DBGEN- Database (Test) GENerator - An Automated Framework for Database Application Testing

[1]Askarunisa A., [2]Prameela P, and [3]Dr. Ramraj N

[1,2]Thiagarajar College of Engineering, Madurai, Tamilnadu, India
[3]Principal Chennai, Tamilnadu, India
[1]nishanazer@yahoo.com, [2]prameela@tce.edu, [3]aacse@tce.edu

Abstract

Database applications play an important role in nearly every organization, yet little has been done on testing of database applications. They are becoming increasingly complex and are subject to constant change. They are often designed to be executed concurrently by many clients. Testing of database application hence is of utmost importance to avoid any future errors encountered in the application, since a single fault in database application can result in unrecoverable data loss. Many tools and frameworks for performing testing of database applications has been proposed to populate the test database and generate test cases which checks the correctness of application. They check database applications for consistency constraints and transactions concurrency. In this paper we present a DBGEN- database (test) GENerator, an automated framework for database application testing. In this framework Test Strategies for testing of embedded SQL queries within imperative language are presented. Finally we present strategies for performing efficient regression tests by reducing the resets that may occur while testing database applications. We have also computed the coverage of various test cases to predict the quality of testing. By this, we reduce the testing time and cost by approximately by 30% , thereby easing the tester to manage his testing activities easily.

Keywords: *Database Testing, Regression Testing, code coverage, slicing, Resets, SQL statements, test cases, Test Framework.*

1. Introduction

Testing determines the validity of the computer solution to a business problem. Testing is used as the demonstration of the validity of the software at each stage in the system development life cycle. The most expensive part is to carry out tests of the software that has been developed. Generally, large software vendors spend 50% of their development cost on testing [13]. Database systems have major importance and wide popularity in the software industry. Database applications are becoming very complex. They are composed of many components and stacked in several layers. Testing is essential for database applications to function correctly and with acceptable performance when deployed. Currently, two approaches dominate database application testing. With the first approach, application developers carry out their tests on their own local development databases. Obviously this approach can not fulfill the requirements of all the testing phases, especially those pertinent to performance and scalability, due to the limitation of relatively small size of data and test cases. Furthermore, the data in local development databases may not be accurate or close to real data. With the second approach, new applications are tested over live production databases. This approach cannot be applied in most situations due to the high risks of disclosure and incorrect updating of confidential information [6].

Testing of database applications is different from the testing of structural programs. It is common for all software applications written in an imperative language to have access to the database through SQL statements embedded in the code. These queries are part of the application's business logic. Because of this, it is necessary to conduct suitable testing in the same way for database as rest of the code. The inputs of database applications involve both the user inputs and the database instances. In addition to checking the outcome with the expected outcome, programmers or testers should also check if the database is consistent and reflects the original environments [14]. In theory, a test run does not fail, if all its requests produce correct answers and the state of the test database is correct after the execution of the test run. In this work, we relax this criterion and only test for correctness of answers. The reason is that checking the state of the test database after each test run can be prohibitively expensive and is difficult to implement for black box tests. After the application has changed (e.g., customization or a software upgrade), the DbUnit tool is to find out how the changes have affected the behavior of the application. Possibly, the tool also looks for differences in response time and for inconsistencies in the test database. At the end, the test tool provides a report with all requests that failed. Logically, the test database must be reset after each test run is recorded and executed. This way, it is guaranteed that all failures during executing test cases are due to updates at the application layer.

In this paper, we propose a framework DBGEN which performs the following tasks viz. reducing the large real –time database into a Intermediate Test database by preserving privacy and closed lookingness. Generation of effective test cases with the use of Database Schema, and the execution of the same. The framework maintains the consistency of the Database states with minimum number of resets and finally calculates the efficiency of the database test cases through coverage Metric.

2. Related Work

Testing of database applications was started earlier by Yuetang Deng et.al.[1]. In his work, in order to check a state constraint that is not enforced by the DBMS, a tool named AGENDA creates temporary tables to store the relevant data and converts the assertion into a check constraint at attribute/row level on the temporary tables. In particular, constraints involving aggregation functions, constraints involving multiple tables, and dynamic constraints involving multiple database states are transformed into simpler constraints on temporary tables, and code to automatically insert relevant values into the temporary tables is generated and executed [1]. As an extension, Yuetang Deng et.al ‘ s work , the first component Agenda parser extracts relevant information from the application’s database schema. State generator uses the database schema and populates the database tables with data satisfying the integrity constraints. Input generator generates input data to be supplied to the application. State validator investigates how the state of the application DB changes during execution of the test. Output validator is similar to the state validator. It captures the outputs and checks them against the query pre conditions and post conditions that are generated by the tool [3].

As a further extension, Yuetang Deng et.al ‘ s work, the technique dataflow analysis for identifying schedules of transaction execution aimed at revealing concurrency faults of this nature, along with techniques for controlling the DBMS or the application so that execution of transaction sequences follows generated schedules. The techniques have been integrated into AGENDA, a tool set for testing relational database application programs [4].

Design mechanisms to create the deterministic rule set, non-deterministic rule set, and statistic data set for a live production database was proposed by Xintao Wu et.al.[6]. A security Analyzer together with security requirements (security policy) and output were also built. The mock database generated from the new triplet can simulate the live environment for testing purpose, while maintaining the privacy of data in the original database [6]. Data flow testing[7] proposed by S. K. Gardikiotis et.al involved generating test data to force execution of different interactions between variable definitions and variable references or uses in a program variable. Here Database applications are reverse engineered in order to felicitate the embedded SQL statements. The derived code contains calls to SQL modules stored in the database server. To test these modules, data flow analysis is provided with respect to the statements of data manipulation language [7]. In [8], the testing approach WHODATE which transforms SQL statements to procedures in general-purpose programming language and application of conventional white box techniques on both these transformed procedures and the host statements to generate test cases were proposed. In [9], development and testing of database applications was considered difficult because the program execution depend on the persistent state stored in the database. Hence versioning of the persistent data stored in the database solved some critical problems in the development and testing of database applications [9].

Testing techniques explicitly considers the inclusion of database instances in the selection of test cases and the generation of test data input in [10]. This describes a supporting tool which generates a set of constraints, which collectively represent a property against which the program is tested. In Gregory M. Kapfhammer, Mary Lou Soffa work, a family of test adequacy criteria can be used to assess the quality of test suites for database driven applications [11]. A unique representation of a database-driven application that facilitates the enumeration of database interaction associations was developed. These associations reflects an application’s definition and use of database entities at multiple levels of granularity [11].In William and Alessandro work [12], generating command forms was the accurate identification of the possible SQL commands that could be issued at a given database interaction point. The execution of the application and which command forms are exercised [12] were monitored and determined. Database Interaction Testing Tool- DITTO was implemented in Java, which provides fully automated support for all aspects of the approach, and can guide the developer in testing database applications written in Java.

In this paper, we propose a framework DBGEN which performs the following tasks viz.

1. Reducing the large real –time database into a Intermediate Test database by preserving privacy and closed looking ness
2. Generation of effective test cases with the use of Database Schema, and the execution of the same.
3. The framework maintains the consistency of the Database states with minimum number of resets and

4. Finally calculates the efficiency of the database test cases through coverage Metric.

The rest of the paper is organized as follows. In Section 2 we review a view of database testing. In Section 3 the scope of study was described. In Section 4 presented the design and methodology of database testing. In Section 5 implementation was described. Section 6 we draw conclusions and describe directions for future work.

3. A View of Database Testing

Considering the widespread use of database systems there has been relatively little research into their testing. The work that has been produced differs by a number of factors, not least in the terminology that is used. In order to provide consistency in this paper we use the following terminology:

Application: a software program designed to fulfill some specific requirement. For example, we might have separate application programs to handle the entry of a new customer into the database, and to cancel dormant accounts once a time-limit has passed.

Database: a collection of interrelated data, structured according to a schema that serves one or more applications.

Database application: an application that accesses one or more databases. A database application will operate on both program and database state.

Database system: a logical collection of databases and associated (database) applications.

Testing is more difficult (or, at least, different) when dealing with database applications. The full behavior of a database application program is described in terms of the manipulation of two very different kinds of state: the program state and the database state. It is not enough to search for faults in program state; we must also generate tests that seek for faults that manifest themselves in the database state and in the interaction between the two forms of state. A further complication for testing is that the effects of changes to the database state may persist beyond the execution of the program that makes them, and may thus affect the behavior of other programs [19]. Thus, it is not possible to test database programs in isolation, as is done traditionally in testing research. For example, a fault may be inserted into the database by one program but then propagate to the output of a completely different program. Hence, we must create sequences of tests that search for faults in the interactions between programs. This issue has not yet been considered by the testing research community. This has been shown to be particularly important for regression testing where the change to the functionality of one program may adversely affect other programs via the database state [19]. The literature on testing database systems varies in a number of ways. A fundamental difference in the literature is in the understanding as to exactly what a database system is. Each definition is constrained to a particular situation. There is no definition general enough to be applied to the different scenarios in which database systems may be used. The simplest view is when a single application interacts with a single database [2, 3, 5]. This has been moderately extended to handle the situation in which multiple databases exist [11]. Whilst the situation in which multiple applications interact with a database has been considered in a constrained form [20, 19] there does not exist a generalized definition that is applicable to both this situation and the previous ones. Therefore, the following is a general definition of a database system that is applicable to all existing work on database testing:

Definition 1 A database system consists of:

- A collection of database applications P1, P2, . . . , P_n,
- A collection of databases D1,D2, . . . ,D_m,
- A schema ∑ describing the databases.

Conceptually we can view each individual database as a single logical database D that matches the data model ∑. Multiple databases are often used as from an implementation perspective they are easier to understand, manage and optimize. Also, database systems are often not constructed from scratch they often must use existing databases. We do not constrain ∑ to a particular data model, for example relational [15, 16], object–relational [16, 17], object–oriented [16, 18] etc..., however for the remainder of this paper for readability we assume that it is relational. As with the definition of a database system there is no agreed view as to what a database test is, but an informal consensus is beginning to emerge. The following is a definition of database test cases and suites that can form the foundation for the proposals for test adequacy criteria (described in the next section) and for future work. A test case usually involves stimulating the system using some form of input, action or event. The output from the system is then compared against a specification describing what is expected and any faulty behavior identified. In terms of database systems, the concept of a test case becomes more complicated. Not only must we consider program inputs and outputs we must also consider the input and output database states. A database test case must therefore describe what these database states are. For initial database states, existing proposals either adopt an extensional approach [11] or do not consider database state on a per test basis instead specifying a fixed initial database state for all tests [2, 3, 5]. For output states, existing approaches adopt either an extensional approach [11] or intentional approach [2, 3, 5]. A robust approach for testing database systems should specify both initial and output database states intentionally. This allows test cases to be executed on a variety of different states (often real world or changing states) allowing for more realistic testing. Before justifying this we present our definition of a database test case and then discuss the advantages of an intentional approach:

Definition 2 A test case t is a quintuple < i, $\Delta^i{}_c$, P, o, $\Delta^o{}_c$ > where:

- P the program on which the test case is executed,
- i is the application input,
- Δ^i_c are the intentional constraints the initial database state must satisfy,
- is the application output, and
- Δ^o_c are the intentional constraints the output database state must satisfy.

In this definition P, i and o represent the same concepts as the traditional notion of a test case. The database aspects of the test case are described by constraints $\Delta^i{}_c$ and $\Delta^o{}_c$. We have chosen to specify the input and output database states using intentional constraints as they allow us to address a number of limitations with extensional states. In terms of input states, extensional states are: difficult to store, especially where either database states or test suites are large; difficult to maintain as each state must often be modified to reflect changes to the test case, application or data model; and difficult to ensure they reflect the real–world and changes to the database state that may occur over time. In terms of the output state, extensional states are: expensive to determine if two large states are identical; difficult to maintain as the output state must be modified to reflect changes to the input state and the functionality of the system; and time consuming to manually create states that reflect complex behavior that a test case may exhibit on the initial state. Our intentional technique specifies constraints that a test case must satisfy to determine (a) applicability (if the input state is valid for the test case) and (b) success (if the output state is correct).Consider the following very simple example in which a new customer is added to the database:

Test Case 1:

Add a new customer with <name>, <email> and <postcode>

- Δ^i_c: initial state constraint

no customer C in CUSTOMER has C.NAME=<name>, C.EMAIL=<email> and C.POSTCODE=<postcode>

- Δ^o_c: output state constraint

at least one customer C in CUSTOMER has C.NAME= <name>,C.EMAIL=<email> and C.POSTCODE=<postcode>

This test case is relatively simple and imposes a single input constraint that specifies that no customer should exist in the database that matches the customer to be added. The output constraint specifies that after executing the test case the database should contain exactly one customer matching the customer to be added. We specify exactly one customer in the output constraint as it allows us to cover faults where no customer was added and where multiple customers were added. The use of intentional constraints against a real–world database raised the question of how we can deal with situations in which the initial constraint does not hold. This is important as whilst using a real–world database state provides us with realistic data, we cannot create opportunities for exposing faults that might arise in the future, but which are not present in existing data. A test case aims to test a particular use of a system.

However, database systems exhibit significantly more complex functionality. For example, a sequence of related tasks may be carried out by a user interspersed with tasks of other users. Tasks may also be spread across a number of individual programs. These cannot be captured by the execution of a single test case since our definition of a test case assumes a single program execution. Consider the situation in which a test case t1 adds an item to a shopping cart and t2 increases the quantity of the item added. If t1 does not correctly add the item, it is not possible for t2 to increase its quality. Therefore, the execution of t2 may fail not as a result of a problem with the program but because t2 is dependent upon t1. This dependency problem can be addressed by modifying database state to satisfy the initial constraints. However, this approach has a number of limitations. The simplest are due to the resources required for generating database states. The most important is due to the fact that whilst we can satisfy t2s requirements from t1 we are unsure if t1 has an unforeseen impact on t2. For example, a test case may change part of the database state that can adversely affect the behavior of a subsequent test case.

Therefore, it is obvious that certain behaviors require the execution of individual tests in an ordered sequence. A test sequence s is a sequence of test cases <t1, . . . , t_n >. Each test of the sequences is executed in the specified order. If a test case does not meet its output conditions (the test fails) the user is notified of the failure. The database state is then modified to allow the sequence to proceed. However, the test result of the sequence is flagged to tell the user that it did not execute correctly. This is done, instead of simply stopping the sequence, as the tests still provide a certain confidence in the system. Our approach to test sequences allows an individual test case to exist in a number of test sequences. It can also be observed that test sequences can be used for more than testing complex functionality. It can potentially take a lot of effort to set up a database for a particular test case. If several test cases require similar input databases, then it will be much more efficient to run them all against the same database. For example, consider the situation where a database contains records for customers. In an example sequence, the first test case would create a customer; the second would modify the customer; and the third would delete the customer. Each test case represents important functionality of the system which is all related through the use of the same customer. It is therefore more efficient to use a sequence to group related test cases.

Components of Database Application

Testing a Database application involves the following components. The terminology is as follows:

Test Database D:

The state of an application at the beginning of each test. In general, this state can involve several database instances, network connections, message queues, etc.

Reset R:

It brings the application back into state D. This operation is potentially needed after the execution of a test that updated the database. Since testing changes the state of an application, this operation needs to be carried out in order to be able to repeat tests.

Request Q:

The execution of a function of the application. The result of the function depends on the parameters of the function call (encapsulated in the request) and the state of the test database at the time the request is executed. A request can have side effects. That is change the state of the test database.

Test Run T:

A sequence of requests Q1,..., Qn that are always executed in the same order. For instance, a test run tests a specific business process that is composed of several actions (login, view product catalog, place order, specify payment, etc.). The test run is the unit in which failures are reported. It is assumed that the test database is in state D at the beginning or the execution of a test run. During the execution of a test run the state may change due to the execution of requests with side effects.

Schedule S:

A sequence of test runs and resets. The test runs and reset operations are carried out one at a time; there is no concurrency in this framework.

Failed Test Run:

A test run for which at least one request does not return the expected result. A failed test run indicates a bug in the application program.

3. Scope of Research

Testing the front end is usual way of testing in most organizations. But testing of any application is complete only when front end and back end are tested. Testing database applications thus increase the reliability of the database in any applications. A suite of tests that covers every feature in the database application can be generated, which increases the comprehensiveness of the database application.

This research fully concentrates on testing database application efficiently by generating intermediate database, instrumented software generation, test case generation, test case execution and test outcome verification. The generation of intermediate databases based on some a-priori knowledge about the current production databases without revealing any confidential information. Conventionally, database application testing is based upon whether or not the application can perform a set of predefined functions. Logically, the test database must be reset after each test run is recorded and executed. To reduce the number of resets in the database, the order in which test runs are tested is important. To maintain the order of execution and to avoid the failures in test run slicing algorithm is proposed. By controlling the state of the database during testing and by ordering the test runs efficiently, the time for testing can be optimized. The tests should cover all the query situations and avoid producing undesired results so as to obtain their maximum possible coverage. It also describes a criterion, which is an analysis that computes the corresponding testing requirements, and an efficient technique for measuring coverage of these requirements. This is done by constructing the coverage tree and calculating the coverage percentage for SQL commands. This improves the efficiency of testing of database application.

4. Design and Methodology of Database Testing Framework

The proposed framework for database testing is shown in Figure1. The database testing consists of the four modules intermediate database generation, DbUnit testing, slicing algorithm and coverage algorithm. The DbUnit testing consists of three modules instrumented software, test case generation and output validation.

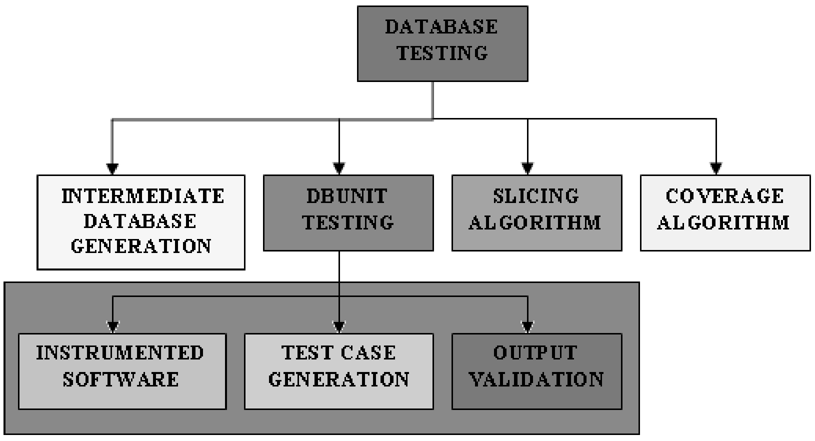


Figure 1. Database Testing

The detailed framework DBGEN is shown in Figure 2.The original database is converted into intermediate database.

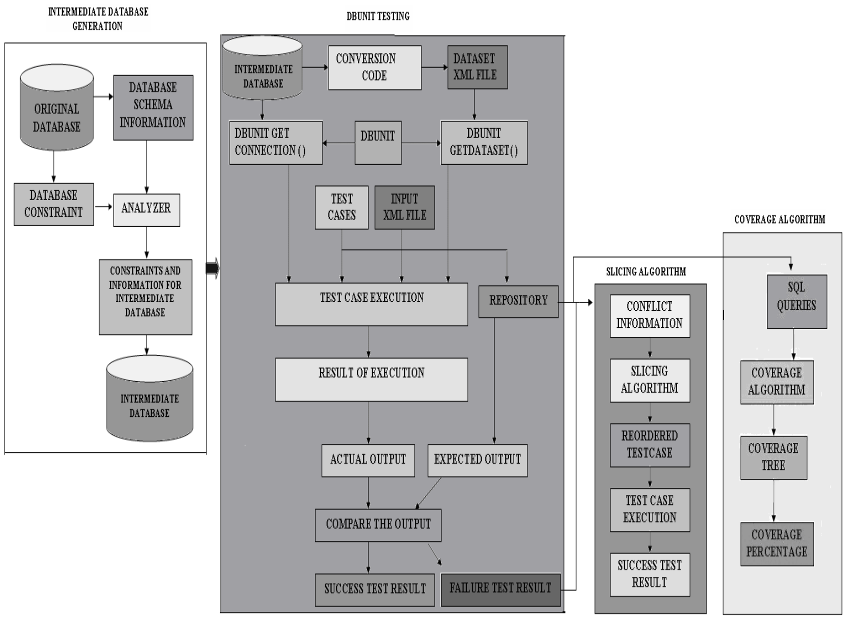


Figure 2. Framework of Database Testing- DBGEN

This intermediate database is used in DbUnit testing. DbUnit tool generally understands only XML statements. Hence there is a need for the conversion of the intermediate database as an xml file and this dataset is got by the getDataSet() method. Database connection in DBUNIT is got by using the getconnection() method. The required test cases are generated, executed and its output verified. A test case may change part of the database state that can adversely affect the behavior of a subsequent test case.

Therefore, it is obvious that certain behaviors require the execution of individual tests in an ordered sequence. In general, the progressive algorithms learn which test runs are in conflict. Based on this conflict information, these algorithms determine an order of test runs with as few resets as possible. The slicing algorithm is proposed to determine the order of test runs. To compute the efficiency of the test cases a metric that calculates the coverage is computed. The SQL query from repository is given to the coverage algorithm, which constructs the coverage tree and calculates the coverage percentage for the test cases.

4.1 Intermediate Database Generation

Testing of database applications is of great importance. A significant issue in database application testing consists in the availability of representative data. The problem is in generating an intermediate database based on a-priori knowledge about a production database. The approach is to fit general location model using various characteristics (e.g., constraints, statistics, rules) extracted from the production database and then generate the intermediate data using the model learnt. The generated data is valid and similar to real data in terms of statistical distribution, hence it can be used for functional and performance testing. As characteristics extracted may contain information which may be used by attacker to derive some confidential information about individuals, it presents disclosure analysis method [6] which applies cell suppression technique for identity disclosure analysis and perturbation for value disclosure.

A block diagram for the first task i.e. intermediate database generation for database application is shown in figure 3.

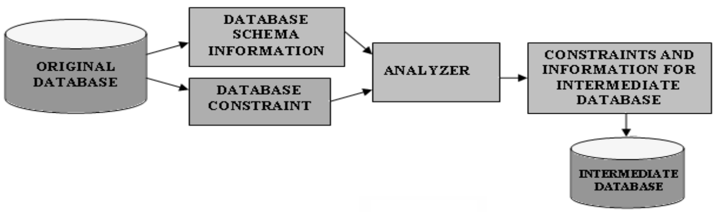

ORIGINAL DATABASE
DATABASE SCHEMA INFORMATION
DATABASE CONSTRAINT
ANALYZER
CONSTRAINTS AND INFORMATION FOR INTERMEDIATE DATABASE
INTERMEDIATE DATABASE

Figure 3. Intermediate Database Generation

In order to make the intermediate database looking closely to the live production database, we can extract some rules and statistical data from the live database and then synthesize random data into the intermediate database according to these rules. In particular, we extract the triplet set <R,NR, S> from the live database in such a way that it will guarantee the generated synthetic data in intermediate databases valid and close looking to real data. We use R, NR, and S to denote deterministic rule set, non-deterministic rule set, and statistics set for a database respectively. The deterministic rule set, R, includes deterministic rules (e.g., domain constraint, uniqueness constraint, referential integrity constraint, functional dependencies, and semantic integrity constraint etc.) while non deterministic rule set, NR, contains non-deterministic information (e.g., association, correlation, pattern etc.). Statistics set, S, contains the statistics about the database instance (e.g., the cardinality of a table, value sets or ranges of each column, the frequencies of column values or statistical distributions etc.).

There are two major problems that need to be addressed:

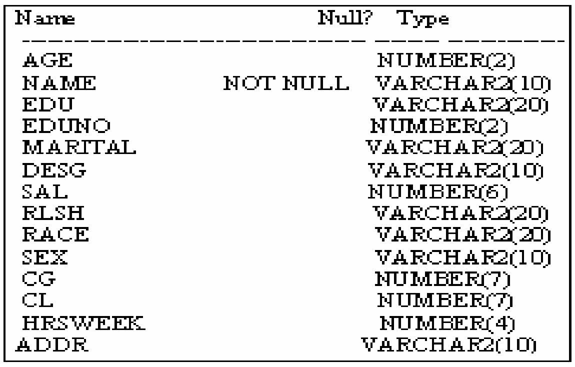
1) Some rules in the triplet set <R, NR, S> may be inaccurate or conflict with another rule due to errors in design or in domain knowledge.

2) Some rules may contain sensitive or confidential information about the database.

Thus the Analyzer component will be applied here to derive an accurate and privacy preserving <R', NR', S'> by hiding or replacing some rules (or statistical data). The information contained in the triplet <R', NR', S'> is the same as the information contained in the intermediate database. Thus it is sufficient to guarantee that the triplet <R', NR', S'> does achieve the three characteristics: valid, resembling (to the original triplet), and privacy preserving (i.e., no confidential information could be inferred from this triplet).

4.1.1 Database Schema and Constraint information

The specification of database testing involves characterizing data values, distributions, and relations. Thus, to achieve the goal of generating valid, close looking data, the users are expected to provide knowledge about the values, distribution, relations, and integrity constraints the data embodies. In this paper we have assumed that databases are based on the relational model. A database in relational model is a collection of one or more relations, where each relation consists of a relation schema and relation instance. The constraints include domain constraint, uniqueness constraint, referential integrity constraint, functional dependencies, and semantic integrity constraint such as business rules It is desirable that the generated data in intermediate databases also satisfy the constraints. For example the census database information is shown in figure4. The employee, department and location table information is shown in figure 5.



```
Name                          Null?    Type
----------------------------- -------- --------
AGE                                    NUMBER(2)
NAME                          NOT NULL VARCHAR2(10)
EDU                                    VARCHAR2(20)
EDUNO                                  NUMBER(2)
MARITAL                                VARCHAR2(20)
DESG                                   VARCHAR2(10)
SAL                                    NUMBER(6)
RLSH                                   VARCHAR2(20)
RACE                                   VARCHAR2(20)
SEX                                    VARCHAR2(10)
CG                                     NUMBER(7)
CL                                     NUMBER(7)
HRSWEEK                                NUMBER(4)
ADDR                                   VARCHAR2(10)
```

Figure 4. Schema information

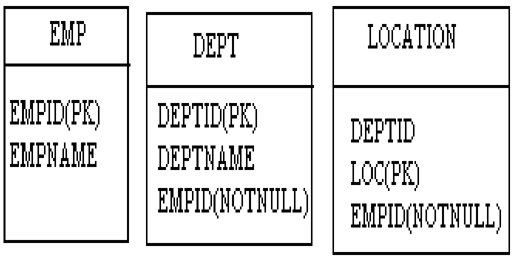


Figure 5. Database Information

4.1.2 Close looking ness and privacy

Two databases DB1 and DB2 are close-looking for application performance testing if the application software cannot tell the difference of the two databases in the sense of performance testing. In other words, for any database application software M, if we run M on both DB1 and DB2 using given test cases x and get the same performance results, then we say that DB1 and DB2 are close-looking for application performance testing [6]. The above intuition about the database close-looking ness can be expressed formally in the following definition.

Definition:

Let DB1 and DB2 be two databases, $x € \{0,1\}^n$ is a binary string representing test cases given by users, t(x) be a time function, and δ(n) be a negligible function 1. We say that DB1 and DB2 are (t; δ)-close-looking for application performance testing if for any nondeterministic Turing machine M, we have

$$\text{Prob}\,[|T(M(DB1, x)) - T(M(DB2, x))| \geq t(x)] \leq \delta(n)$$

Where T (M(DB, x)) is the running time of the Turing machine Mon the inputs DB and x, and the probability is taken over the choices of the input x and internal coin tosses of the Turing machine.

4.1.3 Analyzer

In this section, we discuss effective mechanisms to exclude the confidential information from a triplet <R, NR, S> and to construct a new confidential-information free triplet <R', NR', S'>. In practice, some schema definitions, statistical data, non-deterministic rules, or deterministic rules about the real database as well as domain values for some attributes are considered as confidential information by the database owner. In particular, the confidential information property list may contain the following scenarios about the disclosure of confidential information:

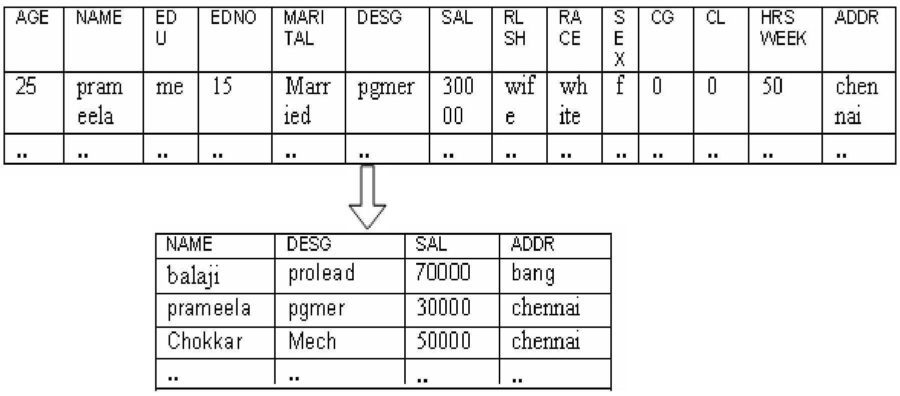
1. **Existence of certain fields and domain values**: For some tables in the live database, the existence of some fields or the name of some fields is confidential information. For example, the existence of a field for the income calculation in census data set had been a secret to the others. Such kind of domain values should be generated randomly.
2. **Direct disclosure of some confidential rules or statistics**: In some applications, some deterministic rules, non-deterministic rules, or statistics about the database are confidential information.
3. **Indirect disclosure of confidential information**:

This includes:

1) Some non-deterministic rules can be used to infer with high probability some deterministic rules or some statistical data.

2) Some statistical data can be used to infer with high probability some deterministic rules or non-deterministic rules.

If the resulting rules or statistical data are confidential, then some rules or statistics should be deleted or revised so that no information about the confidential deterministic rules would be learned from them.



AGE	NAME	EDU	EDNO	MARITAL	DESG	SAL	RLSH	RACE	SEX	CG	CL	HRS WEEK	ADDR
25	prameela	me	15	Married	pgmer	30000	wife	white	f	0	0	50	chennai
..	..	..	..	..	..	..	..	..	..	..	..	..	..

NAME	DESG	SAL	ADDR
balaji	prolead	70000	bang
prameela	pgmer	30000	chennai
Chokkar	Mech	50000	chennai
..	..	..	..

Figure 6a & 6b Intermediate Database Generation

4.2 Testing of Database

To perform Database testing effectively, we have used the DbUnit Tool [21], DbUnit is an open source Framework created by Manuel Laflamme. This is a powerful tool for simplifying Unit Testing of the database operations [21]. It extends the popular JUnit test framework that puts the database into a known state while the test executes.

DbUnit Tool

To effectively generate and execute the test cases, we have used the package DbUnit [22], which is a framework that extends the popular JUnit test framework and puts the database into a known state while the test executes. This strategy helps to avoid the problem that can occur when one test corrupts the database and causes subsequent test to fail. DbUnit provides a very simple XML based mechanism for loading the test data, in the form of data set in XML file, before a test runs. Moreover the database can be placed back into its pre-test state at the completion of the test [22].

Why DbUnit

The reasons to use this testing tool are summarized as follows:

- A framework which simplifies operations for each stage in the life cycle of individual database tests.
- It provides a very simple XML based mechanism for loading test data.
- It provides equally a simple mechanism to export existing test data into the XML format for subsequent use.
- It can work with very large datasets.
- It can help verify your data matches an expected set of values.
- It provides methods for comparing data between flat files, queries and database tables.

Creating a Test Class in DbUnit

DbUnit framework provides an abstract class named DatabaseTestCase which is a sub class of JUnit's TestCase class. Instead of creating a subclass of TestCase class need to extend DatabaseTestCase class. This class provides four abstract methods:

- getConnection()
- getDataSet()
- setUp()
- TearDown()

4.2.1 Instrumented Database Software

The block diagram for the first task in DbUnit Testing i.e. test database generation for database application is shown in figure 7.

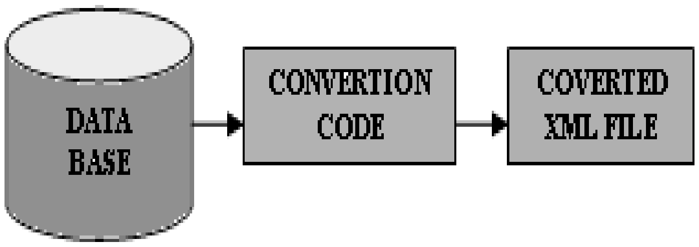
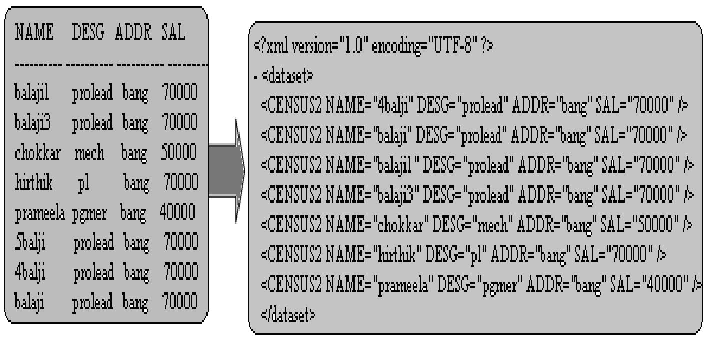


Figure 7. Test Data Generation



```
NAME     DESG    ADDR  SAL
---------- ---------- ---------- ---------
balajil   prolead  bang  70000
balaji3   prolead  bang  70000
chokkar   mech     bang  50000
hirthik   pl       bang  70000
prameela  pgmer    bang  40000
5balji    prolead  bang  70000
4balji    prolead  bang  70000
balaji    prolead  bang  70000
```

```
<?xml version="1.0" encoding="UTF-8" ?>
- <dataset>
  <CENSUS2 NAME="4balji" DESG="prolead" ADDR="bang" SAL="70000" />
  <CENSUS2 NAME="balaji" DESG="prolead" ADDR="bang" SAL="70000" />
  <CENSUS2 NAME="balajil " DESG="prolead" ADDR="bang" SAL="70000" />
  <CENSUS2 NAME="balaji3" DESG="prolead" ADDR="bang" SAL="70000" />
  <CENSUS2 NAME="chokkar" DESG="mech" ADDR="bang" SAL="50000" />
  <CENSUS2 NAME="hirthik" DESG="pl" ADDR="bang" SAL="70000" />
  <CENSUS2 NAME="prameela" DESG="pgmer" ADDR="bang" SAL="40000" />
  </dataset>
```

Figure 8. Conversion Process

Testing the database requires that the data must be in a known-initial state. The database is converted as a XML data set as DbUnit understands only XML. Element names match table names and the attribute names match columns. The developed java code convert the database into an xml file. With the use of schema and information of useful values for attributes provided by the tester, an initial state is generated satisfying the integrity constraints specified in the schema. It takes the advantages of the database schema, which describes the domains, the relations and the constraints the database designer has, explicitly specified. This information is expressed in a formal language, SQL Data Definition Languages (DDL), which makes it possible to automate much of the testing process. Generated XML file representing the database tables and the data within it as shown in figure8.

4.2.2 Test Case Generation

A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. Test cases are generated for Database application.

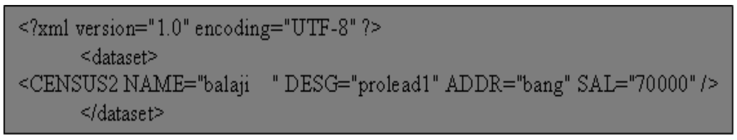
The Testinsert Test Case: This operation inserts the dataset contents into the database. This operation assumes that table data does not exist in the target database and fails if this is not the case. To prevent problems with foreign keys, tables must be sequenced appropriately in the dataset. The testInsert test case gets the input from the newfile.xml. And it inserts this into the database by executing the database operation insert. The input to test case is newfile.xml as shown in figure9.

```
<?xml version="1.0" encoding="UTF-8" ?>
        <dataset>
<CENSUS2 NAME="balaji11" DESG="prolead1" ADDR="bang" SAL="70000" />
        </dataset>
```

Figure 9. Input to Insert Test Case newfile. Xml

The testInsert test case gets the input from the newfile.xml. And it inserts this into the database by executing the database operation insert.

The Testdelete Test Case: This operation deletes only the dataset contents from the database. This operation does not delete the entire table contents but only data that are present in the dataset. The testdelete test case gets the input from the del.xml. And it deletes this into the database by executing the database operation delete. The input to test case is del.xml as shown in figure10.

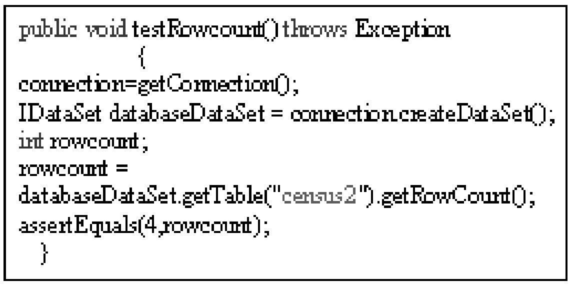


```
<?xml version="1.0" encoding="UTF-8" ?>
        <dataset>
<CENSUS2 NAME="balaji    " DESG="prolead1" ADDR="bang" SAL="70000" />
        </dataset>
```

Figure 10. Input To Delete Test Case

The testdelete test case gets the input from the del.xml. And it deletes this into the database by executing the database operation delete.

The Testdeleteall Test Case: Deletes all rows of tables present in the specified dataset. If the dataset does not contain a particular table, but that table exists in the database, the database table is not affected.**The Testtruncate Test Case**: Truncate tables present in the specified dataset. If the dataset does not contain a particular table, but that table exists in the database, the database table is not affected. Table is truncated in reverse sequence. The sample test case is shown in figure11.



```
public void testRowcount() throws Exception
               {
connection=getConnection();
IDataSet databaseDataSet = connection.createDataSet();
int rowcount;
rowcount =
databaseDataSet.getTable("census2").getRowCount();
assertEquals(4,rowcount);
  }
```

Figure 11 sample test case

The assert method compares data obtained from the database with the data loaded from the XML file by executing the test cases.

4.2.3 Test Case Execution

A Block Diagram for test case execution of a database application is shown in figure 12.

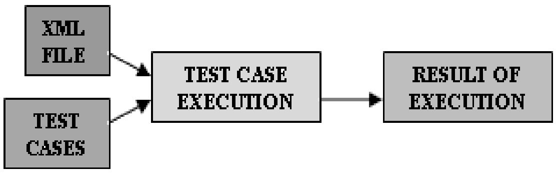
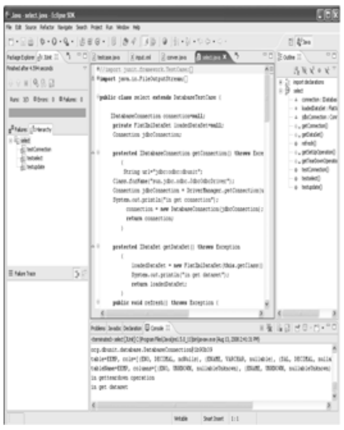


Figure 12. Test Case Execution

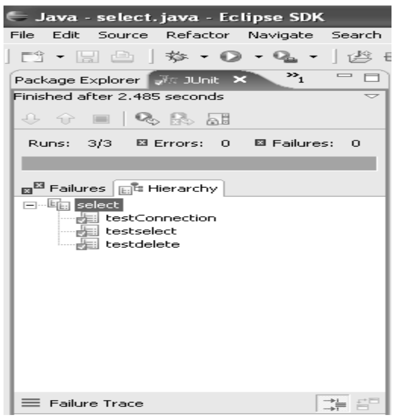
DbUnit includes a mechanism for comparing data loaded from different sources. In this test the assert method compares data obtained from the database with the data loaded from the XML file. In executing test case it compares the actual output to the expected output. The success test case result is shown by green strip in the running environment. The failure test case is shown in the brown strip in the running environment.

The test case execution is shown in eclipse environment in figure13. In the eclipse environment it shows the running test cases in the hierarchy. If testing is successful then a green strip appears at the left of the eclipse window shown in figure14. If any of the tests fails then it turns into a red strip indicating failure of any test shown in figure15.



```
//import junit.framework.TestCase;
import java.io.FileOutputStream;

public class select extends DatabaseTestCase {

    IDatabaseConnection connection=null;
    private FlatXmlDataSet loadedDataSet=null;
    Connection jdbcConnection;

    protected IDatabaseConnection getConnection() throws Exce
      {
        String url="jdbc:odbc:dbunit";
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection jdbcConnection = DriverManager.getConnection(u
    System.out.println("in get connection");
        connection = new DatabaseConnection(jdbcConnection);
        return connection;
      }

    protected IDataSet getDataSet() throws Exception
      {
        loadedDataSet = new FlatXmlDataSet(this.getClass()
        System.out.println("in get dataset");
        return loadedDataSet;
      }
    public void refresh() throws Exception {
```

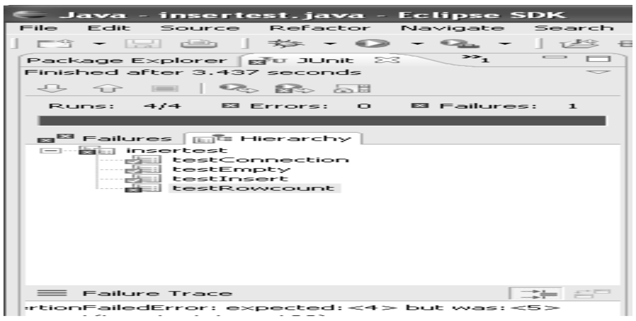
```
org.dbunit.database.DatabaseConnection@[illegible]
[illegible]
[illegible]
in getteardown operation
in get dataset
```



```
Finished after 2.485 seconds
Runs: 3/3   Errors: 0   Failures: 0
select
    testConnection
    testselect
    testdelete
```

Figure 13. Running the Program in Eclipse Figure14. Success Test cases

4.3 Slicing Algorithm



```
Finished after 3.437 seconds
Runs: 4/4   Errors: 0   Failures: 1
insertest
    testConnection
    testEmpty
    testInsert
    testRowcount
rtionFailedError: expected:<4> but was:<5>
```

Figure 15. Failure Test cases

Users interact with a database application. The application provides some kind of interface through which the user issues requests, usually a GUI. The application interprets a request, thereby issuing possibly several requests to the database. Some of these requests might be updates so that the state of the database changes, e.g., a purchase order is entered or a user profile is updated. In any event, the user receives an answer from the application, e.g., query results, acknowledgments, and error messages.

Consider the situation in which a test case t1 adds an item to a shopping cart and t2 increases the quantity of the item added. If t1 does not correctly add the item, it is not possible for t2 to increase its quality. Therefore, the execution of t2 may fail not as a result of a problem with the program but because t2 is dependent upon t1.

This dependency problem can be addressed by modifying database state to satisfy the initial constraints. However, this approach has a number of limitations. The simplest are due to the resources required for generating database states. The most important is due to the fact that whilst we can satisfy t2s requirements from t1 we are unsure if t1 has an unforeseen impact on t2. For example, a test case may change part of the database state that can adversely affect the behavior of a subsequent test case. Therefore, it is obvious that certain behaviors require the execution of individual tests in an ordered sequence. A test sequence s is a sequence of test cases <t1, . . . , t_n>. Each test of the sequences is executed in the specified order [14]. To maintain the order of execution and to avoid the failures in test run the slicing algorithm is proposed. By controlling the state of the database during testing and by ordering the test runs efficiently, the time for testing can be optimized.

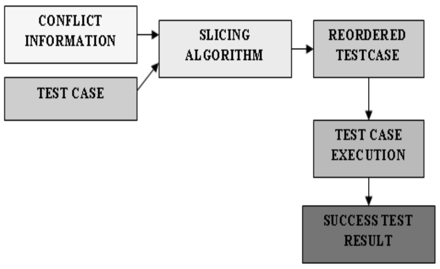


Figure 16. Slicing algorithm

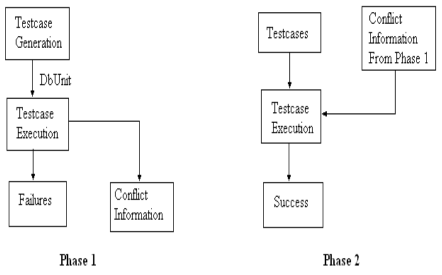


Figure 17. Regression test's test run execution phase

The Block Diagram for slicing algorithm of a database application is shown in figure16. The test case failed due to the conflict with in the database. If we introduce reset in that place then the test case will not fail. In order to reduce the number of resets in the database the slicing algorithm developed. The algorithm reorders the test case so as to minimize the resets in the database testing and execute the test case efficiently.

4.3.1 Testing With Conflict Information

User interacts with the database application in the form of request and receives the answer from the application; e.g., query results, acknowledgments, and error messages. The purpose of the tests is to detect changes in the behavior of an applications or its configuration has been changed. To carry the tests we focus on so-called black-box tests, there is no knowledge of the implementation of the application available [14]. In the first phase, test engineers create test cases. In other words, interesting requests are generated and issued to a test tool DbUnit. The DbUnit executes the test cases generated by the test engineer. If there are any conflicts in executing the test cases means it stores in the conflict database as shown in figure17. We expect the Phase 1 to work correctly so that the answers returned by the application are correct and the new state of the test database is expected to be correct, too. In second phase, as shown in figure 17 we are executing the slicing algorithm with the conflict database. Depending on the conflict information test cases are executed using the slicing algorithm. So that it reduce the no of resets in the test run.

This has two fold advantages. Firstly, building requests into test runs improves the manageability of the regression tests. Secondly, if a whole business process has to be tested, in a specific sequence of requests.

4.3.2 Progressive Algorithms

In general progressive algorithms learn which test runs are in conflict. Based on the conflict information these algorithms determine an order of test runs.

Slice: The Slice approach reorders whole sequences of test runs that can be executed without a reset; these sequences are called slices. The Slice heuristics use the conflict information in order to find a schedule in which as few resets as possible are necessary. The conflict information is gathered .If there is a conflict between test runs < Ti > and T, then Slice executes T before < Ti >. At the same time, however, Slice does not change the order in which the test runs in < Ti > are executed because those test runs can be executed in that order without requiring a database reset. Such a sequence of test runs is called a slice. The Slice heuristics can best be described by an example with five test runs T1, . . . , T5. Initially, no conflict information is available.

Assume that the random order execution of test runs results in the following schedule:

R T1 T2 T3 R T3 T4 T5 R T5

From this schedule, we can derive two conflicts:

< T1T2 >! T3 and < T3T4 >! T5.

Correspondingly, there are 3 slices:

< T1T2 >,< T3T4 >, and < T5 >.

Based on the conflicting information in the conflict database and the collected slices, Slices executes T3 before < T1T2 > and T5 before < T3T4 > in the next iteration. In other words, the test runs in the following order: T5 T3 T4 T1 T2. Let us assume that this execution results in the following schedule:

R T5 T3 T4 T1 T2 R T2

In addition to the already known conflicts, the following conflict is added to the conflict database: < T5T3T4T1 >! T2. As a result, the next time the test runs are executed, the Slice heuristics try the following order: T2 T5 T3 T4 T1. The Slice heuristics reorders the test runs every iteration until reordering does not help anymore either because the schedule is perfect or because of the cycles in the conflict data.

4.3.3 Slicing Algorithm Implementation

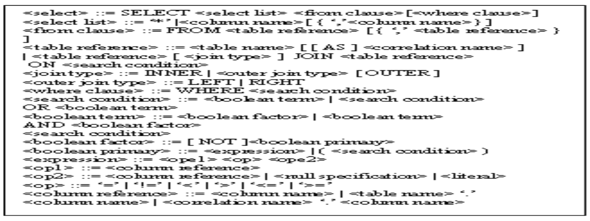
This paper implements the Slicing Algorithm, census database is considered for implementation for testing. testInsert() and testRowcount() are two dependent test cases designed using DbUnit, which is an extension of JUnit. In testInsert(), a check is done whether the entered record is inserted. In testRowcount(), a check is made for the number of rows in the table. When a new row is inserted into the database, the number of rows will be changed. On running testRowcount() after testInsert() it will result in failure even when there are no errors. Thus there is a conflict between the two test cases. This information is entered into the conflict database. In the second run, conflict information is used to find whether there is a conflict between test runs. If so the test cases are re-ordered. In the above example testRowcount() is placed before testInsert(), and now there will be no failure. The failed test run as shown in figure15 is due to the conflict of the row count and the insert. This failure is modified by doing slicing in the test case. The algorithm rearranges the test case that is conflict with one another. By doing repeatedly this reduce the number of resets in the database. From the table it is obvious that slicing algorithm reduces the number of RESETs in the Schedule.

Table 1. Analysis of RESETs

Original form (false positive)	RESET, testInsert(), testDelete(), testRowcount()
Introducing Reset	RESET, testInsert(), RESET , testDelete() RESET, testRowcount()
Slice Algorithm	RESET, testRowcount(),testDelete(), testInsert()

4.4. Measurement of Coverage

The coverage metric establishes a way of measuring the coverage of an SQL query based on the coverage concept whereby the conditions takes into account the true and false values during the explorations of their different combinations. Given the variety of SQL statements that can be found in an application, subset of SELECT queries specified in the grammar in BNF notation shown in Figure18 in order to first achieve testing with simple SQL queries, to subsequently extend the analysis to other, more complex queries.



```
<select> ::= SELECT <select list> <from clause>[<where clause>]
<select list> ::= '*' |<column name>[ { ','<column name> } ]
<from clause> ::= FROM <table reference> [ { ',' <table reference> }
]
<table reference> ::= <table name> [ [ AS ] <correlation name> ]
| <table reference> [ <join type> ] JOIN <table reference>
 ON <search condition>
<jointype> ::= INNER | <outer join type> [ OUTER ]
<outer join type> ::= LEFT | RIGHT
<where clause> ::= WHERE <search condition>
<search condition> ::= <boolean term> | <search condition>
OR <boolean term>
<booleanterm> ::= <boolean factor> | <boolean term>
AND <boolean factor>
<search condition>
<boolean factor> ::= [ NOT ]<boolean primary>
<boolean primary> ::= <expression> | ( <search condition> )
<expression> ::= <ope1> <op> <ope2>
<op1> ::= <column reference>
<op2> ::= <column reference> | <null specification> | <literal>
<op> ::= '=' | '!=' | '<' | '>' | '<=' | '>='
<column reference> ::= <column name> | <table name> '.'
<column name> | <correlation name> '.' <column name>
```

Figure 18. Simplified BNF grammar of SELECT query.

4.4.1 Coverage Algorithm

The coverage algorithm searches for SQL query situations covered with the data stored in the database which evaluates the conditions of SELECT queries that are in the FROM clause, when they include JOIN, and in the WHERE clause. Moreover, the null values of fields will be verified at the same time as the conditions are evaluated. The flow of coverage algorithm is shown in figure19.

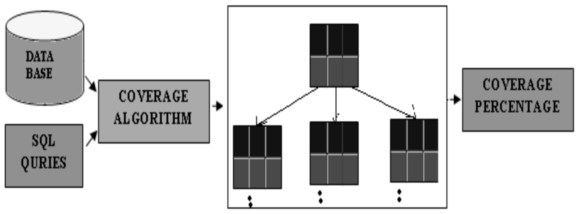


Figure 19. Flow of Coverage Algorithm

As for the inputs:

- Conditions of the SELECT query. The coverage tree will be formed on the basis of these.
- Database structure: tables and columns that appear in the query.
- Data or tuples from the tables: these will be the values used for the evaluation of the conditions.
- The outputs obtained by the process are:
    - o After executing the program, the percentage of coverage of the SELECT query can be determined using the coverage tree, achieving 100% coverage if all possible situations have been verified at any time.
    - o During the evaluation of the coverage tree, a trace of those tuples that give new values for nodes is generated. By revising this information, a subset of tuples can be obtained that supply at least the same coverage as the original data, and that can drastically reduce the size of the test database.
    - o Unevaluated nodes are highlighted taking into consideration the coverage tree. By observing their conditions, their parent information, the database structure and the tuples, the expert can be guided in finding the information missing from the test database to cover all possible cases.

Conditions are not evaluated between a single pair of values, but between sets of values, since the information in each field corresponds to a column from a table and several rows in the database. Therefore, during the evaluation of a condition, each value in the first field must be compared with each one in the second field and each value in the second field with each one in the first, as shown in Figure20.

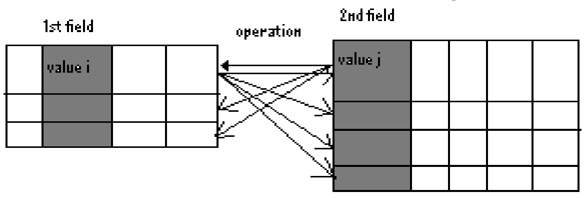


Figure 20. Operation between values of two fields.

The coverage algorithm for search of SQL query situations covered with the data stored in the database is to evaluate the conditions of SELECT queries that are in the FROM clause, when they include JOIN, and in the WHERE clause. The coverage algorithm is shown in figure 4.12.

4.4.2 Coverage Tree

A tree structure, called coverage tree, is created prior to coverage evaluation, in which each level represents a condition of the query beginning with the conditions of the JOIN clause, if it exists, and then with those of the WHERE clause, in the same order in which they are found in the query. The node structure of the coverage tree is shown in figure21.

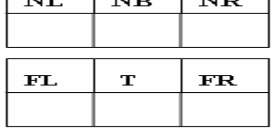
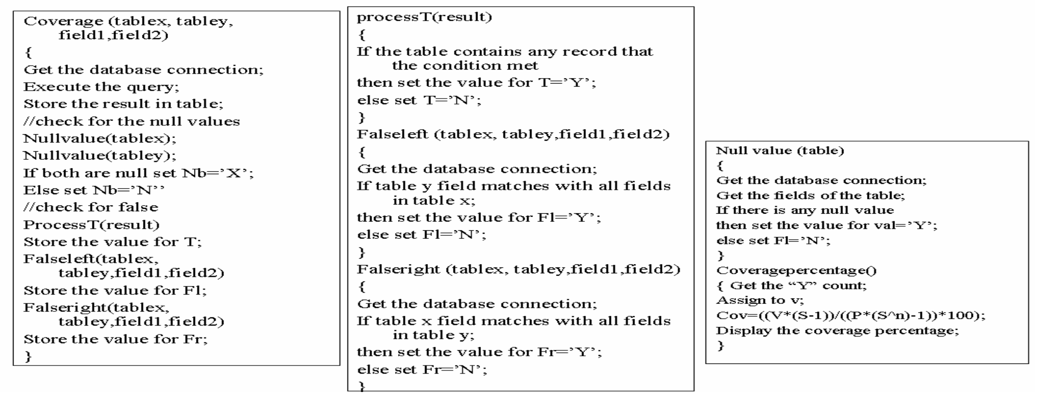


Figure 21. Structure of Coverage Tree

Each node of the tree will store:

- Whether the condition is true for values of the fields it is represented as "Y" in the coverage tree as *T*. Otherwise it is represented by "N".
- Whether the condition is false for values of the fields it is represented in the tree as *Fl* and *Fr*. It is represented by "Y". Otherwise it is represented by "X" Note that, in this case, it is necessary to consider a different treatment for the cases in which the condition is evaluated from left to right and from right to left.
- Whether there are null values in condition fields in the database. This information will then be included in the coverage tree as Nl, Nr and Nb. If there is null it is represented by "X". Otherwise it is represented by "N"

The coverage algorithm used to construct the coverage tree and coverage percentage is shown in figure22.



```
Coverage (tablex, tabley,
      field1,field2)
{
Get the database connection;
Execute the query;
Store the result in table;
//check for the null values
Nullvalue(tablex);
Nullvalue(tabley);
If both are null set Nb='X';
Else set Nb='N''
//check for false
ProcessT(result)
Store the value for T;
Falseleft(tablex,
      tabley,field1,field2)
Store the value for Fl;
Falseright(tablex,
      tabley,field1,field2)
Store the value for Fr;
}
```

```
processT(result)
{
If the table contains any record that
      the condition met
then set the value for T='Y';
else set T='N';
}
Falseleft (tablex, tabley,field1,field2)
{
Get the database connection;
If table y field matches with all fields
      in table x;
then set the value for Fl='Y';
else set Fl='N';
}
Falseright (tablex, tabley,field1,field2)
{
Get the database connection;
If table x field matches with all fields
      in table y;
then set the value for Fr='Y';
else set Fr='N';
}
```

```
Null value (table)
{
Get the database connection;
Get the fields of the table;
If there is any null value
then set the value for val='Y';
else set Fl='N';
}
Coveragepercentage()
{ Get the "Y" count;
Assign to v;
Cov=((V*(S-1))/((P*(S^n)-1))*100);
Display the coverage percentage;
}
```

Figure 22. Coverage Algorithm

4.4.3 Calculation of Coverage

The complete evaluation of the query is carried out by crossing over the tuples of the tables that participate in the conditions at each level of the coverage tree. The evaluation finishes when the entire tree has been covered, i.e. 100% coverage has been covered, or when there are no more values for comparing. For each particular node, the condition is evaluated for a tuples from the first field and another from the second, and:

- If the result is true, these tuples are fixed in order to evaluate the conditions of the lower levels of the tree via the *T* branch.
- If the result is false from left to right, only the tuples from the first field is fixed and, if it is false from right to left, the tuples from the second field is fixed, in order to evaluate the lower levels of the tree, via the branch at which the condition is false, *Fl* or *Fr* respectively.

It is important to fix the tuples, since the same tables, or even the same fields, could appear again at lower levels of the tree, and it is necessary to keep the values of a tuples for the evaluation of all the conditions. After evaluating the coverage tree, the measurement of coverage may be established taking into account the conditions of the SELECT query. The coverage measures are established and automatically calculated:

Theoretical coverage: This takes into account every possible situation at every node.

The percentage of theoretical coverage is calculated using the formula in Figure 23, in accordance with the total number of combinations of values in the conditions and the number of combinations found in the evaluation (*v*). The total number of combinations will be calculated as a function of the number of conditions of the query (*n*), the number of condition values in each node (*p*) and the number of child-nodes of each node (*s*).

$$\%coverage = \frac{v*(s-1)}{P*(s^{n}-1)} *100$$

Figure 23 Coverage Percentage Formula

```
SELECT * FROM
        emp LEFT JOIN dept on
      (emp.empid=dept.empid)
       LEFT JOIN loc on
      (dept.empid=loc.empid) and
      (dept.deptid=loc.deptid)
```

Figure 24 select query

where:

***v*:** number of cases (elements of a node) that it has been possible to verify (those marked with Y).

***s*:** number of child-nodes that a node can have.

***p*:** number of possible values that a condition can adopt once it is evaluated, which in the coverage measurement presented here will have six values (*Nl*, *Nr*, *Nb*, *T*, *Fl*, *Fr*).

***n*:** number of levels of the coverage tree; i.e. the number of conditions in the query

To improve the accuracy of database testing, the coverage metric is used. The proposed algorithm calculates the coverage percentage, thus improves the performance of testing. To perform this, task is to generate the coverage tree for the DDL statements using the coverage algorithm. For this implementation, the query has been chosen. It obtains information about all employees and their respective department, if any, and the location that are working in the employee at that moment. The select query is shown in figure24.From these tables construct the coverage tree for the select query using the coverage algorithm. The coverage tree is shown in figure25.

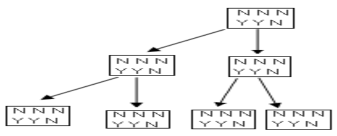


Figure 25. Coverage Tree for Select Query

The first level node creation of the algorithm uses the two table's employee and department. And it checks for the first condition then it created the first node using the algorithm. For each value of T, Fl, and Fr it creates again nodes with the first condition result table and the loc. The level of the tree is depends on the number of condition in the query. The value in the node Fl, T and FR is Y then extend the tree to next three nodes. If any one of the values in Fl, T and FR is N then the next node is not generated for that value node.

5. Implementation

For implementation of testing database application chosen the census data, employee data and University data from the UCI machine learning repository. The database application for testing is shown in table2. The Census Income Data Set predicts the income of the person based on census data. This data set is also known as Adult dataset. The Data Set Characteristics is Multivariate. The Attribute Characteristics is Categorical, Integer. Number of Attributes is fourteen. The Area used is social. The Number of Instances is six hundred and eighty two.

Database application	Attributes	Records
Census	7	35
Employee	12	25
University Data Set	9	35

Table 2. Database application for testing

Database application	Attributes	Records
Census	14	682
Employee	10	425
University Data Set	17	285

Table3 Intermediate database application for Testing

This data set is collected from the machine learning repository. A simple database application the census data for the peoples and the employee details of the employee, department and location data are chosen for testing. The employee data set consists of ten attributes. The Data Set Characteristics is Multivariate.

The real time database is very large and it is converted to the intermediate database with the privacy preserving policy. Extract a triplet <*R*, *NR*, *S*> from the live production database such that a mock database generated from this triplet is close-looking to the live production database for database application testing purpose.

Exclude confidential information from the triplet <*R*, *NR*, *S*> and construct a new triplet <*R'*; *NR'*; *S'*> such that this new triplet contains no confidential information about the live production database and a intermediate database generated from this new triplet is also close-looking to the live production database for database application testing purpose. Use an intermediate database generator to generate an Intermediate database from the new triplet <*R'*; *NR'*; *S'*>. The generated intermediate database information is shown in table3.

Database application	XML File
Census	Census.xml
Employee	Emp.xml
University Data Set	Univ.xml

Table 4. XML file for database

Database application	Number of test cases
Census	15
Employee	15
University Data Set	15

Table 5. Number of test cases for database

The intermediate database is used for database testing with the DbUnit tool. The DbUnit tool will support only XML files. So, the database is converted as xml file. The table4 show the corresponding XML file for the database. The test cases are generated for the database operation. The table5 shows the number of test cases generated for the database application. The test cases are executed in the running environment and the conflict was captured. The table6 shows the number of success and failure test cases executed.

Database application	Number of success test case	Number of failure test case
Census	13	2
Employee	12	3
University Data Set	12	3

Table 6. Number of success and failure test cases

The conflict information of test cases is stored in the repository. Depends on this conflict information the slicing algorithm reordered the test cases and minimized the number of resets in the database. The reordered test cases are shown in Table7.

Database application	Original form	Introducing Reset	Slice Algorithm
Census	**RESET**,testInsert(), testDelete(),testRowcount()	**RESET**,testInsert(), **RESET**,testDelete(), **RESET,**testRowcount()	**RESET**,testRowcount(),testDelete(), testInsert()
Employee	**RESET**,testupdate(), testinsert(), testDelete(), testRowcount()	**RESET**testupdate(), **RESET**,testInsert(), **RESET**,testDelete(), **RESET,**testRowcount()	**RESET**,testRowcount(),testDelete(), testInsert(),**RESET**testupdate()
University Data Set	**RESET**, testdeleteall(), testinsert(), testDelete(), testRowcount()	**RESET**testdeleteall(), **RESET**,testInsert(), **RESET**,testDelete(), **RESET,**testRowcount()	**RESET**,testRowcount(),testDelete(), testInsert(),**RESET**testdeleteall()

Table 7. Reordered test cases

The execution time of the test cases calculated not automatically (manually) and automatically for the generated test cases is shown in table8.

The calculation for three conditions: For calculating coverage percentage the value of "v" is obtained by the coverage tree it is fourteen. The "n" in the formula indicates the level here the n value is three. Each node has three child node so the value of "s "is three. The coverage percentage for three conditions is 17.94%.

The calculation for two conditions: The value of "v" is obtained by the coverage tree for two conditions it is six. The "n" in the formula indicates the level here the n value is two. Each node has three child node so the value of "s "is three. The coverage percentage for two conditions is 25%.

Tables	Condition in the query	Coverage Percentage
One	**One**	**50.0**
One	**Two**	**25.0**
Three	**One**	**2.56**
Three	**Two**	**7.79**
Three	**Three**	**17.94**

Table 8. Execution Time of Test Cases

Database application	No. of Test cases	Execution time (manually) seconds	Execution time (automatically) seconds
Census	15	30	10.678
Employee	15	35	15.954
University Data Set	15	15	9.865

Table 9. Calculated Coverage Percentage

The automated testing reduced the time of execution compared with doing not automatically. The coverage percentage of the test cases is calculated by the formula shown in figure23.

Database application	Original Database		Intermediate Database		XMLFile	No. of test cases	No. of success test case	No. of failure test case	Execution time (manually) seconds	Execution time (automatically) seconds	No. of Resets
	Atributes	Records	Atributes	Records							
Census	14	682	7	35	Census.xml	15	13	2	30	10.678	1
Employee	10	425	12	25	Emp.xml	15	12	3	35	15.954	2
University Data Set	17	285	9	35	Univ.xml	15	12	3	15	9.865	2

Table 10. Process of Database Testing

The calculation for one condition: The value of “v” is obtained by the coverage tree for one condition it is three. The “n” in the formula indicates the level here the n value is one. Each node has three child node so the value of “s “is three. The coverage percentage for one condition is 50%. The calculated coverage percentage is shown in table9.For the one table, one condition the percentage is calculated as 50. For the one table, two conditions the percentage is calculated as 25.In this way the coverage percentage is calculated depends upon the tables and conditions in the SQL queries. The intermediate database attributes, records, xml file generated, test case generated, number of success test case, number of failure test case, execution time of automated, non automated testing and number of resets in the slice is shown in the table10.

6. Conclusion

In response to a lack of existing approaches specifically designed for testing database applications, the proposed framework DBGEN discussed here, is able to address various database issues. It's ability to handle constraints like not-NULL, uniqueness, referential integrity, along with handling of transactions concurrency has made it a prominent framework for testing database applications. A method for automatic generation of database instances has been proposed, which can be used for white-box testing. Improvement of such constraint generation tools will help in the generation of database instances, for the selection of test cases to test the databases, as per the semantics of SQL statements embedded in a application program.

Applying regression tests over database application naively, doesn't scale well and places heavy burden on test engineers. They often limit the number of tests that can be carried out automatically. Coverage measures for the coverage of SQL queries have been established, specifically for the case of the SELECT query, that are automatically calculated taking into consideration the information of database the schema constraints and the SQL query. Like the measurement of coverage for imperative and structured languages, this is an indicator that helps improve designed test cases with the purpose of detecting faults in SELECT queries.

This work can be extended by testing SQL queries that involve multiple tables, constraints etc., test runs could be executed in parallel. To improve the efficiency and apply coverage to the various decisions\conditions present in the queries.

References

[1]Yuetang Deng, Phyllis Frankl, David Chays. Testing Database Transactions with AGENDA. ICSE'05, May 15-21, 2005, St. Louis, Missouri, USA.2005 ACM.

[2] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weber. A framework for testing database applications. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pages 147–157, August 2000

[3] David Chays, Yuetang Deng. Demonstration of AGENDA Tool set for testing relational database applications. Proceedings of the 25th International Conference on Software Engineering (ICSE’03) 2003 IEEE.

[4]Yuetang Deng Phyllis Frankl Zhongqiang Chen. Testing Database Transaction Concurrency. Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE’03) 2003 IEEE.

[5] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J.Weyuker. An AGENDA for testing relational database applications. Software Testing, Verification and Reliability, 14(1):17–44, 2004.

[6] Xintao Wu, Chintan Sanghvi, Yongge Wang, Yuliang Zheng.Privacy Preserving Database Application Testing. WPES’03, October 30, 2003, Washington, DC, USA.2003 ACM.

[7] S. K. Gardikiotis, N. Malevris, T. Konstantinou. A Structural Approach Towards the Maintenance of Database Applications. Proceedings of the International Database Engineering and Applications Symposium (IDEAS'04) 2004 IEEE.

[8] M.Y.Chan and S.C.Cheung, Testing Database Applications with SQL Semantics, In the Proceedings of 2nd International Symposium on Cooperative Database Systems for Advanced Applications(codas'99), Wollongong, Australia, March 1999,pp. 363-374.

[9] Ramkrishna Chatterjee, Gopalan Arun, Sanjay Agarwal, Ben Speckhard, and Ramesh Vasudevan. Using Applications of Data Versioning in Database Application Development. Proceedings of the 26th International Conference on Software Engineering (ICSE'04) 2004 IEEE.

[10]Jian Zhang, Chen Xu, S.-C. Cheung ,Automatic Generation of Database Instances for White-box Testing. 2001 IEEE

[11]Gregory M. Kapfhammer, Mary Lou Soffa. A Family of Test Adequacy Criteria for Database-Driven Applications. ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.2003 ACM .

[12] William G.J. Halfond and Alessandro Orso. Command-Form Coverage for Testing Database Applications. 21st IEEE International Conference on Automated Software Engineering (ASE'06) 2006 IEEE.

[13]D. Chays, P. Frankl, et al. "A Framework for Testing Database Application" ACM International Symposium on Software Testing and Analysis, Portland, Oregon, 2000.

[14] A. Kreutz F. Haftmann, D. Kossmann. Efficient Regression Tests for Database Applications. Proceedings of CIDR Conference, 2005.

[15] E. F. Codd. A relational model of data for large shared data banks. Communications of the ACM (CACM), 13(6):377–387, 1970.

[16] T. Connolly and C. Begg. Database Systems. Addison-Wesley, 3 edition, 2002.

[17] M. Stonebraker, P. Brown, and D. Moore. Object-Relational DBMSs. Morgan Kaufmann, 2 edition, 1998

[18] C. L´ecluse, P. Richard, and F. V´elez. O2, an object-oriented data model. In H. Boral and P.-A° . Larson, editors, Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988, pages 424–433. ACM Press, 1988.

[19] D. Willmor and S. M. Embury. A safe regression test selection technique for database–driven applications. In Proceedings of the 21st International Conference on Software Maintenance (ICSM), pages 421–430. IEEE Computer Society, September 2005.

[20] G.-H. Hwang, S.-J. Chang, and H.-D. Chu. Technology for testing nondeterministic client/server database applications. IEEE Transactions on Software Engineering, 30(1):59–77, 2004.

[21] www.dbunit.org

[22] www.sourceforge.net

Authors

Mrs. A.Askarunisa is working as a Senior Lecturer in Thiagarajar college of Engineering, Madurai. At present she is pursuing her PhD in Software Testing. She has published papers in National and International Conferences. Her research interests include software Engineering, Compilers, Architectures.

Ms. P.Prameela is studying second year M.E computer science in Thiagarajar college of Engineering, Madurai. She worked as a lecturer from 2005 to 2007. Her area of interests includes software testing.

Dr. N.Ramraj, the Principal of GKM College of Engineering, Chennai, and Affiliated to Anna University has published many papers in National and International Conferences and journals. His research interests include power systems, data mining, distributed computing, software Engineering, Compilers, Architectures.