Effectual Mutation Analysis of Relational Database using Minimal Schemata with Parallelization

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Technology

in

Computer Science and Applications

Submitted By

Heena Gupta

(601303014)

Under the supervision of:

Ms. Sunita Garhwal

Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

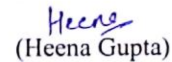
PATIALA – 147004

July 2015

CERTIFICATE

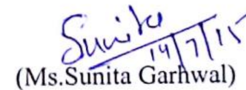
I hereby certify that the work which is being presented in the thesis entitled, "***Effectual Mutation Analysis of Relational Database using Minimal Schemata with Parallelization***", in partial fulfillment of the requirements for the award of degree of Master of Technology in **Computer Science and Application** submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Ms. Sunita Garhwal** and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.



(Heena Gupta)

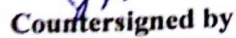
This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.



(Ms.Sunita Garhwal)

Assistant Professor

Computer Science and Engineering Department



Countersigned by

(Dr. Deepak Garg)

Head

Computer Science and Engineering Department

Thapar University

Patiala

(Dr. S. S. Bhatia)

Dean (Academic Affairs)

Thapar University

Patiala

Acknowledgement

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life.

This work would not have been possible without the guidance and encouragement of my supervisor Ms. Sunita Garhwal. I thank my supervisor for her time, patience, and valuable comments.

I am equally thankful to the entire faculty and staff members for their direct and indirect help, and cooperation which made my stay in Thapar University memorable.

Last but not least, I would like to thank my parents and friends for their wonderful love and encouragement, without their support none of this would have been possible.

Heena

Heena Gupta.

Abstract

Today, testing is the most challenging and dominating activity used in every field, therefore, improvement is necessary and efficient with respect to time. Software testing allows the programmer to determine the quality of the software. Mutation testing is the branch of software testing, is a fault-based testing technique that measures the quality of test cases. Faults are injected using a pre-defined set of mutation operators into the program.

Mutation analysis is a successful approach to survey the nature of input values and test cases. Yet, since this strategy requires the execution of numerous mutants, it frequently acquires a generous computational expense and time. This thesis targets the issue of time, and effort that is needed for generating and executing a large amount of mutants. A new algorithm, i.e. *Minimal Schemata with parallelization* is developed which will reduce the time to some extend as compared to the previous algorithms.

This thesis is organized into five chapters. Chapter1 describes the basic concept of the software testing and mutation testing and its types. Chapter2 describes literature survey which has been done during literature survey. Chapter3 describes the motivation behind this, discusses problem statement and its objectives. Chapter4 explains all the results obtained from the algorithm. Chapter5 summarizes the conclusions drawn from the work done along with the directions regarding future scope.

Table of Contents

CERTIFICATE i

Acknowledgement ii

Abstract iii

Table of Contents iv

List of Figures vii

List of Tables x

Chapter1. Introduction

1.1 Software Testing…………….…………………………………………… 1

1.2 Testing Concept………………………………………………………… 2

1.3 Testing Process…………………………………………………………. 2

1.3.1 Test Planning……………………………………………………. 2

1.3.2 Test Design….. 2

1.3.3 Test Cases……………..……………………………………............. 2

1.3.4 Test Execution………………………..………………………… 3

1.3.5 Test Summary Report….……………………………………..…. 3

1.4 Types of Testing..……………………………………………………… 3

1.4.1 Black-Box Testing…… 3

1.4.2 White-Box Testing…… 3

1.4.3 Grey-Box Testing…… 3

1.5 Mutation Testing…… 4

1.5.1 History…… 5

1.5.2 Example…… 5

1.6 Mutation Testing Process…… 6

1.7 Test Data Conditions for Killing Mutants…… 7

1.8 Kinds of Mutation Testing…… 7

1.8.1 Value Mutations…… 7

1.8.2 Decision Mutations…… 7

1.8.3 Statement Mutations…… 7

1.9 Types of Mutants…… 8

1.9.1 Killed Mutant…… 8

1.9.2 Alive/Live Mutant…… 8

1.9.3 Equivalent Mutant…… 8

1.10 Mutation Score…… 8

1.11 Operators…… 8

1.11.1 Method Level Operators…… 9

1.11.1 Class Level Operators…… 11

Chapter 2. Literature Survey

2.1 Mutation Testing for Database Application…………………………… 13

2.1.1 Architecture of Database Applications…………………………. 13

2.2.2 Problems in Testing Database Applications…………………….. 14

2.2.3 SQL Mutation Operators………………………………………. 14

2.2.4 JDAMA……………………………………………………… 15

2.2.5 MutaGen……………………………………………………… 15

2.2.6 Relational Constraint Solver…………………………………… 15

2.2 Mutation Testing on Relational Database……………………………… 16

2.2.1 SynConSMutate System……………………………………….. 16

2.2.2 Search Based Strategy…………………………………………. 17

2.3 Schema Testing and its need…………………………………………… 18

2.4 Search Based Generation of DB Table………………………………… 19

2.4.1 Abstract Representation of Schemas……………………………… 20

2.4.2 Mutation Analysis of Schemas…………………………………… 20

2.5 Mutant Schemata…………………………………………………….. 22

2.5.1 Full Schemata………………………………………………….. 22

2.5.2 Minimal Schemata……………………………………………….. 23

2.6 Parallelization………………………………………………………… 26

Chapter 3. Problem Statement

3.1 Problem Statement………………………………………………………… 27

3.2 Objectives…………………………………………………………….. 27

Chapter 4. Proposed Work

4.1 Approach used for Mutation Testing……..…………………………… 28

4.2 Steps for Mutation Analysis…………………………………………… 30

4.3 Proposed Technique…………………………………………………. 30

4.4 Proposed Algorithm…………………………………………………. 31

4.5 Experimental Results………………………………………………… 32

4.6 Result Analysis………………………………………………………. 37

Chapter 5. Conclusion and Future Scope

5.1 Conclusion……………………………….………………………… 41

5.2 Future Scope……………………………………………………….. 41

References……………………………………………………………... 42

Publication………………………………………………………………… 57

Video Reference………………………………………………………….. 48

List of Figures

Figure 1.1	V&V Process…………………………………………………………	1
Figure 1.2	Testing Process……………………………………………………….	3
Figure 1.3	Program Mutants……………………………………………………..	4
Figure 1.4	Mutation Process…………………………………………………….	6
Figure 2.1	Architecture of Database Application……………………………	13
Figure 2.2	SchemaAnalyst……………………………………………………….	20
Figure 2.3	Relationship between approaches…………………………………	22
Figure 2.4	An example of dependencies between tables using approaches for database UnixUsage……………………………………………….	25
Figure 4.1	Mutation Testing Flow……………………………………………	29
Figure 4.2	Main Window of GUI…………………………………………….	34
Figure 4.3	Next Window of GUI, which consist of information related to case study chosen……………………………………………………………..	35
Figure 4.4	Mutant Created during Full Schemata and Minimal Schemata with case study chosen…………………………………………………..	36
Figure 4.5	Final Results for MSP approach for case study "JWhoisserver"…	37
Figure 4.6	Jwhoisserver analysis time (in sec) displaying average duration of	38

time taken by each approach……………………………………….

Figure 4.7	Office analysis time (in sec) displaying average duration of time taken by each approach…………………………………………….	38
Figure 4.8	NorthWind analysis time (in sec) displaying average duration of time taken by each approach………………………………………	39
Figure 4.9	University analysis time (in sec) displaying average duration of time taken by each approach………………………………………	39
Figure 4.10	Employee analysis time (in sec) displaying average duration of time taken by each approach………………………………………	39
Figure 4.11	Baseball analysis time (in sec) displaying average duration of time taken by each approach…………………………………………….	39
Figure 4.12	UnixUsage analysis time (in sec) displaying average duration of time taken by each approach………………………………………	40

List of Tables

Table 1.1	Method Level Mutation Operators…………………………	10
Table 1.2	Class Level Mutation Operators……………………………	11
Table 2.1	An example of a schema…………………………………….	18
Table 2.2	Full Schemata mutation analysis algorithm…………………	23
Table 2.3	Minimal Schemata mutation analysis algorithm……………	24
Table 2.4	Just-in-Time mutation analysis algorithm (FSP)……………	26
Table 4.1	Minimal Schemata with parallelization mutation analysis algorithm…………………………………………………..	31
Table 4.2	Data structures used for experimental study………………	33
Table 4.3	Summary of tables and Test Suites i.e. Insert statements….	33
Table 4.4	Mutation analysis time for all case studies…………………	37

Chapter1: Introduction

This chapter includes a basic introduction of software testing, testing process, and its types. It also consists of a brief introduction of the mutation testing, types, and operators of it with examples.

1.1 Software Testing

Testing is the procedure of inspecting programming or its segments so as to recognize any gaps, errors, or missing requirements with respect to the actual requirements [1]. It is a practice that executes the program with the motive of finding errors [2, 36]. It is a V&V process (i.e. Verification and Validation) [5].Verification is checking whether the product/item is error free or not. It is basically an internal process which includes the evaluation of the software requirements, specifications, and conditions applied to it. It is the procedure of assessing intermediate work to check is we progressing nicely or not of making software while approval is watching that the product/item meets the client necessity or not. It also checks whether software is fit, ready for use and will satisfy the business requirements.

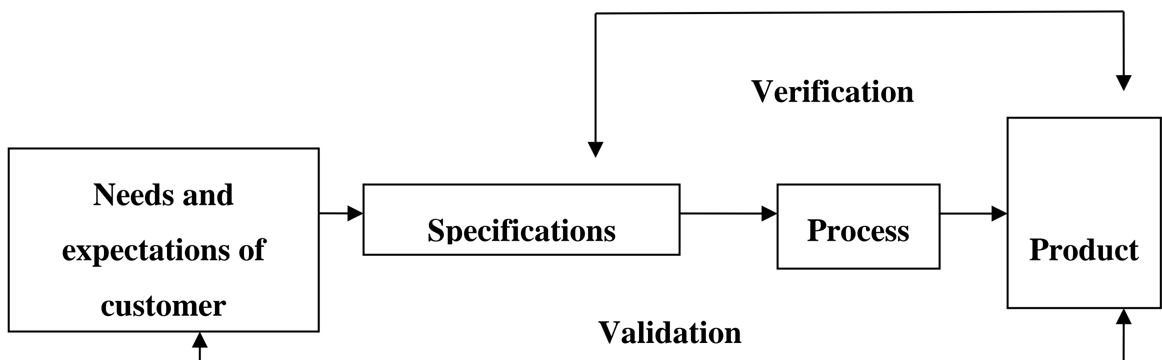


Figure1.1: V&V Process [5]

1.2 Testing Concept

A clue to the correctness perspective of software quality is in the notion of error, fault, and failure. The error refers to the disparity between a processed, watched, or measured quality/value and the genuine, specified, or hypothetically right value. The error is basically used to refer to human activity that outcome in programming containing an imperfection [2, 3]. The fault is a false, wrong step, procedure or information, a definition of a product. A fault is the fundamental purpose behind programming breakdown and is basically synonymous with the regularly utilized term bug. Fault refers to a basic condition inside of the programming that causes an inability to happen [2, 4]. Failure is the incapability of a system or part to perform an ordered function as per its specifications. A product failure happens if the performance of the product is different from the specified performance. Failure may be brought on by functional variables [2].

1.3 Testing Process

There are many activities involved in the process of software testing. All these activities are handled sequentially. In this, faults were fixed at the time of testing. The life cycle of the testing includes various phases, i.e. test planning, test design, and test execution [2, 13].

1.3.1 Test Planning: A software test plan is a document elaborating scope, resources, agenda, objectives, schedule, and approach of testing process. It also identifies the item, and characteristics of an item, which is to be tested or not.

1.3.2 Test Design: It is the process of creating test cases for testing software. It requires the knowledge of the software, the functionality being tested, and testing techniques. This document comprises of input and output requirements and other useful things required.

1.3.3 Test Cases: It is a set of constraints that help the testers to determine whether the product/software system is working as it was originally. Test cases are also known as test scripts, and collectively known as test suites.

1.3.4 Test Execution: It is the process of executing test cases manually or in an automated way, listing test results, comparing results with actual results, verifying the results and testing again the fixed errors/bugs.

1.3.5 Test Summary Report: The test summary report is a document which comprises of analysis of test results, and summary of all the activities. It will be different for different types of testing.

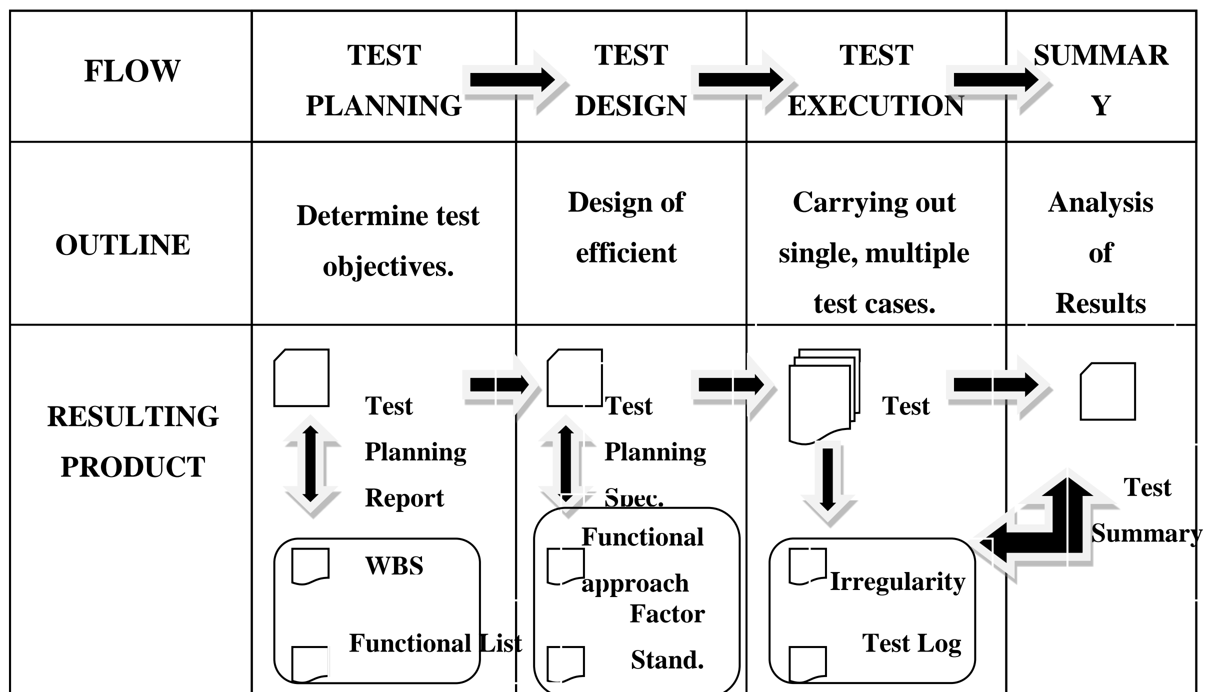


Figure1.2: Testing Process [2]

1.4 Types of Testing

Testing methods can be ordered by root of data that is inferring test information and standard to compute the adequacy of test suite.

1.4.1 Black-Box Testing/Behavioral Testing: It is a type of software testing that examines the software without having any internal knowledge of the software. Black-box testing includes boundary value analysis, equivalence partitioning, and many more. Testers and developers are independent of each other so the test is balanced [5, 14].

1.4.2 White-Box Testing/Open-Box Testing: It is similar to the glass-box testing. It is the detailed examination of the internal structure and working on the code. White-Box testing includes mutation testing, path testing, and many more [5, 14].

1.4.3 Grey-Box Testing: Grey-Box testing is same as translucent testing. It is a technique to test the software with having little internal knowledge. It can be seen as union of white-box and black-box testing. Tester proposes the test cases depending upon the partial information. It includes matrix testing, regression testing, pattern testing, and many more [5, 14].

1.5 Mutation Testing

Mutation testing is a type of fault based white box testing [6]. It is used to test the efficiency of the test cases that are taken to examine the program, not like others that focuses on accurate functionality of the program. It is also known as structural testing technique, which focuses on the structure of the code to direct the testing process.

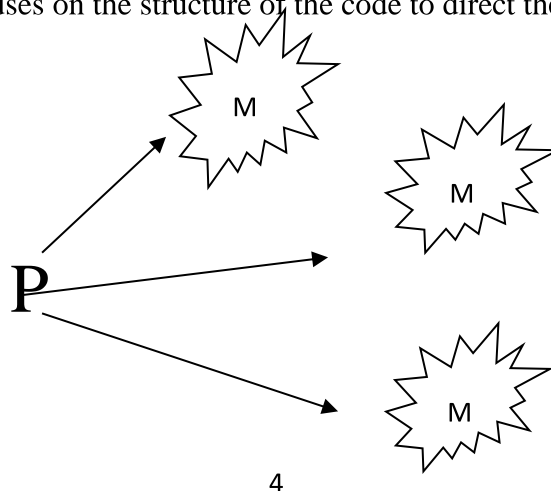


Figure1.3: Program Mutants

Actually, it is the process of recreating the code in different ways in order to remove redundancies in the code, as these uncertainties may cause failures if not fixed. Mutation testing involves modification of the program in different ways then each kind is known as MUTANT [6, 7]. On each mutant, test cases are applied on it, results are compared with the original program's output, and if difference is present in their outputs then mutant will be rejected i.e. known as killing of mutant [7].

1.5.1 History[7,35]

- **Proposed By:** Richard Lipton in 1971 as a student.
- **First Developed & Published By:** DeMillo, Sayward, and Lipton.
- **First Implementation Testing Tool By:** Timothy Budd in 1980 as PhD work Yale University.
- **Mutation Testing Applied:** Object oriented programming and non-procedural languages for example SMV, XML, and state machines Etc.

1.5.2 Example [39]

Original Program		**Mutated Program**
Public static int gcds(a, b)		Public static int gcds(a, b)
{		{
while(b!=0)		while(b!=0)
{		{
int v;		int v;
v=b;		v=b;
b=a%b	*	**b=a/b**
a=v;		a=v;
}		}
return a;		return a;

}   }

1.6 Mutation Testing Process

Mutation testing has following steps [3, 15]:

1. Faults are injected into the program with the help of different operator to create different types of the program known as MUTANT.
2. Test cases are applied on the mutant and the original program too.
3. Then results of each mutant are compared with the actual ones.
4. If results vary, then mutant is destroyed and said to be killed.
5. If any mutant left alive, then new test data are introduced to apply to it.
6. The main aim is to fail the mutant, thus demonstrating effectiveness of the test case.

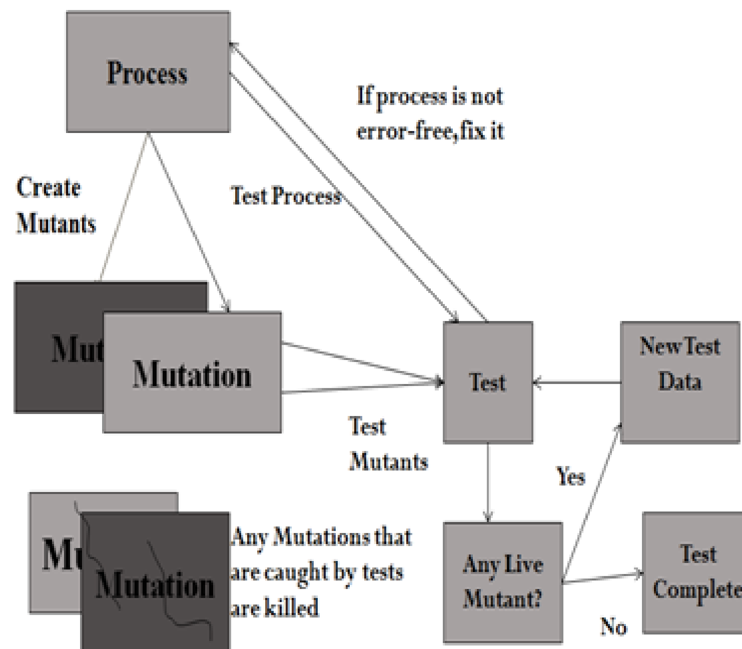


Figure1.4. Mutation Process [15]

Mutation system which injects only one change at a time in the program is called ***first-order*** mutant and the one which injects multiple changes at a time in the program is called ***higher-order*** mutant. The aim is to help the tester to develop effective test cases and locate weaknesses in the test data to be used for testing.

1.7 Test data Conditions for Killing Mutants

There are three conditions which must be required to kill a mutant to satisfy test data .Let a program denotes as *P*, mutant as *M*, statement as *S*, and test data as *T* for *P*. The conditions are given as [16], [17]:

- **Reachability condition:** *M* is only syntactic change in the *S*, so that s must be reached and the other statements in *M* are similar to original program. If *S* is not reachable by *T* then, it is not possible to kill *M*.
- **Necessity condition:** *T* must try to bring *M* to some different state than that of *P* on *S*. It is necessary that *S* must be reachable for *T* to kill *M*. As if output of *M* and *P* do not differ after applying *S* on them then, *M* will never get killed.
- **Sufficiency condition:** The final state of *M* should be different from *P*. Thus, the different state caused by sufficient condition must raise through the program's reckoning to result in a different output.

1.8 Kinds of mutation testing

Three kinds of mutations are:

1.8.1 Value Mutations: These mutations attempt to change the values of constants to detect errors in the programs. For example changing values to larger values or to very small values, swapping values.

1.8.2 Decision Mutations: These mutations attempt to modify the conditions/decisions to check for the design errors. For example replacing arithmetic (+,-,*,/) , relational(>,<,<=,>=,==), and logical (AND,OR,NOT) operators.

1.8.3 Statement Mutations: These mutations attempt to delete certain lines or duplicate some to reflect oversights in coding or swapping the request of lines of code.

1.9 Types of Mutant

Three types of mutant are present in mutation testing [6]:

1.9.1 Killed Mutant: If the test case is able to detect the fault or the change in the mutant, then it is said to be **killed** and mutant is considered as **dead** and no longer needed.

1.9.2 Alive/Live Mutant: If mutant gives some output as the original program's output then, it is called as a **live** mutant.

1.9.3 Equivalent Mutant: If a mutant produces the same output as the original program always then, it is said to be equivalent mutant.

1.10 Mutation Score

The mutation score for a test suite is the percent of killed mutants by test cases [6]. Mutation score is said to be 100 % if a set of test cases are adequate.

$$Mutation\ Score = \frac{100 * D}{N - E}$$

Where,

D = Dead mutants

N = Number of mutants

E = Number of equivalent mutants.

1.11 Operators

Mutation operators are placed into the program to create mutants, which leads to the change in syntax of the program. These are like replacing operand with another type

of operand, modifying expression by placing new operators, or deleting entire statements, etc.

1.11.1 Method level mutation operators: These operators alter the original program by replacing, deleting, inserting the operators. These are of six types and are also divided into binary, unary, and short-cut variants, so there are total 16 method level operators [9].

1.11.1.1 Arithmetic Operators

- Binary: a + a (addition), a - a (subtraction), a * a (multiplication), a / a (division), a % a (modulus).
- Unary: + (positive value), - (negative value).
- Short-cut: a++ (post-increment), ++a (pre-increment), a-- (post-decrement), --a (pre-decrement).

Here ‘a’ means operand on which these operators are applied.

1.11.1.2 Relational Operators

- a > a (greater than), a >= a (greater than equal to), a < a (less than), a <= a (less than equal to), a == a (equal to) and a! = a (not equal to).

Here ‘a’ means operand on which these operators are applied.

1.11.1.3 Conditional Operators

- Binary: a && a (conditional AND), a || a (conditional OR), a & a (bitwise AND), a | a (bitwise OR) and ^ (bitwise XOR).
- Unary: !a (bitwise logical complement).

Here ‘a’ means operand on which these operators are applied.

1.11.1.4 Shift Operators

- >>, <<, and >>>> are three shift operators in java. A shift operator performs bit manipulation on data by shifting the bits of its first operand right or left.

1.11.1.5 Logical Operators

- Binary: &(AND), |(OR) and ^(XOR) are three binary logical operators.
- Unary: unary logical operator is ˜.

1.11.1.6 Assignment Operators

- The basic assignment operator assigns the value of the right side expression to the left side variable.
- Short-cut: +=, -=, *=, /=, %=, &=,|=, ˆ=, <<=, >>=, and >>>=, eleven operators are defined.

Table1.1: Method level Mutation Operators [9]

Category	Operator	Description
Arithmetic	AOR^B	Arithmetic Operator Replacements
	AOR^U	Arithmetic Operator Replacements
	AOR^S	Arithmetic Operator Replacements
	AOI^U	Arithmetic Operator Insertions
	AOI^S	Arithmetic Operator Insertions
	AOD^U	Arithmetic Operator Deletions
	AOD^S	Arithmetic Operator Deletions
Relational	R0R	Relational Operator Replacements
Conditional	COR	Conditional Operator Replacements
	COI	Conditional Operator Insertions
	COD	Conditional Operator Deletions
Shift	SOR	Shift Operator Replacements
Logical	LOR	Logical Operator Replacements
	LOI	Logical Operator Insertions
	LOD	Logical Operator Deletions
Assignment	ASR^S	Assignment Operator Replacements

1.11.2 Class level mutation operators: The class level mutation operators are divided into four groups, based on the highlights/features of language. The first three groups are based on highlights/features which are common to all object oriented languages and the last group includes features in object oriented which are Java-specific. The four groups of class level mutation operators are [8, 38]:

1.11.2.1 Encapsulation: It controls the level of access of data and methods. In java access levels are public, private, and protected.

Table1.2: Class Level Mutation Operators [8]

Language	**Feature Operator**	**Description**
Encapsulation	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletions
	IHI	Hiding variable insertions
	IOD	Overriding method deletions
	IOR	Overriding method rename
	ISI	super keyword insertions
	ISD	super keyword deletions
	IPC	Explicit call to a parent's constructor deletions
Polymorphism	PNC	new method call with child class type
	PMD	Member variable declaration with parent class types
	PPD	Parameter variable declaration with child class types
	PCI	Type cast operator insertion Polymorphism
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variables

	OMR	Overloading method contents replace
	OMD	Overloading method deletions
Language	**Feature Operator**	**Description**
Java-Specific Features	JTI	this keyword insertions
	JTD	this keyword deletions
	JSI	static modifier insertions
	JSD	static modifier deletion Java-Specific
	JID	Member variable initialization deletion Features
	JDC	Java-supported default constructor deletions
Java-Specific Features	EOA	Reference assignment and content assignment replacements
	EOC	Reference comparison and content comparison replacements
	EAM	Assessor method change
	EMM	Modifier method change

1.11.2.2 Inheritance: It allows the use of methods and data of one class (parent class) in another class (child class) and increases code reusability.

1.11.2.3 Polymorphism: It allows the object to behave differently with the same method. It has a same number of methods with same name but different type and execution.

1.11.2.4 Java-Specific: It includes the few features of object oriented that are not included in other OO languages.

Chapter2: Literature Survey

Mutation Testing has been well known for decades to computer scientists. It was introduced to measure the adequacy of the test cases [12]. Basically, Mutation testing is a fault based testing technique that is quite expensive. Application of mutation testing is in every field like XML, web applications, database applications, JavaScript, object oriented programming, and many more.

2.1 Mutation Testing for Database Applications

Database applications are an important component of many software systems in area such as banking, online shopping and education.

2.1.1 Architecture of Database Application: The building design of database applications is multi-layered.

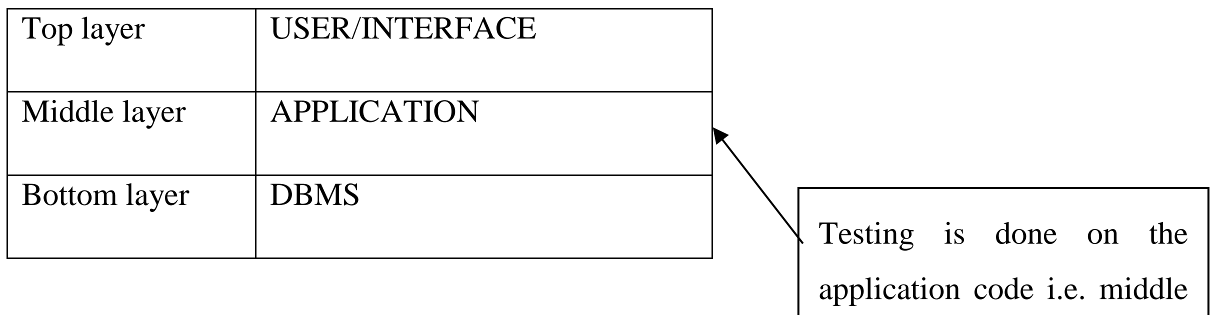


Figure2.1: Architecture of Database Applications

- The top level is the **client/user interface**, which gives collaboration with the end clients/users.
- The center/middle layer is the **application layer**, which actualizes rationale and interfaces with the database.

- The base/bottom layer is the database administration framework (DBMS), which sorts out and deals with the database, which contains tireless information as tables.

Zhou *et al.* [6] said testing is done on the middle layer i.e. application layer.

2.1.2 Problems in Testing (DB) database Applications: Chays *et al*. [26] discussed some problems in testing database applications which are discussed below:

- Role of Database state: It's not only about input and output states of the database but also the database state.
- Observability.
- Controllability.
- Adding another element to database offers ascend to several test cases:
    - Feature effectively subscribed.
    - Not available in the range.
    - Not perfect with officially subscribed components
    - Available in the range and compatible
    - Customer has no components/elements at al.
- Populating the database with live data or synthetic data.

2.1.3 SQL Mutation Operators: Tuya *et al*. [18, 19] built up the "SQLMutation" tool to naturally create mutants of SQL questions.

- **SC – SQL**: It is termed as clause mutation operator. These perform changes on the principle conditions like: select, join, sub-questions, assemble by, union, arrange by and total capacity.
- **OR**: It is termed as Operator replacement mutation operator. These are like the adjustment operators in addition to extra operator's particular to between and like predicates.
- **NL:** It is termed as NULL change mutation operator: It is identified with the treatment of invalid qualities.

- **IR**: It is termed as Identifier replacement change operator. Replacement of segments, constants and inquiry parameters that is available either in the question or in the tables.

2.1.4 JDAMA (Java Database Application Mutation Analyzer): Tuya *et al*. [18, 19] proposed mutation-based adequacy criteria for database queries and added to an arrangement of mutation operators. Zhou *et al*. [6, 24] expands that approach by incorporating it with examination and instrumentation of the application bytecode. JDAMA is a mutation testing tool that communicates with the database via Java Database Connectivity (JDBC) interface.

2.1.5 MutaGen: Pan *et al*. [27] said in testing database applications, tests comprise of both project inputs and database states. Evaluating the ampleness of tests permits focused on era of new tests for enhancing their sufficiency (For example, deficiency recognition capacities). Mutation testing would deliver an arrangement of mutants and afterward measure how high rate of these mutants are executed (i.e., recognized) by the tests under evaluation. Be that as it may, existing test-era approaches for database applications don't give adequate backing to executing mutants in database applications (in either program code or its inserted or came about SQL inquiries). To address such issues, Pan *et al*. [27] proposed a methodology called MutaGen that directed test era for mutation testing on database applications. In this, first apply a current approach that associates different constraints inside of a database application through developing combined database collaborations and changing the requirements from SQL inquiries into typical system code. In view of the changed code, produce system code mutants and SQL-question mutants, and after that determine and consolidate inquiry mutant-killing constraints into the changed code. At that point, produce tests to fulfill the inquiry mutant-killing constraints. Assessment results demonstrate that MutaGen can successfully kill mutants in database applications.

2.1.6 Relational Constraint Solver: Khalik and Khurshid [25] said database management system has been utilized generally for the previous decennary. DBMS

testing, in general, is a work concentrated, lengthy procedure, frequently performed physically. For instance, to test the accuracy of a question execution, the analyzer is obliged to populate the database with fascinating values that empower bug disclosure, furthermore, physically check the execution consequence of the question based on the info information. Computerizing DBMS testing not just lessens improvement costs, additionally expands the dependability in the created frameworks.

Khalik and Khurshid [25] depicted a computerized methodology for black-box testing of DBMS utilizing asocial imperative solver. They diminish the issue of robotized database testing into creating three ancient rarities:

1. SQL inquiries for testing it.
2. Significant data information to populate test databases.
3. Normal after effects of executing the inquiries on the created information.

They have displayed a point by point depiction of our system for Automated SQL Query Generation utilizing the Alloy instrument set, what's more, exploratory after effects of testing database motors utilizing structure. They depicted how the SQL constraints can be comprehended by making an interpretation of them to Alloy requirements to create semantically and linguistically rectify SQL inquiries.

2.2. Mutation testing on Relational Database

A relational database is a key segment of genuine programming frameworks. The ER diagram (representation of an arrangement) of a social database indicates the sorts of information that will be utilized by an application: how the information will be sorted out into tables, which information qualities are legitimate, and what connections may exist between them [23].

2.2.1 SynConSMutate System: Sarkar *et al.* [28] said testing methods for database applications commonly incorporate era of database states (manufactured information) alongside programmed era of experiments. The nature of such test cases is assessed on the basis of basic scope of the host language (For example, Java), though, the

nature of test cases for the installed dialect (For example, SQL) is assessed independently utilizing transformation testing. Sarkar *et al.* [28] proposed a system called SynConSMutate for test cases and synthetic data era for database applications. The created test cases regarding the recently produced, manufactured information guarantee high caliber not just as far as the scope of code written in the host language, additionally as far as the mutant location of the inquiries written in the embedded language.

Testing is fundamental for quality confirmation of database applications. Accomplishing high code scope of the database application is essential in testing. Utilizing a current database state is alluring since it has a tendency to be illustrative of true objects' qualities, helping identify issues that could bring about disappointments in true settings. On the other hand, to cover a particular project code part (For example, piece), proper system/program inputs likewise should be created for the given existing database state [29].

2.2.2 Search based strategy: Kapfhammer *et al.* [21] suggested one strategy for testing a relational database which includes consequently creating a progression of SQL INSERT proclamations and informational values that are intended to highlight potential streams in the database's structure. It is vital to survey the adequacy of the created information. Utilizing mutation operators, which will change the requirements/constraints of a schema. Change investigation regularly causes a high computational expense on the grounds that it requires the era and execution of numerous mutants i.e. adjusted forms of source code.

Kapfhammer *et al.* [21] proposed strategy named Search based strategy that have been effectively connected to numerous other. Types of test information era – most quite auxiliary testing – yet have not, up to this point, been connected to producing database table information. The strategy makes a progression of SQL_INSERT_Statements containing information produced to work out every integrity constraint of a schema (composition) as true or not.

2.3 Schema Testing and its need

A database structure determines the sorts of information which will be utilized by an application, how the information will be composed into tables, which information qualities are substantial, and also checks what connections may exist between them. Any application supported by a database, the commencement of the mapping is one of the first stages of the improvement cycle. A key part of a database pattern is the meaning of its integrity constraints [21, 30].

Table2.1: An example of a Schema [5]

```
CREATE TABLE Flight (
        FLIGHT_ID CHAR(10) NOT NULL,
        SEGMENT_NUMBERs INT NOT NULL,
        ORIGINAL_AIRPORT CHAR(10),
        DEPART_TIME TIME,
        DEST_AIRPORT CHAR(10),
        ARRIVE_TIME TIME,
        MEAL CHAR(10),
        PRIMARY KEY(FLIGHT_ID, SEGMENT_NUMBER),
        );
 CREATE TABLE Flight_Available (
        FLIGHT_ID CHAR(10) NOT NULL,
        SEGMENT_NUMBER INT NOT NULL,
        FLIGHT_DATE DATETIME NOT NULL,
        ECONOMY_SEAT_TAKEN INT,
        BUSINESS_SEAT_TAKEN INT,
        FIRSTCLASS_SEAT_TAKEN INT,
        PRIMARY KEY(FLIGHT_ID, SEGMENT_NUMBER),
        FOREIGNKEY(FLIGHT_ID,SEGMENT_NUMBER)REFERENCES Flight
 (FLIGHT_ID, SEGMENT_NUMBER));
```

Integrity constraints [31, 32] secure the constancy of information in a social/ relational database by creating the database administration framework to reject the insertion of table columns – by means of SQL INSERT proclamations – which don't fulfill certain limitations. In Table 2.1, the meaning of the structure of the pattern is one of the initial phases in building up an application supported by a database. Case in point, assume that the segment SEGMENT_NUMBER (indicating an individual phase of a whole deal trip) was removed from the primary key for the Flight table, with lines particularly recognized by FLIGHT_ID just. Presently, flight information including more than one flight stage may be dismisses, because of the infringement of the table's essential key. Such faults can be gotten ahead of schedule in the meaning of the construction through testing. Test information should be created that first embeds some subjective information into the database. Case in point, consider

INSERT INTO Flight VALUES ('b', 3, ...);

Took after by an INSERT rehashing the same FLIGHT_ID:

INSERT INTO Flight VALUES (’b’, 4, ...);

2.4 Search-Based Generation of Database Table

Search-based technique for generating database table, implemented through SchemaAnalyst shown in Figure below [21].

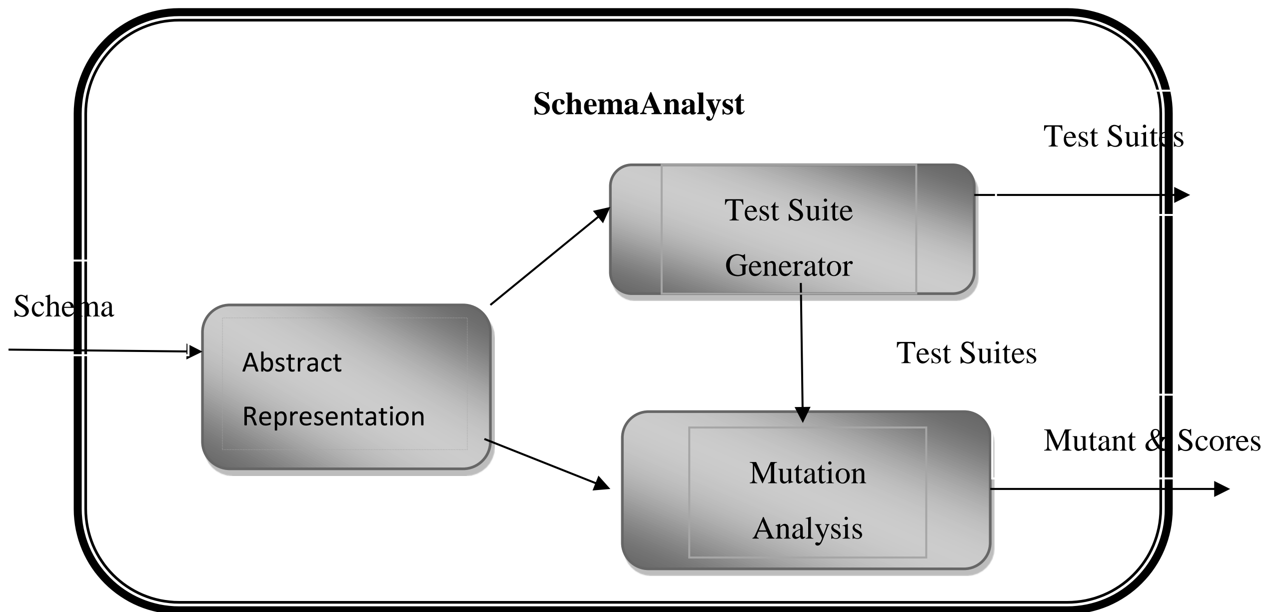


Figure2.2: SchemaAnalyst [21]

2.4.1 Abstract Representation of Schemas: The main stride of our method includes changing over a database diagram to a DBMS-free shape known as its "abstract representation." Different DBMSs support distinctive arrangements of data types, and as a component of this transformation prepare, a mapping must be produced using every concrete DBMS data type to "all inclusive" types.

2.4.2 Mutation Analysis of Schemas: Keeping in mind the end goal to evaluate the adequacy of test suites produced by strategy.

Mutants are delivered as takes after [21]:

- **Primary Keys**: Primary key mutants take three structures i.e.:
    - Add Column
    - Replace Column
    - Remove Column

Mutants are created by repeating through a table's column. In the event that the column is a piece of the first table's primary key, a mutant is created with that column expelled from the primary_key. A mutant is created with the column included, alongside the making of further mutants where the column replaces every current primary key column thusly. For the Flight table of table 2.1, a case of an uproot section mutant would be unified with the primary key affirmation PRIMARY KEY (FLIGHT_ID) (i.e., with SEGMENT_NUMBER missing) while an illustration of an include segment mutant incorporates PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER, ORIGINAL_AIRPORT).

- **Unique Constraints:** Mutant generation for UNIQUE requirements meets expectations in a comparative manner to primary keys, with the exception of that, while a table can have a few UNIQUE affirmations, it can just have one essential key. The arrangement of UNIQUE limitations is ordered, alongside an extra "exhaust" limitation with no columns. For every requirement in the set, the calculation for including and uprooting column is taken after with respect to primary keys, taking consideration to keep away from the generation of indistinguishable mutants from the transformation of unique constraint inside of the same table.
- **Not Null Constraints:** The NOT NULL constraint repeats through the non-primary key segments/columns of every table of a pattern and inverts it's NOT NULL status. In the event that a column if found to not have a NOT NULL constraint, a mutant is delivered with one included.
- **Foreign Keys Constraint**: The creation of mutants for FOREIGN KEY constraints (requirements) includes repeating through the foreign keys and delivering a mutant where each foreign key column is expelled from the table.

Mutation analysis is a powerful approach to survey the nature of data values and test suites. Yet, since this method requires the era and execution of numerous mutants, it frequently causes a considerable computational cost [21]. In the connection of

mutation, the utilization of mutant schemata and parallelization can decrease the expenses of mutation analysis [33, 34].

Wright *et al*. [20] implies that these methods i.e. mutant schemata and parallelization can reduce expenses of mutation analysis of a relational schema/database.

2.5 Mutant Schemata

To conceivably accelerate unique procedure, Wright *et al*. [20] applied the mutant schemata approach by making a meta-mutant [22] –a vast database construction containing the greater part of the tables needed for every mutant. There can't be more than one table with the same name, and there will be a few mutant forms of the table that should be consolidated into the meta-mutant.

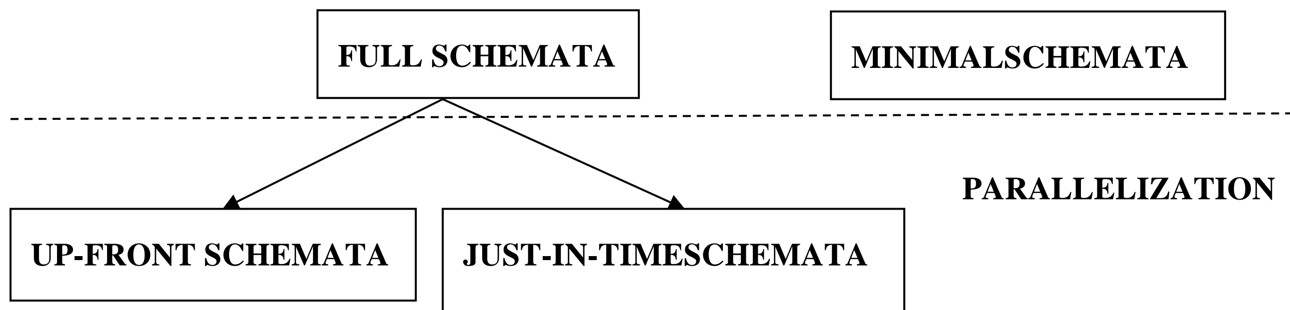


Figure2.3: Relationship between approaches [20]

2.5.1 Full Schemata: Wright *et al*. [20] proposed "Full Schemata" methodology which includes making of the renamed tables of every mutant, which are utilized to make meta- mutant.

Table 2.2: Full Schemata mutation analysis algorithm [20]

ALGORITHM
1) Meta-mutant creation i.e. Database mutant
a) for each mutant do
b) Prefix name of mutant tables with unique ID
c) end for
d) Create mutant tables in database
2) Evaluation of Mutant
(a) K $\Leftarrow$ Φ
(b) for each mutant in database do
(c) killed $\Leftarrow$ false
(d) for each SQL_INSERT_Statement in test Case do
(e) SQL_INSERT_Statement $\Leftarrow$ SQL_INSERT_Statement modified to use unique ID of mutant table
(f) originalResults $\Leftarrow$ Pre-calculated result of insert with original_table
(g) mutantResults$\Leftarrow$ execute_With_DBMS(SQL_INSERT_Statement)
(h) if originalResults $\neq$ mutantResults then
(i) K $\Leftarrow$ K $\cup$ {mutant}
(j) end if
(k) end for
(l) end for
3) Clean up
(a) Remove mutant tables from database

2.4.2Minimal Schemata: Wright *et al*. [20] proposed "Minimal Schemata" which creates a meta-mutant by combining the original database schema with just the tables for every mutant that have been altered – the mutant1s called as affected tables.

Table 2.3: Minimal Schemata mutation analysis algorithm [20]

ALGORITHM

1) Meta-mutant creation i.e. database mutant
   a) for each mutant in database do
   b) mutant1 $\Leftarrow$ mutant with non-affected tables removed
   c) Prefix names of mutant1 table with unique ID
   d) end for and Create mutant tables in database along with original tables
2) Evaluation of Mutant
   a) K $\Leftarrow$ Φ
   b) for each SQL_INSERT_Statement in test Case do
   c) originalResults $\Leftarrow$ Pre-calculated result of insert with original_table
   d) affectedTable $\Leftarrow$ The table the insert is involving
   e) affectedMutants $\Leftarrow$ The mutants that mutated affectedTable
   f) execute_With_DBMS(SQL_INSERT_Statement)
   g) for each affectedMutant do
      (a) SQL_INSERT_Statement $\Leftarrow$ SQL_INSERT_Statement modified to use unique ID of affectedMutant for table name
      (b) mutantResults $\Leftarrow$ execute_With_DBMS(SQL_INSERT_Statement)
      (c) if originalResults $\neq$ mutantResults then
      (d) K $\Leftarrow$ K $\cup$ {mutant}
      (e) end if
   h) end for
   i) end for
3. Clean up: Remove mutant tables from database

There is a slight difference between Full Schemata and Minimal Schemata approaches which is illustrated in the Figure 2.4.

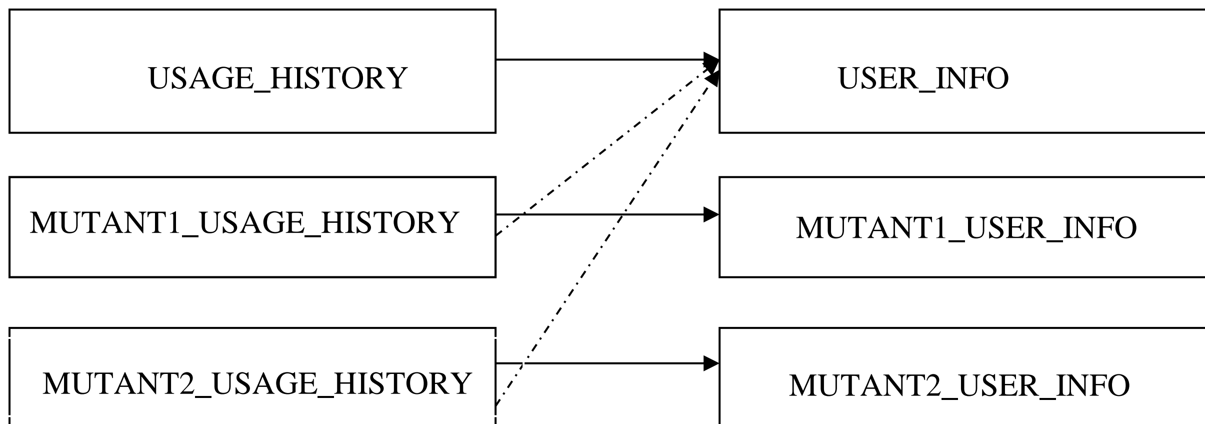


Figure2.4: An example of dependencies between tables using both approaches for the database UnixUsage [20]

The "Minimal" methodology depends upon regular tables, though the "Full" approach incorporates duplicates of those tables for every mutant.

2.6 Parallelization

Wright *et al.*[20] said the utilization of mutant schemata for database offers ascend to the likelihood of effortlessly parallelizing mutant assessment; mutants exist all the while in a database, and numerous database administration frameworks consider parallel access and control of the information.

- **Just-in-Time Schemata**: Wright *et al.* [20] said the perception behind the "*Just-in-Time Schemata*" approach the database tables comparing to every mutant can be included after some time, furthermore in parallel, just before the mutant should be assessed, and erased straight a while later. Just-in-Time Schemata mutation analysis time also termed as Full Schemata with parallelization.

Table 2.4: Just-in-Time Schemata mutation analysis algorithm (FSP) [20]

ALGORITHM

1) Meta-mutant creation i.e. database mutant
   a) for each mutant do
   b) Prefix name of mutant tables with unique ID
   c) end for
2) Evaluation of Mutant
   a) K $\Leftarrow$ Φ
   b) parallel for each mutant in database do
      (a) Create mutant tables in database
      (b) for each SQL_INSERT_Statement in test Suite do
      (c) originalResults $\Leftarrow$ Pre-calculated result of insert with original_table
      (d) mutantResults $\Leftarrow$ execute_With_DBMS(SQL_INSERT_Statement)
      (e) if originalResults $\neq$ mutantResults then
      (f) K $\Leftarrow$ K $\cup$ {mutant}
      (g) end if
      (h) end for
      (i) Remove tables in database for mutant
   c) end parallel for

Chapter 3: Problem Statement

This chapter includes the problem statement and objectives of the thesis.

3.1 Problem Statement

Despite the fact that mutation analysis is a viable approach to evaluating the nature of data values and test suites, but it is a computationally expensive system and time consuming. Full Schemata with Parallelization approach is better among both Full Schemata and Minimal Schemata, but the redundancy is there.

By developing an effective algorithm, we can provide less time execution with zero percent redundancy. So that it could provide faster and better performance among other existing algorithms.

3.2 Thesis Objectives

Based on literature survey, following objectives are outlined:

1. Analysis of various techniques used for testing of databases.
2. To propose an approach of Minimal Schemata with Parallelization.
3. Comparison among the proposed approach and existing approaches.

Chapter 4: Proposed Worked

This chapter contains the details of the work that has been carried out meet the objective of the thesis.

4.1 Approach used for Mutation Testing

Initially, mutants are generated for the database schema. It is generated by introducing a single fault in the program [5, 7].These deformities/bugs are presented with the assistance of the administrators which are now very much characterized. After mutant creation, all the experiments are executed identified with the whole mutants and output will be observed.

Figure 4.1 demonstrates an iteration of mutation testing procedure. Mutation testing process begins with the generation of mutants. A mutant is actually a copy of original program containing a fault which is syntactically correct. These faults are introduced using a pre-defined set of arrangement of changes called mutation operators. Subsequent to creating mutants, the next step is to execute all the test cases against all mutants and compare the outputs with the original program's outputs. If a test case produces distinctive output on a mutant then the mutant is said to be killed by the test case otherwise alive. The percentage of mutants killed is known as mutation score. Analyzer can re-create test cases to kill the remaining alive mutants and to raise the mutation score in light of the fact that a test suite/case with higher score is viewed as more successful [35].

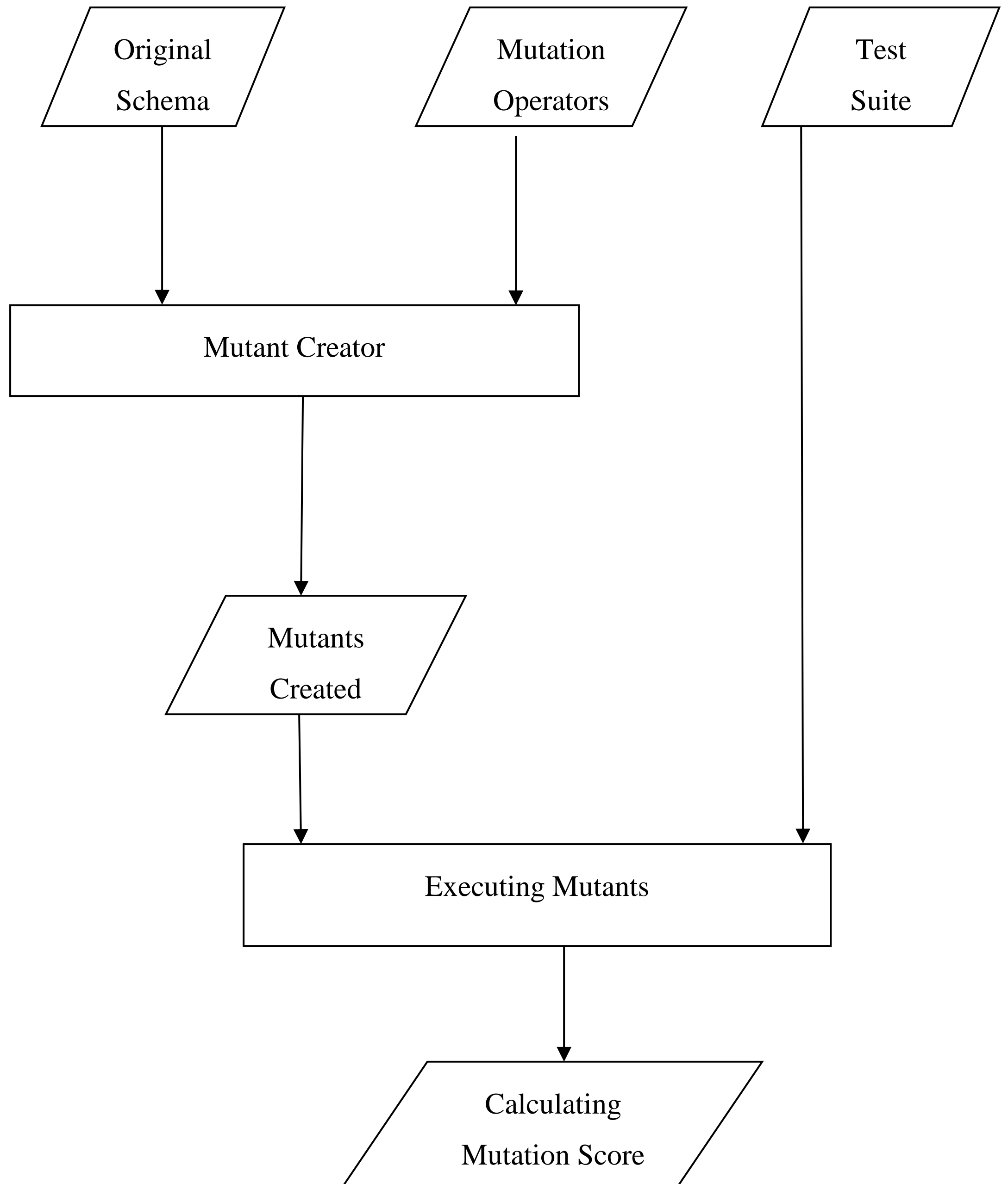


Figure4.1: Mutation Testing Flow [35]

4.2 Steps for Mutation Analysis

For achieving mutation analysis, following three steps are performed [27]:

- **Generation of Mutants:** Multiple copies of the program (here database schema) are created with small syntactic changes in the program/schema, which are called mutants.
- **Execution of Mutants:** Entire test cases are executed corresponding to the main program, any whole mutants. Moreover, killed and alive mutants are detected.
- **Result Analysis:** Mutation score is calculated that gives the quality of the executed tests.

4.3 Proposed Technique

In the proposed technique, a modified version of Minimal Schemata is developed, i.e. Minimal Schemata with parallelization (MSP) that will reduce mutation analysis time too little bit as compare to Minimal Schemata with no redundant schema in Meta mutant i.e. in mutant database. This technique works on the Java with the database used is PostgreSQL; connectivity is done through the JDBC interface.

In this approach, Meta mutant is generated, i.e. mutants are created by modifying the integrity constraints of the table that are presented in the original database and stored in the Meta mutant i.e. mutant database with original database tables in it. Then redundant tables are removed from the Meta mutant and test cases are applied on it. Mutants are created by adding, removing and replacing integrity constraints, i.e. Primary Key, Unique Key, Not Null, and FOREIGN KEY constraints (requirements). Test cases used are few INSERT statements that are applied on the both original and Meta mutant and then results of both are compared.

Then mutation score with mutation analysis time is calculated i.e. how much time is taken for creating mutants, removing redundant mutants and then comparing results.

Mutation score is calculated as [20]:

$$Mutation\ Score = \frac{abs(K1)}{Number\ of\ Mutants}$$

Where, K1 is the set of killed Mutants.

4.4 Proposed Algorithm

This section explains how algorithm works with pseudo code which explains the working of this algorithm. This algorithm aims to reduce the number of tables' i.e. mutant tables that the DBMS must store. In this, firstly mutants are created by adding, removing and replacing the data integrity constraints which results into a number of mutants in which some has a similar structure to the actual table structure. In order to avoid testing them in this before applying the test suite to this these tables are removed and only affected mutants are left in the Meta mutant with the original tables. Then test cases are applied to it.

Original tables are also present in the Meta mutant as in this foreign key are dependent on the actual table not on the mutant table. All this process is done in parallel i.e. sequence is first mutant created related to one table in the database then redundant tables removed then test cases applied on it and so on.

Table 4.1: Minimal Schemata with parallelization mutation analysis algorithm

Input: Various databases in PostgreSQL and program in java. Output: Reduction in mutation analysis time with zero percent redundancy. Algorithm: **Minimal Schemata with Parallelization**
1. Create Meta-mutant i.e. Mutant database a. for each mutant do b. Prefix name of mutant tables with unique ID c. End for. 2. Evaluation of Mutant a. K1 $\Leftarrow \Phi$

b. Parallel for each mutant in database do
c. Create mutant table in database
d. Mutant1 ⇐ mutant with non-affected tables removed
e. for each SQL_INSERT_Statement in test suite do
   i. originalResults ⇐ Pre-calculated result of insert with original_table
   ii. affectedMutant ⇐ The mutants that mutated affectedTable
   iii. for each affectedMutant do
      - SQL_INSERT_STATEMENT1⇐ SQL_INSERT_STATEMENT modified with unique ID of affectedMutant for table name
      - mutantResults⇐execute_With_DBMS(SQL_INSERT_STATEMENT)
      - if originalResults ≠ mutantResults then
      - K1 ⇐ K1 ∪ {mutant}
      - End if
   iv. end for
f. end for
   Remove mutant tables from database.
g. end parallel for.

4.5 Experimental Results

PostgreSQL is used as the database with java as frontend to apply these approaches. The performance of the approach is studied using seven database case study, which varies in the number of tables, columns and constraints. All case study database schemas are available as open source except employee which is taken from the book [5].

Table 4.2: Database structures used for experimental study

Database Name	Tables	Columns	Primary Keys	Foreign Keys	Unique Keys	NOT-NULLS	Total
Jwhoisserver	7	58	7	12	4	52	75
Office	8	59	8	8	0	45	61
Northwind	15	95	17	0	0	33	50
University	4	16	3	3	1	4	11
Employee	4	13	4	2	0	10	16
BaseBall	24	317	70	0	0	70	140
UnixUsage	8	32	8	6	0	10	24
Total	70	590	117	31	5	224	377

Table 4.3: Summary of Tables and Test Suites i.e. No. of Insert Statements

Database Name	Tables	Mutants Created	No. of Insert Statements
Jwhoisserver	7	256	246
Office	8	259	3864
Northwind	15	297	3368
Employee	4	56	22
Baseball	107	1657	15698
University	4	63	18
UnixUsage	8	126	250545
Total	153	2714	273761

Following steps are performed:

1. Initially, when we execute the code our main window of GUI is displayed on the screen which is shown in figure 4.2. In this, we selected case study as "JWhoisserver" and "Minimal Schemata with Parallelization" approach for "PostgreSQL".

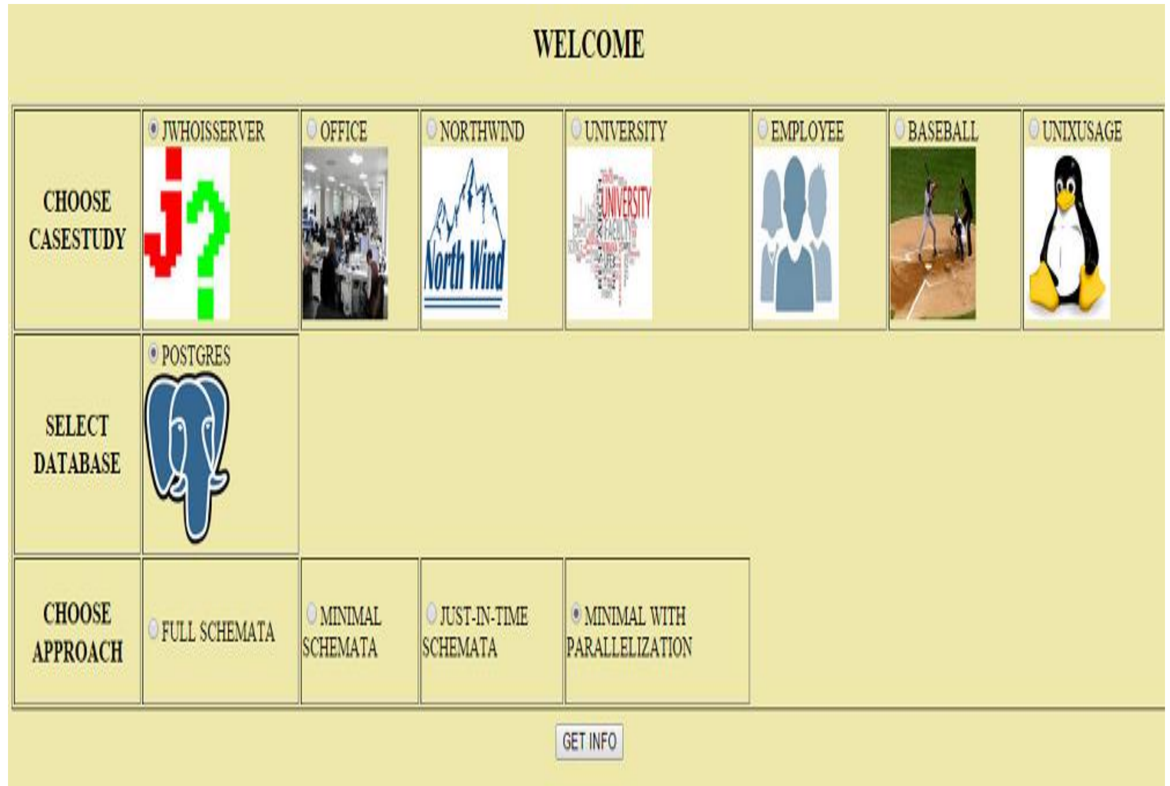


Figure 4.2: Main Window of GUI

2. Then, click on the button i.e. "GET INFO", which will give the information of the case study which includes the table name, column name with its description and constraints present in it which is presented in figure 4.3.

INFORMATION OF CASE STUDY - JWHOISSERVER

TABLE_NAME:COUNTRY

COLUMN_NAME	COLUMN_DATA_TYPE	COLUMN_SIZE	COLUMN_NULL_ABLE?	AUTO_INCREMENT?
country_key	serial	10	NOT NULL	Auto_Increment
short	varchar	2	NOT NULL	Not_Auto_Increment
country	varchar	255	NOT NULL	Not_Auto_Increment
country_key	country_pkey	country		
country_pkey	country_key			
unique_country	short			

TABLE_NAME:DOMAIN

COLUMN_NAME	COLUMN_DATA_TYPE	COLUMN_SIZE	COLUMN_NULL_ABLE?	AUTO_INCREMENT?
domain_key	bigserial	19	NOT NULL	Auto_Increment
domain	varchar	255	NOT NULL	Not_Auto_Increment
registered_date	timestamp	29	NOT NULL	Not_Auto_Increment
registerexpire_date	timestamp	29	NOT NULL	Not_Auto_Increment
changed	timestamp	29	NOT NULL	Not_Auto_Increment
remarks	varchar	255	NULL	Not_Auto_Increment
holder	int8	19	NOT NULL	Not_Auto_Increment
admin_c	int8	19	NOT NULL	Not_Auto_Increment

phone	varchar	100	NOT NULL	Not_Auto_Increment	
fax	varchar	100	NULL	Not_Auto_Increment	
email	varchar	255	NOT NULL	Not_Auto_Increment	
remarks	varchar	255	NULL	Not_Auto_Increment	
changed	timestamp	29	NOT NULL	Not_Auto_Increment	
mntnr_fkey	int8	19	NOT NULL	Not_Auto_Increment	
disabled	int2	5	NOT NULL	Not_Auto_Increment	
person_key	person_pkey	person			
fk_person_country	country_fkey	CASCADE	RESTRICT	country_key	country
fk_person_mntnr	mntnr_fkey	CASCADE	RESTRICT	mntnr_key	mntnr
fk_person_type	type_fkey	CASCADE	RESTRICT	type_key	type
person_pkey	person_key				

TABLE_NAME:TYPE

COLUMN_NAME	COLUMN_DATA_TYPE	COLUMN_SIZE	COLUMN_NULL_ABLE?	AUTO_INCREMENT?
type_key	serial	10	NOT NULL	Auto_Increment
type	varchar	100	NOT NULL	Not_Auto_Increment
type_key	type_pkey	type		
type_pkey	type_key			
unique_type	type			

GETMUTANT

Figure 4.3: Next Window is of the GUI which consists of information related to case study chosen.

3. Now, click on the "GETMUTANT" button to show mutant created during the approach which is only shown during Full Schemata and Minimal Schemata approach, shown in figure 4.4.

MUTANTS CREATED

S_NO.	TABLE NAME
0	country
1	domain
2	inetnum
3	mntnr
4	mutant0_country
5	mutant0_domain
6	mutant0_inetnum
7	mutant0_mntnr
8	mutant0_nameserver
9	mutant0_person
10	mutant0_type
11	mutant10_country
12	mutant10_domain
13	mutant10_inetnum
14	mutant10_mntnr
15	mutant10_nameserver
16	mutant10_person
17	mutant11_domain
18	mutant11_inetnum
19	mutant11_mntnr
20	mutant11_nameserver
21	mutant11_person
22	mutant12_domain
237	mutant6_inetnum
238	mutant6_mntnr
239	mutant6_nameserver
240	mutant6_person
241	mutant6_type
242	mutant7_country
243	mutant7_domain
244	mutant7_inetnum
245	mutant7_mntnr
246	mutant7_nameserver
247	mutant7_person
248	mutant8_country
249	mutant8_domain
250	mutant8_inetnum
251	mutant8_mntnr
252	mutant8_nameserver
253	mutant8_person
254	mutant9_country
255	mutant9_domain
256	mutant9_inetnum
257	mutant9_mntnr
258	mutant9_nameserver
259	mutant9_person
260	nameserver
261	person
262	type

NEXT

Figure 4.4: Mutant Created during Full Schemata and Minimal Schemata approach with case study chosen.

4. Click on the "NEXT" button to get the desired result window shown in figure 4.5.

TABLE_NAME	COUNTRY	DOMAIN	INETNUM	MNTNR	NAMESERVER	PERSON	TYPE
No._Of_Mutants	11	51	52	56	11	56	7
No._Of_Mutants NON-Affected	1	1	0	0	1	2	1
No._Of_Mutants Left	0	36	0	51	0	0	0
MUTATION_SCORE	0.9166666666666666	0.28846153846153844	0.9811320754716981	0.08771929824561403	0.9166666666666666	0.9824561403508771	0.875

COMPLETE MUTATION SCORE: 0.6254980079681275

TOTAL TIME TAKEN : 418.0

NEXT

Figure 4.5: Final results for the MSP approach for case study "JWhoisserver".

5. Final step, click next button, which will remove all the mutants from the database and back to the initial window. Similarly, results are generated for all the case studies with all different approaches.

4.6 Result Analysis

From the above, the following analysis has been made.

Table 4.4: Mutation Analysis Time for all the case studies

Database Name	FS	FSP	MS	MSP
Jwhoisserver	1201	589	673	418
Office	439	197	468	399
Northwind	112	196	374	312
University	76	28	104	95
Employee	63	29	97	91
Baseball	519	489	2863	2453
UnixUsage	350	100	280	259

In table 4.4, i.e. Mutation Analysis Time for all the case studies, following terms are used:

- FS: Full Schemata
- FSP: Full Schemata with Parallelization
- MS: Minimal Schemata
- MSP: Minimal Schemata with Parallelization.

From the table 4.4 it is cleared that we are able to reduce time in some aspect as compare to the Minimal Schemata (MS) analysis time algorithm. Mutation analysis time is reduced as well as no redundant mutant is present during testing of the mutant database which give better results and reduction in time and improving performance.

From table 4.4 graphs for every case study is given below. In which, horizontal axis represents the approaches and vertical axis time in seconds.

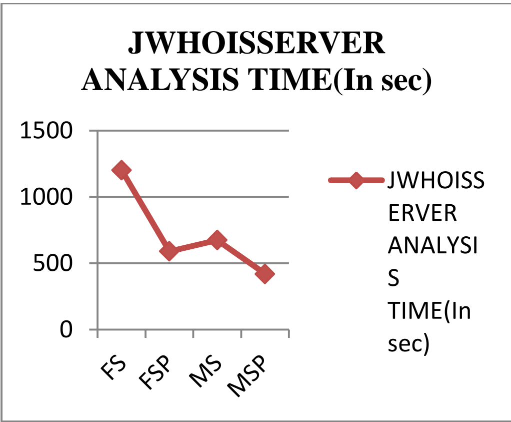


Figure 4.6: JWhoisserver analysis time (in sec) displaying average duration of time taken by each approach.

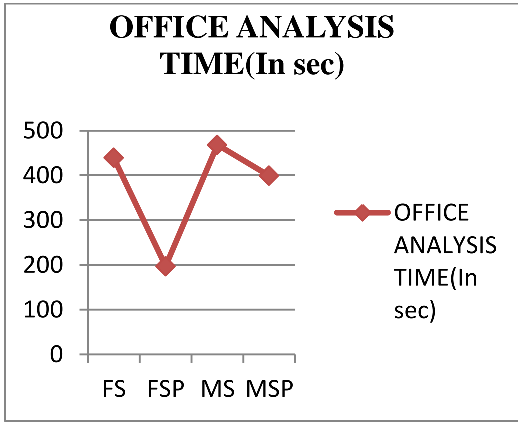


Figure 4.7: Office analysis time (in sec) displaying average duration of time taken by each approach.

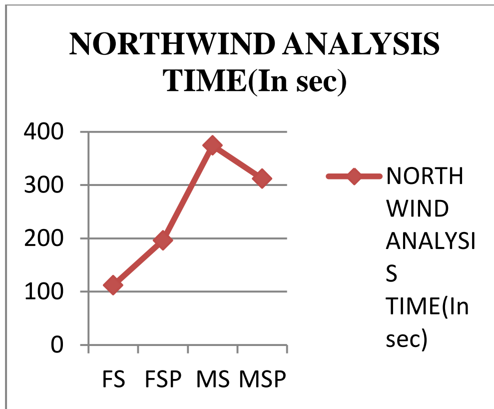


Figure 4.8: Northwind analysis time (in sec) displaying average duration of time taken by each approach.

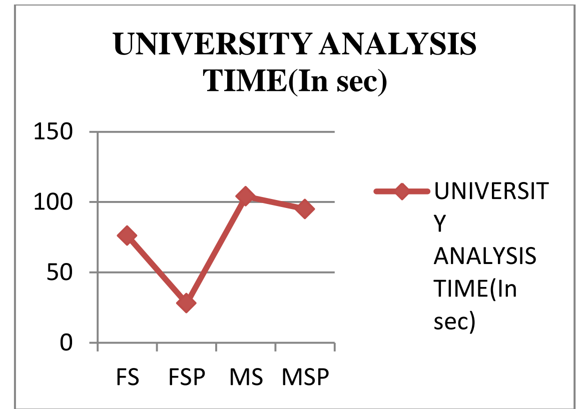


Figure 4.9: University analysis time (in sec) displaying average duration of time taken by each approach.

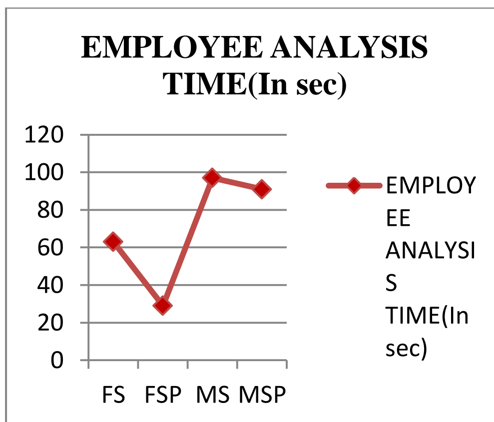


Figure 4.10: Employee analysis time (in sec) displaying average duration of time taken by each approach.

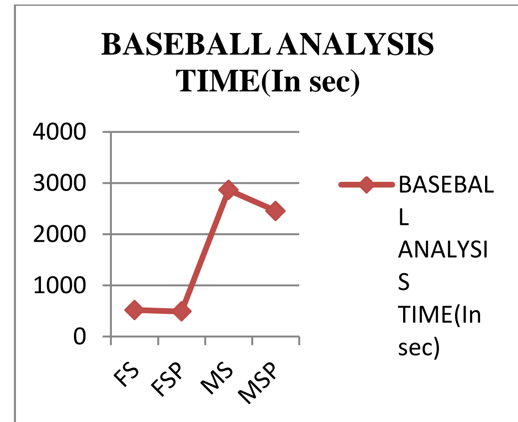


Figure 4.11: Baseball analysis time (in sec) displaying average duration of time taken by each approach.

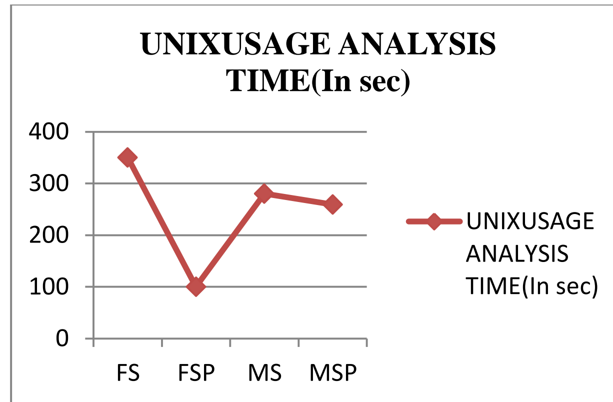


Figure 4.12: UnixUsage analysis time (in sec) displaying average duration of time taken by each approach.

The overall result analysis says:

- Yes, a minimal schema with parallelization reduces mutation analysis time.
- It depends on the various things like number of constraints present, the number of tables and number of columns too.
- Mutation analysis time also dependent on the number of test cases taken, i.e. the number of INSERT statements.
- It leads to 0% redundancy of mutants as this reduces testing efforts and increases performance.

Chapter 5: Conclusion and Future Scope

5.1 Conclusion

Mutation Testing is a fault based technique, done by generating faults and observes the effect by going through three test data conditions i.e. reachability, necessity and sufficiency. The main problem is cost and time.

Testing of the database is important as it plays an important role in various applications. Full Schemata and Minimal Schemata [20] were introduced to reduce overall mutation analysis time, but redundancy was minimized only in Minimal Schemata. Then full schemata are reintroduced along with parallelization [20] which reduced the mutation analysis time but was not able to overcome the redundancy problem.

In this thesis, we proposed an algorithm Minimal Schemata with Parallelization furthermore; this algorithm is applied on seven databases case studies using PostgreSQL. The result shows that there is an improvement in reduction of mutation analysis time as compared to the previous approaches with keeping zero percent redundancy.

5.2 Future Scope

In future following things can be done:

- This proposed algorithm can be applied in more number of database case studies.
- This proposed algorithm can be tested using different database software other than PostgreSQL and results can be compared.

References

[1] tutorialspoint.com,"Software Testing- Quick Guide", 2015. [Online]. Available:http://www.tutorialspoint.com/software_testing/software_testing_quick_guide.htm.

[2] P. Jalote, *A concise introduction to software engineering*, Springer, New York, ch. 8, pp. 226-228, 2008.

[3] softwaretestingtimes.com, "Software Testing Times - Tutorials, QTP, Manual Testing Automation Testing, Load Runner", 2010. [Online]. Available:http://www.softwaretestingtimes.com/2010/04/fault-error-failure.html.

[4] "Definitions and Meaning: Error, Fault, Failure and Defect", 2011. [Online].Available:http://blog.qatestlab.com/2011/12/06/definitions-and meaning-error-fault-failure-and-defect/.

[5] Pressman, *Software Engineering: A practitioner's Approach*, Pressman and Associates, pp. 450-451,484-485, 2005.

[6] C. Zhou and P. Frankl, "Mutation Testing for Java Database Applications", in *Software Testing Verification and Validation, 2009. ICST "09. International Conference*, Denver, CO, pp. 396 – 405, 2009.

[7] Wikipedia," Mutation testing", 2015. [Online]. Available: https://en.wikipedia.org/wiki/Mutation_testing.

[8] Ma, Y. and Offutt, J. "Description of Class-level Mutation Operators for Java". August 2014.

[9] Ma, Y. and Offutt, J. "Description of Method-level Mutation Operators for Java". November 2011.

[10] M. Harman and Y. Jia, "Mutation testing", 2009.

[11] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing", *IIEEE Trans. Software Eng.*, vol. 37(5), pp. 649-678, 2011.

[12] J.C. Maldonado and A. Rashid "Mutation Testing for Aspect-Oriented Programs", in *Software Testing, Verification, and Validation, 2008 1st International Conference*, Lillehammer, pp. 52 – 61, 2008

[13] IEEE Standard Association,"829-1998 IEEE Standard for Software Test Documentation", 1998.

[14] M. Ehmer, "Different Approaches To Black box Testing Technique For Finding Errors", *IJSEA*, vol. 2(4), pp. 31-40, 2011.

[15] P. May, J. Timmis and K. Mander, "Immune and Evolutionary Approaches to Software Mutation Testing.", in *Artificial Immune Systems, 6th International Conference, ICARIS*, Santos, Brazil, 2007.

[16] M. Polo, S. Tendero and M. Piattini, "Integrating techniques and tools for testing automation", *Softw. Test. Verif. Reliab.*, vol. 17(1), pp. 3-39, 2007.

[17] M. Umar, "An Evaluation of Mutation Operators for Equivalent Mutants", Department of Computer Science King's College, London, 2006.

[18] J. Tuya, M. Jose Suarez-Cabal and C. de la Riva, 'SQLMutation: A tool to generate mutants of SQL database queries', in *MUTATION '06 Proceedings of the Second Workshop on Mutation Analysis*, Washington, DC, USA, 2006.

[19] J. Tuya, M. Jose Suarez Cabal and C. de la Riva, 'Mutating database queries',

Information and Software Technology, vol. 49(4), pp. 398-417, 2007.

[20] C. J. Wright, G. M. Kapfhammer and, P. McMinn,"Efficient Mutation Analysis of Relational Database Structure Using Mutant Schemata and Parallelisation", in *IEEE Sixth Conference on Software Testing Verification and Validation Workshops* , pp.63-72, 2013

[21] G. M Kapfhammer, P. McMinn and C. J Wright, "Search-based testing of relational schema integrity constraints across multiple database management systems", in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference*, Luembourg, pp. 31-40, 2013

[22] R. Untch, A. Offutt and M. Harrold, 'Mutation analysis using mutant schemata', *SIGSOFT Softw. Eng. Notes*, vol. 18(3), pp. 139-148, 1993.

[23] Wikipedia,2015.[Online].Available:https://en.wikipedia.org/wiki/Relationaldatabase.

[24] C. Zhou and P. Frankl, 'JDAMA: Java database application mutation analyser', *Softw. Test. Verif. Reliab.*, vol. 21(3), pp. 241-263, 2011.

[25] K. S.A. and K. S., "Systematic Testing of Database Engines Using a Relational Constraint Solver", in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference*, Berlin, pp. 50-59, 2011

[26] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos and E. J. Weyuker, 'A framework for testing database applications', in *ISSTA '00 Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 147-157, 2000.

[27] K. Pan, X. Wu and T. Xie, "Automatic test generation for mutation testing on database applications", in *Automation of Software Test (AST), 2013 8th International Workshop*, San Francisco, CA, pp. 111-117, 2013.

[28] T. Sarkar, "SynConSMutate: Concolic Testing of Database Applications via Synthetic Data Guided by SQL Mutants", in *Information Technology: New*

Generations (ITNG), 2013 Tenth International Conference, Las Vegas, NV, pp. 337-342, 2013

[29] T. Sarkar, "Testing database applications using coverage analysis and mutation analysis", Graduate college, Iowa State University, 2013.

[30] D. Willmor and S. M. Embury, 'An intensional approach to the specification of test cases for database applications', in *ICSE '06 Proceedings of the 28th international conference on Software engineering*, pp. 102-111, 2006

[31] Beginner-sql-tutorial.com, "SQL Integrity Constraints", 2007. [Online]. Available: http://beginner-sql-tutorial.com/sql-integrity-constraints.htm.

[32] Wikipedia,"Dataintegrity",2015.[Online].Available:https://en.wikipedia.org/wiki/Data_integrity.

[33] J. Offutt, R. Pargas, S. Fichter and P. Khambekar, 'Mutation Testing of Software Using a MIMD Computer', in *1992 International Conference on Parallel Processing (ICPP' 92)*, Chicago, Illinois, pp. 257-266, 1992.

[34] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for Java", in *the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, New York, USA, pp. 297-298, 2009.

[35] M. Bashir and A. Nadeem, "Object oriented mutation testing: A survey", in *Emerging Technologies (ICET), 2012 International Conference*, Islamabad, pp. 1-6, 2012.

[36] B. B., *Software Testing Technique*. Dreamtech Press, 2003.

[37] M. Papadakis, M.E. Delmaro and Y.L. Traon, "Proteum/Fl:A tool for localizing faults using mutation analysis", *13th IEEE international working conference on source code analysis and manipulation*, SCAM, pp. 94-99, 2013.

[38] T. Singla and A. Kumar, "Mutation Operators corresponding Conditions Contributing in Deporting them Equivalently", *IJCST, vol.* 4(2), 656-658, 2013.

[39] T. Singla, A. Kumar and S. Garwhal, "Reducing Mutation Testing Endeavor using the Similar Conditions for the same Mutation Operators Occurs at Different Locations", *Applied Mathematics& Information Science,* vol. 8(5), 2389-2393, 2014.

Publication

Communicated

1. H. Gupta, and S. Garhwal, “Effectual Mutation Analysis of Relational Database using Minimal Schemata with Parallelization”, *IEEE International Conference on Information Processing, Vishwakarma Institute of Technology,* Pune, 2015.

Video Reference

1. https://www.youtube.com/watch?v=xxsF4HxeME8