Maximizing Code Coverage in Database Applications

TULSI CHANDWANI, The University of Texas at Arlington
SUPERVISED BY: DR. CHENGKAI LI, The University of Texas at Arlington
COMMITTEE MEMBER: DR. CHRISTOPH CSALLNER, The University of Texas at Arlington
COMMITTEE MEMBER: DR. LEONIDAS FEGARAS, The University of Texas at Arlington

A database application takes input as user-defined queries and determines the program logic to be executed based on the results returned by the queries. A change in existing application or a new application is expected pass through extensive testing to cover the entire code and check all the cases possible in execution. Testing the code coverage of traditional or CRUD-based applications is a straightforward process backed by various tools and libraries. Unlike traditional applications, checking the code coverage of database applications is a complex procedure due to its inherent structure and the inputs passed to it. Measuring the code coverage of such programs involves the participation of DBA's to generate mock databases that can epitomize the existing data and trigger as many paths in the program as possible. Recent studies have introduced techniques to directly test the programs using the existing data. In this paper, we are comparing two methods of code coverage that will help software engineers to test their database applications without being dependent on mock databases. We aim to evaluate the performance of Baseline and Leaf Query approaches for testing such applications. Our work focuses on using decision trees as database applications and testing them by using both the methods. The efficiency of each method is assessed on several parameters by conducting experiments to understand the advantages and disadvantages of each technique.

1 Introduction

Testing plays a critical role in the life-cycle of software development. It ensures that a new software or any change in existing software functions as expected by the requirement. One way of testing the program is determining its code coverage. Code coverage is a measure used to depict how much the source code of a program is executed when a specific test suite runs[8]. The more coverage of source code, the more chances to detect software bugs in the application.

Database applications are the programs in which the input can be user-defined queries. The results of the query determine the subsequent logic to be executed and the program flow. This means different query results can cover different parts of the program. Applications like book search and insurance claim take input as a query defined by the user and could be considered as examples of database applications. Testing such applications is considered as a complex procedure due to its definition of working on user-defined queries and determining the branching conditions based on the results of these queries. At the same time, the problem is compounded by the expectation of executing the program on the existing database. In order to address this issue, Leaf Query technique is proposed which is based on Dynamic Symbolic Testing(Li and Csallner 2010)[1]. It works on generating database queries that would fetch results covering various program flows. By covering maximum logic, it aims to maximize the code coverage of the application ensuring proper testing and reducing the software bugs.

The intuition to the technique is to consider the program with various branching conditions as a binary tree. Every distinct program flow in the application thus represents a path in the tree from root to leaf node. We aim to cover the code by generating queries for paths unexplored by the program as the unexplored paths would result in uncovered code. By using the dynamic program execution, we can observe the current execution and use it to generate inverted queries[1] for unexplored paths, thereby maximizing the code coverage of the application.

Apart from Leaf Query, code coverage can also be achieved by Baseline approach. Unlike Leaf Query, Baseline aims at testing code by using all the information in the database and traversing it throughout the program. It

works on the logic that each input will cover some path in the application and by using all the records in the table, we ultimately cover all the code in the application.

In this paper, we focus on comparing both the approaches by testing decision tree applications. Decision tree applications follow the structure of database application and by checking its code coverage, both the approaches can be analyzed. We conducted several experiments on three different datasets to examine code coverage of decision trees. With these experiments, we examined both the approaches and concluded the results with the better approach to be used in testing in specific scenarios.

2 Motivation

Software engineers continuously make new or update existing database applications that are expected to run on production data. Recent studies[4, 5] show using of mock databases created by Database Administrator's (DBA'S) to test database applications however this creates a dependency on them. Software developers need to wait for mocks every time a change is made on the application. Moreover, mock databases need to be carefully crafted so that there is a high representation of the existing production data. Scenarios where Deep Web or hidden databases are used, its difficult to create mock databases.

In modern real-world applications, the underlying databases get new data added constantly. The applications created are required to run on the existing data as well the appended data. Also, continuous addition of data makes creating mock databases difficult. Given this constant append characteristic, engineers need effective ways to directly test their application on the existing database.

Leaf Query and Baseline approaches allow software engineers to bypass the mock database and test the application directly using the existing database. It generates queries to be used on existing data and use the query results to test the application.

In Figure 1[1], method dbfoo() represents a small database application. It takes where clause as the input and creates a query. Depending on the results retrieved from the database, the values passed to method bar() will pass through different branching conditions. In order to cover all the combinations of branches, various database queries with varying results need to be passed as the input to cover all the code in the method bar(). Considering the time and resources required to test the application are limited, the goal is to efficiently generate these queries such that maximum code coverage is achieved with minimum time and resources. Advances in Software Engineering has opened multiple ways to test these applications. Two such techniques are Baseline and Leaf Query, however, the comparative performance of each is not clearly known. In this paper, we aim to compare these two approaches by testing decision tree applications.

3 Database Applications and Decision Trees

Unlike traditional applications, input to a database application is a query. The results fetched from the database by the query are used throughout the application and the values of each record in the result are traversed in the code statements. Such applications usually consist of several conditional branches and loops to be tested. In order to cover maximum conditions, we need to derive inputs such that most code of the application is tested against the inputs.

For our paper, we choose to use decision trees as our database applications. Decision trees follow the structure of database applications where it will have many control statements and loops. Listing 1 shows one such example of a decision tree application. Each tuple passed into the decision tree application traces its path depending on the values of the attributes. The inputs given to the decision tree are select statements with where clause specifying constraints on columns and values. The decision tree applications to be used for testing are created using Weka, a tool used for Data Mining and Machine Learning[17]. A random sample of the dataset is used as the training set and decision tree is created by j48 algorithm. Source code of the generated tree provided by Weka is changed

```
public void dbfoo(String q) { // "db app"
  query = "Select * From r Where "+q;
  tuples = db.execute(query);
  for (Tuple t: tuples) {
    int x = t.getValue(1);
    bar(x);
  }
}
public void foo(int[] arr) {  // "app"
  for (int x: arr)
    bar(x);
}
public void bar(int x) {
  int z = -x;
  if (z > 0) {              // c1
    if (z < 100)            // c2
      // ..
}
```

Fig. 1. Database application and regular application

to execute on the Java-based SQL resultset and used for testing its code coverage. The generated application is then deployed on Baseline and Leaf Query system environments and evaluated against the existing test data to check the code coverage of the application.

```
public static void createQuery(String input) {
    ProgramNetworkIntrusion p = new ProgramNetworkIntrusion();
    p.programEntryPoint(p.rewriteQuery(input));
}
protected void programEntryPoint(String query) {
    resultSet = executeUserQuery(query);
      do {
        processCurrentTuple(resultSet);
        if(countPaths)
          DBTestMethodExploration.nrPathsCovered += 1;
      } while (resultSet.next());
}
protected void processCurrentTuple(ResultSet rs) throws SQLException{
        duration= rs.getInt("duration");
    protocol_type= rs.getInt("protocol_type");
    service= rs.getInt("service");
    flag= rs.getInt("flag");
        ...
        N1947e1be0(duration,protocol_type,service,flag,...);
}
static void N1947e1be0(int duration,int protocol_type,int service,int flag,...) {
        if (srv_count <= 324) {
            Ne6ef4111(duration,protocol_type,service,flag,...);
        } else if (srv_count > 324) {
```

```
            N7a9364176(duration,protocol_type,service,flag,..);
        }
}
static void Ne6ef4111(int duration,int protocol_type,int service,int flag,...) {
        if (same_srv_rate <= 0) {
            N267bd1ea2(duration,protocol_type,service,flag,...);
        } else if (same_srv_rate > 0) {
            N3b5179ba19(duration,protocol_type,service,flag,...);
        }
}
```

Listing 1. Decision Tree Application

4 Code Coverage Methods

4.1 Baseline

One way of ensuring code coverage of the application is to use the data in a static exhaustive way. In the dbfoo() example, this would mean using the select query without where clause[1]. This query results in all the tuples in the database and the entire resultset would be used to test all the branching conditions in the method bar(). Using the baseline approach might not be efficient always. We may or may not have enough resources and time to test the application since real-world applications might contain a huge amount of information in the database. We might come across a situation where the entire data is inaccessible due to security constraints, for example, a database might allow only a specific number of tuples to be retrieved for every query. Database sources like Deep-Web and Hidden databases[1, 6] provide limited query capabilities and interfaces to query the underlying data which makes the baseline approach difficult.

4.2 Leaf Query

Leaf Query uses a software engineering technique, Dynamic Symbolic Execution to automatically generate inputs that aim to achieve maximum code coverage of any program. For database applications, these inputs are in the form of queries. The results of the input queries determine the code covered in the database application. In order to maximize the code coverage, inputs need to be designed in a way such that different execution paths should be covered with each input given.

Symbolic execution[9] (also symbolic evaluation) is a method for investigating a program to figure out what inputs make each piece of a program execute. An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would, a case of abstract interpretation. Dynamic symbolic execution (DSE)[7] is a path-based symbolic execution based on two insights. First, the approach starts by executing program P on some input x, initializing the symbolic execution process with a feasible path. Second, DSE uses values from the execution P(x) in place of symbolic expressions whenever symbolic reasoning is not possible or desired. As DSE combines both concrete and symbolic reasoning, it also has been called “concolic” execution.

In a regular application as shown in Figure1, the method foo() can be tested by DSE to cover all the paths in the program. In order to test all the branching conditions, DSE uses a symbolic variable as the input to the program, traces the execution path taken by the variable and treats the branching conditions as constraints on the variable. Taking intuition from this execution, we have used DSE to consider the branching conditions as

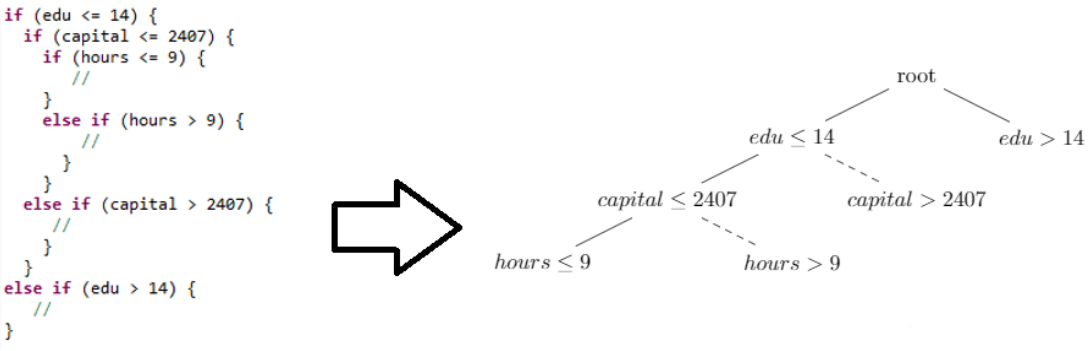

Fig. 2. Execution Paths in Database application

the constraints on the database query. This will allow creating database queries such that the results will pass through different lines of the program and maximize the code coverage.

5 Problem Statement and Method

5.1 Problem Statement

In a database application, a query is taken as input and the results of the query are used to determine the control paths. A path could be defined as the sequence of statements(if-else conditions) that each tuple in the resultset passes through. The application has various paths and the values in the tuple determine which tuple takes which path. Let us consider database program D with a set of paths (P) = { p | p = $c_1 \wedge c_2 \ldots \wedge c_n$ } where c is a condition of the form $a \odot v$ and i=1 to n represents the number of conditions in each path. Our goal is to check how efficiently we can cover most of the paths in P such that the application has been tested for maximum code coverage.

One way of checking code coverage is the Leaf Query approach that follows a dynamic way of testing the application. Another way is Baseline, a brute force testing mechanism. Our aim is to evaluate and compare both the approaches using experiments and suggest a way to test the applications based on the results. For comparison, we use decision tree programs as our database applications and test the application using both the approaches. The efficiency of each method is established by time and tuples required to test the application.

5.2 Overview of the Method

Leaf Query uses Dynamic Symbolic Execution to maximize the paths covered by the test application. In order to get query results, we start with an initial seed query. Each tuple in the query result is executed through the program. While execution, each tuple will cover various branching conditions, hence covering paths in the program. The DSE engine monitors the program execution and path coverage. Branching conditions that remains unsatisfied by tuples constitutes the uncovered paths. The engine analyzes the satisfied conditions and paths and generates new queries with unsatisfied conditions. In this manner, every new query generated will fetch results that will automatically cover unexplored paths in the program. Doing this iteratively will generate new queries that will maximize the code coverage in each execution.

Baseline follows a static approach which doesn't include any analysis by the system. Instead of using a subset the data, all the data is used in checking the control flows of the application. The input query to the database

does not place any constraints on the data. This results in redundant computations as multiple tuples follow the same execution path in the program.The test completes once all the records in the resultset are finished.

5.2.1 Details of the method - Leaf Query

The steps taken in process of Leaf Query are shown in Algorithm 1. The test starts with a seed query and the resultant tuples are passed through the application. Based on the paths covered, a new query is generated and used as input for next iteration which covers more paths in the application. This process is repeated until the stopping criteria are met.

Algorithm 1 Leaf Query

```
1:  q ← Initialize the seed query
2:  repeat
3:      T ← run q and get all the tuples
4:      for each t ∈ T do
5:          run the program over t and cover each path
6:          update the execution tree, add new paths and nodes covered
7:      end for
8:      Q ← set of candidate queries
9:      q ← get the first query from the set Q
10: until stopping criteria satisfied
```

Figure 2 shows a basic database application and its corresponding execution paths. Let us consider the program in the left represents the application to be tested. Each distinct program flow is represented by a path in the tree from the root node to leaf node. While testing the application, the DSE Engine maintains this tree structure to monitor the code coverage. After every query result, the tree explores the paths and appends new branching conditions encountered as new nodes. In order to generate new queries, candidate nodes are determined at each execution. Candidate nodes are unformed leaf nodes which are siblings of existing leaf nodes. They can be viewed of inverse branching conditions of existing branches. For example, in figure 2 after n_{th} execution, node (hours > 9) and node (capital > 2407) could be stated as candidate nodes. The iterations of Leaf Query could be described in following steps:

(1) Iteration 1

In order to start with the execution, we use a seed query that will retrieve 10 random tuples(where 10 is an arbitrary number selected to get initial paths). The initial query used is:

```
SELECT * FROM (SELECT ROW_NUMBER() OVER () AS rowNr,table.* FROM table)
AS tmp WHERE rowNr <= 10
```

The 2 paths covered by the records in the above query could be represented as :

(a) `c1^c2^c3^c4`

(b) `c1^c2^!c3`

After all the tuples are finished executing, Leaf Query approach finds out the candidate nodes, i.e. !c1, !c2, and !c4. After listing the candidate nodes, the first candidate node is chosen i.e !c1 and used to generate the next query. Now, the input to the program is the generated query, using this query tuples are fetched from the database and traversed through the application. Also, the new query generated gets only one tuple from the database following the intuition that we just need to check if there are any tuples satisfying the inverse branch.

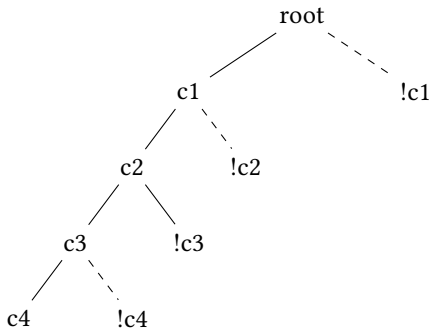

Fig. 3. First Iteration: Leaf Query

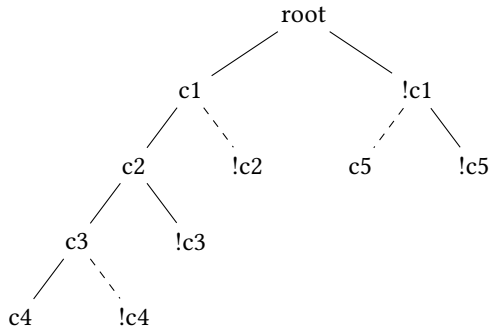

Fig. 4. Second Iteration: Leaf Query

(2) Iteration 2
The new query created to be used as the input:

```
SELECT * FROM table WHERE !c1 limit 1
```

After executing the second query, the maintained tree is shown in Figure 4. The tree gets new nodes c5 and !c5 and covers new paths:

```
!c1^!c5
```

Candidate nodes are updated to !c2, !c4, and c5. !c2 is chosen as from the candidate nodes and the corresponding query is passed as the input to next execution

(3) Iteration 3
Query used as input:

```
SELECT * FROM table WHERE !c2 AND c1 limit 1
```

After executing the third query, the maintained tree is shown in Figure 5. The tree covers new path:

```
c1^!c2
```

Candidate nodes are updated too !c4, and c5. As observed in previous execution, a new candidate node !c4 is chosen and the corresponding query is passed as the input to next execution.

The iterations continue until the tree is complete or all the tuples in the database are exhausted.

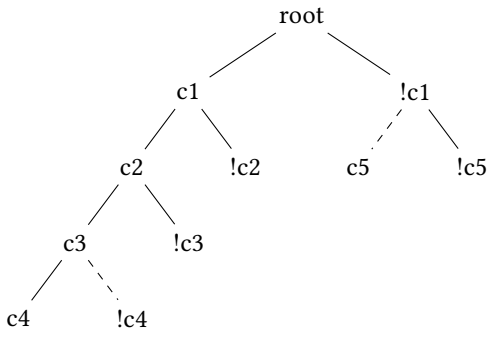

Fig. 5. Third Iteration: Leaf Query

5.2.2 Details of the method: Baseline

The outline of the Baseline method can be described by algorithm 2. All the records from the table are fetched before exploring paths. Each tuple is traversed through the application, satisfying conditions in the program. These tuples cover all the paths that are possible in the program. The testing stops when all the records from the table are finished.

Algorithm 2 Baseline

1: $q \leftarrow$ *Select * from table*
2: $T \leftarrow$ *run q and get all the tuples*
3: **for each** $t \in T$ **do**
4: *run the program over t and cover each path*
5: **end for**

5.3 Stopping Condition

As shown above, in order to continuously explore program flow paths, new queries are generated by inversing branching conditions and the results of the queries are used to cover untraced code. Our technique needs to stop when there are no candidate nodes left in the tree. This would mean we have traced all the possible program flows in our test application. It would also make sense to stop testing if the resources (for example, running out of memory) allocated for testing are exhausted.

5.4 Limitations

We aim to test the applications which use a single table as its database. Moreover, the applications work on tuple data, i.e every query in the program results in a set of tuples from the table and the program logic is applied to each tuple. In this paper, the scope is limited to plain select statements with conjunctive where conditions. Each condition[1] is of the format $attr \odot val$, where attr is the attribute of the table, val is a constant of same data type as attribute, and $\odot$ can be <=, <, >, >=. We also assume the data type of the attributes is the Integer.

6 Experiments

We conducted experiments using three datasets to compare and analyze the performance of Leaf Query and Baseline. For each approach, a system environment was created to deploy and test database applications. Further, the approaches were evaluated on total execution time, the number of queries ran, and the number of tuples required to check the code coverage of the database program.

To analyze Leaf Query, an application was created which uses Dsc[2, 3], a dynamic symbolic engine created in Java to test the database applications. Dsc uses instrumentation method to insert code at the bytecode level. It leverages the instrumentation capabilities of Java 5+ to automatically insert callbacks after specific instructions in the code. These callbacks enable Dsc to maintain a symbolic representation of the program in test. As database applications follow tuple-wise semantics, values of each tuple are used for determining the code coverage. Thus, Dsc tracks every tuple in the resultset fetched from the database. During the execution, each tuple value is represented by a unique symbolic variable. After the execution of the method finishes, a complete symbolic representation of the path taken by the tuple values is obtained. This tracking mechanism is used to monitor the execution tree and create new database queries. The queries generated are used as inputs to the database program to allow exploration of uncovered paths.

For Baseline, a relatively simple application was created to fetch all the records from the given table. Each record is passed through the entire database program. Since it is difficult to get and store the complete resultset for large tables in memory, a database cursor is used to continuously fetch subsets of entire result. For all the experiments, a fixed sized cursor is used that returns 50000 tuples at a time. For every 50000 tuples retrieved from the table, each tuple is passed through all the methods and branching conditions of the application to cover the code. Every tuple follows an execution path and once all the tuples in the resultset are finished, all the execution paths of the database program are covered.

6.1 Experiment 1 - Network Intrusion

Network Intrusion Dataset[14]: This dataset contains records of network intrusions simulated in a military network environment. The dataset contains 4,898,431 tuples and 42 attributes including the class label representing the type of network connection. Each record in the dataset consists of various characteristics of a connection created between two network hosts. The dataset contains connections belonging to 23 class labels including a normal connection and 22 different types of network attacks[16]. These attacks belong to 4 major categories:

(1) DOS: denial-of-service, e.g., syn flood;
(2) R2L: unauthorized access from a remote machine, e.g., guessing password;
(3) U2R: unauthorized access to local superuser (root) privileges, e.g., various “buffer overflow” attacks;
(4) probing: surveillance and another probing, e.g., port scanning.

Procedure and Results: The data is pre-processed by converting the String and Float columns to Integers. String columns *protocol_type, service, and flag* are converted into Integers by using the following criteria:

Mapping of Integers 1-3 for three *protocol_type*
Mapping integers 1 to 70 for 70 types of *service*
Mapping integers 1 to 11 for 11 distinct *flag*

A random sample of 10% of processed data is used as the training data to create the decision tree using Weka. The J48 decision tree generated from the training set contains 157 nodes including 79 leaf nodes where each non-leaf node represents condition on a column and leaf node represents the class label. The Java source code of the decision tree consists of all the conditions on the nodes in the form of if-else statements in functions. This decision tree program generated is used for testing its coverage by Baseline and Leaf Query approaches. The decision tree acts as a database application, in which it takes query input, get the query results and determines

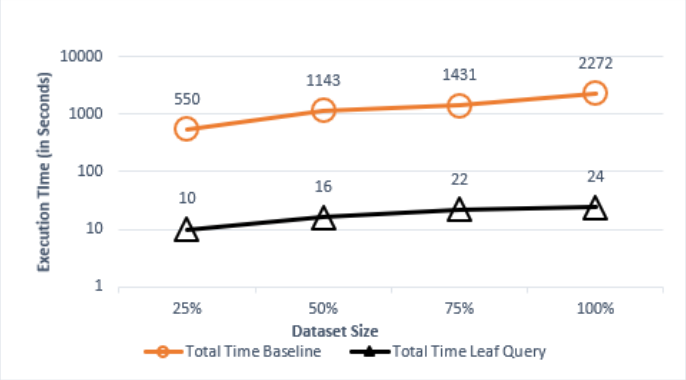

Fig. 6. Execution Time Baseline Vs Leaf Query

the logic to be applied based on the results. The database used for testing the program is created by excluding the sample training data. The application is tested against 25%, 50%, 75%, and 100% of the test set to analyze both the approaches. Leaf Query and Baseline experiments are repeated 5 times to determine the average values of the parameters to be compared. Figure 6 shows a graph representing the execution time taken to test the decision tree program by using Baseline and Leaf Query. The horizontal axis represents the size the dataset used and the vertical axis represents time in seconds log scale. The graph presents the trends of execution time taken against the datasets during the experiment. Similarly, Figure 7 illustrates the relationship between tuples used to test and dataset underlying the program. Baseline approach uses all the tuples in the data and Leaf Query uses maximum of 88 tuples to test varying sizes of the dataset. Also, Baseline will test the program by issuing just one query whereas Leaf Query approximately uses 79 queries (i.e number of leaf nodes in the decision tree) to test the program.

6.2 Experiment 2 - Higgs Process

Higgs Dataset[11]: This dataset is used to classify a signal process which produces Higgs bosons and a background process which does not. Higgs boson is an elementary particle in the Standard Model of particle physics. It is the quantum excitation of the Higgs field, a fundamental field of crucial importance to particle physics theory[12]. The data, produced using Monte Carlo simulations has 11,000,000 instances with 2 classes and 28 attributes including the class label. The first 21 features (columns 2-22) are kinematic properties measured by the particle detectors in the accelerator. The last seven features are functions of the first 21 features; these are high-level features derived by physicists to help discriminate between the two classes[13]. The classes are represented as 1 for Higgs signal and 0 for background noise.

In order to use the dataset for our experiment, the data is pre-processed to convert the Float values to Integers. A decision tree is generated on a random sample of 2% of the data i.e 220,778 tuples. The percentage of data to be used for random sampling depends on the memory capabilities of Weka. In this case, Weka is able to generate tree when training set provided is less than or equal to 2% of the entire data. The complexity of the tree is represented by 220 leaves and 439 nodes. The decision tree generated is tested for its code coverage using Baseline and Leaf Query techniques. The test set is created by remaining 98% of the data. Similar to the previous dataset, the experiment was executed 5 times on 25%, 50%, 75%, and 100% of the test set. The chart in Figure 8

Fig. 7. Number of Tuples Baseline Vs Leaf Query

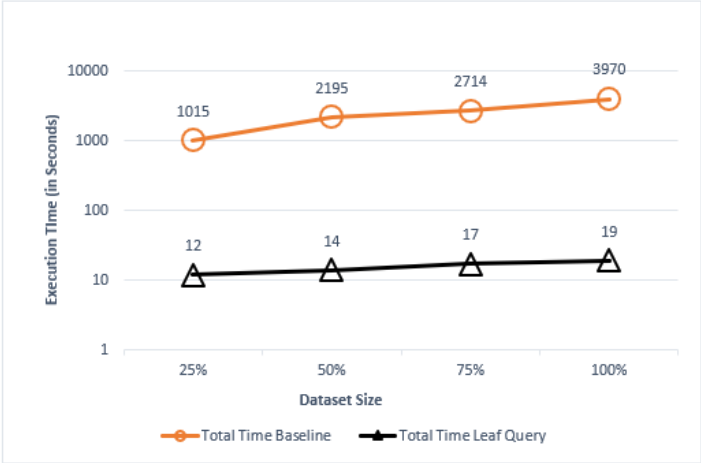

Fig. 8. Execution Time Baseline Vs Leaf Query

displays the effect of dataset size on the execution time required to test the decision tree program. Figure 9 shows the comparison of tuples taken by Baseline and Leaf Query respectively. Results show that Leaf Query uses maximum 220 queries to cover all the paths of the program whereas Baseline always needs one query, regardless of the data and program structure.

6.3 Experiment 3 - New York City Yellow Taxi Trips

New York City Yellow Taxi Trips Dataset [10] This dataset consists of records of all taxi trips completed in yellow taxis in NYC in 2009. It includes fields such as *pick-up and drop-off dates/times, pick-up and drop-off locations,*

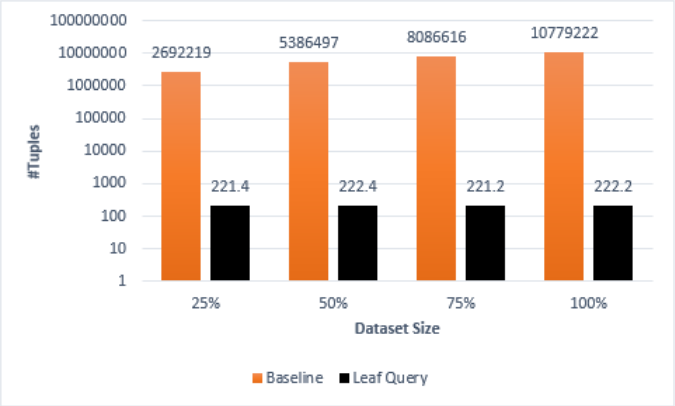

Fig. 9. Number of Tuples Baseline Vs Leaf Query

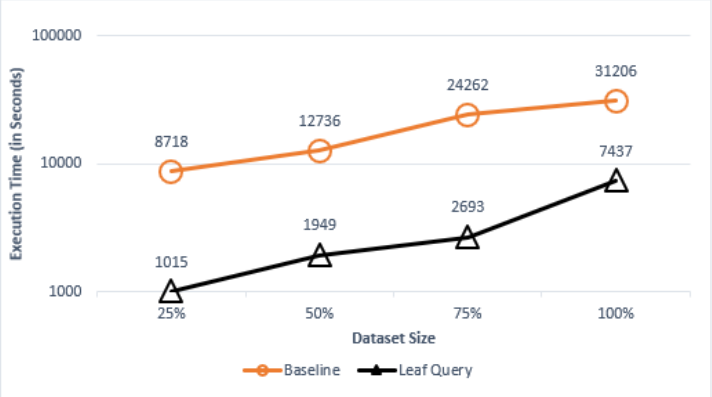

Fig. 10. Execution Time Baseline Vs Leaf Query

trip distances, itemized fares, rate types, and driver-reported passenger counts. The data contains 1,108,779,463 instances and 19 columns including the class *payment-type*. As it was stated in limitations that Dsc allows Integer values to be processed, the data is processed to convert DateTime and Float values to Integer values. Since the size of data is large and Weka has restricted capabilities on dataset allowed to process, only .07% of data (775,408 tuples) was chosen in random sampling for creating decision tree using Weka. The resultant tree had 517 nodes and 259 leaf nodes which corresponded to branching conditions and functions in the decision tree source code. Just like other datasets, the application was tested for code coverage on various size of test datasets to compare both the approaches. Figure 10 shows the execution time taken to check code coverage by Baseline and Leaf Query on 25%, 50%, 75% and 100% of test data. Figure 11 shows the number of tuples taken by each approach for each test set.

Fig. 11. Number of Tuples Baseline Vs Leaf Query

7 Conclusion

In this paper, we attempted to compare two methods of testing the code coverage in database applications. One approach aimed at a static way of testing and another method focused on having a dynamic iterative testing method. Using techniques in this paper, a software engineer can test their database application against the existing data. From the experiments conducted, we can observe that Leaf Query takes significantly less time than Baseline to maximize the code coverage in the database applications. By testing the code of decision tree applications on varying size of datasets, we can deduce that for Baseline, number of tuples used will change according to dataset whereas, for Leaf Query, number of tuples more or less remains same. Further, it is evident that Baseline is a static approach with just one iteration and input to the application and Leaf Query is an iterative approach that generates new queries and sends it as input to the application. It is also worth noting that Leaf Query highly depends on the complexity of the tree, the number of functions needed to test, the branching conditions and data stored in the underlying database.

8 Future work

The systems proposed in this paper allow testing the applications which are based on the Integer-valued dataset. We plan to enhance the systems to accommodate other data types as well. Further, only plain select queries with conjunctive conditions are used as inputs, scenarios where queries are complex and nested need to be incorporated and tested. The inclusion of DML Statements insert, update, and delete into the system needs to be investigated. The effect of having indexes on the table can also be evaluated as it can significantly improve or deteriorate the performance of Leaf Query based on the indexes created.

References

[1] C. Li and C. Csallner. *Dynamic symbolic database application testing*. Proc. of the Third International Workshop on Testing Database Systems (DBTest '10) ACM, Article 7, pages 6. Jun. 2010

[2] M. Islam and C. Csallner. *Dsc+Mock: a test case + mock class generator in support of coding against interfaces*. WODA, pages 26 - 31, July. 2010.

[3] S. Park, I. Hussain, C. Csallner, K. Taneja, B. M. Hossain, M. Grechanik, C. Fu, and Q. Xie. *CarFast: Achieving higher statement coverage faster*. Proc. 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE). ACM, Article 35, pages 1 -

11 Nov. 2012
[4] C. Binnig, D. Kossmann, and E. Lo. *Reverse query processing*. 2007 IEEE 23rd International Conference on Data Engineering, pages 506 - 515 Apr. 2007
[5] M. Veanes, P. Grigorenko, P. Halleux, and N. Tillmann *Symbolic Query Exploration*. Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, pages 46 - 68 Apr. 2009
[6] M. K. Bergman. *The deep web: Surfacing hidden value*. Journal of Electronic Publishing, Volume 7, Issue 1, Aug. 2001
[7] Ball, T., Ball, T., and Daniel, J.
Deconstructing dynamic symbolic execution. Proc. 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering. IOS Press, Jan. 2015
[8] Code Coverage.
https://en.wikipedia.org/wiki/Code_coverage
[9] Symbolic Execution.
https://en.wikipedia.org/wiki/Symbolic_execution
[10] New York Yellow Taxi Trips: Google BigQuery Dataset.
`https://bigquery.cloud.google.com/table/nyc-tlc:yellow.trips`
[11] Higgs Dataset: UCI Machine Learning Repository.
`https://archive.ics.uci.edu/ml/datasets/HIGGS`
[12] Higgs Boson Process.
https://en.wikipedia.org/wiki/Higgs_boson
[13] Baldi, P., P. Sadowski, and D. Whiteson *Searching for Exotic Particles in High-energy Physics with Deep Learning*. Nature Communications 5 July. 2014.
[14] KDD Cup 1999 Dataset.
`http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`
[15] KDD Cup 1999 task.
`http://kdd.ics.uci.edu/databases/kddcup99/task.html`
[16] KDD Cup 1999 task.
`http://kdd.ics.uci.edu/databases/kddcup99/task.html`
[17] Weka.
`https://www.cs.waikato.ac.nz/ml/weka/`