Mutatis Mutandis: Evaluating DBMS Test Adequacy with Mutation Testing

Ivan T. Bowman
SAP AG

ABSTRACT

Testing consumes significant human and machine resources, especially for large, complex systems such as database servers. While a variety of testing approaches have been proposed to improve the efficiency of the testing process, it is difficult to evaluate these approaches. Mutation testing has been proposed as a way to assess the adequacy of a test suite, assigning a score that can be used to compare testing approaches. While promising, serious obstacles appear to prevent mutation testing with large software systems. Recent advances in mutation testing have scaled to medium-sized programs of around 100,000 lines of code but to our knowledge there are no reported studies working with large systems with millions of lines of code and other features of database systems that complicate testing. In this paper we explore using mutation testing on a database server to evaluate its suitability for comparing test suites or testing approaches.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *testing tools*

General Terms

Reliability, Experimentation.

Keywords

Mutation testing, database servers.

1. INTRODUCTION

When developing software systems, it is important to include tests that verify that the system as implemented meets its requirements. Database servers are complex and large software products, and testing them requires significant human and machine resources. In order to maximize the effectiveness of these resources, a number of test generation approaches have been suggested such as random query generation or genetic algorithms. While these approaches are intriguing, it is difficult to compare these testing approaches in order to decide where to invest testing effort.

When evaluating a test suite to assess the reliability of a software system, we would like to know how effective the suite is at detecting faulty versions of the system. We would like to define an adequacy criteria that ranks test suites based on their capability of detecting faults in the system under test. This adequacy criteria can be used to direct testing effort towards developing more useful tests; it can also be used for test suite minimization, offering a way to reduce the number of tests executed by skipping those that don't improve the adequacy metric.

DBtest'13, June 24 2013, New York, NY, USA

One direct approach to evaluating test adequacy is to measure how many faults the test suite can detect. For example, we can monitor which tests detect faults introduced accidentally during development cycles before the tests suites were executed. Test suites that routinely detect more of these *natural faults* may be considered superior.

Natural faults are inconvenient for comparing testing approaches. Fortunately, there are relatively few natural faults, even during early development periods. This relatively small number makes it impossible to draw conclusions of any significance. Further, natural faults would require working with multiple versions of the system with the practical complications of a longitudinal study.

Code coverage is another approach that ranks test suites on how thoroughly they execute the system under test (SUT). Coverage can be measured in various ways (statement, basic block, code paths). While coverage comparison is a natural approach based on the fact that a test cannot detect a fault in code that it does not execute, coverage testing fails to consider how effective the test is at finding possible faults in the code that is covered. Merely executing code without carefully probing the system behavior will not detect subtle faults.

An alternative to natural faults is to artificially generate incorrect versions of the system by *fault seeding*. With this approach, errors are introduced and test suites are evaluated on their ability to detect the faults. While these faulty versions can be generated by hand, an early paper by Lipton [3] suggested using *mutation operators* to automatically generate faulty versions of a program. Mutation testing has a long history of research for evaluating the adequacy of a test suite. However, there are some practical obstacles in applying the proposed techniques to a large and complex system such as a database server. In the remainder of this paper, we describe these obstacles and propose solutions. We evaluate these solutions experimentally using the SAP Sybase SQL Anywhere database server as a system under test.

1.1 Mutation Testing

Mutation testing evaluates the effectiveness of a test suite for detecting incorrect programs. While there are infinitely many incorrect programs, mutation testing focuses on those that are "close" to the correct version. A few *mutation operators* are defined. Each operator modifies the source code for a program P by insertion, deletion or replacement. For example, one mutation operator could change the *and* operator (&&) into *or* (||). A mutant P' is generated by applying a mutation operator at a specific location in the source code and compiling the modified source. A test suite is said to *kill* a mutant P' if it contains a test that passes when run with P and fails when run with P'.

The mutation testing process counts the number of mutants killed by a test suite to give a *mutation score*. When the mutation score is used as an adequacy measure (for example, tests are created until all mutants are killed), we must consider a complication of equivalent mutants. A mutant may be functionally equivalent to the original program even though syntactically different. In this case, it is impossible to generate a test that kills the mutant. This gives an adequacy criteria that cannot be satisfied by any test suites.

The issue of equivalent mutants is challenging because in general it is undecidable whether two programs are equivalent. However, the issue is not important when comparing two test suites (both will have the same denominator (total mutants).

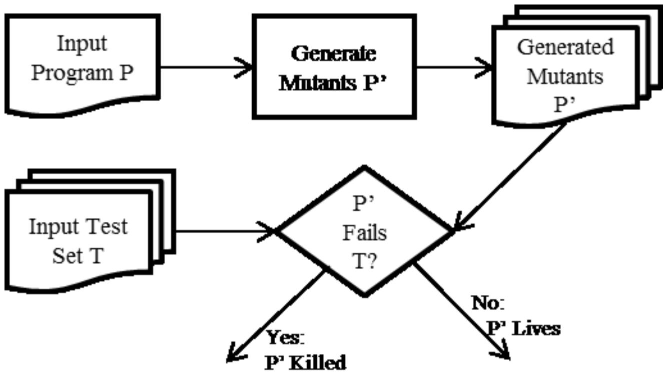

Figure 1 General Approach of Mutation Testing

Besides the issue of equivalent mutants, there are two significant costs to the mutation testing process. First, generating the mutants as separate programs requires a separate compile step for each mutant created. Depending on the number of mutation operators and the size of the source program, the compilation costs may be prohibitive. For our study, this would require over a terabyte of storage for 58,902 mutant binaries. Second, testing the mutants requires running the test suite on each mutant until a test fails. For our environment, this would require at least 25,332 hours.

1.2 Challenges of Testing Database Servers

Given the obstacles to using mutation testing on large systems, it would be much simpler to use some coverage-based adequacy criteria. This could either be statement-coverage or a more targeted definition. For example, Bati, Giakoumakis, Herbert and Surna [1] measured the number of unique function combinations (call from function to function) in order to compare different testing methods.

Coverage based adequacy metrics are relatively easy to compute and they are commonly used for test case selection, prioritization and minimization. However, developers of SQL Anywhere have had serious concerns about using code coverage by itself as an adequacy measure. These developers are reluctant to remove a test from a suite even if it is redundant according to coverage metrics.

Experience with customer-reported software faults contributes to this reluctance to use coverage-based adequacy metrics for test suite minimization. Usually these reported faults in shipping software occur in code locations that are in fact executed by the test suites. The coverage-based adequacy metrics show there is a good level of testing for the affected code, yet software faults are occasionally missed and shipped to customers.

Database systems have a number of characteristics that may impact the use of coverage-based test adequacy measures. In addition to their size and complexity, database servers pose particular testing challenges:

1. **Self-Management Features.** Modern database servers adapt their processing to the current conditions. This adaptation leads to multiple internal states that need to be tested in combination. Tests that verify desired properties of the system may need to be repeated with different internal states.
2. **Relational Equivalence.** A DML statement can be transformed into a number of different semantically equivalent access plans. While logically equivalent, these different plans execute different code paths. Tests that verify that correct answers are returned may miss a software fault that affects only rarely used physical operators (or combinations of operators) leading to intermittent failures. Alternatively, tests may miss flaws in the cost model or query optimizer that lead to the selection of sub-optimal plans. These types of faults can be expensive to detect as they may require large data sets and queries to explore the cost model fully.
3. **Concurrency.** Database systems often manage concurrent requests from different client applications, and they also use intra-query parallelism to execute individual requests. Concurrency control primitives need to be tested; further, concurrent execution introduces more interactions that need to be considered and tested.

These characteristics mean that code within database statements needs to be tested in a number of different configurations. Coverage based adequacy measures don't appear to capture this requirement and this leads to concerns for using these metrics to compare testing approaches.

1.3 Mutation Testing for Database Systems

Mutation testing offers the possibility of a more refined adequacy criteria than coverage based metrics. However, the apparently prohibitive costs of the general mutant testing approach have led to some concerns as to whether it can scale to large software systems.

In this paper, we investigate whether mutation testing can be applied to a database system, and evaluate the benefits of using mutation adequacy measures. We use recent advances in mutation testing and introduce three new techniques that allow us to compute the mutation score for a test suite in a reasonable time and space budget.

2. APPROACH

2.1 Mutation Operators

We define the following mutation operators:

- **Function.** Applies to all methods and C-style functions. When enabled, skip the contents of the function. For functions that do not return a value, this is equivalent to a programmer forgetting to call a function. For those that do return values (directly or through references), this mutation returns uninitialized memory.
- **Condition.** Applies to the condition in `for`, `while`, and `if` statements. When enabled, evaluate a mutated condition.

- **Switch.** Applies to all `switch` statements. When enabled, add one to the *expression*.
- **Case.** Applies to all `case` statements within a `switch`. When enabled, skips the contents of the case (this is equivalent to a programmer forgetting to include a `case` statement for a value.
- **Default.** Applies to `default` statements within a `switch` statement. When enabled, skips the default statements.

A *mutation* is defined to be a single statement in the source program that has been modified by a mutation operator.

2.2 Generating the Mutants

For a large number of mutants, compiling a separate copy of the program is prohibitive in time and space. Instead, we follow an approach suggested by Untch [6] and generate a single program (a *meta-mutant*) that can be dynamically configured at execution time to enable any subset of mutations. With this approach, only one compilation step is needed, and only one copy of the executable program is needed.

The source code is modified to execute the original code unless the mutation is enabled. For example, in Figure 2, the original source line A would be changed to the line in B. Here, the `mutation_on` function checks the array position for this mutation (123 in this example) and returns true if it is enabled.

```
A:if( len > 0 )
B:if(mutation_on(123) ? (len>=0):(len>0) )
```

Figure 2 Example of Mutation Insertion

At test time, the mutation test driver enables the desired mutation(s) and runs tests from the test suite.

Compiling the meta-mutant requires slightly more time and space than the original program, but it requires significantly less than compiling each mutant separately.

2.3 Notation

Given an original system S and a set Ω of mutations that can either be enabled or suppressed, we use S'_M to refer to the meta-mutant system with the set $M \subseteq \Omega$ mutations enabled. A special case with the empty set $M = \emptyset$ gives $S'_\emptyset$ representing the meta-mutant with no mutations enabled. Semantically, this is equivalent to the original system S but it is different because of the extra checks for each mutation to see if it is enabled.

Given a test t_i, we say that t_i *kills* S'_M if t_i fails when executed with S'_M but passes with $S'_\emptyset$. In this case, we write $K(t_i, S'_M)$.

Given a test suite $T = \{t_1 \dots t_n\}$, we say that T *kills* S'_M if there is some test $t_i \in T$ where $K(t_i, S'_M)$. In this case, we write $K(T, S'_M)$.

2.4 Mutation Coverage

The general mutation testing approach executes each test in the suite on each mutant (until the mutant is killed). However, not every mutant is executed by every test. If a test does not execute the code affected by a mutation, running the test on the mutant is wasted work. To avoid this waste, we follow an approach used by Schuler, Dallmeier and Zeller [5] and we use statement coverage information to help direct our testing efforts.

At program startup, a shared memory array (`COV`) is created with one byte per mutation location. The `mutation_on` function sets the corresponding byte to one, indicating the mutation location was executed by the test.

The mutation test driver uses a preliminary step (**MUTANT-COVERAGE**) to evaluate coverage by running each test in the suite once with no mutations enabled. The driver clears `COV` to zero before each test and reads the array after executing the test. We say that a test t *covers* mutation m if, during this preliminary step, the test executed the code location affected by m. During this step, tes execution time is recorded as well.

2.5 Running a Test

We define the function **PASSES**(t, S'_M) to evaluate whether a test $t$passes when executing on the system S' with the mutations in Menabled.

In order to evaluate the test on the modified system, we first enable the desired mutations then start the test, waiting for a maximum amount of time. In our experiments, we limited the test run time to be two times the original test time. This approach prevents an endless loop (resulting from a mutation) from halting testing: such loops are detected as failures.

```
PASSES $(t, S'_M)$
// Returns true if the test passes on system $M$
    Enable only the mutations in $M$
    Start test $t$ on $S'_M$
    Wait at most 2* original test time
    IF test passed
        RETURN TRUE
    IF CRASHED$(S'_M)$ OR TIMEOUT$(t)$
        RESTART-SYSTEM$(S'_\emptyset)$
    RETURN FALSE
```

Figure 3 Determining if a test passes with mutants M enabled

If a test resulted in a system crash or timeout, we restart the system to a known state. Otherwise, we keep the system running. As described below in Section 2.6.2, this allows us to quickly process tests without restarting the server, but it does rely on an assumption that future test behaviour is not affected by this test execution.

2.6 Simplifying Assumptions

We make the following simplifying assumptions. While these assumptions are not true in general, they allow us to greatly reduce the effort of mutation testing.

2.6.1 Independence of Mutations

If M is a set of mutants with $M = L \cup N$ then we assume that T kills M if and only if it kills L or N:

$$K(T, S_M) \leftrightarrow K(T, S_L) \vee K(T, S_N)$$

This assumption allows us to infer that if a test does not kill S_M, then it would not kill any subset of M.

In general this assumption is not true because mutations may interact. However, as we will show this assumption significantly reduces testing cost.

In order to evaluate the impact of this assumption, we chose 50 tests and selected 1000 mutants for each test. For each test, we executed the test once for each individual mutant enabled, then with 500 groups of size 2, 100 groups of 10, 10 groups of 100 and 1 time with all 1000 mutants enabled. The independence assumption held for all but 104 of the 30,550 groups (0.34%). Only 12 of these cases showed a failure with a group where none of the individual mutants showed a failure. The other 92 cases had

```
ORIG-MUTANT-SCORE($T, S$)
// Return ratio of killed mutants to all mutants
    $S', \Omega \leftarrow$ GENERATE-META-MUTANT($S$)
    $L \leftarrow \Omega$        // surviving mutants
    FOR $m \in \Omega$:
        FOR $t \in T$:
            IF NOT PASSES($t, S'_{\{m\}}$)
                $L \leftarrow L - \{m\}$
                BREAK
    RETURN $\left(\frac{|\Omega| - |L|}{|\Omega|}\right)$
```

Figure 4 Original mutation testing algorithm

a failure with an individual mutant that did not appear when the mutant was enabled in a group with other mutants.

2.6.2 *Test failures do not Corrupt State*

A database server is usually expected to be in the running state. If the system needs to be restarted to test each mutant, a significant amount of time will be spent in startup and shutdown code (we measure about 11 seconds per restart. The median test time is 0.4 seconds so this is a significant overhead.

When running a test on a mutant, there are four possible outcomes:

- **Crash.** The system may crash or fail self-checks (assertions)
- **Timeout.** The test may take significantly longer to run
- **Fail.** The system may return an answer that the test rejects
- **Pass.** The test does not report a failure (mutant survives)

When the system crashes or fails self-checks, it is not possible to run further tests without a restart. If a test times out, then it could be due to an endless loop or other ongoing execution within the server that is difficult to detect from the test system. In this case it is best to restart before continuing testing.

When a test executes on a database server without a crash (either passing or reporting a failure), then it is possible that the internal state of the server has been corrupted in a way not detected by the O/S or self-checks. Nevertheless, we assume that this does not happen so that we can execute further tests with the same running server, saving the cost of restarting. In order to eliminate obviously corrupt state, the framework uses a simple “ping” query to ensure the server is minimally responsive and restarts the test process and connections.

2.6.3 *Tests Deterministically Find Faults*

An underlying assumption of mutation testing (and also coverage metrics) is that a test behaves deterministically, executing the same server code paths and either detecting a fault (reporting a test failure) or passing. In reality, we know that the behaviour of the database server is non-deterministic due to self-management features and cost-based plan selection. Further, some of the mutant operators lead to non-deterministic faults that may be equivalent to the original program during some executions while failing in others due to the contents of uninitialized memory.

When evaluating the effect of the independence assumption (Section 2.6.1), we noted the interesting fact that some mutation groups passed a test where individual mutants in the group failed. It is possible this could occur due to mutant interaction where two wrongs (mutations) make a right. Another possibility is that there is a non-deterministic behaviour in the server code that causes a test to have different behaviour on different executions. For the 104 cases that violate the independence assumption, we executed the test case 10 times for each of the mutants that were included in the 104 groups. We found there were 8 mutants that gave non-deterministic answers, explaining 14 of the 104 violations of the independence assumption. Test determinism is an important assumption but it requires further investigation to understand how well it holds in database systems with mutation testing.

2.7 Original Mutant Testing Algorithm

The original mutant testing algorithm defined by Lipton [3] executes the test suite for each mutant until one of the tests in the suite detects a failure. This algorithm is shown in Figure 4.

2.8 Proposed Improvements

A number of improvements have been suggested including using mutant schemata [6] like our meta-mutant and running only tests that execute the code modified by a particular mutant [5].

We propose the following improvements that we believe have not been described in the literature:

2.8.1 *Lethal Mutations*

Some mutations are so severe that any test that executes the mutated code will cause a crash, timeout, or test failure. We propose an algorithm that checks for each mutant whether the cheapest test that executes the mutated code passes. If not, we can quickly kill the mutant using the cheapest possible test.

2.8.2 *Test Ordering*

Some mutants are not completely lethal but they are still killed by a substantial portion of tests. We propose ordering test execution so that cheaper tests are executed first, in the hopes that they will kill mutants quickly.

2.8.3 *Test Independent Mutations Simultaneously*

We observe that individual tests generally do not kill many mutants (we measure an average of 88% of mutants executed by a test do not result in failure). If the mutants are independent and we could enable multiple mutants that are not killed by a test, we can avoid a separate trial for each of the mutants in the set.

We use the independence assumption to infer that PASSES$(t, S'_M) \Rightarrow \bigwedge_{m \in M}$ PASSES$(t, S'_{\{m\}})$. This means that if we test the meta-mutant S'_M with a test t and the test passes, we then infer that the test would pass with any of the individual mutants enabled and there is no need to execute the test on any subset of M. On the contrary, if test t fails when executed with S'_Mthen we are not sure which fault(s) cause the failure.

We propose using this simplifying assumption by using a recursive procedure **TEST-KILLS-MUTANTS** (S', M, t) to find all of the mutants that test t kills. The procedure starts assuming the test will pass with all Mmutants enabled; if so, the test kills none of them. Otherwise, if the set is a singleton, the test kills the individual mutant.

If M contains more than one element, we divide M into two subsets M_1, M_2 of approximately equal size and investigate the behavior of the test with respect to the subsets.

We use the algorithm in Figure 5 to compute the mutation score for a test suite T with a system S. The algorithm begins by compiling a meta-mutant. In our implementation, we applied the five mutation operators at every statement in the subject code that met the operator properties. Next, the algorithm executes each test one time with all mutants disabled, monitoring which mutant locations are executed by which tests. Any tests that fail with no mutants enabled are removed. Total test time is monitored in this step. Next, the algorithm runs one test per mutant to determine if the mutation is lethal.

At this point, the algorithm has a set L of mutants that have survived so far. The tests are considered from fastest to slowest order. For each test, the mutants that are covered by the test are evaluated in **TEST-KILLS-MUTANTS**. The **TEST-KILLS-MUTANTS** procedure begins by assuming that the test will pass with all mutants enabled; if not, it recurses on the subsets to find out which mutants are killed by the test.

```
MUTANT-SCORE($T, S$)
// Return ratio of killed mutants to all mutants
    $S', \Omega \leftarrow$ GENERATE-META-MUTANT($S$)
    $T$, COV $\leftarrow$MUTANT-COVERAGE($S', \Omega$, T)
    $L \leftarrow \emptyset$        // surviving mutants
    FOR $m \in \Omega$ : // kill lethal mutations
        $t \leftarrow$ CHEAPESTCOVERINGTEST($m$)
        IF PASSES($t, S'_{\{m\}}$):
            $L \leftarrow L \cup \{m\}$
    FOR $t \in T$ ORDER-BY TEST-TIME($t$):
        $M_t \leftarrow \{m \in L$ : TEST-COVERS-MUTANT($t, m$, COV)$\}$
        $D_t \leftarrow$ TEST-KILLS-MUTANTS ($S', M_t, t$)
        $L \leftarrow L - D_t$
    RETURN $\left(\frac{|\Omega|-|L|}{|\Omega|}\right)$

TEST-KILLS-MUTANTS ($S', M, t$)
// Return set of mutants killed by test t
    IF PASSES($t, S'_M$):
        RETURN $\emptyset$
    IF $|M| = 1$:
        RETURN $M$
    $M_1, M_2 \leftarrow$ SPLIT($M$)
    $D_t \leftarrow$ TEST-KILLS-MUTANTS ($S_x, M_1, t$)
    $D_t \leftarrow D_t \cup$ TEST-KILLS-MUTANTS ($S_x, M_2, t$)
    RETURN $D_t$
```

Figure 5 Mutant-Score algorithm

3. EVALUATION

We performed experiments on a Dell OptiPlex 990 with an eight core processor (3.4GHz), 16GB of RAM running Microsoft Windows version 7.0 Enterprise.

We worked with a subset of the SQL Anywhere source code related to query processing (scanning, parsing, optimization, scalar expressions and query execution). The subset we selected was over 400,000 lines of code (including comments and whitespace).

Table 1 Mutations and their outcome

	Count	*Covered*	*Killed*	*Lethal*	*Pass*
Function	13,563	75.6%	69.4%	41.1%	30.6%
Condition	35,680	74.8%	65.6%	34.9%	34.4%
Switch	1,176	78.1%	69.0%	34.1%	31.0%
Case	7,528	55.7%	46.1%	27.4%	53.9%
Default	955	24.2%	18.8%	7.9%	81.2%
Total	58,902	71.8%	63.3%	34.9%	36.7%

We applied the mutation operators to every statement in the selected subset that qualified, giving 58,902 mutations. Table 1 shows how many mutations were created by each mutation operator in total along with the percent covered by tests, killed (by a server crash, test timeout, or test failure) or not killed by the test suite (Pass). It also shows the percent of lethal mutations (killed by the cheapest covering test).

We used a single test suite of 617 tests with a total sequential running time of 36 minutes executing on the unmodified system S. Most of the tests are quite cheap (median 0.33s, mean 3.5s) but about 10% take over 20s and 8 tests take between one and two minutes. The tests comprise over 7 million lines of test code (median of 3761 lines per test) and these tests cover 158,676 of the 235,760 statements in the selected code subset (67%).

In the proposed algorithm, we stop evaluating a mutation as soon as it is killed by any test in the suite. For the purposes of evaluation, we evaluated all mutations for each test so that we could find out how many mutations were killed by each test. Each test was executed with a timeout set to twice the run time of the test on the unmodified system.

The original test stream that we work with is normally run in a single sequential stream. This under-uses machine resources on multi-core machines. We modified the test system to run in parallel, with one meta-mutant running and one test being executed per stream. This allowed us to maximize machine resources. We execute each meta-mutant as a separate process with separate data files so that test streams do not interfere.

3.1 Results

3.1.1 Test Time with Proposed Improvements

The original test time with the unmodified system was 14.8 minutes. If we do not use the suggested improvement of testing multiple mutants per trial, we would need an additional 42,214 trials of the entire test suite to prove that each individual mutation was not killed by the test suite (or, alternatively that our assumption was wrong and it was killed). This would require an additional 432.8 days of test execution time, which is not feasible.

We were not able to directly time the execution time with each of the proposed enhancements enabled or disabled. Instead, we simulate the algorithm using the recorded outcome for each (test,mutant) pair, using the original test time as an estimate of the execution time. This is at most out by a factor of 2 due to our timeout rules

Table 2 Run Time (hours) With Proposed Improvements

		Lethal Mutation Step	
Ordering		No	Yes
	No	414.7	54.5
	Yes	49.1	34.2

The total test time with all three improvements is estimated to be 34.2 hours when running in a concurrent suite.

We executed the mutant test framework with all proposed enhancements and found a run time of 65.1 minutes to detect lethal mutations and 21.3 hours to evaluate which mutations were killed by the test suite (this was faster than the predicted time of 34.2 hours because the simulated time predicts more server restarts than were required).

3.1.2 Mutation Score vs. Coverage Adequacy

We are very interested in comparing the mutation score to statement coverage. Figure 6 shows one point for each test with the statement coverage of the test on the X-axis (this is percentage of total statements executed by the test). On the Y-axis, we plot the mutation score for the test.

We observe that there is some correlation between coverage and mutation score. This is not surprising, since mutants can only be killed if the corresponding code is executed. However, the relationship is far from linear (R^2 error is 0.17). Indeed, we see that there are many tests that execute a significant percentage of the statements (up to 30%) but yet kill very few mutants.

We are also interested in the behavior of mutation testing compared to coverage testing when looking at the cumulative adequacy score as tests are executed.

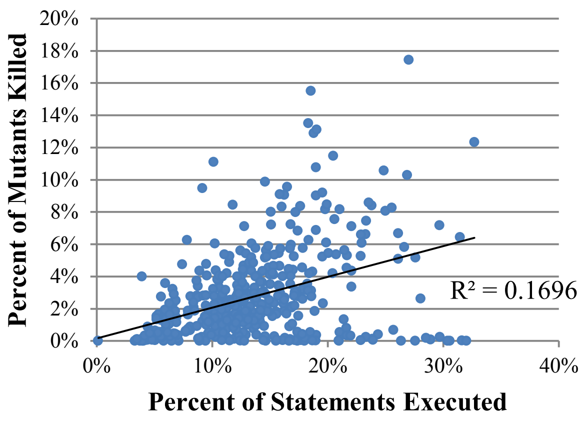

Figure 6 Mutation Score vs. Statement Coverage Percent

Figure 7 compares cumulative statements coverage and mutation score as tests are executed. To make comparison simpler, both are divided by the adequacy score for the test suite after executing all tests. We see that statement coverage increases very quickly to nearly 56% of the maximum coverage after the first five tests. It takes 203 tests before the mutation score reaches the same level. There is a slower increase with the mutation adequacy, but it also increases quickly at the beginning.

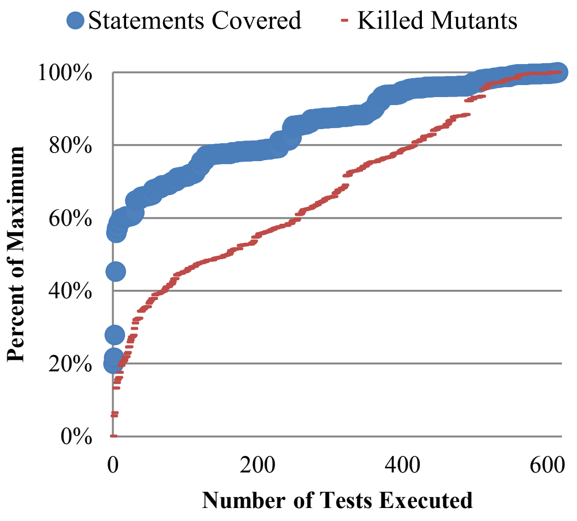

Figure 7 Coverage and Mutation Adequacy Over Time

4. CONCLUSIONS

Mutation testing has been proposed as a general mechanism to assess the adequacy of test suites. Previous studies with small to medium sized programs have shown that mutation outperforms coverage-based adequacy measures at comparing test suites. However, there are high costs associated with mutation testing as proposed in the literature. These costs are excessive and prohibit the techniques from being used with large, complex software systems such as database servers.

In our work, we build on other proposed improvements to scale mutation testing to larger systems and larger test suites. We described three proposals to reduce the overall costs:

1. Detect and remove lethal mutations.
2. Order tests by expected running time to kill mutants efficiently.
3. Enable sets of mutants and assume independence to limit testing of mutants that are not killed by a test.

We used these proposals to implement an algorithm and combined it with practical ideas such as running the test streams in parallel to maximize the benefit of multi-core machines. With these improvements, the human, machine, and time resources needed to perform mutation testing are acceptable: compilation is not noticeably slower; total test time is significant at 21.4 hours, but it is feasible and could be reduced further with more hardware resources for concurrency.

5. ACKNOWLEDGMENTS

Early feedback from other members of the SQL Anywhere Query Processing group was invaluable, especially detailed suggestions from Daniel J. Farrar. We would also like to thank the anonymous reviewers for their helpful and insightful suggestions.

6. REFERENCES

[1] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A genetic approach for random testing of database systems. In *Proceedings of the 33rd international conference on Very large data bases* (VLDB '07). VLDB Endowment 1243-1251.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. Computer 11, 4 (April 1978), 34-41. DOI=http://dx.doi.org/10.1109/C-M.1978.218136

[3] R.J. Lipton, "Fault Diagnosis of Computer Programs," student report, Carnegie Mellon Univ., 1971.

[4] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng*. 37, 5 (September 2011), 649-678. DOI= http://dx.doi.org/10.1109/TSE.2010.62

[5] David Schuler, Valentin Dallmeier, and Andreas Zeller. 2009. Efficient mutation testing by checking invariant violations. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (ISSTA '09). ACM, New York, NY, USA, 69-80. DOI= http://doi.acm.org/10.1145/1572272.1572282

[6] Roland H. Untch. 1992. Mutation-based software testing using program schemata. In *Proceedings of the 30th annual Southeast regional conference* (ACM-SE 30). ACM, New York, NY, USA, 285-291. DOI=http://doi.acm.org/10.1145/503720.503749