalar function, string function, conversion function, date function, group-by-clause, and creating index. He concentrates on defining the mutation operators rather than generating guidelines. Table 2.3 summarizes his proposed mutation operators.

Table 2.3: Mutation Operators by Dhuhan [2]

<table>
<thead>
<tr>
<th>Category</th>
<th>Abbr</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>II (Insert Into)</td>
<td>RNE</td>
<td>Reducing number of expressions</td>
</tr>
<tr>
<td></td>
<td>INE</td>
<td>Increasing number of expressions</td>
</tr>
<tr>
<td>Category</td>
<td>Abbr</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CT</td>
<td>AWC</td>
<td>Add <strong>WHERE</strong> clause if it doesn’t exists in from part of command</td>
</tr>
<tr>
<td>(create table)</td>
<td>RWC</td>
<td>Remove where clause if it exists and further change in conditions in where clause can be applied</td>
</tr>
<tr>
<td>SR (Searching Range)</td>
<td>RNB</td>
<td>Replace not between by between and vice versa</td>
</tr>
<tr>
<td></td>
<td>RNI</td>
<td>Replace not in by in and vice versa</td>
</tr>
<tr>
<td>NF (scalar function)</td>
<td>RGF</td>
<td>(Replace not in) by in and vice versa</td>
</tr>
<tr>
<td></td>
<td>TRF</td>
<td>Truncate, Round and floor can be interchanged with each other</td>
</tr>
<tr>
<td>SF (string function)</td>
<td>LUI</td>
<td>Lower, upper &amp; initcap can be interchanged with each other</td>
</tr>
<tr>
<td></td>
<td>RCD</td>
<td>Replace compose and decompose with each other</td>
</tr>
<tr>
<td></td>
<td>TLR</td>
<td>Interchange trim, ltrim and rtrim with each other</td>
</tr>
<tr>
<td></td>
<td>RLR</td>
<td>Replace lpad and rpad with each other</td>
</tr>
<tr>
<td>CF (Conversion fun.)</td>
<td>RNC</td>
<td>Replace <strong>to-number</strong> function with <strong>to-char</strong> function and vice versa</td>
</tr>
<tr>
<td>DF (Date function)</td>
<td>RDF</td>
<td>Replace each of TH, SP, &amp; SPTH with each other</td>
</tr>
<tr>
<td>MF (misc. function)</td>
<td>RUU</td>
<td>Replace UID by use and vice versa</td>
</tr>
<tr>
<td></td>
<td>RHC</td>
<td><strong>Having</strong> clause can be altogether removed from <strong>group by clause</strong> &amp; all other variations on <strong>having</strong> clause can be done</td>
</tr>
<tr>
<td>GC (group-by clause)</td>
<td>RGF</td>
<td>Replacement of group functions by other group functions in having clause of group</td>
</tr>
<tr>
<td></td>
<td>RHD</td>
<td>Replacement of <strong>having</strong> with <strong>distinct</strong> &amp; vice versa</td>
</tr>
<tr>
<td></td>
<td>RRC</td>
<td>Rollup &amp; cube operators can be replaced by each other</td>
</tr>
<tr>
<td></td>
<td>UIM</td>
<td>Replacement of union, intersection &amp; minus clauses on multiple queries</td>
</tr>
<tr>
<td>CI (create index)</td>
<td>ARU</td>
<td>Add <strong>Unique</strong> keyword if it doesn’t exists &amp; remove if it exists.</td>
</tr>
<tr>
<td></td>
<td>ARR</td>
<td>Add <strong>reverse</strong> keyword if it doesn’t exists &amp; remove if it exists.</td>
</tr>
<tr>
<td></td>
<td>ARB</td>
<td>Add <strong>bitmap</strong> keyword if it doesn’t exists &amp; remove if it exists.</td>
</tr>
<tr>
<td>GP (grant privileges)</td>
<td>IFS</td>
<td>Type of privileges comes from the set {Alter, delete, index, insert, select, update}. Each of these can be replaced with other in object privileges part of the command</td>
</tr>
</tbody>
</table>
Continued from previous page

<table>
<thead>
<tr>
<th>Category</th>
<th>Abbr</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDP</td>
<td></td>
<td>Number of privileges can be increased for some users &amp; for some can be decreased. DQ (data dictionary query) RD-KUSER_TAB_Columns &amp; USER_TYPE_ALTERS can be replaced by ALL_TAB_COLUMNS AND ALL_TYPE_ATTRS &amp; vice versa</td>
</tr>
</tbody>
</table>

Cabeca and his colleagues [3] suggest SQL mutation operator SQL in a different way. They group all the mutation operators in five categories. Table 2.4 shows the list of their proposed mutants.

Table 2.4: Mutation Operators by Cabeca [3]

<table>
<thead>
<tr>
<th>Category</th>
<th>Name of Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Operators</td>
<td>Math Operator Replacement (MaOpR); Comparison Operator Replacement (CoOpR); Conjunctive Operator Replacement (CjOpR); Logical Operator Replacement (LoOpR); Negative Operator Insertion (NeOpIn) and Negative Operator Exclusion (NeOpEx)</td>
</tr>
<tr>
<td>Miscellaneous  Operators</td>
<td>Atribute Position Replacement (AtPOr); Atribute Exclusion (AtEx); Atribute Insertion (AtIn); Atribute Replacement (AtR); Value Position Replacement (VaPoR); Value Replacement (VaR); Variable Type Replacement (VaTyR); Table Name Replacement (TaNaR); Role Name Replacement (RoNaR);</td>
</tr>
</tbody>
</table>

Continued on next page
Table 2.4 – Continued from previous page

<table>
<thead>
<tr>
<th>Category</th>
<th>Name of Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role Insertion (RoIn);</td>
<td></td>
</tr>
<tr>
<td>Role Exclusion (RoEx);</td>
<td></td>
</tr>
<tr>
<td>Cursor Name Replacement (CuNaR);</td>
<td></td>
</tr>
<tr>
<td>Aggregation Function Replacement (AgFuR);</td>
<td></td>
</tr>
<tr>
<td>Intersection Replacement (InR) and</td>
<td></td>
</tr>
<tr>
<td>Join Replacement (JoR)</td>
<td></td>
</tr>
<tr>
<td>Statement Block Replacement in conditional or repeat structures (CoBoR);</td>
<td></td>
</tr>
<tr>
<td>Repeat/Conditional Statement Block Exclusion (CoCoEx);</td>
<td></td>
</tr>
<tr>
<td>Repeat/Conditional Statement Block Insertion (CoCoIn);</td>
<td></td>
</tr>
<tr>
<td>Leave Position Replacement in statement blocks (LePoR);</td>
<td></td>
</tr>
<tr>
<td>Leave Exclusion (LeEx) and</td>
<td></td>
</tr>
<tr>
<td>Leave Insertion (LeIn)</td>
<td></td>
</tr>
<tr>
<td>COMMIT Insertion (CoIn);</td>
<td></td>
</tr>
<tr>
<td>COMMIT Exclusion (CoEx);</td>
<td></td>
</tr>
<tr>
<td>ROLLBACK Insertion (RoIn);</td>
<td></td>
</tr>
<tr>
<td>ROLLBACK Exclusion (RoEx);</td>
<td></td>
</tr>
<tr>
<td>ROLLBACK by COMMIT Replacement (RoCoR);</td>
<td></td>
</tr>
<tr>
<td>COMMIT by ROLLBACK Replacement (CoRoR);</td>
<td></td>
</tr>
<tr>
<td>SAVEPOINT Name Replacement (SaNaR);</td>
<td></td>
</tr>
<tr>
<td>Permission Replacement (PeR);</td>
<td></td>
</tr>
<tr>
<td>Privilege Replacement (PrR);</td>
<td></td>
</tr>
<tr>
<td>GRANT by REVOKE Replacement (GrReR);</td>
<td></td>
</tr>
<tr>
<td>REVOKE by GRANT Replacement (ReGrR) and</td>
<td></td>
</tr>
<tr>
<td>User Name Replacement (UsNaR)</td>
<td></td>
</tr>
<tr>
<td>Function/Procedure/View/Trigger Name Replacement (NaR);</td>
<td></td>
</tr>
<tr>
<td>Function Return Position Replacement (FuPoR);</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Chapter 2. Literature Review

Table 2.4 – Continued from previous page

<table>
<thead>
<tr>
<th>Category</th>
<th>Name of Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Function Return Replacement (FuReR):</td>
</tr>
<tr>
<td></td>
<td>Procedure Parameter Replacement -IN/OUT- (PaProR)</td>
</tr>
<tr>
<td></td>
<td>and</td>
</tr>
<tr>
<td></td>
<td>Trigger Event Replacement (TrEvR)</td>
</tr>
</tbody>
</table>

Chan and Tse [4] define seven types of mutation operators based on the standard types of constraint used in the enhanced entity-relationship model. Entity-relationship modeling is a database modeling method that produces a type of conceptual schema or semantic data model of a system, often a relational database, and its requirements in a top-down fashion. Diagrams created by this process are called entity-relationship diagrams or ER diagrams. Table 2.5 shows a list of all these operators.

Shahriar and Zulkernine [5] describe a different mutation testing strategy for SQL queries. Their main concern is database security. They consider an SQL injection attack as a threat to the SQL query. SQL injection attacks are a type of injection attack, where SQL commands are injected into the data-plane input in order to affect the execution of predefined SQL commands. They term this action as SQL Injection Vulnerabilities (SQLIVs). A successful SQL injection can read sensitive data from the database, can modify (Insert/Update/Delete) table data in a database, can recover the content of a given file present on the DBMS file system and in some cases, they can even issue commands to the operating system. They divide nine mutation operators into two categories. The first category consists of four operators that injected faults into the WHERE clause (WC) of SQL queries and the second category consists of five operators that injected faults into the database API method calls (AMC). Table 2.6 summarizes their proposed mutation operators.
Table 2.5: Mutation Operators by Chan and Tse [4]

<table>
<thead>
<tr>
<th>Semantic Operator</th>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participation constraint Replacement</td>
<td>PTCR</td>
<td>Toggle the participation requirements of entity types in the relation.</td>
</tr>
<tr>
<td>Cardinality Constraint Replacement</td>
<td>CDCR</td>
<td>Replace the cardinalities of entity types in the relation.</td>
</tr>
<tr>
<td>Identifying/Weak Entity Type Replacement</td>
<td>IWKR</td>
<td>Replace (an expression of) identifying type(s) by (an expression of) weak entity type(s), or vice versa.</td>
</tr>
<tr>
<td>attribute Replacement</td>
<td>ATTR</td>
<td>Replace (an expression on) attribute(s) by (an expression on) other attribute(s) of a compatible type.</td>
</tr>
<tr>
<td>Generalization / Specialization Completeness Replacement</td>
<td>GSCR</td>
<td>Replace an expression on a partial superclass by an expression on a subclass and the negation form of the superclass</td>
</tr>
<tr>
<td>Generalization / Specialization Disjointness Replacement</td>
<td>GSDR</td>
<td>Replace (an expression on) sibling entity type(s) by (an expression on) other sibling entity type(s) under the same superclass.</td>
</tr>
<tr>
<td>Union Type Completeness Replacement</td>
<td>UTRC</td>
<td>Replace an entity type by a subclass and/or superclasses of the subclass, such that these superclasses have the same union type constraint</td>
</tr>
</tbody>
</table>

Table 2.6: Mutation Operators by Shahriar and Zulkernine [5]

<table>
<thead>
<tr>
<th>Cat</th>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WC</td>
<td>RMWH</td>
<td>Remove WHERE keywords and conditions.</td>
</tr>
<tr>
<td></td>
<td>NEGC</td>
<td>Negate each of the unit expression inside where condition</td>
</tr>
<tr>
<td></td>
<td>FADP</td>
<td>Add parentheses in where conditions and prepend “FALSE AND” after the WHERE keyword</td>
</tr>
<tr>
<td></td>
<td>UNPR</td>
<td>Unbalance parenthesis of where condition expression</td>
</tr>
<tr>
<td>AMC</td>
<td>MQFT</td>
<td>Set multiple query execution flags to true</td>
</tr>
<tr>
<td></td>
<td>OVCR</td>
<td>Override commit and rollback options.</td>
</tr>
<tr>
<td></td>
<td>SMRZ</td>
<td>Set the maximum number of record returned by a result set to infinite</td>
</tr>
<tr>
<td></td>
<td>SQDZ</td>
<td>set the maximum number of record returned by a result set to infinite</td>
</tr>
<tr>
<td></td>
<td>OVEP</td>
<td>Override the escape character processing flags.</td>
</tr>
</tbody>
</table>
2.3 Test Case Generation for Database Application

Chays and his colleagues [20] develop a tool that satisfy the database constraints and populated database tables with meaningful data. The prototype has four components - (a) state generator, (b) input generator, (c) output verifier, and (d) state validator. One the basis of this prototype, Chays and his colleagues [32] develop a sophisticated tool set AGENDA that has five main components- (i) AGENDA parser, (ii) state generator, (iii) input generator, (iv) output validator and (v) state validator. Among these components input generator is responsible to generate the input data that was passed to the application. These input data are created based on the information by the AGENDA parser and state validator. But later it is found that the output produced by the input generator sometimes generates empty result (dataset) or violates the constraints. As a result Chays and his colleagues [33] develop an improved input generator that returns non-empty result (i.e. dataset), causes update or insert without violating the constraints and also satisfies different properties selected by the tester. Similar to AGENDA [32] there are a number of other commercial tools [68, 69] that produce datasets but sometimes return empty set. To solve this problem Reverse Query Processing (RQP) [34] is proposed. RQP takes a SQL query and a result as input and returns the possible dataset as an output.

Emmi et al. [36], develop an algorithm based on concolic execution [70, 71] that tracks symbolic constraints and uses a constraint solver to generate input data for the database application program and records for the database so that it can explore all the available paths of the program. Rather than any predefined guideline for test case generation, their test generation algorithm is started with random inputs and an initial database state. Another close approach is provided by Taneja and his colleagues [41] which is called MODA. It uses the mock object to generate test cases for a database application. However, neither AGENDA or RQP or MODA considers mutation testing.
2.4 Test Case Generation for SQL

2.4.1 Constraint based Test Case Generation

Chan and Cheung [8] provide an approach to convert the embedded SQL into a classical program. They then propose to use conventional testing method test the program. Zhang et al. [39] propose a fault based approach for automatic generation of database instances for embedded SQL statements along with a supporting tool to generate constraints that can be used in the white-box testing.

Pan and his colleagues [72] provide an approach named Mutagen that uses the concept of mutation testing to generate test cases. It also uses SynDB [73] to correlate various constraints by generating synthesized database interaction. Then it generates a program code mutant [74] and a SQL-query mutant [29] to incorporate query-mutant-killing constraints into the transformed code. Although Mutagen generates test cases using mutation operators defined by Tuya et al. [1], it generates test data mainly based on their defined constraints. Later Pan and his colleagues [75] extend their approaches to achieve high code coverage for database application. Using DSE [71], this approach takes database applications and a database as input and generates appropriate input values for the program. But this thesis emphasizes on proposing guidelines to generate test cases for SQL query based on the characteristics of the original SQL statement and its mutants.

2.4.2 Coverage Based Test Case Generation

Zhou and Frankl [76] develop a mutation testing tool for Java Database Application Analyzer (JDMA). They use SQLMutation tool provided by Tuya et al. [29].

Caball and Tuya [12] propose a coverage of SQL queries and an automated testing tool. They also illustrate how the test data could be modified to achieve better coverage of the SQL statement. In addition, they also present an algorithm that automated the calculation of SQL coverage based technique. Although they cover the SELECT query, they do not consider mutation testing. This study covers the SQL SELECT query specially on certain
Chapter 2. Literature Review

types of mutation operators - ABS, IRC and IRT. Tuya and his colleagues [28] present a coverage criteria based on Masking Modified Condition Decision Coverage (MCDC) which covers a wide range of syntax and semantics of SQL query. They term it as SQL Full Predicate Coverage (SQLFpc) that includes a wide variety of SQL features such as selection, joining, grouping, aggregate functions, subqueries, case expression and null values. They also create a tool to determine the coverage rules which can be operated through web interface or as a stand alone JAVA application. They use an open source ERP and CRM business solution named ‘Compiere’ as their subject program. They also use an open source test database generator tool named ‘dbMonster’. This thesis does not emphasize on MCDC or joining or sub-queries. It concentrates on ABS, IRC, and IRT mutation operators and tries to cover selection, grouping, aggregate functions among those operators. Like [28] this thesis does not create any tool, rather it develops the guidelines to generate test cases.

Cabal and Tuya [27] discuss about the measurement of coverage specially for SELECT queries. They introduce coverage tree for SELECT query and the constraints used in SQL statement. They propose an algorithm to calculate the coverage and furthermore, develop a tool to determine the coverage automatically. De La Riva and his colleagues [77] provide an automatic test dataset generation strategy for SQL queries based on SQL coverage rule (SQLFpc) [28]. According to their approach a database schema and a set of test requirement are provided as input and a database instance is generated as output. They use a constraint solvers called ‘Alloy’ where both schema and the test requirement are modeled. This thesis does not account SQLFpc, rather it considers test case generation strategy based on SQL mutation operator provided by Tuya [1].

2.4.3 Mutation based Test Case Generation

There are some test case generation strategies for Tuya’s[1] mutation operators. But they only confine to the comparison operator, join/outer join, aggregation operator [6, 78]. Chandra and his colleagues [7] propose test generation strategies for some additional mutation operators, such as null values, string constraints, aggregation with constraint on aggregation result and class of subqueries.
They have not provided test generation strategy when aggregate function is associated with \texttt{ABS()} mutation operator. In my study, I provide guidelines for such scenarios. I also propose guidelines for absolute value (\texttt{ABS}), identifier column replacement (\texttt{IRC}) or identifier constant replacement (\texttt{IRT}) mutation operators that are involved with relational operator and all type of aggregate functions.

2.5 Summary

In this chapter, I aim to address the gap in test case generation guidelines in database applications. I carefully compare different studies related to test input generation strategy with and without mutation testing. As my concentration is on SQL mutation testing, I compare various mutation techniques for SQL statements proposed by different researchers. Moreover, I find the existing gaps in generating test cases for different SQL mutation operators.

In summary, there are various researches in generating SQL mutants, generating test cases using coverage approaches and generating test cases to kill some SQL mutation operators. However, they have not considered those mutation operators such as the absolute value (\texttt{ABS}) operator, the identifier column replacement (\texttt{IRC}) operator and the identifier constant replacement (\texttt{IRT}) operator. Since the main goal of this thesis is to study test case generation techniques for these three different SQL mutation operators, I will discuss these in details in the following three chapters, one at a time.
Chapter 3

Test Case Generation for ABS Operator

In this chapter, I first analyse how an ABS mutation operator can be “added” to an SQL statement to create various ABS mutants. My contribution here is to have a comprehensive view on the impact of “adding” ABS mutation operator on SQL statements. In fact, depending on the actual position where the ABS operator is “added” (e.g. added in the SELECT clause, added in the WHERE clause, combined with an AGGREGATE function), there are altogether 6 different types of ABS mutants with 11 different variants. For each of these variants, I then develop guidelines to generate test cases that can kill the corresponding mutants. Finally, I evaluate the effectiveness of the guidelines via an experiment using NIST Database conformance test suite [79].

3.1 Motivation

Many researchers have proposed different techniques for selecting the test cases; among them Shah et al. [6] and Chandra et al. [7] propose test generation guidelines for SQL statements based on mutation testing aiming to cover several join mutation operators, relational mutation operator, and null mutation operator. However, they neglect to provide
CREATE TABLE STAFF {
   STAFFNUM VARCHAR PRIMARY KEY,
   STAFFNAME VARCHAR,
   GRADE INT,
   SALARY INT,
   CITY VARCHAR
}

Figure 3.1: Table Schema of STAFF

detailed insight into the absolute value insertion (ABS) mutation operator. The use of
ABS is very important to find the absolute difference between two numeric values such
as price differences and temperature differences. For example, to calculate the price dif-
ference between the list price and the dealer price from a database table, one can use
“ABS(dealerPrice -listPrice)”. The output (i.e. price difference) is always positive.

The following subsections discuss the characteristics of the ABS mutation operator and a
number of scenarios that are needed to be considered for the ABS mutation operator and
propose the test generation guidelines for killing ABS mutants.

3.2 Absolute Value (ABS) Operator

In an SQL statement, there may be some arithmetic expressions and references to numbers.
The definition of ABS operator is adopted from Tuya [1]. An ABS mutant is created if
one of these arithmetic expressions or number, x, is replaced by ABS(x) or −ABS(x).
However, references to numbers are not mutated either inside of GROUP BY and ORDER BY
clauses or in the select list of EXISTS subquery.

For ease of discussion and illustration, we use the database schema STAFF given in Fig-
ure 3.1 as an example throughout the rest of this chapter. The database table STAFF
has four attributes, namely STAFFNUM of type VARCHAR, STAFFNAME of VARCHAR,
GRADE of INT, SALARY of INT and CITY of VARCHAR. For example, for the SQL state-
ment “SELECT staffname, salary FROM STAFF”, ABS() can only be applied on “salary” to
generate the ABS mutant, “SELECT staffname, ABS(salary) FROM STAFF”.
During SQL mutation testing, a test suite with only positive values cannot distinguish the mutants from its original SQL statement generated by a positive ABS mutation operator. Similarly, a test suite with only negative values cannot distinguish the mutants by a negative ABS mutation operator. Therefore, it is important to create a test dataset for ABS mutation operator where both positive and negative values exist.

### 3.3 Six Types of ABS Mutants

For generating ABS mutants, SQL SELECT statements are considered in this study. Usually, a SQL SELECT statement in a database manipulates data from the database objects (e.g. table, view). This manipulation operation may involve a single clause or multiple clauses such as join (e.g., inner, outer, left, or right), sub-queries, different aggregate or arithmetic functions etc. Based on the manipulation operation, SQL SELECT statements can be classified as simple and complex SQL SELECT statement. Simple SQL SELECT statements refer to those statements that have a WHERE clause but no other features (such as sub-query, join clause, group by) and no other SQL functions (such as arithmetic, round() etc.) except an aggregate function (MAX, MIN, SUM, AVG, COUNT). An example of a simple SQL SELECT statement is “SELECT salary FROM STAFF WHERE grade > 40” where there is a WHERE clause but no other SQL features. On the other hand, a complex SQL SELECT statement is one that has multiple features of SQL (such as join, sub-query, group by etc) in the SQL SELECT statement. An example of a complex SQL SELECT statement is “SELECT city, grade, salary FROM STAFF WHERE grade > 40 GROUP BY city”, where there is a GROUP BY SQL feature. In this thesis, I do not consider complex SQL statements. This is also inline with Tuya’s [1] definition of ABS mutants.

During my analysis, different types of ABS mutants are possible depending on where the ABS mutation operator is inserted into an SQL SELECT statement; which in turn depend on the answers of the following three questions:

Q1 Does the SQL statement have a WHERE clause? There are two types of SQL statements based on this question. If the SQL has a WHERE clause, a possible SQL statement is
“SELECT salary FROM STAFF WHERE grade > 40.” Otherwise the query does not have any WHERE clause. An example of such query is “SELECT salary FROM STAFF.”

Q2 Does the SQL statement have an aggregate function? An SQL query may or may not have an aggregate function. The possible examples of SQL statements with aggregate function are “SELECT MAX(salary) FROM STAFF” or “SELECT AVG(salary) FROM STAFF WHERE grade > 40.”

Q3 Where does the ABS operator occur? In the SELECT clause? In the WHERE clause? In the aggregate function? The ABS operator occurs in three ways for an SQL statement. (a) It can occur in the SELECT clause. An example of such an SQL query is “SELECT ABS(grade) FROM STAFF.” (b) In cases where an aggregate function exists in the SQL statement, ABS() occurs in the SELECT clause with the aggregate field. For example, a possible example with ABS() in the aggregate field is “SELECT ABS(MAX(grade)) FROM STAFF”. (c) If there is a WHERE clause in the SQL statements, an ABS() can be applied in the SELECT clause or in the WHERE clause. A possible example of ABS() in the SELECT clause is “SELECT ABS(grade) FROM STAFF WHERE grade < 40.” The ABS mutation operator can also occur in the WHERE clause, e.g. “SELECT grade FROM STAFF WHERE ABS(grade) < 40.”
Table 3.1: Different Types of $ABS()$ Mutants

<table>
<thead>
<tr>
<th>Type</th>
<th>WHERE</th>
<th>AGGR</th>
<th>$ABS()$</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no</td>
<td>no</td>
<td>SELECT</td>
<td>$ABS(\text{grade})$ FROM STAFF</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>yes</td>
<td>AGGR</td>
<td>$SELECT \ ABS(\text{MAX(grade)})$ FROM STAFF</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>no</td>
<td>SELECT</td>
<td>$ABS(\text{grade})$ FROM STAFF $\text{WHERE grade} &lt; 40$</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>no</td>
<td>WHERE</td>
<td>$SELECT \ \text{grade FROM STAFF WHERE } ABS(\text{grade}) &lt; 40$</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>yes</td>
<td>AGGR</td>
<td>$SELECT \ ABS(\text{MAX(grade)})$ FROM STAFF $\text{WHERE grade} &lt; 40$</td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td>yes</td>
<td>WHERE</td>
<td>$SELECT \ \text{MAX(grade) FROM STAFF WHERE } ABS(\text{grade}) &lt; 40$</td>
</tr>
</tbody>
</table>

Figure 3.2 shows the detailed classification of these six types of $ABS$ mutants. In the rest of this section, I discuss in detail how these 6 different types of mutant are formed. Table 3.1 shows a summary of these 6 different types of mutants with examples.

Type 1 There is no WHERE clause and no aggregate function in the SQL statement. As a result, the $ABS$ operator can only occur in the fields being selected in the SQL statement. For example, a possible mutant of “SELECT salary FROM STAFF” may be “SELECT $ABS(\text{salary})$ FROM STAFF.”

Type 2 Similar to Type 1, there is no WHERE clause in the SQL statement but there is at least one aggregate function (such as MAX, MIN, SUM, COUNT and AVG) in the SELECT clause of the SQL statement. As a result, the $ABS$ mutation operator occurs in the aggregated function. For example, two possible mutants of “SELECT MAX(\text{grade}) \ FROM STAFF” may be “SELECT $ABS(MAX(\text{grade}))$ \ FROM STAFF” or “SELECT $MAX(ABS(\text{grade}))$ \ FROM STAFF.”

Type 3 There is a WHERE clause in the SQL statement, but there is no aggregate function and the $ABS$ mutation operator occurs in the SELECT field not in the WHERE clause. For example, a possible mutant of “SELECT grade \ FROM STAFF \ WHERE \ grade < 40” is “SELECT $ABS(\text{grade})$ \ FROM STAFF \ WHERE \ grade < 40.”

---

1 Due to the syntactic nature of the SQL statement, once an aggregate function appears in the SELECT clause, all fields in the SELECT clause must be aggregate expressions. The only exception is when there is a GROUP BY clause. Since an investigation of GROUP BY clause is out of the scope of this thesis, they are not discussed here.
Type 4 Similar to Type 3, there is a WHERE clause in the SQL statement, but there is no aggregate function and the \textit{ABS} mutation operator occurs in the WHERE clause. For example, a possible mutant of \texttt{“SELECT grade FROM STAFF WHERE grade < 40”} is \texttt{“SELECT grade FROM STAFF WHERE \textit{ABS}(grade) < 40”}.

Type 5 Again, same as Type 3 and Type 4, this type of mutant has a WHERE clause and some aggregate functions in the \texttt{SELECT} clause in the SQL statement, and the \textit{ABS} mutation operator occurs in the aggregate function. For example, a possible mutant of \texttt{“SELECT \textit{MAX}(grade STAFF WHERE grade < 40”} is \texttt{“SELECT \textit{ABS}(\textit{MAX}(grade)) FROM STAFF WHERE grade < 40”}.

Type 6 In this type of mutant, there is a WHERE clause and some aggregate functions in the \texttt{SELECT} clause in the SQL statement, and the \textit{ABS} mutation operator occurs in the \texttt{WHERE} clause. For example, a possible mutant of \texttt{“SELECT \textit{MAX}(grade) FROM STAFF WHERE grade < 40”} is \texttt{“SELECT \textit{MAX}(grade) FROM STAFF WHERE \textit{ABS}(grade) < 40”}.

\subsection*{3.4 Test Case Generation Guidelines}

In this section, I propose test case generation guidelines for killing six types of \textit{ABS} mutants. I use G1 to denote the guideline for Type 1 \textit{ABS} mutant, G2 for Type 2, and so on. During my analysis, there are different test case generation guidelines for the same type of mutant (say, two guidelines for Type 2 mutants) depending on the actual scenarios in context. As a result, I further break these 6 types of mutants into 11 different variants. I then propose test case generation guideline for each of these variants. Table 3.2 summarizes the guidelines for all 11 variants of 6 different types of \textit{ABS} mutants.

G1 Type 1 mutants require that the \textit{ABS} mutation operator occurs in the \texttt{SELECT} clause.

Hence, one possible mutant of \texttt{“SELECT grade FROM STAFF”} is \texttt{“SELECT +\textit{ABS}(grade) FROM STAFF.”} In order to distinguish $+\textit{ABS}(grade)$ (respectively, $-\textit{ABS}(grade)$) from “grade,” I need to set some negative (respectively positive) values for the “grade”
Table 3.2: Test Case Generation Guidelines for ABS mutants

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>—</td>
<td>G1</td>
<td>Use a data set with some positive and some negative values in the field involved in the SELECT clause</td>
<td>4 and −4 for the “grade” field</td>
</tr>
<tr>
<td>2</td>
<td>S1 (COUNT)</td>
<td>G2-S1</td>
<td>Apply OPP rule (a pair of numbers equal in magnitude but opposite in sign) in the field involved in COUNT()</td>
<td>3 and −3 for the “grade” field</td>
</tr>
<tr>
<td></td>
<td>S2 (other aggregate functions)</td>
<td>G2-S2</td>
<td>Use two data sets: one positive and one negative for the field involved in aggregate function</td>
<td>{3, 4, 5} and {−3, −4, −5} for the “grade” field</td>
</tr>
<tr>
<td>3</td>
<td>S1 (the field in the SELECT clause appears in the WHERE clause)</td>
<td>G3-S1</td>
<td>Apply BVA-OPP rule for the required field</td>
<td>{39, 40, 41, −39, −40, −41} for the “grade” field</td>
</tr>
<tr>
<td></td>
<td>S2 (the field in the SELECT clause does not appear in the WHERE clause)</td>
<td>G3-S2</td>
<td>(1) Apply BVA-OPP rule for the field in the WHERE clause and (2) for each BVA-OPP value, set one positive value and one negative value for the field in the SELECT clause such that all these values are different</td>
<td>{grade=39, salary=1500}, (39, −2000), (40, 2500), (40, −3000), (41, 3500), (41, −4000), (−39, 5000), (−39, −5500), (−40, 6000), (−40, −6500), (−41, 7000), (−41, −7500)</td>
</tr>
<tr>
<td>4</td>
<td>S1 (the field in the SELECT also appears in the WHERE clause)</td>
<td>G4-S1</td>
<td>Apply BVA-OPP+1 rule for the field in the WHERE clause</td>
<td>{39, 40, 41, −38, −40, −42} for the “grade” field</td>
</tr>
<tr>
<td></td>
<td>S2 (the field in the SELECT clause does not appear in the WHERE clause)</td>
<td>G4-S2</td>
<td>(1) Apply BVA-OPP rule for the field in the WHERE clause, and (2) Set the values in the field involved in the SELECT clause such that they are different</td>
<td>{grade=39, salary=1000}, (40, 1500), (41, 2000), (−39, 2500), (−40, 3000), (−41, 3500)</td>
</tr>
<tr>
<td>5</td>
<td>S1 (COUNT) Case 1: the field in COUNT() appears in the WHERE clause</td>
<td>G5-S1-1a</td>
<td>Apply BVA-OPP for the values in the field in the COUNT() aggregate function of the SELECT clause</td>
<td>{39, 40, 41, −39, −40, −41} for the “grade” field</td>
</tr>
<tr>
<td></td>
<td>S1 (COUNT) Case 2: the field in COUNT() does not appear in the WHERE clause</td>
<td>G5-S1-1b</td>
<td>(1) Apply BVA-OPP rule for the field involved in the WHERE clause and (2) for each BVA-OPP value, use OPP rule for the field involved in COUNT() and ensure that these values are numerically different from other values set for other BVA-OPP values</td>
<td>{grade=39, salary=1000}, (39, −1000), (40, 1500), (40, −1500), (41, 2000), (41, −2000), (−39, 2500), (−39, −2500), (−40, 3000), (−40, −3000), (−41, 3500), (−41, −3500)</td>
</tr>
<tr>
<td></td>
<td>S2 (other aggregate functions) Case 1: the field in the aggregate function appears in the WHERE clause</td>
<td>G5-S2-1a</td>
<td>(1) Apply BVA-OPP rule for the field involved in the aggregate function and (2) Split these BVA-OPP values into two data sets, one positive and one negative</td>
<td>{39, 40, 41} and {−39, −40, −41} for the “grade” field</td>
</tr>
<tr>
<td></td>
<td>S2 (other aggregate functions) Case 2: the field in the aggregate function does not appear in the WHERE clause</td>
<td>G5-S2-1b</td>
<td>(1) Apply BVA rule for the field involved in the WHERE clause and (2) Use two data sets, one positive and one negative, on the field involved in the aggregate function</td>
<td>{grade=39, salary=1000}, (40, 1500), (41, 2000) and {39, −1000}, (40, −1500), (41, −2000)</td>
</tr>
<tr>
<td>6</td>
<td>S1 (the field in the aggregate function of the SELECT clause appears in the WHERE clause)</td>
<td>G6-S1</td>
<td>Use four data sets, each having one record with the following values for the field involved in the WHERE clause: (1) the one on the boundary in the WHERE condition, (2) one above the boundary, (3) the value that is equal in magnitude but opposite in sign with respect to the boundary, and (4) the value that is equal in magnitude but opposite in sign with respect to the one above the boundary (that is, the one used in (2)).</td>
<td>{40}, {41}, {−40}, {−41} for the “grade” field</td>
</tr>
<tr>
<td></td>
<td>S2 (the field in the aggregate function does not appear in the WHERE clause)</td>
<td>G6-S2</td>
<td>Same guideline as G6-S1</td>
<td></td>
</tr>
</tbody>
</table>
field in the STAFF table. As a result, I need some positive and some negative values in the numeric fields of the database table for detecting this type of ABS mutants.

In cases where the “grade” field can only be positive (or, non-negative) due to system constraints, I cannot set negative values in the database table. In that case, the “+ABS(grade)” mutant will be equivalent to the original SQL statement. A similar situation occurs when the “grade” field can only be negative, the “−ABS(grade)” mutant will be equivalent to the original SQL statement.

G2 In this type of mutant, the ABS mutation operator occurs in the aggregate function such as MAX(), MIN(), SUM(), AVG(), and COUNT(). I have the following two scenarios:

S1 For the COUNT() function, I have six different forms of ABS mutants. They are

\[\pm ABS(COUNT(...)), \text{COUNT}(\pm ABS(...))\] and \[\text{COUNT(DISTINCT} \pm ABS(...))\].

Since \text{COUNT()} returns the number of records, it always returns a non-negative value. Thus, \[+ABS(COUNT(...))\] and \[\text{COUNT}(\pm ABS(...))\] mutants are equivalent to \[\text{COUNT}(...).\] On the other hand, the \[−ABS(COUNT(...))\] mutant is different from \[\text{COUNT}(...).\] because the mutant always returns a negative value. As a result, I do not need to worry about these four mutants. For the remaining two \[\text{COUNT(DISTINCT} \pm ABS(...))\] mutants, my guideline is to have at least a pair of values, both equal in magnitude but opposite in sign, in the field involved in \text{COUNT()} in the SELECT clause so as to distinguish them. This is referred to as the OPPosite rule. For example, on the one hand, if I have 3 and −3 in the “grade” field, \[\text{COUNT(DISTINCT}(grade))\] returns 2. On the other hand, \[\text{COUNT(DISTINCT}(\pm ABS(grade)))\] return 1.

In summary, my guideline for \text{COUNT()} is to use the OPP rule to set a pair of values in the field involved in \text{COUNT()} so that they are both equal in magnitude but opposite in sign.

S2 For other aggregate functions, I propose one guideline for all of them instead of having one guideline per function.

My guideline is to use two datasets, one with all positive values and the other with all negative values in the field involved in the aggregate function. By using two datasets (say, DS1 and DS2), I mean that testers need to perform two testing
executions, one with each dataset. First, testers need to set up the database with
values in one of the dataset, say DS1, and test the SQL mutants. Second, testers
have to repeat the same testing process using values in the second dataset, say DS2,
as before. In this case, the positive value set (e.g. \{3, 4, 5\}, setting the values of
the grade field using 3, 4 and 5 in the database table) can be used to distinguish
those \textit{ABS} mutants from the original function (e.g. “\texttt{SELECT -ABS(\texttt{MAX(grade)})}
\texttt{FROM STAFF}” and “\texttt{SELECT MAX(-ABS(grade)) FROM STAFF}” versus “\texttt{SELECT}
\texttt{MAX(grade) FROM STAFF}”) because the original function (i.e. \texttt{MAX(grade)}) gives
a positive result (i.e. 5) whereas the two \textit{ABS} mutants (i.e. \texttt{-ABS(MAX(grade)})
and \texttt{MAX(-ABS(grade))}) return negative results (i.e. -5). On the other hand,
a negative value set (e.g. \{-3, -4, -5\}) can be used to distinguish those \textit{ABS}
mutants from the original function (e.g. \texttt{ABS(MAX(...))} and \texttt{MAX(ABS(...))}
versus \texttt{MAX(...)}) because the original function returns a negative result (e.g. -3)
whereas the two \textit{ABS} mutants return positive results (e.g 3 and 5 respectively).
It does not matter whether there is a \texttt{DISTINCT} keyword in between the aggregate
function and the \textit{ABS} operator, the above mentioned datasets are adequate to
distinguish the mutants from its original SQL statement.

G3 In Type 3 mutants, the original SQL statement has a \texttt{WHERE} clause but the \textit{ABS}
operator occurs in the \texttt{SELECT} clause. There are two different scenarios depending on
whether the field involved in the \textit{ABS} operator appears in the \texttt{WHERE} clause or not.

S1 The field involved in the \textit{ABS} operator in the \texttt{SELECT} clause also appears in the
\texttt{WHERE} clause. For example, a possible mutant of “\texttt{SELECT grade FROM STAFF}
\texttt{WHERE grade < 40}” is “\texttt{SELECT -ABS(grade) FROM STAFF WHERE grade < 40}.”

My guideline consists of two steps. First, I apply boundary value analysis (BVA)
on the condition in the \texttt{WHERE} clause that is related to the field involved in the
\texttt{SELECT} clause. For example, I apply BVA on the condition “grade < 40” in the
\texttt{WHERE} clause to select 39, 40, and 41 for the “grade” field\footnote{In applying the BVA when there are more than one condition in the \texttt{WHERE} clause, testers have to (1) apply BVA for each condition and (2) collectively use all the boundary values (in case the conditions are “\texttt{grade < 40 AND salary > 3000}”) or combine them (in case the conditions are “\texttt{grade > 10 AND grade < 40}”) to form the required test data values.}. In fact, these numbers
could also be used on conditions like “grade ≥ 40” with other relational comparison operators such as ≤, >, ≥, = and ≠. Second, I apply the OPP rule on each of these BVA values so as to set the values such that they are equal in magnitude but opposite in sign with respect to the BVA values. For example, I apply OPP rule on 39, 40 and 41 to obtain the values −39, −40 and −41 for the “grade” field. These negative values can be used to distinguish “+ABS(grade)” from “grade”. I refer to this rule of applying BVA (that is, 39, 40 and 41 for “grade < 40”) and the applying OPP rule to set values of the opposite sign (that is, −39, −40 and −41) as the BVA-OPP rule.

As a reminder, in cases where the condition in the WHERE clause only selects positive values of “grade” (e.g. “grade > 40”), I cannot distinguish “grade” from “+ABS(grade)” because they are equivalent. Similarly, I cannot distinguish “grade” from “−ABS(grade)” when the condition in the WHERE clause only selects negative values of “grade” (e.g. “grade < −40”).

S2 The field involved in the ABS operator in the SELECT clause does not appear in the WHERE clause. It does not matter whether the WHERE clause has other numeric fields or whether it does not have any numeric fields at all, the guideline is the same.

The proposed guideline is to set some positive and some negatives values in the field involved in the SELECT clause for those records satisfying the condition in the WHERE clause. Similar to the case of G1 above, those positive values can help to distinguish the −ABS() mutants from the original SQL statement whereas the negative values distinguish the +ABS() mutants.

If the condition in the WHERE clause involves comparison operator such as <, I apply the BVA-OPP rule to set the numerical values in the field involved in the WHERE clause. For example, for the SQL statement “SELECT salary FROM STAFF WHERE grade < 40,” a possible mutant is ‘SELECT −ABS(salary) FROM STAFF WHERE grade < 40.” Since the condition in the WHERE clause is “grade < 40,” I apply the BVA-OPP rule to set the values 39, 40, 41, −39, −40 and −41 in the “grade” field. I then need to set some positive and some negative values in the “salary” field for those records whose “grade” < 40 (that is, 39, −39, −40, and
One possible option is to use “(grade=39, salary=1500), (−39, −2000),
(−40, 2000), (−41, −2500)”. However, this is too tedious when considering other
possible relational comparison operators in the WHERE clause (e.g. “grade ≥ 40”).
My suggestion is to set one positive value and one negative value of “salary” for
each individual “grade” value in such a way that all these “salary” values are
distinct, that is, “(39, 1500), (39, −2000), (40, 2500), (40, −3000), (41, 3500), (41,
−4000), (−39, 5000), (−39, −5500), (−40, 6000), (−40, −6500), (−41, 7000), (−41,
−7500)”. By doing so, I can use the positive (respectively negative) “salary” values
to distinguish “−ABS(salary)” (respectively “+ABS(salary)” from “salary” for
all possible relational comparison operators (e.g. “grade ≤ 40” or “grade ≥ −40”).
When the field involved in the WHERE clause must be unique due to database
constraints (e.g. a primary key), I can use two different datasets— one contains
only positive values and the other contains only negative values.

G4 In Type 4 mutants, there is no aggregate function and the ABS operator occurs in the
WHERE clause. Similar to the case of Type 3 mutants, there are two different scenarios,
depending on whether the field involved in the ABS operator in the WHERE clause
appears in the SELECT clause or not.

S1 The field involved in the ABS operator in the WHERE clause also appears in the
SELECT clause. The guideline is similar to that of S1 in G3. Instead of using
BVA-OPP, I use the BVA-OPP+1 rule.

The BVA-OPP+1 rule is to first apply BVA on the condition to select the BVA
groups. Then, I use the “opposite-in-sign” value of the boundary value. Finally,
instead of using just the “opposite-in-sign” values of the one-below and one-above
the boundary value, I select the “opposite” values that are “further away from
the boundary” by at least 1 than these one-below and one-above values. In cases
where the condition is “grade < 40”, the BVA values are 39, 40, 41. Instead of
using −39, −40 and −41, I use −38 (as 38 is further away from boundary than
39), −40 (the opposite of 40 as this is the boundary) and −42 (as 42 is further
away from the boundary than 41). As a note, I can choose to use −37 (for −38)
or −43 (for −42).
Chapter 3. Test Case Generation for ABS Operator

Table 3.3: Number of Records satisfying “SELECT salary FROM STAFF WHERE condition” with \{39, 40, 41, -39, -40, -41\} in the “grade” Field

<table>
<thead>
<tr>
<th>SQL statement</th>
<th>condition in WHERE clause</th>
<th>Relational operator ( (op) )</th>
<th>&lt;</th>
<th>≤</th>
<th>&gt;</th>
<th>≥</th>
<th>=</th>
<th>≠</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original SQL</td>
<td>grade ( op ) 40</td>
<td></td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>ABS mutant</td>
<td>( ABS(grade) ) ( op ) 40</td>
<td></td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>(-ABS) mutant</td>
<td>(-ABS(grade) ) ( op ) 40</td>
<td></td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

For example, for the original SQL statement “SELECT grade FROM STAFF WHERE grade < 40”, a possible Type 4 mutant is “SELECT grade FROM STAFF WHERE \( ABS(grade) \) < 40.” In this case, I set the values of the “grade” field to 39, 40, 41, -38, -40 and -42. The positive values can be used to detect the \(-ABS\) mutant whereas the negative values detect the \(+ABS\) mutant.

S2 The field involved in the \( ABS\) operator in the WHERE clause does not appear in the SELECT clause. It does not matter whether the fields in SELECT clause are numeric or not, the guideline will still be the same. However, there is a complication here. I will use the following example to explain it and illustrate my guideline as a resolution.

Suppose the original SQL statement is “SELECT salary FROM STAFF WHERE grade < 40”. A possible mutant is “SELECT salary FROM STAFF WHERE \( ABS(grade) \) < 40”. Since the values from the field of the SELECT clause, say “salary” or other non-numeric fields, do not relate to the “grade” field or 40, a simple way that I could distinguish the results of the original SQL statement from those of the mutants is to examine the details of the values in the results. However, this is a very tedious task, in cases where the database table has many records. It is also very easy to make a mistake in comparing the values of the records. My guideline discussed below is formulated to resolve this issue. In a nutshell, my intention is to set up the data in the database so as to ensure that the number of records returned by the original SQL statements and that of the mutants are different. Therefore, I can simply count\(^3\) the numbers of records in the returned results and use it to tell that they are different without going into their details.

\(^3\)Either by manual counting or using COUNT() in the SQL
My guideline is to first apply the $BVA\text{-}OPP$ rule on the condition in the $\text{WHERE}$ clause (that is “grade < 40”) to set the values of the field involved in the condition (e.g. set “grade” to 39, 40, 41, −39, −40 and −41), and then set the values of the field involved in the $\text{SELECT}$ clause (e.g. “salary”) such that they are different from each other. An example is to set the values such that (grade=39, salary=1000), (40, 1500), (41, 2000), (−39, 2500), (−40,3000) and (−41, 3500). By setting the six records in the database in this way, Table 3.3 shows that the numbers of records returned by the original SQL statement and the corresponding mutants are different irrespective of the relational operator. As a result, these records can be used to distinguish the $ABS$ mutants from the original SQL statement.

G5 In this type of mutant, there are aggregate functions and the $\text{WHERE}$ clause in the original SQL statement and the $ABS$ mutation operator occurs in the aggregate function in the $\text{SELECT}$ clause. Similar to the case of G2, I discuss my guidelines according to the aggregate function used in the SQL statement.

S1 For the $\text{COUNT()}$ function, as explained in G2-S1 above, I need to consider only $\text{COUNT(DISTINCT } \pm \text{ABS(...))}$ mutants. I have the following two cases depending on whether the field involved in $\text{COUNT()}$ appears in the condition in the $\text{WHERE}$ clause.

(a) When the field involved in $\text{COUNT()}$ also appears in the $\text{WHERE}$ clause, I apply the $BVA\text{-}OPP$ rule to set the values in that field.

For example, for the original SQL statement “$\text{SELECT COUNT(DISTINCT grade) FROM STAFF WHERE grade < 40}$”, I apply the $BVA\text{-}OPP$ rule to set the “grade” field to 39, 40, 41, −39, −40 and −41. Since the condition “grade < 40” will select at least 39 and −39, both $ABS$ mutants “$\text{SELECT COUNT(DISTINCT } \pm \text{ABS(grade)) FROM STAFF WHERE grade < 40}$” return 3 instead of 4 returned by the original SQL statement. In the case of the relational operator used in the $\text{WHERE}$ clause is $\leq$ or $\neq$, a similar argument applies. For the cases of “grade > 40”, “grade $\geq$ 40” and “grade = 40”, the $ABS$ mutants are equivalent to the original statement because the values selected are always positive.
(b) When the field involved in \texttt{COUNT()} does not appear in the \texttt{WHERE} clause, I first apply the \texttt{BVA-OPP} rule to set the values of the field involved in the \texttt{WHERE} clause. Then, for each of these BVA values, I need to set the values for the field involved in \texttt{COUNT()} so that (1) they are equal in magnitude but opposite in sign and (2) they are numerically different from those values set for other BVA values.

For example, for the original SQL statement 
\begin{verbatim}
SELECT COUNT(DISTINCT salary)
FROM STAFF WHERE grade < 40
\end{verbatim}
I first apply the \texttt{BVA-OPP} rule to set the “grade” field to 39, 40, 41, −39, −40 and −41. Then, for each of these BVA values, I set the values in the “salary” field so that they are (1) equal in magnitude but opposite in sign, and (2) different from other “salary” values set for other BVA values. For example, one option is to set (grade=39, salary=1000), (39, −1000), (40, 1500), (40, −1500), (41, 2000), (41, −2000), (−39, 2500), (−39, −2500), (−40, 3000), (−40, −3000), (−41, 3500) and (−41, −3500). By setting the values like these, I can distinguish the \texttt{ABS} mutants (e.g. 
\begin{verbatim}
SELECT COUNT(DISTINCT ABS(salary))
FROM STAFF WHERE grade < 40
\end{verbatim}
) from the original SQL statement.

S2 For other aggregate functions, as discussed in G2-S2, it is easier to use two datasets, one positive and one negative, to distinguish the original SQL statement from the mutants. Again, I have the following two cases depending on whether the field involved in the aggregate function appears in the condition in the \texttt{WHERE} clause.

(a) When the field involved in the aggregate function also appears in the \texttt{WHERE} clause, I first apply the \texttt{BVA-OPP} rule to identify the values required in that field and then split these values into two datasets, one using the positive values and the other using the negative values. As explained in G2-S2, the positive (respectively negative) dataset can be used to distinguish the \texttt{−ABS} (respectively \texttt{+ABS}) mutants.

For example, for the original SQL statement 
\begin{verbatim}
SELECT MAX(grade)
FROM STAFF WHERE grade < 40
\end{verbatim}
we use \{39, 40, 41\} as the positive dataset and \{−39, −40, −41\} as the negative dataset. The positive dataset can be used to distinguish the original SQL statement from the \texttt{−ABS} mutants (e.g. 
\begin{verbatim}
SELECT MAX(ABS(grade))
FROM STAFF WHERE grade < 40
\end{verbatim}
) from the mutants.
Chapter 3. Test Case Generation for ABS Operator

\(-ABS(MAX(grade))\) FROM STAFF WHERE grade < 40”). The negative dataset can be used to distinguish the original SQL statement from the \(+ABS\) mutants (e.g. “SELECT MAX(+ABS(grade)) FROM STAFF WHERE grade < 40”).

(b) When the field involved in the aggregate function does not appear in the WHERE clause, I first apply the BVA rule to identify the values required in the field involved in the WHERE clause, then I set two datasets, one positive and one negative, on the values of the field involved in the aggregate function. As explained in G2-S2, the positive (respectively negative) dataset can be used to distinguish the \(-ABS\) (respectively \(+ABS\)) mutants.

For example, for the original SQL statement “SELECT MAX(salary) FROM STAFF WHERE grade < 40”, I apply BVA on “grade < 40” to set 39, 40 and 41 to the “grade” field. I then set two datasets, one positive and one negative, for the “salary” field using the required values for the “grade” field. One possible option is to set the positive dataset as \{(grade=39, salary=1000), (40, 1500), (41, 2000)\} and the negative dataset as \{(39, -1000), (40, -1500), (41, -2000)\}. The positive dataset can be used to distinguish the original statement from the \(-ABS\) mutants (e.g. “SELECT MAX(-ABS(salary)) FROM STAFF WHERE grade < 40”).

G6 In Type 6 mutants, the SQL statement has at least an aggregate function (e.g. COUNT, MAX, MIN, SUM and AVG) and a WHERE clause, with the ABS mutation operator in the WHERE clause. For example, given the original SQL statement “SELECT MAX(grade) FROM STAFF WHERE grade < 40”, a possible mutant is “SELECT MAX(grade) FROM STAFF WHERE ABS(grade) < 40”. Since there are at least 5 different aggregate functions (e.g. COUNT, MAX, MIN, SUM and AVG) and 6 relational operators (\(<, \leq, >, \geq, =\) and \(\neq\)), the formulation of test case generation guidelines is quite a challenging task.

There are several approaches to set the test case generation guidelines. At one extreme, I can have one guideline per aggregate function and relational operator pair. Hence, there are at least 30 such guidelines. This is very tedious, especially when I need to keep track of which guideline is for which aggregate-relational pair. At the other extreme,
Table 3.4: Number of records returned by “SELECT AGGR(grade) FROM STAFF WHERE condition”

(a) Database has only 1 record with grade = 40

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>condition</th>
<th>Relational operator (op)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original SQL</td>
<td>grade ( op ) 40</td>
<td>(&lt;</td>
</tr>
<tr>
<td>(ABS) mutant</td>
<td>(ABS(\text{grade}) \ op ) 40</td>
<td>0</td>
</tr>
<tr>
<td>(-ABS) mutant</td>
<td>(-ABS(\text{grade}) \ op ) 40</td>
<td>1*</td>
</tr>
</tbody>
</table>

(b) Database has only 1 record with grade = 41

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>condition</th>
<th>Relational operator (op)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original SQL</td>
<td>grade ( op ) 40</td>
<td>(&lt;</td>
</tr>
<tr>
<td>(ABS) mutant</td>
<td>(ABS(\text{grade}) \ op ) 40</td>
<td>0</td>
</tr>
<tr>
<td>(-ABS) mutant</td>
<td>(-ABS(\text{grade}) \ op ) 40</td>
<td>1*</td>
</tr>
</tbody>
</table>

(c) Database has only 1 record with grade = -40

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>condition</th>
<th>Relational operator (op)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original SQL</td>
<td>grade ( op ) 40</td>
<td>(&lt;</td>
</tr>
<tr>
<td>(ABS) mutant</td>
<td>(ABS(\text{grade}) \ op ) 40</td>
<td>0*</td>
</tr>
<tr>
<td>(-ABS) mutant</td>
<td>(-ABS(\text{grade}) \ op ) 40</td>
<td>1</td>
</tr>
</tbody>
</table>

(d) Database has only 1 record with grade = -41

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>condition</th>
<th>Relational operator (op)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original SQL</td>
<td>grade ( op ) 40</td>
<td>(&lt;</td>
</tr>
<tr>
<td>(ABS) mutant</td>
<td>(ABS(\text{grade}) \ op ) 40</td>
<td>0*</td>
</tr>
<tr>
<td>(-ABS) mutant</td>
<td>(-ABS(\text{grade}) \ op ) 40</td>
<td>1</td>
</tr>
</tbody>
</table>

*: This indicates that the number of records satisfying the original SQL and that satisfying the corresponding \(ABS\) mutants are different. Hence, the relevant data in the database can be used to distinguish the original SQL and the corresponding \(ABS\) mutants (irrespective of the aggregate function being used) under the corresponding circumstances.

It may be good to have just one guideline that works for all aggregate-relational pairs. I adopt this latter approach in this thesis. Similarly, there are two scenarios depending on whether the field involved in the \(ABS\) operator appears in the aggregate function or not.

S1 The field involved in the \(ABS\) operator in the \texttt{WHERE} clause also appears in the aggregate function. For example, given the SQL statement “SELECT \texttt{MAX(grade)} FROM \texttt{STAFF WHERE grade < 40}”, a possible mutant is “SELECT \texttt{MAX(grade)} FROM
STAFF WHERE \( \text{ABS(grade)} < 40 \)." In order to distinguish the above two different SQL statements, I need to use a dataset whose values in the “grade” field are such that grade \( \leq -40 \). Otherwise, if there are any values in the “grade” field between \(-39\) and \(39\), both SQL statements return the same maximum value.

However, adopting the above approach means the original SQL will return some values and the \( \text{ABS} \) mutant will return null. The merit of this approach is that I do not need to worry about which aggregate function is used in the SQL statement. Therefore, in this thesis, I adopt this approach.

My guideline is to set four different datasets and each dataset has just one value for the field involved (both in the WHERE condition and the aggregate function). These four values are (1) the one on the boundary in the WHERE condition, (2) one above the boundary, (3) the value that is equal in magnitude but opposite in sign with respect to the boundary, (4) the value that is equal in magnitude but opposite in sign with respect to the one above the boundary (that is, the one used in (2)).

So, for the example SQL statement and its mutants, I need to use four datasets, each having one record, and the “grade” values in these records are 40 (on the boundary), 41 (one above), \(-40\) (the opposite of 40), and \(-41\) (the opposite of 41). Table 3.4 shows the number of records returned by the corresponding SQL statements (the original and the \( \text{ABS} \) mutants) using these 4 different datasets. Collectively, these 4 datasets can distinguish the original SQL statement from its various \( \text{ABS} \) mutants irrespective of the aggregate function and the relational operator.

S2 The field involved in the \( \text{ABS} \) operator in the WHERE clause does not appear in the SELECT clause. For example, given the SQL statement “SELECT \( \text{MAX(salary)} \) FROM STAFF WHERE grade < 40”, a possible mutant is “SELECT \( \text{MAX(salary)} \) FROM STAFF WHERE \( \text{ABS(grade)} < 40 \).”

Surprisingly, the guideline for this scenario is the same as the one above because it is the number of records returned that matters rather than the actual values in the field involved in the aggregate function.
Last, but not least, I do not investigate the situations where an $ABS$ operator can occur over another $ABS$ operator because the original SQL and the $ABS$ mutants are either equivalent to each other or they can be easily distinguished. For example, given the original SQL “\texttt{SELECT } ABS(\texttt{grade}) \texttt{ FROM STAFF WHERE grade < 40}”, one $ABS$ mutant may be “\texttt{SELECT } -ABS(ABS(\texttt{grade})) \texttt{ FROM STAFF WHERE grade < 40}”. The mutant “$ABS(ABS(...))$” is equivalent to the original “$ABS(...)$”, and “$-ABS(-ABS(...))$” is equivalent to “$-ABS(...)$”. Furthermore, the pair “$-ABS(ABS(...))$” and “$ABS(...)$” always returns different signed values, likewise the pair “$ABS(-ABS(...))$” and “$-ABS(...)$”. Hence they can be easily distinguished.

3.5 Experimental Study

3.5.1 Subject SQL Statements

In this experiment, I use the SQL statements developed by the Information Technology Laboratory of the National Institute of Standards and Technology (NIST)[79]. The NIST SQL statements are originally found in test programs described in [29]. These SQL statements have been widely used by many researchers. For example, Tuya et al. [1] use this NIST test suite for their studies. Furthermore, these NIST SQL statements are used by McCormick et al. [80] to compare the fault detection capabilities of a real world database application test suite to those of an SQL vendor (Oracle, MySql, Microsoft SQL server) based on mutation score. These SQL statements are classified into four categories. They are:

- **Entry SQL**: This covers the most basic features of SQL. It is a minor enhancement over the requirements of FIPS \textsuperscript{4} PUB \textsuperscript{5} 127-1

- **Transitional SQL**: These represent more advanced features of the SQL level of conformance. Major database vendors have features of this level [1]

- **Intermediate SQL**: This supports more advanced features than transitional SQL

\textsuperscript{4}FIPS-Federal Information Processing Standard
\textsuperscript{5}PUB-Publication
Chapter 3. Test Case Generation for ABS Operator

- Full SQL: This supports the most SQL features.

Among these NIST SQL statements, 19 of them involves numeric expressions or fields. These 19 statements are used as the original SQL statements for mutant generation.

### 3.5.2 Mutant Generation

In this experiment, I generate the ABS mutants for each subject SQL statement with the help of SQLMutation tool [29] and some extra mutants according to my classification in Section 3.3. Note that, for type 2 and Type 5 ABS() mutant, the SQLMutation tool misses some mutants according to my classification. For example, in Type 2, two possible mutants of “SELECT MAX(grade) FROM STAFF” are “SELECT MAX(ABS(grade)) FROM STAFF” and “SELECT ABS(MAX(grade)) FROM STAFF.” The tool generates the first mutant but fails to generate the second one. According to this observation, the second type of mutant is also possible and these two mutants are different. For example, a test suite with “grade” = {-3, -5}, the MAX(ABS(grade)) returns 5 while the ABS(MAX(grade)) returns 3, hence they are different. In total, I find 68 mutants. Table 3.5 shows the distribution of mutants for the variant types of ABS mutation operators. Among these 68 mutants, 1 mutant is equivalent to its original SQL statement. As a consequence, there are 67 non-equivalent mutants in this experiment. Table 3.5 also shows that, no mutant is generated for the Type 6 of ABS mutation operator, as none of the subject SQL statements are applicable under that type of ABS mutation operator.

### 3.5.3 Test Dataset Generation Procedure

The guidelines proposed in the chapter are different for the six types of SQL query (shown in Table 3.2). So, in order to generate the test dataset, I manually classify all the original SQL statements into five categories so that the mutants, generated from these original SQL statements also fall within the same classification. For each type of mutant, I generate the test datasets according to corresponding guidelines.
Table 3.5: Number of Mutants for each Type Against Original SQL

<table>
<thead>
<tr>
<th>Type</th>
<th>Original SQL</th>
<th>Scenario</th>
<th>Generated # of Mutants</th>
<th>Equiv. to Original</th>
<th>Non Equiv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>10</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>S1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S2</td>
<td>24</td>
<td>-</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>S1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S2</td>
<td>10</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>S1-(a)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S2-(a)</td>
<td>12</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>S1-(b)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S2-(b)</td>
<td>12</td>
<td>-</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>S1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td>67</td>
</tr>
</tbody>
</table>

3.5.4 Experimental Process and Data Collection

For each type of mutant, I execute the corresponding test dataset. Next, I check whether the test dataset can kill the mutants from its original SQL statements or not. I count the number of killed mutants and finally calculate the mutation score with the number of killed mutants as a percentage of total number of non-equivalent mutants.

3.5.5 Results Analysis

Table 3.6 illustrates the number of killed mutants against their corresponding number of non equivalent mutants. It also shows the mutation score for different scenarios of each type. For all five variants\(^6\), the test cases generated through my guidelines kill all non equivalent mutants. As a result, from the Table 3.6, it is clear that for all available types of ABS mutant, the test datasets kill 100% of mutants.

\(^6\)As mentioned earlier, only 5 variants of the ABS mutants can be generated from the selected NIST SQL statements
Table 3.6: Results of killed Mutants according to Guidelines

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th># of Mutants</th>
<th>Non Equiv.</th>
<th>Killed</th>
<th>Mutation Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>10</td>
<td>10</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>24</td>
<td>24</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>10</td>
<td>10</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S1-(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S1-(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S2-(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S2-(b)</td>
<td>12</td>
<td>12</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>67</td>
<td>67</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

Test cases generated through these guidelines kill all $ABS$ mutants. Thus, it can be said that test cases selected through my guidelines are adequate for testing the five variants as shown in Table 3.1.

### 3.6 Threats and Validity

The external validity of this experiment is the selection of the SQL statements. I have selected 19 original SQL statements from the NIST. Although there is a large number of SQL queries in NIST, not all the queries have the same features that I have proposed. For instance, in this experiment, SQL statements with join, inner join, outer join, sub-query, query with ‘group by’ keyword are avoided. Moreover, in this chapter as I am proposing guidelines for the $ABS$ mutation operator, the SQL statements need numeric fields. However, numeric field does not exist in all SQL statements. As a result, I have selected only 19 original SQL statements. These 19 SQL statements falls into five of the 11 possible variants. Because of the small number of original queries the number of the $ABS$ mutants is also small. Since the results of this experiment enable me to generalize them across different database applications, it would be useful to select more SQL queries so that each scenario can incorporate some mutants.
The internal validity of this experiment is the selection of a test suite. According to my guidelines, a test suite consists of positive and negative values in the same dataset or separate datasets. Due to table constraints, if the negative or positive values cannot be set as the dataset, a number of mutants may not be killed with my guidelines. As a result the mutation score may decrease which may lead to weakness in my proposed guidelines.

The construct validity of this experiment is some mutants are missed by the SQLMutation tool for my proposed guidelines for \textit{ABS} mutation operator. For a wide range of original SQL statement if the number of missed mutants generated by the tool is unknown, the result may not be exact.

3.7 Summary

A summary of this chapter is as follows.

- The importance of \textit{ABS} mutants in SQL statements is explained.
- Six different types of mutants are identified with altogether 11 variants.
- For each of these variants a guideline has been proposed.
- An experiment is performed to validate my guidelines. According to guidelines different sets of test suites were designed. And the mutants are executed against these test suites to calculate the mutation score. For this experiment, all mutants are killed.
Chapter 4

Test Case Generation for IRC Operator

In this chapter, I investigate how an identifier column replacement (IRC) mutation operator can be applied on SELECT SQL statements to create various types of IRC mutants. Unlike the situation of the ABS mutation operator, the IRC mutation operator can also be applied on both numeric and non-numeric fields, this adds further complications in classifying possible IRC mutants related to numeric and non-numeric fields. There are altogether 24 different types of IRC mutants with 44 different variants. For each of these variants, I develop a test case generation guideline to generate test cases that can kill the corresponding mutants. Finally, I evaluate the effectiveness of the guidelines via an experiment using NIST Database conformance test suite [79].

Similar to the previous chapter, for ease of discussion and illustration, I use the example of the database schema STAFF given in Figure 3.1 throughout the rest of this chapter.

4.1 Identifier Column Replacement (IRC) Operator

While developing database applications, programmers may make mistakes in writing SQL queries. When they incorrectly use a field different from the original intended one, an
IRC mutant is created. As a result, this causes the application to yield wrong results.

For example, given the database schema of the STAFF database table in Figure 3.1, if a developer needs to retrieve the details (e.g. name, grade and salary) of those staffs whose salary are greater than $3000 from the STAFF table, a possible SQL statement is “SELECT name, grade, salary FROM STAFF WHERE salary > 3000” (denoted as $S_1$ for ease of reference in this section). In case, a developer makes a mistake by writing “grade” instead of “salary” (or, “staffid” instead of “name”), the output is wrong. Hence, it leads to incorrect results.

The definition of the identifier column replacement mutation operator ($IRC$), is first introduced by Tuya and his colleagues [1] as follows:

With $IRC$ mutation operator, a column reference in an SQL statement can be replaced by another column reference, constant or parameter that (1) appears in the SQL statement and (2) is type compatible with the original column.

For example, a possible $IRC$ mutant of $S_1$ can be achieved by replacing “salary” by “grade”. The mutant will then be “SELECT name, grade, grade FROM STAFF WHERE salary $>$ 3000”. Another possible mutant can be obtained by replacing “salary” in $S_1$ with 3000. However, $IRC$ mutants could not be generated from $S_1$ by replacing “salary” with “age” nor 2000 because they do not appear in the original SQL statement. Moreover, according to [1], one could not replace “salary” by “name” to create an $IRC$ mutant because these two fields are incompatible.

### 4.2 Twenty Four Types of $IRC$ Mutants

In this section, I explore and explain how many different types of $IRC$ mutants could be generated from a simple SQL statement. Similar to the discussion of $ABS$ mutants in Chapter 3, simple SQL statements are statements that may involve the use of aggregate functions in the SELECT clause and the use of the WHERE clause with one simple condition. Depending on whether there are aggregate functions in the SELECT clause and there is a
Table 4.1: Possible Types of Original SQL Statement

<table>
<thead>
<tr>
<th>Type</th>
<th>simple</th>
<th>aggregate</th>
<th>WHERE clause in SELECT clause</th>
<th>Example (identifier for ease of reference in this chapter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>No</td>
<td>No</td>
<td></td>
<td>SELECT staffid, name FROM STAFF (O1-1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT grade, salary FROM STAFF (O1-2)</td>
</tr>
<tr>
<td>O2</td>
<td>No</td>
<td>Yes</td>
<td></td>
<td>SELECT MAX(grade), MIN(salary) FROM STAFF (O2-1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT MAX(staffid), MIN(name) FROM STAFF (O2-2)</td>
</tr>
<tr>
<td>O3</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td>SELECT grade, salary FROM STAFF WHERE grade &gt; 40 (O3-1a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT grade, salary FROM STAFF WHERE age &gt; 30 (O3-1b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT grade, salary FROM STAFF WHERE staffid = 'S1' (O3-1c)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT staffid, name FROM STAFF WHERE grade &gt; 40 (O3-2a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT staffid, name FROM STAFF WHERE staffid = 'S1' (O3-2b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT staffid, name FROM STAFF WHERE city = 'C8' (O3-2c)</td>
</tr>
<tr>
<td>O4</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>SELECT MAX(grade), MIN(salary) FROM STAFF WHERE grade &gt; 40 (O4-1a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT MAX(grade), MIN(salary) FROM STAFF WHERE age &gt; 30 (O4-1b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT MAX(staffid), MIN(name) FROM STAFF WHERE staffid = 'S1' (O4-1c)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT MAX(staffid), MIN(name) FROM STAFF WHERE grade &gt; 40 (O4-2a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT MAX(staffid), MIN(name) FROM STAFF WHERE staffid = 'S1' (O4-2b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SELECT MAX(staffid), MIN(name) FROM STAFF WHERE city = 'C8' (O4-2c)</td>
</tr>
</tbody>
</table>

simple condition in the WHERE clause, there are 4 different forms of SQL statements that can be used as the original SQLs for mutant generation. Table 4.1 summarizes all these 4 forms of original SQL statements, namely O1, O2, O3 and O4. In addition, it shows several examples of SQL statements that may be used to generate different types of IRC mutants. Even though these original SQL statements are very simple, there are altogether 24 types of IRC mutants depending on whether the field involved in the IRC mutation operator and the field involved in the WHERE clause are numeric or non-numeric. Table 4.2 summarizes all these 24 types of IRC mutants, namely IRC-m1, IRC-m2, ..., IRC-m24. Table 4.3 gives an example of each of these mutants with reference to different original SQL statements given in Table 4.1.
Table 4.2: Different Types of IRC Mutant

<table>
<thead>
<tr>
<th>Type</th>
<th>Origin</th>
<th>IRC occurs in</th>
<th>IRC-Field&lt;sup&gt;b&lt;/sup&gt; type</th>
<th>Field type in WHERE clause</th>
<th>IRC-Field exists in</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRC-m1</td>
<td>O1</td>
<td>S</td>
<td>Numeric</td>
<td>n/a&lt;sup&gt;d&lt;/sup&gt;</td>
<td>S&lt;sup&gt;e&lt;/sup&gt;</td>
</tr>
<tr>
<td>IRC-m2</td>
<td>O1</td>
<td>S</td>
<td>Non-numeric</td>
<td>n/a</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m3</td>
<td>O2</td>
<td>A</td>
<td>Numeric</td>
<td>n/a</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m4</td>
<td>O2</td>
<td>A</td>
<td>Non-numeric</td>
<td>n/a</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m5</td>
<td>O3</td>
<td>S</td>
<td>Numeric</td>
<td>Numeric</td>
<td>B</td>
</tr>
<tr>
<td>IRC-m6</td>
<td>O3</td>
<td>S</td>
<td>Numeric</td>
<td>Numeric</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m7</td>
<td>O3</td>
<td>S</td>
<td>Numeric</td>
<td>Non-numeric</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m8</td>
<td>O3</td>
<td>S</td>
<td>Non-numeric</td>
<td>Numeric</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m9</td>
<td>O3</td>
<td>S</td>
<td>Non-numeric</td>
<td>Non-numeric</td>
<td>B</td>
</tr>
<tr>
<td>IRC-m10</td>
<td>O3</td>
<td>S</td>
<td>Non-numeric</td>
<td>Non-numeric</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m11</td>
<td>O3</td>
<td>W</td>
<td>Numeric</td>
<td>Numeric&lt;sup&gt;f&lt;/sup&gt;</td>
<td>B</td>
</tr>
<tr>
<td>IRC-m12</td>
<td>O3</td>
<td>W</td>
<td>Numeric</td>
<td>Numeric</td>
<td>W</td>
</tr>
<tr>
<td>IRC-m13</td>
<td>O3</td>
<td>W</td>
<td>Non-numeric</td>
<td>Non-numeric</td>
<td>B</td>
</tr>
<tr>
<td>IRC-m14</td>
<td>O3</td>
<td>W</td>
<td>Non-numeric</td>
<td>Non-numeric</td>
<td>W</td>
</tr>
<tr>
<td>IRC-m15</td>
<td>O4</td>
<td>A</td>
<td>Numeric</td>
<td>Numeric</td>
<td>B</td>
</tr>
<tr>
<td>IRC-m16</td>
<td>O4</td>
<td>A</td>
<td>Numeric</td>
<td>Numeric</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m17</td>
<td>O4</td>
<td>A</td>
<td>Numeric</td>
<td>Non-numeric</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m18</td>
<td>O4</td>
<td>A</td>
<td>Non-numeric</td>
<td>Numeric</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m19</td>
<td>O4</td>
<td>A</td>
<td>Non-numeric</td>
<td>Non-numeric</td>
<td>B</td>
</tr>
<tr>
<td>IRC-m20</td>
<td>O4</td>
<td>A</td>
<td>Non-numeric</td>
<td>Non-numeric</td>
<td>S</td>
</tr>
<tr>
<td>IRC-m21</td>
<td>O4</td>
<td>W</td>
<td>Numeric</td>
<td>Numeric</td>
<td>B</td>
</tr>
<tr>
<td>IRC-m22</td>
<td>O4</td>
<td>W</td>
<td>Numeric</td>
<td>Numeric</td>
<td>W</td>
</tr>
<tr>
<td>IRC-m23</td>
<td>O4</td>
<td>W</td>
<td>Non-Numeric</td>
<td>Non-Numeric</td>
<td>B</td>
</tr>
<tr>
<td>IRC-m24</td>
<td>O4</td>
<td>W</td>
<td>Non-Numeric</td>
<td>Non-Numeric</td>
<td>W</td>
</tr>
</tbody>
</table>

<sup>a</sup>S: SELECT clause, A: AGGREGATE function in SELECT clause, W: WHERE clause
<sup>b</sup>IRC-Field is the field that is changed by the IRC mutation operator
<sup>c</sup>S: SELECT clause only, W: WHERE clause only, B: both SELECT and WHERE clauses
<sup>d</sup>Both O1 and O2 do not have WHERE clause.
<sup>e</sup>Since IRC occurs in SELECT clause, the field must be in SELECT clause.
<sup>f</sup>Since the IRC operator occurs in the WHERE clause, the field in the simple WHERE clause must be the same as the IRC-Field.

In the rest of this section, I discuss how to classify such 24 types of IRC mutants in details.
Table 4.3: Examples of Various IRC Mutants

<table>
<thead>
<tr>
<th>Mutant</th>
<th>Example</th>
<th>cf. with original SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRC-m1</td>
<td><code>SELECT grade, grade FROM STAFF</code></td>
<td>O1-1, salary→grade&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>IRC-m2</td>
<td><code>SELECT name, name FROM STAFF</code></td>
<td>O1-2, staffid→name</td>
</tr>
<tr>
<td>IRC-m3</td>
<td><code>SELECT MAX(grade), MIN(grade) FROM STAFF</code></td>
<td>O2-1, salary→grade</td>
</tr>
<tr>
<td>IRC-m4</td>
<td><code>SELECT MAX(name), MIN(name) FROM STAFF</code></td>
<td>O2-2, staffid→name</td>
</tr>
<tr>
<td>IRC-m5</td>
<td><code>SELECT salary, salary FROM STAFF WHERE grade &gt; 40</code></td>
<td>O3-1a, grade→salary</td>
</tr>
<tr>
<td>IRC-m6</td>
<td><code>SELECT grade, grade FROM STAFF WHERE age &gt; 30</code></td>
<td>O3-1b, salary→grade</td>
</tr>
<tr>
<td>IRC-m7</td>
<td><code>SELECT grade, grade FROM STAFF WHERE staffid = 'S1'</code></td>
<td>O3-1c, salary→grade</td>
</tr>
<tr>
<td>IRC-m8</td>
<td><code>SELECT name, name FROM STAFF WHERE grade &gt; 40</code></td>
<td>O3-2a, staffid→name</td>
</tr>
<tr>
<td>IRC-m9</td>
<td><code>SELECT name, name FROM staff WHERE staffid = 'S1'</code></td>
<td>O3-2b, staffid→name</td>
</tr>
<tr>
<td>IRC-m10</td>
<td><code>SELECT name, name FROM staff WHERE staffid = 'C8'</code></td>
<td>O3-2c, staffid→name</td>
</tr>
<tr>
<td>IRC-m11</td>
<td><code>SELECT grade, grade FROM STAFF WHERE salary &gt; 40</code></td>
<td>O3-1a, grade→salary</td>
</tr>
<tr>
<td>IRC-m12</td>
<td><code>SELECT grade, grade FROM STAFF WHERE salary &gt; 30</code></td>
<td>O3-1b, grade→salary</td>
</tr>
<tr>
<td>IRC-m13</td>
<td><code>SELECT staffid, name FROM STAFF WHERE name = 'S2'</code></td>
<td>O3-2a, staffid→name</td>
</tr>
<tr>
<td>IRC-m14</td>
<td><code>SELECT staffid, name FROM STAFF WHERE name = 'C8'</code></td>
<td>O3-2b, city→name</td>
</tr>
<tr>
<td>IRC-m15</td>
<td><code>SELECT MAX(salary), MIN(salary) FROM STAFF</code></td>
<td>O4-1a, salary→grade</td>
</tr>
<tr>
<td></td>
<td>WHERE grade &gt; 40</td>
<td></td>
</tr>
<tr>
<td>IRC-m16</td>
<td><code>SELECT MAX(grade), MIN(grade) FROM STAFF</code></td>
<td>O4-1b, salary→grade</td>
</tr>
<tr>
<td></td>
<td>WHERE staffid = 'S1'</td>
<td>O4-1c, salary→grade</td>
</tr>
<tr>
<td>IRC-m17</td>
<td><code>SELECT MAX(grade), MIN(grade) FROM STAFF</code></td>
<td>O4-2a, staffid→name</td>
</tr>
<tr>
<td></td>
<td>WHERE grade &gt; 40</td>
<td>O4-2b, staffid→name</td>
</tr>
<tr>
<td>IRC-m18</td>
<td><code>SELECT MAX(name), MIN(name) FROM STAFF</code></td>
<td>O4-2b, staffid→name</td>
</tr>
<tr>
<td></td>
<td>WHERE staffid = 'S1'</td>
<td>O4-2c, staffid→name</td>
</tr>
<tr>
<td>IRC-m19</td>
<td><code>SELECT MAX(name), MIN(name) FROM staff</code></td>
<td>O4-2c, city→name</td>
</tr>
<tr>
<td></td>
<td>WHERE name = 'C8'</td>
<td></td>
</tr>
<tr>
<td>IRC-m20</td>
<td><code>SELECT MAX(name), MIN(name) FROM staff</code></td>
<td>O4-1a, grade→salary</td>
</tr>
<tr>
<td></td>
<td>WHERE salary &gt; 40</td>
<td>O4-1b, age→salary</td>
</tr>
<tr>
<td>IRC-m21</td>
<td><code>SELECT MAX(grade), MIN(salary) FROM STAFF</code></td>
<td>O4-1b, grade→salary</td>
</tr>
<tr>
<td></td>
<td>WHERE salary &gt; 40</td>
<td></td>
</tr>
<tr>
<td>IRC-m22</td>
<td><code>SELECT MAX(grade), MIN(salary) FROM STAFF</code></td>
<td>O4-2b, staffid→name</td>
</tr>
<tr>
<td></td>
<td>WHERE staffid = 'S1'</td>
<td>O4-2c, city→name</td>
</tr>
<tr>
<td>IRC-m23</td>
<td><code>SELECT MAX(staffid), MIN(name) FROM STAFF</code></td>
<td>O4-2c, city→name</td>
</tr>
<tr>
<td></td>
<td>WHERE name = 'C8'</td>
<td></td>
</tr>
</tbody>
</table>

<sup>a</sup>This example mutant can be obtained from O1-1 in Table 4.1 by replacing “salary” with “grade”.

4.2.1 Original SQL without Aggregate Function and WHERE Clause

I first discuss those IRC mutants that are generated from the original SQL statements without an aggregate function and a WHERE clause. Therefore, the IRC mutation operator can only occur in those fields (or, columns) in the SELECT clause. There are two types of IRC mutants in this group. They are

IRC-m1 This type of mutants occurs when an IRC mutation operator replaces a numeric field by another numeric field that appears in the original SQL statement. For example, for O1-1 in Table 4.1, when the IRC mutation operator replaces “salary” with “grade”, the resulting IRC mutant becomes “SELECT grade, grade FROM STAFF”.

IRC-m2 This type of mutants occurs when an IRC mutation operator replaces a non-numeric field by another non-numeric field that appears in the original SQL statement. For example, for O1-2 in Table 4.1, when the IRC mutation operator replaces “staffid” with “name”, the resulting IRC mutant becomes “SELECT name, name FROM STAFF”.

4.2.2 Original SQL with Aggregate Function but without WHERE clause

In this section, I discuss those IRC mutants created from those original SQL statements that have aggregate functions (such as \texttt{MAX()}, \texttt{MIN()}, \texttt{SUM()}, \texttt{AVG()} and \texttt{COUNT()}) in the SELECT clause but do not have any WHERE clause. Similar to the situation above, the IRC mutation operator can only occur in those fields (or, columns) involved in the aggregate function(s) in the SELECT clause. There are two types of IRC mutants in this group. They are as follows.

IRC-m3 This type of mutant occurs when an IRC mutation operator replaces a numeric field by another numeric field that appears in the original SQL statement. For example, for O2-1 in Table 4.1, when the IRC mutation operator replaces...
“salary” with “grade”, the resulting IRC mutant becomes “SELECT MAX(grade), MIN(grade) FROM staff”\(^1\).

IRC-m4 Unlike the IRC-m3, this type of mutant occurs when an IRC mutation operator replaces a non-numeric field by another non-numeric field that appears in the original SQL statement. For example, for O2-2 in Table 4.1, when the IRC mutation operator replaces “staffid” with “name”, the resulting IRC mutant becomes “SELECT MAX(name), MIN(name) FROM STAFF”.

4.2.3 Original SQL with WHERE Clause but without Aggregate Function

I now discuss IRC mutants created from those original SQL statements having a simple WHERE clause but there is no aggregate function in the SELECT clause. Please be reminded that the IRC mutation operator may occur in the SELECT clause or in the WHERE clause.

4.2.3.1 IRC in SELECT Clause

When an IRC mutation operator occurs in the SELECT clause, the following six types of mutant are possible.

IRC-m5 This type of mutant occurs when an IRC mutation operator replaces a numeric field in the SELECT clause by another numeric field that appears in the original SQL statement and the field being replaced by the IRC mutation operator also exists in the WHERE clause. For example, for O3-1a in Table 4.1, when the IRC mutation operator replaces “grade” with “salary”, the resulting IRC mutant becomes “SELECT salary, salary FROM STAFF WHERE grade > 40”.

IRC-m6 This type of mutant occurs when an IRC mutation operator replaces a numeric field in the SELECT clause by another numeric field that appears in the original SQL statement and the IRC-field does not exist in the WHERE clause. For example,

\(^{1}\)Due to the syntax of SQL statement, once an aggregate function appears in the SELECT clause, all fields in the SELECT clause must be aggregate expressions. The only exception is when there is a GROUP BY clause. Since I do not consider GROUP BY clause in this thesis, the relevant discussion is out of the scope of this thesis.
for O3-1b in Table 4.1, when the IRC mutation operator replaces “salary” with “grade”, the resulting IRC mutant becomes “\texttt{SELECT grade, grade FROM STAFF WHERE age > 30}”.

\textbf{IRC-m7} This type of mutant occurs when an IRC mutation operator replaces a numeric field in the \texttt{SELECT} clause by another numeric field that appears in the original SQL statement and the \texttt{WHERE} clause only involves non-numeric field(s). For example, for O3-1c in Table 4.1, when the IRC mutation operator replaces the numeric field “salary” with “grade”, the resulting IRC mutant becomes “\texttt{SELECT grade, grade FROM STAFF WHERE staffid = ‘S1’}”.

\textbf{IRC-m8} This type of mutant occurs when an IRC mutation operator replaces a non-numeric field in the \texttt{SELECT} clause by another non-numeric field that appears in the original SQL statement but the \texttt{WHERE} clause only involves numeric field(s). For example, for O3-2a in Table 4.1, when the IRC mutation operator replaces the non-numeric field “staffid” with “name”, the resulting IRC mutant becomes “\texttt{SELECT name, name FROM STAFF WHERE grade > 40}”.

\textbf{IRC-m9} This type of mutant occurs when an IRC mutation operator replaces a non-numeric field in the \texttt{SELECT} clause by another non-numeric field that appears in the original SQL statement and the IRC-field also exists in the \texttt{WHERE} clause. For example, for O3-2b in Table 4.1, when the IRC mutation operator replaces “staffid” with “name”, the resulting IRC mutant becomes “\texttt{SELECT name, name FROM staff WHERE staffid = ‘S1’}”.

\textbf{IRC-m10} This type of mutant occurs when an IRC mutation operator replaces a non-numeric field in the \texttt{SELECT} clause by another non-numeric field that appears in the original SQL statement but the \texttt{WHERE} clause involves a different non-numeric field. For example, for O3-2c in Table 4.1, when the IRC mutation operator replaces “staffid” with “name”, the resulting IRC mutant becomes “\texttt{SELECT name, name FROM STAFF WHERE city = ‘C8’}”.
4.2.3.2 IRC in WHERE Clause

When an IRC mutation operator occurs in the WHERE clause, the following four types of mutant are possible.

IRC-m11 This type of mutant occurs when an IRC mutation operator replaces a numeric field in the WHERE clause by another numeric field that appears in the original SQL statement and the IRC-field exists in the SELECT clause. For example, for O3-1a in Table 4.1, when the IRC mutation operator replaces “grade” with “salary” in the WHERE clause, the resulting IRC mutant becomes “SELECT grade, salary FROM STAFF WHERE salary > 40”.

IRC-m12 This type of mutant occurs when an IRC mutation operator replaces a numeric field in the WHERE clause by another numeric field that appears in the original SQL statement but the IRC-field does not exist in the SELECT clause. For example, for O3-1b in Table 4.1, when the IRC mutation operator replaces “age” by “salary”, the resulting IRC mutant becomes “SELECT grade, salary FROM STAFF WHERE salary > 30”.

IRC-m13 This type of mutant occurs when an IRC mutation operator replaces a non-numeric field in the WHERE clause by another non-numeric field that appears in the original SQL statement and the IRC-field exists in the SELECT clause. For example, for O3-2b in Table 4.1, when the IRC mutation operator replaces “staffid” with “name”, the resulting mutant becomes “SELECT staffid, name FROM STAFF WHERE name = ‘S1’ ”.

IRC-m14 This type of mutant occurs when an IRC mutation operator replaces a non-numeric field in the WHERE clause by another non-numeric field that appears in the original SQL statement but the IRC-field does not exist in the SELECT clause. For example, for O3-2c in Table 4.1, when the IRC mutation operator replaces “city” with “name”, the resulting mutant becomes “SELECT staffid, name FROM STAFF WHERE name = ‘C8’ ”.
4.2.4 Original SQL with WHERE Clause and Aggregate Functions

I now discuss the IRC mutants created from those original SQL statements having aggregate functions (such as $\text{MAX}()$, $\text{MIN}()$, $\text{SUM}()$, $\text{AVG}()$ and $\text{COUNT}()$) in the SELECT clause and a simple WHERE clause. Similar to the case discussed in previous section, the IRC mutation operator may occur in the aggregate function in the SELECT clause or in the field involved in the WHERE clause.

4.2.4.1 IRC in Aggregate Function in SELECT Clause

When an IRC mutation operator occurs in a field involved in an aggregate function of the SELECT clause, the following six types of mutants are possible.

IRC-m15 This type of mutant occurs when an IRC mutation operator replaces a numeric field in an aggregate function of the SELECT clause by another numeric field and the IRC-field also exists in the WHERE clause. For example, for O4-1a in Table 4.1, when the IRC mutation operator replaces “grade” with “salary”, the resulting IRC mutant becomes “SELECT $\text{MAX}(\text{salary})$, $\text{MIN}(\text{salary})$ FROM STAFF WHERE grade > 40”.

IRC-m16 This type of mutant occurs when an IRC mutation operator replaces a numeric field in an aggregate function in the SELECT clause by another numeric field that appears in the original SQL statement but the IRC-field does not exist in the WHERE clause. For example, for O4-1b in Table 4.1, when the IRC mutation operator replaces “salary” with “grade” the resulting IRC mutant becomes “SELECT $\text{MAX}(\text{grade})$, $\text{MIN}(\text{grade})$ FROM STAFF WHERE age > 30”.

IRC-m17 This type of mutant occurs when an IRC mutation operator replaces a numeric field in an aggregation function of the SELECT clause by another numeric field that appears in the original SQL statement but the WHERE clause only involves

\footnote{Once an aggregate function appears in the SELECT clause, all fields in the SELECT clause must be in aggregate expressions. The only exception is when there is a GROUP BY clause. Since, I do not consider GROUP BY clause in this thesis, the discussion is out of the scope of this thesis.}
non-numeric field. For example, for O4-1c in Table 4.1, when the IRC mutation operator replaces a numeric field “salary” with another numeric field “grade” while a non-numeric field “staffid” exists in the WHERE clause, the resulting IRC mutant becomes “\[SELECT max(grade), min(grade) FROM STAFF WHERE staffid = ‘S1’ \].

IRC-m18 This type of mutant occurs when an IRC mutation operator replaces a non-numeric field in an aggregate function of the SELECT clause by another non-numeric field that appears in the original SQL statement but the WHERE clause only involves numeric fields. For example, for O4-2a in Table 4.1, when the IRC mutation operator replaces the non-numeric field “staffid” with “name”, the resulting IRC mutant becomes “\[SELECT max(name), min(name) FROM STAFF WHERE grade > 40 \].

IRC-m19 This type of mutant occurs when an IRC mutation operator replaces a non-numeric field in an aggregate function of the SELECT clause by another non-numeric field that appears in the original SQL statement and the IRC-field also exists in the WHERE clause. For example, for O4-2b in Table 4.1, when the IRC mutation operator replaces “staffid” with “name”, the resulting IRC mutant becomes “\[SELECT max(name), min(name) FROM STAFF WHERE staffid = ‘S1’ \].

IRC-m20 This type of mutant occurs when an IRC mutation operator replaces a non-numeric field in an aggregate function of the SELECT clause by another non-numeric field that appears in the original SQL statement but the WHERE clause contains a different non-numeric field. For example, for O4-2c in Table 4.1, when the IRC mutation operator replaces “staffid” with “name”, the resulting IRC mutant becomes “\[SELECT max(name), min(name) FROM STAFF WHERE city = ‘C8’ \].

4.2.4.2 IRC in WHERE Clause

When an IRC mutation operator occurs in the WHERE clause, the following four types of mutant are possible.
Chapter 4. Test Case Generation for IRC Operator

IRC-m21 This type of mutant occurs when an IRC mutation operator replaces a numeric field in the WHERE clause by another numeric field that appears in the original SQL statement and the IRC-field exists in the SELECT clause. For example, for O4-1a in Table 4.1, when the IRC mutation operator replaces “grade” with “salary” in the WHERE clause, the resulting IRC mutant becomes “SELECT MAX(grade), MIN(salary) FROM STAFF WHERE salary > 40”.

IRC-m22 This type of mutant occurs when an IRC mutation operator replaces a numeric field in the WHERE clause by another numeric field that appears in the original SQL statement but the IRC-field does not exist in the SELECT clause. For example, for O4-1b in Table 4.1, when the IRC mutation operator replaces “age” by “salary”, the resulting IRC mutant becomes “SELECT MAX(grade), MIN(salary) FROM STAFF WHERE salary > 30”.

IRC-m23 This type of mutant occurs when an IRC mutation operator replaces a non-numeric field in the WHERE clause by another non-numeric field that appears in the SQL statement and the IRC-field exists in the SELECT clause. For example, for O4-2b in Table 4.1, when the IRC mutation operator replaces “staffid” with “name”, the resulting mutant becomes “SELECT MAX(staffid), MIN(name) FROM STAFF WHERE name = ‘S1’”.

IRC-m24 This type of mutant occurs when an IRC mutation operator replaces a non-numeric field in the WHERE clause by another non-numeric field that appears in the original SQL statement but the IRC-field does not exist in the SELECT clause. For example, for O4-2c in Table 4.1, when the IRC mutation operator replaces “city” with “name”, the resulting mutant becomes “SELECT MAX(staffid), MIN(name) FROM STAFF WHERE name = ‘C8’”.

4.3 Test Case Generation Guidelines

As discussed previously, there are altogether 24 types of IRC mutants as indicated and illustrated in Tables 4.2 to 4.3. In this section, I propose test case generation guidelines
for detecting each type of IRC mutants. The guidelines for IRC-m1, . . . , IRC-m24 will be denoted as G-IRC-m1, . . . , G-IRC-m24, respectively. For ease of discussion and illustration, I will use the database table “STAFF” as given in Figure 3.1, those original SQL statements in Table 4.1 and the relevant mutants in Table 4.3 as examples throughout the rest of this section. Again, the discussion is subdivided into 4 subsections based on the original SQL statements.

4.3.1 Guidelines for Mutants from SQL without Aggregate Function and WHERE clause

There are two types of mutant, namely IRC-m1 and IRC-m2, as discussed earlier. The test case generation guidelines for each of these mutants are discussed below.

G-IRC-m1 When the IRC mutation operator replaces a numeric field in the SELECT clause by another numeric field that appears in original SQL statement, a mutant of IRC-m1 is generated. The resultant mutant will have two identical fields in the SELECT clause\(^3\). To distinguish the results of the original SQL (say, O1-1) from those of the mutants (say, IRC-m1), one needs to set the values in the database table so that the values of those numeric fields are distinct for every record in the table. Thus, the results of the original SQL statements will have different values in the relevant fields whereas those of the IRC-m1 mutants will have duplicate values in the corresponding fields.

For example, two possible mutants of O1-1 in Table 4.1 (that is, “SELECT grade, salary FROM STAFF”) are “SELECT grade, grade FROM STAFF”, denoted as M1-1, and “SELECT salary, salary FROM STAFF”, denoted as M1-2. By setting the values of “grade” and “salary” to different values (e.g. \{(grade = 5, salary = 1500), (8, 3000), (10, 4000), . . .\}), one can distinguish the results of the original SQL statements from those of the mutants because O1-1 returns \{(5, 1500), (8, 3000), (10, 4000), . . .\} whereas M1-1 returns \{(5, 5), (8, 8), (10, 10), . . .\} and M1-2 returns \{(1500, 1500), (3000, 3000), (4000, 4000), . . .\}.

\(^3\)Please refer to the relevant example in Table 4.3.
G-IRC-m2  Similar to IRC-m1, mutants of IRC-m2 have “duplicated” non-numeric columns in the SELECT clause. Similarly, the guideline is to set the values in the database table so that the values of those non-numeric fields are different for every record in the table. Thus the results of the original SQL statements will have distinct values in the relevant fields whereas those of the IRC-m2 mutants will have duplicate values in the corresponding fields.

For example, two possible mutants of O1-2 in Table 4.1 (that is, “SELECT staffid, name FROM STAFF”) are “SELECT name, name FROM STAFF”, denoted as M2-1, and “SELECT staffid, staffid FROM STAFF”, denoted as M2-2. By setting the values of “staffid”, and “name” to distinct values (e.g. \{("staffid"="S1", "name"="Jake"), ("S2", "David"), ("S3", "Michael")\}), one can distinguish the original SQL statement from those mutants because O1-2 will return \{("S1", "Jake"), ("S2", "David"), ("S3", "Michael")\} whereas M2-1 returns \{("Jake", "Jake"), ("David", "David"), ("Micheal", "Micheal")\} and M2-2 returns \{("S1", "S1"), ("S2", "S2"), ("S3", "S3")\}.

4.3.2  Guidelines for Mutants from SQL with Aggregate Function but without WHERE clause

There are two types of mutant in this group namely IRC-m3 and IRC-m4. The test case generation guidelines for each of these mutants are discussed below.

G-IRC-m3: As this type of IRC mutant involves numeric fields, any aggregate functions can be applied on these numeric fields to achieve the desired goal of the program. For ease of discussion, two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function COUNT() and in the second set, the guidelines are about other aggregate functions. The test case generation guidelines for these two sets are as follows:
1. (a) Situation when the IRC field is in the aggregate function \( \text{COUNT}() \). Since, \( \text{COUNT}() \) returns the number of non-null values in the required field, when the fields involved in the IRC mutation operator do not allow null values, there is no way to distinguish the original SQL statement from its mutants. In other words, the mutants are equivalent to its original SQL statement. However, when the fields involved allow null values, one can distinguish the original SQL statement from its mutants by setting different null values in those numeric fields so that the results are different. Hence, the guideline is to set those numeric fields that allow null values so that they will have different null values.

For example, one possible mutant of "SELECT COUNT(salary), \( \text{MAX}(\text{grade}) \) FROM STAFF" is "SELECT COUNT(\text{grade}), \( \text{MAX}(\text{grade}) \) FROM STAFF". In order to distinguish "\( \text{COUNT}(\text{salary}) \)" from "\( \text{COUNT}(\text{grade}) \)" it needs different number of null values for the "grade" and the "salary" in the database table. By setting the values of "grade" and "salary" to different null values (e.g. \{("grade" = 3, "salary" =3), (4, 5), (5, null)\}), one can distinguish the results of the original SQL statement from those mutants because the original SQL statement returns "2, 5" whereas the mutant returns "3, 5".

(b) In the case of \( \text{COUNT}(%\text{DISTINCT} \ldots) \), the test case generation guideline is different from the case of \( \text{COUNT}() \). First, please note that when all the values of selected fields are different, the behaviour of \( \text{COUNT}(%\ldots) \) is the same as \( \text{COUNT}(%\text{DISTINCT} \ldots) \). When the fields involved in the IRC mutation operator allow null values, it is possible to distinguish the mutants by setting different number of null values to the field involve in the SQL statement. Second, unlike the situation in \( \text{COUNT}() \), if the fields cannot be null, one can still distinguish the mutants with \( \text{COUNT}(%\text{DISTINCT} \ldots) \) from its original SQL by setting the values in the numeric fields so that these fields have different number of distinct values.

For example, when the IRC mutation operator replaces "salary" by "grade" in the SQL statement, "SELECT COUNT(DISTINCT salary), \( \text{MAX}(\text{grade}) \) FROM STAFF", the IRC mutant becomes "SELECT COUNT(DISTINCT grade), \( \text{MAX}(\text{grade}) \) FROM STAFF". By setting the values in the fields of "grade" and "salary" so
that they will have different number of distinct values, (e.g. \{(grade = 1, salary = 3), (2, 3), (3, 2)\}). One can distinguish the results of the mutants from its original SQL statement because \texttt{COUNT(DISTINCT salary)} returns 2 whereas \texttt{COUNT(DISTINCT grade)} returns 3. Hence, the returned results by the SQL statements are different.

2. Situation when the IRC field is in an aggregate function other than \texttt{COUNT()} (i.e. \texttt{MAX()}, \texttt{MIN()}, \texttt{SUM()}, \texttt{AVG()}), instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an IRC mutation operator replaces a numeric field \texttt{f1} in the aggregate function \texttt{AGGR()} (may be \texttt{MAX()}, \texttt{MIN()}, ...) by another numeric field \texttt{f2}, the IRC mutant will have \texttt{AGGR(f2)} instead of \texttt{AGGR(f1)}. In order to distinguish between \texttt{AGGR(f1)} and \texttt{AGGR(f2)}, the guideline consists of the following two steps:

(a) Set any non-zero values for \texttt{f1} in the records of the database and

(b) Set \texttt{f2} so that it is equal to corresponding values in \texttt{f1} plus a fixed non-zero constant \texttt{C} for each record in the database table. In other word, \texttt{f2} = \texttt{f1} + \texttt{C}.

It does not matter whether there is a \texttt{DISTINCT} keyword in the SQL statement, the same guideline is adequate to distinguish the mutant from the original SQL statement.

For example, one possible mutant of O2-1 in Table 4.1 (that is, \texttt{SELECT MAX(grade), MIN(salary) FROM STAFF}) is \texttt{SELECT MAX(grade), MIN(grade) FROM STAFF"}, denoted as M3-1. By setting the values for “grade” and “salary” in such a way so that salary= grade + 1000 such as \{(“grade”=3, “salary”=1003), (4, 1004), (5, 1005)\}, one can distinguish the mutant from its original SQL statement because O2-1 returns “5, 1003” whereas M3-1 returns “5, 3”.

\textbf{G-IRC-m4}: As this type of IRC mutant involves non-numeric fields, \texttt{MAX()}, \texttt{MIN()} and \texttt{COUNT()} are the only legitimate aggregate functions. Similar to G-IRC-m3, depending on the characteristics of aggregate functions, the guidelines are discussed in two scenarios. The first scenario discusses those mutants related to the aggregate function \texttt{COUNT()}. The
second scenario discusses mutants related to $\text{MAX}()$ and $\text{MIN}()$ aggregate functions. The test case generation guidelines for these two scenarios are as follows:

1. (a) Situation when $\text{IRC}$-field is in the aggregate function $\text{COUNT}()$. Similar to the previous guideline G-IRC-m3(1a), when the fields involved in the $\text{IRC}$ mutation operator do not allow null values, the mutants will be equivalent to the original SQL statement. However, when the fields involved allow null values, one can distinguish the original SQL statement from its mutant by setting different number of null values in those non-numeric fields so that the results are different. Hence, the guideline is to set those non-numeric fields that allow null values so that they will have different null values.

For example, one possible mutant of $\text{SELECT COUNT(}\text{staffid}), \text{MAX(name) FROM STAFF}$ is $\text{SELECT COUNT(}\text{name}), \text{MAX(name) FROM STAFF}$. In order to distinguish $\text{COUNT(staffid)}$ from $\text{COUNT(name)}$ one needs to set different number of null values for “staffid” and “name” in the database table. By setting the values of “staffid” and “name” to different null values (e.g. \{\{“staffid”=’S1’, “name”=’Jack’\}, \{null, ‘David’\}, \{’S2’, ‘Adam’\}\}), it is possible to distinguish the mutants from its original SQL statement because the original SQL statement returns “2, ‘Jack’ ” whereas the mutant returns “3, ‘Jack’ ”.

(b) For the case of $\text{COUNT(DISTINCT ...)}$, the test case generation guidelines are different from the case of $\text{COUNT()}$. Similar to previous guidelines G-IRC-m3(1b), when all the values of selected fields are different, the behaviour of $\text{COUNT(...)}$ is the same as $\text{COUNT(DISTINCT ...)}$. In that case same guideline is applicable for the SQL statements involved in $\text{COUNT(DISTINCT ...)}$. However, unlike the situation in $\text{COUNT(...)}$, if the fields cannot be null, one can still distinguish the mutants with $\text{COUNT(DISTINCT ...)}$ from its original SQL by setting the values in the non-numeric fields so that these fields have different number of distinct values.

For example, when the $\text{IRC}$ mutation operator replaces “staffid” by “name” in the SQL statement, “$\text{SELECT COUNT(DISTINCT staffid), MAX(name) FROM STAFF}$”, the $\text{IRC}$ mutant becomes “$\text{SELECT COUNT(DISTINCT name), MAX(name)}$"
FROM STAFF”. By setting the values in the fields of “staffid” and “name” so that they will have different number of distinct values, (e.g. \{(“staffid” = ‘S1’, “name” = ‘Jack’), (‘S2’, ‘Jack’), (‘S3’, ‘Adam’))\}, \text{COUNT(DISTINCT staffid)} returns 3 whereas \text{COUNT(DISTINCT name)} returns 2.

2. Situation when the IRC field is in an aggregate function other than \text{COUNT()} (i.e. \text{MAX()} and \text{MIN()}). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an IRC mutation operator replaces a non-numeric field \text{f1} in the aggregate function \text{AGGR()} by another non-numeric field \text{f2}, the IRC mutant will have \text{AGGR(f2)} instead of \text{AGGR(f1)}. In order to distinguish between \text{AGGR(f1)} and \text{AGGR(f2)}, the guideline consists of the following two steps:

(a) Set any non-null values for \text{f1} in the records of the database and
(b) Set the values in \text{f2} so that it is equal to corresponding values in \text{f1} plus a fixed constant string \text{S} for each record in the database table. In other word, \text{f2} = \text{f1+S}.

It does not matter whether there is a ‘DISTINCT’ keyword in the SQL statement, the same guideline will work.

For example, one possible mutant of O2-2 in Table 4.1 (\text{“SELECT MAX(staffid), MIN(name) FROM STAFF”}) is \text{“SELECT MAX(name), MIN(name) FROM STAFF”}, denoted as M4-1. By setting the values for “staffid” and “name” in such a way so that \text{name} = \text{staffid + ‘xy’}^4, such as \{(“staffid”=‘S2’, “name”=‘S2xy’), (‘S5’, ‘S5xy’), (‘S10’, ‘S10xy’)\}, one can distinguish the mutant from its original SQL statement because O2-2 returns “‘S10’, ‘S2xy’” whereas M4-1 returns “‘S10xy’, ‘S2xy’”.

4.3.3 Guidelines for Mutants From SQL without Aggregate Function but with WHERE clause

This section discusses the test case generation guidelines for those IRC mutants created from original SQL statements that do not have any aggregate functions in the SQL statement but have a simple WHERE clause. An IRC mutation operator may replace a field

^4constant may be a single or multiple (a group of) characters from 0-9, or a-z, or A-Z
in the SELECT clause or in the WHERE clause to generate the mutants. Depending on the position of IRC mutation operator occurred, there are two major categories of mutants. In the following the test case generation guidelines are described for these two categories.

4.3.3.1 IRC in the SELECT clause

When an IRC mutation operator replaces a field in the SELECT clause by another field that appears in the original SQL statement, six types of mutants, namely IRC-m5 to IRC-m10, are generated depending on whether the numeric or non-numeric field is involved in the SELECT clause in the WHERE clause. In the following, I discuss the test case generation guidelines for these six types of mutants.

G-IRC-m5: For the IRC-m5 mutants, the IRC mutation operator replaces a numeric field in the SELECT clause of the SQL statement by another numeric field that appears in the SQL statement and the IRC-field also appears in the simple condition in the WHERE clause of the SQL statement.

For example, for the SQL statement O3-1a in Table 4.1, if an IRC mutation operator replaces the field “grade” in the SELECT clause by “salary”, the resulting mutant is “SELECT salary, salary FROM STAFF WHERE grade > 40”. I now propose the test case generation guideline that could distinguish this type of mutants from the original SQL statement, using the above example to illustrate the idea. The guideline consists of the following three major steps:

1. Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the IRC-field involved in the WHERE clause.

2. Apply the reverse-BVA rule to set the values of the other numeric fields for those records mentioned in 1 above in such a way that

---

5the field being replaced by the IRC mutation operator
6These values are the boundary value (the one on the boundary), “one-below” the boundary value, and “one-above” the boundary value.
7These BVA values can also be used for other five relational operators like ≤, >, =, < and ≠.
Chapter 4. Test Case Generation for IRC Operator

Table 4.4: Dataset for IRC-m5 Mutants in STAFF

<table>
<thead>
<tr>
<th>grade</th>
<th>salary</th>
<th>Guidelines at a glance</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>82</td>
<td>“grade” &lt; boundary value, other fields &gt; boundary value</td>
</tr>
<tr>
<td>40</td>
<td>85</td>
<td>“grade” = boundary value, other fields ≠ boundary value</td>
</tr>
<tr>
<td>41</td>
<td>28</td>
<td>“grade” &gt; boundary value, other fields &lt; boundary value</td>
</tr>
<tr>
<td>39</td>
<td>40</td>
<td>“grade” ≠ boundary value, other fields = boundary value</td>
</tr>
<tr>
<td>41</td>
<td>40</td>
<td>“grade” ≠ boundary value, other fields = boundary value</td>
</tr>
</tbody>
</table>

(a) they are greater than the boundary value when the value of the IRC-field is less than the boundary value,

(b) they are less than the boundary value when the value of the the IRC-field is greater than boundary value, and

(c) they are not equal to the boundary value when the value of the IRC-field is equal to the boundary value.

3. Add extra records in the database table so that the values of the other numeric fields are equal to the boundary value and the values of the IRC-field are not equal to the boundary value. In this case, I need at least two records, one for the IRC-field whose value is less than the boundary value whereas the other is greater.

For example, to distinguish the mutant from O3-1a, I need to apply BVA on the condition “grade > 40” in the WHERE clause so as to set the records in the database table such that, each record will have the values 39, 40, and 41 to the “grade” field in various records in the database table. For these three records, I then apply the reverse-BVA rule on “salary” to set the records like (“grade”=39, “salary” = 82), (40, 85), and (41, 28). Finally, I then add two further records (39, 40) and (41, 40) to the database. Table 4.4 illustrates the use of the proposed guidelines in this example. One can distinguish O3-1a from the mutant because O3-1a returns “(41, 28), (41, 40)” whereas the mutant returns “(28, 40), (28, 40)”.

G-IRC-m6: Similar to IRC-m5, for the IRC-m6 mutants, the IRC mutation operator replaces a numeric field in the SELECT clause of the SQL statement by another numeric field that appears in the SQL statement but the field involved in the WHERE clause does not appear in the SELECT clause. For example, for the SQL statement O3-1b in Table 4.1, when
an IRC mutation operator replaces the field “salary” in the SELECT clause by “grade” the resulting mutant is “SELECT grade, grade FROM STAFF WHERE age > 30”. I now propose the test case generation guideline that could distinguish this type of mutants from the original SQL statement, illustrating the idea with the help of the above example. The guideline consists of the following three major steps:

1. Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values\(^8\), and then set the records in the database table so that each record will have one of these values in the field involved in the WHERE clause\(^9\)

2. Apply the reverse-BVA\(^10\) rule to set the values of the other numeric fields for those records mentioned in 1 above in such a way that

   (a) they are greater than the boundary value when the value of the field in the WHERE clause is less than the boundary value,

   (b) they are less than the boundary value when the value of the field in the WHERE clause is greater than the boundary value,

   (c) they are not equal to the boundary value when the value of the field in the WHERE clause is equal to the boundary value.

3. Add extra records in the database table so that the values of the other numeric fields are equal to the boundary value and the values of the field in the WHERE clause are not equal to the boundary value. In this case, I need at least two records, one for the field in the WHERE clause whose value is less than the boundary value whereas the other is greater.

For example, to distinguish the mutant from O3-1b, I need to apply BVA on the condition “age > 30” in the WHERE clause so as to set the records in the database table such that, each record will have 29, 30, and 31 in the field “age”. For these three records, I then apply

---

\(^8\)These values are the boundary value (the one on the boundary), “one-below” the boundary value, and “one-above” the boundary value.

\(^9\)This is similar to Step 1 of G-IRC-m5 except that it sets the boundary values to the field involved in the WHERE clause instead of the IRC-field.

\(^10\)This is the same as Step 2 of G-IRC-m5.
the reverse-BVA rule on “grade” and “salary” in the SELECT clause to set the records like (“age” = 29, “grade” = 32, “salary” = 42), (30, 25, 35), and (31, 28, 18). Finally, I then add two further records (29, 30, 30) and (31, 30, 30) to the database. One can distinguish O3-1b from the mutant because O3-1b returns “(28, 18), (30, 30)” whereas the mutant returns “(28, 28), (30, 30)”.

G-IRC-m7: For the IRC-m7 mutant, the IRC mutation operator replaces a numeric field by another numeric field in the SELECT clause of the SQL statement and a non-numeric field is involved in the WHERE clause\(^\text{11}\). In order to distinguish the results of the original SQL statement from the mutants, the guideline is to

1. Apply BVA on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the non-numeric field in the WHERE clause\(^\text{12}\).

2. Set the values in the numeric fields in the SELECT clause for those records mentioned in 1 above so that these values are all distinct.

For example, for the SQL statement O3-1c in Table 4.1, when an IRC mutation operator replaces the numeric field “salary” in the SELECT clause of the SQL statement by another numeric field “grade”, the resultant mutant is “\text{SELECT grade, grade FROM STAFF WHERE staffid = `S1` }”. In order to distinguish the mutant from O3-1c, I need to apply BVA on the condition “staffid = `S1` ” in the WHERE clause so as to set three records in the database table such that, each record will have ‘S0’, ‘S1’, and ‘S2’ in the field “staffid”. For these three records, I then set the values for “grade” and “salary” so that they are different. An example may be (“staffid” = ‘S0’, “grade” = 5, “salary” = 1000), (‘S1’, 10, 2000), and (‘S2’, 15, 3000). One can distinguish O3-1c from the mutant because O3-1c returns “10, 2000” whereas the mutant returns “10, 10”.

\(^{11}\)Please refer to the relevant example in Table 4.3
\(^{12}\)This is similar to Step 1 of G-IRC-m5 except that the field in the condition is non-numeric. For non-numeric field, we can use its natural ordering to select the values that are “1 below the boundary” and “1 above the boundary.”
G-IRC-m8: For the IRC-m8 mutants, the IRC mutation operator replaces a non-numeric field in the \texttt{SELECT} clause of the SQL statement by another non-numeric field\textsuperscript{13} that appears in the SQL statement and a numeric field is involved in the simple condition in the \texttt{WHERE} clause. In order to distinguish the results of the original SQL statement from those mutants, the test case generation guideline is to

1. Apply BVA on the condition in the \texttt{WHERE} clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the numeric field in the \texttt{WHERE} clause\textsuperscript{14}

2. Set the values in the non-numeric fields of the \texttt{SELECT} clause for those records mentioned in 1 above so that these values are all distinct

For example, for the SQL statement O3-2a in Table 4.1, when an IRC mutation operator replaces the field “staffid” in the \texttt{SELECT} clause by “name”, the resultant mutant is \texttt{SELECT name, name FROM STAFF WHERE grade > 40". In order to distinguish the mutant from O3-2a, I first need to apply BVA on the condition “grade > 40” in the \texttt{WHERE} clause to set the values 39, 40, and 41 to the “grade” field in various records in the database table. For these three records, I then set the values for “staffid” and “name” so that they are distinct. An example may be (“grade” = 39, “staffid” =‘S1’, “name” = ‘Jack’), (40, ‘S2’, ‘David’), (41, ‘S3’, ‘James’). One can distinguish O3-2a from the mutant because O3-2a returns “S3, James”, whereas the mutant returns “James, James”.

G-IRC-m9: For the IRC-m9 mutants, the IRC mutation operator replaces a non-numeric field in the \texttt{SELECT} clause of the SQL statement by another non-numeric field that appears in the SQL statement and the IRC-field\textsuperscript{15} also appears in the simple condition in the \texttt{WHERE} clause. In order to distinguish the results of the original SQL statement from those mutants, the guideline is to

\textsuperscript{13}Please refer to Table 4.1 and Table 4.3
\textsuperscript{14}This is same as Step 1 of G-IRC-m5.
\textsuperscript{15}Please refer to Table 4.3
Chapter 4. Test Case Generation for IRC Operator

1. Apply BVA on the condition in the **WHERE** clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the non-numeric field in the **WHERE** clause\(^\text{16}\).

2. Set the values in the other non-numeric fields in the **SELECT** clause for those records mentioned in 1 above so that these values are all distinct and different from the corresponding BVA values.

For example, for the SQL statement O3-2b in Table 4.1, when an *IRC* mutation operator replaces a non-numeric field “staffid” in the **SELECT** clause of the SQL statement by another non-numeric field “name” that appears in the SQL statement, the resultant mutant is **SELECT name, name FROM STAFF WHERE staffid = ‘S1’**. In order to distinguish the mutants from O3-2b, I need to apply BVA on the condition “staffid = ‘S1’ ” in the **WHERE** clause so as to set three records in the database table such that, each record will have ‘S0’, ‘S1’, and ‘S2’ in the field “staffid”. For these three records, I then set the values for “name” so that they are different. An example may be (‘staffid’ = ‘S0’, “name” = ‘Jack’), (‘S1’, ‘David’), (‘S2’, ‘Micheal’). One can distinguish O3-2b from the mutant because O3-2b returns “S1, David” whereas the mutant returns “David, David”.

**G-IRC-m10:** Similar to IRC-m9, for the IRC-m10 mutants, the *IRC* mutation operator replaces a non-numeric field in the **SELECT** clause of the SQL statement by another non-numeric field but the *IRC*-field\(^\text{17}\) does not appear in the **WHERE** clause. In order to distinguish the results of the original SQL statement from those mutants, the guideline is to

1. Apply BVA on the condition in the **WHERE** clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the non-numeric field in the **WHERE** clause\(^\text{18}\).

\(^{16}\)This is similar to Step 1 of G-IRC-m7

\(^{17}\)Please refer to Table 4.3

\(^{18}\)This is similar to Step 1 of G-IRC-m7
2. Set the values in the other non-numeric fields in the `SELECT` clause for those records mentioned in 1 above so that these values are all distinct and different from the corresponding BVA values.

For example, for the original SQL statement O3-2c in Table 4.1, when an IRC mutation operator replaces a non-numeric field “staffid” by another non-numeric field “name, the resultant mutant is `SELECT name, name FROM STAFF WHERE city = 'C8'` . To distinguish the mutant from O3-2c, I need to apply BVA on the condition “city = ‘C8’ ” in the `WHERE` clause so as to set three records in the database table such that, each record will have ‘C7’, ‘C8’, and ‘C9’ for the field “city” in the `WHERE` clause. For these three records, I then set the values for “staffid” and “name” so that they are distinct. An example may be (‘city’=‘C7’, “staffid” =‘S1’, “name” = ‘Jack’), (‘C8’, ‘S2’, ‘David’), (‘C9’, ‘S3’, ‘Aron’). One can distinguish O3-2c from the mutant because O3-2c returns “S2, David ” whereas the mutant returns “David, David”.

4.3.3.2 IRC in WHERE clause

When an IRC mutation operator occurs in the `WHERE` clause, four types of mutants are generated namely, IRC-m11 to IRC-m14 depending on the numeric or non-numeric field involved in the `SELECT` and the `WHERE` clause. The test generation guidelines for these mutants are described as follows:

**G-IRC-m11**: For the IRC-m11 mutants, the IRC mutation operator replaces a numeric field in the simple `WHERE` clause of the SQL statement by another numeric field that appears in the SQL statement and the IRC-field also appears in the `SELECT` clause. I now propose the test case generation guideline that could distinguish this type of mutants from the original SQL. The guideline is similar to that of G-IRC-m5, it consists of the following three major steps:

---

19 Please refer to Table 4.3
1. Apply BVA on the condition in the `WHERE` clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the `IRC`-field involved in the `WHERE` clause\(^\text{20}\)

2. Apply the `reverse-BVA` rule to set the values of the other numeric fields for those records mentioned in 1 above in such a way that

   (a) they are greater than the boundary value when the value of the `IRC`-field is less than the boundary value,

   (b) they are less than the boundary value when the value of the `IRC`-field is greater than the boundary value, and

   (c) they are not equal to the boundary value when the value of the `IRC`-field is equal to the boundary value.

3. Add extra records in the database table so that the values of the other numeric fields are equal to the boundary value and the values of the `IRC`-field are not equal to the boundary value. In this case, I need at least two records, one for the `IRC`-field whose value is less than the boundary value whereas the other is greater.

For example, for the SQL statement O3-1a in Table 4.1, when an `IRC` mutation operator replaces the field “grade” in the `WHERE` clause by “salary” the resultant mutant is “\textbf{SELECT grade, salary FROM STAFF WHERE salary > 40}”. I need to apply BVA on the condition “\textbf{grade > 40}” so as to set the records in the database table such that, each record will have the values 39, 40, and 41 in the “grade” field in various records in the database table.

For these three records, I then apply the reverse-BVA rule on “salary” to set the records like (“grade”=39, “salary” = 82), (40, 85), and (41, 28). Finally, I then add two further records (39, 40) and (41, 40) to the database. One can distinguish O3-1a from the mutant (whose `WHERE` clause is “salary > 40”) because O3-1a returns “(41, 28), (41, 40)” whereas the mutant returns “(39, 82), (40, 85)”.

\textbf{G-IRC-m12}: Similar to IRC-m11, for the IRC-m12 mutants the `IRC` mutation operator replaces a numeric field in the simple `WHERE` clause of the SQL statement by another

\[^{20}\text{These BVA values can also be used for other five relational operators like } \leq, >, =, < \text{ and } \neq.\]
numeric field that also appears in the SQL statement but the IRC-field does not appear in the SELECT clause\textsuperscript{21}. I now propose the test case generation guideline that could distinguish this type of mutants from the original SQL statement. The guideline is similar to that of G-IRC-m6, and consists of the following three major steps:

1. Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the IRC-field involved in the WHERE clause\textsuperscript{22}.

2. Apply the reverse-BVA\textsuperscript{23} rule to set the values of the other numeric fields for those records mentioned in 1 above in such a way that
   
   (a) they are greater than the boundary value when the value of the field in the WHERE clause is less than the boundary value,
   
   (b) they are less than the boundary value when the value of the field in the WHERE clause is greater than the boundary value,
   
   (c) they are not equal to the boundary value when the value of the field in the WHERE clause is equal to the boundary value.

3. Add extra records in the database table so that the values of the other numeric fields are equal to the boundary value and the values of the field in the WHERE clause are not equal to the boundary value. In this case, I need at least two records, one for the field in the WHERE clause whose value is less than the boundary value whereas the other is greater.

For example, for the SQL statement O3-1b in Table 4.1, when an IRC mutation operator replaces the field “age” in the WHERE clause by “salary”, the resultant mutant is “SELECT grade, salary FROM STAFF WHERE salary > 30”. In order to distinguish the mutant from O3-1b, I need to apply the BVA on the condition “age> 30” in the WHERE clause so as to set the records in the database table such that, each record will have 29, 30, and 31 in the

\textsuperscript{21}Please refer to Table 4.3

\textsuperscript{22}These BVA values can also be used for other five relational operators like \(\leq, >, =, <\) and \(\neq\).

\textsuperscript{23}This is the same as Step 2 of G-IRC-m5
“age” field. For these three records, I then apply the reverse-BVA rule on “grade” and “salary” in the SELECT clause to set the records like (“age” = 29, “grade” = 32, “salary” = 42), (30, 25, 35), and (31, 28, 18). Finally, I then add two further records (29, 30, 30) and (31, 30, 30) to the database. One can distinguish O3-1b from the mutant because O3-1b returns “(28, 18), (30, 30)” whereas the mutant returns “32, 42”.

**G-IRC-m13:** For the IRC-m13 mutants, the IRC mutation operator replaces a non-numeric field in the simple WHERE clause of the SQL statement by another non-numeric field that also appears in the SQL statement and the IRC-field also exists in the SELECT clause. In order to distinguish the results of the original SQL from the mutant, the guideline is similar to that of G-IRC-m9, and consists of the following two major steps:

1. Apply BVA on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the non-numeric field in the WHERE clause.

2. Set the values in the other non-numeric field in the SELECT clause for those records mentioned in 1 above so that these values are all distinct and different from the corresponding BVA values.

For example, for the SQL statement O3-2b in Table 4.1, when an IRC mutation operator replaces a non-numeric field “staffid” in the WHERE clause of the SQL statement by another non-numeric field “name” that appears in the SQL statement, the resultant mutant is “SELECT staffid, name FROM STAFF WHERE name = ‘S1’ “. In order to distinguish the mutant from O3-2b, I need to apply BVA on the condition “staffid = ‘S1’ ” in the WHERE clause so as to set three records in the database table such that, each record will have ‘S0’, ‘S1’, and ‘S2’ in the field “staffid”. For these three records, I then set the values for “name” so that they are different. An example may be (“staffid” = ‘S0’, “name” = ‘Jack’), (‘S1’, ‘David’), (‘S2’, ‘Micheal’). One can distinguish O3-2b from the mutant because O3-2b returns “S1, David” whereas the mutant returns empty.

\[24\] This is similar to Step 1 of G-IRC-m9.
**G-IRC-m14**: Similar to IRC-m13, for the IRC-m14 mutants the *IRC* mutant replaces a non-numeric field in the *WHERE* clause by another non-numeric field that also appears in the SQL statement and the *IRC*-field does not appear in the *SELECT* clause. In order to distinguish the results of the original SQL statements from those mutants, the guideline is similar to G-IRC-m10, it consists of the following two major steps:

1. **Apply BVA on the condition in the *WHERE* clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the non-numeric field in the *WHERE* clause**.\(^{25}\)

2. **Set the values in the other non-numeric fields in the *SELECT* clause for those records mentioned in 1 above so that these values are all distinct and different from the corresponding BVA values.**

For example, for the original SQL statement O3-2c in Table 4.1, when an *IRC* mutation operator replaces a non-numeric field “city” in the *WHERE* clause by another non-numeric field “name”, the resultant mutant is “*SELECT* staffid, name FROM STAFF *WHERE* name = ‘C8’”. To distinguish the mutant from O3-2c, I need to apply BVA on the condition “city = ‘C8’” in the *WHERE* clause so as to set three records in the database table such that, each record will have ‘C7’, ‘C8’, and ‘C9’ for the field “city” in the *WHERE* clause. For these three records, I then set the values for “staffid” and “name” so that they are distinct. An example may be (‘city’=’C7’, “staffid” =’S1’, name = ‘Jack’), (‘C8’, ‘S2’, ‘David’), and (‘C9’, ‘S3’, ‘Aron’). One can distinguish O3-2c from the mutant because O3-2c returns “S2, David” whereas the mutant returns empty.

### 4.3.4 Guideline for Mutants from SQL with both Aggregate Function and *WHERE* Clause

This section discusses test case generation guidelines for *IRC* mutants created from those original SQL statements having aggregate functions (such as *MAX()*, *MIN()*, *SUM()*, *AVG()*, and *COUNT()* in the *SELECT* clause and a simple *WHERE* clause. Similar to the

\(^{25}\)This is similar to Step 1 of G-IRC-m10
previous sections, an IRC mutation operator may occur in an aggregate function of the SELECT clause or in the WHERE clause. The following two sections discuss these two situations.

4.3.4.1 IRC in Aggregate Functions of the SELECT Clause

As discussed earlier, when an IRC mutation operator occurs in an aggregate function of the SELECT clause, six types of mutant, namely IRC-m15 to IRC-m20, are possible depending on whether a numeric or a non-numeric field is involved in the SELECT clause or in the WHERE clause. In the following I discuss the test case generation guidelines for these six types of mutant represented as G-IRC-m15 to G-IRC-m20. Note that, each of these mutants is involved in aggregate functions (COUNT(), MAX(), MIN(), SUM(), and AVG()). For ease of discussion, depending on the characteristics of aggregate functions, there are two different sets of guidelines. In the first set, the guidelines are related to the detecting of mutants involving the aggregate function COUNT() and in the second set, the guidelines are about other aggregate functions functions (MAX(), MIN(), SUM(), and AVG()). The test case generation guidelines are discussed as follows:

G-IRC-m15: For the IRC-m15 mutants, the IRC mutation operator replaces a numeric field in the aggregate function of the SELECT clause by another numeric field that also appears in the SQL statement and the IRC-field also appears in the WHERE clause of the SQL statement. Because of the involvement of numeric fields, any aggregate functions can be applied on these numeric fields. Similar to G-IRC-m3, two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function COUNT() and in the second set, the guidelines are aimed for other aggregate functions.

1. (a) Situation when the IRC-field is in the aggregate function COUNT(). Since COUNT() returns the number of non-null values in the required field, when the fields involved in the IRC mutation operator do not allow null values, there is no way to distinguish the original statement from its mutant. In other words,
the mutants are equivalent to the original SQL statement. However, when the fields involved allow null values, it is possible to distinguish the original SQL statement from the mutants. To do so, the guideline consists of the following two steps:

i. Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values\(^\text{27}\), and then set the records in the database table so that they will have one of these values in the IRC-field involved in the WHERE clause\(^\text{28}\).

ii. Set the values in the other numeric fields of the SELECT clause for those records mentioned in (i) above so that the IRC-field and these numeric fields have different number of null values. By doing so, I need to set multiple records for each of the BVA values found in (i) above.

For example, for the SQL statement \(\text{SELECT COUNT(grade), MAX(salary) FROM STAFF WHERE grade > 40}\)\), when an IRC mutation operator replaces “grade” in the aggregate function of the SELECT clause by “salary”, the resultant mutant is \(\text{SELECT COUNT(salary), MAX(salary) FROM STAFF WHERE grade > 40}\)\). In order to distinguish this type of mutants from the original SQL statement, I need to apply BVA on the condition “grade > 40” in the WHERE clause so as to set records in the database table such that they will have 39, 40, and 41 in the IRC-field (that is, “grade”). For these records, I then set some null values for the “salary” field that appears in the SELECT clause in such a way that “grade” and “salary” have different number of null values. A possible example may be (“grade” = 39, “salary” = 1500), (39, null), (40, 1600), (40, null), (41, 1700), and (41, null). One can distinguish \text{COUNT(salary)}\) from \text{COUNT(grade)}\) for the condition “grade > 40” in WHERE clause because \text{COUNT(grade)}\) returns 2 whereas \text{COUNT(salary)}\) returns 1.

(b) In the case of \text{COUNT(DISTINCT \ldots)}\), the test case generation guidelines are different from the case of \text{COUNT(\ldots)}\). Please note that when the fields involved in the IRC mutation operator allow null values, it is possible to distinguish

\(^{27}\)These values are the boundary value (the one on the boundary), “one-below” the boundary value, and “one-above” the boundary value.

\(^{28}\)These BVA values can also be used for other five relational operators like ≤, >, =, < and ≠.
Table 4.5: Result showing the SQL Statements with COUNT(DISTINCT ...)

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>Condition</th>
<th>Relational operator (op)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original: COUNT(DISTINCT grade)</td>
<td>grade op 40</td>
<td>&gt; 2 1 2 1 2 1 2 1 2</td>
</tr>
<tr>
<td>Mutant: COUNT(DISTINCT salary)</td>
<td>grade op 40</td>
<td>2 4 2 4 2 4</td>
</tr>
</tbody>
</table>

For example, one possible mutant of “SELECT COUNT(DISTINCT grade), MAX(salary) FROM STAFF WHERE grade > 40” is “SELECT COUNT(DISTINCT salary), MAX(salary) FROM STAFF WHERE grade > 40”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “grade>40” in the WHERE clause so as to set records in the database table such that the IRC-field (that is, “grade”) will have 39, 40, and 41. I then set some distinct values for the “salary” in the SELECT clause against the same value of the “grade” field to have the records like (“grade” = 39, “salary” = 1500), (39, 2500), (40, 1600), (40, 2600), (41, 1700), and (41, 2700). By doing so, one can then distinguish the result of COUNT(DISTINCT grade) from COUNT(DISTINCT salary) as shown in Table 4.5.
2. Situation when the *IRC*-field is involved in an aggregate function other than \texttt{COUNT()} (i.e. \texttt{MAX()}, \texttt{MIN()}, \texttt{SUM()}, \texttt{AVG()}). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an *IRC* mutation operator replaces a numeric field \(f_1\) in the aggregate function \texttt{AGGR()} (may be \texttt{MAX()}, \texttt{MIN()}, \ldots) by another numeric field \(f_2\), the *IRC* mutant will have \texttt{AGGR}(f_2) instead of \texttt{AGGR}(f_1). Moreover, \(f_1\) is also involved in \texttt{WHERE} clause. In order to distinguish between \texttt{AGGR}(f_1) and \texttt{AGGR}(f_2), the guideline consists of the following two major steps:

(i) Apply BVA on \(f_1\) to obtain the boundary values, and set the records in the database table so that each record will have one of these values in the numeric field in the \texttt{WHERE} clause.

(ii) Set the values of \(f_2\) so that it is equal to the corresponding values in \(f_1\) plus a fixed non-zero constant \(C\) for each record in the database table. In other words, \(f_2 = f_1 + C\). It does not matter whether there is a \texttt{`DISTINCT'} keyword in the aggregate function, the same guideline is adequate to distinguish the mutants from the original SQL statement.

For example, for the SQL statement O4-1a in Table 4.1, when an *IRC* mutation operator replaces the numeric field “grade” in the aggregate function of the \texttt{SELECT} clause by another numeric field “salary”, the resultant mutant is \texttt{SELECT MAX(salary), MIN(salary) FROM STAFF WHERE grade > 40”}. In order to distinguish this type of mutant from O4-1a, I need to apply BVA on the condition “grade > 40” in the \texttt{WHERE} clause so as to set three records in the database table such that each record will have 39, 40 and 41 in the field “grade”. For these three records, I then set the values of “salary” in such a way that “salary” = “grade” + 1000. An example may be (“grade” = 39, “salary” = 1039), (40, 1040), and (41, 1041). By doing so, one can then distinguish O4-1a from the mutant because O4-1a returns “41, 1041” whereas the mutant returns “1041, 1041”.

**G-IRC-m16:** Similar to IRC-m15, for the IRC-m16 mutants, the *IRC* mutation operator replaces a numeric field in the aggregate function of the \texttt{SELECT} clause by another numeric
field that also appears in the SQL statement but the IRC-field does not appear in the **WHERE** clause. Two different sets of guidelines are proposed, depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function **COUNT()** and in the second set, the guidelines are about other aggregate functions.

1. (a) Situation when the IRC-field is in the aggregate function **COUNT()**. Similar to G-IRC-m15, since **COUNT()** returns the number of non-null values in the required field, when the fields involved in the IRC mutation operator do not allow null values, the mutants are equivalent to the original SQL statement. However, when the fields involved allow null values, it is possible to distinguish the results of the mutants from the original SQL statement. The guideline consists of the following two major steps:

   i. Apply boundary value analysis (BVA) on the condition in the **WHERE** clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the numeric field involved in the **WHERE** clause and

   ii. Set the values in the numeric fields of the **SELECT** clause for those records mentioned in (i) above so that these numeric fields have different number of null values. By doing so, I need to set multiple records for each of the BVA values found in (i) above.

   For example, for the SQL statement, 
   ```sql
   SELECT MAX(grade), COUNT(salary) FROM STAFF WHERE age > 30
   ```
   when an IRC mutation operator replaces “salary” in the aggregate function of the SQL statement by “grade”, the resultant mutant is 
   ```sql
   SELECT MAX(grade), COUNT(grade) FROM STAFF WHERE age > 30
   ```
   In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “age > 30” in the **WHERE** clause so as to set records in the database table so that they will have 29, 30 and 31 in the field “age”. For these records, I then set different number of null values for the “grade”

\[29\]Note that, each field in the **SELECT** clause must have at least one null value to distinguish all related mutants.
and the “salary” fields that appear in the SELECT clause in such a way that “grade” and “salary” have different number of null values. An example may be 

(age=29, grade=5, salary=500), (29, null, 600), (29, null, null) (30, 10, 1000), 
(30, 12, null), (30, null, null), (31, 15, 1500), (31, null, 1600) and (31, null, null). By doing so, one can then distinguish COUNT(salary) from COUNT(grade) for the condition “age>30” in the WHERE clause because COUNT(salary) returns 2 whereas COUNT(grade) returns 1.

In the case of COUNT(DISTINCT ...), the test case generation guidelines are different from the case of COUNT(). Please note that when the fields involved in the IRC mutation operator allow null values, it is possible to distinguish the mutants from the original SQL statement by setting the values in the related fields such that they will have different number of null values. Hence, the guidelines of setting different null values for COUNT() is also applicable for COUNT(DISTINCT ...) when the fields allow null values. On the other hand, unlike the situation in COUNT(), if the fields cannot be null, one can still distinguish the mutants with COUNT(DISTINCT ...) from its original SQL by the following steps:

i. Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the numeric field involved in the WHERE clause and

ii. Set the values in the other numeric fields in the SELECT clause for those records mentioned in (i) above so that these fields have different number of distinct values. To achieve this, I need to have multiple distinct values for the other numeric fields for the same boundary value in the field of the WHERE clause.

For example, for the SQL statement, “SELECT MAX(grade), COUNT(DISTINCT salary) FROM STAFF WHERE age > 30”, when an IRC mutation operator replaces “salary” in the aggregate function of the SQL statement by another numeric field “grade”, the resultant mutant is “SELECT MAX(grade), COUNT(DISTINCT grade) FROM STAFF WHERE age > 30”. In order to distinguish the mutants
from the original SQL statement, I need to apply BVA on the condition “age > 30” in the \texttt{WHERE} clause so as to set records in the database table such that the “age” field will have the values 29, 30 and 31. I then set different distinct values for the “grade” and the “salary” fields in these records so as to have records like ("age"=29, “grade”=6, “salary”=500), (29, 7, 600), (29, 7, 600) (30, 10, 1000), (30, 25, 2000), (30, 35, 2000), (31, 20, 2500), (31, 25, 3000), and (31, 25, 3500). By doing so, one can then distinguish the mutants from the original SQL statement because the original SQL statement returns “25, 3”, whereas the mutant returns ‘25, 2’.

2. Situation when the IRC-field is involved in an aggregate function other than \texttt{COUNT()} (i.e. \texttt{MAX()}, \texttt{MIN()}, \texttt{SUM()}, \texttt{AVG()}). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an IRC mutation operator replaces a numeric field \( f_1 \) in the aggregate function \texttt{AGGR()} (may be \texttt{MAX()}, \texttt{MIN()}, \ldots) by another numeric field \( f_2 \), the IRC mutant will have \texttt{AGGR}(f_2) instead of \texttt{AGGR}(f_1). Moreover, a different field, \( f_3 \) is involved in the \texttt{WHERE} clause. In order to distinguish between \texttt{AGGR}(f_1) and \texttt{AGGR}(f_2), the guideline consists of the following steps:

(a) Apply BVA on the condition involving \( f_3 \) in the \texttt{WHERE} clause to obtain the boundary values, and set the records in the database table so that each record will have one of these values in the field \( f_3 \)

(b) Set the values of \( f_1 \) and \( f_2 \) so that \( f_1 \equiv f_3+C_1 \) and \( f_2 \equiv f_3+C_2 \) where \( C_1 \) and \( C_2 \) are two fixed non-zero constants for each record in the database table.

It does not matter whether there is a ‘\texttt{DISTINCT}’ keyword in the aggregate function, the same guideline is adequate to distinguish the mutants from the original SQL statement.

For example, for the SQL statement O4-1b in Table 4.1, when an IRC mutation operator replaces the numeric field “salary” in the aggregate function \texttt{(MIN())} of the \texttt{SELECT} clause by another numeric field “grade”, the resultant mutant is \texttt{SELECT MAX(grade), MIN(grade) FROM STAFF WHERE age > 30"}. In order to distinguish this type of mutant from O4-1b, I need to apply BVA on the condition “age> 30” in the
WHERE clause so as to set three records in the database table such that, each record will have 29, 30 and 31 in the field “age”. For these three records, I then set the values for the “grade” and the “salary” fields so that “grade” = “age” + 1000 and “salary” = “age” + 2000. An example may be (“age” = 29, “grade” = 1029, “salary” = 2029), (30, 1030, 2030), and (31, 1031, 2031). By doing so, one can then distinguish O4-1b from the mutant because O4-1b returns “1031, 2031” whereas the mutant returns “1031, 1031”. In cases, where the SQL statements involve ‘DISTINCT’ keyword the above dataset is adequate to distinguish all related mutants from the original SQL statement.

G-IRC-m17: For the IRC-m17 mutant, the IRC mutation operator replaces a numeric field in the aggregate function of the SELECT clause by another numeric field that also appears in the SQL statement and a non-numeric field appears in the WHERE clause. Two different sets of guidelines are proposed, depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function COUNT() and in the second set, the guidelines are about other aggregate functions.

1. (a) Situation when the IRC-field is in the aggregate function COUNT(). Since COUNT() returns the number of non-null values in the required field, when the fields involved in the IRC mutation operator do not allow null values, the mutants are equivalent to the original SQL statement. However, when the fields involved allow null values, it is possible to distinguish the mutants from the original SQL statement. The guideline consists of the following two major steps:

   i. Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the non-numeric field involved in the WHERE clause.

   \(^{30}\)Please refer to Table 4.3
ii. Set the values for the numeric fields in the aggregate functions of the `SELECT` clause for those records mentioned in (i) above so that these numeric fields have different number of `null` values. By doing so, I need to set multiple records for each of the BVA values found in (i) above.

For example, for the SQL statement `SELECT MAX(grade), COUNT(salary) FROM STAFF WHERE staffid = 'S1'`, when an `IRC` mutation operator replaces a numeric field “salary” in the aggregate function (`COUNT()` ) by another numeric field “grade”, the resultant mutant is `SELECT MAX(grade), COUNT( grade) FROM STAFF WHERE staffid = 'S1'`. To distinguish this type of mutant from the the original SQL statement, I need to apply BVA on the condition “staffid = ‘S1’ ” in the `WHERE` clause so as to set records in the database table such that, they will have ‘S0’, ‘S1’, and ‘S2’ in the field in the `WHERE` clause (that is “staffid”). For these records, I then set the values for the “grade” and the “salary” field that appear in the `SELECT` clause in such a way that “grade” and “salary” have different number of null values. A possible example may be (‘staffid’ = ‘S0’, "grade" = 10, “salary” = 100), (‘S0’, 15, null), (‘S1’, 5, 500), (‘S1’, null, 1000), (‘S2’, 20, 1500), and (‘S2’, null, 2000). By doing so, one can then distinguish the results of the mutant from O4-1c because O4-1c returns “5, 2” whereas the mutant returns “5, 1”.

(b) For the case of `COUNT(DISTINCT ... )`, the test case generation guideline is different from that of `COUNT()`. Please note that when the fields involved in the `IRC` mutation operator allow `null` values, it is possible to distinguish the mutants from the original SQL statement by setting the values in the related fields such that they will have different number of `null` values. Hence, the guidelines of setting different `null` values for `COUNT()` is also applicable for `COUNT(DISTINCT ... )` when the fields allow `null` values. On the other hand, unlike the situation in `COUNT()`, if the fields cannot be `null`, one can still distinguish the mutants with `COUNT(DISTINCT ... )` from its original SQL by the following steps:

i. Apply BVA on the condition in the `WHERE` clause to obtain the boundary values, and then set the records in the database table so that they will have
one of these values in the non-numeric field in the WHERE clause.\(^{31}\)

ii. Set the values in the numeric fields in the SELECT clause for those records mentioned in (i) above so that they will have different distinct values. To achieve this, I need to have multiple distinct values for the other numeric fields for the same boundary value in the non-numeric field.

For example, for the SQL statement "SELECT MAX(grade), COUNT(DISTINCT salary) FROM STAFF WHERE staffid = 'S1'", when an IRC mutation operator replaces "salary" in the aggregate function of the SELECT clause by "grade", the resultant mutant is "SELECT MAX(grade), COUNT(DISTINCT grade) FROM STAFF WHERE staffid = 'S1'". In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition "staffid = 'S1'" in the WHERE clause so as to set records in the database table such that the non-numeric field in the WHERE clause (that is, "staffid") will have 'S0', 'S1', and 'S2'. For these records, I then set values for the "grade" and the "salary" field in the aggregate function of the SELECT clause so that they will have different number of distinct values. An example may be ('staffid' = 'S0', 'grade' = 10, "salary" = 100), ('S0', 10, 200), ('S0', 15, 300), ('S1', 5, 500), ('S1', 6, 200), ('S1', 7, 200), ('S2', 15, 1000), and ('S2', 15, 1500), ('S2', 20, 2000). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns "7, 2" whereas the mutant returns "7, 3".

2. Situation when the IRC field is involved in an aggregate function other than COUNT() (e.g. MAX(), MIN, SUM, AVG()). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an IRC mutation operator replaces a numeric field f1 in the aggregate function AGGR() (may be MAX(), MIN(), ...) by another numeric field f2, the IRC mutant will have AGGR(f2) instead of AGGR(f1). To distinguish between AGGR(f1) and AGGR(f2), the guidelines consist of the following two steps:

\(^{31}\)This is similar to Step 1 of G-IRC-m7
(i) Apply boundary value analysis (BVA) on the condition in the **WHERE** clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the field involved in the non-numeric field in the **WHERE** clause.

(ii) Set the values in the numeric field \( f_1 \) in the **SELECT** clause for those records mentioned in (i) above so that \( f_2 = f_1 + C \) where \( C \) is a fixed non-zero constant.

It does not matter whether there is a `DISTINCT` keyword in the aggregate function, the same guidelines will work.

For example, for the SQL statement O4-1c, in Table 4.1 when an **IRC** mutation operator replaces the numeric field “salary” in the aggregate function of the **SELECT** clause by another numeric field “grade”, the resultant mutant is “**SELECT** \( \text{MAX}(\text{grade}), \text{MIN}(\text{grade}) \) FROM **STAFF** WHERE \text{staffid} = 'S1' ”. In order to distinguish this type of mutant from O4-1c, I need to apply BVA on the condition “staffid = ‘S1’ ” in the **WHERE** clause so as to set records in the database table such that, they will have ‘S0’, ‘S1’, and ‘S2’ in the non-numeric field in the **WHERE** clause (that is, “staffid”). For these records, I then set some non-zero values for the “grade” field. Furthermore, I set the values of “salary” so that “salary” = “grade” + 1000. An example may be (“staffid” = ‘S0’, “grade” = 1, “salary” = 1001), (‘S0’, 5, 1005), (‘S1’, 10, 1010), (‘S1’, 15, 1015), (‘S2’, 20, 1020), and (‘S2’, 25, 1025). By doing so, one can then distinguish the mutant from O4-1c because O4-1c returns “15, 1010” whereas the mutant returns “15, 15”.

**G-IRC-m18**: For the IRC-m18 mutants, the **IRC** mutation operator replaces a non-numeric field in an aggregate function of the **SELECT** clause by another non-numeric field that also appears in the SQL statement and a numeric field is involved in the simple condition in the **WHERE** clause. As the non-numeric fields are involved in the aggregate functions of the **SELECT** clause, **COUNT()**, **MAX()**, and **MIN()** are the only legitimate aggregate functions. Two different sets of guidelines are proposed depending on the characteristics of
aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function COUNT() and in the second set, the guidelines are about other aggregate functions.

1. (a) Situation when the IRC-field is involved in the aggregate function COUNT(). Since COUNT() returns the number of non-null values in the required field, when the fields involved in the IRC mutation operator do not allow null values, there is no way to distinguish the original statement from its mutants. However, when the fields involved allow null values, it is possible to distinguish the mutants from the original SQL statement. The guideline consists of the following two major steps:

i. Apply BVA on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the numeric field in the WHERE clause.  

ii. Set the values in the non-numeric fields in the aggregate function of the SELECT clause for those records mentioned in (i) above so that these non-numeric fields have different number of null values. By doing so, I need to set multiple records for each of the BVA value found in (i) above.

For example, for the SQL statement "SELECT COUNT(staffid), MIN(name) FROM STAFF WHERE grade > 40", when an IRC mutation operator replaces a non-numeric field “staffid” in the aggregate function (COUNT()) of the SELECT clause by another non-numeric field “name” the resultant mutant is "SELECT COUNT(name), MIN(name) FROM STAFF WHERE grade > 40". To distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition "grade > 40" in the WHERE clause so as to set records in the database table such that, they will have 39, 40, and 41 in the numeric field in the WHERE clause (that is "grade"). For these records, I then set the values for the “staffid” and the “name” field that appear in the SELECT clause in such a way that “staffid” and “name” have different number of null values. A possible example may be ("grade" = 39, “staffid” =’S1’ , “name”= ‘Jack’), (39, ‘S2’, null), (40, ‘S3’,  

32This is similar to Step 1 of G-IRC-m8


Chapter 4. Test Case Generation for IRC Operator

‘David’), (40, ‘S4’, null), (41, ‘S5’, ‘James’), and (41, ‘S6’, null). By doing so, one can then distinguish the results of the mutant from the original SQL statement because the original SQL statement returns “2, James” whereas the mutant returns “1, James”.

(b) In the case of \texttt{COUNT(DISTINCT ...)} , the test case generation guidelines are different from the \texttt{COUNT()} . Please note that when the fields involved in the IRC mutation operator allow \texttt{null} values, it is possible to distinguish the mutants from the original SQL statement by setting the values in the related fields such that they will have different number of \texttt{null} values. Hence, the guidelines of setting different \texttt{null} values for \texttt{COUNT()} is also applicable for \texttt{COUNT(DISTINCT ...)} when the fields allow \texttt{null} values. On the other hand, unlike the situation in \texttt{COUNT()}, if the fields cannot be \texttt{null}, one can still distinguish the mutants with \texttt{COUNT(DISTINCT ...)} from its original SQL by the following steps:

i. Apply BVA on the condition in the \texttt{WHERE} clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the numeric field in the \texttt{WHERE} clause\footnote{This is same as Step 1 of G-IRC-m8.}

ii. Set the values in the non-numeric fields of the \texttt{SELECT} clause for those records mentioned in (i) above so that these fields have different number of distinct values. To achieve this, I need to have multiple distinct values for the non-numeric fields for the same boundary value in the numeric field in the \texttt{WHERE} clause.

For example, for the SQL statement \texttt{"SELECT COUNT(DISTINCT staffid), MIN(name) FROM STAFF WHERE grade > 40"}, when an IRC mutation operator replaces a non-numeric field “staffid” in the aggregate function (\texttt{COUNT()}) of the \texttt{SELECT} clause by another non-numeric field “name” the resultant mutant is \texttt{"SELECT COUNT(DISTINCT name), MIN(name) FROM STAFF WHERE grade > 40"}. To distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “grade > 40” in the \texttt{WHERE} clause so as to set records in the database table such that, they will have 39, 40, and 41 in the numeric field of the \texttt{WHERE} clause (that is, “grade”). I then set some
different number of distinct values for the “staffid” and the “name” field in the aggregate function of the **SELECT** clause against the same value of the “grade” field to have the records like (‘grade’ = 39, “staffid” =‘S1’, “name” = ‘Jack’), (39, ‘S2’, Jack), (40, ‘S4’, ‘Gill’), (40, ‘S4’, ‘David’), (41, ‘S5’, ‘James’), and (41, ‘S6’, ‘James’). By doing so, one can then distinguish the mutant from the original because the original returns “2, James” whereas the mutant returns “1, James”.

2. Situation when the **IRC-field** is involved in an aggregate function other than **COUNT()** (i.e. **MAX()** and **MIN()**). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an **IRC** mutation operator replaces a non-numeric field f1 in the aggregate function **AGGR()** by another non-numeric field f2, the **IRC** mutant will have **AGGR(f2)** instead of **AGGR(f1)**. In order to distinguish between **AGGR(f1)** and **AGGR(f2)**, the guidelines consist of the following two steps:

   (a) Apply boundary value analysis (BVA) on the condition in the **WHERE** clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the field involved in the numeric field in the **WHERE** clause.

   (b) Set the values in the non-numeric field f1 in the **SELECT** clause for those records mentioned in (i) above so that it will have some non-null values and set the values of f2 so that f2 = f1+S where S is a fixed non-null constant string.

It does not matter whether there is a ‘**DISTINCT**’ keyword in the SQL statement, the same guideline is adequate to distinguish all related mutants from the original SQL statement.

For example, for the SQL statement O4-2a in Table 4.1, when an **IRC** mutation operator replaces “staffid” in the **SELECT** clause by another non-numeric field “name”, the resultant mutant is “**SELECT MAX(name), MIN(name) FROM STAFF WHERE grade > 40**”. In order to distinguish this type of mutant from O4-2a, I need to apply BVA on the condition “grade>40” in the **WHERE** clause so as to set records in the database table such that, they will have 39, 40, 41 in the “grade” field. For these records, I
then set some non-null values for the “staffid” field. Furthermore, I set the values of “name” in such a way so that \text{“name”} = \text{“staffid” + “xy”}^{34}. An example may be (“grade” = 39, “staffid”=‘S2’, “name”=‘S2xy’), (40, ‘S5’, ‘S5xy’), and (41, ‘S10’, ‘S10xy’). By doing so, one can then distinguish the mutant from O4-2a because O4-2a returns “S10, S10xy” whereas the mutant returns “S10xy, S10xy”.

**G-IRC-m19:** For the IRC-m19 mutants, the *IRC* mutation operator replaces a non-numeric field in the aggregate function of the \texttt{SELECT} clause by another non-numeric field and the \texttt{IRC-field}^{35} also appears in the \texttt{WHERE} clause. As this type of \texttt{IRC} mutant involves non-numeric fields; \texttt{MAX()}, \texttt{MIN()}, and \texttt{COUNT()} are the only legitimate aggregate functions. Again, two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function \texttt{COUNT()} and in the second set, the guidelines are about other aggregate functions.

1. (a) Situation when \texttt{IRC-field} is in the aggregate function \texttt{COUNT()}. Since \texttt{COUNT()} returns the number of non-null values in the required field, when the fields involved in the \texttt{IRC} mutation operator do not allow null values, there is no way to distinguish the original statement from its mutant. In other words, the mutants are equivalent to the original SQL statement. However, when the fields involved allow null values, it is possible to distinguish the original SQL statement from the mutants. The guideline consists of the following two major steps:

   i. Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the \texttt{IRC-field} involved in the \texttt{WHERE} clause\textsuperscript{36}.

---

\textsuperscript{34} Constant string may be a single or multiple characters.
\textsuperscript{35} Please refer to Table 4.3
\textsuperscript{36} These BVA values can also be used for other five relational operators like ≤, >, =, < and ≠
ii. Set the values in the other non-numeric fields of the \texttt{SELECT} clause for those records mentioned in (i) above so that the IRC-field and these non-numeric fields have different number of null values. By doing so, I need to set multiple records for each of the BVA values found in (i) above.

For example, for the SQL statement \texttt{"SELECT COUNT(staffid), MIN(name) FROM STAFF WHERE staffid = ‘S1’"}, when an IRC mutation operator replaces “staffid” in the aggregate function of the \texttt{SELECT} clause by “name”, the resultant mutant is \texttt{"SELECT COUNT(name), MIN(name) FROM STAFF WHERE staffid = ‘S1’"}. In order to distinguish this type of mutant from original SQL statement, I need to apply BVA on the condition “staffid = ‘S1’” in the \texttt{WHERE} clause so as to set records in the database table such that, they will have ‘S0’, ‘S1’, and ‘S2’ in the IRC-field (that is “staffid”). For these records, I then set some \texttt{null} values for the “name” field that appears in the \texttt{SELECT} clause in such a way that “staffid” and “name” have different number of null values. A possible example is (‘staffid’ = ‘S0’, “name” = ‘Jack’), (‘S0’, \texttt{null}), (‘S1’, \texttt{null}), (‘S1’, ‘David’), (‘S2’, \texttt{null}), and (‘S2’, ‘Mark’). By doing so, one can then distinguish \texttt{COUNT(staffid)} from \texttt{COUNT(name)} for the condition “staffid = ‘S1’” in the \texttt{WHERE} clause because \texttt{COUNT(staffid)} returns 2 whereas \texttt{COUNT(name)} returns 1.

(b) In the case of \texttt{COUNT(DISTINCT ...)}, the test case generation guidelines are different from the case of \texttt{COUNT()}. Please note that when the fields involved in the IRC mutation operator allow \texttt{null} values, it is possible to distinguish the mutants from the original SQL statement by setting the values in the related fields such that they will have different number of \texttt{null} values. Hence, the guidelines of setting different \texttt{null} values for \texttt{COUNT()} is also applicable for \texttt{COUNT(DISTINCT ...)} when the fields allow \texttt{null} values. On the other hand, unlike the situation in \texttt{COUNT()}, if the fields cannot be \texttt{null}, one can still distinguish the mutants with \texttt{COUNT(DISTINCT ...)} from its original SQL by the following steps:

i. Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values, and then set the records in the database
table so that they will have one of these values in the IRC-field involved in the WHERE clause and

ii. Set the values in the other non-numeric fields in the SELECT clause for those records mentioned in (i) above so that these fields have different number of distinct values. To achieve this, I need to have multiple distinct values for the non-numeric fields for the same boundary value in the IRC-field in the WHERE clause.

For example, for the SQL statement, “SELECT COUNT(DISTINCT staffid), MIN(name) FROM STAFF WHERE staffid = ‘S1’ ”, when an IRC mutation operator replaces “staffid” in the aggregate function of the SELECT clause by “name”, the resultant mutant is “SELECT COUNT(DISTINCT name), MIN(name) FROM STAFF WHERE staffid = ‘S1’ ”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “staffid = ‘S1’ ” in the WHERE clause so as to set records in the database table such that, they will have ‘S0’, ‘S1’, and ‘S2’ in the IRC-field (that is, “staffid”). For these records, I then set some distinct values for the “name” field in the aggregate function in the SELECT clause. A possible example may be ("staffid" = 'S0', "name"='Alex'), (‘S0’, ‘Luke’), (‘S1’, ‘Jack’), (‘S1’, ‘David’), (‘S2’, ‘Mike’), (‘S2’, ‘Peter’)). By doing so, one can then distinguish the mutant from the original SQL statement because the original returns “1, David” whereas the mutant returns “2, David”.

In the case of due to system constraints “staffid” is primary key, mutants are equivalent to the original SQL statement.

2. Situation when the IRC-field is in an aggregate function other than COUNT() (i.e. MAX(), MIN()). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an IRC mutation operator replaces a non-numeric field f1 in the aggregate function AGGR() (may be MAX(), MIN(), ...) by another numeric field f2, the IRC mutant will have AGGR(f1) instead of AGGR(f2). Moreover, f1 also exists in the WHERE clause. In order to distinguish between AGGR(f1) and AGGR(f2), the guideline is to
(i) Apply BVA on the condition involving $f_1$ in the \texttt{WHERE} clause to obtain the boundary values, and set the records in the database table so that each record will have one of these values in the field $f_1$.

(ii) Set the values of $f_2$ so that it is equal to the corresponding values in $f_1$ plus a fixed non-\texttt{null} string constant $S$ for each record in the database table. In other words, $f_2 \equiv f_1 + S$.

It does not matter whether there is a \texttt{DISTINCT} keyword in the aggregate function, the same guideline is adequate to distinguish the mutants from the original SQL statement.

For example, for the SQL statement O4-2b in Table 4.1 when an IRC mutation operator replaces the non-numeric field “staffid” in the aggregate function of the \texttt{SELECT} clause by another non-numeric field “name”, the resultant mutant is \texttt{SELECT MAX(name), MIN(name) FROM STAFF WHERE staffid = ‘S1’ ”}. In order to distinguish this type of mutant from O4-2b, I need to apply BVA on the condition “staffid = ‘S1’ ” in the \texttt{WHERE} clause so as to set records in the database table such that, they will have ‘S0’, ‘S1’, and ‘S2’ to the IRC-field (that is, “staffid”). For these records, I then set the values of the “name” field in the aggregate function in the \texttt{SELECT} clause in such a way that “name” = “staffid” + ‘xy’ \textsuperscript{37}. An example may be (‘staffid’ =‘S0’, “name” =‘S0xy’), (‘S1’, ‘S1xy’), and (‘S2’, ‘S2xy’). By doing so, one can then distinguish the mutant from O4-2b because O4-2b returns “S1, S1xy” whereas the mutant returns “S1xy, S1xy”.

**G-IRC-m20**: For the IRC-m20 mutants, the IRC mutation operator replaces a non-numeric field in the aggregate function of the \texttt{SELECT} clause by another non-numeric field and the IRC-field\textsuperscript{38} does appear in the \texttt{WHERE} clause. Similar to IRC-m19, this type of IRC mutant also involves non-numeric fields and \texttt{MAX()}, \texttt{MIN()}, and \texttt{COUNT()} are the only legitimate aggregate functions. Again, two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are

\textsuperscript{37}constant may be a single or multiple (a group of) characters from 0-9, or a-z, or A-Z

\textsuperscript{38}Please refer to Table 4.3
related to detecting mutants involving the aggregate function \texttt{COUNT()} and in the second set, the guidelines are about other aggregate functions.

1. (a) Situation when the IRC-field is in the aggregate function \texttt{COUNT()}. Since \texttt{COUNT()} returns the number of non-null values in the required field, when the fields involved in the IRC mutation operator do not allow null values, there is no way to distinguish the original statement from its mutant. In other words, the mutants are equivalent to the original SQL statement. However, when the fields involved allow null values, it is possible to distinguish the original SQL statement from the mutants. The guideline consists of the following two major steps:

   i. Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values\(^\text{39}\), and then set the records in the database table so that they will have one of these values in the non-numeric field involved in the \texttt{WHERE} clause\(^\text{40}\).

   ii. Set the values in the other non-numeric fields of the \texttt{SELECT} clause for those records mentioned in (i) above so that the non-numeric fields in the \texttt{SELECT} clause have different number of null values. By doing so, I need to set multiple records for each of the BVA values found in (i) above.

   For example, for the SQL statement \\
   \texttt{SELECT COUNT(staffid), MIN(name) FROM STAFF WHERE city = `C8' }, when an IRC mutation operator replaces “staffid” in the aggregate function of the \texttt{SELECT} clause by “name”, the resultant mutant is \\
   \texttt{SELECT COUNT(name), MIN(name) FROM STAFF WHERE city = `C8' }. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “city = `C8' ” in the \texttt{WHERE} clause so as to set records in the database table such that, they will have ‘C7’, ‘C8’, and ‘C9’ in the non-numeric field in the \texttt{WHERE} clause (that is, “city”). For these records, I then set some different number of null values for the “staffid” and the “name” fields that appears in the aggregate function of the \texttt{SELECT} clause. A possible

\(^{39}\)Same as step 1 of G-IRC-m9
\(^{40}\)These BVA values can also be used for other five relational operators like \(\leq, >, =, <\) and \(\neq\).
example may be (‘city’ = ‘C7’, “staffnum” = ‘S0’, “name” = ‘Jack’), (‘C7’, ‘S1’, null), (‘C7’, null, null), (‘C8’, ‘S2’, null), (‘C8’, ‘S3’, ‘David’), (‘C8’, null, null), (‘C9’, ‘S4’, null), (‘C9’, ‘S5’, ‘Mark’), and (‘C9’, null, null). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “3, David” whereas the mutant returns “1, David”.

(b) In the case of \texttt{COUNT(DISTINCT \ldots)}), the test case generation guidelines are different from the case of \texttt{COUNT()}. Please note that when the fields involved in the \texttt{IRC} mutation operator allow \texttt{null} values, it is possible to distinguish the mutants from the original SQL statement by setting the values in the related fields such that they will have different number of \texttt{null} values. Hence, the guidelines of setting different \texttt{null} values for \texttt{COUNT()} is also applicable for \texttt{COUNT(DISTINCT \ldots)} when the fields allow \texttt{null} values. On the other hand, unlike the situation in \texttt{COUNT()}, if the fields cannot be \texttt{null}, one can still distinguish the mutants with \texttt{COUNT(DISTINCT \ldots)} from its original SQL by the following steps:

i. Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the non-numeric field involved in the \texttt{WHERE} clause and

ii. Set the values in the other non-numeric fields in the \texttt{SELECT} clause for those records mentioned in (i) above so that these fields have different number of distinct values. To achieve this, I need to have multiple distinct values for the other non-numeric fields in the \texttt{SELECT} clause for the same boundary value in the non-numeric field in the \texttt{WHERE} clause.

For example, for the SQL statement “\texttt{SELECT COUNT(DISTINCT staffid), MIN(name) FROM STAFF WHERE city = ‘C8’ “}, when an \texttt{IRC} mutation operator replaces “staffid” in the aggregate function of the \texttt{SELECT} clause by “name”, the resultant mutant is “\texttt{SELECT COUNT(DISTINCT name), MIN(name) FROM STAFF WHERE city = ‘C8’ “}. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “city =
Chapter 4. Test Case Generation for IRC Operator

94

`C8’ " in the WHERE clause so as to set records in the database table such that, they will have ‘C7’, ‘C8’, and ‘C9’ in the non-numeric field in the WHERE clause (that is, “city”). For these records, I then set some distinct values for the “staffid” and the “name” field in the aggregate function of the SELECT clause. A possible example may be (“city” = ‘C7’, “staffid” = ‘S1’, “name”=‘Jack’), (‘C7’, ‘S2’, ‘Alex’), (‘C7’, ‘S3’, ‘Alex’), (‘C8’, ‘S3’, ‘Mike’), (‘C8’, ‘S3’, ‘Peter’), (‘C8’, ‘S4’, ‘James’) (‘C9’, ‘S4’, ‘David’), (‘C9’, ‘S5’, ‘David’), and (‘C9’, ‘S5’, ‘Adam’). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “2, James” whereas the mutant returns “3, James”.

2. Situation when the IRC-field is in an aggregate function other than COUNT() (i.e. MAX(), MIN()). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an IRC mutation operator replaces a non-numeric field f1 in the aggregate function AGGR() (may be MAX(), MIN(), ...) by another non-numeric field f2, the IRC mutant will have AGGR(f2) instead of AGGR(f1). Moreover, a different field, f3 is involved in the WHERE clause. In order to distinguish between AGGR(f1) and AGGR(f2), the guideline consists of the following two major steps:

(a) Apply BVA on the condition involving f3 in the WHERE clause to obtain the boundary values, and set the records in the database table so that each record will have one of these values in the field f3.

(b) Set the values of f1 and f2 so that f1 ≡ f3+S1 and f2 ≡ f3+S2 where S1 and S2 are two fixed non-null string constants for each record in the database table.

It does not matter whether there is a ‘DISTINCT’ keyword in the aggregate function, the same guideline is adequate to distinguish the mutants from the original SQL statement.

For example, for the SQL statement O4-2c in Table 4.1, when an IRC mutation operator replaces the non-numeric field “staffid” in the aggregate function (MAX()) of the SQL statement by another numeric field “name”, the resultant mutant is “SELECT MAX(name), MIN( name) FROM STAFF WHERE city = ‘C8’ ”. In order to
distinguish this type of mutant from O4-2c, I need to apply BVA on the condition 
“city = ‘C8’ ” in the \texttt{WHERE} clause so as to set three records in the database table 
such that, they will have ‘C7’, ‘C8’, and ‘C9’ in the non-numeric field in the \texttt{WHERE} 
clause (that is, “city”). For these records, I then set the values for the “staffid” and 
the “name” field so that “staffid” = “city” + ‘xy’ \footnote{constant may be a single or 
multiple (a group of) characters from 0-9, or a-z, or A-Z} and “name” = “city” + ‘pq’. 
An example may be (“city” = ‘C7’, “staffid” = ‘C7xy’, “name” = ‘C7pq’), (‘C8’, 
‘C8xy’, ‘C8pq’), and (‘C9’, ‘C9xy’, ‘C9pq’). By doing so, one can then distinguish 
O4-2c from the mutant because O4-2c returns “C8xy, C8pq” whereas the mutant 
returns “C8pq, C8pq”.

\subsection{IRC in the \texttt{WHERE} Clause}

As mentioned earlier, this subsection discusses the test case generation guidelines for SQL 
statements having aggregate functions and \texttt{WHERE} clause and the IRC mutation operator 
occurs in the \texttt{WHERE} clause. As explained earlier, four types of mutant, namely IRC-m21 
to IRC-m24, are possible, depending on whether the fields involved in the \texttt{SELECT} and 
\texttt{WHERE} clauses are numeric or non-numeric. The test case generation guidelines for these 
mutants are described as follows:

\textbf{G-IRC-m21:} For the IRC-m21 mutant, the IRC mutation operator replaces a numeric 
field in the \texttt{WHERE} clause of the SQL statement by another numeric field that also appears 
in the SQL statement and the IRC-field \footnote{Please refer to Table 4.3} also appears in an aggregate function of the 
\texttt{SELECT} clause. Because of the involvement of the numeric field, any types of aggregate 
functions can be applied to these numeric fields. Two different sets of guidelines are 
proposed depending on the characteristics of aggregate functions. In the first set, the 
guidelines are related to detecting mutants involving the aggregate function \texttt{COUNT()} and 
in the second set, the guidelines are about other aggregate functions.
1. Situation when the \textit{IRC}-field in the aggregate function \texttt{COUNT()}. In order to distinguish such mutants from the original SQL statement, the guideline consists of the following two major steps:

(i) Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the \textit{IRC}-field involved in the \texttt{WHERE} clause\footnote{These BVA values can also be used for other five relational operators like \texttt{\textless;}, \texttt{\textgreater;}, \texttt{\textless;=}, \texttt{\textgreater;=} and \texttt{\neq}.}.

(ii) Set the values for the other numeric fields of the \texttt{SELECT} clause for those records mentioned in (i) above so that

a) they are all distinct, and

b) either all of them are far greater or less than the boundary value.

For example, for the original SQL statement, \texttt{``SELECT COUNT(salary), COUNT(grade) FROM STAFF WHERE grade \textgreater; 40''}, when an \textit{IRC} mutation operator replaces \texttt{``grade''} in the \texttt{WHERE} clause by \texttt{``salary''}, the resultant mutant is \texttt{``SELECT COUNT(salary), COUNT(grade) FROM STAFF WHERE salary \textgreater; 40''}. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition \texttt{``grade \textgreater; 40''} in the \texttt{WHERE} clause so as to set three records in the database table such that they will have 39, 40, and 41 in the \textit{IRC}-field (that is, \texttt{``grade''}). For each of these values in the records, I then set the values for the \texttt{``salary''} field such that either all of them are far greater or less than the boundary value. An example may be \texttt{(```grade'' = 39, ``salary'' = 3)}, \texttt{(40, 5)}, \texttt{(41, 8)}. By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns \texttt{``1, 1''} whereas the mutant return \texttt{``0, 0''}. It should be noted that this guideline also works for the case of \texttt{COUNT(DISTINCT ...)} because all the values in the related numeric fields are distinct.

2. Situation when SQL statement is involved in an aggregate function other than \texttt{COUNT()} (i.e. \texttt{MAX()}, \texttt{MIN()}, \texttt{SUM()}, \texttt{AVG()}). Instead of having one guideline per aggregate function, I propose a guideline for all these aggregate functions. The
Table 4.6: Dataset that can kill the Mutants

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Relational Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX, MIN, SUM, AVG</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

guideline requires two datasets to be generated in order to kill all mutants. Table 4.6 summaries which dataset can be used to kill the relevant mutants involving different aggregate functions and relational operators. For example, for the SQL statement O4-1a in Table 4.1, a possible mutant is “SELECT MAX(grade), MIN(salary) FROM STAFF WHERE salary > 40”. Table 4.6 indicates that the first dataset can be used to distinguish this mutant. On the other hand, if the original SQL statement involves a non-equal relational operator(≠) in the WHERE clause, the first dataset cannot be used to distinguish the mutant from the original SQL statement. In that case, I need to use the second dataset. In the following I discuss the guidelines to generate these two datasets.

A. Guideline for the first dataset consists of the following two major steps:

(i) Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values for the IRC-field, and then set three records in the database table so that they will have one of these values in the IRC-field involved in the WHERE clause.

(ii) Apply the following rules to set the values of the other numeric fields for the three records mentioned in (i) above.

(a) For the record whose value of the IRC-field is less than the boundary value: For each numeric field other than the IRC-field, the value of this numeric field must satisfy the following conditions.

1. It is the maximum value among other values in the same field in other records and
2. It is at least twice as big as the boundary value.

(b) For the record whose value of the IRC-field is greater than the boundary value: For each numeric field other than the IRC-field, the value of this numeric field must satisfy the following conditions.

These BVA values can also be used for other five relational operators like ≤, >, =, < and ≠.
numeric field must be the minimum value among other values in the same field in other records and

c) For the record whose value of the IRC-field is equal to the boundary value: For each numeric field other than the IRC-field, the value of this numeric field must be greater than that mentioned in (b) above and smaller than the boundary value

For example, for the SQL statement O4-1a in Table 4.1, when an IRC mutation operator replaces “grade” in the WHERE clause by “salary”, the resultant mutant is “SELECT MAX(grade), MIN(salary) FROM STAFF WHERE salary > 40”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “grade>40” so as to set the records in the database table such that, they will have the values 39, 40, and 41 in the IRC-field (that is, “grade”). For these records, I then apply the rule in (ii) above on “salary” to set the records like (“grade”=39, “salary”=85), (40, 20), and (41, 15). By doing so, one can then distinguish the mutant from O4-1a because O4-1a returns “41, 15” whereas the mutant returns “39, 85”.

B. Guideline for second dataset consists of the following two steps:

(i) Set the values for the IRC-fields in the records of the database table so that they are all distinct and different from the boundary value and

(ii) Set the values for the other fields in the records of the database mentioned in (i) above so that the values are equal to the boundary value.

For example, one possible mutant of “SELECT MAX(grade), MIN(salary) FROM STAFF WHERE grade ≠ 40” is “SELECT MAX(grade), MIN(salary) FROM STAFF WHERE salary ≠ 40”. According to the guideline for the second dataset, an example may be (“grade” = 15, “salary” = 40) and (55, 40). By doing so, one can then distinguish the original SQL statement from the mutant because the original SQL statement returns “55, 40” whereas the mutant returns empty result set.

G-IRC-m22: Similar to IRC-m21, for the IRC-m22 mutant, the IRC mutation operator replaces a numeric field in the WHERE clause by another numeric field that also appears in
the SQL statement and the IRC-field\[^{45}\] does not appear in an aggregate function of the SELECT clause. Similar to that of G-IRC-m21, two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function \texttt{COUNT()} and in the second set, the guidelines are about other aggregate functions.

1. Situation when the aggregate function \texttt{COUNT()} is involved in the SQL statement. Similar to that of G-IRC-m21(1), to distinguish such mutants from the original SQL statement, the guideline consists of the following two major steps:

(i) Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the IRC-field involved in the \texttt{WHERE} clause\[^{46}\].

(ii) Set the values for the other numeric fields of the \texttt{SELECT} clause for those records mentioned in (i) above so that

i. they are all distinct, and

ii. either all of them are far greater or less than the boundary value.

For example, for the original SQL statement, "\texttt{SELECT COUNT(grade), COUNT(salary) FROM STAFF WHERE age > 30}" , when an IRC mutation operator replaces "age" in the \texttt{WHERE} clause by "salary", the resultant mutant is "\texttt{SELECT COUNT(grade), COUNT(salary) FROM STAFF WHERE salary > 30}". In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition "age >30" in the \texttt{WHERE} clause so as to set records in the database table such that, they will have 29, 30, and 31 in the IRC-field (that is "age"). For each of these values in the records, I then set the values for the "salary" and the "grade" fields such that either all of them are far greater than or less than the boundary value. An example may be ("age" = 29, "grade" = 3, "salary" = 5), (30, 6, 8), (31, 10, 12). By doing so, one can then distinguish the mutant from the original SQL statement because

\[^{45}\]Please refer to Table 4.3

\[^{46}\]These BVA values can also be used for other five relational operators like \(\leq\), \(>\), \(=\), \(<\) and \(\neq\)
Table 4.7: Dataset that can kill the Mutants

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Relational Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX, MIN, SUM, AVG</td>
<td>First First First First First Second</td>
</tr>
</tbody>
</table>

the original SQL statement returns “1, 1” whereas the mutant returns “0, 0”. It should be noted that this guideline also works for the each case of \texttt{COUNT(DISTINCT ... )} because all the values in the related numeric fields are distinct.

2. Situation when SQL statement is involved in an aggregate function other than \texttt{COUNT()} (e.g. \texttt{MAX()}, \texttt{MIN()}, \texttt{SUM()}, \texttt{AVG()}). Instead of having one guideline per aggregate function, I propose a guideline for all these aggregate functions. The guideline requires two datasets to be generated in order to kill all mutants. Table 4.7 summarizes which dataset can be used to kill the relevant mutants involving different aggregate functions and relational operators. For example, for the SQL statement O4-1b in Table 4.1, a possible mutant is \( \texttt{SELECT MAX(grade), MIN(salary) FROM STAFF WHERE salary > 30} \). Table 4.7 indicates that the first dataset can be used to distinguish this mutant. On the other hand, if the original SQL statement involves a non-equal relational operator \((\neq)\) in the \texttt{WHERE} clause, the first dataset cannot be used to distinguish the mutant from the original SQL statement. In that case, I need to use the second dataset. In the following I discuss the guidelines to generate these two datasets.

A. Guidelines for the first dataset consists of the following three major steps:

(i) Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values for the \texttt{IRC}-field, and then set three records in the database table so that they will have one of these values in the \texttt{IRC}-field involved in the \texttt{WHERE} clause\(^{47}\).

(ii) Apply the following rules to set the values of the other numeric fields for the three records mentioned in (i) above.

\(^{47}\)These BVA values can also be used for other five relational operators like \(\leq, \geq, =, <\) and \(\neq\).
(a) For the record whose value of the *IRC*-field is less than the boundary value: For each numeric field other than the *IRC*-field, the value of this numeric field must satisfy the following conditions.

1. it is the maximum value among other values in the same field in other records and
2. it is at least twice as big as the boundary value.

(b) For the record whose value of the *IRC*-field is greater than the boundary value: For each numeric field other than the *IRC*-field, the value of this numeric field must be the minimum value among other values in the same field in other records and

(c) For the record whose value of the *IRC*-field is equal to the boundary value: For each numeric field other than the *IRC*-field, the value of this numeric field must be greater than that mentioned in (b) above and smaller than the boundary value.

For example, for the SQL statement O4-1b in Table 4.1, when an *IRC* mutation operator replaces a numeric field “age” in the *WHERE* clause by another numeric field “salary” that appears in the aggregate function of the *SELECT* clause, the resultant mutant is `SELECT MAX(grade), MIN(salary) FROM STAFF WHERE salary > 30`]. In order to distinguish this type of mutant from O4-1b, I need to apply the BVA on the condition “age > 30” in the *WHERE* clause so as to set the records in the database table such that, they will have 29, 30, and 31 in the *IRC*-field (that is “age”). For these records, I then apply the rule in (ii) above on the “grade” and the “salary” fields to set the records like (“age” = 29, “grade” = 65, “salary” = 75), (30, 20, 25), and (31, 15, 18). By doing so, one can then distinguish the mutant from O4-1b because O4-1b returns “15, 18” whereas the mutant returns “65, 75”.

B. Guideline for second dataset consists of the following two steps:

(i) Set the values for the *IRC*-fields in the records of the database table so that they are all distinct and different from the boundary value and
(ii) Set the values for the other fields in the records of the database mentioned in (i) above so that the values are equal to the boundary value.

For example, one possible mutant of "\texttt{SELECT MAX(grade), MIN(salary) FROM STAFF WHERE age \neq 30}" is "\texttt{SELECT MAX(grade), MIN(salary) FROM STAFF WHERE salary \neq 30}". According to the guideline for the second dataset, an example may be ("age" = 15, "grade" = 30, "salary" = 30) and (55, 30, 30). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns "30, 30" whereas the mutant returns empty result set.

\textbf{G-IRC-m23}: For the IRC-m23 mutants, the \textit{IRC} mutation operator replaces a non-numeric field in the simple \texttt{WHERE} clause of the SQL statement by another non-numeric field that also appear in the SQL statement and the \textit{IRC-field} also appears in the aggregate function of the \texttt{SELECT} clause. As this SQL statement involves non-numeric fields; \texttt{MAX()}, \texttt{MIN()}, and \texttt{COUNT()} are the only legitimate aggregate functions. Again, two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function \texttt{COUNT()} and in the second set, the guidelines are about other aggregate functions.

1. Situation when the aggregate function \texttt{COUNT()} is involved in the SQL statement.

In order to distinguish such mutants from the original SQL statement, the guideline is proposed using the similar technique of G-IRC-m21(1), and it consists of the following two major steps:

(i) Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the \textit{IRC-field} involved in the \texttt{WHERE} clause\footnote{Please refer to Table 4.3}.

(ii) Set the values for the other non-numeric fields of the \texttt{SELECT} clause for those records mentioned in (i) above so that

\footnote{These BVA values can also be used for other five relational operators like \texttt{\leq}, \texttt{\geq}, \texttt{=}, \texttt{<} and \texttt{\neq}}
Table 4.8: Dataset that can kill the Mutants

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Relational Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&gt;</td>
</tr>
<tr>
<td>MAX, MIN</td>
<td>First</td>
</tr>
</tbody>
</table>

i. they are all distinct, and

ii. either all of them are far greater or less than the boundary value.

For example, for SQL statement “SELECT COUNT(staffid), COUNT(name) FROM STAFF WHERE staffid = ‘S1’ ”, when an IRC mutation operator replaces a non-numeric field “staffid” in the WHERE clause of the SQL statement by “name”, the resultant mutant is “SELECT COUNT(staffid), COUNT(name) FROM STAFF WHERE name = ‘S1’ ”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “staffid = ‘S1’ ” in the WHERE clause so as to set three records in the database table such that, they will have ‘S1’, ‘S2’, and ‘S3’ in the IRC-field (that is, “staffid”). For each of these values in the records, I then set the values of the “name” field such that either all of them are far greater or less than the boundary value. An example may be (staffid = ‘S1’, name=‘Adam’), (‘S2’, ‘David’), and (‘S3’, ‘Jack’). By doing so, one can then distinguish the mutants from the original SQL statement because the original SQL returns “1, 1” whereas the mutant returns “0, 0”. It should be noted that this guideline also works for the case of COUNT(DISTINCT ...) because all the values in the related non-numeric fields are distinct.

2. Situation when the IRC-field is involved in an aggregate function other than COUNT() (i.e. MAX(), MIN()). Instead of having one guideline per aggregate function, I propose a guideline for all these aggregate functions. The guideline requires two datasets to be generated in order to kill all mutants. Table 4.8 summaries which dataset can be used to kill the relevant mutants involving different aggregate functions and relational operators. For example, for the SQL statement “SELECT MAX(name), MIN(staffid) FROM STAFF WHERE staffid = ‘S1’ ”, a possible mutant is “SELECT MAX(name), MIN(staffid) FROM STAFF WHERE name = ‘S1’ ”. Table 4.8 indicates that the first dataset can be used to distinguish this mutant. On the other hand, if
the original SQL statement involves a non-equal relational operator (≠) in the `WHERE` clause, the first dataset cannot be used to distinguish the mutant from the original SQL statement. In that case, I need to use the second dataset to distinguish the mutants from the original SQL statement. In the following I discuss the guidelines to generate these two datasets.

A. Guidelines for the first dataset consists of the following three major steps:

(i) Apply boundary value analysis (BVA) on the condition in the `WHERE` clause to obtain the boundary values for the `IRC`-field, and then set three records in the database table so that they will have one of these values in the `IRC`-field involved in the `WHERE` clause.

(ii) Apply the following rules to set the values of other non-numeric fields for the three records mentioned in (i) above.

(a) For the record whose value of the `IRC`-field is less than the boundary value: For each non-numeric field other than the `IRC`-field, the value of this non-numeric field must be the maximum value among other values in the same field in other records,

(b) For the record whose value of the `IRC`-field is greater than the boundary value: For each non-numeric field other than the `IRC`-field, the value of this non-numeric field must be the minimum value among other values in the same field in other records, and

(c) For the record whose value of the `IRC`-field is equal to the boundary value: For each non-numeric field other than the `IRC`-field, the value of this non-numeric field must be greater than that mentioned in (b) above and smaller than the boundary value

For example, for the SQL statement O4-2b in Table 4.1, when an `IRC` mutation operator replaces the non-numeric field “staffid” in the `WHERE` clause by another non-numeric field “name”, the resultant mutant is “SELECT MAX(staffid), MIN(name) FROM STAFF WHERE name = ‘S1’ ”. In order to distinguish this type of mutant from O4-2b, I need to apply BVA on the condition “staffid = ‘S1’ ” in

\[50\] These BVA values can also be used for other five relational operators like ≤, >, =, < and ≠.
the WHERE clause so as to set records in the database table such that, they will have ‘S0’, ‘S1’, and ‘S2’ in the IRC-field (that is, “staffid”). For these records, I then apply the rule in (ii) above on the “name” field in the aggregate function in the SELECT clause to set the records like (“staffid” = ‘S0’, “name” = ‘Tony’), (‘S1’, ‘David’), (‘S2’, ‘Adam’). By doing so, one can then distinguish the mutant from O4-2b because O4-2b returns “S1, David” whereas the mutant returns “null, null”.

B. Guideline for second dataset consists of the following two steps:

(i) Set the values for the IRC-fields in the records of the database table so that they are all distinct and different from the boundary value and

(ii) Set the values for the other fields in the records of the database mentioned in (i) above so that the values are equal to the boundary value.

For example, one possible mutant of “SELECT MAX(staffid), MIN(name) FROM STAFF WHERE staffid ≠ ‘S1’ ” is “SELECT MAX(staffid), MIN(name) FROM STAFF WHERE name ≠ ‘S1’ ”. According to the guideline for the second dataset, an example may be (”staffid” = ‘S5’, “name” = ‘S1’), (‘S7’, ‘S1’). By doing so, one can then distinguish the mutants from the original SQL statement because the original SQL statement returns “S7, S1” whereas the mutant returns “null, null”.

G-IRC-m24: Similar to IRC-m23, for the IRC-m24 mutant, the IRC mutation operator replaces a non-numeric field in the WHERE clause by another non-numeric field that also appear in the SQL statement and the IRC-field does not appear in the aggregate function of the SELECT clause. In order to distinguish the results of the original SQL statements from those mutants, two different sets of guidelines are proposed depending on the characteristic of the aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function COUNT() and in the second set, the guidelines are about other aggregate functions.

Please refer Table 4.3
Chapter 4. Test Case Generation for IRC Operator

1. Situation when the aggregate function $\text{COUNT()}$ is involved in the SQL statement and $IRC$-field does not appear in $\text{SELECT}$ clause, the guideline consists of the following steps:

   (i) Apply boundary value analysis (BVA) on the condition in the $\text{WHERE}$ clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the $IRC$-field involved in the $\text{WHERE}$ clause\(^{52}\).

   (ii) Set the values for the other non-numeric fields of the $\text{SELECT}$ clause for those records mentioned in (i) above so that
   
   i. they are all distinct, and
   
   ii. either all of them are far greater or less than the boundary value.

   For example, for the SQL statement, \texttt{"SELECT COUNT(staffid), COUNT(name), FROM STAFF WHERE city = ‘C8’"}, when \texttt{"city"} is replaced by \texttt{"name"} the resultant mutant is \texttt{"SELECT COUNT(staffid), COUNT(name), FROM STAFF WHERE name = ‘C8’"}. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition \texttt{"city = ‘C8’"} so as to set records in the database table such that, they will have \texttt{‘C7’}, \texttt{‘C8’}, and \texttt{‘C9’} in the $IRC$-field (that is, \texttt{"city"}). For these records, I then set the values for the \texttt{"staffid"} and the \texttt{"name"} fields such that either all of them are far greater or less than the boundary value. An example may be (\texttt{"city"}= \texttt{‘C7’}, \texttt{"staffid"}= \texttt{‘S1’}, \texttt{"name"}= \texttt{‘Jack’}), (\texttt{‘C8’}, \texttt{‘S2’}, \texttt{‘David’}), (\texttt{‘C9’}, \texttt{‘S3’}, \texttt{‘Tom’}). By doing so, one can then distinguish the mutant from the original SQL statement because the original returns \texttt{“1, 1”} whereas the mutant returns \texttt{“0, 0”}. It should be noted that this guideline also works for the case of $\text{COUNT(DISTINCT . . .)}$ because all the values in the related non-numeric fields are distinct.

2. Situation when the $IRC$-field is involved in an aggregate function other than $\text{COUNT()}$ (i.e. $\text{MAX()}$, $\text{MIN()}$). Instead of having one guideline per aggregate function, I propose a guideline for all these aggregate functions. The guideline requires two datasets to be generated in order to kill all mutants. Table 4.9 summaries which dataset can

\(^{52}\text{These BVA values can also be used for other five relational operators like } \leq, \geq, =, < \text{ and } \neq\)
Table 4.9: Dataset that can kill the Mutants

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Relational Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX(staffid), MIN(name)</td>
<td>≥, ≤, =, ≠</td>
</tr>
</tbody>
</table>

be used to kill the relevant mutants involving different aggregate functions and relational operators. For example, for the SQL statement O4-2c in Table 4.1, a possible mutant is “SELECT MAX(staffid), MIN(name) FROM STAFF WHERE name = ‘C8’”. Table 4.9 indicates that the first dataset can be used to distinguish this mutant. On the other hand, if the original SQL statement involves a non-equal relational operator (≠) in the WHERE clause, the first dataset cannot be used to distinguish the mutant from the original SQL statement. In that case, I need to use the second dataset. In the following I discuss the guidelines to generate these two datasets.

A. Guidelines for the first dataset consists of the following three major steps:

(i) Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values for the IRC-field, and then set three records in the database table so that they will have one of these values in the IRC-field involved in the WHERE clause.

(ii) Apply the following rules to set the values of other non-numeric fields for the three records mentioned in (i) above.

(a) For the record whose value of the IRC-field is less than the boundary value: For each non-numeric field other than the IRC-field, the value of this non-numeric field must be the maximum value among other values in the same field in other records,

(b) For the record whose value of the IRC-field is greater than the boundary value: For each non-numeric field other than the IRC-field, the value of this non-numeric field must be the minimum value among other values in the same field in other records, and

(c) For the record whose value of the IRC-field is equal to the boundary value: For each non-numeric field other than the IRC-field, the value

53 These BVA values can also be used for other five relational operators like ≤, >, =, < and ≠.
of this non-numeric field must be greater than that mentioned in (b) above and smaller than the boundary value.

For example, for the SQL statement O4-2c in Table 4.1, when an IRC mutation operator replaces “city” in the WHERE clause by “name” the resultant mutant is “SELECT MAX(staffid), MIN(name) FROM STAFF WHERE name = ‘C8’ ”. In order to distinguish this type of mutant from the original, I need to apply BVA on the condition “city = ‘C8’ ” in the WHERE clause so as to set records in the database table such that, they will have ‘C7’, ‘C8’, and ‘C9’ in the IRC-field (that is “city”). For these records, I then apply the rule in (ii) above on the “staffid” and the “name” field to set the records like (“city” = ‘C7’, “staffid” = ‘S1’, “name” = ‘Tony’), (‘C8’, ‘B1’, ‘Bill’), (‘C9’, ‘A1’, ‘Adam’). By doing so, one can then distinguish the results of the original from those mutants as the original SQL returns “B1, Bill” whereas the mutant returns “null, null”.

B. Guideline for second dataset consists of the following two steps:

(i) Set the values for the IRC-fields in the records of the database table so that they are all distinct and different from the boundary value and

(ii) Set the values for the other fields in the records of the database mentioned in (i) above so that the values are equal to the boundary value.

For example, one possible mutant of “SELECT MAX(staffid), MIN(name) FROM STAFF WHERE city ≠ ‘C8’ ” is “SELECT MAX(staffid), MIN(name) FROM STAFF WHERE name ≠ ‘C8’ “. According to the guideline for the second dataset, an example may be (“city” = ‘C9’, “staffid” = ‘C8’, “name” = ‘C8’), (‘C5’, ‘C8’, ‘C8’). By doing so, one can then distinguish the mutants from the original SQL statement because the original SQL statement returns “C8, C8” whereas the mutant returns “null, null”.

Table 4.10 summarizes all guidelines for 24 different types of IRC mutants that may occur in the SQL statement as discussed in Table 4.1 and Table 4.3.
<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><em>IRC</em> mutation operator occurs on the numeric field in the <em>SELECT</em> clause</td>
<td>G1</td>
<td>Set the values of the numeric fields so that they are distinct for every record in the database table.</td>
<td>(grade = 5, salary = 1500), (8, 3000), (10, 4000)</td>
</tr>
<tr>
<td>2</td>
<td><em>IRC</em> mutation operator occurs on the non-numeric field in the <em>SELECT</em> clause</td>
<td>G2</td>
<td>Set the values of the non-numeric fields so that they are distinct for every record in the database table.</td>
<td>(“staffid”=‘S1’, “name”=‘Jake’), (“S2”, ‘David’), (“S3’, ‘Michael’)</td>
</tr>
<tr>
<td>3</td>
<td><em>COUNT()</em>-<em>IRC</em> mutation operator occurs on the numeric field in the aggregate function of the <em>SELECT</em> clause</td>
<td>G3-1(a)</td>
<td>If <em>null</em> values are allowed to the fields, set different number of null values for those numeric fields in the <em>SELECT</em> clause. Otherwise, the mutants are equivalent to the original SQL statement.</td>
<td>(grade = 1, salary = 3), (2, 3), (3, 2)</td>
</tr>
<tr>
<td></td>
<td><strong>COUNT</strong> <em>(DISTINCT...)</em>-<em>IRC</em> mutation operator occurs on the numeric field in the aggregate function of the <em>SELECT</em> clause</td>
<td>G3-1(b)</td>
<td>If <em>null</em> values are allowed to the fields guidelines is same as G3-1(a). Otherwise, set the values in the numeric fields so that these fields have different number of distinct values.</td>
<td>(“grade”=3, “salary”=1003), (4, 1004), (5, 1005)</td>
</tr>
<tr>
<td></td>
<td>(Other aggregate functions) <em>IRC</em> mutation operator occurs on the numeric field in the <em>SELECT</em> clause</td>
<td>G3-2</td>
<td>If there are two fields -f1 and f2; (1) set any non-zero values for f1, (2) set f2 so that it is equal to corresponding values in f1 plus a fixed non-zero constant C for each record in the database table (f2 ≡ f1+C).</td>
<td>(“grade”=3, “salary”=1003), (4, 1004), (5, 1005)</td>
</tr>
</tbody>
</table>
### Chapter 4. Test Case Generation for IRC Operator

Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>COUNT- IRC mutation operator occurs on the non-numeric field in aggregate function of the SELECT clause</td>
<td>G4-1(a)</td>
<td>Set values for the non-numeric fields in the SELECT clause using the same technique of guideline G3-1(a)</td>
<td>(“staffid”='S1', “name”='Jack'), (null, ‘David’), (‘S2’, ‘Adam’)</td>
</tr>
<tr>
<td></td>
<td>COUNT (DISTINCT...) IRC mutation operator occurs on the non-numeric field in the aggregate function of the SELECT clause</td>
<td>G4-1(b)</td>
<td>Set values for the non-numeric fields in the SELECT clause using the same technique of guideline G3-1(b)</td>
<td>{ (“staffid” = ‘S1’, “name” = ‘Jack’), (‘S2’, ‘Jack’), (‘S3’, ‘Adam’)}</td>
</tr>
<tr>
<td></td>
<td>(Other aggregate functions): IRC mutation operator occurs on the non-numeric field in the SELECT clause</td>
<td>G4-2</td>
<td>If there are two fields f1 and f2; (1) set any non-null values for f1 in the records of the database, (2) set f2 so that it is equal to corresponding values in f1 plus a fixed constant string S for each record in the database table (f2 ≡ f1+S).</td>
<td>(“staffid”=‘S2’, “name”=‘S2xy’), (‘S5’, ‘S5xy’), (‘S10’, ‘S10xy’)}</td>
</tr>
</tbody>
</table>
Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td><em>IRC</em> mutation operator occurs on the numeric field in the <em>SELECT</em> clause, and the <em>IRC</em>-field also appears in the <em>WHERE</em> clause</td>
<td>G5</td>
<td>(1) Apply BVA rule on the condition in the <em>WHERE</em> clause to obtain BVA values in the records of the database, then (2) Apply reverse-BVA rule for other numeric field in the <em>SELECT</em> clause, and (3) Add some extra records so that the values of other numeric fields are equal to the boundary value and the values of the <em>IRC</em>-field are not equal to the boundary value.</td>
<td>(“grade”=39, “salary” = 82), (40, 85), and (41, 28), (39, 40), (41, 40)</td>
</tr>
<tr>
<td>6</td>
<td><em>IRC</em> mutation operator occurs on the numeric field in the <em>SELECT</em> clause, and the <em>IRC</em>-field does not appear in the <em>WHERE</em> clause</td>
<td>G6</td>
<td>Same as G5</td>
<td>(“age” = 29, “grade” = 32, “salary” = 42), (30, 25, 35), (31, 28, 18), (29, 30, 30), and (31, 30, 30)</td>
</tr>
<tr>
<td>7</td>
<td><em>IRC</em> mutation operator occurs on the numeric field in the <em>SELECT</em> clause, and a non-numeric field appears in the <em>WHERE</em> clause</td>
<td>G7</td>
<td>(1) Apply BVA rule on the condition in the <em>WHERE</em> clause to obtain BVA values in those records of the database, (2) Set distinct values for the numeric fields in the <em>SELECT</em> clause for those records mentioned in (1).</td>
<td>(“staffid” = ‘S0’, “grade” = 5, “salary” = 1000), (‘S1’, 10, 2000), and (‘S2’, 15, 3000)</td>
</tr>
<tr>
<td>8</td>
<td><em>IRC</em> mutation operator occurs on the non-numeric field in the <em>SELECT</em> clause, and a numeric field appears in the <em>WHERE</em> clause</td>
<td>G8</td>
<td>(1) Apply BVA rule on the condition in the <em>WHERE</em> clause to obtain BVA values in those records of the database, (2) Set distinct values for the non-numeric fields in the <em>SELECT</em> clause for those records mentioned in (1).</td>
<td>(“grade” = 39, “staffid”=‘S1’, “name” = ‘Jack’), (40, ‘S2’, ‘David’), (41, ‘S3’, ‘James’)</td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
<td>Example</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>9</td>
<td><em>IRC</em> mutation operator occurs on the non-numeric field in the <code>SELECT</code> clause and the <em>IRC</em>-field also appears in the <code>WHERE</code> clause</td>
<td>G9</td>
<td>(1) Apply BVA rule on the condition in the <code>WHERE</code> clause to obtain BVA values in the records of the database, (2) Set different and distinct values for other fields in the <code>SELECT</code> clause for those records mentioned in (1)</td>
<td>(&quot;staffid&quot; = 'S0', &quot;name&quot; = 'Jack'), ('S1', 'David'), ('S2', 'Micheal')</td>
</tr>
<tr>
<td>10</td>
<td><em>IRC</em> mutation operator occurs on the non-numeric field in the <code>SELECT</code> clause and the <em>IRC</em>-field does not appear in the <code>WHERE</code> clause</td>
<td>G10</td>
<td>Same as G9</td>
<td>(&quot;city&quot; = 'C7', &quot;staffid&quot; = 'S1', &quot;name&quot; = 'Jack'), ('C8', 'S2', 'David'), ('C9', 'S3', 'Aron')</td>
</tr>
<tr>
<td>11</td>
<td><em>IRC</em> mutation operator occurs on the numeric field in the <code>WHERE</code> clause and the <em>IRC</em>-field also appears in the <code>SELECT</code> clause</td>
<td>G11</td>
<td>Same as G5; (1) Apply BVA rule on the condition in the <code>WHERE</code> clause to set the BVA values in the records of the database (2) Apply reverse-BVA rule for other numeric fields in the <code>SELECT</code> clause, and (3) Add some extra records so that the value of the other numeric fields are equal to the boundary value and the values of the <em>IRC</em>-field not equal to the boundary value.</td>
<td>(&quot;grade&quot;=39, &quot;salary&quot; = 82), (40, 85), (41, 28), (39, 40) and (41, 40)</td>
</tr>
<tr>
<td>12</td>
<td><em>IRC</em> mutation operator occurs on the numeric field in the <code>WHERE</code> clause and the <em>IRC</em>-field does not appear in the <code>SELECT</code> clause</td>
<td>G12</td>
<td>Same as G11</td>
<td>(&quot;age&quot; = 29, &quot;grade&quot; = 32, &quot;salary&quot; = 42), (30, 25, 35), (31, 28, 18), (29, 30, 30) and (31, 30, 30)</td>
</tr>
</tbody>
</table>
### Chapter 4. Test Case Generation for IRC Operator

Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td><strong>IRC</strong> mutation operator occurs on the non-numeric field in the <strong>WHERE</strong> clause and the <strong>IRC</strong>-field also appears in the <strong>SELECT</strong> clause</td>
<td>G13</td>
<td>Same as G9</td>
<td>(‘staffid’ = ‘S0’, ‘name’ = ‘Jack’), (‘S1’, ‘David’), (‘S2’, ‘Micheal’)</td>
</tr>
<tr>
<td>14</td>
<td><strong>IRC</strong> mutation operator occurs on the non-numeric field in the <strong>WHERE</strong> clause and the <strong>IRC</strong>-field does not appear in the <strong>SELECT</strong> clause</td>
<td>G14</td>
<td>Same as G10</td>
<td>(‘city’ = ‘C7’, ‘staffid’ = ‘S1’, name = ‘Jack’), (‘C8’, ‘S2’, ‘David’), and (‘C9’, ‘S3’, ‘Aron’)</td>
</tr>
<tr>
<td>15</td>
<td><strong>COUNT()</strong> - <strong>IRC</strong> occurs on the numeric field in the aggregate function of SQL statement and the <strong>IRC</strong>-field also appears in the <strong>WHERE</strong> clause</td>
<td>G15-1(a)</td>
<td>If the fields involved in <strong>IRC</strong> do not allow null values, mutants are equivalent to the original SQL statement. Otherwise, (1) apply BVA rule on the condition in the <strong>WHERE</strong> clause to obtain BVA values in the records of the database, (2) Set different number of null values in the other numeric fields of the <strong>SELECT</strong> clause for those records mentioned in (1)</td>
<td>(‘grade’ = 39, “salary” = 1500), (39, null), (40, 1600), (40, null), (41, 1700), and (41,null)</td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
<td>Example</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>16</td>
<td>COUNT() - IRC occurs on the numeric field in the aggregate function of the SQL statement but the IRC-field does not appear in the WHERE clause</td>
<td>G16-1(a)</td>
<td>Same as G15-1(a)</td>
<td>(“age” = 29, “grade” = 6, “salary” = 500), (29, 7, 600), (29, 7, 600) (30, 10, 1000), (30, 25, 2000), (30, 35, 2000), (31, 20, 2500), (31, 25, 3000), and (31, 25, 3500)</td>
</tr>
<tr>
<td>15</td>
<td>COUNT(DISTINCT ...) - IRC occurs on the numeric field in the aggregate function of the SQL statement and the IRC-field also appears in the WHERE clause</td>
<td>G15-1(b)</td>
<td>If fields involved in IRC allow null value, same as G15-1(a). But when fields involved in IRC do not allow null value, (1) apply BVA rule on the condition in the WHERE clause to obtain BVA values in the records of the database, (2) Set different number of distinct values in the other numeric fields in the SELECT clause for those records mentioned in (1)</td>
<td>(“grade” = 39, “salary” = 1500), (39, 2500), (40, 1600), (40, 2600), (41, 1700), and (41, 2700)</td>
</tr>
<tr>
<td></td>
<td>Other aggregate function - IRC occurs on the numeric field in the aggregate function of the SQL statement and the IRC-field also appears in the WHERE clause</td>
<td>G15-2</td>
<td>If two numeric fields - f1 and f2, where f2 appears only in the SELECT clause and f1 appears both in the WHERE and the SELECT clause; (1) Apply BVA on f1 to obtain and set BVA values in the records of the database table, (2) Set the values of f2 so that f2 = f1+C, where C is a fixed non-zero constant for those records in the database table mentioned in (1) above.</td>
<td>(“grade” = 39, “salary” = 1039), (40, 1040), and (41, 1041)</td>
</tr>
</tbody>
</table>
### Chapter 4. Test Case Generation for IRC Operator

**Continued from previous page**

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COUNT(DISTINCT ...)</strong> - IRC occurs on the numeric field in the aggregate function of the SQL statement but the IRC-field does not appear in the <strong>WHERE</strong> clause</td>
<td>G16-1(b)</td>
<td>Same as G15-1(b)</td>
<td>(“age” = 29, “grade” = 6, “salary” = 500), (29, 7, 600), (29, 7, 600) (30, 10, 1000), (30, 25, 2000), (30, 35, 2000), (31, 20, 2500), (31, 25, 3000), and (31, 25, 3500)</td>
<td></td>
</tr>
<tr>
<td>Other aggregate function - IRC occurs on the numeric field in the aggregate function of the SQL statement but the IRC-field does not appear in the <strong>WHERE</strong> clause</td>
<td>G16-2</td>
<td>If there are three numeric fields - f1, f2, and f3. Among them f1 and f2 appear in the SELECT clause whereas f3 appears in the WHERE clause; (1) Apply BVA on f3 to obtain and set BVA values in the records of the database, (2) Set f1 and f2 so that f1 = f3 + C1 and f2 = f3 + C2, where C1 and C2 are two fixed non-zero constants for those records mentioned in (1) above.</td>
<td>(“age” = 29, “grade” = 1029, “salary” = 2029), (30, 1030, 2030), and (31, 1031, 2031). Here constants are 1000 and 2000.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td><strong>COUNT()</strong> - IRC occurs on the numeric field in the aggregate function of the SQL statement and a non-numeric field appears in the <strong>WHERE</strong> clause</td>
<td>G17-1(a)</td>
<td>If the field involved in IRC does not allow null values, the mutants are equivalent to the original SQL statement. Otherwise, (1) Apply BVA rule on the condition in the WHERE clause to obtain and set the BVA values in the records of the database, (2) Set different number of null values for the other fields in the SELECT clause for those records mentioned in (1)</td>
<td>(“staffid” = ‘S0’, “grade” = 10, “salary” = 100), (‘S0’, 15, null), (‘S1’, 5, 500), (‘S1’, null, 1000), (‘S2’, 20, 1500), and (‘S2’, null, 2000)</td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
<td>Example</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------</td>
<td>----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>17</td>
<td>COUNT(DISTINCT ...) - IRC occurs on the numeric field in the aggregate function of the SQL statement and a non-numeric field appears in the WHERE clause</td>
<td>G17-1(b)</td>
<td>If fields involved in IRC allow null value, same as G17-1(a). But when fields involved in IRC do not allow null values, (1) Apply BVA rule on the condition in the WHERE clause to obtain and set the BVA values in the records of the database, (2) Set different distinct values for the other fields in the SELECT clause for those records mentioned in (1).</td>
<td>(“staffid” = ‘S0’, “grade” = 10, “salary” = 100), (‘S0’, 10, 200), (‘S0’, 15, 300), (‘S1’, 5, 500), (‘S1’, 6, 200), (‘S1’, 7, 200), (‘S2’, 15, 1000), and (‘S2’, 15, 1500), (‘S2’, 20, 2000)</td>
</tr>
<tr>
<td>17</td>
<td>Other aggregate function - IRC occurs on the numeric field in the aggregate function of the SQL statement and a non-numeric field appears in the WHERE clause</td>
<td>G17-2</td>
<td>(1) Apply BVA rule on the condition in the WHERE clause to obtain and set the BVA values in the records of the database, (2) If there are two numeric fields, f1 and f2 in the SELECT clause; (1) Set any non-zero values for f1, (2) Set f2 so that f2 = f1 + C, where C is a fixed non-zero constant for those records mentioned in (1) above.</td>
<td>(“staffid” = ‘S1’, “grade” = 1, “salary” = 1001), (‘S0’, 5, 1005), (‘S1’, 10, 1010), (‘S1’, 15, 1015), (‘S2’, 20, 1020), and (‘S2’, 25, 1025) (Note that, C = 1000)</td>
</tr>
<tr>
<td>18</td>
<td>COUNT() - IRC occurs on the non-numeric field in the aggregate function of SQL statement and a numeric field appears in the WHERE clause</td>
<td>G18-1(a)</td>
<td>When fields involved in IRC allow null values, (1) Apply BVA rule on the condition in the WHERE clause to obtain and set BVA values in the records of the database, (2) Set different number of null values for the other non-numeric fields in the SELECT clause for those records mentioned in (1).</td>
<td>(grade= 39, staffid =‘S1’, name= ‘Jack’), (39, ‘S2’, ‘Jim’), (40, ‘S3’, null), (40, ‘S4’, null), (41, ‘S5’, ‘James’), and (41, ‘S6’, null)</td>
</tr>
</tbody>
</table>
### Chapter 4. Test Case Generation for IRC Operator

Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>COUNT(DISTINCT ...) - IRC occurs on the numeric field in the aggregate function of the SQL statement and a non-numeric field appears in the <strong>WHERE</strong> clause</td>
<td>G18-1(b)</td>
<td>If fields involved in IRC allow null value, same as G15-1(a). But when fields involved in IRC do not allow null values, (1) Apply BVA rule on the condition in the <strong>WHERE</strong> clause, (2) Set different distinct values for the other fields in the <strong>SELECT</strong> clause for those records mentioned in (1)</td>
<td>(“grade” = 39, “staffid” = ‘S1’, “name” = ‘Jack’), (39, ‘S2’, Jack), (40, ‘S4’, ‘Gill’), (40, ‘S4’, ‘David’), (41, ‘S5’, ‘James’), and (41, ‘S6’, ‘James’)</td>
</tr>
<tr>
<td>18</td>
<td>Other aggregate function - IRC occurs on the numeric field in the aggregate function of the SQL statement and a non-numeric field appears in the <strong>WHERE</strong> clause</td>
<td>G18-2</td>
<td>(1) Apply BVA rule on the condition in the <strong>WHERE</strong> clause, (2) If there are two fields -f1 and f2 in the <strong>SELECT</strong> clause; (i) set any non-null values for f1, (ii) set f2 so that f2 ( \equiv f1+S ), where S is a fixed non-null string constant for each record in the database table mentioned in (i) above.</td>
<td>(“grade” = 39, “staffid” = ‘S2’, “name” = ‘S2xy’), (40, ‘S5’, ‘S5xy’), and (41, ‘S10’, ‘S10xy’)</td>
</tr>
<tr>
<td>19</td>
<td>COUNT() - IRC occurs on the non-numeric field in the aggregate function of the <strong>SELECT</strong> clause and the IRC-field also appears in the <strong>WHERE</strong> clause</td>
<td>G19-1(a)</td>
<td>Same technique as guideline G15-1(a); When the fields involved in IRC allow null values, (1) Apply BVA rule on the condition in the <strong>WHERE</strong> clause to obtain and set BVA values in the records of the database table, (2) Set different number of null values in the other non-numeric fields of the <strong>SELECT</strong> clause for those records mentioned in (1)</td>
<td>(“staffnum” = ‘S0’, “name” = ‘Jack’), (‘S0’, null), (‘S1’, null), (‘S1’, ‘David’), (‘S2’, null), and (S2, ‘Mark’)</td>
</tr>
</tbody>
</table>
**Continued from previous page**

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>COUNT(DISTINCT ...)</code> - IRC occurs on the non-numeric field in the aggregate function of the SQL statement and the IRC-field also appears in the <strong>WHERE</strong> clause</td>
<td>G19-1(b)</td>
<td>Same technique as guideline G15-1(b); When fields involved in IRC allow null values, guidelines are same as G19-1(a). Otherwise, (1) Apply BVA rule on the condition in the <strong>WHERE</strong> clause to obtain BVA values in the records of the database (2) Set different number of distinct values in the other non-numeric fields in the <strong>SELECT</strong> clause for those records mentioned in (1) above.</td>
<td>(“staffid” = ‘S1’, “name”=‘Jack’), (‘S1’, ‘David’), (‘S2’, ‘Mike’), (‘S2’, ‘Peter’), (‘S0’, ‘Alex’), (‘S0’, ‘Luke’)</td>
<td></td>
</tr>
<tr>
<td>Other aggregate function - IRC occurs on the non-numeric field in the aggregate function of the <strong>SELECT</strong> clause and the IRC-field also appears in the <strong>WHERE</strong> clause</td>
<td>G19-2</td>
<td>If there are two non-numeric fields - f1 and f2 and f2 appears only in the <strong>SELECT</strong> clause whereas f1 appears both in the <strong>WHERE</strong> and the <strong>SELECT</strong> clause; (1) Apply BVA on f1 to obtain BVA values in the records of the database, (2) set f2 so that f2 (\equiv f1+S), where S is a fixed non-null string constant S for those records mentioned in (1) above.</td>
<td>(“staffid” =’S0’, “name”=’S0xy’), (‘S1’, ‘S1xy’), and (‘S2’, ‘S2xy’)</td>
<td></td>
</tr>
</tbody>
</table>
Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other aggregate function - IRC occurs on the non-numeric field in the aggregate function of the SQL statement and the IRC-field does not appear in the WHERE clause</td>
<td>G20-2</td>
<td>Same as G19-2</td>
<td>(“city” = ‘C7’, “staffid” = ‘C7xy’, “name” = ‘C7pq’), (‘C8’, ‘C8xy’, ‘C8pq’), and (‘C9’, ‘C9xy’, ‘C9pq’)</td>
<td></td>
</tr>
<tr>
<td><strong>COUNT() - IRC occurs on the numeric field in the WHERE clause of the SQL statement and the IRC-field also appears in the SELECT clause</strong></td>
<td>G21-1</td>
<td>(1) Apply BVA rule on the condition in the WHERE clause to obtain BVA values in the records of the database, (2) Set other numeric fields of the SELECT clause for those records mentioned in (1) so that values are distinct and they are either far greater or less than the boundary value.</td>
<td>(“grade” = 39, “salary” = 3), (40, 5), (41, 8)</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
<td>Example</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>Other aggregate function - IRC occurs on the numeric field in the WHERE clause of the SQL statement and the IRC-field also appears in the SELECT clause</td>
<td>G21-2</td>
<td>Prepare two datasets as follows.</td>
<td>First dataset: (“grade” = 39, “salary” = 85), (40, 20), and (41, 15); Second dataset: (“grade” = 15, “salary” = 40) and (55, 40)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Dataset1.</strong> (1) Apply BVA rule on WHERE condition to obtain and set the BVA values in the records of the dataset. (2) Apply 3 rules for other numeric fields for three records mentioned in (1) above so that (i) the record whose value of IRC-field is less than the boundary value, other field has (a) the maximum value among other records in the same field, (b) the value at least twice as big as the boundary value. (ii) the record whose value of the IRC-field is greater than than the boundary value, the other field has the minimum value among other records in the same field and (iii) the record whose value of IRC-field is equal to boundary value, other field has value greater than the value mentioned in (ii) above and smaller than boundary value. <strong>Dataset2.</strong> (1) Set values in IRC-field so that they are all distinct and different from boundary value and (2) Set values for other fields in records mentioned in (1) above so that values are equal to boundary value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
<td>Example</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------</td>
<td>----------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>22</td>
<td>COUNT() - IRC occurs on the numeric field in the <code>WHERE</code> clause of the SQL statement and the IRC-field also does not appear in the <code>SELECT</code> clause</td>
<td>G22-1</td>
<td>Same as G21-1</td>
<td>(“age” = 29, “grade” = 3, “salary” = 5), (30, 6, 8), (31, 10, 12)</td>
</tr>
<tr>
<td>23</td>
<td>COUNT() - IRC occurs on the non-numeric field in the <code>WHERE</code> clause and the IRC-field also appears in the <code>SELECT</code> clause.</td>
<td>G23-1</td>
<td>(1) Apply BVA rule on the condition in the <code>WHERE</code> clause to obtain BVA values in the records of the database, (2) Set other non-numeric fields of the <code>SELECT</code> clause for those records mentioned in (1) so that values are distinct and they are either far greater or less than the boundary value.</td>
<td>(staffid = ‘S1’, name=’Adam’), (‘S2’, ‘David’), and (‘S3’, ‘Jack’)</td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
<td>Example</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| Other aggregate function - IRC occurs on the non-numeric field in the `WHERE` clause and the `IRC-field` also appears in the `SELECT` clause. | G23-2 | Prepare two datasets as follows. | **Dataset1.** (1) Apply BVA rule on the `WHERE` condition to obtain and set the BVA values in the records of the dataset. 
(2) Apply 3 rules for other numeric fields for three records mentioned in (1) above so that (i) the record whose value of the `IRC-field` is less than the boundary value, other field has the maximum value among other records in the same field, (ii) the record whose value of the `IRC-field` is greater than than the boundary value, the other field has the minimum value among other records in the same field, and (iii) the record whose value of the `IRC-field` is equal to boundary value, other field has value greater than the value mentioned in (ii) above and smaller than boundary value. **Dataset2.** (1) Set values in the `IRC-field` so that they are all distinct and different from boundary values and (2) Set values for other fields in records mentioned in (1) above so that values are equal to the boundary value. |
<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>COUNT()-IRC occurs on the non-numeric field in the WHERE clause and the IRC-field does not appear in the SELECT clause.</td>
</tr>
<tr>
<td></td>
<td>G24-1</td>
</tr>
<tr>
<td></td>
<td>G24-2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4 Experimental Study

4.4.1 Subject SQL Statements

In this experiment, I use those SQL statements found in NIST [79]. Some of them also appeared in the subject SQL statements used in the experiment of Chapter 3 in Section 3.5.1. Unlike the ABS mutation operator, for this experiment both numeric and non-numeric fields may exist in the SQL statements. Among these NIST SQL statements, 41 of them involve simple SQL SELECT statements. And these 41 statements are used as the original SQL statements for this experiment.
CREATE TABLE WORKS {
  EMPNUM VARCHAR,
  PNUM VARCHAR,
  HOUR INT,
  UNIQUE KEY(EMPNUM, PNUM)
}

Figure 4.1: Table Schema of WORKS

### 4.4.2 Mutant Generation

Similar to the experiment on the *ABS* mutation operator, I generated *IRC* mutants for each subject SQL statement with the help of SQLMutation tool [29]. This time, I do not discover any extra mutants as compared to the previous *ABS* experiment. For some SQL statements, more than one types of mutant are generated from the same SQL statement depending on the actual *IRC* mutation operator occurred in the *SELECT* or in the *WHERE* clause. As an example, for the table schema in Figure 4.1, a SQL statement is “*SELECT* empnum, hours *FROM* WORKS *WHERE* pnum = ‘P2’”. When an *IRC* mutation operator replaces “empnum” in the *SELECT* clause by “pnum” a mutant of the type IRC-m10 is generated. Again, for the same SQL statement, when an *IRC* mutation operator occurs in the *WHERE* clause to replace “pnum” by “empnum”, a mutant of type IRC-m14 is generated. Thus, multiple types of mutants are possible from the same SQL statement. In this experiment, I generated a total of 249 mutants. Table 4.11 shows the distribution of the mutants for the various types of the *IRC* mutation operator. Among these 249 mutants, 23 mutants are equivalent to their original SQL statements. Therefore, there are 226 non-equivalent mutants for this experiment. It also shows that for a number of types of mutant there is no mutant, this happens due to unavailability of appropriate original SQL statements.

### 4.4.3 Test Dataset Generation and Data Collection

In order to generate test dataset, I applied a similar procedure as described in the Section 3.5.2. The only difference is I used the test case generation guidelines as described in Section 4.3. I then executed the corresponding test dataset against each type of *IRC*
mutants. Next, I checked whether the test dataset can kill those IRC mutants. I counted the number of killed mutants and finally calculated the mutation score with the number of killed mutants as a percentage of total number of non-equivalent mutants.

4.4.4 Results Analysis

Table 4.12 shows the number of killed mutants against their corresponding number of non-equivalent mutants. It also illustrates the mutation score for different types of the IRC mutants. As a result, from the Table 4.12, it is clear that for all available types of IRC mutants, the generated test datasets kill 100% of non-equivalent mutants.
Table 4.12: Number of Killed and Alive Mutants for IRC (only shows the available mutants)

<table>
<thead>
<tr>
<th>Type</th>
<th># of Mutants</th>
<th>Non-Equivalent</th>
<th>Killed</th>
<th>Alive</th>
<th>Score(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRC-m1</td>
<td>6</td>
<td>6</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m5</td>
<td>59</td>
<td>59</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m6</td>
<td>27</td>
<td>27</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m7</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m9</td>
<td>13</td>
<td>13</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m10</td>
<td>32</td>
<td>32</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m11</td>
<td>17</td>
<td>17</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m12</td>
<td>12</td>
<td>12</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m13</td>
<td>11</td>
<td>11</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m14</td>
<td>27</td>
<td>27</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m17</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m22</td>
<td>17</td>
<td>17</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRC-m24</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

226 226 - 100

4.5 Threads and Validity

The external validity of this experiment is the selection of the SQL statements. I have selected 41 original SQL statements from the NIST. Although there is a large number of SQL queries in NIST, not all the queries have the same features that I have proposed. For example, in this experiment, SQL statements with join, inner join, outer join, sub-query, and query with ‘group by’ keyword are avoided. As a result, there are only 41 original SQL statements. These 41 SQL statements falls into 13 of the 24 possible types of mutants. Because of the small number of original SQL statements, the number of the IRC mutants is also small. Since the results of this experiment enable me, to generalize them across different database applications, it would be useful to select more SQL queries so that each scenario can incorporate some mutants.

The internal validity of this experiment is the selection of a test dataset. According to my guidelines, in some cases, test dataset needs multiple number of same value in the same dataset. In some other cases, it needs to set null values in the test dataset. Due to database constraints, if the values cannot be set in the test dataset, a number of mutants
may not be distinguished by my guidelines. As a result the mutation score may decrease which may lead to weakness in my proposed guidelines.

The *construct validity* of this experiment is that some mutants may be missed by the SQLMutation tool for my proposed guidelines for the *IRC* mutation operator. For a wide range of original SQL statements if the number of missed mutants generated by the tool is unknown, the result may not be exact.

### 4.6 Summary

- The importance of the *IRC* mutant in SQL statements is explained.

- Twenty four different types of mutants with altogether 44 variants are identified from four major forms of original SQL statements.

- For each of these variants, one guideline has been proposed to generate one test dataset. But in some cases for different complex variant, proper guidelines are provided to generate two or more datasets.

- An experiment has been performed to verify the effectiveness of the guidelines. In this experiment all non-equivalent mutants are killed.
Chapter 5

Test Case Generation for IRT Operator

In this chapter, I investigate how a identifier constant replacement (IRT) mutation operator can be applied on SELECT SQL statements to create various types of IRT mutants. Similar to IRC mutation operator, the IRT mutation operator can also be applied on both numeric and non-numeric fields. There are altogether 16 different types of IRT mutants with 32 different variants. For each of these variants, I develop a test case generation guideline to generate test cases that can kill the corresponding mutants. Finally, I evaluate the effectiveness of the guidelines via an experiment using NIST database conformance test suite [79].

Similar to chapter 3, for ease of discussion and illustration, I use the example of the database schema STAFF given in Figure 3.1 throughout the rest of this chapter.

5.1 Identifier Constant Replacement (IRT) Operator

While developing database applications, programmers may make mistake in writing SQL query. When they incorrectly use a field or a constant different from the original intended constant, an IRT mutant is created. As a result, this causes the application to return
incorrect results. For example, given a database schema of the STAFF database table in Figure 3.1, if a developer needs to retrieve the details (e.g. name, salary, and grade) of those staff whose salary is greater than $3000 from the STAFF table, a possible SQL statement is “SELECT name, grade, salary FROM STAFF WHERE salary > 3000” (denoted as $S_1$ for ease of reference in this section). In case, developers make mistakes by writing “grade” instead of 3000, the output is wrong. Hence, it leads to incorrect results.

The definition of the identifier constant replacement mutation operator ($IRT$), denoted as $IRT$, was first proposed by Tuya and colleagues [1] as follows:

With the $IRT$ mutation operator, a constant is replaced by another constants, columns and parameters that are present in the query and are type compatible with the original constant.

For example, a possible $IRT$ mutant of $S_1$ can be achieved by replacing 3000 by “grade”. The mutant will then be “SELECT name, grade, salary FROM STAFF WHERE salary > grade”. But 3000 will not be replaced with “age” because it does not appear in the original SQL statement. Moreover, 3000 can not be replaced by “staffid” or “name” as they are not type compatible with 3000.

5.2 Sixteen Types of $IRT$ Mutants

In this section, I explore and explain how many different types of $IRT$ mutants could be generated based on simple SQL statements. Similar to the discussion of $ABS$ mutants in Chapter 3, simple SQL statements are those statements that may involve the use of aggregate functions in the SELECT clause and the use of WHERE clause with a simple condition. Depending on whether there are aggregate functions in SELECT clause or in a simple condition in the WHERE clause, 4 different forms of SQL statements can be used as the original SQLs to generate the mutants. Table 5.1 summarizes all these 4 forms of original SQL statements, namely O1, O2, O3, and O4.
Table 5.1: Possible Types of Original SQL Statements

<table>
<thead>
<tr>
<th>Type</th>
<th>simple WHERE clause</th>
<th>aggregate function in SELECT clause</th>
<th>Example</th>
<th>(identifier for ease of reference in this chapter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>No</td>
<td>No</td>
<td>SELECT grade, salary, 60, 50 FROM STAFF</td>
<td>(O1-1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT staffid, name, ‘S6’, ‘S10’ FROM STAFF</td>
<td>(O1-2)</td>
</tr>
<tr>
<td>O2</td>
<td>No</td>
<td>Yes</td>
<td>SELECT MAX(grade), MIN(salary), MAX(60), MIN(50) FROM STAFF</td>
<td>(O2-1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT MAX(staffid), MIN(name), MAX(‘S6’), MIN(‘S10’) FROM STAFF</td>
<td>(O2-2)</td>
</tr>
<tr>
<td>O3</td>
<td>Yes</td>
<td>No</td>
<td>SELECT grade, salary, 60 FROM STAFF WHERE age &gt; 30</td>
<td>(O3-1a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT grade, salary, 60, 50 FROM STAFF WHERE staffid = ‘S1’</td>
<td>(O3-1b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT staffid, name, ‘S6’, ‘S10’ FROM STAFF WHERE grade &gt; 40</td>
<td>(O3-2a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT staffid, name, ‘S6’ FROM STAFF WHERE city = ‘C8’</td>
<td>(O3-2b)</td>
</tr>
<tr>
<td>O4</td>
<td>Yes</td>
<td>Yes</td>
<td>SELECT MAX(grade), MIN(salary), MAX(60) FROM STAFF WHERE age &gt; 30</td>
<td>(O4-1a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT MAX(grade), MIN(salary), MAX(60), MIN(50) FROM STAFF WHERE staffid = ‘S1’</td>
<td>(O4-1b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT MAX(staffid), MIN(name), MAX(‘S6’), MIN(‘S10’) FROM STAFF WHERE age &gt; 30</td>
<td>(O4-2a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT MAX(staffid), MIN(name), MAX(‘S6’) FROM STAFF WHERE city = ‘C8’</td>
<td>(O4-2b)</td>
</tr>
</tbody>
</table>

Table 5.1 shows that for O1 and O2 there are some constants (e.g. 60, 50 or ‘S6’, ‘S10’) in the SELECT clause. As I am discussing IRT mutants, without any constant in the SELECT clause, no IRT mutant is possible. Even though the original SQL statements are very simple, there are altogether 16 types of IRT mutants depending on whether the fields are numeric or non-numeric. Table 5.2 summarizes all these 16 types of IRT mutants, namely IRT-m1, IRT-m2, ..., IRT-m16. Table 5.3 gives two examples of each type of these mutants with reference to different original SQL statements given in Table 5.1. First example illustrates the replacement of constant by another field in SQL statement whereas the second example shows the replacement of constant by another constant. In the rest of the section, I discuss how I classify such 16 types of IRT mutants in details.

5.2.1 Original SQL without Aggregate Function and WHERE clause

I first discuss those IRT mutants that are generated from the original SQL statements without any aggregate function and a WHERE clause. Therefore, the IRT mutation operator
Table 5.2: Different Types of IRT Mutant

<table>
<thead>
<tr>
<th>Type</th>
<th>Origin</th>
<th>IRT occurs in\textsuperscript{a}</th>
<th>IRT-constant\textsuperscript{b}</th>
<th>Field in WHERE clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRT-m1</td>
<td>O1</td>
<td>S</td>
<td>Numeric</td>
<td>n/a\textsuperscript{c}</td>
</tr>
<tr>
<td>IRT-m2</td>
<td>O1</td>
<td>S</td>
<td>Non-numeric</td>
<td>n/a</td>
</tr>
<tr>
<td>IRT-m3</td>
<td>O2</td>
<td>A</td>
<td>Numeric</td>
<td>n/a</td>
</tr>
<tr>
<td>IRT-m4</td>
<td>O2</td>
<td>A</td>
<td>Non-numeric</td>
<td>n/a</td>
</tr>
<tr>
<td>IRT-m5</td>
<td>O3</td>
<td>S</td>
<td>Numeric</td>
<td>Numeric</td>
</tr>
<tr>
<td>IRT-m6</td>
<td>O3</td>
<td>S</td>
<td>Numeric</td>
<td>Non-numeric</td>
</tr>
<tr>
<td>IRT-m7</td>
<td>O3</td>
<td>S</td>
<td>Non-numeric</td>
<td>Numeric</td>
</tr>
<tr>
<td>IRT-m8</td>
<td>O3</td>
<td>S</td>
<td>Non-numeric</td>
<td>Non-numeric</td>
</tr>
<tr>
<td>IRT-m9</td>
<td>O3</td>
<td>W</td>
<td>Numeric</td>
<td>Numeric</td>
</tr>
<tr>
<td>IRT-m10</td>
<td>O3</td>
<td>W</td>
<td>Non-numeric</td>
<td>Non-numeric</td>
</tr>
<tr>
<td>IRT-m11</td>
<td>O4</td>
<td>A</td>
<td>Numeric</td>
<td>Numeric</td>
</tr>
<tr>
<td>IRT-m12</td>
<td>O4</td>
<td>A</td>
<td>Numeric</td>
<td>Non-numeric</td>
</tr>
<tr>
<td>IRT-m13</td>
<td>O4</td>
<td>A</td>
<td>Non-numeric</td>
<td>Numeric</td>
</tr>
<tr>
<td>IRT-m14</td>
<td>O4</td>
<td>A</td>
<td>Non-numeric</td>
<td>Non-numeric</td>
</tr>
<tr>
<td>IRT-m15</td>
<td>O4</td>
<td>W</td>
<td>Numeric</td>
<td>Numeric</td>
</tr>
<tr>
<td>IRT-m16</td>
<td>O4</td>
<td>W</td>
<td>Non-Numeric</td>
<td>Non-Numeric</td>
</tr>
</tbody>
</table>

\textsuperscript{a}S: SELECT clause, A: AGGREGATE function in SELECT clause, W: WHERE clause
\textsuperscript{b}IRT-constant is the constant that is changed by the IRT mutation operator
\textsuperscript{c}Both O1 and O2 do not have WHERE clause

can only occur in those constants in the SELECT clause. There are two types of IRT mutants in this group. They are

IRT-m1 This type of mutant occurs when an IRT mutation operator replaces a numeric constant by another numeric field or constant that appears in the original SQL statement. For example, for O1-1 in Table 5.1, when the IRT mutation operator replaces a constant 60 with another numeric field “grade”, the resulting IRT mutant becomes \texttt{SELECT grade, salary, grade, 50 FROM STAFF"}. Again for O1-1 in Table 5.1, when the IRT mutation operator replaces a constant 60 with another constant 50, the resulting IRT mutant becomes \texttt{SELECT grade, salary, 50, 50 FROM STAFF"}.

IRT-m2 Similar to IRT-m1, this type of mutant occurs when an IRT mutation operator replaces a non-numeric (string) constant by another non-numeric field or non-numeric constant that appears in the original SQL statement. For example, for
### Table 5.3: Examples of Various IRT Mutants

<table>
<thead>
<tr>
<th>Mutant</th>
<th>Example</th>
<th>cf. with original SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRT-m1</td>
<td>SELECT grade, salary, grade, 50 FROM STAFF</td>
<td>O1-1, 60→grade&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>IRT-m2</td>
<td>SELECT staffid, name, staffid, ‘S10’ FROM STAFF</td>
<td>O1-2, ‘S6’→staffid</td>
</tr>
<tr>
<td>IRT-m3</td>
<td>SELECT MAX(grade), MIN(salary), MAX(grade), MIN(50) FROM STAFF</td>
<td>O2-1, 60→grade</td>
</tr>
<tr>
<td>IRT-m4</td>
<td>SELECT MAX(staffid), MIN(name), MAX(name), MIN(‘S10’) FROM STAFF</td>
<td>O2-2, ‘S6’→name</td>
</tr>
<tr>
<td>IRT-m5</td>
<td>SELECT grade, salary, grade FROM STAFF WHERE age &gt; 30</td>
<td>O3-1a, 60→grade</td>
</tr>
<tr>
<td>IRT-m6</td>
<td>SELECT staffid, name, staffid, ‘S10’ FROM STAFF WHERE grade = ‘S1’</td>
<td>O3-1b, 60→grade</td>
</tr>
<tr>
<td>IRT-m7</td>
<td>SELECT staffid, name, staffid, ‘S10’ FROM STAFF WHERE grade &gt; 40</td>
<td>O3-2a, ‘S6’→staffid</td>
</tr>
<tr>
<td>IRT-m8</td>
<td>SELECT staffid, name, staffid FROM STAFF WHERE city = ‘C8’</td>
<td>O3-2b, ‘S6’→staffid</td>
</tr>
<tr>
<td>IRT-m9</td>
<td>SELECT staffid, name, ‘S10’, ‘S10’ FROM STAFF WHERE grade &gt; 40</td>
<td>O3-2a, ‘S6’→‘S10’</td>
</tr>
<tr>
<td>IRT-m10</td>
<td>SELECT staffid, name, ‘S6’ FROM STAFF WHERE staffid = ‘S6’</td>
<td>O3-2b, ‘C8’→‘S6’</td>
</tr>
<tr>
<td>IRT-m11</td>
<td>SELECT MAX(grade), MIN(salary), MAX(grade) FROM STAFF WHERE age &gt; 30</td>
<td>O4-1a, 60→grade</td>
</tr>
<tr>
<td>IRT-m12</td>
<td>SELECT MAX(grade), MIN(salary), MIN(30) FROM STAFF WHERE age &gt; 30</td>
<td>O4-1a, 60→30</td>
</tr>
<tr>
<td>IRT-m13</td>
<td>SELECT MAX(staffid), MIN(name), MAX(staffid), MIN(‘S10’) FROM STAFF WHERE age &gt; 30</td>
<td>O4-2a, ‘S6’→staffid</td>
</tr>
<tr>
<td>IRT-m14</td>
<td>SELECT MAX(staffid), MIN(name), MAX(staffid) FROM STAFF WHERE city = ‘C8’</td>
<td>O4-2b, ‘S6’→staffid</td>
</tr>
</tbody>
</table>

<sup>a</sup>This example mutant can be obtained from O1-1 in Table 5.1 by replacing 60 with another field “grade”.

<sup>b</sup>This example mutant can be obtained from O1-1 in Table 5.1 by replacing 60 with another constant 50.
5.2.2 Original SQL with Aggregate Function but without WHERE clause

In this section, I discuss those IRT mutants created from those original SQL statements that have aggregate function(s) (such as $\text{MAX}()$, $\text{MIN}()$, $\text{SUM}()$, $\text{AVG}()$, and $\text{COUNT}()$) in the SELECT clause but do not have any WHERE clause. Similar to the situation above, the IRT mutation operator can only occur in those constants involved in the aggregate function(s) in the SELECT clause. There are two types of IRT mutants in this group. They are:

IRT-m3 This type of mutant occurs when an IRT mutation operator replaces a numeric constant in the aggregate function of the SELECT clause by another numeric field or constant that also appears in the original SQL statement. For example, for O2-1 in Table 5.1, when the IRT mutation operator replaces 60 with “grade”, the resulting IRT mutant becomes “SELECT $\text{MAX}(\text{grade})$, $\text{MIN}($salary$)$, $\text{MAX}(60)$ FROM STAFF”. Again for the same original SQL statement when an

\footnote{Due to the syntax of SQL statement, once an aggregate function appears in the SELECT clause, all fields in the SELECT clause must be aggregate expressions. The only exception is when there is a GROUP BY clause. Since I do not consider GROUP BY clause in this thesis, the relevant discussion is out of the scope of this thesis.}
IRT mutation operator replaces 60 with 50 the IRT mutant becomes “SELECT MAX(grade), MIN(salary), MAX(50), MIN(50) FROM STAFF”.

IRT-m4 Same as IRT-m3 this type of mutant occurs when an IRT mutation operator replaces a non-numeric constant in the aggregate function of the SELECT clause by another non-numeric field or constant that also appears in the original SQL statement. For example, for O2-2 in Table 5.1, when the IRT mutation operator replaces ‘S6’ with “name”, the resulting IRT mutant becomes “SELECT MAX(staffid), MIN(name), MAX(name), MIN(S10) FROM STAFF”. For the same original SQL statement when an IRT mutation operator replaces ‘S6’ by ‘S10’, the resultant IRT mutant becomes “SELECT MAX(staffid), MIN(name), MAX(S10), MIN(S10) FROM STAFF”.

5.2.3 Original SQL with WHERE clause but no Aggregate Function

I now discuss those IRT mutants created from those original SQL statements having a simple WHERE clause but no aggregate functions in the SELECT clause. It is notable that the IRT mutation operator may occur in the SELECT clause or in the WHERE clause.

5.2.3.1 IRT in the SELECT clause

When an IRT mutation operator occurs in the SELECT clause, the following four types of mutants are possible.

IRT-m5 This type of mutant occurs when an IRT mutation operator replaces a numeric constant in the SELECT clause by another numeric field or constant that appears in the original SQL statement. For example, for O3-1a in Table 5.1, when the IRT mutation operator replaces 60 with “grade”, the resulting IRT mutant becomes “SELECT grade, salary, grade FROM STAFF WHERE age > 30”. Again for the same original SQL statement when the IRT mutation operator replaces ‘S6’ by ‘S10’ the resulting IRT mutant is “SELECT grade, salary, 30 FROM STAFF WHERE age > 30”
IRT-m6 This type of mutant occurs when an IRT mutation operator replaces a numeric constant in the SELECT clause by another numeric field or constant that also appears in the original SQL statement. Moreover, a non-numeric field is involved in the WHERE clause. For example, for O3-1b in Table 5.1, when the IRT mutation operator replaces 60 with “grade”, the resulting IRT mutant becomes “SELECT grade, salary, grade, 50 FROM STAFF WHERE staffid = ‘S1’ ”. Again for the same original SQL statement when the IRT mutation operator replaces 60 by 50, the resulting IRT mutant becomes “SELECT grade, salary, 50, 50 FROM STAFF WHERE staffid = ‘S1’ ”.

IRT-m7 This type of mutant occurs when an IRT mutation operator replaces a non-numeric (string) constant in the SELECT clause by another non-numeric field or constant that appears in the original SQL statement. Moreover, a numeric field is involved in the WHERE clause. For example, for O3-2a in Table 5.1, when the IRT mutation operator replaces ‘S6’ with “staffid” in the SELECT clause, the resulting IRT mutant becomes “SELECT staffid, name, staffid, ‘S10’ FROM STAFF WHERE grade > 40”. Again for the same original SQL statement when the IRT mutation operator replaces ‘S6’ by ‘S10’ the resulting IRT mutant becomes “SELECT staffid, name, ‘S10’, ‘S10’ FROM STAFF WHERE grade > 40”.

IRT-m8 This type of mutant occurs when an IRT mutation operator replaces a non-numeric constant in the SELECT clause by another non-numeric field or constant that appears in the original SQL statement. For example, for O3-2b in Table 5.1, when the IRT mutation operator replaces ‘S6’ with “staffid”, the resulting IRT mutant becomes “SELECT staffid, name, staffid FROM STAFF WHERE city = ‘C8’ ”. Again for the same original SQL statement, when the IRT mutation operator replaces ‘S6’ with ‘S10’, the resulting IRT mutant becomes “SELECT staffid, name, ‘S10’ FROM STAFF WHERE city = ‘C8’ ”.
Chapter 5. Test Case Generation for IRT Operator

5.2.3.2 IRT in the WHERE clause

When an IRT mutation operator occurs in the WHERE clause, the following two types of mutants are possible.

IRT-m9 This type of mutant occurs when an IRT mutation operator replaces a numeric constant in the WHERE clause by another numeric field or constant that appears in the SELECT clause of the original SQL statement. For example, for O3-1a in Table 5.1, when the IRT mutation operator replaces 30 with “salary” the resulting mutant becomes “SELECT grade, salary, 60 FROM STAFF WHERE age > salary”. Again for the same original SQL statement when the IRT mutation operator replaces 30 with 60, the resulting IRT mutant becomes “SELECT grade, salary, 60 FROM STAFF WHERE age > 60”.

IRT-m10 This type of mutant occurs when an IRT mutation operator replaces a non-numeric constant in the WHERE clause by another non-numeric field or constant that appears in the SELECT clause of the original SQL statement. For example, for O3-2b in Table 5.1, when the IRT mutation operator replaces ‘S1’ with “staffid” the resulting IRT mutant becomes “SELECT staffid, name, ‘S6’ FROM STAFF WHERE city = staffid”. Again for the same original SQL statement when the IRT mutation operator replaces ‘S1’ with ‘S6’, the resulting IRT mutant becomes “SELECT staffid, name, ‘S6’ FROM STAFF WHERE city = ‘S6’”.

5.2.4 Original SQL with WHERE clause and Aggregate Function

I now discuss the IRT mutants created from those original SQL statements having aggregate functions (such as MAX(), MIN(), SUM(), AVG() and COUNT()) in the SELECT clause and a simple WHERE clause. Similar to the case discussed in Section 5.2.3, the IRT mutation operator may occur in the aggregate function in the SELECT clause or in the field involved in the WHERE clause.
5.2.4.1 *IRT* in Aggregate Function in SELECT clause

When an *IRT* mutation operator occurs in a field involved in an aggregate function of the SELECT clause, the following four types of mutants are possible.

IRT-m11 This type of mutant occurs when an *IRT* mutation operator replaces a numeric constant in an aggregate function of the SELECT clause by another numeric field that also appears in the SQL statement. For example, for O4-1a in Table 5.1, when the *IRT* mutation operator replaces 60 with “grade”, the resulting *IRT* mutant becomes “SELECT MAX(grade), MIN(salary), MAX(grade) FROM STAFF WHERE age > 30". Again for the same original SQL statement when the *IRT* mutation operator replaces 60 with 30 the resulting *IRT* mutant becomes “SELECT MAX(grade), MIN(salary), MAX(30) FROM STAFF WHERE age > 30”.

IRT-m12 This type of mutant occurs when an *IRT* mutation operator replaces a numeric constant in an aggregate function of the SELECT clause by another numeric field or constant that also appears in the SQL statement. Moreover, a non-numeric field appears in the WHERE clause. For example, for O4-1c in Table 5.1, when the *IRT* mutation operator replaces 60 with “grade”, the resulting *IRT* mutant becomes “SELECT MAX(grade), MIN(salary), MAX(grade), MIN(50) FROM STAFF WHERE staffid = ‘S1’ ". Again for the same original SQL statement when the *IRT* mutation operator replaces 60 by 50 the resulting *IRT* mutant is “SELECT MAX(grade), MIN(salary), MAX(50), MIN(50) FROM STAFF WHERE staffid = ‘S1’ ".

IRT-m13 This type of mutant occurs when an *IRT* mutation operator replaces a non-numeric (string) constant in an aggregate function of the SELECT clause by another non-numeric field or constant that also appears in the SQL statement. Moreover, a numeric field is involved in the WHERE clause. For example, for O4-2a in Table 5.1, when the *IRT* mutation operator replaces ‘S6’ with “staffid”, the resulting *IRT* mutant becomes “SELECT MAX(staffid), MIN(name), MAX(staffid), MIN(‘S10’) FROM STAFF WHERE age > 30". Again for the same original SQL statement when the *IRT* mutation operator replaces ‘S6’ with ‘S10’, the resulting *IRT* mutant becomes “SELECT MAX(staffid), MIN(name), MAX(S10), MIN(‘S10’) FROM STAFF WHERE age > 30".
Chapter 5. Test Case Generation for IRT Operator

mutant becomes “SELECT MAX(stafﬁd), MIN(name), MAX(‘S10’), MIN(‘S10’) FROM STAFF WHERE age > 30”.

IRT-m14 This type of mutant occurs when an IRT mutation operator replaces a non-numeric constant in an aggregate function of the SELECT clause by another non-numeric ﬁeld or constant that also appears in the SQL statement. For example, for O4-2b in Table 5.1, when the IRT mutation operator replaces ‘S6’ with “staffid”, the resulting IRT mutant becomes “SELECT MAX(stafﬁd), MIN(name), MAX(stafﬁd) FROM STAFF WHERE city = ‘C8’ ”. Again for the same original SQL statement, when the IRT mutation operator replaces ‘S6’ with “staffid”, the resulting IRT mutant becomes “SELECT MAX(stafﬁd), MIN(name), MAX(‘C8’) FROM STAFF WHERE city = ‘C8’ ”.

5.2.4.2 IRT in the WHERE clause

When an IRT mutation operator occurs in the WHERE clause, the following two types of mutants are possible.

IRT-m15 This type of mutant occurs when an IRT mutation operator replaces a numeric constant in the WHERE clause by another numeric ﬁeld or constant that also appears in the original SQL statement. For example, for O4-1a in Table 5.1, when the IRT mutation operator replaces 30 by “salary”, the resulting IRT mutant becomes “SELECT MAX(grade), MIN(salary), MAX(60) FROM STAFF WHERE age > salary”. Again for the same original SQL statement, when the IRT mutation operator replaces 30 with 60 in the WHERE clause, the resulting IRT mutant becomes “SELECT MAX(grade), MIN(salary), MAX(60) FROM STAFF WHERE age > 60”.

IRT-m16 Similar to IRT-m15, this type of mutant occurs when an IRT mutation operator replaces a non-numeric constant in the WHERE clause by another non-numeric ﬁeld or constant that also appears in the original SQL statement. For example, for O4-2b in Table 5.1, when the IRT mutation operator replaces ‘C8’ with “staffid” in the WHERE clause, the resulting IRT mutant becomes “SELECT MAX(stafﬁd),
\[
\text{MIN(name), MAX('S6')} \text{ FROM STAFF WHERE city = 'staffid'}. \text{ Again for the same original SQL statement, when the IRT mutation operator replaces 'C8' with 'S6' in the WHERE clause, the resulting IRT mutant becomes ‘SELECT MAX(staffid), MIN(name), MAX('S6') FROM STAFF WHERE city = 'S6' ”.}
\]

5.3 Test Case Generation Guidelines

As discussed previously, there are altogether 16 types of IRT mutants as indicated and illustrated in Tables 5.2 to 5.3. In this section, I propose test case generation guidelines for detecting each type of IRT mutants. The guidelines for IRT-m1, . . . , IRT-m16 will be denoted as G-IRT-m1, . . . , G-IRT-m16, respectively. For ease of discussion and illustration, I will use the database table “STAFF” as given in Figure 3.1, those original SQL statements in Table 5.1 and the relevant mutants in Table 5.3 as examples throughout the rest of this section. Again, the discussion is subdivided into four subsections based on the original SQL statements.

5.3.1 Guidelines for Mutants from SQL without Aggregate Function and WHERE clause

There are two types of mutants, namely IRT-m1 and IRT-m2, as discussed in Section 5.2.1. The test case generation guidelines for each of these mutants are described as follows:

**G-IRT-m1:** For this type of IRT-m1 mutants, the IRT mutation operator replaces a numeric constant in the \textit{SELECT} clause of the SQL statement by any of the numeric fields or constants that appear in the SQL statement. The resultant mutant will have two identical fields or two identical constants in the \textit{SELECT} clause. In order to distinguish the mutants from the original SQL statement, the guideline is to set the values in the database table so that the values of those numeric fields are distinct and they are not equal to the constants involved in the SQL statement.

For example, for the SQL statement O1-1 in the Table 5.1, when an IRT mutation operator replaces 60 in the \textit{SELECT} clause by another numeric field “grade”, the
resultant mutant is “SELECT grade, salary, grade, 50 FROM STAFF”. In order to
distinguish this type of mutant from the original SQL statement, one needs to set
some distinct values for the “grade” and the “salary” fields. An example may be
(“grade” = 5, “salary” = 1500), (8, 3000), and (10, 4000). By doing so, one can
then distinguish the mutant from O1-1 because O1-1 returns “(5, 1500, 60, 50), (8,
3000, 60, 50), (10, 4000, 60, 50)” whereas the mutant returns “(5, 1500, 5, 50), (8,
3000, 8, 50), (10, 4000, 10, 50)”.

In the same way for the original SQL statement O1-1 the in Table 5.1, when 60
is replaced by another constant 50, the resultant mutant is “SELECT grade, salary,
50, 50 FROM STAFF”. The above dataset can also distinguish the mutant from O1-1
because O1-1 returns “(5, 1500, 60, 50), (8, 3000, 60, 50), (10, 4000, 60, 50)” whereas
the mutant returns “(5, 1500, 50, 50), (8, 3000, 50, 50), (10, 4000, 50, 50)”.

**G-IRT-m2:** Similar to IRT-m1, for this type of IRT-m2 mutant, the IRT mutation op-
erator replaces a non-numeric (string) constant in the SELECT clause of the SQL
statement by any of the non-numeric fields or constants that appear in the SQL
statement. The resultant mutant will have two identical non-numeric fields or two
identical constant strings in the SELECT clause. In order to distinguish the mutants
from the original SQL statement the guideline is to set the values in the database
table so that the values of those non-numeric fields are distinct and they are not
equal to the constants involved in the SQL statement.

For example, for the SQL statement O1-2 in Table 5.1, when an IRT mutation
operator replaces ‘S6’ in the SELECT clause by another non-numeric field “staffid”,
the resultant mutant is “SELECT staffid, name, staffid, ‘S10’ FROM STAFF”. In order
to distinguish this type of mutant from the original SQL statement, I need to set some
distinct values for the “staffid” and the “name” fields so that ‘S6’ and ‘S10’ does not
exist in the selected values. An example may be (“staffid”={‘S1’, “name”=‘Jake’},
(‘S2’, ‘David’), (‘S3’, ‘Michael’)). By doing so, one can then distinguish O1-2 from the
(‘S3’, ‘Micheal’, ‘S6’, ‘S10’)” whereas the mutant returns “(‘S1’, ‘Jake’, ‘S1’, ‘S10’),

5.3.2 Guidelines for Mutants from SQL with Aggregate Function but without WHERE clause

There are two types of mutants in this group namely IRT-m3 and IRT-m4. The test case generation guidelines for each of these mutants are discussed below.

G-IRT-m3: For the IRT-m3 mutants, the IRT mutation operator replaces a numeric constant in the aggregate function of the SELECT clause by another numeric field or constant that also appears in the SQL statement. Because of the involvement of numeric fields and constants, any aggregate functions can be applied on these numeric fields. Two different sets of guidelines are proposed depending on the characteristic of the aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function COUNT() and in the second set, the guidelines are about other aggregate functions.

1. (a) Situation when the IRT-constant is in the aggregate function COUNT(): Since COUNT() returns the number of non-null values in the required field, when the fields involved in the IRT mutation operator do not allow null values, there is no way to distinguish the original SQL statement from its mutants. In other words, the mutants are equivalent to the original SQL statement. However, when the fields involved allow null values, one can distinguish the original SQL statement from its mutants by setting the values in the database table so that they satisfy the following conditions:
(i) set the values in the numeric fields so that different numeric fields have different number of null values.

(ii) set at least one null value in each numeric field and

(iii) for each record, it can only have at most one null value in its numeric fields.

For example, for the SQL statement "SELECT COUNT(grade), COUNT(salary), COUNT(60), COUNT(50) FROM STAFF", when the numeric constant 60 is replaced by "grade", the resultant mutant is "SELECT COUNT(grade), COUNT(salary), COUNT(grade), COUNT(50) FROM STAFF". In order to distinguish this type of mutant from the original SQL statement, I need to set different number of null values for the "grade" and the "salary" fields in the records of the database table. An example of such dataset is ("grade" = 3, "salary" = 3), (4, 5), (5, null), (null, 7), (null, 8). By doing so, one can then distinguish the results of the original SQL statement from those mutants because the original SQL statement returns "3, 4, 5, 5" whereas the mutant returns "3, 4, 3, 5".

As \texttt{COUNT()} returns number of non-null values, for this type of SQL statement, when an IRT mutation operator replaces a numeric constant by another constant, the mutant is always equivalent to the original SQL statement, hence it cannot be distinguished.

(b) For the case of \texttt{COUNT(DISTINCT \ldots)}, the test case generation guideline is different from the case of \texttt{COUNT()}. First, please note that when all the values of selected fields are distinct, the behavior of \texttt{COUNT(\ldots)} is the same as that of \texttt{COUNT(DISTINCT \ldots)}. When the fields involved allow null values, one can distinguish the mutants by setting different number of null values to the fields involved in the SQL statement. Second, unlike the situation of \texttt{COUNT()}, if the fields cannot be null, one can still distinguish the mutants with \texttt{COUNT(DISTINCT \ldots)} from its original SQL statement by setting the values in the numeric fields so that these fields have different number of distinct values.
For example, for the SQL statement “SELECT COUNT(grade), COUNT(salary), COUNT(DISTINCT 60), COUNT(50) FROM STAFF”, when the numeric constant 60 is replaced by “grade”, the resultant mutant is "SELECT COUNT(grade), COUNT(salary), COUNT(DISTINCT grade), COUNT(50) FROM STAFF". In order to distinguish this type of mutant from the original SQL statement, I need to set the values in the fields of the “grade” and the “salary” fields so that they have different number of distinct values. An example may be (“grade” = 1, “salary” = 3), (2, 3), (3, 2). By doing so, one can then distinguish COUNT(DISTINCT grade) from COUNT(DISTINCT 60) because COUNT(DISTINCT 60) returns 1 whereas COUNT(DISTINCT grade) returns 3.

Similar to COUNT(), in a SQL statement involved in COUNT(DISTINCT ...), when an IRT mutation operator replaces a numeric constant by another constant, the mutant is always equivalent to the original SQL statement, hence it cannot be distinguished.

2. Situation when the IRT field is in an aggregate function other than COUNT() (i.e. MAX(), MIN(), SUM(), AVG()). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an IRT mutation operator replaces a numeric constant $c_1$ in the aggregate function \( AGGR() \) (may be MAX(), MIN(), ...) by another numeric field $f_1$ or an other constant $c_2$, the IRT mutant will have \( AGGR(f_1) \) or \( AGGR(c_2) \) instead of \( AGGR(c_1) \). In order to distinguish \( AGGR(f_1) \) or \( AGGR(c_2) \) from \( AGGR(c_1) \), the guideline is to set some non-zero distinct values for every numeric field in the database records so that they are far greater or less than those constants (e.g. $c_1$ and $c_2$) already existed in the SQL statement.

For example, for the SQL statement O2-1 in Table 5.1, when the IRT mutation operator replaces 60 in the aggregate function of the SELECT clause by “grade” the resultant mutant is “SELECT MAX(grade), MIN(salary), MAX(grade), MIN(50) FROM STAFF”. In order to distinguish this type of mutant from the original SQL statement, I need to set some non-zero values for the “grade” and the “salary” fields so that they are far greater than 60. An example may be
By doing so, one can then distinguish the mutant from O2-1 because O2-1 returns “200, 1100, 60, 50” whereas the mutant returns “200, 1100, 200, 50”.

It does not matter whether there is a `DISTINCT’ keyword in the SQL statement, the same guideline is adequate to distinguish the mutant from the original SQL statement.

G-IRT-m4: For the IRT-m4 mutants, the IRT mutation operator replaces a non-numeric constant in the aggregate function of the SELECT clause by another non-numeric field or constant string that also appears in the SQL statement. As this type of IRT mutants involve non-numeric fields and constants; MAX(), MIN() and COUNT() are the only legitimate aggregate functions. Similar to G-IRT-m3, two different sets of guidelines are proposed depending on the characteristic of the aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function COUNT() and in the second set, the guidelines are about other aggregate functions. The test case generation guidelines for these two scenarios are as follows:

1. (a) Situation when the IRT-constant is in the aggregate function COUNT(). Similar to G-IRT-m3-1a, since COUNT() returns the number of non-null values in the required field, when the fields involved in the IRT mutation operator do not allow null values, the mutants will be equivalent to the original SQL statement. However, when the fields involved allow null values, one can distinguish the original SQL statement from its mutant by setting the values in the database table so that they satisfy the following conditions:
   (i) set the values in the non-numeric fields so that different non-numeric fields have different number of null values
   (ii) set at least one null value in each non-numeric field and
   (iii) for each record, it can only have at most one null value in its non-numeric fields.

For example, for the SQL statement “SELECT COUNT(staffid), COUNT(name), COUNT(‘S6’), COUNT(‘S10’) FROM STAFF”, when the IRT mutation operator replaces ‘S6’ in the aggregate function of the SELECT clause by
“name” the resultant mutant is “SELECT COUNT(staffid), COUNT(name), COUNT(name), COUNT(‘S10’) FROM STAFF”. In order to distinguish this type of mutant from its original SQL statement, I need to set different number of null values for the “staffid” and the “name” fields in the records of database table. An example may be (“staffid”=‘S1’, “name”=‘Jack’), (null, ‘David’), (‘S2’, null), (‘S3’, null). By doing so, one can then distinguish the mutants from its original because the original SQL statement returns “1, 2, 4, 4” whereas the mutant returns “1, 2, 2, 4”.

As COUNT() returns number of not null values, for this type of SQL statement, when an IRT mutation operator replaces a constant string by another constant string, the mutant is always equivalent to the original SQL statement, hence it cannot be distinguished.

(b) For the case of COUNT(DISTINCT ...), the test case generation guidelines are different from the case of COUNT(). First, please note that when all the values of selected fields are distinct, the behaviour of COUNT(...) is the same as COUNT(DISTINCT ...). When the fields involved allow null values, one can distinguish the mutants by setting different number of null values to the fields involve in the SQL statement. Second, unlike the situation in COUNT(), if the fields cannot be null, one can still distinguish the mutants with COUNT(DISTINCT ...) from its original SQL statement by setting the values in the numeric fields so that these fields have different number of distinct values.

For example, for the SQL statement, “SELECT COUNT(staffid), COUNT(name), COUNT(DISTINCT ‘S6’), COUNT(‘S10’) FROM STAFF”, when the IRT mutation operator replaces ‘S6’ by “name”, the resultant IRT mutant is “SELECT COUNT(staffid), COUNT(name), COUNT(DISTINCT name), COUNT(‘S10’) FROM STAFF”. In order to distinguish this type of mutant from the original SQL statement, I need to set the values for the “staffid” and the “name” fields so that they have different number of distinct values. An example may be (“staffid” = ‘S1’, “name” = ‘Jack’), (‘S2’, ‘Jack’), (‘S3’, ‘Adam’). By doing so, one can then distinguish COUNT(DISTINCT ‘S6’) from COUNT(DISTINCT...
name), because \texttt{COUNT(DISTINCT ‘S6’)} returns 1 whereas \texttt{COUNT(DISTINCT name)} returns 2.

Similar to \texttt{COUNT()}, in a SQL statement involves in \texttt{COUNT(DISTINCT ...)}, when an \textit{IRT} mutation operator replaces a constant string by another constant string, the mutant is always equivalent to the original SQL statement, hence it cannot be distinguished.

2. Situation when the IRT-constant is in an aggregate function other than \texttt{COUNT()} (i.e. \texttt{MAX()} and \texttt{MIN()}). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an \textit{IRT} mutation operator replaces a constant string \texttt{s1} by another non-numeric field \texttt{f1} or constant string \texttt{s2}, the \textit{IRT} mutant will have \texttt{AGGR(f1)} or \texttt{AGGR(s2)} instead of \texttt{AGGR(s1)}. In order to distinguish \texttt{AGGR(f1)} or \texttt{AGGR(s2)} from \texttt{AGGR(s1)}, the guideline is to set some distinct \textit{non-null} values for every non-numeric field in the database records so that they are far greater or less than those constant strings (e.g. \texttt{s1} and \texttt{s2}) already existed in the SQL statement.

For example, for the SQL statement \texttt{“SELECT MAX(staffid), MIN(name), \texttt{MAX(‘S6’)}, MIN(‘S10’) FROM STAFF”}, when \texttt{‘S6’} is replaced by \texttt{“name”}, the resultant mutant is \texttt{“SELECT MAX(staffid), MIN(name), \texttt{MAX(name)}, MIN(‘S10’) FROM STAFF”}. In order to distinguish this type of mutant from the original SQL statement, I need to set some distinct \textit{non-null} values for the \texttt{“staffid”} and the \texttt{“name”} fields so that the values are far less than \texttt{‘S6’} and \texttt{‘S10’}. An example may be (\texttt{“staffid”}=`S2`, \texttt{“name”}=`Adam’), (‘S5’, ‘James’), (‘S10’, ‘David’). By doing so, one can then distinguish the mutant from its original SQL statement because the original SQL statement returns “S10, Adam, S6, S10” whereas the mutant returns “S10’, Adam, James, S10”.

In cases where the \textit{IRT} mutation operator replaces a constant string by another constant string, the results returned by the mutants are always different.

Again, it does not matter whether there is a \texttt{DISTINCT} keyword in the SQL statement, the same guideline is adequate to distinguish the mutant from the original SQL statement.
5.3.3 Guidelines for Mutants from SQL without Aggregate Function but with \texttt{WHERE} clause

This section discusses the test case generation guidelines for those IRT mutants created from original SQL statements that do not have any aggregate functions in the SQL statement but have a simple \texttt{WHERE} clause. An IRT mutation operator may replace a constant in the \texttt{SELECT} clause or in the \texttt{WHERE} clause to generate the mutants. Depending on the position of the IRT mutation operator occurred, there are two major categories of mutants. In the following the test case generation guidelines are described for these two categories.

5.3.3.1 \textit{IRT} in the \texttt{SELECT} clause

When an IRT mutation operator replaces a constant in the \texttt{SELECT} clause by another field or constant that appears in the original SQL statement, four types of mutants, namely IRT-m5 to IRT-m8, are generated depending on whether a numeric or a non-numeric field is involved in the \texttt{SELECT} clause or the \texttt{WHERE} clause. In the following, I discuss the test case generation guidelines for these six types of mutants.

G-IRT-m5: For the IRT-m5 mutants, the IRT mutation operator replaces a numeric constant in the \texttt{SELECT} clause of the SQL statement by any of the numeric field or constant that appears in the SQL statement. For example, for the SQL statement O3-1a in Table 5.1, when an IRT mutation operator replaces the constant 60 in the \texttt{SELECT} clause by “grade”, the resulting mutant is “\texttt{SELECT grade, salary, grade FROM STAFF WHERE age} > 30”. I now propose the test case generation guideline that could distinguish this type of mutants from the original SQL statement, using the above example to illustrate the idea. The guideline consists of the following three major steps:

(i) Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values\(^2\), and then set the records in the database table so

\(^2\)These values are the boundary value (the one on the boundary), “one-below” the boundary value, and “one-above” the boundary value.
that each record will have one of these values in the field involved in the **WHERE** clause\(^3\). Note that these boundary values should not be equal to the constants that appear in the SQL statement.

(ii) Apply the *reverse*-BVA rule to set the values of other numeric fields for those records mentioned in (i) above in such a way that

(a)  
   i. the values are greater than the boundary value when the value of the field involved in the **WHERE** clause is less than the boundary value,
   ii. the values are less than the boundary value when the value of the field involved in the **WHERE** clause is greater than boundary value, and
   iii. the values are not equal to the boundary value when the value of the field involved in the **WHERE** clause is equal to the boundary value.

(b) the values used in (a) above must not be equal to constant that appears in the SQL statement.

For example, to distinguish the mutant from O3-1a, I need to apply BVA on the condition “\textit{age} > 30” in the **WHERE** clause so as to set the records in the database table such that, each record will have the values 29, 30, and 31 to the “\textit{age}” field in various records in the database table. For these three records, I then apply the *reverse*-BVA rule on the “\textit{grade}” and the “\textit{salary}” fields to set the records like (\textit{age} = 29, \textit{grade} = 42, \textit{salary} = 82), (30, 25, 35), and (31, 28, 18). Table 5.4 illustrates the use of the proposed guidelines in this example. By doing so, one can then distinguish O3-1a from the mutant because O3-1a returns “28, 18, 60” whereas the mutant returns “28, 18, 28”.

When the **IRT** mutation operator replaces a constant in the **SELECT** clause with another constant, the same guideline is adequate to distinguish the mutant from the

\(^3\)These BVA values can also be used for other five relational operators like \textless, \textgreater, =, \textless\textless and \textneq.
original. But, in some cases where same constant is involved in the **SELECT** clause and in the **WHERE** clause, SQL statements containing equal (\(=\)) relational operator will be equivalent. For example, for the SQL statement “**SELECT** grade, salary, 30 **FROM** STAFF **WHERE** age = 30”, when 30 in the **SELECT** clause is replaced by “age” the resultant mutant is “**SELECT** grade, salary, **age** **FROM** STAFF **WHERE** age = 30”. No dataset can distinguish this mutant because it is equivalent to the original SQL statement.

**G-IRT-m6:** For the IRT-m6 mutant, the *IRT* mutation operator replaces a numeric constant in the **SELECT** clause of the SQL statement by another numeric field or constant and a non-numeric field appears in the simple condition in the **WHERE** clause. In order to distinguish the mutants from the original SQL statement, the guideline consists of the following two steps:

(i) Apply boundary value analysis (BVA) on the condition in the **WHERE** clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the non-numeric field in the **WHERE** clause.

(ii) Set the values in the numeric fields in the **SELECT** clause for those records mentioned in (i) above so that these values are all distinct and are not equal to any of the constants involved in the SQL statement.

For example, for the SQL statement O3-1b in Table 5.1, when an *IRT* mutation operator replaces the numeric constant 60 in the **SELECT** clause of the SQL statement by another numeric field “grade”, the resultant mutant is “**SELECT** grade, salary, grade, 50 **FROM** STAFF **WHERE** staffid = ‘S1’ ”. In order to distinguish this type of mutant from O3-1b, I need to apply BVA on the condition “staffid = ‘S1’ ” in the **WHERE** clause so as to set three records in the database table such that, they will have ‘S0’, ‘S1’, and ‘S2’ in “staffid”. For these records, I then set the values for “grade” and “salary” so that they are distinct and are not equal to 60 or 50. An example

\[\text{Please refer to the relevant example in Table 5.3}\]

\[\text{This is similar to Step 1 of G-IRT-m5 except that the field in the condition is non-numeric. For non-numeric field, we can use its natural ordering to select the values that are “1 below the boundary” and “1 above the boundary.”}\]
may be ("staffid" = ‘S0’, “grade” = 5, “salary” = 1000), (‘S1’, 10, 2000), and (‘S2’, 15, 3000). By doing so, one can then distinguish the mutant from O3-1b because O3-1b returns “10, 2000, 60, 50” whereas the mutant returns “10, 2000, 10, 50”.

In the same way, for the original SQL statement O3-1b the in Table 5.1, when 60 is replaced by another constant 50, the resultant mutant is “SELECT grade, salary, 50, 50 FROM STAFF WHERE staffid = ‘S1’ ”. The above dataset can also distinguish the mutant from O3-1b because O3-1b returns “10, 2000, 60, 50” whereas the mutant returns “10, 2000, 50, 50”.

G-IRT-m7: For the IRT-m7 mutants, the IRT mutation operator replaces a non-numeric constant in the select clause of the SQL statement by another non-numeric field or a non-numeric constant that appears in the SQL statement and a numeric field appears in the simple condition in the WHERE clause. In order to distinguish the original SQL statement from those mutants, the test case generation guideline consists of the following two major steps:

(i) Apply BVA on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the numeric field in the WHERE clause.

(ii) Set the values in the non-numeric fields of the SELECT clause for those records mentioned in (i) above so that these values are all distinct and are not equal to any of the constants involved in the SQL statement.

For example, for the SQL statement O3-2a in Table 5.1, when an IRT mutation operator replaces the constant ‘S6’ in the SELECT clause by “staffid”, the resultant mutant is “SELECT staffid, name, staffid, ‘S10’ FROM STAFF WHERE grade > 40”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “grade> 40” in the WHERE clause so as to set records in the database table such that they have 39, 40, and 41 in the field “grade”. For these records, I then set the values for the “staffid” and the “name” field so that they are distinct and are not equal to ‘S6’ or ‘S10’ in the records of the database. An example may be ("grade" = 39, “staffid”="S1", “name” = ‘Jack’), (40, ‘S2’,

Please refer to Table 5.1 and Table 5.3
‘David’), (41, ‘S3’, ‘James’). By doing so, one can then distinguish the mutant from O3-2a because O3-2a returns “S3, James, S6, S10”, whereas the mutant returns “S3, James, S3, S10”.

In the same way, for the original SQL statement O3-2a the in Table 5.1, when ‘S6’ is replaced by another constant ‘S10’, the resultant mutant is “SELECT staffid, name, S10, ‘S10’ FROM STAFF WHERE grade > 40”. The above dataset can also distinguish the mutant from O3-2a because because O3-2a returns “S3, James, S6, S10”, whereas the mutant returns “S3, James, S10, S10”.

G-IRT-m8: For the IRT-m8 mutants, the IRT mutation operator replaces a non-numeric constant in the SELECT clause of the SQL statement by another non-numeric field or non-numeric constant that appears in the SQL statement and a non-numeric field is also involved in the simple condition in the WHERE clause. In order to distinguish the mutants from the original SQL statement, the guideline consists of the following two steps:

(i) Apply BVA on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the non-numeric field in the WHERE clause.

(ii) Set the values in the other non-numeric fields in the SELECT clause for those records mentioned in (i) above so that these values are all distinct and different from the corresponding BVA values. Moreover, these values must not be equal to any constants include the constants already existed in the SQL statement.

For example, for SQL statement O3-2b in Table 5.1, when an IRT mutation operator replaces non-numeric constant ‘S6’ in the SELECT clause of the SQL statement by another non-numeric field “staffid” that also appears in the SQL statement, the resultant mutant is “SELECT staffid, name, staffid FROM STAFF WHERE city = ‘C8’ ”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “city = ‘C8’ ” in the WHERE clause so as to set records in the database table such that, they will have ‘C7’, ‘C8’, and ‘C9’ in the field “city”. For these records, I then set the values for the “staffid” and the “name” field of the record to any value that is not equal to any constant included in the original SQL statement.

---

7This is similar to Step 1 of G-IRT-m7
Chapter 5. Test Case Generation for IRT Operator

field so that they are distinct and do not include ‘S6’.

An example may be (“city” = ‘C7’, “staffid” = ‘S0’, “name” = ‘Adam’), (‘C8’, ‘S1’, ‘David’), (‘C9’, ‘S2’, ‘Tom’).

By doing so, one can then distinguish the mutant from O3-2b because O3-2b returns “S1, David, S6” whereas the mutant returns “S1, David, S1”.

When the IRT mutation operator replaces a non-numeric constant in the SELECT clause by another constant, the same guideline is adequate to distinguish all relevant mutants. For example, for SQL statement O3-2b in Table 5.1, when an IRT mutation operator replaces non-numeric constant ‘S6’ in the SELECT clause of the SQL statement by another constant ‘C8’, the resultant mutant is “SELECT staffid, name, ‘C8’ FROM STAFF WHERE city = ‘C8’”. The same dataset can distinguish the mutant from O3-2b because O3-2b returns “S1, David, S6” whereas the mutant returns “S1, David, C8”. But in cases where the same constant is involved in the SELECT and the WHERE clause the mutants are equivalent to the original SQL statement.

Last, but not least, in some cases when the same non-numeric constant appeared in the SELECT and in the WHERE clause, the mutant with the equality relational operator (=) will be equivalent to the original SQL statement. For example, for the SQL statement, “SELECT staffid, name, ‘C8’ FROM STAFF WHERE city = ‘C8’ ”, when the IRT mutation operator replaces ‘C8’ in the SELECT clause by “city”, the resultant mutant is “SELECT staffid, name, city FROM STAFF WHERE city = ‘C8’ ”. As there is a condition “city = ‘C8’ ” and in the mutant ‘C8’ is also replaced by “city”, this mutant is equivalent to the original SQL statement.

5.3.3.2 IRT in WHERE clause

When an IRT mutation operator occurs in the WHERE clause, two types of mutants are generated namely, IRT-m9 to IRT-m10 depending on the numeric or non-numeric field involved in the SELECT and the WHERE clause. The test generation guidelines for these mutants are described as follows:

G-IRT-m9: For the IRT-m9 mutants, the IRT mutation operator replaces a numeric constant in the simple WHERE clause of the SQL statement by another numeric field or
constant that appears in the SQL statement. I now propose the test case generation
guideline that could distinguish this type of mutants from the original SQL. The
guideline consists of the following three major steps:

(i) Apply BVA on the condition in the \texttt{WHERE} clause to obtain the boundary values,
and then set the records in the database table so that each record will have one
of these values in the IRT-constant involved in the \texttt{WHERE} clause\footnote{These BVA values can also be used for other five relational operators like $\leq$, $>$, $=$, $<$ and $\neq$.}.

(ii) Set the values for the other numeric fields of the \texttt{SELECT} clause for those records
mentioned in (i) above so that
   (a) they are all distinct, and
   (b) each of these values is either far greater or smaller than the boundary value
and any constants that appears in the SQL statement.

For example, for the SQL statement O3-1a in Table 5.1, when an IRT mutation
operator replaces the 30 in the \texttt{WHERE} clause by “salary” the resultant mutant is
\texttt{SELECT grade, salary, 60 FROM STAFF WHERE age > salary}. In order to distinguish
this type of mutant from the original SQL statement, I need to apply BVA on the
condition “age > 30” so as to set the records in the database table such that, they will
have 29, 30, and 31 in the numeric field in the \texttt{WHERE} clause (that is “age”). For each
of these values in the records, I then set the values for the “grade” and the “salary”
fields such that either all of them are far greater or less than the boundary value (i.e.
30) and the constant (i.e. 60). An example may be ("age" = 29 “grade”=2, “salary” = 8), (30, 4, 9), (31, 6, 10). By doing so, one can then distinguish the mutant from
O3-1a because O3-1a returns “(6, 10, 60)” whereas the mutant returns “(2, 8, 60),
(4, 9, 60), (6, 10, 60)”.

In case of the numeric constant in the \texttt{WHERE} clause is replaced by the constant in
the \texttt{SELECT} clause the same guidelines is adequate to distinguish the mutant from
the original SQL statement. For example, for the SQL statement O3-1a in Table 5.1,
when an IRT mutation operator replaces 30 in the \texttt{WHERE} clause by 60 the resultant
mutant is “SELECT grade, salary, 60 FROM STAFF WHERE age > 60”. The above
mentioned dataset (e.g. ("age" = 29 “grade”=2, “salary” = 8), (30, 4, 9), (31, 6,
10)) can distinguish the mutant from O3-1a because O3-1a returns “(6, 10, 60)” whereas the mutant returns empty.

**G-IRT-m10:** For the IRT-m10 mutants, the IRT mutation operator replaces a non-numeric constant in the simple **WHERE** clause of the SQL statement by another non-numeric field or constant that also appears in the SQL statement. In order to distinguish the mutant from the original SQL statement, the guideline consists of the following two major steps:

(i) Apply BVA on the condition in the **WHERE** clause to obtain the boundary values, and then set the records in the database table so that each record will have one of these values in the non-numeric field in the **WHERE** clause and

(ii) Set the values for the other non-numeric fields of the **SELECT** clause for those records mentioned in (i) above so that

(a) they are all distinct, and

(b) either all of them are far greater or less than the boundary value. Moreover, these values are not equal to any constants that appear in the **SELECT** clause of the SQL statement.

For example, for the original SQL statement O3-2b in Table 5.1, when an IRT mutation operator replaces a non-numeric constant ‘C8’ by another non-numeric field “staffid” the resultant mutant is “**SELECT** staffid, name, ‘S6’ **FROM** STAFF **WHERE** city = staffid”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “city = ‘C8’ ” in the **WHERE** clause so as to set records in the database table such that, they have ‘C7’, ‘C8’, and ‘C9’ in the non-numeric field in the **WHERE** clause (that is “city”). For each of these values in the records, I then set the values of the “staffid” and the “name” field such that either all of them are far greater or less than the boundary value. An example may be (“city” = ‘C7’, “staffid” = ‘S1’, “name” = ‘Adam’), (‘C8’, ‘S2’, ‘David’), and (‘C9’,‘S3’, ‘Jack’). By doing so, one can then distinguish the mutants from O3-2b because O3-2b returns “S2, David, S6” whereas the mutant returns empty.

When the non-numeric constant in the **WHERE** clause is replaced by another non-numeric constant that appears in the **SELECT** clause of the SQL statement, the same
guide is adequate to distinguish the mutants from the original SQL statement. For example, for the original SQL statement O3-2b in Table 5.1, when an IRT mutation operator replaces a non-numeric constant ‘C8’ by another non-numeric constant ‘S6’, the resultant mutant is “SELECT staffid, name, ‘S6’ FROM STAFF WHERE city = ‘S6’”. The same dataset can distinguish the mutant from O3-2b because O3-2b returns “S2, David, S6” whereas the mutant returns empty.

5.3.4 Guidelines for Mutants from SQL with both Aggregate Function and WHERE clause

This section discusses test case generation guidelines for IRT mutants created from those original SQL statements having aggregate functions (such as \texttt{MAX()}, \texttt{MIN()}, \texttt{SUM()}, \texttt{AVG()}, and \texttt{COUNT()}\) in the \texttt{SELECT} clause and a simple \texttt{WHERE} clause. Similar to the previous sections, an IRT mutation operator may occur in an aggregate function of the \texttt{SELECT} clause or in the \texttt{WHERE} clause. The following two sections discuss these two situations.

5.3.4.1 IRT in Aggregate Function of the \texttt{SELECT} clause

As discussed earlier, when an IRT mutation operator occurs in an aggregate function of the \texttt{SELECT} clause, four types of mutant, namely IRT-m11 to IRT-m14, are possible depending on whether a numeric or a non-numeric field is involved in the \texttt{SELECT} clause or in the \texttt{WHERE} clause. In the following I discuss the test case generation guidelines for these four types of mutant represented as G-IRT-m11 to G-IRT-m14. Note that, each of these mutants is involved in aggregate functions (\texttt{COUNT()}, \texttt{MAX()}, \texttt{MIN()}, \texttt{SUM()}, and \texttt{AVG()}). For the ease of discussion, two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function \texttt{COUNT()} and in the second set, the guidelines are related to other aggregate functions functions (\texttt{MAX()}, \texttt{MIN()}, \texttt{SUM()}, and \texttt{AVG()}). The test case generation guidelines are discussed as follows:
**G-IRT-m11:** For the IRT-m11 mutants, the \textit{IRT} mutation operator replaces a numeric constant in the aggregate function of the \texttt{SELECT} clause by another numeric field\(^9\) or numeric constant that also appears in the SQL statement and a numeric field appears in the \texttt{WHERE} clause of the SQL statement. Because of the involvement of numeric fields, any aggregate functions can be applied on these numeric fields. As discussed earlier, two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function \texttt{COUNT()} and in the second set, the guidelines are related to other aggregate functions.

1. (a) Situation when the \textit{IRT} mutation operator occurs in the aggregate function \texttt{COUNT()}. Since \texttt{COUNT()} returns the number of non-null values in the required field, when the fields involved in the \textit{IRT} mutation operator do not allow \textit{null} values, there is no way to distinguish the original statement from its mutant. In other words, the mutants are equivalent to the original SQL statement. However, when the fields involved allow \textit{null} values, it is possible to distinguish the original SQL statement from the mutants. To do so, the guideline consists of the following two steps:

   (i) Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values and then set the records in the database table so that they will have one of these values in the numeric field involved in the \texttt{WHERE} clause\(^{10}\).

   (ii) Set the values in the other numeric fields of the \texttt{SELECT} clause for those records mentioned in (i) above so that

   (a) different numeric fields have different number of \textit{null} values

   (b) there is at least one \textit{null} value in each numeric field and

   (c) for each record, it can only have at most one \textit{null} value in its numeric field

\(^9\)Please refer to Table 4.3

\(^{10}\)These BVA values can also be used for other five relational operators like \(\leq, >, =, <\) and \(\neq\)
By doing so, I need to set multiple records for each of the BVA values found in (i) above.

For example, for the SQL statement “SELECT COUNT(grade), COUNT(salary), COUNT(60) FROM STAFF WHERE age > 30”, when an IRT mutation operator replaces 60 in the aggregate function of the SELECT clause by “grade”, the resultant mutant is “SELECT COUNT(grade), COUNT(salary), COUNT(grade) FROM STAFF WHERE age > 30”. In order to distinguish this type of mutants from the original SQL statement, I need to apply BVA on the condition “age > 30” in the WHERE clause so as to set records in the database table such that they have 29, 30, and 31 in the numeric field in the WHERE clause (that is, “age”). For these records, I then set different number of null values for the “grade” and the “salary” fields in the database records. A possible example may be ("age" = 29, "grade" = 2, "salary" = null), (29, 4, null), (29, null, 3), (30, 6, null), (30, null, 5), (30, null, 7), (31, 8, null), (31, 10, null), (31, null, 9). By doing so, one can then distinguish COUNT(grade) from COUNT(60) for the condition “age > 30” in the WHERE clause because COUNT(60) returns 3 whereas COUNT(grade) returns 2.

Note that, in the above example if 60 is replaced by the field in the WHERE clause (that is “age”), the mutant is equivalent to the original SQL statement, because the WHERE condition (“age” > 30) always filters out the same number of records, and hence, COUNT(60) and COUNT(age) return the same result.

Again, for the COUNT() aggregate function, if the numeric constant in the SELECT clause is replaced by another numeric constant, the mutants are always equivalent to the original SQL statement.

(b) In the case of COUNT(DISTINCT ...), the test case generation guidelines are different from the case of COUNT(). Please note that when the fields involved in the IRT mutation operator allow null values, it is possible to distinguish the mutants from the original SQL statement by setting the values in the related fields such that they have different number of null values. Hence, the guidelines of setting different null values for the COUNT() are also applicable for COUNT(DISTINCT ...) when the fields allow null values. On the other
hand, unlike the situation in the \texttt{COUNT()}, if the fields cannot be null, one can still distinguish the mutants with \texttt{COUNT(DISTINCT ...)} from its original SQL statement by the following steps:

i. Apply modified boundary value analysis on the condition in the \texttt{WHERE} clause to obtain the boundary values that are 1-off and 2-off the boundary value$^{11}$

ii. For those values mentioned in (i) above, set the database records as follows:

(a) For the boundary value, there are at least two records having the boundary value in the numeric field involved in the \texttt{WHERE} clause

(b) For the remaining values, there is at least one record having each value in the numeric field involved in the \texttt{WHERE} clause

iii. Set the values in the other numeric fields in the \texttt{SELECT} clause for those records mentioned in (ii) above so that they are all distinct

As a reminder, for the five values obtained in (i) there are at least 6 records in the database for testing.

For example, for the SQL statement \texttt{“SELECT COUNT(grade), COUNT(salary), COUNT(DISTINCT 60) FROM STAFF WHERE age > 30”}, when 60 is replaced by “grade” the resultant mutant is \texttt{“SELECT COUNT(grade), COUNT(salary), COUNT(DISTINCT grade) FROM STAFF WHERE age > 30”}. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “age > 30” in the \texttt{WHERE} clause so as to set records in the database table such that the numeric field in \texttt{WHERE} clause (that is, “age”) will have 28, 29, 30, 31, and 32. I then set some distinct values for the “grade” and the “salary” fields in the \texttt{SELECT} clause against the same value of the “age” field to have the records like (“age” = 28, “grade” = 1, “salary” = 300), (29, 2, 400), (30, 4, 500), (30, 5, 600), (31, 7, 800), and (32, 7, 1000). By doing so, one can then distinguish the result of \texttt{COUNT(DISTINCT grade)} from \texttt{COUNT(DISTINCT 60)} for the condition “age>30” in the \texttt{WHERE} clause, because \texttt{COUNT(DISTINCT 60)} returns 1 whereas \texttt{COUNT(DISTINCT grade)} returns 2.

$^{11}$These values are the boundary value, “one-below” the boundary value, “one-above” the boundary value, “two-below” the boundary value, and “two-above” the boundary value.
Note that, for the SQL statement involved in the equality relational operator (=), if the constant (i.e. 60) in the aggregate function of the SELECT clause is replaced by the field in the WHERE clause (that is “age”), the mutant is equivalent to the original SQL statement, because the WHERE condition (“age” = 30) always filters out the same number of records, and hence, `COUNT(DISTINCT 60)` and `COUNT(DISTINCT age)` return the same result.

Again, for `COUNT(DISTINCT ...)` aggregate function, if the numeric constant in the SELECT clause is replaced by another numeric constant, the mutants are always equivalent to the original SQL statement. Hence, no test dataset can distinguish this type of mutant.

2. Situation when the constant is involved in an aggregate function other than `COUNT()` (i.e. `MAX()`, `MIN()`, `SUM()`, `AVG()`). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an IRT mutation operator replaces a numeric constant `c1` in the aggregate function `AGGR()` (may be `MAX()`, `MIN()`, ...) by another numeric field `f1` or by another constant `c2`, the IRT mutant will have `AGGR(f1)` or `AGGR(c2)` instead of `AGGR(c1)`. In order to distinguish `AGGR(f1)` or `AGGR(c2)` from `AGGR(c1)`, the guideline consists of the following steps:

(i) Apply BVA on the condition in the WHERE clause to obtain the boundary values, and set the records in the database table so that each record will have one of these values in the field involved in the WHERE clause.

(ii) Set some non-zero distinct values for every numeric field for those records mentioned in (i) above so that they are far greater or less than those constants (e.g. `c1` and `c2`) already existed in the SQL statement.

It does not matter whether there is a ‘DISTINCT’ keyword in the aggregate function, the same guideline is adequate to distinguish the mutants from the original SQL statement.

For example, for the SQL statement O4-1a in Table 5.1, when an IRT mutation operator replaces 60 in the aggregate function (`MAX()`) of the SELECT clause by another
numeric field “grade”, the resultant mutant is “\(\text{SELECT } \max(\text{grade}), \min(\text{salary}), \max(\text{grade}) \text{ FROM } \text{STAFF WHERE } \text{age} > 30\)”. In order to distinguish this type of mutant from O4-1a, I need to apply BVA on the condition “age > 30” in the \texttt{WHERE} clause so as to set three records in the database table such that, each record will have 29, 30 and 31 in the field “age”. For these three records, I need to set some non-zero values for the “grade” and the “salary” fields so that they are far greater or less than those constants (e.g. 60 and 30). An example may be (“age” = 29, “grade” = 100, “salary” = 500), (30, 200, 1000), and (31, 300, 1500). By doing so, one can then distinguish the mutant from O4-1a because O4-1a returns “300, 1500, 60” whereas the mutant returns “300, 1500, 300”.

Again, for the SQL statement O4-1a in Table 5.1, when an \textit{IRT} mutation operator replaces 60 in the aggregate function of the \texttt{SELECT} clause by another constant 30, the resultant mutant is “\(\text{SELECT } \max(\text{grade}), \min(\text{salary}), \max(30) \text{ FROM } \text{STAFF WHERE } \text{age} > 30\)”. The dataset selected in the previous example (e.g. (“age” = 29, “grade” = 100, “salary” = 500), (30, 200, 1000), and (31, 300, 1500)) is adequate to distinguish the mutant as for the \texttt{WHERE} condition “age > 30”, \texttt{MAX}(60) always returns 60 and the \texttt{MAX}(30) always returns 30.

There is an exception scenario where the mutant is equivalent to the original SQL statement. If (1) the same constant is used in the \texttt{SELECT} and the \texttt{WHERE} clause, (2) the constant in the \texttt{SELECT} clause is replaced by the field in the \texttt{WHERE} clause, and (3) the SQL statement involves the equality relational operator (=), the mutants will be equivalent the original SQL statement. For example, for the SQL statement “\(\text{SELECT } \max(\text{grade}), \min(\text{salary}), \max(30) \text{ FROM } \text{STAFF WHERE } \text{age} = 30\)”, when 30 is replaces by the field in the \texttt{WHERE} clause (that is “age”), the mutant is equivalent to the original SQL statement, because the \texttt{WHERE} condition always filters out the same records, and hence, \texttt{MAX}(30) and \texttt{MAX}(age) return the same result.

\textbf{G-IRT-m12:} For the IRT-m12 mutant, the \textit{IRT} mutation operator replaces a numeric constant in the aggregate function of the \texttt{SELECT} clause by another numeric field or constant that also appears in the SQL statement and a non-numeric\footnote{Please refer to Table 5.3} field appears in the
WHERE clause. Two different sets of guidelines are proposed, depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function COUNT() and in the second set, the guidelines are related to other aggregate functions.

1. (a) Situation when the IRT mutation operator occurs in the aggregate function COUNT(). Since COUNT() returns the number of non-null values in the required field, when the fields involved in the IRT mutation operator do not allow null values, the mutants are equivalent to the original SQL statement. However, when the fields involved allow null values, it is possible to distinguish the mutants from the original SQL statement. The guideline consists of the following two major steps:

i. Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the non-numeric field involved in the WHERE clause.

ii. Set the values for the numeric fields in the aggregate functions of the SELECT clause for those records mentioned in (i) above so that
   (a) different numeric fields have different number of null values
   (b) there is least one null value in each numeric field and
   (c) for each record, it can only have at most one null value in its numeric field

By doing so, I need to set multiple records for each of the BVA values found in (i) above.

For example, for the SQL statement “SELECT COUNT(grade), COUNT(salary), COUNT(60), COUNT(50) FROM STAFF WHERE staffid = ‘S1’ ”, when an IRT mutation operator replaces 60 in the aggregate function (COUNT()) by another numeric field “grade”, the resultant mutant is “SELECT COUNT(grade), COUNT(salary), COUNT(grade), COUNT(50) FROM STAFF WHERE staffid = ‘S1’ ”. In order to distinguish this type of mutant from the original SQL statement,
I need to apply BVA on the condition “staffid = ‘S1’” in the WHERE clause so as to set records in the database table such that, they have ‘S0’, ‘S1’, and ‘S2’ in the field in the WHERE clause (that is “staffid”). For these records, I then set the values for the “grade” and the “salary” field that appear in the SELECT clause in such a way that “grade” and “salary” have different number of null values. A possible example may be (‘staffid’ = ‘S0’, “grade” = 2, “salary” = null), (‘S0’, 4, null), (‘S0’, null, 3), (‘S1’, 6, null), (‘S1’ null, 5), (‘S1’, null, 7), (‘S2’, 8, null), (‘S2’, 10, null), (‘S2’, null, 9). By doing so, one can then distinguish COUNT(grade) from COUNT(60) for the condition “staffid = ‘S1’ in the WHERE clause because COUNT(60) returns “3” whereas the mutant returns “1”.

Again, for the COUNT() aggregate function, if the numeric constant in the SELECT clause is replaced by another numeric constant, the mutants are always equivalent to the original SQL statement. Hence, no test dataset can kill this type of mutant.

(b) For the case of COUNT(DISTINCT ...), the test case generation guidelines are different from that of COUNT(). Please note that when the fields involved in the IRT mutation operator allow null values, it is possible to distinguish the mutants from the original SQL statement by setting the values in the related fields such that they have different number of null values. Hence, the guidelines of setting different null values for COUNT() is also applicable for COUNT(DISTINCT ...) when the fields allow null values. On the other hand, unlike the situation in COUNT(), if the fields cannot be null, one can still distinguish the mutants with COUNT(DISTINCT ...) from its original SQL by the following steps:

i. Apply BVA on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the non-numeric field in the WHERE clause\(^\text{13}\)

ii. Set the values in the numeric fields in the SELECT clause for those records mentioned in (i) above so that they have different number of distinct values.

To achieve this, I need to set multiple distinct values for the numeric fields

\(^{13}\text{This is similar to Step 1 of G-IRT-m7}\)
in the SELECT clause against the same boundary value of the non-numeric field in the WHERE clause.

For example, for the SQL statement “SELECT COUNT(grade), COUNT(salary), COUNT(DISTINCT 60), COUNT(50) FROM STAFF WHERE staffid = ‘S1’ ”, when an IRT mutation operator replaces “salary” in the aggregate function of the SELECT clause by “grade”, the resultant mutant is “SELECT COUNT(grade), COUNT(salary), COUNT(DISTINCT grade), COUNT(50) FROM STAFF WHERE staffid = ‘S1’ ”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “staffid = ‘S1’ ” in the WHERE clause so as to set records in the database table such that the non-numeric field in the WHERE clause (that is, “staffid”) will have ‘S0’, ‘S1’, and ‘S2’. For these records, I then set values for the “grade” and the “salary” field in the aggregate function of the SELECT clause so that they have different number of distinct values. An example may be (“staffid” = ‘S0’, “grade” = 10, “salary” = 100), (“S0’, 10, 200), (“S0’, 15, 300), (“S1’, 5, 500), (“S1’, 6, 200), (“S1’, 7, 200), (“S2’, 15, 1000), and (“S2’, 15, 1500), (“S2’, 20, 2000). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “3, 2, 1, 1” whereas the mutant returns “3, 2, 3, 1”.

Again, for COUNT(DISTINCT ...) aggregate function, if the numeric constant in the SELECT clause is replaced by another numeric constant, the mutants are always equivalent to the original SQL statement. Hence, no test dataset can distinguish this type of mutant.

2. Situation when the IRT mutation operator occurs in an aggregate function other than COUNT() (e.g. MAX(), MIN, SUM, AVG()). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an IRT mutation operator replaces a numeric constant c1 in the aggregate function AGGR() (may be MAX(), MIN(), ...) by another numeric field f1 or other constant c2, the IRT mutant will have AGGR(f1) or AGGR(c2) instead of AGGR(c1). In order to distinguish AGGR(f1) or AGGR(c2) from AGGR(c1) the guidelines consist of the following two steps:
(i) Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the field involved in the non-numeric field in the \texttt{WHERE} clause.

(ii) Set some non-zero distinct values for every numeric field for those records mentioned in (i) above so that these values are far greater or less than those constants (e.g. \texttt{c1} and \texttt{c2}) already existed in the SQL statement.

It does not matter whether there is a \texttt{DISTINCT} keyword in the aggregate function, the same guidelines will be adequate.

For example, for the SQL statement O4-1b, in Table 5.1 when an \textit{IRT} mutation operator replaces 60 in the aggregate function of the \texttt{SELECT} clause by another numeric field \texttt{“grade”}, the resultant mutant is \texttt{“SELECT MAX(grade), MIN(salary), MAX(grade), MIN(50) FROM STAFF WHERE staffid = ‘S1’ “}. In order to distinguish this type of mutant from O4-1b, I need to apply BVA on the condition \texttt{“staffid = ‘S1’ “} in the \texttt{WHERE} clause so as to set records in the database table such that, they will have \texttt{‘S0’, ‘S1’, and ‘S2’} in the non-numeric field in the \texttt{WHERE} clause (that is, \texttt{“staffid”})

For these records, I then set some non-zero distinct values for the \texttt{“grade”} and the \texttt{“salary”} fields. An example may be (\texttt{“staffid”} = \texttt{‘S0’, “grade”} = 1, \texttt{“salary”} = 10), (\texttt{‘S0’, 2, 12}), (\texttt{‘S1’, 3, 15}), (\texttt{‘S1’, 4, 18}), (\texttt{‘S2’, 5, 20}), and (\texttt{‘S2’, 6, 25}). By doing so, one can then distinguish the mutant from O4-1b because O4-1b returns \texttt{“4, 18, 60, 50”} whereas the mutant returns \texttt{“4, 18, 4, 50”}.

Again, for the SQL statement O4-1b in Table 5.1, when an \textit{IRT} mutation operator replaces 60 in the aggregate function of the \texttt{SELECT} clause by another constant 50, the resultant mutant is \texttt{“SELECT MAX(grade), MIN(salary), MAX(50), MIN(50) FROM STAFF WHERE staffid = ‘S1’ “}. The mutant can be always distinguished from the original SQL statement because for the \texttt{WHERE} condition \texttt{“staffid = ‘S1’ “}, \texttt{MAX(60)} always returns 60 and the \texttt{MAX(30)} always returns 30.

\textbf{G-IRT-m13:} For the IRT-m13 mutants, the \textit{IRT} mutation operator replaces a non-numeric constant in an aggregate function of the \texttt{SELECT} clause by another non-numeric
field or constant that also appears in the SQL statement and a numeric field is involved in the simple condition in the WHERE clause. As the non-numeric fields and constants are involved in the aggregate functions of the SELECT clause, COUNT(), MAX(), and MIN() are the only legitimate aggregate functions. Two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function COUNT() and in the second set, the guidelines are about other aggregate functions.

1. (a) Situation when the IRT mutation operator occurs in the aggregate function COUNT(). Since COUNT() returns the number of non-null values in the required field, when the fields involved in the IRT mutation operator do not allow null values, there is no way to distinguish the original statement from its mutants. However, when the fields involved allow null values, it is possible to distinguish the mutants from the original SQL statement. The guideline consists of the following two major steps:

   i. Apply BVA on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the numeric field in the WHERE clause and

   ii. Set the values in the non-numeric fields in the aggregate function of the SELECT clause for those records mentioned in (i) above so that

      (a) different non-numeric fields have different number of null values

      (b) there is least one null value in each non-numeric field and

      (c) for each record, it can only have at most one null value in its non-numeric fields.

For example, for the SQL statement “SELECT COUNT(staffid), COUNT(name), COUNT(’S6’), COUNT(’S10’) FROM STAFF WHERE age > 30”, when an IRT mutation operator replaces a non-numeric constant ‘S6’ in the aggregate function (COUNT()) of the SELECT clause by another non-numeric field “staffid” the resultant mutant is “SELECT COUNT(staffid), COUNT(name), COUNT(staffid), COUNT(’S10’) FROM STAFF WHERE age > 30”. In order to distinguish this type of mutant from the the original SQL statement, I need to apply BVA on the
condition “age > 30” in the WHERE clause so as to set records in the database table such that, they will have 29, 30, and 31 in the numeric field in the WHERE clause (that is “age”). For these records, I then set different number of null values for the “staffid” and the “name” fields. A possible example may be (“age” = 29, “staffid” = ‘S1’, “name” = null), (29, ‘S2’, null), (29, null, ‘Jack’), (30, ‘S3’, null), (30, null, ‘David’), (30, null, Adam), (31, ‘S4’, null), (31, ‘S5’, null), (31, null, ‘Tony’). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “2, 1, 3, 3” whereas the mutant returns “2, 1, 2, 3”.

Again, for COUNT() aggregate function, when the non-numeric constant in the SELECT clause is replaced by another non-numeric constant, the mutants are always equivalent to the original SQL statement. Hence, no test dataset can distinguish this type of mutant.

(b) In the case of COUNT(DISTINCT ...), the test case generation guidelines are different from the COUNT(). Please note that when the fields involved in the IRT mutation operator allow null values, it is possible to distinguish the mutants from the original SQL statement by setting the values in the related fields such that they have different number of null values. Hence, the guidelines of setting distinct null values for COUNT() is also applicable for COUNT(DISTINCT ...) when the fields allow null values. On the other hand, unlike the situation in COUNT(), if the fields cannot be null, one can still distinguish the mutants with COUNT(DISTINCT ...) from its original SQL by the following steps:

(i) Apply BVA on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the numeric field in the WHERE clause and

(ii) Set the values in the non-numeric fields of the SELECT clause for those records mentioned in (i) above so that these fields have different number of distinct values. To achieve this, I need to set multiple distinct values

\[ \text{Need to set the values of the other numeric fields so that they have at least two distinct values. Otherwise, } \text{COUNT(DISTINCT } c) = \text{COUNT(DISTINCT } f) \text{, where } c \text{ is a constant and } f \text{ is a non-numeric field.} \]
for the non-numeric fields in the SELECT clause against the same boundary value of the numeric field in the WHERE clause.

For example, for the SQL statement “SELECT COUNT(staffid), COUNT(name), COUNT(DISTINCT ‘S6’), COUNT(‘S10’) FROM STAFF WHERE age > 30”, when an IRT mutation operator replaces a non-numeric constant ‘S6’ in the aggregate function of the SELECT clause by another non-numeric field “staffid” the resultant mutant is “SELECT COUNT(staffid), COUNT(name), COUNT(DISTINCT staffid), COUNT(‘S10’) FROM STAFF WHERE age > 30”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “age > 30” in the WHERE clause so as to set records in the database table such that, they will have 29, 30, and 31 in the numeric field of the WHERE clause (that is, “age”). I then set some different number of distinct values for the “staffid” and the “name” field in the aggregate function of the SELECT clause against the same value of the “age” field to have the records like (“age” = 29, “staffid” = ‘S1’, “name” = ‘Jack’), (29, ‘S2’, ‘Jack’), (29, ‘S3’, ‘Adam’), (30, ‘S4’, ‘Gill’), (30, ‘S4’, ‘David’), (30, ‘S5’, ‘James’), (31, ‘S7’, ‘Tony’), (31, ‘S9’, ‘Tony’), and (31, ‘S12’, ‘Bill’). By doing so, one can then distinguish COUNT(DISTINCT staffid) from COUNT(DISTINCT ‘S6’) for the condition “age > 30” in the WHERE clause because COUNT(DISTINCT ‘S6’) returns “1” whereas COUNT(DISTINCT staffid) returns “3”.

2. Situation when an IRT mutation operator occurs in an aggregate function other than COUNT() (i.e. MAX() and MIN()). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an IRT mutation operator replaces a non-numeric constant s1 in the SELECT clause of the SQL statement by another non-numeric field f1 or a non-numeric constant s2 in the aggregate function AGGR() the IRT mutant will have AGGR(f1) or AGGR(s2) instead of AGGR(s1). In order to distinguish AGGR(f1) or AGGR(s2) from AGGR(s1) the guidelines consist of the following two steps:

(a) Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database table so
that they will have one of these values in the field involved in the numeric field
in the WHERE clause.

(b) Set some non-null distinct values for every non-numeric field for those records
mentioned in (i) above so that these values are far greater or less than those
constants (e.g. s1 and s2) already existed in the SQL statement.

For example, for the SQL statement O4-2a in Table 5.1, when an IRT mutation
operator replaces ‘S6’ in the SELECT clause by another non-numeric field “staffid”,
the resultant mutant is “SELECT MAX(staffid), MIN(name), MAX(staffid), MIN(‘S10’) FROM STAFF WHERE age > 30”. In order to distinguish this type of mutant from
O4-2a, I need to apply BVA on the condition “age > 30” in the WHERE clause so
as to set records in the database table such that, they have 29, 30, 31 in the “age”
field. For these records, I then set some non-null distinct values for the “staffid”
and the “name” fields so that these values are far greater or less than ‘S6’ and
‘S10’. An example may be (“age” = 29, “staffid”=‘S0’, “name”=‘Adam’), (30, ‘S1’,
‘Bill’), and (31, ‘S2’, ‘David’). By doing so, one can then distinguish MAX(staffid)
from MAX(‘S6’) for the condition “age > 30” in the WHERE clause because MAX(‘S6’)
returns ‘S6’ whereas MAX(staffid) returns ‘S2’.

Again, for the SQL statement O4-2a in Table 5.1, when an IRT mutation operator
replaces ‘S6’ in the aggregate function of the SELECT clause by another constant ‘S10’,
the resultant mutant is “SELECT MAX(staffid), MIN(name), MAX(‘S10’), MIN(‘S10’) FROM STAFF WHERE age > 30”. The mutant can be always distinguished from the
original SQL statement because MAX(‘S6’) always returns ‘S6’ and the MAX(‘S10’) always returns ‘S10’.

It does not matter whether there is a ‘DISTINCT’ keyword in the SQL statement, the same
guideline is adequate to distinguish all related mutants from the original SQL statement.

**G-IRT-m14:** For the IRT-m14 mutants, the IRT mutation operator replaces a non-
numeric constant in the aggregate function of the SELECT clause by another non-numeric
field or constant and a non-numeric field appears in the WHERE clause. As this type of IRT
mutant involves non-numeric fields, MAX(), MIN(), and COUNT() are the only legitimate
aggregate functions. Again, two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function \( \text{COUNT()} \) and in the second set, the guidelines are related to other aggregate functions.

1. (a) Situation when an \textit{IRT} mutation operator occurs in the aggregate function \( \text{COUNT()} \). Since \( \text{COUNT()} \) returns the number of \textit{non-null} values in the required field, when the fields involved in the \textit{IRT} mutation operator do not allow \textit{null} values, there is no way to distinguish the original statement from its mutant. In other words, the mutants are equivalent to the original SQL statement. However, when the fields involved allow \textit{null} values, it is possible to distinguish the original SQL statement from the mutants. The guideline consists of the following two major steps:

   i. Apply boundary value analysis (BVA) on the condition in the \textit{WHERE} clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the non-numeric field involved in the \textit{WHERE} clause\(^\text{15}\).

   ii. Set the values in the other non-numeric fields of the \textit{SELECT} clause for those records mentioned in (i) above so that

      (a) different non-numeric fields have different number of \textit{null} values

      (b) there is at least one \textit{null} value in each numeric field and

      (c) for each record, it can only have at most one \textit{null} value in its non-numeric field

For example, for the SQL statement \texttt{“SELECT COUNT(staffid), COUNT(name), COUNT(’S6’) FROM STAFF WHERE city = ‘C8’”}, when an \textit{IRT} mutation operator replaces ‘S6’ in the aggregate function of the \textit{SELECT} clause by “staffid”, the resultant mutant is \texttt{“SELECT COUNT(staffid), COUNT(name), COUNT(staffid) FROM STAFF WHERE city = ‘C8’”}. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “city = ‘C8’

\(^{15}\text{These BVA values can also be used for other five relational operators like } \leq, \geq, =, 
< \text{ and } \neq \text{.} \)
" in the WHERE clause so as to set records in the database table such that, they will have ‘C7’, ‘C8’, and ‘C9’ in the non-numeric field in the WHERE clause (that is, “city”). For these records, I then set some different number of null values for the “staffid” and the “name” fields that appears in the aggregate function of the SELECT clause. A possible example may be (“city” = ‘C7’, “staffid” =’S1’, “name”= null), (‘C7’, ‘S2’, null), (‘C7’, null, ‘Jack’), (‘C8’, ‘S3’, null), (‘C8’, null, ‘David’), (‘C8’, null, Adam), (‘C9’, ‘S4’, null), (‘C9’, ‘S5’, null), (‘C9’, null, ‘Tony’). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “1, 2, 3” whereas the mutant returns “1, 2, 1”.

Note that, in the above example if ‘S6’ is replaced by the field in the WHERE clause (that is “city”), the mutant is equivalent to the original SQL statement, because the WHERE condition always filters out the same number of records, and hence, \( \text{COUNT('S6')} \) and \( \text{COUNT(city)} \) return the same result.

Again, for \( \text{COUNT()} \) aggregate function, if the non-numeric constant in the SELECT clause is replaced by another non-numeric constant, the mutants are always equivalent to the original SQL statement. Hence, no test dataset can distinguish this type of mutant.

(b) In the case of \( \text{COUNT(DISTINCT ...)} \), the test case generation guidelines are different from the case of \( \text{COUNT()} \). Please note that when the fields involved in the IRT mutation operator allow null values, it is possible to distinguish the mutants from the original SQL statement by setting the values in the related fields such that they have different number of null values. Hence, the guidelines of setting different null values for \( \text{COUNT()} \) are also applicable for \( \text{COUNT(DISTINCT ...)} \) when the fields allow null values. On the other hand, unlike the situation in \( \text{COUNT()} \), if the fields cannot be null, one can still distinguish the mutants with \( \text{COUNT(DISTINCT ...)} \) from its original SQL by the following steps:

i. Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values, and then set the records in the database
table so that they will have one of these values in the non-numeric field involved in the \texttt{WHERE} clause and

ii. Set the values in the other non-numeric fields in the \texttt{SELECT} clause for those records mentioned in (i) above so that these fields have different number\footnote{Need to set the values of the other numeric fields so that they have at least two distinct values. Otherwise, \texttt{COUNT(DISTINCT c)} = \texttt{COUNT(DISTINCT f)}, where \texttt{c} is a constant and \texttt{f} is a non-numeric field.} of distinct values. To achieve this, I need to have multiple distinct values for the other non-numeric fields in the \texttt{SELECT} clause against the same boundary value in the non-numeric field of the \texttt{WHERE} clause.

For example, for the SQL statement “\texttt{SELECT COUNT(staffid)}, \texttt{COUNT(name)}, \texttt{COUNT(DISTINCT ‘S6’)} FROM STAFF \texttt{WHERE city = ‘C8’ “}, when an IRT mutation operator replaces ‘S6’ in the aggregate function of the \texttt{SELECT} clause by \texttt{“staffid”}, the resultant mutant is “\texttt{SELECT COUNT(staffid)}, \texttt{COUNT(name)}, \texttt{COUNT(DISTINCT staffid)} FROM STAFF \texttt{WHERE city = ‘C8’ “}. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition \texttt{“city = ‘C8’ “} in the \texttt{WHERE} clause so as to set records in the database table such that, they will have ‘C7’, ‘C8’, and ‘C9’ in the non-numeric field in the \texttt{WHERE} clause (that is, \texttt{“city”}). For these records, I then set some distinct values for the \texttt{“staffid”} and the \texttt{“name”} fields in the aggregate function of the \texttt{SELECT} clause. A possible example may be (\texttt{“city”} = ‘C7’, \texttt{“staffid”} = ‘S1’, \texttt{“name”} = ‘Jack’), (‘C7’, ‘S2’, ‘Alex’), (‘C7’, ‘S3’, ‘Alex’), (‘C8’, ‘S3’, ‘Mike’), (‘C8’, ‘S3’, ‘Peter’), (‘C8’, ‘S4’, ‘James’) (‘C9’, ‘S4’, ‘David’), (‘C9’, ‘S5’, ‘David’), and (‘C9’, ‘S5’, ‘Adam’). By doing so, one can then distinguish \texttt{COUNT(DISTINCT staffid)} from \texttt{COUNT(DISTINCT ‘S6’)} for the condition \texttt{“city = ‘C8’ “} in the \texttt{WHERE} clause because \texttt{COUNT(DISTINCT ‘S6’)} returns “1” whereas \texttt{COUNT(DISTINCT staffid)} returns “2”.

Note that, same as \texttt{COUNT()} in the above example, when ‘S6’ is replaced by the field in the \texttt{WHERE} clause (that is \texttt{“city”}), the mutant is equivalent to the original SQL statement, because the \texttt{WHERE} condition always filters out the same number of records and hence \texttt{COUNT(DISTINCT ‘S6’)} and \texttt{COUNT(DISTINCT staffid)} return the same result (i.e. 1). In the same way, SQL statement involves in ‘less
than’ (\(<\)) and ‘greater than’ (\(>\)) relational operators the mutants are equivalent. But for the SQL statement involves in \(\geq\), \(\leq\) and \(\neq\) relational operators, the above mentioned guideline is adequate to distinguish all relevant mutants.

Again, for \texttt{COUNT(DISTINCT ...)} aggregate function, if the non-numeric constant in the \texttt{SELECT} clause is replaced by another non-numeric constant, the mutants are always equivalent to the original SQL statement. Hence, no test dataset can distinguish this type of mutant.

2. Situation when an \textit{IRT} mutation operator occurs in an aggregate function other than \texttt{COUNT()} (i.e. \texttt{MAX()}, \texttt{MIN()}). Instead of having one guideline per aggregate function, I propose one guideline for all these aggregate functions. If an \textit{IRT} mutation operator replaces a non-numeric constant \texttt{s1} in the aggregate function \texttt{AGGR()} (may be \texttt{MAX()}, \texttt{MIN()}, \ldots) by another non-numeric field \texttt{f1} or non-numeric constant \texttt{s2} the \textit{IRT} mutant will have \texttt{AGGR(f1)} or \texttt{AGGR(s2)} instead of \texttt{AGGR(s1)}. In order to distinguish \texttt{AGGR(f1)} or \texttt{AGGR(s2)} from \texttt{AGGR(s1)}, the guideline consists of the following two major steps:

(i) Apply BVA on the condition in the \texttt{WHERE} clause to obtain the boundary values, and set the records in the database table so that each record will have one of these values in the field in the \texttt{WHERE} clause.

(ii) Set some \textit{non-null} distinct values for every non-numeric field for those records mentioned in (i) above so that these values are far greater or less than those constant strings (e.g. \texttt{s1} and \texttt{s2}) already existed in the SQL statement.

For example, for the SQL statement O4-2b in Table 5.1, when an \textit{IRT} mutation operator replaces ‘S6’ in the aggregate function (\texttt{MAX()}) of the SQL statement by “staffid”, the resultant mutant is “\texttt{SELECT MAX(staffid), MIN(name), MAX(staffid)} FROM STAFF WHERE city = ‘C8’ ”. In order to distinguish this type of mutant from O4-2b, I need to apply BVA on the condition “city = ‘C8’ ” in the \texttt{WHERE} clause so as to set three records in the database table such that, they will have ‘C7’, ‘C8’, and ‘C9’ in the non-numeric field in the \texttt{WHERE} clause (that is, “city”). For these records, I then set some \textit{non-null} distinct values for the “staffid” and the “name” field so that these values are far greater or less than ‘S6’ and ‘C8’. An example may
be ("city" = ‘C7’, “staffid” = ‘A1’, “name” = ‘Adam’), (‘C8’, ‘A2’, ‘Bill’), and (‘C9’, ‘A3’, ‘Alex’). By doing so, one can then distinguish O4-2b from the mutant because O4-2b returns “A2, Bill, S6” whereas the mutant returns “A2, Bill, A2”.

There is an exception scenario where the mutant is equivalent to the original SQL statement. If (1) the same constant is used in the SELECT and the WHERE clause, (2) the constant in the SELECT clause is replaced by the field in the WHERE clause, and (3) the SQL statement involves the equality relational operator (=), the mutants will be equivalent to the original SQL statement. For example, for the SQL statement “SELECT MAX(staffid), MIN(name), MAX(‘C8’) FROM STAFF WHERE city = ‘C8’ ”, when an IRT mutation operator replaces ‘C8’ in the SELECT clause by “city”, the mutant is equivalent to the original SQL statement, because the WHERE condition “city = ‘C8’ ” always filter out the same records, and hence, MAX(‘C8’) and MAX(city) returns the same result.

Again, for the SQL statement O4-2b in Table 5.1, when an IRT mutation operator replaces ‘S6’ in the aggregate function of the SELECT clause by another constant ‘C8’, the resultant mutant is “SELECT MAX(staffid), MIN(name), MAX(‘C8’) FROM STAFF WHERE city = ‘C8’ ”. The dataset selected in the previous example is adequate to distinguish the mutant as MAX(‘S6’) always returns ‘S6’ and the MAX(‘C8’) always returns ‘C8’.

It does not matter whether there is a ‘DISTINCT’ keyword in the aggregate function, the same guideline will work.

5.3.4.2 IRT in the WHERE clause

As mentioned earlier, this subsection discusses the test case generation guidelines for SQL statements having aggregate functions and WHERE clause and the IRT mutation operator occurs in the WHERE clause. As explained earlier, two types of mutant, namely IRT-m15 and IRT-m16, are possible, depending on whether the fields involved in the SELECT and WHERE clauses are numeric or non-numeric. The test case generation guidelines for these mutants are described as follows:
G-IRT-m15: For the IRT-m15 mutant, the *IRT* mutation operator replaces a numeric constant in the *WHERE* clause by another numeric field or constant that also appears in the SQL statement. Two different sets of guidelines are proposed depending on the characteristics of aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function *COUNT()* and in the second set, the guidelines are related to other aggregate functions.

1. Situation when the aggregate function *COUNT()* is involved in the SQL statement.

   In order to distinguish such mutants from the original SQL statement, the guideline consists of the following two major steps:

   (i) Apply boundary value analysis (BVA) on the condition in the *WHERE* clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the field involved in the *WHERE* clause.

   (ii) Set the values for the other numeric fields of the *SELECT* clause for those records mentioned in (i) above so that

       a) they are all distinct, and

       b) either all of them are far greater than the biggest constant appeared in the SQL statement or far less than the smallest constant appeared in the SQL statement.

   For example, for the original SQL statement, “*SELECT COUNT*(grade), *COUNT*(salary), *COUNT*(60) FROM STAFF WHERE age > 30”, when an *IRT* mutation operator replaces 30 in the *WHERE* clause by “salary”, the resultant mutant is “*SELECT COUNT*(grade), *COUNT*(salary), *COUNT*(60) FROM STAFF WHERE age > salary”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “age > 30” in the *WHERE* clause so as to set records in the database table such that, they have 29, 30, and 31 in the “age” field. For each of these values in the records, I then set the values for the “salary” and the “grade” fields such that either all of them are far greater than 60 or far less than 30. An example may be (“age”

---

17These BVA values can also be used for other five relational operators like ≤, >, =, < and ≠.
Table 5.5: Dataset that can kill the Mutants

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Relational Operator</th>
<th>&gt;</th>
<th>≥</th>
<th>&lt;</th>
<th>≤</th>
<th>=</th>
<th>≠</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX, MIN</td>
<td>First</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>First</td>
<td>Second</td>
</tr>
<tr>
<td>SUM, AVG</td>
<td>First</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>First</td>
<td>First</td>
</tr>
</tbody>
</table>

= 29, “grade” = 3, “salary” = 5), (30, 6, 8), (31, 10, 12). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “1, 1, 1” whereas the mutant returns “3, 3, 3”. It should be noted that this guideline also works for the each case of COUNT(DISTINCT ... ) because all the values in the related numeric fields are distinct.

Again, for the above SQL statement, when an IRT mutation operator replaces 30 in the WHERE clause by 60, the resultant mutant is “SELECT COUNT(grade), COUNT(salary), COUNT(60) FROM STAFF WHERE age > 60”. The above mentioned dataset (e.g. (“age” = 29, “grade” = 3, “salary” = 5), (30, 6, 8), (31, 10, 12)) can distinguish the mutant from the original SQL statement because the original SQL statement returns “1, 1, 1” whereas the mutant returns “0, 0, 0”.

2. Situation when the SQL statement is involved in an aggregate function other than COUNT() (e.g. MAX(), MIN(), SUM(), AVG()). Instead of having one guideline per aggregate function, I propose a guideline for all these aggregate functions. As the constant can be replaced by another field or constant. In the following, I discuss the guidelines in two separate sections. In the first section, I discuss the guidelines where a constant is replaced by another numeric field. And in the second section, I discuss the guidelines where a constant is replaced by another constant.

**Constant is replaced by another field:** The guideline requires two datasets to be generated in order to kill all mutants. Table 5.5 summarizes which dataset can be used to kill the relevant mutants involving different aggregate functions and relational operators. For example, for the SQL statement O4-1a in Table 5.1, a possible mutant is “SELECT MAX(grade), MIN(salary), MAX(60) FROM STAFF WHERE age > 30”. Table 5.5 indicates that the first dataset can be used to distinguish this mutant. On the other hand, for some aggregate functions, if the original SQL
statement involves a non-equal relational operator (\(\neq\)) in the \texttt{WHERE} clause, the first dataset cannot be used to distinguish the mutant from the original SQL statement. In that case, I need to use the second dataset. In the following I discuss the guidelines to generate these two datasets.

A. Guidelines for the first dataset consists of the following three major steps:

(i) Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values for the IRT-constant, and then set three records in the database table so that they will have one of these values in the IRT-constant involved in the \texttt{WHERE} clause.\footnote{These BVA values can also be used for other five relational operators like \(\leq\), \(>\), \(=\), \(<\) and \(\neq\).}

(ii) Apply the following rules to set the values of other numeric fields for the three records mentioned in (i) above.

(a) For the record whose value of the field in the \texttt{WHERE} clause is greater than the boundary value — For each numeric field other than the field in the \texttt{WHERE} clause, the value of this numeric field must satisfy the following conditions:
   1. it is the maximum value among other values in the same field in other records and
   2. it is at least twice as big as the boundary value

(b) For the record whose value of the field in the \texttt{WHERE} clause is less than the boundary value: For each numeric field other than the field in the \texttt{WHERE} clause, the value of this numeric field must be the minimum value among other values in the same field in other records and

(c) For the record whose value of the field in the \texttt{WHERE} clause is equal to the boundary value — For each numeric field other than the field in the \texttt{WHERE} clause, the value of this numeric field must be greater than that mentioned in (b) above and smaller than the boundary value

For example, for the SQL statement \texttt{O4-1a} in Table 5.1, when an \textit{IRT} mutation operator replaces a numeric constant 30 in the \texttt{WHERE} clause by another numeric field “salary” that appears in the aggregate function of the \texttt{SELECT} clause, the
resultant mutant is “SELECT MAX(grade), MIN(salary), MAX(60) FROM STAFF WHERE age > salary”. In order to distinguish this type of mutant from O4-1a, I need to apply the BVA on the condition “age > 30” in the WHERE clause so as to set the records in the database table such that, they will have 29, 30, and 31 in the field in the WHERE clause (that is “age”). For these records, I then apply the rule in (ii) above on the “grade” and the “salary” fields to set the records like (“age” =29, “grade” = 15, “salary” = 18), (30, 20, 25), and (31, 65, 75). By doing so, one can then distinguish the mutant from O4-1a because O4-1a returns “65, 75, 60” whereas the mutant returns “20, 25, 60”.

B. The guideline for the second dataset is to set same values other than boundary value to all fields in the SQL statement.

For example, one possible mutant of “SELECT MAX(grade), MIN(salary), MIN(60) FROM STAFF WHERE age ≠ 30” is “SELECT MAX(grade), MIN(salary), MIN(60) FROM STAFF WHERE age ≠ salary”. According to the guideline for the second dataset, an example may be (“age” = 15, “grade” = 15, “salary” = 15) and (55, 55, 55). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “15, 15, 60” whereas the mutant returns “null, null, 60”.

In the case of a constant is replaced by another constant: In order to kill all mutants involved in all types of relational operator (>, ≥, <, ≤, =, and ≠), the guidelines requires three datasets. First and second dataset are generated depending on the condition in the WHERE clause. And the third dataset is generated depending on whether the constant C2 in the SELECT clause is greater or less than the constant C1 in the WHERE clause. Table 5.6 summarizes which dataset can be used to kill the relevant mutants involving different aggregate functions and relational operators. In the following, I discuss guidelines for these three datasets in detail.

A. Guideline for the first dataset consists of the following two major steps:

(i) Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values, and then set three records in the
Table 5.6: Dataset that can kill the Mutants

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Relational Operator</th>
<th>First</th>
<th>Second</th>
<th>Third</th>
<th>Third</th>
<th>Third</th>
<th>Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX, MIN, SUM, AVG</td>
<td>C2&gt;C1</td>
<td>First</td>
<td>First</td>
<td>Third</td>
<td>Third</td>
<td>Third</td>
<td>Second</td>
</tr>
<tr>
<td>MAX, MIN, SUM, AVG</td>
<td>C2&lt;C1</td>
<td>Third</td>
<td>Third</td>
<td>First</td>
<td>First</td>
<td>Third</td>
<td>Second</td>
</tr>
</tbody>
</table>

database table so that they will have one of these value in the field involved in the WHERE clause.

(ii) Set the values for the other numeric fields of the SELECT clause for those records mentioned in (i) above so that

- they are all distinct,
- they are different from any constants in the SQL statement and
- they are different from the values in the field involved in the WHERE clause.

For example, for the SQL statement O4-1a in Table 5.1, when an IRT mutation operator replaces a numeric constant 30 in the WHERE clause by 60, the resultant mutant is `SELECT MAX(grade), MIN(salary), MAX(60) FROM STAFF WHERE age > 60`. In order to distinguish this type of mutants from the original SQL statement, I need to apply BVA on the condition “age>30” in the WHERE clause so as to set records in the database table such that they will have 29, 30, and 31 in the “age” field (the field in the WHERE clause). For these records, I then set values for the “grade” and the “salary” fields so that these values are all distinct and different from the values of the field “age” and the constants 60 and 30. An example may be (“age”=29, “grade”=12, “salary” = 100), (30, 18, 150), and (31, 25, 200). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “25, 200, 60” whereas the mutant returns “null, null, null”.

B. The guideline for the second dataset consists of the following two steps:

(i) Set at least two records in the database table so that the values of the numeric field involved in the WHERE clause are equal to the boundary value.
(ii) Set the values of the other numeric fields in the database for those records mentioned in (i) above so that

- they are all distinct,
- they are different from any constants in the SQL statement and
- they are different from the values in the field involved in the `WHERE` clause.

For example, one possible mutant of “SELECT `max(grade)`, `min(salary)`, `max(60)` FROM `STAFF` WHERE `age` ≠ 30” is “SELECT `max(grade)`, `min(salary)`, `max(60)` FROM `STAFF` WHERE `age` ≠ 60”. According to the guidelines for the second dataset, an example may be (“age” = 30, “grade” = 15 “salary” = 100 ) and (30, 25, 200). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “null, null, null” whereas the mutant returns “25, 100, 60”.

C. The guideline for the third dataset consists of the following steps:

(i) Use C2 as the boundary value for BVA to obtain two other values, one for “one-below C2” and one for “one-above C2”, and then set three records in the database table so that they will have one of these values in the field involved in the `WHERE` clause

(ii) Set the values of other numeric fields in the database table for those records mentioned in (i) above so that

- they are all distinct,
- they are different from any constants in the SQL statement and
- they are different from the values in the field involved in the `WHERE` clause.

For example, for the SQL statement “SELECT `max(grade)`, `min(salary)`, `max(60)` FROM `STAFF` WHERE `age` ≤ 30”, when an IRT mutation operator replaces a numeric constant 30 in the `WHERE` clause by 60, the resultant mutant is “SELECT `max(grade)`, `min(salary)`, `max(60)` FROM `STAFF` WHERE `age` ≤ 60”. According to the guidelines for the third dataset, I use 60 as the boundary value to obtain 59 and 61. Then, I set the records in the database table such that
they will have 59, 60, and 61 in the “age” field (the field in the \texttt{WHERE} clause).
For these records, I need to set values for the “grade” and the “salary” fields so that these values are all distinct and different from the values of the field “age” and constants 60 and 30. An example may be (“age”=59, “grade”=10, “salary” = 15), (60, 20, 25), (61, 40, 35). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “40, 15, 60” whereas the mutant returns “20, 15, 60”.

\textbf{G-IRT-m16}: For the IRT-m16 mutant, the \textit{IRT} mutation operator replaces a constant string in the \texttt{WHERE} clause by another non-numeric field or constant string that also appears in the SQL statement. In order to distinguish the mutants from the original SQL statements, two different sets of guidelines are proposed depending on the characteristic of the aggregate functions. In the first set, the guidelines are related to detecting mutants involving the aggregate function \texttt{COUNT()} and in the second set, the guidelines are related to other aggregate functions.

\textbf{A. The situation when the aggregate function COUNT()} is involved in the SQL statement, the guideline consists of the following steps:

(i) Apply boundary value analysis (BVA) on the condition in the \texttt{WHERE} clause to obtain the boundary values, and then set the records in the database table so that they will have one of these values in the field involved in the \texttt{WHERE} clause\footnote{These BVA values can also be used for other five relational operators like \texttt{\leq}, \texttt{\geq}, \texttt{=}, \texttt{<} and \texttt{\neq}}

(ii) Set the values for the other non-numeric fields of the \texttt{SELECT} clause for those records mentioned in (i) above so that

(a) they are all distinct, and

(b) either all of them are far greater than the biggest constant string appeared in the SQL statement or far less than the smallest constant string appeared in the SQL statement.

For example, for the SQL statement, \texttt{“SELECT COUNT(staffid), COUNT(name), COUNT(‘S6’) FROM STAFF WHERE city = ‘C8’”}, when ‘C8’ is replaced by “staffid” the resultant mutant is \texttt{“SELECT COUNT(staffid), COUNT(name), COUNT(‘S6’) FROM}
Table 5.7: Dataset that can kill the Mutants

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Relational Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX, MIN</td>
<td>&gt;, ≥, &lt;, ≤, =, ≠</td>
</tr>
<tr>
<td>First</td>
<td>First</td>
</tr>
<tr>
<td>First</td>
<td>First</td>
</tr>
<tr>
<td>First</td>
<td>Second</td>
</tr>
</tbody>
</table>

STAFF WHERE city = staffid”. In order to distinguish this type of mutant from the original SQL statement, I need to apply BVA on the condition “city = ‘C8’” so as to set records in the database table such that, they have ‘C7’, ‘C8’, and ‘C9’ in the field in the WHERE clause (that is, “city”). For these records, I then set the values for the “staffid” and the “name” fields such that either all of them are far greater than ‘S6’ or far less than ‘C8’. An example may be (“city”= ‘C7’, “staffid” = ‘S7’, “name” = ‘Adam’), (“C8’, ‘S8’, ‘Bill’), (‘C9’, ‘S8’, ‘Alex’). By doing so, one can then distinguish the mutant from the original SQL statement because the original returns “1, 1, 1” whereas the mutant returns “0, 0, 0”.

It should be noted that this guideline also works for the case of COUNT(DISTINCT ...) because all the values in the related non-numeric fields are distinct.

B. The situation when an aggregate function other than COUNT() (i.e. MAX(), MIN()) is involved in the SQL statement. Instead of having one guideline per aggregate function, I propose a guideline for all these aggregate functions. As the constant string in the WHERE clause is replaced by another field or constant, in the following I discuss the guidelines in two separate sections. In the first section, I discuss the guidelines where a constant string is replaced by another non-numeric field. And in the second section, I discuss the guidelines where a constant string is replaced by another constant string.

Situation when a constant string is replaced by another non-numeric field: The guideline requires two datasets to be generated in order to kill all mutants involved in all types of relational operator (> , ≥, <, ≤, = and ≠). Table 5.7 summarizes which dataset can be used to kill the relevant mutants involving different aggregate functions and relational operators. For example, for the SQL statement O4-2b in Table 5.1, a possible mutant is “SELECT MAX(staffid), MIN(name), MAX(‘S6’) FROM STAFF WHERE city = ‘C8’”. Table 5.7 indicates that the first dataset can be used to distinguish this mutant. On the other hand,
if the original SQL statement involves a non-equal relational operator (\(\neq\)) in the WHERE clause, the first dataset cannot be used to distinguish the mutant from the original SQL statement. In that case, I need to use the second dataset. In the following I discuss the guidelines to generate these two datasets.

**A.** Guidelines for the first dataset consists of the following three major steps:

(i) Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values, and then set three records in the database table so that they will have one of these values in the field involved in the WHERE clause\(^{20}\)

(ii) Apply the following rules to set the values of other non-numeric fields for the three records mentioned in (i) above.

(a) For the record whose value of the field involved in the WHERE clause is greater than the boundary value: For each non-numeric field other than the field in the WHERE clause, the value of this non-numeric field must be the maximum value among other values in the same field in other records

(b) For the record whose value of the field involved in the WHERE clause is less than the boundary value: For each non-numeric field other than the field in the WHERE clause, the value of this non-numeric field must be the minimum value among other values in the same field in other records and

(c) For the record whose value of the field involved in the WHERE clause is equal to the boundary value: For each non-numeric field other than the field in the WHERE clause, the value of this non-numeric field must be greater than that mentioned in (b) above and smaller than the boundary value

For example, for the SQL statement O4-2b in Table 5.1, when an IRT mutation operator replaces ‘C8’ in the WHERE clause by “staffid” the resultant mutant is “SELECT MAX(staffid), MIN(name), MAX('S6') FROM STAFF WHERE

\(^{20}\)These BVA values can also be used for other five relational operators like \(\leq\), \(\geq\), =, < and \(\neq\).
city = staffid”. In order to distinguish this type of mutant from the original, I need to apply BVA on the condition “city = ‘C8’” in the \texttt{WHERE} clause so as to set records in the database table such that, they will have ‘C7’, ‘C8’, and ‘C9’ in the field in the \texttt{WHERE} clause (that is “city”). For these records, I then apply the rule in (ii) above on the “staffid” and the “name” field to set the records like (\texttt{city}= ‘C7’, “staffid” = ‘A1’, “name” = ‘Adam’), (‘C8’, ‘B1’, ‘Bill’), (‘C9’, ‘S1’, Tony’). By doing so, one can then distinguish the results of the original from those mutants as the original SQL returns “B1, Bill, S6” whereas the mutant returns “null, null, S6”.

\textbf{B.} The guideline for the second dataset is to set multiple records in the database table so that the values of all fields are not equal to the boundary value and constants involved in the SQL statement.

For example, one possible mutant of \texttt{SELECT MAX(staffid), MIN(name), MAX(‘S6’) FROM STAFF WHERE city \neq ‘C8’} is \texttt{SELECT MAX(staffid), MIN(name), MAX(‘S6’) FROM STAFF WHERE city \neq staffid}. According to the guideline for the second dataset, an example may be (\texttt{city} = ‘C9’, “staffid” = ‘C9’, “name” = ‘C9’), (‘C5’, ‘C5’, ‘C5’). By doing so, one can then distinguish the mutants from the original SQL statement because the original SQL statement returns “C9, C5, S6” whereas the mutant returns “null, null, null”.

\textbf{Constant string is replaced by another constant string:} Similar to G-IRT-m15, in order to kill all mutants involved in all types of relational operator (>, \geq, <, \leq, =, and \neq), the guidelines requires three datasets. First and second dataset are generated depending on the condition in the \texttt{WHERE} clause. And the third dataset is generated depending on whether the constant string K2 in the \texttt{SELECT} clause is greater or less than the constant string K1 in the \texttt{WHERE} clause. Table 5.8 summarizes which dataset can be used to kill the relevant mutants involving different aggregate functions and relational operators. In the following, I discuss guidelines for these three datasets in detail.

\textbf{A.} The guideline for the first dataset consists of the following two major steps:
Table 5.8: Dataset that can kill the Mutants

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Relational Operator</th>
<th>First</th>
<th>First</th>
<th>Third</th>
<th>Third</th>
<th>Third</th>
<th>Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX, MIN</td>
<td>&gt;</td>
<td>First</td>
<td>First</td>
<td>Third</td>
<td>Third</td>
<td>Third</td>
<td>Second</td>
</tr>
<tr>
<td>MAX, MIN</td>
<td>=</td>
<td>Third</td>
<td>Third</td>
<td>First</td>
<td>First</td>
<td>Third</td>
<td>Second</td>
</tr>
</tbody>
</table>

(i) Apply boundary value analysis (BVA) on the condition in the WHERE clause to obtain the boundary values, and then set three records in the database table so that they will have one of these value in the field involved in the WHERE clause.

(ii) Set the values for the other non-numeric fields of the SELECT clause for those records mentioned in (i) above so that:

- they are all distinct,
- they are different from any constant strings in the SQL statement and
- they are different from the values in the field involved in the WHERE clause.

For example, for the SQL statement "SELECT MAX(staffid), MIN(name), MAX('S6') FROM STAFF WHERE city > 'C8' ", when an IRT mutation operator replaces a constant string ‘C8’ in the WHERE clause by ‘S6’, the resultant mutant is “SELECT MAX(staffid), MIN(name), MAX('S6') FROM STAFF WHERE city > 'S6' ". In order to distinguish this type of mutants from the original SQL statement, I need to apply BVA on the condition “city > ‘C8’” in the WHERE clause so as to set records in the database table such that they will have ‘C7’, ‘C8’, and ‘C9’ in the “city” field (the field in the WHERE clause). For these records, I then set values for the “staffid” and the “name” fields so that these values are all distinct and different from the values of the field “city” and the constant strings ‘S6’ and ‘C8’. An example may be (“age”=‘C7’, “staffid”=‘A1’, “name” =‘Alex’), (‘C8’, ‘A2’, ‘Bill’), (‘C9’, ‘A3’, ‘Tony’). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “A3, Tony, S6” whereas the mutant returns “null, null, null”.

B. Guideline for the second dataset consists of the following two steps
(i) Set at least two records in the database table so that the values of the non-numeric field involved in the \texttt{WHERE} clause are equal to the boundary value.

(ii) Set the values of the other non-numeric fields in the database for those records mentioned in (i) above so that

- they are all distinct.
- they are different from any constant strings in the SQL statement and
- they are different from the values in the field involved in the \texttt{WHERE} clause.

For example, one possible mutant of \texttt{``SELECT MAX(staffid), MIN(name), MAX(`S6') FROM STAFF WHERE city = `C8''} is \texttt{``SELECT MAX(staffid), MIN(name), MAX(`S6') FROM STAFF WHERE city = `S6''.} According to the guidelines for the second dataset, an example may be ('age' = 'C8', 'staffid' = 'A1', "name" = 'Alex'), ('C8', 'A2', 'Bill'). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns \texttt{"null, null, null"} whereas the mutant returns \texttt{"A2, Alex, S6"}.

C. Guideline for the third dataset consists of the following steps:

(i) Use K2 as the boundary value for BVA to obtain two other values, one for "one-below K2" and one for "one-above K2", and then set three records in the database table so that they will have one of these values in the field involved in the \texttt{WHERE} clause.

(ii) Set the values of other non-numeric fields in the database for those records mentioned in (i) above so that

- they are all distinct,
- they are different from any constant strings in the SQL statement and
- they are different from the values in the field involved in the \texttt{WHERE} clause.

For example, for the SQL statement \texttt{``SELECT MAX(staffid), MIN(name), MAX( `B6' ) FROM STAFF WHERE city \leq `C8' '}, when an \textit{IRT} mutation operator replaces a constant string 'C8' in the \texttt{WHERE} clause by 'B6', the resultant
mutant is “SELECT MAX(staffid), MIN(name), MAX('B6') FROM STAFF WHERE city ≤ 'B6’”. According to the guidelines for the third dataset, I use ‘B6’ as the boundary value to obtain ‘B5’ and ‘B7’. Then, I set the records in the database table such that they will have ‘B5’, ‘B6’, and ‘B7’ in the “city” field (the field in the WHERE clause). For these records, I need to set values for the “staffid” and the “name” fields so that these values are all distinct and different from the values of the field “city” and constant strings ‘B6’ and ‘C8’. An example may be (“city”=‘B5’, “staffid”=‘A1’, “name” = ‘Adam’), (‘B6’, ‘A2’, ‘Bill’), (‘B7’, ‘A3’, ‘Tony’). By doing so, one can then distinguish the mutant from the original SQL statement because the original SQL statement returns “A3, Adam, B6” whereas the mutant returns “A2, Adam, B6”.

Table 5.9 summarizes all guidelines for 16 different types of IRT mutants that may occur in the SQL statement as discussed in Table 5.1 and Table 5.3.

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IRT mutation operator occurs on the numeric constant in the SELECT clause</td>
</tr>
<tr>
<td>2</td>
<td>IRT mutation operator occurs on the non-numeric constant in the SELECT clause</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>Set the values of the numeric fields so that they are distinct and are not equal to the constants involved in the SQL statement.</td>
<td>(grade = 5, salary = 1500), (8, 3000), (10, 4000)</td>
</tr>
<tr>
<td>G2</td>
<td>Set the values of the numeric fields so that they are distinct and are not equal to the constants involved in the SQL statement.</td>
<td>(“staffid”=‘S1’, “name”=‘Jake’), (‘S2’, ‘David’), (‘S3’, ‘Michael’)</td>
</tr>
</tbody>
</table>
## Chapter 5. Test Case Generation for IRT Operator

Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td><code>COUNT()</code> – IRT mutation operator occurs on the numeric constant in the <code>COUNT()</code> aggregate function</td>
<td>G3-1(a)</td>
<td>If null values are allowed to the fields, (i) Set different number of null values for the numeric fields in the <code>SELECT</code> clause (ii) Set at least one null value in each numeric field, and (iii) For each record, it can only have at most one null value in the numeric fields. Otherwise, the mutants are equivalent to the original SQL statement.</td>
<td>(“grade” = 3, “salary” = 3), (4, 5), (5, null), (null, 7), (null, 8)</td>
</tr>
<tr>
<td>3</td>
<td><code>COUNT (DISTINCT ...)</code> – IRT mutation operator occurs on the numeric constant in the <code>COUNT(DISTINCT ...)</code> aggregate function</td>
<td>G3-1(b)</td>
<td>If null values are allowed, guideline is same as G3-1(a). Otherwise, set the values so that these fields have different number of distinct values.</td>
<td>(“grade” = 1, “salary” = 3), (2, 3), (3, 2)</td>
</tr>
<tr>
<td>3</td>
<td>Other aggregate function – IRT mutation operator occurs on the numeric constant in the aggregate function such as <code>MAX()</code>, <code>MIN()</code>, <code>SUM()</code>, and <code>AVG()</code></td>
<td>G3-2</td>
<td>Set some non-zero distinct values for every numeric field in the database records so that they are far greater or less than those constants already existed in the SQL statement.</td>
<td>(“grade”=100, “salary”=1100), (150, 1200), and (200, 1400)</td>
</tr>
</tbody>
</table>
### Chapter 5. Test Case Generation for IRT Operator

Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>COUNT() – IRT mutation operator occurs on the non-numeric constant in the COUNT() aggregate function</td>
<td>G4-1(a)</td>
<td>Set values for the non-numeric fields in the SELECT clause using the same technique of the guideline G3-1(a)</td>
<td>(“staff”=‘S1’, “name”=‘Jack’), (null, ‘David’), (‘S2’, null), (‘S3’, null)</td>
</tr>
<tr>
<td>4</td>
<td>COUNT(DISTINCT ...) – IRT mutation operator occurs on the non-numeric constant in the COUNT(DISTINCT ...) aggregate function</td>
<td>G4-1(b)</td>
<td>Set values for the non-numeric fields in the SELECT clause using the same technique of guideline G3-1(b)</td>
<td>(‘staff’= ‘S1’, “name” = ‘Jack’), (‘S2’, ‘Jack’), (‘S3’, ‘Adam’)</td>
</tr>
<tr>
<td>4</td>
<td>Other aggregate function – IRT mutation operator occurs on the non-numeric constant in the aggregate function such as MAX() and MIN()</td>
<td>G4-2</td>
<td>Set some non-null distinct values for every non-numeric field in the database records so that they are far greater or less than those constants already existed in the SQL statement.</td>
<td>(“staff”=‘S2’, “name”=‘Adam’), (‘S5’, ‘James’), (‘S10’, ‘David’)</td>
</tr>
<tr>
<td>5</td>
<td>IRT mutation operator occurs on the numeric constant in the SELECT clause and a numeric field appears in the WHERE clause.</td>
<td>G5</td>
<td>(1) Apply BVA rule on the condition in the WHERE clause to obtain and set BVA values in the records of the database, then (2) Apply reverse-BVA rule for other numeric field in the SELECT clause</td>
<td>(“age” = 29, “grade”=42, “salary” = 82), (30, 25, 35), and (31, 28, 18)</td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
<td>Example</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>6</td>
<td><em>IRT</em> mutation operator occurs on numeric constant in the <strong>SELECT</strong> clause, and a non-numeric field appears in the <strong>WHERE</strong> clause</td>
<td>G6</td>
<td>(1) Apply BVA rule on the condition in the <strong>WHERE</strong> clause to obtain and set BVA values in the database records, then (2) Set values in the other fields in the <strong>SELECT</strong> clause those records mentioned in (1) above so that these values are all distinct and not equal to the constant that already exists in the SQL statement.</td>
<td>(“staffid” = ‘S0’, “grade” = 5, “salary” = 1000), (‘S1’, 10, 2000), and (‘S2’, 15, 3000)</td>
</tr>
<tr>
<td>7</td>
<td><em>IRT</em> mutation operator occurs on the non-numeric constant in the <strong>SELECT</strong> clause, and a numeric field appears in the <strong>WHERE</strong> clause</td>
<td>G7</td>
<td>Same as G6</td>
<td>(“grade” = 39, “staffid”=’S1’, “name” = ‘Jack’), (40, ‘S2’, ‘David’), (41, ‘S3’, ‘James’)</td>
</tr>
<tr>
<td>8</td>
<td><em>IRT</em> mutation operator occurs on the non-numeric constant in the <strong>SELECT</strong> clause and a non-numeric field also appears in the <strong>WHERE</strong> clause</td>
<td>G8</td>
<td>(1) Apply BVA rule on the condition in the <strong>WHERE</strong> clause to obtain and set BVA values in the database records, (2) Set values for the other fields in the <strong>SELECT</strong> clause for those records mentioned in (1) above so that they are (i) all distinct, (ii) different from BVA values, and (iii) not equal to the constants that already exist in the SQL statement.</td>
<td>(“city” = ‘C7’, “staffid” = ‘S0’, “name” = ‘Adam’), (‘C8’, ‘S1’, ‘David’), (‘C9’,’S2’, ‘Tom’)</td>
</tr>
</tbody>
</table>
Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td><em>IRT</em> operator occurs on the numeric constant in the <em>WHERE</em> clause and numeric fields also appear in the <em>SELECT</em> clause.</td>
<td>G9</td>
<td>(1) Apply BVA on the condition in the <em>WHERE</em> clause to obtain and set BVA values in the database records, (2) For those records mentioned in (1) above set the values for other numeric fields so that (i) they are all distinct, (ii) either all of them are far greater than or far less than the boundary value and the constant appeared in the SQL statement.</td>
<td>(“age” = 29 “grade” = 2, “salary” = 8), (30, 4, 9), (31, 6, 10)</td>
</tr>
<tr>
<td>10</td>
<td><em>IRT</em> operator occurs on the non-numeric constant in the <em>WHERE</em> clause and non-numeric fields also appear in the <em>SELECT</em> clause</td>
<td>G10</td>
<td>Set the values for the non-numeric fields in the database records using the same technique of G9.</td>
<td>(“city” = ‘C7’, “staffid” = ‘S1’, “name” = ‘Adam’), (“C8”, “S2”, “David”), and (“C9”, “S3”, “Jack”)</td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
<td>Example</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------</td>
<td>----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>11</td>
<td>( \text{COUNT()} ) – IRT occurs on the numeric constant in the ( \text{COUNT()} ) aggregate function in the ( \text{SELECT} ) clause and a numeric field also appears in the ( \text{WHERE} ) clause</td>
<td>G11-1(a)</td>
<td>If null values are allowed (1) Apply BVA on the condition in the ( \text{WHERE} ) clause to obtain and set the boundary values in the database records, (2) For those records mentioned in (1) above, (i) set different number of null values for the numeric fields in the ( \text{SELECT} ) clause (ii) set at least one null value in each numeric field, and (iii) for each record, it can only have at most one null value in the numeric fields. Otherwise, the mutants are equivalent to the original SQL statement.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{COUNT(DISTINCT ...)} ) – IRT occurs on the numeric constant in the ( \text{COUNT(DISTINCT ...)} ) aggregate function in the ( \text{SELECT} ) clause and a numeric field also appears in the ( \text{WHERE} ) clause</td>
<td>G11-1(b)</td>
<td>If null values are allowed guideline is same as G11-1(a). Otherwise, (1) Apply BVA on the condition in the ( \text{WHERE} ) clause to obtain and set BVA values in the database records, (2) For those records mentioned in (1) above, set different number of distinct values for the other numeric fields.</td>
<td></td>
</tr>
</tbody>
</table>

\( \text{G11-1(a)} \)

\( \text{G11-1(b)} \)
### Type: Other aggregate function – \textit{IRT} occurs on the numeric constant in the \texttt{MAX()}, \texttt{MIN()}, ... aggregate function in the \texttt{SELECT} clause and a numeric field also appears in the \texttt{WHERE} clause

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>G11-2</td>
<td>(1) Apply BVA on the condition in the \texttt{WHERE} clause to obtain and set BVA values in the database records, (2) Set some non-zero distinct values for every numeric field for those records mentioned in (1) above.</td>
<td>(\texttt{age} = 29, \texttt{grade} = 100, \texttt{salary} = 500), (30, 200, 1000), and (31, 300, 1500)</td>
<td></td>
</tr>
</tbody>
</table>

### Type: \texttt{COUNT()} – \textit{IRT} occurs in the numeric constant in the \texttt{SELECT} clause and a non-numeric field appears in the \texttt{WHERE} clause

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>G12-1(a)</td>
<td>When \textit{null} values are allowed, (1) Apply BVA on the condition in the \texttt{WHERE} clause to obtain and set BVA values in the database records, (2) For those records mentioned in (1) above, (i) set different number of \textit{null} values to different numeric fields, (ii) set at least one \textit{null} value in each numeric field, and (iii) for each record, it can only have at most one \textit{null} value in its numeric field</td>
<td>(\texttt{staffid} = ‘S0’, \texttt{grade} = 2, \texttt{salary} = \texttt{null}), (‘S0’, 4, \texttt{null}), (‘S0’, \texttt{null}, 3), (‘S1’, 6, \texttt{null}), (‘S1’ \textit{null}, 5), (‘S1’, \texttt{null}, 7), (‘S2’, 8, \texttt{null}), (‘S2’, 10, \texttt{null}), (‘S2’, \texttt{null}, 9)</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>12</td>
<td>COUNT(DISTINCT ...) – IRT occurs in the numeric constant in the SELECT clause and a non-numeric field appears in the WHERE clause</td>
<td>G12-1(b)</td>
<td>When the field allows null values guideline is same as G12-1(a). Otherwise, (1) Apply BVA on the condition in the WHERE clause to obtain and set BVA values in the records in the database table (2) Set different number of distinct values in other fields of the SELECT clause for those records mentioned in (1) above.</td>
</tr>
<tr>
<td>12</td>
<td>Other aggregate function – IRT occurs in the numeric constant in the SELECT clause and a non-numeric field appears in the WHERE clause</td>
<td>G12-2</td>
<td>(1) Apply BVA on the condition in the WHERE clause to obtain and set BVA values in the database records (2) For those records mentioned in (1) above, set some non-zero distinct values for every numeric field so that these values are far greater or less than those constants already existed in the SQL statement.</td>
</tr>
</tbody>
</table>
### Chapter 5. Test Case Generation for IRT Operator

Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Other aggregate function – IRT occurs in the non-numeric constant in the aggregate function of the SELECT clause and a numeric field appears in the WHERE clause</td>
<td>G13-2</td>
<td>Set the numeric and non-numeric values using the similar technique of G12-2</td>
<td>(“age” = 29, “staffid” =‘S0’, “name” =‘Adam’), (30, ‘S1’, ‘Bill’), and (31, ‘S2’, ‘David’)</td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
<td>Example</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>14</td>
<td><strong><code>COUNT(DISTINCT ...)</code> – <code>IRT</code> occurs on the non-numeric constant in the <code>COUNT(DISTINCT ...)</code> aggregate function in the <code>SELECT</code> clause and a non-numeric field also appears in the <code>WHERE</code> clause</strong></td>
<td>G14-1(b)</td>
<td>Set the values of the non-numeric fields using the similar technique of G11-1(b)</td>
<td>(&quot;city&quot; = 'C7', &quot;staffid&quot; = 'S1', &quot;name&quot; = 'Jack'), ('C7', 'S2', 'Alex'), ('C7', 'S3', 'Alex'), ('C8', 'S3', 'Mike'), ('C8', 'S3', 'Peter'), ('C9', 'S4', 'James'), ('C9', 'S4', 'David'), ('C9', 'S5', 'David'), and ('C9', 'S5', 'Adam')</td>
</tr>
<tr>
<td></td>
<td><strong>Other aggregate function – <code>IRT</code> occurs on the non-numeric constant in the <code>MAX()</code> and <code>MIN()</code> aggregate functions in the <code>SELECT</code> clause and a non-numeric field also appears in the <code>WHERE</code> clause</strong></td>
<td>G14-2</td>
<td>Set the values of the non-numeric fields using the similar technique of G11-2</td>
<td>(&quot;city&quot; = 'C7', &quot;staffid&quot; = 'A1', &quot;name&quot; = 'Adam'), ('C8', 'A2', 'Bill'), and ('C9', 'A3', 'Alex')</td>
</tr>
<tr>
<td>15</td>
<td><strong><code>COUNT()</code> – <code>IRT</code> mutation operator replaces numeric constant in the <code>WHERE</code> clause</strong></td>
<td>G15-1</td>
<td>Same as G11-1(a)</td>
<td>(&quot;age&quot; = 29, &quot;grade&quot; = 3, &quot;salary&quot; = 5), (30, 6, 8), (31, 10, 12)</td>
</tr>
</tbody>
</table>
### Other aggregate function – IRT

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Other aggregate function – IRT</td>
<td>G15-2(a)</td>
<td><strong>Replaced by field</strong>: Prepare two datasets as follows: <strong>Dataset1.</strong> (1) Apply BVA rule on the <em>WHERE</em> condition to obtain and set the BVA values in the records of the dataset. (2) Apply 3 rules for other numeric fields for three records mentioned in (1) above so that (i) the record whose value of the field in the <em>WHERE</em> clause is less than the boundary value, other field has (a) the maximum value among other records in the same field, (b) the value at least twice as big as the boundary value. (ii) the record whose value of the field in the <em>WHERE</em> clause is greater than than the boundary value, the other field has the minimum value among other records in the same field and (iii) the record whose value of the field in the <em>WHERE</em> clause is equal to boundary value, other field has value greater than the value mentioned in (ii) above and smaller than boundary value. <strong>Dataset2.</strong> Set same values other than the boundary value to all fields in the SQL statement.</td>
<td>First dataset: (“age” = 29, “grade” = 15, “salary” = 18), (30, 20, 25), and (31, 65, 75); Second dataset: (“age” = 15, “grade” = 15, “salary” = 15) and (55, 55, 55)</td>
</tr>
<tr>
<td>Type</td>
<td>Scenario</td>
<td>Guideline</td>
<td>Guideline Description</td>
<td>Example</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-----------</td>
<td>----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>15</td>
<td>Other aggregate function – IRT mutation operator replaces a numeric constant in the WHERE clause by another constant.</td>
<td>G15-2(b)</td>
<td><strong>Replaced by constant:</strong> Prepare three datasets as follows. <strong>Dataset1.</strong> (1) Apply BVA on the condition in the WHERE clause to obtain and set BVA values in the database records, (2) For those values mentioned in (i) above, set the values so that they are (a) all distinct, (b) different from any constants in the SQL statement and (c) different from BVA values. <strong>Dataset2.</strong> (1) Set at least two records in the database so that numeric fields involved in the WHERE clause are equal to boundary, (2) Set the values of the other numeric field for those records mentioned in (1) above so that they are (a) all distinct, (b) different from any constants in the SQL statement, and (c) different from BVA values. <strong>Dataset3.</strong> (1) Use the constant involved in SELECT clause to obtain BVA values for the field in WHERE clause and (2) For those values mentioned in (1) above set some different, distinct, and other than BVA values in the other fields in the SELECT clause.</td>
<td>First dataset: (“age”=29, “grade”=12, “salary” = 100), (30, 18, 150), and (31, 25, 200); Second dataset: (“age” = 30, “grade”= 15 “salary” = 100 ) and (30, 25, 200); Third Dataset: (“age”=59, “grade”=10, “salary” = 15), (60, 20, 25), (61, 40, 35)</td>
</tr>
</tbody>
</table>
## Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Scenario</th>
<th>Guideline</th>
<th>Guideline Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>IRT</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mutation operator replaces a non-numeric constant in the WHERE clause</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CREATE TABLE VTABLE {
    COL1 INTEGER,
    COL2 INTEGER,
    COL3 INTEGER,
    COL4 INTEGER,
    COL5 INTEGER
}

Figure 5.1: Table Schema VTABLE

5.4 Experimental Study

5.4.1 Subject SQL Statements

In this experiment, I use SQL statements found in NIST [79]. Some of them also appear in the subject SQL statements used in the experiment of ABS (Chapter 3, Section 3.5) and IRC (Chapter 4, Section 4.4) mutation operators. Among these NIST SQL statements, 37 of them have constants in it. And these 37 statements are used as the original SQL statements for this experiment.

5.4.2 Mutant Generation

Similar to experiment on IRC mutation operator, I generate IRT mutants using the SQL-Mutation tool [29]. This time, I also do not discover any extra mutants as compared to ABS experiment. There are some SQL statements for which more than one types of mutant are generated from the same SQL statement depending on the actual IRT mutation operator occurred in the SELECT or in the WHERE clause. One example of such SQL statement is \( \text{SELECT COL1, (COL3*COL2/COL1 - COL2 + 10) FROM VTABLE WHERE COL1 > 0} \) (table schema in Figure 5.1). When an IRT mutation operator replaces 10 in the SELECT clause by other fields (col1, col2, ...) or constant (0), a mutant of type IRT-m5 is generated. Again for the same SQL statement, when an IRT mutation operator replaces 0 in the WHERE clause by other fields (col1, col2, ...) or constant (10), a mutant of type IRT-m9 is generated. In this experiment, I generate a total of 174 mutants. Table 5.10 shows the distribution of the mutants for the various types of IRT mutation operator.
Table 5.10: Number of Mutants for IRT

<table>
<thead>
<tr>
<th>Type</th>
<th># of Original SQL</th>
<th>COUNT()</th>
<th>Other AGGR</th>
<th>Generated</th>
<th>Equivalent</th>
<th>Non-Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRT-m1</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m2</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m3</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m4</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m5</td>
<td>1</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>IRT-m6</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m7</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m8</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m9</td>
<td>8*</td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>IRT-m10</td>
<td>21</td>
<td></td>
<td></td>
<td>65</td>
<td>4</td>
<td>61</td>
</tr>
<tr>
<td>IRT-m11</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m12</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m13</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m14</td>
<td>n/a</td>
<td></td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m15</td>
<td>7</td>
<td>COUNT()</td>
<td></td>
<td>63</td>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Other AGGR</td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRT-m16</td>
<td>1</td>
<td>COUNT()</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Other AGGR</td>
<td></td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>38</td>
<td>174</td>
<td>9</td>
<td>165</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* One of the original SQL statements was the same as the one used to generate the mutants in ‘IRT-m5’

Among these 174 mutants, there are 9 mutants that are equivalent to their original SQL statements. Therefore, there are 165 non-equivalent mutants for this experiment. It also shows that for a number of types of IRT mutants there is no mutant. This happens due to unavailability of appropriate original SQL statements.

5.4.3 Test Dataset Generation and Data Collection

For the test dataset generation, I apply similar procedures as described in Section 4.4.3. The only difference is I use the test case generation guidelines as described in Section 5.3. I then execute the test dataset against all 165 non-equivalent mutants. Next, I check whether the test dataset can kill the mutants from its original SQL statement or not. I count the number of killed mutants and finally calculate the mutation score with the number of non-equivalent mutants.
Table 5.11: Number of Killed and Alive Mutants for \textit{IRT} (Only shows the available mutants)

<table>
<thead>
<tr>
<th>Mutant Type</th>
<th># of Mutants</th>
<th>Non-Equivalent</th>
<th>Killed</th>
<th>Alive</th>
<th>Mutation Score(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRT-m5</td>
<td>5</td>
<td>5</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRT-m9</td>
<td>40</td>
<td>40</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRT-m10</td>
<td>61</td>
<td>61</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRT-m15</td>
<td>58</td>
<td>58</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>IRT-m16</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>165</td>
<td>165</td>
<td>-</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

5.4.4 Results Analysis

Table 5.11 shows the number of killed mutants against their corresponding number of non-equivalent mutants. It also illustrates the mutation score for different types of the \textit{IRT} mutants. As a result, from the Table 5.11, it is clear that for all available types of the \textit{IRT} mutants, the generated test datasets kill 100\% of non-equivalent mutants.

5.5 Threads and Validity

The \textit{external validity} of this experiment is the selection of the SQL statements. I have selected 37 original SQL statements from the NIST. Although there is a large number of SQL queries in NIST, not all these SQL statements have the same features that I have proposed. For instance, in this experiment, SQL statements with join, inner join, outer join, sub-query, and SQL statements with ‘group by’ keyword are avoided. As a result, I selected only 37 original SQL statements. These 37 SQL statements cover 5 scenarios among 16 possible scenarios. Because of the small number of original queries the number of the \textit{IRT} mutants is also small. Since the results of this experiment enable me to generalize them across different database applications, it would be useful to select more SQL queries so that each scenario can incorporate some mutants.

The \textit{internal validity} of this experiment is the selection of a test suite. In some cases the test dataset generated through my guidelines contains null values, sometimes it needs to set multiple values against the same values. Due to database constraints if test dataset
can not meet corresponding required criteria, many mutants will be alive. As a result the mutation score may decrease which may lead to weakness in my proposed guidelines.

The *construct validity* of this experiment is that some mutants may be missed by the SQLMutation tool for my proposed guidelines for the *IRT* mutation operator. For a wide range of original SQL statements if the number of missed mutants generated by the tool is unknown, the result may not be exact.

### 5.6 Summary

- Sixteen different types of mutants with altogether thirty two variants are identified from four forms of original SQL statements.

- For each of these variants, one guideline has been proposed to generate one test dataset. But in some cases for different complex variant, proper guidelines are provided to generate two or more datasets.

- An experiment has been performed to verify the effectiveness of the guidelines. In this experiment all non-equivalent mutants are killed.
Chapter 6

Conclusion

6.1 Summary and Contributions

This study focuses on test case generation strategy for three types of mutation operators-
(i) absolute value mutation operator (\textit{ABS}), (ii) identifier column replacement (\textit{IRC}) and
(iii) identifier constant replacement (\textit{IRT}). The following summarizes my contributions in
this thesis:

1. **Analyzing and proposing types of mutants**: In order to generate the guidelines,
   I analyze and propose all different types of mutant that can be created by each of
   the mutation operators. I outline a set of criteria for SQL statements to generate
   different types of mutants. These criteria includes the answers of the following three
   questions: 1) does the SQL statement have a \texttt{WHERE} clause, 2) does it have
   aggregate functions and 3) where does the mutation operator occur - in \texttt{SELECT}
   clause or in \texttt{WHERE} clause?

2. **Generation of guidelines**: For each type of mutant there is one or more variants
   depending on the various scenarios of the mutant. For each variant I propose one
   guideline.
• For absolute value (ABS) mutation operator (Chapter 3), there are 6 types of mutants with altogether 11 variants. There is one guideline for each type of these variants.

• Again for identifier column replacement (IRC) mutation operator (Chapter 4), there are 24 types of mutants with altogether 44 variants. For example if the SQL statements are involved in aggregate functions, the guidelines are described in major two scenarios- (i) SQL statements with \texttt{COUNT()} and (ii) SQL statements with other aggregate functions. Again for each scenario there are two variants - (a) SQL statements with only aggregated function and SQL statement with aggregate function along with the \texttt{DISTINCT} keyword.

• For identifier column replacement (IRT) mutation operator (Chapter 5), there are 16 types of mutants. Many of these types have multiple variants. In total there are 32 variants. For each of these variants, one guideline is proposed.

3. Experimental Evaluation of the Guidelines: Three experiments are performed to measure the effectiveness of the proposed guidelines for these three mutation operators. The subject SQL statements are selected from NIST [79]. Different number of SQL statements are selected for different mutation operators. For example, SQL statements with numeric fields are selected for the experiment of ABS mutation operator. Again, SQL statements with constants are selected for the experiment of IRT mutation operator. For ABS, there were about 67 non-equivalent mutants from 19 original queries. For IRC there is about 226 mutants from 41 original SQL statements and 165 non-equivalent mutants are generated from 37 original SQL statements in the experiment of the IRT mutation operator. Test dataset generated through my corresponding guidelines kills all non-equivalent mutants.

6.2 Future Works

• This thesis considers only first order mutant\textsuperscript{1}. It does not include higher order mutants [14, 81]. For example, for the SQL statement, “\texttt{SELECT salary FROM STAFF}

\textsuperscript{1}the mutant that is generated by making a single change of original SQL statement
WHERE grade > 40”, when \( ABS() \) mutation operator is applied in the \texttt{SELECT} clause, the resultant mutant is \texttt{“SELECT ABS(salary) FROM staff WHERE grade > 40”}. This is an example of first order mutant. However, according to the concept of higher order mutant, it is possible to apply \texttt{ABS} both in \texttt{SELECT} and \texttt{WHERE} clause to generate the mutants. Such an example of above stated original SQL query is \texttt{“SELECT ABS(salary) FROM STAFF WHERE ABS(grade) > 40”}. Similar to \texttt{ABS}, in some cases higher order mutants are possible for \texttt{IRC} and \texttt{IRT} mutation operators. But in this thesis, these types of mutants are avoided. In future, guidelines can be proposed for these three mutation operators to cover higher order mutants.

- Again, in this thesis, some of the SQL features are not considered. For example, I have not considered the SQL statements with ‘\textit{group by}’ to avoid complex multiple scenarios. With these scenarios the guidelines would be more complex. These complex guidelines can be proposed in future.

There are some other SQL features such as join, sub-query, multiple conditions in \texttt{WHERE} clause that are not considered as well in this thesis. In future these features could be considered in generating guidelines.

In this thesis, I consider numeric and non-numeric fields. Only numeric type of field is accounted for generating guidelines for \texttt{ABS} mutation operator because of the character of \texttt{ABS()}. And for \texttt{IRC} and \texttt{IRT} both numeric and non-numeric types of fields are considered. But there are some other data types such as data types related to date functions (e.g. date, datetime etc.) are not considered for \texttt{IRC} or \texttt{IRT}. In future these data types also could be investigated for generating guidelines for the mutants by \texttt{IRC} and \texttt{IRT} mutation operators.

- This study only focuses on proposing test case generation guidelines for three mutation operators. In future, these guidelines may be used in developing a test case generation tool for testing SQL statements.

- It is possible to investigate testing of SQL statements using a model-based approach in the future.
Bibliography


