Towards a Sufficient Set of Mutation Operators for Structured Query Language (SQL)

Donald W. McCormick II

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science

William B. Frakes, Chair
Chang-Tien Lu, Member
Gregory Kulczycki, Member

April 28, 2010
Falls Church, VA

Keywords: Database Testing, SQL Testing, Mutation Testing, Test Adequacy Criteria

Towards a Sufficient Set of Mutation Operators for Structured Query Language (SQL)

Donald W. McCormick II

ABSTRACT

Test suites for database applications depend on adequate test data and real-world test faults for success. An automated tool is available that quantifies test data coverage for database queries written in SQL. An automated tool is also available that mimics real-world faults by mutating SQL, however tests have revealed that these simulated faults do not completely represent real-world faults. This paper demonstrates how half of the mutation operators used by the SQL mutation tool in real-world test suites generated significantly lower detection scores than those from research test suites. Three revised mutation operators are introduced that improve detection scores and contribute toward re-defining a sufficient set of mutation operators for SQL. Finally, a procedure is presented that reduces the test burden by automatically comparing SQL mutants with their original queries.

Table of Contents

Table of Figures

List of Tables

1 Introduction

The hypotheses of this paper are: (1) the overall mutation adequacy score (AM) for real-world test suites, defined as the ratio of mutants detected to the total number of non-equivalent mutants (Namin, Andrews et al. 2008), will be significantly lower for all commonly observed mutant categories than the AM from experiments conducted using the NIST (research) vendor compliance test suite; (2) AM for the lowest scoring mutant categories observed from the preliminary experiment will be significantly higher when their mutation operators are revised and compared against the original operators in follow-on experiments using industrial data. The goals of this research are to: (1) produce a revised set of mutation operators for SQL that better approximates the real-world faults made by developers, and (2) progress towards redefining a sufficient set of mutation operators for SQL.

Mutation testing involves systematically generating and introducing faults into an application in order to verify its fault-detection capability (Woodward 1993). The mutations, or faulty versions of the code, are derived from a systematic application of mutation operators. A mutation operator mutates the original code, for example, by modifying integer constants at boundary values, substituting similar arithmetic operators, negating if-then and while decisions, deleting code segments, etc. Application of the mutation operators is designed to produce errors similar to those introduced in real-world applications through human error. Research has demonstrated that the ability for a test suite to detect mutated faults approximates its ability to detect real-world faults (Andrews, Briand et al. 2005).

Most of the mutation testing research to date has been performed on imperative programs (Offutt and Untch 2001). Each line of code is analyzed for every valid application of a mutation operator. Each change generates a distinct program mutation, or mutant. After the mutants are generated, black-box testing techniques such as category partitioning are used to generate test cases designed to identify a mutant's fault-detection capability. Test inputs can be derived solely from the pre-conditions and post-conditions defined in the program specification. In the absence of a program specification formal mathematical descriptions can derive test inputs based on the program's actual behavior (Harder, Mellen et al. 2003).

Research into mutation testing of SQL has lagged mutation testing of imperative programs but is gaining momentum. For example, SQL Mutation has been used to successfully identify applications with SQL injection vulnerabilities (SQLIV) (Shahriar and Zulkernine 2008). Here mutation operators were designed to mimic SQL injection attacks (SQLIAs) that either: (1) inject comment characters, creating a tautology that bypasses logon criteria; or (2) append union clauses that exploit information not filtered in the original query. A set of 4 mutation operators were designed to modify the WHERE condition of logon queries by either: (1) removing it; (2) negating the criteria expression; (3) pre-pending a 'false and' condition; or (4) unbalancing the parentheses.

The criteria for killing a mutant were customized as well so that mutants could only be killed by test cases containing SQLIAs. Mutants were generated from the logon queries identified in a group of open-source web-based applications written in JSP. Test data was derived from a benchmark set of SQLIA queries that did not vary by

application, but were selected randomly for test against each application's mutants. Mutation Analysis is performed here as well to identify a sufficient set of mutation operators, however there was no attempt to quantify the adequacy or redundancy of the mutants. A sufficient set of mutation operators is defined when a test set that kills all mutants under selective mutation has a high mutation score under nonselective mutation. Selective mutation filters out the mutation operators that create the most mutants while nonselective mutation uses the complete set of mutation operators (Offutt, Lee et al. 1996).

A survey of SQL query usage in industry revealed problems inherent to the SQL language suggesting that database applications are good candidates for mutation testing (Lu, Chan et al. 1993). Errors resulting from SQL syntax ambiguities, table design over-normalization, and improper handling of nulls, can prove hard to detect without a disciplined approach to testing.

SQL is a declarative computer language for use with "SQL databases" (ISO 1992). When it comes to querying databases, its strength lies in the interoperability of its component parts or clauses: (1) SELECT, (2) FROM, (3) WHERE, (4) GROUP BY, (5) HAVING and (6) ORDER BY. This also makes it too easy for novices to formulate queries where they have no confidence in the results (Lu, Chan et al. 1993) (Chan, Wei et al. 1993) (Brass and Goldberg 2006). An example is improper use of the Cartesion Join that over-utilizes the CPU, returning all possible combinations between 1 or more tables.

In addition, SQL has not been standardized among database vendors. Companies have introduced proprietary language to suit their individual needs. In turn, other vendors have introduced versions of the SQL that accommodates the various permutations and as a result the SQL standard has become ambiguous. As a result, web developers including a SQL query in their application code, for example, are faced with one or more methods for accomplishing the same task. This can lead to errors, especially when deciding the best way to join tables.

The relational database model encourages normalization among data tables. In some cases this leads to over-normalization, creating subsets of larger tables that could just as easily remained a part of the larger table without negatively affecting retrieval performance. The challenge of manually writing a query linking more than 2 tables leads to errors.

A principle design decision that faces all database architects is whether or not a field accepts a null value. If the field accepts a null value, then data entry is simplified, however the table cannot be indexed on that field which may lead to performance issues. If the field does not accept a null value, then a value must be present at every stage of the record's lifecycle. Null value fields dictate separate handling considerations for the query developer. For example, if a null is not converted to a missing value in the WHERE clause, then a Null combined with a valid result (NULL AND Valid) will always return an unknown value and an error. Similarly, an unhandled null value in the SELECT clause resulting from an improper table join can also return an error. For example, the SQL statement: 'SELECT M.primarykey, D.detailfield FROM M LEFT OUTER JOIN D on M.primarykey = D.foreignkey' generates an error for records in Master table (M) without related records in Detail table (D) since D.detailfield will always evaluate to Null.

The SQLMutation tool automatically generates query mutations designed to emulate errors made by professional database developers (Tuya, Suarez-Cabal et al.

2006). The mutations are designed to assess the adequacy of the query schema and test cases at revealing faults. If a mutant generates a different result than the original query then the mutant is "killed" and the test data is considered sufficient to reveal the inserted fault. If the mutant result is equivalent to the original query then either the query schema prevents a different result or the test data is insufficient to reveal the fault.

For example, SQLMutation emulates the first null handling error illustrated above by first inspecting the query schema to identify fields from the SELECT clause that allow null values. These fields are then modified to return a value outside the field type's domain. The purpose is to introduce an error that will be detected during testing. If the results from the mutated query are different than those from the original query then there are unhandled null values in the test data and the mutation is effective at revealing the error. The corrective action is to handle the nulls either by disallowing them in the table structure, or by converting them to a non-null value in the SELECT clause. If, however, the results from the mutated query are identical to those from the original query then there are no unhandled nulls in the test data and the mutant is ineffective at revealing the error. The test data should be modified until the mutation generates a different result and then the corrective action outlined above can be applied. In this case at least 1 null value should be inserted into the table for the field(s) in question.

SQLMutation emulates the second null handling error illustrated above by systematically replacing all INNER JOIN clauses with OUTER JOIN clauses. The purpose is to introduce an error when fields from the right side of a relation are selected, yet no records from the right side satisfy the relation, as illustrated in the Master-Detail example above. If the results from the mutated query are different than those from the original query then the mutation has revealed the error: either the join type is invalid or there are unhandled null values in the SELECT clause. The corrective action is to revert to the original join type and/or handle the nulls properly. If, however, the results from the mutated query are identical to those from the original query then the mutation did not reveal the potential error since there are no key field values in tuples from the left side of the relation that are absent from the tuples on the right side of the relation. The test data should be embellished until the mutation generates a different result and then the corrective action outlined above can be applied. In this case at least 1 record should be inserted in the left side relation whose key field does not also occur in the right side relation.

SQLMutation automatically generates mutants for: (1) the main SQL clauses, (2) operators in conditions and expressions, (3) handling of NULL values and (4) replacement of identifiers such as column references, constants and parameters. Syntactic and semantic operators are applied in single order to all four categories using every possible permutation to produce a set of legal mutants guaranteed to approximate actual faults generated from improper structuring of the SQL statement. The problems with developing mutants using this approach are: (1) A large number of mutants are generated that produce duplicate results when executed, adding to the test burden and (2) the nature of the test data is not considered when the mutants are generated. With insufficient test data, some mutated queries are hard to "kill". They will always produce the same result as the non-mutated query, and thus remain alive, until the test data is modified.

A 2-level condition coverage tree that quantifies a coverage percentage for each query before it is mutated has been introduced to confront the problem of test data

adequacy in SQL database applications (Suárez-Cabal and Tuya 2009). The upper level of the tree contains 6 coverage, or c-value, placeholders for each JOIN condition in a SQL statement. The lower level of the tree contains 6 c-value placeholders for each condition in the WHERE clause. The 6 c-value placeholders define coverage as: (1) Null-left (Nl) when at least 1 tuple exists from the left-side operand of the join that is null; (2) Null-right (Nr) when at least 1 tuple exists from the right-side operand of the join that is null; (3) Null-both (Nb) when at least a pair of tuples exist, 1 each from the left and right-side operands of the join, that is null; (4) False-left when at least 1 tuple ti exists in the relation X, where x=ti[X], that does not satisfy the condition xRz for any tuple tj in the relation Z, where z=tj[Z]; (5) False-right when at least 1 tuple tj exists in the relation Z, where z=tj[Z], that does not satisfy the condition xRz for any tuple ti in the relation X, where x=ti[X]; and (6) True, when at least 1 pair of tuples exist, ti in the relation X, where x=ti[X], and tj in the relation Z, where z=tj[Z], that does satisfy the condition xRz.

All of the c-values in a condition coverage tree are evaluated against all of the test data generating Yes (Y) when covered, No (N) when not covered, Impossible (I) if it cannot be covered due to a limitation imposed by the database schema, and Unreachable (U) if it cannot be covered due to some other factor, such as the breadth of the test data. The c-coverage percentage is calculated as: (sum(covered c-values)\sum(total c-values)-sum(impossible values)) * 100. The c-coverage percentage is limited to c-values that are not unreachable. The maximum c-coverage percentage can be improved by adding test data to cover previously unreachable c-values. In this manner, the condition coverage tree not only functions to define the adequacy criterion for the test suite but can also be used for test input selection criterion. Tuya automates these coverage rules with SQLRules, a tool that can predict the effectiveness of query mutations by quantifying the percentage of query condition possibilities that are covered by the test data. The higher the coverage percentage the more effective the query mutation.

SQLRules addresses the issue of test data adequacy in SQL database applications but does not address the problem of duplicate mutants that result from using automated mutation tools like SQLMutation. The Java Database Application Mutation Analyzer (JDAMA) automates the application of SQLMutation to JAVA/JDBC applications (Zhou and Frankl 2009). When query strings are dynamically constructed based on varying input values and execution paths, the number of possible queries to test can be significant. Generating mutants for each query and testing all of the mutants increases the test burden several fold. JDAMA counteracts this by creating abstract queries, where some values are replaced by placeholders representing variable values. Instead of mutating all of the possible concrete query strings, JDAMA mutates the abstract query thereby reducing the test burden. The technique minimizes the number of mutants by minimizing the number of queries mutated.

Mutation Analysis attempts to address the problem of excessive mutants by selecting a set of sufficient operators. When these mutation operators are run against a test suite they produce the highest percentage of non-equivalent mutants killed by the test suite. A variant of the forward selection variable reduction algorithm known as Least Angle Regression (LARS) was used in imperative programs to generate the highest AM from the least amount of mutants (Namin, Andrews et al. 2008). The work focused on how well a sufficient set of mutation operators generated detectable faults on average

against all of the test suites. However, test data for the test suites was chosen at random and therefore was not considered as a variable in the experiment.

This work continues the mutation analysis that was begun under Tuya for SQL database applications, proposing a revised sufficient set of mutation operators that improve AM in experiments conducted as part of this research. Section II describes Tuya's research defining the original set of SQL mutation operators, and documents the results of a preliminary experiment aimed at replicating Tuya's research test suite results by using a sampling of real-world "developer" test suites. Preliminary experiment results are used to support the hypothesis that certain poor actors – mutation operators that consistently contribute low scores – are responsible for a test suites poor overall mutation score (AM). Section III details experiments exploring the application of a revised set of mutation operators intended to improve a test suite's overall AM. Section IV provides analysis of the results obtained from the work conducted in sections II and III. Section V summarizes the contributions of this paper and introduces possibilities for future research in this area.

2 Problem Background

Section 2.1 describes the original set of SQL mutation operators. Section 2.2 details results from the original experiment involving those SQL mutation operators and Section 2.3 describes the motivation for the work in this paper.

2.1 Original SQL Mutation Operators

The original set of SQL mutation operators, designed to mutate SQL Data Manipulation Language (DML) SELECT commands, or queries, were first introduced in a paper hereafter referred to as the original study (Tuya, Suarez-Cabal et al. 2007). It presented SQL Clause (SC), Operator Replacement (OR), NULL (NL) and Identifier Replacement (IR) mutation operators that were designed to anticipate the errors resulting from the ambiguity of the SQL language, complexity of table joins, and challenges that handling NULL field values introduce. The SC mutation operators include subtypes like Select (SEL) that replaces each occurrence of either the SELECT or SELECT DISTINCT keywords with the other, and Join (JOI) that replaces each occurrence of various join-type keywords such as INNER JOIN or OUTER JOIN with all other possible join-type keywords. The OR mutation operators include subtypes like Unary Operator Insertion (UOI), that replaces each arithmetic expression or number reference e with –e, e + 1, and e-1, and Arithmetic Operator Replacement (AOR) which replace each arithmetic operator (+, -, *, /,%) with all of the other possible arithmetic operators. The NULL mutation operators include subtypes like Null check predicates (NLF) that replace each occurrence of either the keywords IS NULL or IS NOT NULL with the other. The IR mutation operators include subtypes like column replacement (IRC) that replaces each column reference with each of the other column references, constants and parameters of like type in the query. Each mutation operation generates a mutant or variation of the query that is designed to emulate the faults found in real-world SQL applications.

The majority of the SQL mutation operators are syntactic since they rely exclusively on the existence of SQL keywords when generating mutants. For example, the JOI operator is a syntactic operator that replaces every occurrence of a JOIN type with all other possible JOIN types. Only token replacement is used. Some SQL mutation operators are syntactic and semantic. The Null in the Select List (NLS) operator, for example, first checks the query schema for fields that can store a null value, or are null-eligible. This indicates semantically to the operator that the application should be able to handle null values for that field. If the NLS operator syntactically identifies a null-eligible field in the SELECT list, it mutates it so its results are guaranteed to be different than the original query when a null value exists for that field.

2.2 Original Experiment

Test cases from the SQL Test Suite, available for download from the NIST Conformance Test Suite Software web site (NIST 2008), were used to demonstrate the

mutant generating capabilities of the mutation operators from the study. The SQL Test Suite includes Data Definition Language (DDL) for creating the test database schema, as well as the Data Manipulation Language (DML) for populating and querying the schema. The test suite is divided into modules designed to validate commercial vendors conformance to ISO, ANSI and FIPS SQL standards. Each module focuses on a particular feature of the standard and includes one or more queries and their expected results.

Each of the mutation operators from the study was run against all of the queries from the test suite. The resulting mutants were then executed against the test database. A mutant was considered to be dead if the mutant query result differed from the original query result. If execution of the mutant resulted in a run-time error, the mutant was considered to be dead. If execution of the mutant produced an equivalent result then the mutant remained alive. A 5-step process was followed to try to kill all equivalent mutants and achieve a 100% mutation score for each operator. After each step a mutation score was calculated for each mutation operator as the number of dead mutants it generated divided by the number of non-equivalent mutants it generated.

Step 1 in the process simply involved running the mutation operators against the test suite as is, and served as the basis for comparison in this paper. No attempts were made during step 1 to identify equivalent mutants or to complete the test data in order to improve the mutation score. The average mutation score for all operators at the end of step 1 (69.6%) encouraged the study authors to continue the experiment by documenting the level of effort required to kill the remaining live mutants. Their goal was to kill as near to 100% of the mutants as possible. After experimenting with introducing additional parameter values, modifying the test database to include duplicate rows and negative attribute values, and introducing more null values, they were able to achieve an average mutation score of 85.6%. As a last step they manually create new test cases to kill as many of the remaining live mutants as possible, achieving in the end an average mutation score for all operators of 94.2%. During the last step equivalent mutants were determined to be those mutants that could not be killed because constraints identified in the query schema, or otherwise, prevented entering test data to complete the query coverage.

2.3 Research motivation

The queries from the research test suite used in the original study were designed to verify vendor compliance with 141 different SQL features. As such, the test suites were constructed to validate results and the mutation operators were specifically designed to generate mutants based on the features being tested. Were the results typical considering the nature of the testing, and could they be generalized to the real world population? The original study mutation scores from Table 3 provided some clues. The average mutation score for all operator types from the original study was almost 70% but these scores varied by 26%. A histogram of the scores revealed a distribution skewed to the left with a peak between 80-90%. At first glance it appeared that some operators consistently generated mutants that were killed more often than others regardless of the test suite. Would the converse be true – some operators consistently generated mutants that were killed less often than others regardless of the test suite?

There were enough questions to warrant a scientific evaluation. A preliminary experiment was planned to compare the fault detection capability of mutated queries from real-world test suites against the fault detection capability of mutated queries from the original study. The hypothesis was that the mutation scores from the real-world database test suites would be significantly lower than mutation scores from the original study test suite. The basis for the hypothesis was that both the test data and test queries from the original study would be more comprehensive then those from real-world test suites. The research test suite was designed to exercise all 141 features of the SQL standard and provide guaranteed query results while real-world test suites are generally designed to support a specific application such as a shopping cart, library search, etc.

Moreover, the test data adequacy of the NIST test suite is expected to be better than in real-world test suites. In the NIST test suite queries were constructed first and then the data was backfilled to provide the expected results, whereas in most real-world test suites the data exists before the queries are developed, either because it has been inherited from another system or has never been mined. Based on these assumptions the experimental model for this hypothesis was a test suite's overall mutation score (AM) depends on the set of mutation operators used and the test suite environment (real-world or research) they are executed in (AM = F(Mutation Operator Set, Test Suite Environment)).

3 Methods

Section 3.1 describes the process for generating the mutants for the preliminary experiment. Section 3.2 describes the process for comparing the mutants with their original query. Section 3.3 presents an analysis of results from the preliminary experiment. Section 3.4 presents a revised experimental model and in-depth analysis of domain variables. Section 3.5 describes a follow-on experiment that progresses towards re-defining a sufficient set of mutation operators for SQL.

3.1 Generating the Mutants

The set of mutation operators used in the real-world experiment was the same set used in the original study. Real-world test suites were selected at random from a pool of entry-level test suites so as to compare with the entry-level test suites from the original experiment. The queries, and consequently their query schemas, were also selected at random from the real-world test suites. Otherwise the test data from the real world test suites were left to vary naturally.

Database vendors typically include sample databases with their product releases. These are intended to demonstrate new and existing features of the database. Beginning database developers often leverage the tables, views, stored procedures, functions and constraints from the entry-level samples when designing their first database. Vendors also include intermediate and advanced level schemas for developers to use according to their own experience level with the product. The original study included separate entry level and intermediate level test suites separated by data module number. For this experiment, one each of the entry-level sample schemas available from Microsoft SQL Server (Microsoft 2004), Oracle (Oracle 2008) and MySQL (MySQL 2008) were selected as the real world test suites. Available resources prevented testing more than three sample schemas so the schemas were selected at random in order to minimize bias due to the small sample size.

The SQL, Oracle and MySQL sample schemas were installed on 2005 Express Edition (XE), 10g R1 Personal, and 5.0 versions of the databases, respectively. Each query from all of the sample schemas was run against SQLMutation. SQLMutation can be automated through a web service or run in a browser using the interface shown in Figure 1.

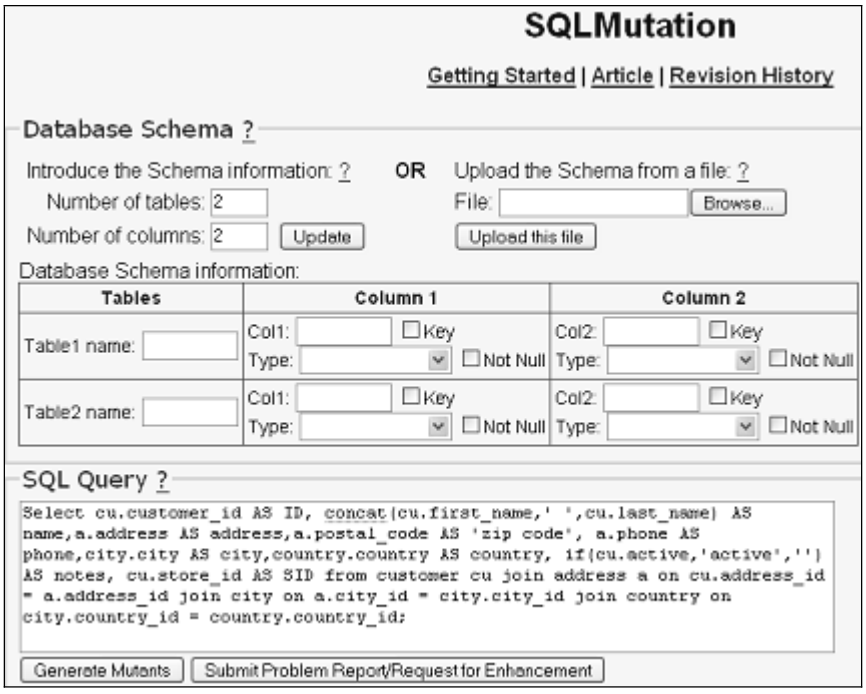

Figure 1 - SQL Mutation Online Interface

SQLMutation requires the query text be entered along with the schemas for all associated tables. Fig. 1 includes a sample query to be mutated. Figure 2 provides an example of the XML-like format for the query schema. It includes the table name and column name, as well as the filed type, whether it can contain a null value and whether they field is a primary key that can participate in table joins.

```
<schema>
  <table name="Customer">
     <column name="customer_id" type="smallint" notnull="true" key="true"/>
     <column name="store_id" type="tinyint" notnull="true"/>
     <column name="first_name" type="varchar" notnull="true"/>
     <column name="last_name" type="varchar" notnull="true"/>
    <column name="email" type="varchar"/>
    <column name="address_id" type="smallint" notnull="true"/>
    <column name="active" type="tinyint" notnull="true"/>
    <column name="create_date" type="datetime" notnull="true"/>
    <column name="last_update" type="timestamp"notnull="true"/>
  </table>
  <table name="address">
     <column name="address_id" type="smallint" notnull="true" key="true"/>
     <column name="address" type="varchar" notnull="true"/>
     <column name="address2" type="varchar"/>
     <column name="district" type="varchar" notnull="true"/>
    <column name="city_id" type="smallint" notnull="true"/>
    <column name="postal_code" type="varchar"/>
    <column name="phone" type="varchar" notnull="true"/>
    <column name="last_update" type="timestamp"notnull="true"/>
  </table>
</schema>
```

Figure 2 - SQLMutation Sample Query Schema

Once the query schema and query text have been entered mutants can be generated. Table 1 lists sample mutants generated by SQLMutation for the real world MySQL Customer List query.

Table 1 - Sample SQLMutation Mutants

Mutation Operator	Type	Subtype	Mutant
SC	SEL	SLCT	SELECT DISTINCT cu.customer_id AS ID , concat (cu.first_name , ' ' , cu.last_name) AS name , a.address AS address , a.postal_code AS 'zip code' , a.phone AS phone , city.city AS city , country.country AS country , if (cu.active , 'active' , '') AS notes , cu.store_id AS SID FROM customer cu INNER JOIN address a ON cu.address_id = a.address_id INNER JOIN city ON a.city_id = city.city_id INNER JOIN country ON city.country_id = country.country_id
SC	JOI	JOIN	SELECT cu.customer_id AS ID , concat (cu.first_name , ' ' , cu.last_name) AS name , a.address AS address , a.postal_code AS 'zip code' , a.phone AS phone , city.city AS city , country.country AS country , if (cu.active , 'active' , '') AS notes , cu.store_id AS SID FROM customer cu LEFT JOIN address a ON cu.address_id = a.address_id INNER JOIN city ON a.city_id = city.city_id INNER JOIN country ON city.country_id = country.country_id
OR	ABS	ABSS	SELECT ABS(cu.customer_id) AS ID , concat (cu.first_name , ' ' , cu.last_name) AS name , a.address AS address , a.postal_code AS 'zip code' , a.phone AS phone , city.city AS city , country.country AS country , if (cu.active , 'active' , '') AS notes , cu.store_id AS SID FROM customer cu INNER JOIN address a ON cu.address_id = a.address_id INNER JOIN city ON a.city_id = city.city_id INNER JOIN country ON city.country_id = country.country_id
OR	UOI	UOIS	SELECT ((cu.customer_id)+1) AS ID , concat (cu.first_name , ' ' , cu.last_name) AS name , a.address AS address , a.postal_code AS 'zip code' , a.phone AS phone , city.city AS city , country.country AS country , if (cu.active , 'active' , '') AS notes , cu.store_id AS SID FROM customer cu INNER JOIN address a ON cu.address_id = a.address_id INNER JOIN city ON a.city_id = city.city_id INNER JOIN country ON city.country_id = country.country_id
NL	NLS	NLSS	SELECT cu.customer_id AS ID , concat (cu.first_name , ' ' , cu.last_name) AS name , a.address AS address , COALESCE(a.postal_code ,'9999') AS 'zip code' , a.phone AS phone , city.city AS city , country.country AS country , if (cu.active , 'active' , '') AS notes , cu.store_id AS SID FROM customer cu INNER JOIN address a ON cu.address_id = a.address_id INNER JOIN city ON a.city_id = city.city_id INNER JOIN country ON city.country_id = country.country_id
IR	IRC	IRCCS	SELECT CU.STORE_ID AS ID , concat (cu.first_name , ' ' , cu.last_name) AS name , a.address AS address , a.postal_code AS 'zip code' , a.phone AS phone , city.city AS city , country.country AS country , if (cu.active , 'active' , '') AS notes , cu.store_id AS SID FROM customer cu INNER JOIN address a ON cu.address_id = a.address_id INNER JOIN city ON a.city_id = city.city_id INNER JOIN country ON city.country_id = country.country_id
IR	IRT	IRTCS	SELECT cu.customer_id AS ID , concat (cu.first_name , COUNTRY.COUNTRY , cu.last_name) AS name , a.address AS address , a.postal_code AS 'zip code' , a.phone AS phone , city.city AS city , country.country AS country , if (cu.active , 'active' , '') AS notes , cu.store_id AS SID FROM customer cu INNER JOIN address a ON cu.address_id = a.address_id INNER JOIN city ON a.city_id = city.city_id INNER JOIN country ON city.country_id = country.country_id

The first SC mutant from Table I tests the effect the DISTINCT clause has on the query results. The second SC mutant tests the effect of replacing an INNER JOIN clause with a LEFT JOIN clause. The first OR mutant tests the effect of replacing a key value with its absolute value (ABS). The second OR mutant tests the effect of replacing the key value with (key value +1) in the SELECT clause. The NL mutant tests the affect of replacing any NULL value with a non-NULL value. The first IR mutant replaces the customer_id field with STORE_ID and the second IR mutant concatenates the Country field with the first_name and last_name to form the name alias. Table 2 lists the total mutants generated by each mutation operator and type for the original study and all three

real world test suites. The mutant ratio, calculated as the number of mutants generated for each type divided by the number of mutants generated for all types, is an indicator of mutant coverage and is also provided for comparison. Real world testing generated 1070 mutants.

Table 2 - Total Mutants Generated from Preliminary Experiment

Mutation Operator	Type	Original Study Mutants	Real World Mutants			Original Study Mutant Ratios	Real World Mutant Ratios		
			MySQL	SQL Server	Oracle		MySQL	SQL Server	Oracle
SC	SEL	241	2	2	1	0.04	0.01	0.01	0
	JOI	84	28	8	0	0.01	0.09	0.02	0
	SUB	379	0	0	0	0.06	0	0	0
	GRU	72	0	0	0	0.01	0	0	0
	AGR	560	0	0	0	0.08	0	0	0
	UNI	23	0	0	0	0	0	0	0
	ORD	39	0	0	0	0.01	0	0	0
OR	ROR	1211	49	17	35	0.18	0.15	0.05	0.09
	LCR	145	0	0	17	0.02	0	0	0.04
	UOI	741	57	69	33	0.11	0.17	0.21	0.08
	ABS	510	40	46	22	0.07	0.12	0.14	0.05
	AOR	253	0	25	0	0.04	0	0.07	0
	BTW	76	0	0	0	0.01	0	0	0
	LKE	33	0	0	0	0	0	0	0
NL	NLF	8	0	0	0	0	0	0	0
	NLS	153	1	7	9	0.02	0	0.02	0.02
	NLI	92	0	0	3	0.01	0	0	0.01
	NLO	276	0	0	9	0.04	0	0	0.02
IR	IRC	989	90	129	260	0.14	0.28	0.38	0.64
	IRT	200	24	25	0	0.03	0.07	0.07	0
	IRH	238	0	0	0	0.03	0	0	0
	IRP	562	0	0	0	0.08	0	0	0
	IRD	0	35	8	19	0	0.11	0.02	0.05
Total Mutants:		6,885	326	336	408				

3.2 Computing the Mutation Scores

A repeatable process was developed for each database environment to run the mutant queries and count the dead mutants. Mutants were “killed” when they generated a result different from the original query, or they produced a run-time error.

The process for executing the MySQL mutants involved running the original query in the MySQL Query Browser, opening a new result pane, running the mutant query and then selecting to compare the results. A row that appears in one result set but not the other appears in green, while the missing row appears in red. If the result sets contain matching rows but different field values, the mismatched fields appear in blue. Figure 3 shows the results when comparing a mutant from the ‘Customer_List’ query from the Sakila sample schema. Both name columns in the split screen of results for IDs 1-11 are highlighted indicating differences between the mutant and original query.

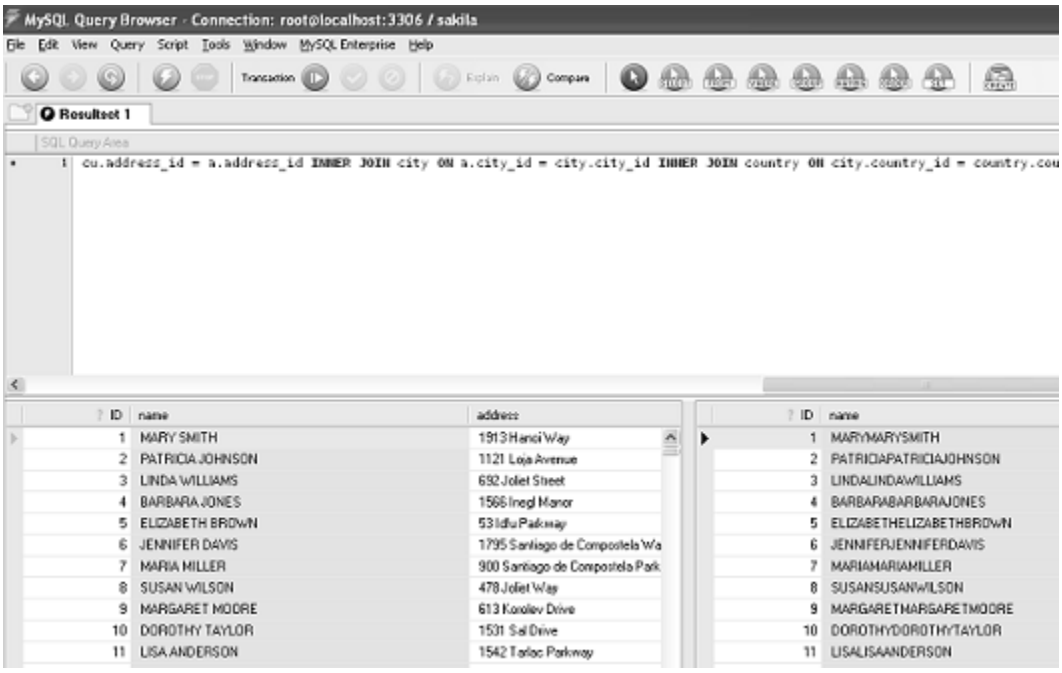

Figure 3 - MySQL Query Compare Interface

The process for executing the SQL Server mutants involved executing a SELECT EXCEPT clause comparing the original query with each mutant and recording the results in a log file. The comparison query was wrapped inside of a stored procedure so that run-time errors could be trapped and recorded as dead mutants. The DDL script for the stored procedure was saved as a template and the template was customized for each query mutant and saved as a batch file that was executed against the database. Figure 4 shows the script used for identifying dead mutants in SQL Server.

```
IF OBJECT_ID ('usp_CompareSQL', 'P') IS NOT NULL
   DROP PROCEDURE usp_CompareSQL;
GO
CREATE PROCEDURE usp_CompareSQL
AS
   DECLARE @CompareCount INT
   Select @CompareCount = Count(Compare.OrderID) From
   (<CompareTo Query> EXCEPT <Compare Query>) as Compare;
   If @CompareCount > 0
      Print N'<MutantID>, Y'
   Else
      Print N'<MutantID>, N'
GO
BEGIN TRY
   EXECUTE usp_CompareSQL;
END TRY
BEGIN CATCH
   Print N'<MutantID>, Y';
END CATCH;
GO
```

Figure 4 - SQL Server Compare Script

The process for executing the Oracle mutants was similar to that for SQL Server. The Oracle PL/SQL equivalent to the SQL Server EXCEPT clause is MINUS. Using the same logic principals as with SQL Server, a DDL script was constructed for each mutant that included the MINUS clause, error handling and recording for dead mutants. These scripts were saved to a batch file and executed against the Oracle database. Figure 5 shows the script for identifying dead mutants in Oracle.

```
CREATE OR REPLACE PROCEDURE  COMPARE_SQL
is
 compare_count integer;
Begin
  Select Count (*) into compare_count From
  (<CompareTo Query> MINUS <Compare Query>);
IF compare_count = 0 then
  DBMS_OUTPUT.PUT_LINE('<MutantID>,N');
Else
  DBMS_OUTPUT.PUT_LINE('<MutantID>,Y');
 end if;
end;
/
exec compare_sql;
/
```

Figure 5 - Oracle Compare Script

Table 3 lists the mutation scores for the preliminary experiment. The mutation score is a ratio of the number of mutants that were killed (dead mutants) divided by the number of mutants that were generated, by mutation operator and type. The absence of a mutation score does not indicate a score of zero (0) rather it indicates that there were no mutants generated for that type and therefore the mutation score cannot be calculated.

Table 3 - Preliminary Experiment Mutation Scores

Mutation Operator	Type	Original Study Mutation Scores	Real World Mutation Scores		
			MySQL	SQL Server	Oracle
SC	SEL	.05	0	0	0
	JOI	.62	.46	0	
	SUB	.85			
	GRU	.89			
	AGR	.73			
	UNI	.87			
	ORD	.82			
OR	ROR	.70	.43	.57	.57
	LCR	.82			.24
	UOI	.69	.56	1.00	1.00
	ABS	.45	.30	.61	.50
	AOR	.91		1.00	
	BTW	.55			
	LKE	.58			
NL	NLF	1.00			
	NLS	.72	0		.56
	NLI	.98			0
	NLO	.88			.67
IR	IRC	.81	.93	1.00	.99
	IRT	.88	1.00	1.00	
	IRH	.83			
	IRP	.67			
	IRD		.49	1.00	1.00
Overall Score:		.70	.46	.69	.55

3.3 Analyzing the Preliminary Experiment Results

The hypothesis of the preliminary experiment was that the mutation scores in a real world environment would be significantly different from the mutation scores in a research environment for every observed mutant type. The results in Table 3 allow us to compare an overall mutation score, or percentage of dead mutants, of 71% for mutants generated in a research environment with 57% for mutants generated in a real world environment. The missing values in the real-world columns from Table 3 demonstrate how the total query space was more adequately represented in the research test suite than

in the real-world test suites. This is to be expected since the research test suite contains over 100 varied queries in the beginner-intermediate modules alone, designed to guarantee software vendor compliance to ISO, ANSI and FIPS standards, whereas the real-world test suites target particular applications . The missing values in the real-world columns are also a side affect of the small real-world sample size.

The burden of proof lie in demonstrating that the overall difference was the sum of individual differences between commonly observed mutant types, and was not due to higher performing research mutants that were not observed in the real world. Figure 6 is a representation of mutation scores by environment and indicates a higher research mutation score in the majority of mutant types observed in both environments.

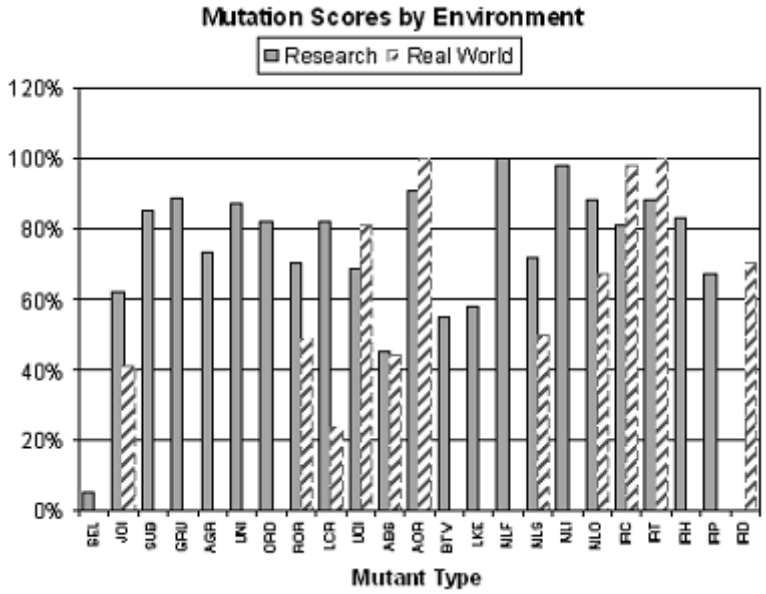

Figure 6 - Mutation Scores By Environment

Figure 7 is a representation of mutant coverage by environment and indicates that 83% of the mutant types that scored higher than their real world counterparts also demonstrated equal or better mutant coverage. This evidence of higher scoring mutant types occurring more often in a research environment than in the real world supported the argument that there was a significant difference between mutation scores from different environments.

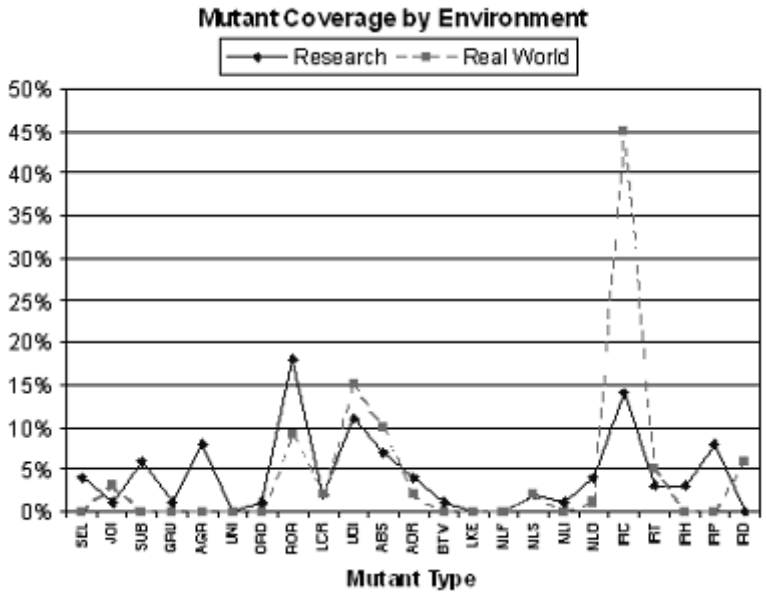

Figure 7 - Mutant Coverage by Environment

In order to validate the hypothesis that there were significant differences in the mutations scores for mutant types observed in the research and real world environments the null hypothesis (H_0) first had to be rejected. H_0 stated there were no significant differences between mutation scores for all mutant types across both environments. Figure 8 shows the frequency distribution for all mutation scores across all mutant types and environments. A mutation score of 0% indicates a mutant type that could not be killed because the mutant results were equivalent to the original query results. This result is important since the only 0% score was observed in the real world environment. By contrast, a mutation score of 100% indicates that for some mutant types all of the mutants generated were killed. This result is also important since 8 of the 9 100% score observations also came from the real world.

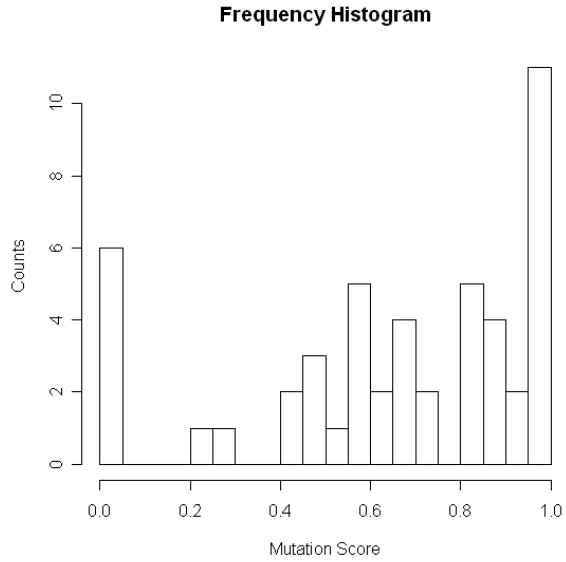

Figure 8 - Frequency Histogram of Mutation Scores

Box plots were selected next to contrast the five-number summaries (smallest observation, lower quartile (Q1), median, upper quartile (Q3), and largest observation) of the mutation scores for the mutation operators, and their subtypes, that were observed in both the research and real-world environments. Figure 9 displays the box plots for the SQL Clause (SC) mutation operators. The data points are mutation scores from Table 3 for the commonly observed SEL and JOI subtypes. Mutation scores are a ratio of the total number of dead mutants divided by the total number of non-equivalent mutants. No attempt was made to identify the equivalent mutants, however, in order to remain consistent with the comparison group from the original study. Therefore all non-dead mutants were deemed to be non-equivalent.

Two research data points were calculated from the 325 mutant results in Table 2. Five real-world data points were calculated from the 41 mutant results in Table 2. Since a mutant can only result in a one (dead) or zero (alive) the score must be calculated over the group, resulting in a reduced sample size. The features of each box plot can be explained in part by the small sample sizes. For example, the large inter-quartile spread for the research plot is based on a 5% mutation score and a 62% mutation score. Similarly, the flat real-world box plot is based on three zero percent (0%) scores out of five total scores.

A comparison of the notched pair reveals that the confidence intervals do not overlap, indicating a significant difference between the groups. The low scores for the research plot's Q_1 minimum boundary and the real-world plot's median represent the SEL subtype mutants. These switch SELECT with SELECT DISTINCT, and vice versa, and often return equivalent mutants that cannot be killed if the original query returns an exclusive set of records. The scores for the research plot's Q_3 maximum boundary and the real-world plot's outlier represent JOI subtype mutants, which replace each join type (INNER, OUTER, LEFT, RIGHT, etc.) with all other possible join types. These subtypes also score poorly if the original query is filtered by a WHERE clause that guarantees an exclusive set of records.

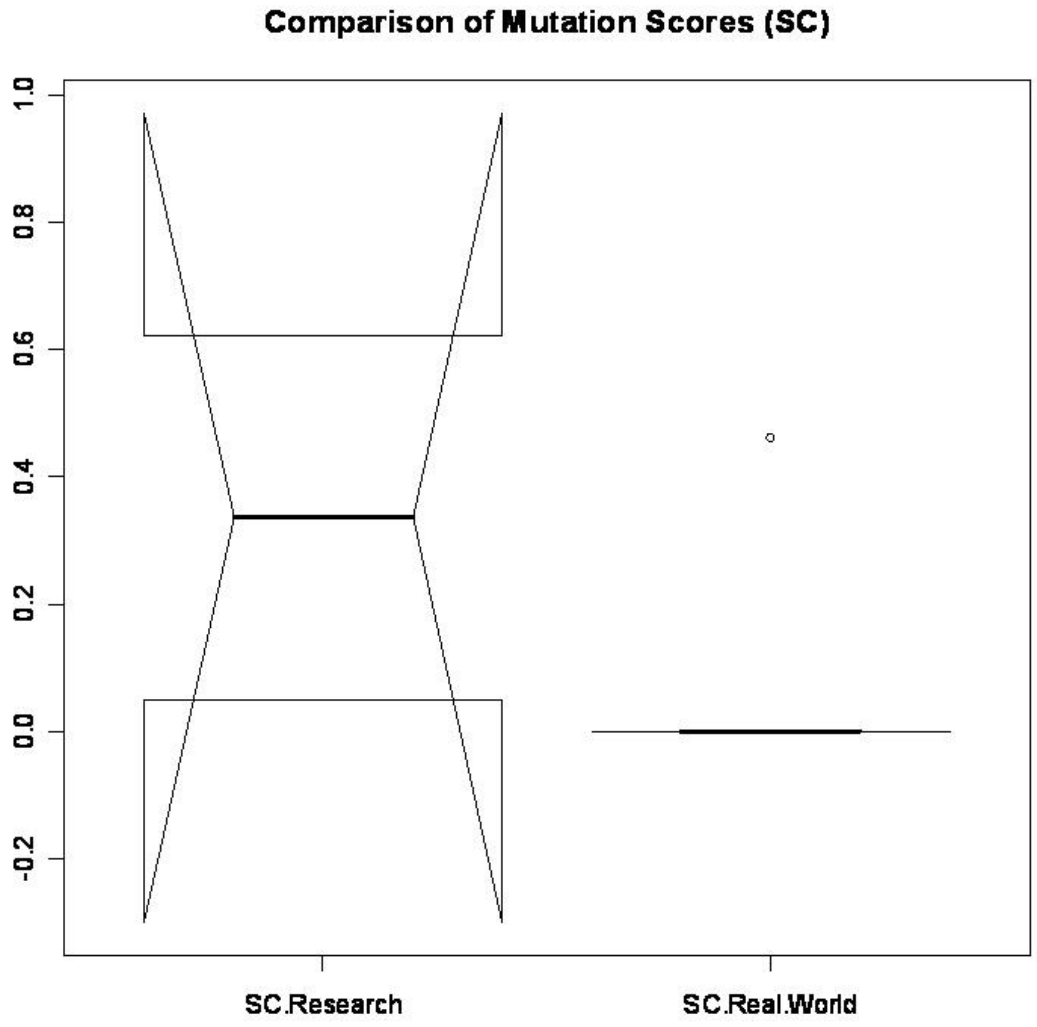

Figure 9 - Box plot of Research vs. Real-World Mutation Scores (SC)

Figure 10 displays the box plots for the Operator Replacement (OR) mutation operators. The data points are mutation scores from Table 3 for the commonly observed ROR, LCR, UOI, ABS and AOR subtypes. The mutation scores were generated from the dead/alive results of the mutants in Table 2 for their respective OR subtypes, in a process described previously for the SC subtypes. The result was five research data points based on 2860 mutant results, and eleven real-world data points based on 400 mutant results. The large inter-quartile spreads for both box plots can be explained in part by the reduced sample size.

A comparison of the notched pair reveals that the confidence intervals overlap, indicating no significant difference between the groups. The scores for both the research plot outlier (45%) and real-world plot Q1 minimum boundaries represent absolute value (ABS) subtype mutants, which replace each numeric type column in the select list or arithmetic expression in the SELECT, JOIN or WHERE clause with ABS(expression)

and –ABS(expression). These subtypes can score poorly if the column type being mutated is a known positive value, or the expression being mutated is a join between key values also known to be positive.

The difference in spread between Q3 and Q1 for the real-world box plot is attributable to high scores for the Unary Operator Insertion (UOI) mutation operator. This operator replaces each arithmetic expression x with the boundary values –x, x+1 and x-1. In this case key fields that were in the SELECT list, were a part of the JOIN expression between two tables, or were a part of the WHERE criteria joining two tables were mutated. Although key fields are primarily numeric, since the numbers themselves do not appear in the SQL, the decision to mutate them in order to mimic real-world faults is questionable. They always produce different results than the original query and so are always killed. At best they are low-value mutants. A low-value mutant is a mutant that does not occur consistently enough in real-world applications to represent a real-world fault. Low-value mutants that are easily killed, or trivial, can artificially inflate the mutation score, while low-value mutants that are hard to kill, or complex, can artificially deflate the mutation score.

In contrast to a low-value mutant, a high-value mutant is a mutant that occurs consistently enough in real-world applications to represent a real-world fault. A high-value mutant that is easily killed, or trivial, occurs when its mutation operator can be applied to most statements (Offutt, Lee et al. 1996). A high-value mutant that is hard to kill, or complex, occurs less often. Mutation operators that generate high-value trivial mutants while mutating the same statements as operators that generate high-value complex mutants are good candidates when defining a sufficient set of mutation operators under selective mutation (Offutt, Lee et al. 1996).

The Identifier Replacement (Column) (IRC) mutation operator is an example of a high-value trivial mutant. It interchanges same-type fields from the query schema that occur in the SELECT list, and JOIN and WHERE expressions. Figures 6 reveals that the average score for real-world IRC mutants (97%) ranked third among all real-world mutant types, and Figure 7 shows that the IRC mutant was applied more often (45%) than any other mutant subtype to the real-world test sets.

The SELECT Clause (SEL) mutation operator is an example of a high-value complex mutant. It interchanges SELECT with SELECT DISTINCT, and vice versa in a SQL statement in an attempt to detect incorrect usage of the DISTINCT quantifier that can result in duplicate rows, invalid aggregate calculations or invalid ordering of the result set (Tuya, Suarez-Cabal et al. 2006). Figures 6 reveals that the average mutation score for SEL mutants for both the research and real-world test sets was lowest, while Figure 7 shows that the SEL mutant was applied less often than any other mutant subtype to the real-world test sets.

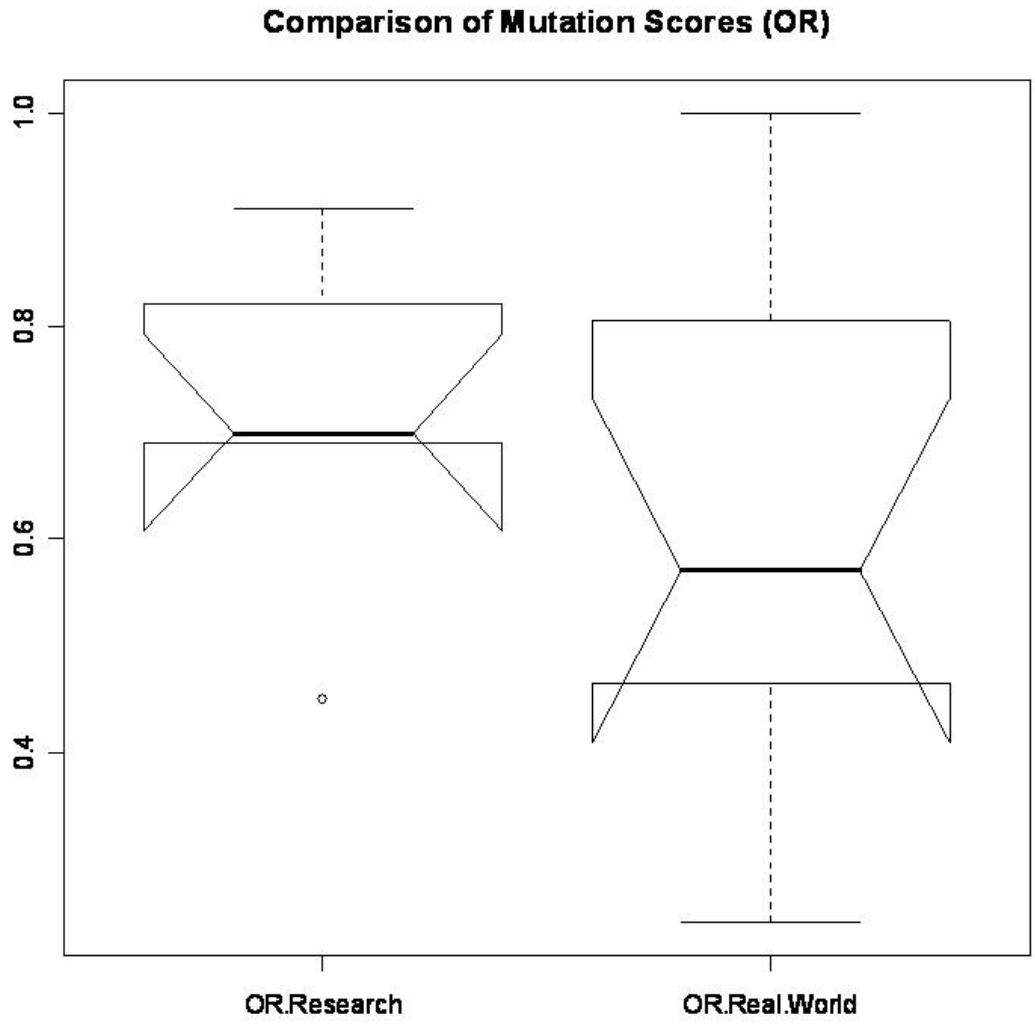

Figure 10 - Box plot of Research vs. Real-world Mutation Scores (OR)

Figure 11 displays the box plots for the Null (NL) mutation operators. The data points are mutation scores from Table 3 for the commonly observed NLS, NLI, and NLI subtypes. The mutation scores were generated from the dead/alive results of the mutants in Table 2 for their respective NL subtypes, in a process described previously for the SC subtypes. The result was three research data points based on 521 mutant results, and four real-world data points based on 29 mutant results. As with the other subtypes, the features of each box plot can be partly explained by the reduced sample size. A comparison of the notched pair reveals that the confidence intervals do not overlap, indicating a significant difference between the groups. The large inter-quartile spread for the real-world box plot is attributable to results from the NLS subtype mutants, which replace each null-eligible column in the select list with a function that returns a value outside the field type's domain. A null value in a mutated column guarantees a dead mutant, however if

the test suite contains no nulls, as was the case for the MySQL test suite, the operator scores poorly.

The difference in median values between the research box plot (86%) and the real-world box plot (23%) is attributable to a difference in mutation scores for the include nulls (NLI) operator. The NLI data point for the research test suite (98%) was considerably higher than the real-world Oracle test suite (0%). The NLI operator ensures different results when nulls exist in the test data that prevent the condition aRb from being evaluated, where a and b are attributes of the condition and R is the relation between a and b. By replacing the condition aRb with (aRb or a IS NULL) it guarantees a true result, and a dead mutant, when nulls exist in the test data for attribute a. None of the real-world mutants were killed because the conditions that were mutated were relations between table keys. Since most table keys cannot be null by definition these can be classified as low-value mutants that artificially deflate the overall mutation score.

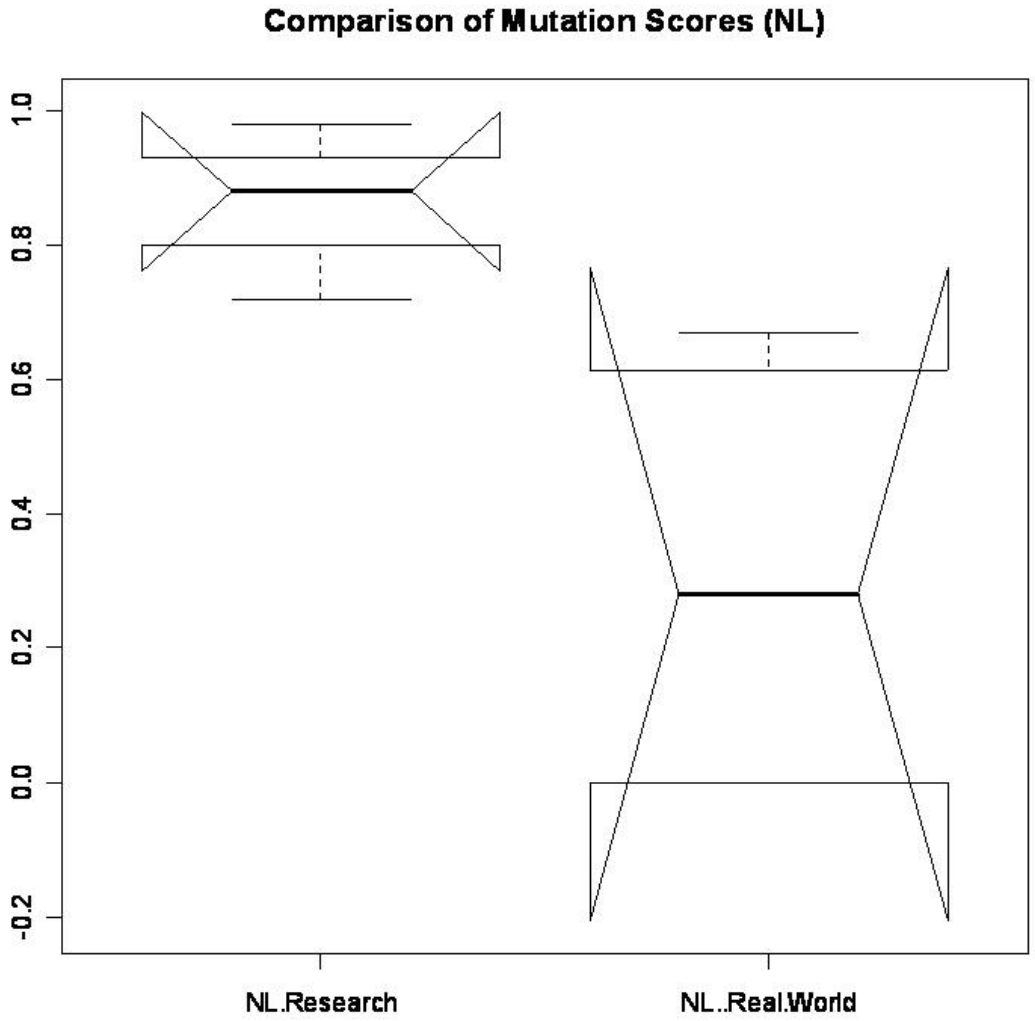

Figure 11 - Box plot of Research vs. Real-world Mutation Scores (NL)

Figure 12 displays the box plots for the Identifier Replacement (IR) mutation operators. The data points are mutation scores from Table 3 for the commonly observed IRC and IRT subtypes. The mutation scores were generated from the dead/alive results of the mutants in Table 2 for their respective IR subtypes, in a process described previously for the SC subtypes. The result was three research data points based on 1189 mutant results, and five real-world data points based on 528 mutant results. As with the other subtypes, the features of each box plot can be partly explained by the reduced sample size.

A comparison of the notched pair reveals that the confidence intervals do not overlap, indicating a significant difference between the groups. The IR mutants replace each column, constant, and parameter present in the query with all other same-type columns, constants, and parameters, respectively, in the query. The medians for both plot scores are above 80% which is predictable considering the mutants are designed to emulate the easiest and most often occurring errors made by developers. The outlier for the real-world box plot is attributable to a lower than average score (93%) for the IRC subtype on the MySQL test suite. The IRC mutants for one query were impossible to kill due to the particular query design. This query returned parent information only using a SELECT DISTINCT clause in combination with a series of LEFT JOINs between parent and detail tables. This design effectively rendered any permutations to the RHS of the joins moot.

Figure 12 - Box plot of Research vs. Real-world Mutation Scores (IR)

Figure 13 displays box plots of the overall mutation scores from all the test suites in the preliminary experiment. The research test suite outlier represents the 5% score for the SEL mutation operator from Table 3. SEL switches SELECT with SELECT DISTINCT, and vice versa, often returning equivalent mutants that cannot be killed if the original query returns an exclusive set of records. The MySQL box plot is significantly lower than the SQL Server and Oracle box plots. This difference can be attributed to individual queries.

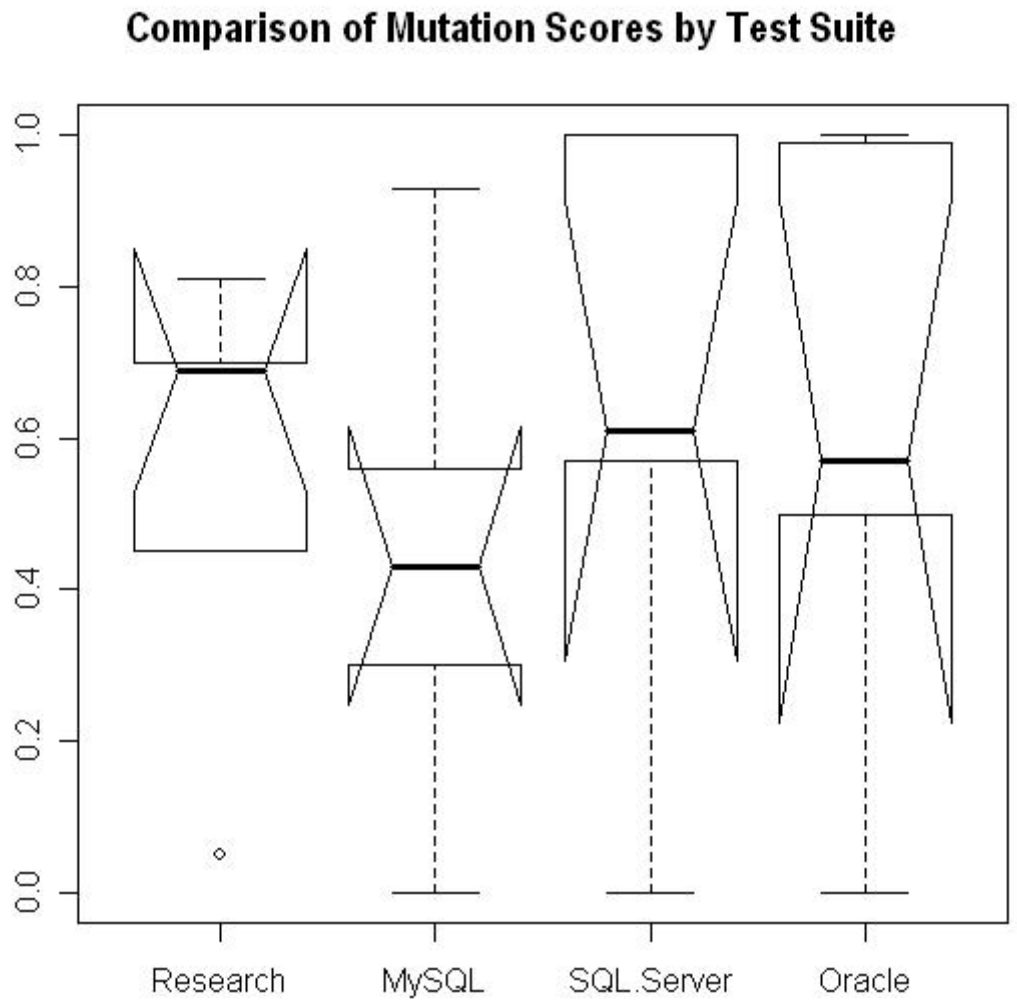

Figure 13 - Comparison of Mutation Scores by Test Suite

The MySQL test suite generated identifier replacement (IRD) and absolute value (ABS) mutations of SELECT DISTINCT queries that could not be killed. In this case the queries only selected fields from the LHS of a LEFT JOIN between tables so each mutation of the RHS of the join expression, by either the IRD or ABS operator, produced results equivalent to the original query. The MySQL test suite also generated null in the select list (NLS) mutations of queries that could not be killed. In this case null-eligible columns were replaced with a function that returns a value outside the domain of the field type. However, since there were no nulls in the test data for those fields the mutations returned equivalent results.

3.4 Experimental Model Revisited

The hypothesis for the preliminary experiment was that test suite mutation scores would be significantly lower for all categories in the real world environment than in the research environment. The results from the preliminary experiment supported the hypothesis for only 2 of the 4 observed operators. Further, a look at the outliers revealed common mutation operators that consistently scored low across both environments. The experimental model supporting the original hypothesis was the test suite mutation score was dependent on the mutation operator type and test suite environment. It was clear from the preliminary experiment results, however that other factors common to both environments were influencing the mutation score.

Ongoing research in this area lent support to the notion that in addition to the mutation operator, the query, query schema, and test data adequacy played roles when detecting SQL faults in database applications (Suárez-Cabal and Tuya 2009). It further contended that the adequacy of the test suite data directly influenced a test suite's mutation score. In order to verify whether these and possibly other variables, as yet undefined, had influenced the preliminary experiment results, an in-depth domain analysis was conducted on each variable.

3.4.1 Query Domain Variable

Table 4 presents the distinct SQL features from the real-world queries, their corresponding research test, and whether they were mutated as a part of the experiment. Seven (7) of the 8 real-world SQL features were also a part of the research suite. Five (5) of the 7 real-world SQL features that were part of the research suite were mutated in both the real-world and research environments. Two (2) of the real-world SQL features were not mutated because there were no mutation operators for those features. There were no SQL features mutated from the real-world test suite that were not also mutated from the research test suite. This suggests that the query types alone were not enough to impact the difference in mutation scores between the real-world and research test suite in the preliminary experiment. However, an analysis of results for the real-world Join Clause (JOI) mutation operator did reveal a relationship between the query type and the adequacy of the test data.

Table 4 - Real-world Query Coverage

Database	Query	SQL Feature	Research Suite	Mutated?(Yes/No)
MySQL	1. Select cu.customer_id AS ID, concat(cu.first_name,' ',cu.last_name) AS name,a.address AS address,a.postal_code AS 'zip code', a.phone AS phone,city.city AS city,country.country AS country, if(cu.active,'active','') AS notes, cu.store_id AS SID from customer cu join address a on cu.address_id = a.address_id join city on a.city_id = city.city_id join country on city.country_id = country.country_id;	1. 4-table join	1. dml020/0082	1. Yes
		2. Field Aliasing	2. dml085/0508	2. No
		3. if replacement function	3. Not observed	3. No
	2. SELECT DISTINCT a.actor_id, a.first_name, a.last_name FROM actor AS a LEFT JOIN film_actor AS b ON a.actor_id = b.actor_id LEFT JOIN film ON b.film_id = film.film_id LEFT JOIN film_category ON film.film_id = film_category.film_id LEFT JOIN category ON film_category.category_id = category.category_id ORDER BY a.actor_id;	1. Select Distinct	1. dml008/0017	1. Yes
		2. LEFT JOIN	2. dml147/0842	2. Yes
Oracle	1. SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.location_id, l.country_id, e.first_name, e.last_name, e.salary, e.commission_pct, d.department_name, j.job_title, l.city, l.state_province, c.country_name, r.region_name FROM employees e, departments d, jobs j, locations l, countries c, regions r WHERE e.department_id = d.department_id AND d.location_id = l.location_id AND l.country_id = c.country_id AND c.region_id = r.region_id AND j.job_id = e.job_id	1. Table alias used in join condition	1. dml160/0859	1. No
SQL Server	1. SELECT p.ProductID, p.ProductName, p.SupplierID, p.CategoryID, p.QuantityPerUnit, p.UnitPrice, p.UnitsInStock, p.UnitsOnOrder, p.ReorderLevel, p.Discontinued, c.CategoryName FROM Categories c INNER JOIN Products p ON c.CategoryID = p.CategoryID WHERE (p.Discontinued = 0);	1. Simple two-table join with filter	1. dml020/0081	1. Yes
	2. SELECT OrderID, [Order Details].ProductID, ProductName, [Order Details].UnitPrice, Quantity, Discount, (([Order Details].UnitPrice * Quantity) * (1 - Discount) / 100) * 100 AS ExtendedPrice FROM Products INNER JOIN [Order Details] ON Products.ProductID = [Order Details].ProductID;	1. Order of expression evaluation	1. dml026/0123	1. Yes

Mutants generated from the real-world test suite using JOI operators scored significantly lower than their research counterparts. Table 3 reveals that 46% of the MySQL JOI mutants were killed, whereas 67% of their research counterparts died. MySQL query 1 from Table 4 is a 4-table join representing a customer address listing. SQLMutation generated one-order mutants by replacing each (INNER) join in the query with LEFT, RIGHT, FULL OUTER and CROSS joins. One-order mutants are generated by applying 1 mutation operator at a time to a program or query. Test data for the customer-address, address-city and city-country relations support no False-left (Fl) conditions where tuples exist in the left-side relation of the join but not in the right (Suárez-Cabal and Tuya 2009). This renders the LEFT join mutations moot. In this case

the success of the JOI mutation operators on MySQL query 1 depended on test data adequate enough to support the Fl condition.

3.4.2 Query Schema Domain Variable

The query schemas were analyzed before generating the mutants for 2 of the 3 significantly different mutant operator categories – Identifier Replacement (IR) and Nulls (NL). Mutants generated from the real-world test suite using IR operators scored significantly higher than their research counterparts. Table IV reveals that 100% of the SQL Server IR Column replacement (IRC) mutants and 93% of the MySQL IRC mutants were killed, whereas only 81% of the research mutants died. A closer look at the 7% of the MySQL IRC mutants that were not killed reveals a relationship between the query and query schema that affects a query's mutation score. MySQL query 2 from Table 4 generated all of the real-world IRC mutants that were not killed in the preliminary experiment. This query returns a distinct list of actors that played in every film, by category in the test suite. The combination of the SELECT DISTINCT clause and the LEFT JOIN from the Actor table to the Film table guarantees that no legal mutation of the columns in the join conditions will generate different results.

SQLMutation analyzes the query schema and replaces column identifiers in the join condition with other like-type columns from the same table to generate an IRC Join (IRCJ) mutant. In this case the IRCJ mutants that were generated did not produce results that differed from the original query, rather the query type overrode them. If the SELECT DISINCT clause was replaced by a SELECT clause and the LEFT JOIN were replaced with an INNER JOIN than the IRCJ mutants would have generated different results and been killed. In this case the likelihood of success for IRC mutants is increased when the SELECT Clause (SC) and Join Clause (JOI) operators are applied simultaneously. This underscores a potential weakness of SQLMutation. All of the mutants generated are one-order mutants where each mutant is generated by applying 1 mutation operator at a time and none are applied in combination.

Mutants generated from the real-world test suite using NL operators scored significantly lower than their research counterparts. Table IV reveals that 56% of the SQL Server Null in SELECT List (NLS) mutants were killed, whereas 72% of their research counterparts died. Oracle query 1 from Table 4 generated the majority of real-world NLS mutants that were not killed in the preliminary experiment. This query returns employee information, including employee name, salary, job title, department and address.

SQLMutation analyzes the query schema to identify all the columns in select list that allow nulls. The NLS mutation operator then replaces null-eligible columns in the select list with a function that returns a value outside the domain of acceptable values. If there is a null value in the test suite for that column, a different value will be returned and the mutant will be killed. All of the columns from the select list in Oracle query 1 allow nulls so individual mutants were generated for each select list column. However, 4 of the select list columns contained no null values in the test suites, therefore their mutants generated equivalent results and were not killed. This confirms that for the NLS mutation operator success cannot be guaranteed by the query schema alone but is dependent on the relationship between the query schema and adequacy of the test suite data.

3.4.3 Test Suite Data Domain Variable

A direct relationship between the adequacy of test data and a query's mutation score has been established in related research (Suárez-Cabal and Tuya 2009). An analysis of the query and query schema variables revealed ancillary relationships with the adequacy of the test data that also influence a query's mutation score. An analysis of real-world mutation scores for the majority of operators (73%) demonstrated a direct relationship between their success and the adequacy of the test data. Half of these mutation operators depended solely on the adequacy of the test data for their success. The other half revealed relationships with 2 additional domain variables from the preliminary experiment (query, query schema).

3.4.4 Mutation Operator Domain Variable

An analysis of the Relational Operator Replacement (ROR), Unary Operator Insertion (UOI), Absolute Value Insertion (ABS), Identifier Replacement Column (IRC) and IRD mutation operators revealed an additional relationship, referential integrity constraints between tables, that was not documented in the original study.

The ROR operators replace equal sign (=) join operators with <>, <, <=, > and >= operators. The UOI, ABS, IRC and IRD operators replaces numeric left and right-side operands of a JOIN condition with boundary values (number + 1, -number, number – 1), absolute values (ABS(number), -ABS(number)), and other columns and constants from the query that are the same type, respectively. The only mutants that were not killed during the experiment were ABS(number) replacements, where the value for key fields never changes, and coincidental results where the replacing column happened to be the same value as the replaced column. The fact that the mutants were killed depended more on established conventions of database design such as requiring key values to be positive integers, and setting constraints between the primary and foreign keys of joined tables. It appears that SQLMutation does not consult such referential integrity constraints when it generates mutants that replace join condition operands. As a result a large number of low-value trivial mutants are generated that are easily killed and do not represent real faults that developers would make.

Table 5 is a matrix showing the relationships between the mutation operators applied to the real world test suite and the experiment variables that contributed to the success of their mutants. An analysis of the relationships helps identify the criteria that were used to select and apply the mutations. All of the mutation operators are dependent on the existence of a particular SQL clause in the query itself. Since they only generate one-order mutants, however, and do not consider a combination of SQL clauses, this can lead to low-value trivial mutants – mutants that are killed easily but are not representative of real world faults.

Table 5 - Mutation Operator Relationship Matrix

Mutation Operator	Query	Query Schema	Test Data	Referential Integrity	Arithmetic Operations
SEL	X		X		
JOI	X		X		
ROR	X		X	X	
UOI	X			X	
ABS	X			X	
AOR	X				X
NLS	X	X	X		
NLO	X		X		
IRC	X	X	X	X	
IRT	X	X	X		
IRD	X	X	X	X	

Four (4) of the 11 mutation operators consult the query schema before generating their mutants. Six (6) of the remaining 7 mutation operators, however, depend on the adequacy of test data or referential integrity between tables for success, yet do not consider these variables before generating their mutants. This can lead to a large proportion of equivalent mutants – mutants that are not easily killed. For example, only 46% of mutants that did not consider other dependent variables were killed whereas 76% of mutants consulting the query schema were killed.

3.5 Follow-on Experiment

The analysis of relationships between the query, query schema, test suite data and mutation operators domain variables demonstrated that each one affects a test suite's mutation score. It also identified relationships between domain variables that were not previously documented. Another revelation was how all of the domain variables rely heavily on the adequacy of the test data for success yet query coverage is not a criteria when generating mutants using SQLMutation. Related work has produced a tool for automatically calculating query coverage that confirms there is a direct relationship between query coverage and mutation score but the tools have not been integrated (Suárez-Cabal and Tuya 2009). Also, only one-order mutants are generated when mutation combinations may prove more successful at generating high-value mutants. Finally, several of the mutation operators consult the query schema but do not consult referential integrity constraints between tables when generating their mutants.

The revised experimental model includes all of the aforementioned domain variables, plus database constraints (AM = F(Mutation Operator, Query, Query schema, Test Suite Data, Database Constraints). However, the only domain variable that can effectively be controlled across both research and real-world environments is the mutation operator. The query, query schema and test data depend on the test suite under analysis. SQLMutation has been demonstrated in this study to generate some low-value trivial mutants. These are easily killed but since they do not consult referential integrity constraint information they are not representative of faults that all developers are going to

make all the time. If SQLMutation generates high scoring mutants that are not representative of faults made by developers (low-value) there is a possibility that the high scoring mutants that are representative of developer faults (high-value) do not cover the entire mutation space. The remainder of this work will introduce new high-value mutation operators that outperform the low-value mutation operators that were confirmed by the preliminary experiment results.

A subset of mutation operators from Table 3, with observed results from both the research and real-world environments, that scored below 50% were selected as candidates for the follow-on experiment. By far the lowest scoring operators were both from the SC class: the SEL type (1.25%), which replaces instances of SELECT with SELECT DISTINCT and vice versa, and the ORD type (.25%) which alters the ORDER BY clause by substituting Ascending with Descending, and vice versa, and switches the order of column names in order by clauses that reference multiple columns. The other mutation types that were selected were the NLS type (43%), which replaces each null-eligible column reference in the select list with a function that returns a value outside the domain of possible values when a null value is encountered, and the ABS type (46%) , which precedes each arithmetic expression with the unary operators ABS and –ABS.

Actual queries form a real-world research library application, along with the accompanying real-world database test suite were obtained for the follow-on experiment. Table 6 provides a sample of the actual queries that were mutated, by type, during the experiment. It includes the original query, a mutation made by the original version of the mutation operator, and a mutation made by the enhanced version of the mutation operator. All of the mutation scores from the original mutation operators were then compared against the mutation scores for the enhanced mutation operators. The results are presented in Table 7.

Table 6 - Mutation Operator Before/After Syntax Comparison

No.	Type(s)	Original Query	Original Mutation	Revised Mutation
1	SEL/ORD	SELECT Project_ID AS RecID, PN AS ProjectNumber, PI AS PrincipalInvestigator FROM [tbl Projects] WHERE [tbl Projects].[Current] IN('yes','new') ORDER BY PI DESC	SELECT Project_ID AS RecID, PN AS ProjectNumber, PI AS PrincipalInvestigator FROM [tbl Projects] WHERE [tbl Projects].[Current] IN ('yes' , 'new') ORDER BY PI ***ASC***	SELECT Project_ID AS RecID, PN AS ProjectNumber, PI AS PrincipalInvestigator FROM [tbl Projects] WHERE [tbl Projects].[Current] IN ('yes' , 'new') ORDER BY ***RecID***
2	NLS	SELECT [tbl Projects].TI, [tbl Projects].PN, [tbl Projects].Start_Date, [tbl Projects].LN, 'AB Descriptor' = CASE WHEN AB LIKE '% provide %' THEN 'Provides' WHEN AB LIKE '% test %' THEN 'Tests' ELSE 'Normal' END, [lookup PD Officers].Officer_Name, [tbl Projects].PI FROM [tbl Projects] INNER JOIN [lookup PD Officers] ON [tbl Projects].Officer_ID = [lookup PD Officers].Officer_ID	SELECT [tbl Projects].TI, [tbl Projects].PN , [tbl Projects].Start_Date , [tbl Projects].LN , 'AB Descriptor' = CASE WHEN AB LIKE '% provide %' THEN 'Provides' WHEN AB LIKE '% test %' THEN 'Tests' ELSE 'Normal' END, [lookup PD Officers].Officer_Name , ***COALESCE([tbl Projects].PI ,'9999') AS PI*** FROM [tbl Projects] INNER JOIN [lookup PD Officers] ON [tbl Projects].Officer_ID = [lookup PD Officers].Officer_ID	SELECT [tbl Projects].TI, [tbl Projects].PN , [tbl Projects].Start_Date , [tbl Projects].LN , 'AB Descriptor' = CASE WHEN AB LIKE '% provide %' THEN 'Provides' WHEN AB LIKE '% test %' THEN 'Tests' ***WHEN COALESCE([tbl Projects].AB,'9999') = '9999' THEN '9999'*** ELSE 'Normal' END, [lookup PD Officers].Officer_Name , [tbl Projects].PI FROM [tbl Projects] INNER JOIN [lookup PD Officers] ON [tbl Projects].Officer_ID = [lookup PD Officers].Officer_ID
3	ABS	SELECT LnkSubToTopic.Subscriber_ID, tblRehabConnection.ID, MIN(LnkSubToTopic.SearchTopicID) AS SearchTopicID FROM LnkSubToTopic INNER JOIN tblRehabConnection ON LnkSubToTopic.SearchTopicID = tblRehabConnection.SearchTopicID GROUP BY LnkSubToTopic.Subscriber_ID, tblRehabConnection.ID	SELECT ***ABS(LnkSubToTopic.Subscriber_ID) AS Subscriber_ID,*** tblRehabConnection.ID, MIN(LnkSubToTopic.SearchTopicID) AS SearchTopicID FROM LnkSubToTopic INNER JOIN tblRehabConnection ON LnkSubToTopic.SearchTopicID = tblRehabConnection.SearchTopicID GROUP BY LnkSubToTopic.Subscriber_ID, tblRehabConnection.ID	N/A (Remove Mutation)

Table 7 - Mutation Operator Before/After Score Comparison

Trial No.	Type	Description	No. Queries	Results	No. Mutants	No. Dead	No. Equivalent	Mutation Score
1	SEL/ORD	Mutate Table Alias' in Order By Clause	3	Before	8	6	2	100%
				After	22	22	0	100%
2	NLS	Mutate Null Value in Concatenated String	3	Before	4	2	0	50%
				After	9	7	0	78%
3	ABS	Remove mutations of table keys that are identity columns from select and join clauses	3	Before	34	17	14	85%
				After	6	3	0	50%
Overall Before					**46**	**25**	**16**	**83%**
Overall After					**37**	**32**	**0**	**86%**

4 Results

A review of the box plots from section 3 (Figures 9-12) reveals that mutation scores for three of the four mutation operator categories common to both the research and real-world test suites (SC, NL, IR) were significantly different. Only two of the three significantly different operator categories (SC, NL) scored lower in the real-world test suites than their research counterparts. Therefore the first hypothesis of this paper, that all mutation operators will score lower on real-world test suites than the research test suite, cannot be proven outright. The second hypothesis of this paper was that the AM from the lowest scoring mutant categories in the preliminary experiment would be significantly higher in follow-on experiments using industrial data, when their mutation operators were revised and compared against the original operators.

Sample query 1 from Table 6 shows a query with several column aliases, identified by the AS syntax. The original mutation operator for the SEL and ORD types does not mutate column aliases that also appear in the order by clause. The sample from Table 6 replaces the Descending clause with an Ascending clause. The revised mutation operator, however, performs the same functionality as the original operator but also replaces column aliases not already present in the original query order by clause with column aliases from the select list until all possible combinations are exhausted. The result are syntactically correct, legal mutants that mirror faults developers make when they fail to verify that the column alias they select to sort by represents the actual column they wanted to sort by. The before and after results from Table 7 show that the Mutation Score (AM) for the SEL/ORD revision remained the same but the number of equivalent mutants was reduced.

Sample query 2 from Table 6 shows a query with a CASE WHEN decision statement. The original mutation operator for the NLS type analyzes the query schema for columns that can be null then mutates those columns in the select list to return a value outside the field type's domain when a null value is encountered. The sample from Table 6 mutates the PI field. The original mutation operator does not however mutate a null-eligible column in the select list if it is contained in a decision statement like CASE WHEN. The revised mutation operator performs the same functionality as the original mutation operator, but also mutates null-eligible columns inside decision statements like CASE WHEN. The results are syntactically correct, legal mutants that mirror faults developers would make if they failed to handle null values properly. In the case of the revised mutation sample from Table 6 if the AB field contains a NULL value it will be killed, alerting the developer that the original query returns unhandled nulls. The before and after results from Table 6 show that the Mutation Score (AM) for the NLS revision improved by 28%.

Sample query 3 from Table 6 shows a query with a CASE WHEN decision statement. The original mutation operator for the ABS type wraps positive and negative absolute value functions (ABS, -ABS) around number type columns in the select list, or arithmetic expressions in the WHERE or JOIN clauses. The sample from Table 6 mutates the Subscriber_ID integer field, which is the Primary Key for the LnkSubToTopic table. Subscriber_ID is the Primary Key and has been defined as an auto-increment (identity) type with a default seed value of 1. As such its value can never be non-positive and this

mutant will always produce a result equivalent to the original query. Restrictions for applying ABS mutation operators to arithmetic expressions have been previously published (King and Offutt 1991), including no. 6 that ABS and NEGABS (-ABS) are not applied to an expression that is known to be non-negative or non-positive. The revised NLS mutation operator from Table 6 identifies this restriction from the query schema and does not perform the mutation. As a result of this change the number of mutants for the NLS mutation operator were reduced by 82% while the number of equivalent mutants shrank from 14 to zero (0).

The overall results from the follow-on experiment show an improvement in the mutation score (AM) from 83% to 86%. Due to resource constraints limiting the sample size it is not possible to make a statistical comparison between the overall mutation score before revision and after. The second hypothesis of this paper therefore cannot be proven. The results are encouraging however for continuing to explore the hypothesis in future research. The mutation operators that were targeted in this paper produced mutation scores below 50% in the preliminary experiment but only represented 12% of the total mutants generated, on average (Table 2). Other higher scoring mutation operators such as IRC, with an average AM of 93% (Table 3) and an average coverage percentage of 36% (Table 2), are candidates for revision in future work where improvements on a similar scale could translate into something more significant.

Both before and after mutation scores from the follow-on experiment were higher than the overall mutation scores from either the research test suite or real-world test suites in the preliminary experiment. This is true despite the fact that the mutation operators represented only 12% of the mutants generated in the preliminary study. Selective mutation dictates that when the scores of a test suite ran against a subset of operators approaches the scores of a test suite ran against all operators the nonselective mutants (and their operators) can be dropped, resulting in a sufficient set of mutation operators (Offutt, Lee et al. 1996). This "do fewer" approach can help reduce the cost of mutation testing as long as it does not result in "intolerable information loss" (Offutt and Untch 2001). The original study identified the Arithmetic Operator Replacement (AOR), Logical Connector Operator (LCR), Other Nulls (NLO) and Union (UNI) mutation operators as a sufficient set for SQL but did not reduce further due to concern over loss of effectiveness (Tuya, Suarez-Cabal et al. 2007).

The results from Table 7 support a modified approach to selective mutation. In addition to excluding low-value mutation operators, the selected operators were analyzed to determine if their behavior can be modified to produce: (1) mutants that more accurately reflect real-world faults, and (2) mutants that generate a fewer number of low-value equivalent mutants. Sample queries 1 and 2 from Table 6 achieved this by adding rules to the existing rule set dictating the operator's behavior, whereas sample query 3 achieved this by removing some rules. As a result the revised set increased the mutation score, reduced the total number of mutants by 19% and improved the effectiveness of the test suite by generating mutants that more closely represent real-world faults.

5 Conclusions

Section 5.1 describes the contributions this paper has made towards the goals outlined in the Introduction. Section 5.2 describes the future work that should be done to fully realize those goals.

5.1 Contributions

Mutation testing involves inserting faults in an application by systematically modifying the code to simulate real world faults introduced through human error. Each code modification is a mutant. Mutants are ran and when they produce a different result than the original they are "killed". All of the mutants along with the test data required to run their programs constitute a test suite. A mutation score (AM) for a test suite is calculated as the number of dead mutants divided by the number of non-equivalent (cannot be killed) mutants.

The higher the mutation score the more successful the test suite will be at detecting faults when run against a candidate application. The majority of research to date has focused on mutation testing with imperative programs. Recent work has focused on developing a sufficient set of mutation operators for SQL (Tuya, Suarez-Cabal et al. 2007). These operators have been tested against the NIST research test suite used by database vendors to guarantee software compliance with SQL standards. When ran against the beginner to intermediate The original sufficient set of SQL mutation operators produced a mutation score of 70% against beginner to intermediate level queries.

The first hypothesis of this paper is that mutation scores from a real world environment would score significantly lower than those observed in the research environment of the original NIST study. The basis for this hypothesis was since the research test data was created after the NIST queries were designed in order to guarantee results, research mutants would score higher than their real-world counterparts, where queries are developed after the data is available.

A preliminary experiment was run using a random sample of beginner-level schemas that are included as part of vendor's database products. One each MySQL, SQL Server and Oracle sample schema was selected for the experiment. The queries from these sample schemas were run against the mutation operators from the original study in order to produce a set of real world mutants. These mutants were then executed in their native database environments and mutation scores were generated based on the results. A frequency histogram indicated a non-normal distribution of 0% and 100% mutation scores.

The SC, and NL mutation operator scores were demonstrated to be significantly lower in the real world environment than in the research environment. The IR mutation operator scores were demonstrated to be significantly higher in the real world environment than in the research environment. The only mutant category that was not significantly different between environments was OR, therefore the null hypothesis (H_0) could not be rejected for every mutant category. The first research hypothesis (H_A) can be accepted for 3 out of 4 commonly observed mutation operator categories.

The second hypothesis of this paper is the AM for the lowest scoring mutant categories observed from the preliminary experiment will be significantly higher when their mutation operators are revised and compared against the original operators in follow-on experiments using industrial data. The mutation operators that scored below 50% in the preliminary experiment were selected as candidates for a follow-on experiment where the operators were modified to improve their AM while reducing the number of mutants. The goals were to: (1) produce a revised set of mutation operators for SQL that better approximates the real-world faults made by developers, and (2) progress towards redefining a sufficient set of mutation operators for SQL. An industry test suite from a research library application was used in the experiment.

For imperative programs, the number of mutants generated is proportional to the number of data references multiplied by the number of data objects (Offutt, Lee et al. 1996). For SQL the formula is the number of eligible clauses multiplied by the number of mutation operators that mutate each clause, multiplied by the number of eligible permutations per operator. Table 8 shows how 1 JOIN clause from a sample query generates 25 mutants. Mutation testing is computationally expensive due to the large number of mutants that have to be run and evaluated. One of the contributions of this paper is a SQL Server stored procedure that was used during the follow-on experiment to automate the task of running and comparing queries (Appendix I).

Query	(A) Eligible (JOIN) Clauses	(B) Mutation Operator	(C) Eligible permutations	(D) Operator Mutants (A x C)
Select A.ID, B.Description From tblAlpha AS A INNER JOIN tblBravo AS B on A.ID = B.ID	1	ABS	4	4
		IRC	4	4
		JOI	4	4
		UOI	6	6
		ROR	7	7
			Total mutants generated:	25

Table 8 - Total Mutants Generated From 1 JOIN Clause

The overall mutation score for the revised set of mutation operators tested in the follow-on experiment did improve, but a lack of resources kept the query sample size small and prevented a statistical comparison. Therefore the second hypothesis of this experiment could not be evaluated. However, the goals of the follow-on experiment were achieved: (1) a subset of mutation operators that had previously generated consistently low scores across test suite environments was improved to better reflect real-world faults, and (2) an improved AM for the revised set of operators, as well as a reduction in the overall number of mutants signified progress towards redefining a sufficient set of mutation operators for SQL.

5.2 Future Work

In order to fully realize the goal of identifying a revised set of sufficient mutation operators for SQL the scope of the effort should be expanded to include all current operators, not just the poor actors identified in this paper. The focus should remain the same however – revising existing mutation operators to better reflect real-world faults while reducing the number of low-value equivalent mutants. The major benefit is improved test data adequacy. Research has demonstrated that when artificially inserted, or seeded, faults are detected by a test suite then that same test suite is able to detect non-seeded faults as well (Andrews, Briand et al. 2005). Improved test data adequacy can only contribute to improved application reliability.

An added benefit of using sufficient mutation operators is that less test data is required to isolate faults resulting in faster test execution. This reduces the test burden. For example, QAShrink integrates SQLMutation with a coverage criteria tool to analyze an existing database and automatically insert the minimum amount of data necessary to preserve test data adequacy in a reduced database (Tuya, Suarez-Cabal et al. 2009). JDAMA reduces the test burden for JAVA/JDBC applications by integrating SQLMutation with a mutant (score) checker (Zhou and Frankl 2009). These tools are encouraging signs that a comprehensive mutation testing tool can be developed. Such a tool would completely automate the process by 1) selecting the queries for test, 2) determining test data adequacy, 3) generating the minimum number of mutants based on a sufficient set of mutation operators, 4) scoring the mutants, and 5) modifying the test data to a achieve an AM as close to 100% as possible.

A mutation operator that accurately predicts real-world faults can still generate duplicate mutants if another mutation operator identifies the same fault. The current set of mutation operators do not adequately cross-reference each other to determine in advance whether an eligible mutant is necessary. Table 8 illustrates how SQLMutation generates 25 separate mutants for 1 join expression. Some of these create duplicate faults. For example, both the UOI and ABS operators negate the LHS and RHS of join expressions to create mutants. Normally both sides of a join expression are represented by key fields and often key fields are designed to auto-increment from a seed value of 1. This prevents the possibility of a negative value. Under these circumstances both the UOI and ABS mutants are killed.

SQLMutation requires users to submit a query schema when generating mutants but such field properties as auto-increment and seed value are not captured in the schema. If SQLMutation were revised to include this information then better decisions could be made by the mutation operators, especially concerning mutating key fields. This type of information is considered in related work that references the conceptual data model before generating mutants, but the process has not been automated (Chan, Cheung et al. 2005).

SQLMutation currently only produces one-order mutants, where each mutant is generated by applying 1 mutation operator at a time and none are applied in combination. This allows for a cleaner interpretation of results, since the impact of each mutant can be inspected individually without concern for side affects from other operators. While this approach makes more sense in imperative programs, where minor changes can be far

reaching and must be reversible before testing continues, it makes less sense in SQL applications. Here the majority of queries being tested are SELECT statements that do not permanently alter the state of the test suite.

The benefit of multiple-order mutations is that mutation operators can share information that could result in fewer equivalent mutants. Section 3.4.2 provides an illustration of this where mutations from the SEL and JOI mutation operators cancel each other out. In this case if the JOI mutation operator was aware that the SELECT query was eligible for a SELECT DISTINCT mutation then it would not have applied a LEFT JOIN mutation to the INNER JOIN and one less equivalent mutant would have been generated.

Bibliography

Andrews, J. H., L. C. Briand, et al. (2005). Is mutation an appropriate tool for testing experiments? Proceedings of the 27th international conference on Software engineering. St. Louis, MO, USA, ACM.

Brass, S. and C. Goldberg (2006). "Semantic errors in SQL queries: A quite complete list." J. Syst. Softw. **79**(5): 630-644.

Chan, H. C., K. K. Wei, et al. (1993). "User-Database Interface: The Effect of Abstraction Levels on Query Performance." MIS Quarterly **17**(4): 441-464.

Chan, W. K., S. C. Cheung, et al. (2005). Fault-Based Testing of Database Application Programs with Conceptual Data Model. Proceedings of the Fifth International Conference on Quality Software, IEEE Computer Society.

Harder, M., J. Mellen, et al. (2003). Improving test suites via operational abstraction. Proceedings of the 25th International Conference on Software Engineering. Portland, Oregon, IEEE Computer Society.

ISO (1992). Information Technology - Database Language - SQL. ISO/IEC 9075:1992. Maynard, MA, Digital Equipment Corporation. **3rd Edition**.

King, K. N. and A. J. Offutt (1991). "A Fortran language system for mutation-based software testing." Softw. Pract. Exper. **21**(7): 685-718.

Lu, H., H. C. Chan, et al. (1993). "A survey on usage of SQL." SIGMOD Rec. **22**(4): 60-65.

Microsoft. (2004). "Northwind and Pubs sample databases for SQL Server 2000." Retrieved April 2008, from http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46a0-8da2-eebc53a68034&DisplayLang=en.

MySQL. (2008). "Sakila database download file." Retrieved April 2008, from http://downloads.mysql.com/docs/sakila-db.zip.

Namin, A., J. Andrews, et al. (2008). Sufficient mutation operators for measuring test effectiveness. Proceedings of the 30th international conference on Software engineering, Leipzipg, Germany, ACM New York, NY, USA.

NIST. (2008). "Conformance Test Suite Software." Retrieved April 2008, from http://www.itl.nist.gov/div897/ctg/software.htm.

Offutt, A. J., A. Lee, et al. (1996). "An experimental determination of sufficient mutant operators." ACM Trans. Softw. Eng. Methodol. **5**(2): 99-118.

Offutt, A. J. and R. H. Untch (2001). Mutation 2000: uniting the orthogonal. Mutation testing for the new century, Kluwer Academic Publishers**:** 34-44.

Oracle. (2008). "Installing the sample schemas and establishing a connection." Retrieved April 2008, from http://www.oracle.com/technology/obe/obe1013jdev/common/OBEConnection.htm.

Shahriar, H. and M. Zulkernine (2008). MUSIC: Mutation-based SQL Injection Vulnerability Checking. Proceedings of the 2008 The Eighth International Conference on Quality Software - Volume 00, IEEE Computer Society.

Suárez-Cabal, M. and J. Tuya (2009). "Structural Coverage Criteria for Testing SQL Queries." Journal of Universal Computer Science **15**(3): 584-619.

Tuya, J., M. J. Suarez-Cabal, et al. (2006). SQLMutation: A tool to generate mutants of SQL database queries. Proceedings of the Second Workshop on Mutation Analysis, IEEE Computer Society.

Tuya, J., M. J. Suarez-Cabal, et al. (2007). "Mutating database queries." Inf. Softw. Technol. **49**(4): 398-417.

Tuya, J., M. J. Suarez-Cabal, et al. (2009). Query-aware shrinking test databases. Proceedings of the Second International Workshop on Testing Database Systems. Providence, Rhode Island, ACM.

Woodward, M. R. (1993). "Mutation testing--its origin and evolution." Information and Software Technology **35**(3): 163-169.

Zhou, C. and P. Frankl (2009). Mutation Testing for Java Database Applications. Proceedings of the 2009 International Conference on Software Testing Verification and Validation, IEEE Computer Society.

Appendix A - Mutant Scoring Stored Procedure

```
USE [NARIC]
GO
/****** Object:  StoredProcedure [dbo].[usp_CompareSQL_SortOrder_mod11]
Script Date: 04/22/2010 14:42:22 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Author: Don McCormick
-- Create date: 11/17/2009
-- Description:   Compares Sort Order of Matching Queries
-- =============================================
ALTER PROCEDURE [dbo].[usp_CompareSQL_SortOrder_mod11] (@QueryID int)
AS
      DECLARE @strSourceSQL nvarchar(4000)
      DECLARE @strTargetSQL nvarchar(4000)
      DECLARE @strCompareSQL nvarchar(4000)
      DECLARE @intMutantID int
      DECLARE cursorMutants CURSOR FOR SELECT SQL, MutantID from
thesis_mutants WHERE QueryID = @QueryID
      DECLARE @strExec nvarchar(4000)
      --
      DECLARE @strDeclare nvarchar (4000)
      DECLARE @strCursor nvarchar (4000)
      DECLARE @strSQLSource nvarchar(4000)
      DECLARE @strSQLTarget nvarchar(4000)
BEGIN
      -- SET NOCOUNT ON added to prevent extra result sets from
      -- interfering with SELECT statements.
      SET NOCOUNT ON;
      -- Initialize source query variable
      Select @strSourceSQL = QuerySQL FROM thesis_queries WHERE QueryID
= @QueryID;

      Open cursorMutants
      -- Retrieve First Record
      FETCH NEXT FROM cursorMutants INTO @strTargetSQL, @intMutantID

      WHILE @@FETCH_STATUS = 0
      BEGIN
            Select @strSQLSource = (Select Replace(@strSourceSQL,' FROM
', ' INTO #tempSource FROM ')) + ' ';
            Select @strSQLTarget = (Select Replace(@strTargetSQL,' FROM
', ' INTO #tempTarget FROM ')) + ' ';
            Set @strDeclare = 'DECLARE @SourceSum int ';
            Set @strDeclare = @strDeclare + 'DECLARE @TargetSum int ';
            Set @strDeclare = @strDeclare + 'DECLARE cursorSource
CURSOR FOR SELECT BINARY_CHECKSUM(*) FROM #tempSource ';
            Set @strDeclare = @strDeclare + 'DECLARE cursorTarget
CURSOR FOR SELECT BINARY_CHECKSUM(*) FROM #tempTarget ';
            Set @strDeclare = @strDeclare + 'DECLARE @CompareCount int
';
```

```
            Set @strCursor = 'CREATE TABLE #tempSourceBC (id_num int
IDENTITY(1,1), checksum INT);'
            Set @strCursor = @strCursor + 'Open cursorSource ';
            Set @strCursor = @strCursor + 'FETCH NEXT FROM cursorSource
INTO @SourceSum ';
            Set @strCursor = @strCursor + 'WHILE @@FETCH_STATUS = 0 ';
            Set @strCursor = @strCursor + 'BEGIN ';
            Set @strCursor = @strCursor + 'INSERT INTO
#tempSourceBC(checksum) Values(@SourceSum); ';
            Set @strCursor = @strCursor + 'FETCH NEXT FROM cursorSource
INTO @SourceSum; ';
            Set @strCursor = @strCursor + 'END ';
            Set @strCursor = @strCursor + 'CLOSE cursorSource; ';
            Set @strCursor = @strCursor + 'DEALLOCATE cursorSource; ';
            Set @strCursor = @strCursor + 'CREATE TABLE #tempTargetBC
(id_num int IDENTITY(1,1), checksum INT);'
            Set @strCursor = @strCursor + 'Open cursorTarget ';
            Set @strCursor = @strCursor + 'FETCH NEXT FROM cursorTarget
INTO @TargetSum ';
            Set @strCursor = @strCursor + 'WHILE @@FETCH_STATUS = 0 ';
            Set @strCursor = @strCursor + 'BEGIN ';
            Set @strCursor = @strCursor + 'INSERT INTO
#tempTargetBC(checksum) Values(@TargetSum); ';
            Set @strCursor = @strCursor + 'FETCH NEXT FROM cursorTarget
INTO @TargetSum; ';
            Set @strCursor = @strCursor + 'END ';
            Set @strCursor = @strCursor + 'CLOSE cursorTarget; ';
            Set @strCursor = @strCursor + 'DEALLOCATE cursorTarget; ';
            Set @strCursor = @strCursor + 'Select @CompareCount =
Count(*) From (Select a.id_num FROM #tempSourceBC a INNER JOIN
#tempTargetBC b ON a.id_num = b.id_num WHERE a.checksum <> b.checksum)
AS Compare; ';
            Set @strCursor = @strCursor + 'IF @CompareCount > 0 INSERT
INTO thesis_mutant_results11(MutantID, Result) VALUES(' +
Cast(@intMutantID AS nvarchar(10)) + ', N''Dead Mutant'') ELSE INSERT
INTO thesis_mutant_results11(MutantID, Result) VALUES(' +
Cast(@intMutantID AS nvarchar(10)) + ', N''Equivalent Mutant''); ';
            Set @strExec = @strSQLSource + @strSQLTarget + @strDeclare
+ @strCursor;
            BEGIN TRY
                  -- Check for run-time error
                  EXECUTE (@strSQLSource + '; ' + @strSQLTarget + '; '
+ @strDeclare + @strCursor);
            END TRY
            BEGIN CATCH
                  SELECT
                  @strExec AS strExec,
                  ERROR_NUMBER() AS ErrorNumber,
                  ERROR_SEVERITY() AS ErrorSeverity,
                  ERROR_STATE() AS ErrorState,
                  ERROR_PROCEDURE() AS ErrorProcedure,
                  ERROR_LINE() AS ErrorLine,
                  ERROR_MESSAGE() AS ErrorMessage;
                  INSERT INTO thesis_mutant_results11(MutantID, Result)
VALUES(@intMutantID, N'Run-time Error');
            END CATCH;
            -- Retrieve next record
```

```
            FETCH NEXT FROM cursorMutants INTO @strTargetSQL,
@intMutantID;
      END
END
Return 0
```

Appendix B - Glossary

AM: mutation adequacy score. The ratio of mutants detected to the total number of non-equivalent mutants.

Complex Mutant: a mutant generated from a mutation operator that can be applied to proportionately less test statements or queries than most mutation operators.

Dead Mutant: a mutant that generates different results than the original query or program being tested.

Equivalent Mutant: a mutant that generates a result that is equivalent to the original query. Mutants are classified as equivalent if they cannot be killed due to a query schema constraint or other constraint that prevents the test data from isolating the fault.

High-Value Mutant: a mutant that occurs consistently enough in real-world applications to represent a real-world fault.

Low-Value Mutant: a mutant that does not occur consistently enough in real-world applications to represent a real-world fault.

Mutation Analysis: identifying a sufficient set of mutation operators that can predict AM, where AM is the mutation adequacy score when using all mutation operators.

Mutation Testing: systematically generating and introducing faults into an application in order to verify its fault-detection capability.

Non-Selective Mutation: mutation testing using mutation operators without distinction for the number of mutants they generate.

One-order Mutant: a mutant generated by applying 1 mutation operator at a time to a program or query. One-order mutants are not applied in combination with other mutants.

Selective Mutation: mutation testing without the mutation operators that create the most mutants.

Sufficient Set: a subset of mutation operators, and a linear model that predicts *AM* accurately from the Am_i measures, while generating only a small number of mutants such that $AM \approx k + c_1Ams_1 + c_1Ams_1 + ... + c_jAms_j$ where:

- *AM* is the mutation score for the full set of non-equivalent mutants
- *k* is an intercept
- *c* is a set of coefficients $\{c_1, c_2.... c_j\}$
- *s* is the subset $\{s_1, s_1... s_j\}$
- Am is the mutation score for a particular operator (Namin, Andrews et al. 2008).

Test Data Adequacy: the ability of a test suite to cover the queries being tested.

Trivial Mutant: a mutant generated from a mutation operator that can be applied to proportionately more test statements or queries than most mutation operators.