A genetic approach for random testing of database systems

Hardik Bati, Leo Giakoumakis, Steve Herbert, Aleksandras Surna
Microsoft Corporation
One Microsoft Way
Redmond WA 98052 USA
{hardikb, leogia, stevhe, asurna}@microsoft.com

ABSTRACT

Testing a database engine has been and continues to be a challenging task. The space of possible SQL queries along with their possible access paths is practically unbounded. Moreover, this space is continuously increasing in size as the feature set of modern DBMS systems expands with every product release. To tackle these problems, random query generator tools have been used to create large numbers of test cases. While such test case generators enable the creation of complex and syntactically correct SQL queries, they do not guarantee that the queries produced return results or exercise desired DBMS components. Very often the generated queries contain logical contradictions, which cause "short-circuits" at the query optimization layer, failing to exercise the lower layers of the database engine (query optimization, query execution, access methods, etc.)

In this paper we present a random test case generation technique, which provides solutions to the above problems. Our technique utilizes execution feedback, obtained from the DBMS under test, in order to guide the test generation process toward specific DBMS subcomponents and rarely exercised code paths. Test cases are created incrementally using a genetic approach, which synthesizes query characteristics that are of interest for the purposes of test coverage. Our experiments indicate that our technique can outperform other methods of random testing in terms of efficiency and code coverage. We also provide experimental results which show that the use of execution feedback improves code coverage of specific DBMS components. Finally, we share our experiences gained from using this testing approach during the development cycles of Microsoft SQL Server.

1. INTRODUCTION

Modern database servers are immensely large and complex software systems. The expressive power of the SQL language combined with the large number of optimization and execution strategies that database systems (DBMS) support today result in a test matrix with practically infinite number of dimensions.

Every major DBMS product release contains several dozens of new features which result in an ever-increasing number of test dimensions. At the same time, as hardware technology advances and becomes more affordable, new database applications include more complicated queries which process increasingly larger amounts of data. For instance, many decision support applications allow the user to define complex queries via query-builder user interfaces. Such SQL queries tend to be complex and large in size.

All these factors make the testing of a DBMS an overwhelming task.

A decade ago functional testing of database systems relied mainly on large test suites consisting of thousands of hand-crafted tests. These tests were created using partitioning and sampling methods over a large number dimensions in the test matrix. Additionally, thousands of queries collected using SQL traces from existing customer applications were also used to prove result correctness and to ensure backward compatibility. Since then, it has become apparent that such testing techniques are neither scalable nor sustainable solutions. The cost of developing new test suites and extending the existing ones is increasing as DBMS become larger and more complex. During the development of SQL Server we saw several cases in which the cost of test development for a new feature significantly exceeded the cost of code development. In addition, there have been a number of cases that indicated that traditional testing techniques on their own are no longer effective; defects were discovered long after they were originally introduced, simply because they involved a rare combination of events or product features.

In order to overcome some of the above problems and limitations of traditional test engineering methods, software quality assurance groups responsible for complex software systems have started employing stochastic testing techniques [14]. These techniques involve the creation of stochastic models that encapsulate the expected behavior of the system under test, and the usage of these models to create a large number of test cases. In the domain of database systems an example of such a stochastic test system is RAGS [13]. RAGS uses a stochastic parse tree to create complex SQL statements that can be utilized as test cases in various ways. Our experience with using RAGS for past releases of SQL Server is that the more complex the generated queries become, the less likely it is that they return results. While such queries can still be useful tests, especially for testing the language parser, they often terminate early during the phases of optimization or execution, e.g. due to logical contradictions in predicates, or empty data intersections. This fact makes these queries less interesting for testing the lower layers of a DBMS such as the query execution, and access methods. Another inherent limitation of random test case generators like RAGS is that it is hard to utilize them when test coverage of specific database components is required. Although controlling the test generation process is possible mainly by controlling the SQL syntax and its complexity, this alone is not enough to exercise specific DBMS components in the desired fashion.

Motivated from our experience using RAGS during the development of SQL Server, we have developed a new system for creating random test cases for testing DBMS and specifically the query processor component.

Our approach enables the creation of complex queries which always return results and contain a set of desired characteristics. These characteristics describe the effect that the execution of the query has on the DBMS, in terms of code coverage or changes in its internal state. We call those characteristics *genes*. The definition of genes is based on information collected from the DBMS after a query is executed. We refer to such information as *execution feedback*. Different sources of information can be used as execution feedback and hence as a basis for query genes, e.g. query results, query plan, traces that expose internal DBMS state, etc. Execution feedback can be customized to meet certain testing goals. For example if we are interested in testing different join implementations, we could use the query plan as execution feedback and the physical join operators as interesting query genes.

Our query generation technique resembles a genetic algorithm. Genetic algorithms are evolutionary algorithms which can be used as a general purpose problem solving technique for various types of problems. A basic trait of evolutionary algorithms is a set of individuals that evolve according to some rules of selection. Evolution takes place using genetic operations like mutation and recombination. In a similar fashion our algorithm creates new queries by mutating and synthesizing queries with interesting gene combinations. A *fitness function* is used to determine whether a newly created query will be used further in the generation process to create more queries. The fitness function can be defined in ways that reflect specific testing goals, e.g. testing of particular DMBS components, or testing of interactions between particular subcomponents.

During the development and test cycle of SQL Server 2005 our technique significantly outperformed RAGS in number of product defects found. In this paper we present results from controlled experiments, which show that our technique outperforms RAGS in terms of code coverage, as well. Additionally, our experiments indicate that the use of feedback increases the test coverage of internal DBMS components when compared to purely random testing.

The remainder of paper is organized as follows: In section 2 we present some of the motivating problems related to query processor testing. In section 3 we provide some information about how random testing is used in SQL Server. We continue with a description of the design and the mechanics of our test case generation system in section 4. Section 5 includes experimental results from three different variations of our test generation technique and provides an evaluation based on code coverage metrics. In sections 6 and 7 we review existing related work and discuss some possible future extensions to our method. Finally, we conclude in section 8.

2. QUERY PROCESSOR TESTING

There are several aspects of DBMS testing that lend themselves to the use of random testing techniques. In this section we will examine some of those related to query processor testing. Although testing the query processor involves multiple testing methods, e.g. performance, reliability, stress, tuning and calibration, etc., for the purposes of this paper we are only interested in functional testing; that is testing which aims to ensure the functional correctness of the system.

2.1 Infinite input space

The practically infinite space of the possible query statements, database schema, data distributions, large number of potential query plan choices, and execution and runtime conditions, makes exhaustive testing impossible. Random testing is a particularly attractive solution for tackling such testing problems (section 6 includes some applications of random testing for tackling large input spaces).

2.2 Dynamic code paths

Two widely used metrics for measuring testing effectiveness are block and arc code coverage. These code coverage metrics aim to ensure that the code is exercised by tests but they don't necessarily reflect the context under which the code executes. For example class and interface inheritance allows object methods to be potentially called by multiple caller methods. Specifically, the concepts of polymorphism and dynamic binding which can be found in all modern object-oriented languages allow *dynamic code paths* which can be formed during runtime. It is desirable to test such code paths in all possible contexts. This is a known problem in the domain of object-oriented software testing [1].

SQL Server's query execution component is based on an abstract iterator interface similar to the Volcano [9]. According to this model the query execution tree is built from physical operators which support a standard iterator[1] interface. These iterators can be thought as stand-alone building blocks for assembling execution trees. Execution trees can become arbitrarily complex, creating a vast space of caller-callee combinations that need to be tested. In addition, each physical operator may contain different implementations or alternative code paths. For example a Hash Join may use main memory or "spill to disk" if the build side of the join does not fit in memory.

Similar types of dynamic code paths can found throughout the query optimizer component.

2.1 Difficult to test in isolation

It is a standard practice to divide large software systems into smaller more manageable subcomponents. The separation into subcomponents also allows unit-testing at the subcomponent level. Unit-testing fits well with today's rapid and agile development practices.

[1] In this paper we will use the terms *iterators* and *physical operators* interchangeably.

Although the boundaries between the main components of a DBMS are well-defined and understood [4], i.e. language parsing, binding, optimization, execution, access methods, etc., testing these components in isolation is hard. Main database components assume that their input is validated by the previous component higher in the stack. Therefore, even though these components and their subcomponents may be well-architected with clear interfaces, the contracts between these interfaces are difficult to verify independently. For example, the primary input to the query optimizer is a tree of logical operators which is typically provided after binding takes place. It is easy to craft a trivial case of such a tree programmatically and use it as a unit test for the query optimizer directly, without having to go through the language parsing and binding layers. However, for non-trivial cases it very quickly becomes hard to verify that the semantics of the unit test are correct. If such a test case uncovers a defect in the query optimizer the test case itself becomes an equally probable suspect for investigation.

For the same reasons, building detailed stochastic test models for a complex system such as the query processor is extremely hard to do beyond a few well- contained areas and subcomponents.

Therefore in practice the development of end-to-end tests (at SQL the language level) is often the only practical option.

3. RANDOM TESTING IN SQL SERVER

Random testing has been an integral part of the testing process of SQL Server since RAGS was first used during the development of SQL Server 7.0. RAGS has been invaluable for testing the SQL parser and compiler components that underwent significant restructuring at the subsequent release (SQL Server 2000). Since then, several other methods of random testing have been explored and used in parallel with regular testing techniques.

According to our experience, random testing provides significant benefits when used in parallel with traditional test development.

First, random testing helps in exploring code paths which are not easily accessible without the development of very complex test cases. In most cases the manual development and maintenance of such test cases is costly.

Second, random test case generators have been useful tools for *smoke testing*; that is quick sanity testing in order to find relatively simple bugs, which were easily exposed by the volume of random queries that are generated. This scenario is most useful in providing an initial quality bar that has to be met before we begin investing manpower on developing, running and verifying manually written tests. As well, this helped us screen risky code changes by ensuring that at least the fundamentals were in place. In some cases we found that extending the query generator tools to support a new product feature prior to the development of regular test cases, allowed us to find some of the non-trivial defects earlier than usual. That allowed the investigation and resolution of these defects to take place in parallel with traditional test development. Moreover, since traditional test development is typically done in incremental fashion, the more complex defects are not found until the very end of the test development cycle, a fact that oftentimes introduces risk to the project schedule.

Finally, we made extensive use of random test generation methods for generating regression test cases. We implemented a solution which allows the archiving of each generated query in a data warehouse along with its characteristics. This enables us to do data mining over the data warehouse in order to choose queries that fit our needs for various regression testing projects.

The technique that we present in this paper is similar to RAGS since it involves the generation of SQL queries, which can be used to generate other types of SQL statements and specific test cases.

Our technique was first developed and deployed as a testing tool for SQL Server 2005. In its early form, the tool was used along with RAGS on an ad-hoc basis to test changes to the query processor. Later we used different variations of feedback for each new feature that involved changes in the query processing layer. During that time, we observed that our test case generator consistently outperformed RAGS, and by the end of the product development cycle had discovered almost ten times more defects. The defects found by random generator tools have been a substantial percentage of the total functional defects found in the query processor component[2]. In their vast majority these defects were raised by SQL Server's self-check mechanisms (assert conditions and debug-only code).

4. METHOD

Our technique utilizes a simple genetic algorithm to create new SQL statements by combining or mutating existing ones with known interesting characteristics/genes. Certain genes are considered to be interesting if they support the desired test coverage goals. In this section we describe architecture of the test system and the mechanics of the test generation process.

4.1 Test system Architecture

The architecture of our test generation system is displayed in Figure 1.

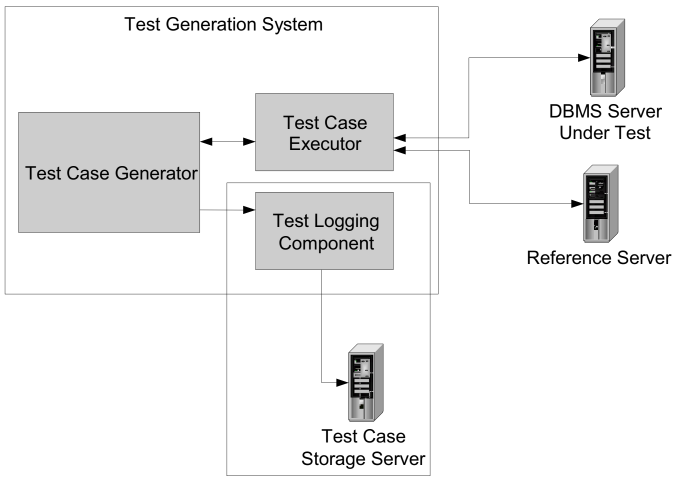

Figure 1. The test generation system

The *Test Case Generator* is the primary query generation component of the system. It analyzes the database schemas in order to extract tables, columns, views, functions, and other items which can be used for query generation. It generates queries on

[2] Specific information related to bug counts is proprietary and cannot be disclosed.

top of this schema and processes execution feedback[3] from these queries in order to evaluate their effectiveness for future use.

The *Test Case Executor* acts as a connection layer between the Test Case Generator and the DBMS Server Under Test (SUT). It is used to execute the generated queries against the server under test.

The *Reference Server* is a trusted reference DBMS which is used by the test case executor to verify correctness of query results. Results are compared against this reference server, and differences are reported for investigation. This part of the process applies to queries that produce deterministic results.

The *Test Logging Component* is used to archive queries and associated information *into* a data warehouse (*Test Case Storage System*) for future use, which can include the assembly of regression test suites.

4.2 Initialization

During the initialization phase the query generator creates a first set of basic queries which are used to populate the *best query pool.* The best query pool contains the query population to be used by the genetic algorithm. The query generator reads the database schema of all the active databases on the server under test (SUT) and randomly chooses up to *k* tables or views. It then creates the initial population of queries as simple SELECT statements. At this stage the generated SELECT statements are trivial; they simply project a randomly chosen number of table columns.

4.3 SQL Statement Reproduction

The creation of a new query statement is done by mutating or combining selected queries from the best query pool. The query generator uses a variety of methods: *query mutation, query simplification, synthesis using join, synthesis using sub-query and synthesis using union.* A random decision is made about which method is to be used and for any parameters required by each method. Figure 2 shows an example of a query based on the TPCH database schema which is generated after 4 iterations of the genetic process.

4.3.1 *Query mutation*

A query is mutated into a new query using a one or more of the following methods, chosen randomly:

- Modification of the projection list, e.g. add/remove one or more columns or expressions, replace a column with an expression, CAST/convert the type of one or more projected columns to another type, etc.
- Modification of the WHERE clause, e.g. addition/removal of predicates
- Addition of aggregates and GROUP BY, HAVING clauses
- Addition of TOP and ORDER BY clauses

[3] We will refer to feedback collected from executing a query/test-case as execution feedback. The term execution feedback is not limited to the query execution component of the DBMS but it can include information from any database component, e.g. language parser, query optimizer, etc.

```
SELECT _s12_ _s13_ ,_n14_ + _n14_ _n15_
FROM
(
   SELECT [L_ORDERKEY] _n16_, [L_PARTKEY]
   _n17_, [L_EXTENDEDPRICE] _n18_, [L_DISCOUNT]
   _n19_, [L_TAX] _n20_, [L_RETURNFLAG] _s21_
   FROM tpch100m.dbo.[LINEITEM]
) t0 RIGHT OUTER JOIN (
   SELECT [O_TOTALPRICE] _n14_, [O_COMMENT]
   _s12_
   FROM tpch100m.dbo.[ORDERS]
   ) t1 ON _s12_ > _s21_ and _n14_ = _n16_
WHERE _s12_ in
        (
        SELECT max(tt._s12_)
        FROM
         (
           SELECT [O_TOTALPRICE] _n14_,
           [O_COMMENT] _s12_
          FROM tpch100m.dbo.[ORDERS]) tt
          where tt._n14_ = t1._n14_
         )
```

Figure 2. A simple query after four process steps

4.3.2 *Query simplification*

Query simplification is a form of query mutation which removes random parts of the query such as predicates or clauses (WHERE, GROUP BY, or ORDER BY, etc). The new query that results may or may not retain all of the genes of the original. However, if it does, the simplified query is preferred and will likely replace the original in the best query pool since it is shorter and more readable. This is the ultimate goal of query simplification – to replace an existing query with an equally interesting one that is either more readable, or executes faster. Since other mutations tend to increase the query text size, simplification is necessary to keep queries from overgrowing into huge, unreadable statements. This aspect of simplification is important since it makes it easier for engineers to investigate and diagnose defects. Note that it is possible for simplified queries to exercise new code paths and create new genes as well.

4.3.3 *Query synthesis using Join*

This method creates a new query out of two or more queries from query-pool by joining them together. It searches the best query pool for queries with one or more columns of the same type. The two queries are joined together using the JOIN clause and the compatible columns (one or more column pairs are chosen randomly) are placed as a join condition in the ON clause. The join condition may include different types predicates. Multiple join types are possible, i.e. LEFT, RIGHT, OUTER, CROSS, etc.

4.3.4 *Query synthesis using sub-query*

This method combines multiple queries into a single query statement by adding an EXIST or NOT EXIST condition on the WHERE clause of the first query and using the second query as a sub-query. Alternatively if the second query returns a single row it can be mutated to one that projects only one column and then placed in the WHERE clause of the first query as a predicate on one of its columns, or in the SELECT clause.

Optionally the above alternatives can be mutated to include *correlation.* This is done by changing/adding the WHERE clause of the sub-query so that it includes a predicate containing a column from the outer query.

4.3.5 Query synthesis using Union

This query generation method uses the UNION clause to combine two queries with compatible projection lists. In most cases there will be no queries with compatible projection lists in the query pool and the query generation process will have to mutate one of the two queries to force the compatibility of the projection lists.

4.4 SQL Statement Transformations

The generated queries can also be transformed to other types of SQL statements such as Data Modification Language (DML) statements, cursor statements, etc. These transformations are different from the reproductions mentioned in section 4.3 since they are *terminal*, i.e. they are not used to generate additional statements.

4.4.1 Transformation to INSERT/UPDATE/DELETE

SELECT statements can be transformed DML statements, i.e. INSERT, UPDATE and DELETE. Transforming a SELECT statement into an INSERT is simply done as an INSERT <*table*> SELECT[…] statement. SELECT statements are transformed into UPDATE and DELETE statements by placing the SELECT statement into the WHERE clause as a sub-query with correlation. The UPDATE and DELETE statements are built using a table name selected randomly from the database schema.

The DML statements are enclosed inside a transaction which is rolled back at the end of the test. Alternatively, the DML statements can be allowed to execute. This will create random changes to the data in the test database and may bring additional test coverage depending on how diverse are the existing data distributions in the test database. However, it will also invalidate some of the genes of the queries in the best query pool making them less favorable for creating new queries.

4.4.2 Transformation to CURSOR

The transformation of SELECT statements to cursors is straightforward. Typically cursors involve different optimization, execution and locking strategies depending on a variety of user-specified options. A set of cursor options is selected randomly by the query generator. The results returned by the cursor can be verified against the ones returned by the select statement for correctness.

4.5 Genetic Algorithm

The outline of the genetic algorithm used by the query generator is shown in Figure 3. The algorithm selects one or more queries from the best query pool and uses one of the aforementioned query mutation and combination techniques to create a new query. The new query is executed and feedback information is collected from the SUT, which should capture all the interesting characteristics of a query. We refer to those characteristics as *query genes*. Query genes can be as simple and accessible as the number of rows returned by a query, or as complex and esoteric as the sequence of state transitions of an iterator during query execution[4]. The feedback information also includes whether the query succeeded or returned an error, along with the result set returned by the SUT. Queries that returned errors are typically discarded unless negative testing is the desired test goal. Queries that return empty results are also discarded and not used for future query generation. This ensures that, unlike RAGS and other tools that do not use feedback, we avoid iterating down paths that have no hope of generating a result due to logical contradictions in predicates or non-correlated joins. Feedback information is either returned along with the query result set or at a subsequent step, depending on what type of feedback information is used. As soon as the feedback information is collected the new query is evaluated for fitness against the best query set population. If it is found fit (or in other words it is found to extend the test coverage) it is added in the best query pool. If the new query is equally fit with an existing query in the best query pool but its SQL text is smaller in size, then the existing query is removed from the pool. Alternatively, the existing query can be removed based on the length of execution time.

```
while timePeriod not expired do
  (Q_1,...Q_k) ← getQueriesFromBestQueryPool(randNum)
  Q' ← generateNewQuery(Q_1,...Q_k)
  resultSet ← executeQuery(Q')
  if resultSet = empty then
    continue
  end if
  feedback ← collectFeedback(Q')
  isFit ← evaluateFitness(Q', feedback)
  if isFit = true then
    addReplaceQueryToBestPool(Q')
  end if
end while
```

Figure 3. Genetic algorithm for query generation

The new query contains combinations of genes inherited from queries in the best query pool, and as soon as the new query itself enters the best query pool, its set of genes will become input for later iterations of the query generation process. We allow unlimited best query pool growth, and empirically we see that it typically contains somewhere on the order of a few hundred queries.

4.5.1 Feedback and fitness function

Feedback is consumed by the generator in the form of a set of strings describing the coverage achieved while running particular query. Each string describes an "interesting" code path exercised, as well as context information about under which circumstances given code path was reached. Some simple examples of feedback strings (translated in human-readable form) are:

- *"exercised hash join" + "parallel query plan"*

[4] In some cases the SUT needs to be instrumented or modified in order to expose internal state and events.

- *"executed part X of bitmap filtering code" + "3 join columns involved.*"
- *"reached line 555 in file hash.cpp"*.

The genetic process doesn't parse feedback strings but just considers them to be important genes to be tracked and makes sure they are not get lost. For every such string/gene the generator remembers which query was able to achieve it. The total number of possible feedback strings usually is in order of hundreds to few thousands. Along with feedback strings the generator remembers the frequency of appearance of this string.

During the process of mutating and combining queries, preference is given to queries that include rarely seen genes, i.e. unique feedback strings. This achieves two goals: first, rare code paths are exercised more frequently, and second, mutations of queries with rare code paths are more likely to exercise other rare code paths. As an example: suppose that a particular rare gene indicates some uncommon data type was processed by this query. Mutations of this query are likely to still process the same uncommon data type, but will do so in a different context (such as a different join flavor) and thus are likely to touch other rare code paths related to the data type.

The fitness function decides whether query goes into best query pool or not. If query includes a gene that we see for the first time, then that query always passes the fitness check and is added to the best query pool. Otherwise the fitness decision depends on whether a new query is more readable/shorter or faster.

The fitness function can be customized so that particular feedback strings are given additional weight, or specific logic can be added to make particular gene combinations more fit than others.

The following two sections discuss two testing problems and the feedback and fitness function used to tackle them.

4.5.2 Testing physical operators

As we mentioned SQL Server uses an iterator model for implementing physical operators and query plans, similar to the one of Volcano. Oftentimes, during the product development cycle new code changes are introduced that may affect multiple iterators. Such changes generate the need for testing those iterators in the context of different query plans; that includes their possible placement in a query plan, whether they run in parallel or serial execution mode, etc. As an example let's assume that we are introducing a new large object data type (LOB) to DBMS. This new feature requires code changes across multiple physical operators. Our test goal, described in simple terms, is to *ensure that every physical operator is tested in combination with LOB columns*. For this particular testing problem our fitness function is designed to favor queries with:

- operators that consume LOB columns
- unique combinations of operators in a query plan
- unique combinations of state transitions per operator

The feedback information is based on the query plan as it is returned by the SUT [11] and on an internal query execution trace which is used for testing and debugging. The trace indicates whether a LOB column was used by an operator, along with its state transitions during the execution of the query. Figure 4 shows a snippet of the trace for a query that has exercised the Sort operator along with some of its state transitions. The fLOB attribute indicates that the Sort operator consumes a LOB column.

Every time a new query is generated and executed our fitness function checks if it includes any LOB columns, and then compares the set of physical operators in the query plan to those corresponding to the queries stored in the best query pool. If the new set of operators is similar to existing ones in the best query pool then the sets of state transitions are compared. If those are significantly different then the query is entered in the best query pool.

The test plan can rely on self-checking mechanisms in the server to detect defects (debug-only code and assert conditions), or additionally the results of each query can be compared to a trusted DBMS implementation. As an alternative (which can also be used on release builds), results correctness can be verified automatically by the system as it generates variations of the same query with different query plans (with the use of query hints).

```
<Iterator PhysicalOp="Sort" LogicalOp="Sort" fLob="1">
    <NewStateChange OldState="Dormant"
    NewState="ScanStart" Method="Open" />
</Iterator>
<Iterator PhysicalOp="Sort" LogicalOp="Sort" fLob="1">
    <NewStateChange OldState="Nil" NewState="Dormant"
    Method="Constructor" />
</Iterator>
<Iterator PhysicalOp="Sort" LogicalOp="Sort" fLob="1">
    <NewStateChange OldState="Scan" NewState="EOS"
    Method="GetRow" />
</Iterator>
<Iterator PhysicalOp="Sort" LogicalOp="Sort" fLob="1">
    <NewStateChange OldState="Scan"
    NewState="ScanRowOut" Method="GetRow" />
</Iterator>
<Iterator PhysicalOp="Sort" LogicalOp="Sort" fLob="1">
    <NewStateChange OldState="ScanRowOut"
    NewState="Scan" Method="GetRow" />
</Iterator>
<Iterator PhysicalOp="Sort" LogicalOp="Sort" fLob="1">
    <NewStateChange OldState="ScanStart"
    NewState="ScanRowOut" Method="GetRow" />
</Iterator>
```

Figure 4. Snippet of feedback information for iterator testing

4.5.3 Testing the correctness of optimization rules

SQL Server's query optimizer is similar to the architecture described in the Cascades Framework [8]. The query optimizer makes use of transformation rules which create the search space of query plan alternatives. For the purposes of this example we assume that there is a need to test code changes in the optimization rules framework. Such changes may include the addition of new exploration or implementation rules, changes in property derivation, etc., and may affect the correctness of the query plan and consequently the correctness of query results. To test such changes we want to include queries with large and diverse query plan space. The size of the plan space is a function of the query's size and complexity but also proportional to the number of exploration rules that created alternatives during optimization. The diversity of search space is proportional to the number of different optimization rules which executed successfully during optimization.

Feedback is collected from an internal system table which provides counters for the optimization rules. A partial example of such information is shown in Table 1. The "Succeeded" column indicates how many times each rule has been used by the optimizer.

This fitness function favors rule diversity over rule frequency. Alternative implementations may favor frequency over diversity, or specific rules, e.g. queries that use the *Join to Hash Join* rule.

Table 1. Query optimization rule feedback

Rule	Succeeded
Join to Nested Loops	3
Left Outer Join to Nested Loops	2
Left Semi-Join to Nested Loops	1
Left Anti-Semi-Join to Nested Loops	0
Join to Hash Join	1
Full Outer Join to Hash Join	0

5. EVALUATION

In this section we compare the experimental results from three different variations of our test generator. Our experiments were aimed at exploring the hypothesis that the use of execution feedback would allow us to target a specific component within the DBMS more effectively than when no feedback is used. Also, we compare the performance of our technique against RAGS.

5.1 Experimental Results

We define three different variations of our technique and we compare their performance and code coverage.

- The first variation of our technique uses execution feedback and a fitness function that aims to maximize the coverage of the query optimization rules. We described this execution feedback and fitness function in section 4.5.3. We will refer to this variation as *QO Rules feedback.*
- The second variation aims to maximize the coverage of physical operators. This variation is identical to the one described in section 4.5.2 with one difference; it doesn't favor specially LOB types. We will refer to this variation as *Iterator feedback.*
- To measure the effect of feedback to the query generation process our third variation doesn't make use of execution feedback. It does, however, discard statements that do not return results. We will refer this variation as *No feedback.*

For all the experiments presented in this section we used the database used by the TPCH benchmark [15] (100MB) and SQL Server 2005. We measure code coverage in *number of unique function combinations* (call from function to function). We have chosen this metric rather than the more widely used function block and arc metrics since it represents better dynamic code paths such as the iterator trees in query execution. In addition, we have performed a parallel experiment with RAGS using the same hardware and software configuration.

Figure 5 shows the total code coverage achieved by the three variations of our test generation techniques and RAGS. All three variations outperform RAGS and achieve significant code coverage very quickly. Their performance is comparable initially but as time progresses both the variations which use feedback provide additional coverage.

The graph shown in Figure 6 illustrates the code coverage achieved specifically over the query optimizer component. As intended, QO rules feedback achieves the highest code coverage much faster than the other two variations. The variation that uses Iterator feedback starts slow but eventually achieves better code coverage than the variation without feedback. The code coverage achieved by RAGS is significantly lower and for this reason we do not include it this graph.

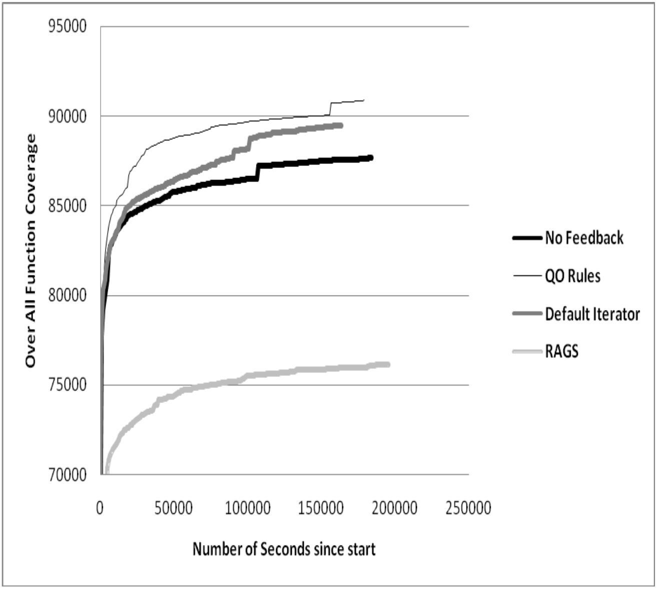

Figure 5. Code coverage over time

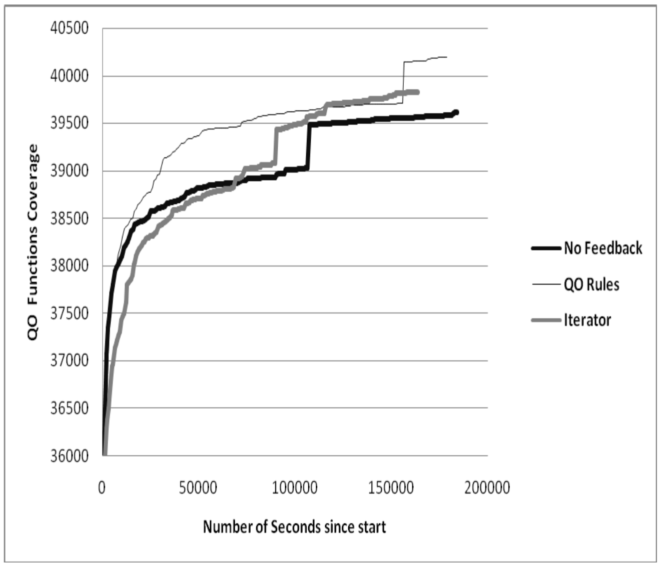

Figure 6. Code coverage of the optimizer component over time

Finally, in Figure 7 we show the number of different optimization rules exercised by the three test generation techniques. The same code coverage pattern is observed; QO Rules exercises the largest number of optimization rules in the shortest period of time.

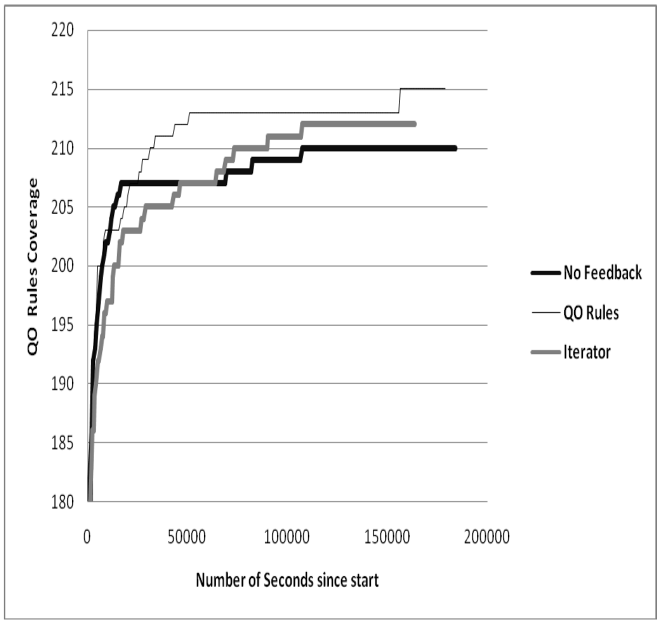

Figure 7. Query optimization rules covered over time

The results of our experiments are summarized in Table 2. The results represent the final measurements after a 48-hour run of each variation. Each column in the table corresponds to the test generation method used, and the results are grouped by the different DBMS components that were exercised. The row titled as "DBMS" includes the total code coverage achieved across SQL Server's code base. "QO" and "QE" correspond to the Query Optimizer and Query Execution components respectively. All numbers express unique function caller/callee invocations with the exception of QO Rules which is measured in number of distinct optimization rules exercised.

Table 2. Coverage by each method per DBMS component

	RAGS	**No Feedback**	**Iterator**	**QO Rules**
DBMS	75975	87600	89603	90820
QO	32108	39581	39869	40193
QE	7066	7431	7538	7655
# QO Rules	173	210	213	215

5.1.1 Results discussion and conclusions

The QO Rules variation aims in exercising the query optimizer code and as intended it achieves the highest code coverage of that component. Since it favors queries which exercise a variety of optimization rules, these queries involve a more diverse set of alternative plans. The larger number of alternatives in the plan space contributes in exercising more code beyond the query optimizer component than the two other variations, resulting in higher overall code coverage.

Although the number of different iterators used by a query is loosely correlated with the number of different optimization rules used, i.e. each different physical operator in a query plan is a result of a corresponding implementation rule, that correlation is not expected to increase the coverage of the query optimizer significantly. In fact, as it is evident in Figure 6 and Figure 7 initially the iterator feedback doesn't significantly help the test coverage of the query optimizer in both functions and rules covered. However, as the queries in the best query pool become very complex, more possibilities for query optimization open up, and the variation with iterator feedback seems to get closer to the variation without feedback and eventually exceeds it (as shown in Figure 6 and Figure 7 after the 50000th second). In terms of overall code coverage iterator feedback is consistently more efficient than when no feedback is used.

At the end of the experiment, the best queries pool for the QO Rules variation included a set of queries that exercised every rule exercised during the duration of the whole experiment; something which is self-evident from the definition of the feedback and fitness function. This is significant because as the best set evolves over time, it encourages more frequent usage of interesting genes from previous queries, which helps to test specific components more intensively. The final set of best queries can be used as a regression test suite for future changes in the component under test. We observed over several repeated experiments that, with similar fitness functions, the set of QO rules covered was also similar. From the definition of the best query set, this means we saw similar convergence towards a set of queries that will exercise the areas defined in the fitness function. Note that this does not imply that the best query sets themselves are always similar, but it does demonstrate that they will converge towards a set which exercises similar feedback genes as defined by the fitness function. We will discuss this further in the remainder of this section.

The difference in code coverage achieved by the three different variations may appear small comparing to the overall number of functions covered. However, what needs to be emphasized is that increases in code coverage tend to become harder as code coverage increases. That fact is not particular to our technique but also noticeable when manual methods are used. For example, an increase in code coverage from 50% to 51% is much easier to achieve than an increase from 90% to 91%. It's hard to compute an approximation of the upper code coverage bound which includes the coverage of dynamic code paths and we do not attempt it in this paper.

When compared to RAGS our test generation technique is performing significantly better in terms of both efficiency and code coverage, with or without the use of feedback.

From our experiments, we noted that early random decision making can affect the future generation of queries by creating some genes that get heavily favored over others. This happens because of the fact that at the start, there are not many gene flavors to work with, and so almost any new gene is given very high importance. In order to overcome this and give the system more time to work through the early genes, we delay the use of feedback until a certain amount of time has elapsed.

In other experiments that were performed using more complex databases schemas, we observed similar patterns to the ones that we showed based on the TPCH database. In general, we try to use test databases which include a variety of SQL Server features in order to maximize the code coverage. These test databases can also be configured in a way that assists the system in generating more interesting genes, such as by creating views that include query statements that are interesting or otherwise difficult to reach.

6. RELATED WORK

There are many existing approaches to random software testing. There is ongoing evidence that random testing provides great value to testing systems that are large and complex [10]. In the area of databases systems Slutz [13] developed the RAGS framework which has motivated our work. The ways that our approach differs has already been described earlier in this paper. Waas and Galindo-Legaria have developed a technique that allows enumeration and random sampling of query plans across a query's plan space [16]. The effectiveness of this technique depends on the set of queries used for plan space exploration. Our technique partners well with plan enumeration and sampling and specifically the variation of our technique based on QO-rules feedback, since works towards creating queries with large and diverse plan spaces.

Beyond the domain of DBMS, the area of random testing is being actively researched. For some random testing techniques the usage of execution feedback has been also explored to create test valid inputs for functions [4] or to facilitate the selection of random test cases without replacement [12]. Other techniques aim toward constraining, sampling the random space of test cases/inputs [2] [12]. Godefroid [5] attempts to compute test inputs to drive a program along a specific path using methods of compositional creation. His method aims to predict how changing inputs can result in changing code paths.

7. FUTURE WORK

The query mutation and synthesis techniques can be extended to include a wider coverage of the SQL language. More complex mutation techniques can be implemented by exchanging query parts between two or more best queries.

The best query pool could potentially be seeded with manually created queries that include specifically selected query genes. That should give a head start to the test generation algorithm and allow the use of query combinations that may not be possible by the current implementation of the generation process.

The database schema and the data stored in the test database include important parameters which can constrain and influence the generation of queries. The use of random schema and data generation methods perhaps combined with execution feedback to alter and extend the schema and datasets can also be explored as an extension our to technique.

8. CONCLUSION

We presented a practical method for generating queries that can be used as test cases for testing a DBMS and we offered examples specific to query processor testing. We have shown that the use of execution feedback improves the efficiency of the test generation process and increases code coverage of specific DBMS components.

9. REFERENCES

[1] Binder V. R. *Testing object-oriented systems: models, patterns, and tools*, Addison-Wesley object oriented series, Addison-Wesley Longman, 2000

[2] Chen T.Y., Kuo F.C. Is Adaptive Random Testing Really Better than Random Testing. In *Proceedings of the First International Workshop on Random Testing (RT '06)* (Jul 20, 2006, Portland, ME, USA) , 64-69

[3] Ciupa I., Leitner A., Oriol M., Meyer B. Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the First International Workshop on Random Testing (RT '06)* (Jul 20, 2006, Portland, ME, USA), 55-63

[4] Ferguson R. and Korel B. The Chaining Approach for Software Test Generation. *ACM Transactions on Software Engineering and Methodology,* 5,1 (Jan 1996) , 63-86

[5] Godefroid P. Compositional dynamic test generation, In Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (Nice, France), ACM Press, New York, NY, 2007, 47-54

[6] Godefroid P., Klarlund N., Sen k. DART: Directed Automated Random Testing, In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (Chicago, IL, USA). ACM Press, New York, NY, 2005, 213 - 223

[7] Graefe G. Query evaluation techniques for large databases, *ACM Computing Surveys (CSUR),* 25,2, (June 1993), 73-169

[8] Graefe G. The Cascades Framework for Query Optimization, *IEEE Data Engineering Bulletin*, 18,3 (1995), 19-29

[9] Graefe G. Volcano - An Extensible and Parallel Query Evaluation System, *IEEE Trans. Knowl. Data Eng.*, 6,1 (1994), 120-135

[10] Hamlet D. When Only Random Testing will do. In *Proceedings of the First International Workshop on Random Testing (RT '06)* (Jul 20, 2006, Portland, ME, USA)

[11] Microsoft Coproration, XML Showplans, SQL Server 2005 Books Online, http://msdn2.microsoft.com/en-us/library/ms189298.aspx

[12] Pacheco C., Lahiri S.K., Ernst M.D., and Ball T. Feedback-directed Random Test Generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, (Minneapolis, MN, USA), 2007

[13] Slutz, D. Massive Stochastic Testing of SQL, In Proceedings of the 24th VLDB Conference, (New York USA 1998), 618-622

[14] Stobie, K. Too Darned Big to Test. ACM Queue, 3, 1, (February 2005)

[15] TPC Benchmark H. Decision Support.http://www.tpc.org.

[16] Waas F. and C. A. Galindo-Legaria, Counting, Enumerating, and Sampling of Execution Plans in a Cost-Based Query Optimizer. In Proceedings of the 2000 ACM SIGMOD international conference on Management of data (Dallas, Texas, USA). ACM Press, New York, NY, 2000, 499 - 509

[17] Whittaker, J. Stohastic Software Testing. Annals of Software Engineering, 4, , J. C. Baltzer AG, Science Publishers Red Bank, NJ, USA, 1997, 115-1