Efficient Mutation Analysis of Relational Database Structure Using Mutant Schemata and Parallelisation

Chris J. Wright
Department of Computer Science
University of Sheffield

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College

Phil McMinn
Department of Computer Science
University of Sheffield

***Abstract*—Mutation analysis is an effective way to assess the quality of input values and test oracles. Yet, since this technique requires the generation and execution of many mutants, it often incurs a substantial computational cost. In the context of program mutation, the use of mutant schemata and parallelisation can reduce the costs of mutation analysis. This paper is the first to apply these approaches to the mutation analysis of a relational database schema, arguably one of the most important artefacts in a database application. Using a representative set of case studies that vary in both their purpose and structure, this paper empirically compares an unoptimised method to four database structure mutation techniques that intelligently employ both mutant schemata and parallelisation. The results of the experimental study highlight the performance trade-offs that depend on the type of database management system (DBMS), underscoring the fact that every DBMS does not support all types of efficient mutation analysis. However, the experiments also identify a method that yields a one to ten times reduction in the cost of mutation analysis for relational schemas hosted by both the Postgres and SQLite DBMSs.**

I. Introduction

Having recently found use in application areas ranging from politics and government [1] to the simulation of astrophysical phenomenon [2], the relational database is a key component of real-world software. The schema of a relational database specifies the types of data that will be used by an application, how the data will be organized into tables, which data values are valid, and what relationships may exist between them. Since the relational schema defines what data will be both accepted into and rejected by the database management system (DBMS), it is an important part of a database application that must be tested.

One method for testing a relational schema involves automatically generating a series of SQL `INSERT` statements and data values that are designed to highlight potential flaws in the database's structure [3]. Of course, it is important to assess the effectiveness of the generated data. While it may be useful to accomplish this task by measuring the coverage of, for instance, the source code of the database application [4] or constraints in the relational schema [3], this paper instead presents a method that employs mutation analysis.

Using mutation operators that change the constraints of a relational schema, a database structure mutation analysis technique gives an indication of the thoroughness with which the `INSERT` statements and values exercise the schema. Yet, mutation analysis often incurs a substantial computational cost because it requires the generation and execution of many mutants [5]. Since the use of mutant schemata and parallelisation can reduce the cost of mutation analysis in the context of programs [6], [7], this paper applies these methods to the mutation analysis of a relational schema.

In contrast to Kapfhammer et al.'s approach to the mutation analysis of database structure [3], this paper presents a "Full Schemata" method that creates a large database schema containing all of the mutant tables and a "Minimal Schemata" technique that improves upon "Full Schemata" by reducing both the number of database manipulation statements executed during mutation analysis and the size of the schema holding all of the mutants. Since "Full Schemata" requires the execution of many database interactions that can run at the same time, this paper describes two additional approaches – "Up-Front Schemata" and "Just-in-Time Schemata" – that use parallelisation to further reduce the cost of mutation analysis with "Full Schemata". While the "Up-Front Schemata" method evaluates each independent iteration of mutant evaluation in parallel, the "Just-in-Time Schemata" technique adds the mutants to the database both in parallel and at the moment before they are needed.

Employing six case studies and two widely-used DBMSs, Postgres and SQLite, this paper's empirical study compares and contrasts the performance of the original method from [3] and the four new methods that leverage both mutant schemata and parallelisation. The experimental results reveal a one to ten times reduction in the cost of mutation analysis, with "Minimal Schemata" being the best for databases hosted by SQLite and both "Just-in-Time Schemata" and "Minimal Schemata" being good choices for Postgres.

In summary, the contributions of this paper are as follows:

1) The application of the mutant schemata technique for relational database schema testing, including a schemata representation, a schemata generation algorithm, and an approach for execution (Section III);
2) An empirical study that uses real-world case studies to evaluate the improvement in time-efficiency associated with four new methods that employ both mutant schemata and parallelisation (Section IV); and
3) The description of how differences between DBMSs influence mutation analysis and the demonstration of a method that yields significant cost reductions despite these variations in functionality (Sections III and IV).

```
CREATE TABLE USER_INFO (
  USER_ID VARCHAR(50) NOT NULL PRIMARY KEY,
  ...
);
CREATE TABLE USAGE_HISTORY (
  USER_ID VARCHAR(50) NOT NULL,
  SESSION_ID INTEGER,
  LINE_NO INTEGER,
  COMMAND_SEQ INTEGER,
  COMMAND VARCHAR(50),
  PRIMARY KEY (USER_ID)
  FOREIGN KEY (USER_ID)
    REFERENCES USER_INFO (USER_ID)
);
CREATE TABLE UNIX_COMMAND (...);
CREATE TABLE TRANSCRIPT (...);
CREATE TABLE RACE_INFO (...);
CREATE TABLE OFFICE_INFO (...);
CREATE TABLE DEPT_INFO (...);
CREATE TABLE COURSE_INFO (...);
```

(a) Snippet of a relational database schema with a mutant, corresponding to the addition of a primary key (highlighted)

```
INSERT INTO USER_INFO
  VALUES ('laura', ...)
INSERT INTO USAGE_HISTORY
  VALUES ('laura', 1, 10, 1, 'awk')
INSERT INTO USAGE_HISTORY
  VALUES ('laura', 2, 10, 2, 'grep')
```

(b) A sequence of INSERT statements that "kill" the mutant

Figure 1. An example relational database schema (part a) and an accompanying test suite (part b)

II. Background

A. Mutation Analysis of Database Structure

A *relational database schema* defines the types of data that will be stored in a database and how that data is organized into tables. Through the specification of so-called *integrity constraints*, it defines both what relationships exist between different columns of data and, in addition to the basic type information (e.g., INTEGER), the sorts of values that are permissible for each column. Figure 1 shows a snippet of the database schema used in an open-source example called *UnixUsage* (available at *http://sourceforge.net/projects/se549unixusage*), used to record UNIX commands run by students together with personal student information. The database schema includes a PRIMARY KEY integrity constraint, ensuring all rows of the USER_INFO table are uniquely identifiable; a NOT NULL constraint, preventing the omission of values for the USER_ID field; and a FOREIGN KEY constraint, ensuring all values in the USER_ID column of the USAGE_HISTORY table refer to some existing value in a row of the USER_INFO table. Although not present in this example, CHECK constraints can also be used to place additional limitations on the values accepted in a column, (e.g. LINE_NO > 0). Any SQL INSERT statement involving data that do not conform to these integrity constraints will be "rejected" (i.e., the DBMS will not add that data to the tables of the database). For example, the relational database schema in Figure 1(a) will disallow an attempt to insert a row into USAGE_HISTORY when the value for USER_ID is NULL.

```
K ← ∅
for each mutant do
    Create tables in database for mutant
    for each sqlInsertStatement in testSuite do
        originalResult ← Pre-computed result of insert with non-mutant
        mutantResult ← executeWithDBMS(sqlInsertStatement)
        if originalResult ≠ mutantResult then
            K ← K ∪ {mutant}
        end if
    end for
    Remove tables in database for mutant
end for
```

Figure 2. Kapfhammer et al.'s mutation analysis algorithm [3], referred to as the "Original" approach in this paper

Defining a suitable database schema is one of the first steps in developing a database application. As such, any mistakes in the relational schema can ripple to subsequent stages of the application's development, thus potentially increasing the costs of fixing the problem and decreasing overall quality. Responding to this issue, Kapfhammer et al. [3] introduced a mutation analysis technique for assessing the quality of test suites that exercise a database's integrity constraints. The method involves mutating the constraints of a database's schema – for instance, by removing columns from a primary key or adding a NOT NULL constraint to a column. (The reader is referred to reference [3] for complete details about this approach to database structure mutation.)

The evaluated test suites are a series of SQL INSERT statements. If an INSERT is accepted for the original unmutated database structure (i.e., the data adheres to the database schema's integrity constraints and was successfully inserted into the database) but rejected for the mutant – or vice versa – the mutant is killed because the test suite revealed a difference in behaviour. As an example, Figure 1(b) lists a sequence of INSERT statements that can kill the mutant shown in part (a) of the same figure (highlighted to show the addition of a PRIMARY KEY column to the USAGE_HISTORY table). Even though each INSERT is accepted with the original database schema, the mutant database schema rejects the third one because the additional PRIMARY KEY constraint prevents duplication of USER_ID values.

Kapfhammer et al.'s approach to the mutation analysis of database structure can be summarised by these steps:

1) Create the tables of a mutant structure in a database;
2) Execute the INSERT statements of the test suite; and
3) Compare the result of submitting each INSERT statement to the mutant schema with the output from the execution of the same statement with the original database schema. (Typically, the result of an INSERT is a boolean value returned by the database indicating whether or not the INSERT was successful.)

The above steps are repeated for each mutant, with a final mutation score computed as $|K|/$*Number of Mutants*, where K is the set of killed mutants. The complete algorithm appears more formally in Figure 2.

Although useful in assessing test suites, database schema mutation – like most forms of mutation – is a method that often incurs substantial computational costs. For traditional mutation analysis involving programs, two approaches to improving the time efficiency have included the creation of "meta-mutants" through *mutant schemata* and simultaneous mutant evaluation through *parallelisation*.

B. Mutant Schemata

Proposed by Untch et al. [8], the *mutant schemata* approach aims to reduce the amount of time taken for mutation analysis by combining an original program and all of its mutants into a single "meta-mutant". The creation of this meta-mutant is realised at the source code level by inserting conditional branches for each mutant, enabling the execution of code specific to each individual mutant as for their standalone versions, but without the need to create a multitude of individual programs containing mostly identical code. Furthermore, the approach avoids the need to compile multiple files and repeatedly recompile the unchanged segments of source code for each mutant.

Note that to avoid any potential confusion between a *relational database* schema and a *mutant* schemata in the remainder of this paper, we will fully qualify the type of schema being referenced, unless grammatical constraints prevent us from doing so. Also, we will prefer "database structure" to "database schema" to further reduce confusion.

C. Parallelisation

If a problem can be divided into a number of independent tasks, it is often possible to reduce the overall running time needed to complete them by executing them in parallel. This may involve exploiting multi-core processors on a single machine or using a multiple machine configuration such as a grid. Previously, Schuler et al. [7] successfully employed parallelisation for program mutation analysis with the JAVALANCHE tool. JAVALANCHE can execute multiple Java program mutants simultaneously as a means of reducing the time taken to evaluate a test suite.

III. Improving The Efficiency of Mutation Analysis for Database Structure

Using the ideas of mutant schemata and parallelisation, we now present four different approaches to improving the efficiency of mutation analysis of database structure.

A. Mutant Schemata

With the "Original" approach, introduced in Section II, each mutant is evaluated one-by-one by creating the mutant's database tables, running the test suite, and then removing the tables. To potentially speed up this process, we apply the mutant schemata approach by creating a meta-mutant – a large database schema containing all of the tables required for each mutant. Some "housekeeping" is required to make this work, since there cannot be more than one table in the database with the same name, and there will likely be several mutant versions of a table that need to be combined into the meta-mutant. Therefore, we assign a unique identifier to each mutant and use this to prefix identifiers pertaining to a mutant so as to produce unique table names (and named constraints etc.). Using an identifier we can then "activate" a single mutant by prefixing insert statements with it.

Following renaming, there are different ways in which the mutants may be combined into a single meta-mutant. We explore two approaches:

1) Full Schemata: The "Full Schemata" approach involves creating a meta-mutant out of all the renamed tables of each mutant. An example snippet can be seen in Figure 3 for the *UnixUsage* example, which was introduced in Figure 1 and discussed in the last section.

```
CREATE TABLE mutant_1_USAGE_HISTORY (
  USER_ID VARCHAR(50) NOT NULL,
  SESSION_ID INTEGER,
  LINE_NO INTEGER,
  COMMAND_SEQ INTEGER,
  COMMAND VARCHAR(50),
  PRIMARY KEY (USER_ID)
  FOREIGN KEY (USER_ID)
    REFERENCES USER_INFO (mutant_1_USER_ID)
);
CREATE TABLE mutant_1_USER_INFO (...);
CREATE TABLE mutant_1_UNIX_COMMAND (...);
CREATE TABLE mutant_1_TRANSCRIPT (...);
CREATE TABLE mutant_1_RACE_INFO (...);
CREATE TABLE mutant_1_OFFICE_INFO (...);
CREATE TABLE mutant_1_DEPT_INFO (...);
CREATE TABLE mutant_1_COURSE_INFO (...);

CREATE TABLE mutant_2_USAGE_HISTORY (
  USER_ID VARCHAR(50) NOT NULL,
  SESSION_ID INTEGER,
  LINE_NO INTEGER,
  COMMAND_SEQ INTEGER,
  COMMAND VARCHAR(50) NOT NULL,
  FOREIGN KEY (USER_ID)
    REFERENCES USER_INFO (mutant_2_USER_ID)
);
CREATE TABLE mutant_2_USER_INFO (...);
CREATE TABLE mutant_2_UNIX_COMMAND (...);
CREATE TABLE mutant_2_TRANSCRIPT (...);
CREATE TABLE mutant_2_RACE_INFO (...);
CREATE TABLE mutant_2_OFFICE_INFO (...);
CREATE TABLE mutant_2_DEPT_INFO (...);
CREATE TABLE mutant_2_COURSE_INFO (...);
...
```

(a) Meta-mutant snippet

```
1. INSERT INTO mutant_1_USER_INFO
  VALUES ('laura', ...);
2. INSERT INTO mutant_1_USAGE_HISTORY
  VALUES ('laura', 1, 10, 1, 'awk');
3. INSERT INTO mutant_1_USAGE_HISTORY
  VALUES ('laura', 2, 10, 2, 'grep');
4. INSERT INTO mutant_2_USER_INFO
  VALUES ('laura', ...)
5. INSERT INTO mutant_2_USAGE_HISTORY
  VALUES ('laura', 1, 10, 1, NULL)
```

(b) Snippet of an example test suite

Figure 3. Using the "Full Schemata" approach. (a) Snippet of the meta-mutant database schema for *UnixUsage*, showing the prefixed tables included for each mutant, with text highlighted to show mutations and modified references to other mutant tables. (b) Form of the test suite, with the `INSERT`s modified to match the table names in the meta-mutant.

```
▷ 1. Meta-mutant creation
for each mutant do
    Prefix names of tables in mutant with unique mutant ID
end for
Create tables in database for all mutants

▷ 2. Mutant evaluation
K ← ∅
for each mutant do
    killed ← false
    for each sqlInsertStatement in testSuite do
        sqlInsertStatement′ ← sqlInsertStatement modified to use unique
                                  mutant ID of mutant for table names
        originalResult ← Pre-computed result of insert with non-mutant
        mutantResult ← executeWithDBMS(sqlInsertStatement)
        if originalResult ≠ mutantResult then
            K ← K ∪ {mutant}
        end if
    end for
end for

▷ 3. Clean up
Remove tables in database for all mutants
```

Figure 4. "Full Schemata" mutation analysis algorithm

The algorithm for mutation analysis using "Full Schemata" is shown in Figure 4. Since all tables involving the original database structure and its mutants may be created together at the same time, and also later all removed simultaneously, the respective SQL `CREATE TABLE` and `DROP TABLE` commands may be issued in a single database interaction, potentially speeding up the complete process.

Evaluation of mutants using a test suite can then be conducted in a similar fashion to the "Original" approach, repeating each `INSERT` statement for each mutant, but with the automated adjustment of table names referenced in each `INSERT` statement so that it exercises the correct table belonging to the mutant. An example of `INSERT` renaming and execution of the test suite for each mutant in the meta-mutant can be seen in Figure 3(b).

Following mutant evaluation, all tables in the meta-mutant may be deleted from the database by concatenating the SQL `DROP TABLE` commands into a single database interaction.

2) Minimal Schemata: The "Minimal Schemata" produces a meta-mutant by combining the original database schema with *only* the tables for each individual mutant that have actually been modified – the mutant's so-called *affected tables*. An example meta-mutant for *UnixUsage* can be seen in Figure 5(a). The first two mutants involve modifications to the `USAGE_HISTORY` table only, and not any of the other tables also belonging to the database schema. Therefore, only the modified versions of `USAGE_HISTORY` appear in the meta-mutant (`mutant1_USAGE_HISTORY` and `mutant2_USAGE_HISTORY`) and not additional duplications of the other tables. If any affected tables have foreign key relationships with unaffected tables, the foreign key is left unchanged to reference a common unmodified version of the table – which is copied from the original, unmutated, database schema. Therefore, the mutant versions of

```
CREATE TABLE USAGE_HISTORY (
  USER_ID VARCHAR(50) NOT NULL,
  SESSION_ID INTEGER,
  LINE_NO INTEGER,
  COMMAND_SEQ INTEGER,
  COMMAND VARCHAR(50),
  FOREIGN KEY (USER_ID)
    REFERENCES USER_INFO (USER_ID)
);
CREATE TABLE USER_INFO (...);
CREATE TABLE UNIX_COMMAND (...);
CREATE TABLE TRANSCRIPT (...);
CREATE TABLE RACE_INFO (...);
CREATE TABLE OFFICE_INFO (...);
CREATE TABLE DEPT_INFO (...);
CREATE TABLE COURSE_INFO (...);

CREATE TABLE mutant_1_USAGE_HISTORY (
  USER_ID VARCHAR(50) NOT NULL,
  SESSION_ID INTEGER,
  LINE_NO INTEGER,
  COMMAND_SEQ INTEGER,
  COMMAND VARCHAR(50),
  PRIMARY KEY (USER_ID)
  FOREIGN KEY (USER_ID)
    REFERENCES USER_INFO (USER_ID)
);

CREATE TABLE mutant_2_USAGE_HISTORY (
  USER_ID VARCHAR(50) NOT NULL,
  SESSION_ID INTEGER,
  LINE_NO INTEGER,
  COMMAND_SEQ INTEGER,
  COMMAND VARCHAR(50) NOT NULL,
  PRIMARY KEY (USER_ID)
  FOREIGN KEY (USER_ID)
    REFERENCES USER_INFO (USER_ID)
);
...
```

(a) Meta-mutant snippet

```
1. INSERT INTO USER_INFO
  VALUES ('laura', ...);
2. INSERT INTO mutant_1_USAGE_HISTORY
  VALUES ('laura', 1, 10, 1, 'awk');
3. INSERT INTO mutant_1_USAGE_HISTORY
  VALUES ('laura', 2, 10, 2, 'grep');
4. INSERT INTO mutant_2_USAGE_HISTORY
  VALUES ('laura', 1, 10, 1, NULL);
```

(b) Snippet of an example test suite

Figure 5. Using the "Minimal Schemata" approach. (a) Snippet of the meta-mutant database schema for *UnixUsage* – including only the "affected" tables of each mutant – showing the prefixed tables included for each mutant, with text highlighted to show mutations and (unmodified) references to other mutant tables. (b) Form of the test suite, with the `INSERT` statements modified to match the table names in the meta-mutant, in order to manage the dependencies between affected and original tables.

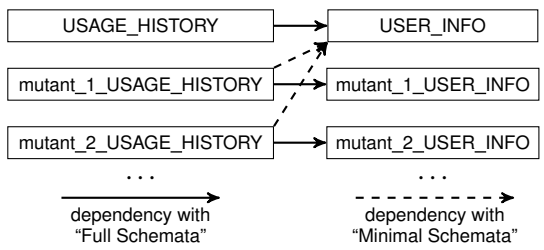

Figure 6. Dependencies between tables using the "Full Schemata" and "Minimal Schemata" approaches for the *UnixUsage* example. The "Minimal" approach relies upon common tables where the table is unaffected by mutation, whereas the "Full" approach includes copies of those tables for each mutant.

```
▷ 1. Meta-mutant creation
for each mutant do
    mutant′ ← mutant with non-affected tables removed
    Prefix names of table in mutant′ with unique mutant ID
end for
Create tables in database for all mutants and original structure

▷ 2. Mutant evaluation
K ← ∅
for each sqlInsertStatement in testSuite do
    originalResult ← Pre-computed result of insert with non-mutant
    affectedTable ← The table the insert is involving
    affectedMutants ← The mutants that mutated affectedTable
    executeWithDBMS(sqlInsertStatement)
    for each affectedMutant do
        sqlInsertStatement′ ← sqlInsertStatement modified to use unique
                              mutant ID of affectedMutant for table name
        mutantResult ← executeWithDBMS(sqlInsertStatement′)
        if originalResult ≠ mutantResult then
            K ← K ∪ {mutant}
        end if
    end for
end for

▷ 3. Clean up
Remove tables in database for all mutants
```

Figure 7. "Minimal Schemata" mutation analysis algorithm

USAGE_HISTORY reference the original USAGE_INFO table in Figure 5(a). This critical difference with the "Full Schemata" approach is illustrated in Figure 6.

While the "Minimal Schemata" approach aims to reduce the number of tables that the DBMS must store, it influences the way mutation analysis must be performed and the manner in which the INSERT statements of the test suite are applied. Care must be taken because the meta-mutant involves tables that are now common to several mutants. As such, the insertion of any common data must be managed to ensure that it is present at the time it is needed by the dependent mutant tables, and not before, so that it does not interfere with the correct evaluation of each mutant.

Therefore, instead of re-running the test suite from start to finish each time for each mutant, as with the approaches previously described, evaluation of mutants works on an INSERT statement-by-statement basis, as shown in Figure 7. The algorithm works down the list of INSERTs in the test suite, one-by-one. First, the INSERT is executed unmodified, so that common data is added to the database for the later evaluation of mutant tables that may reference it. Then, the INSERT is automatically modified for each mutant version of the table in the meta-mutant and executed. In this way, each mutant with an affected version of the table is partially evaluated within the same step of the test suite. Full evaluation of all mutants is not completed until the final INSERT statement of the test suite is processed. An example set of INSERTs can be seen in Figure 5(b). First, an INSERT is made to the USER_INFO table. Then INSERTs can be made for each of the mutant tables that have a common reference to it.

```
▷ 1. Meta-mutant creation
for each mutant do
    Prefix names of tables in mutant with unique mutant ID
end for

▷ 2. Mutant evaluation
K ← ∅
parallel for each mutant do
    Create tables in database for mutant
    for each sqlInsertStatement in testSuite do
        originalResult ← Pre-computed result of insert with non-mutant
        mutantResult ← executeWithDBMS(sqlInsertStatement)
        if originalResult ≠ mutantResult then
            K ← K ∪ {mutant}
        end if
    end for
    Remove tables in database for mutant
end parallel for
```

Figure 8. "Just-in-Time Schemata" mutation analysis algorithm

B. Parallelisation

The use of mutant schemata for database structure gives rise to the possibility of easily parallelising mutant evaluation; mutants exist simultaneously in a database, and many database management systems allow for concurrent access and manipulation of the data. We describe two ways in which the mutant schemata approach may be parallelised:

1) Up-Front Schemata: The "Up-Front" approach is essentially the same as the "Full Schemata" approach (Figure 4), but with mutation evaluation performed in parallel in order to decrease overall mutation analysis time.

2) Just-in-Time Schemata: The observation behind the "Just-in-Time Schemata" approach is that the meta-mutant does not have to be completely created in the database before mutation evaluation. The database tables corresponding to each mutant can be added over time, and also in parallel, just before (hence the name) the mutant needs to be evaluated, and deleted straight afterwards. As such, following table and identifier relabelling, the main loop the "Just-in-Time" approach (Figure 8) is almost identical to that of the original approach (Figure 2), except it is executed in parallel.

C. Summary of techniques

Figure 9 shows the hierarchy and relationships of the techniques to improve the efficiency of the "Original" approach, described in Section II-A. We have implemented two mutant schemata approaches: "Full Schemata" (Section III-A1) and "Minimal Schemata" (Section III-A2). The "Full" approach involves a lot of duplication of the tables of the original database, however it is very easy to parallelise – as implemented with the "Up-Front" (Section III-B1) and "Just-in-Time" (Section III-B2) methods. The "Minimal" approach involves implicit dependencies of INSERT statements across mutants' tables and tables from the original database structure, rendering parallelisation difficult. We leave this as an issue for future work, as mentioned in Section VI.

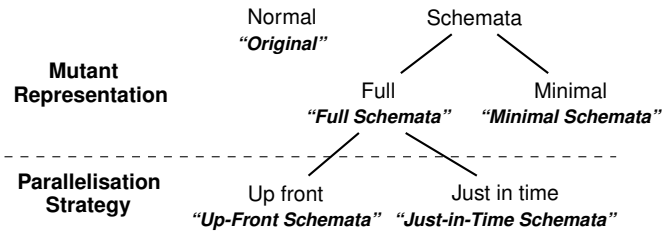

Figure 9. The relationships between techniques in the empirical study

Table I lists some equations describing the characteristics of each algorithm with respect to the database. Naturally, the "Original" approach has the fewest tables present in the database at any one time, as it evaluates each mutant one-by-one, resulting in the database having the lowest "load" at any one time. In this aspect, only "Just-in-Time Schemata" is similar, creating tables in parallel as it needs them. However, both are more expensive in terms of atomic `CREATE` and `DROP` table statements – all others have a constant number of interactions (just 2). All approaches involve executing the entire test suite for every mutant, except "Minimal", which tries to minimize this aspect of mutation analysis. While this is designed to reduce the number of database interactions, this depends on the database structure and the number of dependencies between tables.

However, the characteristics of the underlying DBMS limit a purely formal analysis, and for this reason this paper also reports on an empirical study involving a series of case study database schemas hosted by two different DBMSs.

IV. Empirical Study

A. Experimental Setup

The performance of the techniques under various conditions are evaluated using a selection of 6 case study database schemas, which range in the number of tables, columns, and constraints they involve (as well as the number of mutants that can be generated for them), as detailed in Table II. *Cloc* is part of the data output functionality for a popular open-source application (*http://cloc.sourceforge.net*) that counts the number of code, blank, and comment lines in a directory or archive for a large number of programming languages. *JWhoisServer* is from a freely available Java-based "WHOIS" server implementation (*http://jwhoisserver.net*). *RiskIt* is used in an insurance adjustment application (*http://sourceforge.net/projects/riskitinsurance*) that models the probability of an individual making a claim to calculate an appropriate premium cost. Both *NistDML182* and *NistDML183* are from the SQL Conformance Test Suite of the National Institute of Standards and Technology (NIST) (*http://www.itl.nist.gov/fipspubs/fip193.htm*). Finally, *UnixUsage* is a schema from the logging application introduced in Section II, presented in digested form in Figure 1.

Table I
Database interaction characteristics of the techniques

Technique	Simultaneous tables in the database	'Create' and 'Drop' statements executed	'Insert' statements executed
Original	T	$2 \times T$	$M \times I$
Full Schemata	$T \times M$	2	$M \times I$
Minimal Schemata	$T + M$	2	$I + (M \times I_{min})$
Up-Front Schemata	$T \times M$	2	$M \times I$
Just-in-Time Schemata	$T \leq Sim \leq T \times P$	$2 \times M$	$M \times I$

T = number of tables in a case study
P = number of parallel processes
M = number of mutants
I = number of `INSERT` statements
I_{min} = minimal number of `INSERT`s ($0 < I_{min} \leq I$)
Sim = number of simultaneous tables in the database

Each of the case studies were evaluated using the Postgres and SQLite DBMSs, chosen because of their significantly different architecture and wide use. SQLite is an embedded database that writes data to one file, designed for single clients. Alternatively, Postgres is a client-server database that is highly scalable, supporting tables up to 32TB in size [9], and designed to allow many concurrent users.

To perform the experiments, we used the Java programming language to implement our approach in the *SchemaAnalyst* tool [3]. *SchemaAnalyst* was compiled with the JDK 7 compiler and executed with the Oracle Java 1.7 64-bit virtual machine for Linux. We executed the experiments on an Ubuntu 12.04 machine, 3.2.0-27 GNU/Linux 64-bit kernel, with a quad-core 2.4 GHz CPU and 12GB RAM. All files were stored on a 280 GB local disk. For the parallel method, we configured the Java code to use a fixed pool of eight threads. The specific DBMS versions were Postgres 9.1.6 and SQLite 3.7.9, used in their default configurations. The mutation operators of Kapfhammer et al. [3] produce some mutants that are still-born [10] for some DBMSs – referred to as "quasi-mutants" because they are immediately rejected by some DBMSs and not others. The operators capable of producing quasi-mutants were not used in this study.

We measured wall-clock time for mutation analysis with each combination of approach, case study, and DBMS, repeating an experiment 30 times to estimate reliable averages and medians, as well as to enable statistical analysis.

As SQLite does not allow multiple connections to modify the database simultaneously [11], we could not run the parallel mutant schemata approaches for this DBMS (i.e., "Full" and "Just-in-Time" Schemata). This is because SQLite locks the database when running the SQL commands that create, drop, and insert data into tables, thus rendering our parallel approaches useless for this DBMS. Regardless of this limitation, we included SQLite as it supports the performance evaluation of the remaining techniques on a DBMS with an architecture very different from Postgres.

Table II
DATABASE STRUCTURES USED FOR THE EMPIRICAL STUDY

Case study	Tables	Columns	Checks	Foreign keys	Not Nulls	Primary keys	Uniques	Total Constraints	Mutants
Cloc	2	10	0	0	0	0	0	0	30
JWhoisServer	6	49	0	0	44	6	0	50	184
NistDML182	2	32	0	1	0	1	0	2	66
NistDML183	2	6	0	1	0	0	1	2	18
RiskIt	13	56	0	10	15	11	0	36	160
UnixUsage	8	32	0	7	9	7	0	23	69
Total	33	185	0	19	68	25	1	113	527

Table III
SUMMARY OF TABLES AND INSERTS USED FOR TECHNIQUES

The number of tables used, first, and the number of `INSERT` statements executed, second and in italics, for mutation analysis.

Case study	Original		Full Schemata		Minimal Schemata	
Cloc	60	*120*	60	*120*	32	*64*
JWhoisServer	1104	*11408*	1104	*11408*	190	*2582*
NistDML182	132	*462*	132	*462*	68	*220*
NistDML183	36	*126*	36	*126*	20	*70*
RiskIt	2080	*16320*	2080	*16320*	173	*1511*
UnixUsage	552	*5658*	552	*5658*	77	*714*
Total	3964	*34094*	3964	*34094*	560	*5161*

B. Empirical Results

Figures 10 and 11 show box plots comparing each of our mutation schemata approaches with the "Original" approach for the Postgres and SQLite DBMSs respectively, using the set of 30 execution times measured for each approach. For each box, the middle line through the box represents the median, while the upper and lower bounds of the box represent the 1st and 3rd quartiles respectively, with whiskers representing 1.5 $\times$ the interquartile range and circles representing outliers. Where not obvious from the box plots, we conducted tests for significance with the nonparametric Wilcoxon rank sum test, using the sets of 30 execution times obtained with a particular DBMS and the pair of approaches under scrutiny. A p-value of less than 0.05 is deemed to be significant. To complement significance tests, the nonparametric $\hat{A}_{12}$ statistic of Vargha and Delaney [12] was used to compute effect sizes, which determine the average probability that one approach out performs another. We followed the guidelines of Vargha and Delaney in that an effect size is deemed to be "large" if the value of $\hat{A}_{12}$ is < 0.29 or > 0.71, "medium" if $\hat{A}_{12}$ is < 0.36 or > 0.64 and "small" if $\hat{A}_{12}$ is < 0.44 or > 0.56. Values of $\hat{A}_{12}$ close to the 0.5 value are deemed to have no size.

We now present an analysis of these results in the context of four research questions (RQs):

RQ1: Do mutation schemata approaches improve the efficiency of mutation analysis for database structure?: For the Postgres DBMS, each mutant schemata approach clearly outperforms the "Original" technique, with reductions of up to an order of magnitude with the "Minimal Schemata" approach evident for the *RiskIt* case study.

Yet, the picture is not so straightforward for SQLite, the DBMS for which only two mutant schemata approaches work because of its inability to handle concurrent connections. With SQLite, "Full Schemata" takes longer to perform mutation analysis with all of the case studies. This is likely due to the increase in the number of tables and the amount of data that the database must store at one time. However, big gains were still possible – for example, a reduction of approximately 9.9 times with "Minimal Schemata" for *UnixUsage* and a decrease in mean time from ∼20.5 minutes to ∼2.5 minutes with "Minimal Schemata" for *RiskIt*. This is likely caused by the consistently reduced number of tables and `INSERT` statements used for mutation analysis which, as shown in Table III, is considerable for all case studies. For example, the number of tables used for the mutation analysis of *RiskIt* falls from 2,080, when using the "Original" or "Full Schemata" approaches, to 173, when using the "Minimal Schemata" technique. Similarly, the number of `INSERT` statements reduces greatly from 16,320 to 1,511.

The conclusion for this research question therefore, is *yes*, mutation schemata approaches can improve efficiency, using the "Minimal Schemata" approach for SQLite and all of the approaches for the Postgres DBMS.

RQ2: Does parallelisation of mutant schemata approaches further reduce the time taken for mutation analysis?: As reported in Section IV-A, due to the limitations of SQLite, we were only able to conduct experiments for parallel approaches with Postgres. The results in Figure 10 show that with Postgres, the "Up-Front Schemata" technique performed similarly to the non-parallel "Full Schemata", for all of the case studies, and was beaten by the non-parallel "Minimal" approach. No significant differences were found between the "Full" and "Up-Front" techniques, with the exception of the *JWhoisServer* and *NistDML183* case studies. For *JWhoisServer*, "Up-Front" was significantly faster than "Full", with a p-value of less than 10^{-6} and a large effect size ($\hat{A}_{12}$= 0.86). Conversely, for *NistDML183*, "Up-Front" was significantly slower than "Full" with $p = 0.011$, and a with medium effect size ($\hat{A}_{12} = 0.69$).

As seen in Figure 10, "Just-in-Time Schemata" clearly outperformed "Full" and "Up-Front", in all cases but for *Cloc*. However, comparisons between the "Just-in-Time" and "Minimal" approaches are not so obvious from the box plots. Closer analysis with the Wilcoxon rank-sum test revealed no significant differences between the two approaches with each individual case study, except for *NistDML183*, where "Just-in-Time" was significantly faster than "Minimal" with a p-value less than 10^{-6} and a large effect size ($\hat{A}_{12} = 0.12$). For *Cloc*, the similar performance of all mutant schemata approaches is likely due to the small size of this case study, which involves only two tables and no integrity constraints.

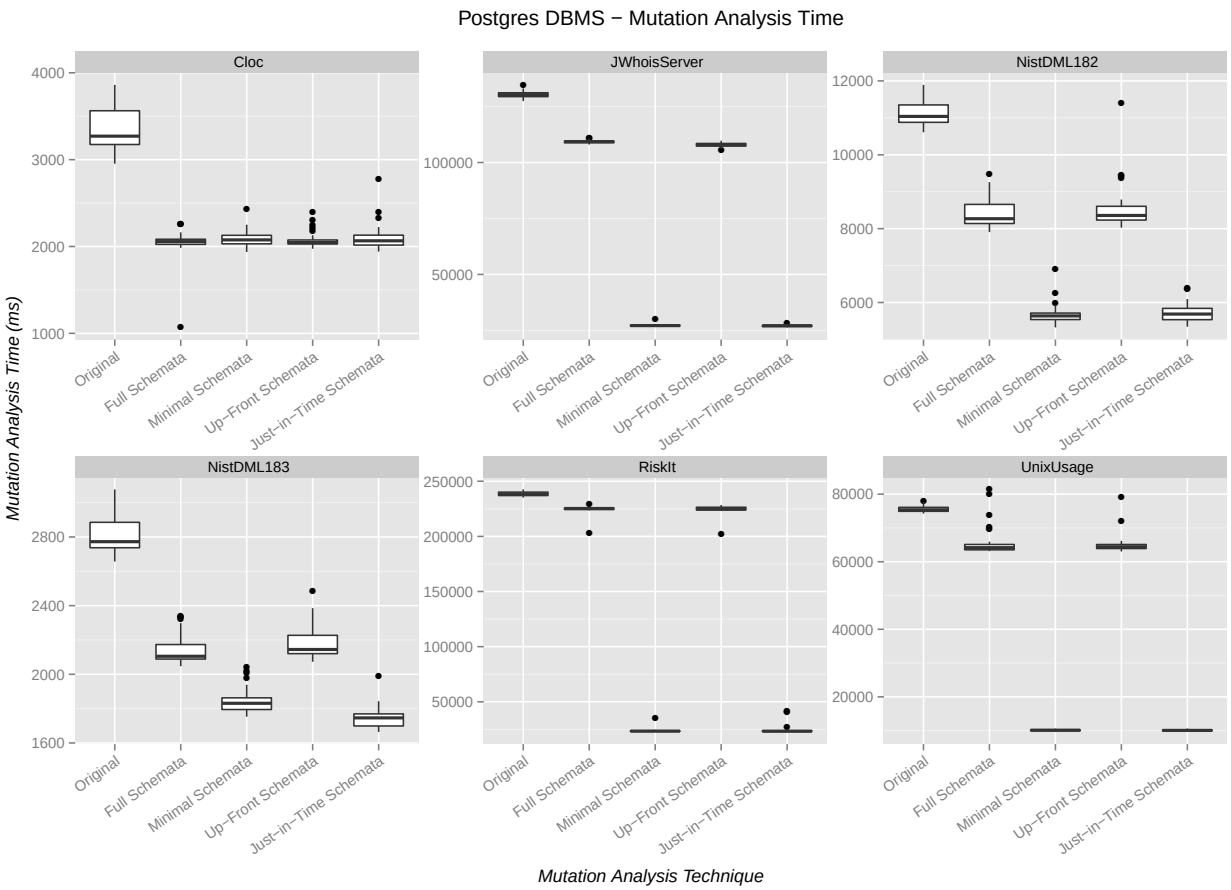

Figure 10. Mutation analysis time results for the Postgres DBMS, for each case study (box spans from 1st to 3rd quartile, line marks the median, whiskers extend up to 1.5× the inter-quartile range, filled circles mark outliers).

In conclusion, the results for the parallelised versions of mutant schemata are mixed. Gains are possible over naïve non-parallel mutant schemata approaches such as "Full", but there is little difference between the parallel "Just-in-Time" and the non-parallel "Minimal", excepting that parallelism yields significantly better performance with *NistDML183*.

RQ3: Which is the best overall approach?: For SQLite, where the parallelised mutant schemata approaches are not available, the "Minimal" technique is clearly the most time-efficient, as seen in Figure 11. For the Postgres DBMS, the "Minimal" approach has a similar performance to the "Just-in-Time" parallel method, as discussed in the last research question. In conclusion, "Minimal" is the best overall approach if parallelisation is not an option; otherwise "Just-in-Time" and "Minimal" are equally as suitable.

RQ4: Does the efficiency of an approach depend on the DBMS used?: The answer to this question is *yes*: For SQLite and the "Full Schemata" approach, the cost associated with storing additional tables simultaneously outweighs the savings from the reduction of `CREATE` and `DROP` statements executed – yet the same is not true of Postgres. While the "Minimal" approach does result in significant time savings for both DBMS types, the parallel mutant schemata approaches could not be evaluated on SQLite – so, a comparison was a non-starter in this regard. Therefore, it does depend on the DBMS as to whether a particular mutant schemata approach is useful in speeding up mutation analysis or not. The reasons for this are particular to the architecture of the DBMS itself. In this study we compared SQLite, a lightweight DBMS intended mainly for single-user access and embedding into stand-alone applications, against Postgres, a "heavy-duty" DBMS that is disk-based and client-server and thus capable of supporting many connections at once. As mentioned in Section VI, as part of future work we intend to investigate this issue further with more DBMSs and different DBMS architectures.

C. Threats to Validity

Potential threats to so-called "external validity" involve factors that may affect the extent to which the results of the study may be more widely generalisable. One such threat of this nature comes from the type of the case studies used to evaluate the presented approaches to mutant schemata. The case studies were chosen based upon a number of criteria, as discussed more thoroughly in Section IV-A, and vary considerably in terms of the number of database tables (2 to 13), columns (6 to 56) and integrity constraints (0 to 50),

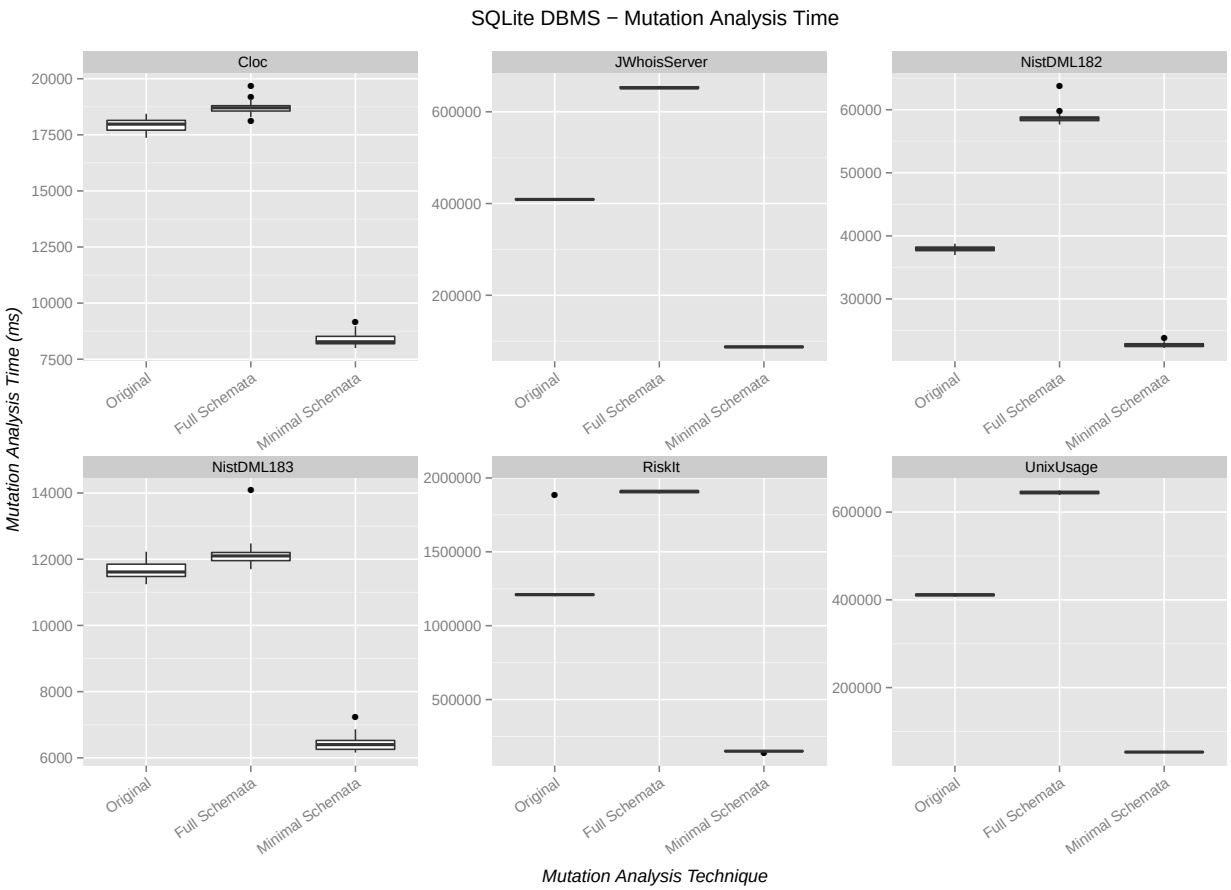

Figure 11. Mutation analysis time results for the SQLite DBMS, for each case study (box spans from 1st to 3rd quartile, line marks the median, whiskers extend up to 1.5× the inter-quartile range, filled circles mark outliers).

as well as the number of mutants produced (16 to 184); Table II contains more information. While the "Nist" case studies were the subject of a previous paper on the mutation testing of database queries [13], we also took four of the case studies from real-world applications, with three of these four – *JWhoisServer*, *RiskIt*, and *UnixUsage* – having been used in previous studies by Pan et al. [4] and Cobb et al. [14]. Of course, as already discussed in the Section IV-B, the use of different DBMSs may also lead to different results.

A further threat may also come from the type of machine used to perform the experiments. Although DBMSs perform a variety of operations in-memory, they also rely heavily on disk-based access. Our machine uses a traditional mechanical hard disk drive, and different results may be obtained with machines using newer solid-state drives. Yet, mechanical disks are still the commonest form of storage available at the time of writing, and as such our results are likely to be applicable to the majority of users. Investigation of different machine configurations and their effects on the performance of our techniques is an issue for future work.

Another such threat comes from the thread pool size used in the implementation of "Up-Front Schemata" and "Just-in-Time Schemata". Only one configuration was included in the empirical study – however this was enough to demonstrate significant reductions in execution time. While it is possible that varying this number could further improve mutant analysis' performance, this is an issue for future work.

Threats to *internal validity* involve factors that could have introduced an error or bias into our results, such that the conclusions drawn are incorrect. Our empirical study was designed to investigate the time required by different techniques to do the same task – mutation analysis – but using a different approach (i.e., through mutation schemata or with a parallelisation strategy). One such threat could therefore result from defects in our implementation of the mutation analysis process. However, we checked the mutation score and the mutants that were killed and not killed by each method. This information was identical in all cases, leading us to judge that we correctly implemented each of the techniques. Another potential source of error may arise from resetting the DBMS into a consistent "state" before applying each trial for the combination of each mutant schemata approach and case study. To this end, we removed all tables and their data at the end of each trial, thus ensuring that the database was correctly emptied.

Finally, in terms of statistical analysis, we employed non-parametric tests, which do not require the need to make or test assumptions about the normality of the sample means, thus avoiding the introduction of a further potential source of error into the empirical study.

V. Related Work

The original mutant schemata work focused on an implementation for Fortran [8]; however, more recent work has extended the approach to other programming languages including Java [15], [16]. While the first parallel mutation testing methods ran on MIMD machines like the hypercube [6], recent approaches focus on the Java programming language and run on general-purpose computers [7].

Parallel approaches have also been explored for the testing of database management systems (DBMSs). For instance, Haftmann et al. [17] describe and evaluate two parallel architectures for comparing and testing different DBMSs: "Shared-Nothing" runs tests on isolated machines that host separate databases while "Shared-Database" employs multiple threads on a single machine that hosts one database instance. Yet, it should be noted that these architectures only support DBMS testing and not parallel mutation analysis of database structure – the problem addressed in this paper.

Some prior work, such as Tuya et al. [13] and Shah et al. [18], focus on the mutation testing of database queries instead of the relational database schema. As an extension to Tuya et al.'s method, Zhou and Frankl [19] present ways to mutate database queries in the context of the database application's source code and execution environment. While Chan et al. [20] do consider the mutation of database structure, they neither describe an implementation of mutation analysis nor furnish an empirical evaluation. Moreover, none of the aforementioned papers describe techniques that employ either mutant schemata or parallelisation to improve the performance of mutation analysis for relational databases.

VI. Conclusions and Future Work

Even though mutation analysis is an effective way to assess the quality of input values and test oracles, it is a computationally expensive method. This paper describes and empirically evaluates four approaches that leverage mutant schemata for improving the time efficiency of evaluating database structure mutants, with two of these approaches also employing parallelisation. While the results vary depending on the DBMS used, they show that, for both SQLite and Postgres, our techniques lead to a one to ten times reduction in mutation analysis time. These promising results suggest that it is possible to use mutation analysis to efficiently evaluate the quality of both automatically and manually generated `INSERT` statements and data values.

As part of future work, we will investigate different strategies for improving approaches to mutant schemata and perform further investigations with additional, larger case studies and different types of DBMS. We also intend to explore how the modification of various experimental parameters, such as the type of disk drive used for hosting the DBMS and the number of threads in the thread pool, influence the efficiency of the presented methods. Finally, we will improve our experimentation tools to record the time taken in the different phases of each approach, thus allowing for a more detailed analysis and comparison.

References

[1] B. Butler, "Amazon: Our cloud powered Obama's campaign," *Network World*, 2012.

[2] S. Loebman, D. Nunley, Y. Kwon, B. Howe, M. Balazinska, and J. P. Gardner, "Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help?" in *Proc. of IASDS*, 2009.

[3] G. M. Kapfhammer, P. McMinn, and C. J. Wright, "Search-based testing of relational schema integrity constraints across multiple database management systems," in *Proc. of ICST*, 2013.

[4] K. Pan, X. Wu, and T. Xie, "Generating program inputs for database application testing," in *Proc. of ASE*, 2011.

[5] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Transactions on Software Engineering*, vol. 37, no. 5, 2011.

[6] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation testing of software using a MIMD computer," in *Proc. of ICPP*, 1992.

[7] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for Java," in *Proc. of ESEC/FSE*, 2009.

[8] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proc. of ISSTA*, 1993.

[9] PostgreSQL Project, "Frequently asked questions," http://www.postgresql.org/about/, (Accessed 17/12/2012).

[10] A. J. Offutt, J. Voas, and J. Payne, "Mutation operators for Ada," Department of Information and Software Systems Engineering, George Mason University, Tech. Rep., 1996.

[11] SQLite Developers, "Frequently asked questions," http://www.sqlite.org/faq.html#q5, (Accessed 21/12/2012).

[12] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Education and Behavioral Statistics*, vol. 25, no. 2, 2000.

[13] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Mutating database queries," *Information and Software Technology*, vol. 49, no. 4, 2006.

[14] J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold, "Dynamic invariant detection for relational databases," in *Proc. of WODA*, 2011.

[15] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, 2005.

[16] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *Proc. of ASE*, 2011.

[17] F. Haftmann, D. Kossmann, and E. Lo, "Parallel execution of test runs for database application systems," in *Proc. of VLDB*, 2005.

[18] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. Gupta, and D. Vira, "Generating test data for killing SQL mutants: A constraint-based approach," in *Proc. of ICDE*, 2011.

[19] C. Zhou and P. Frankl, "JDAMA: Java database application mutation analyser," *Software Testing, Verification and Reliability*, vol. 21, no. 3, 2011.

[20] W. K. Chan, S. C. Cheung, and T. H. Tse, "Fault-based testing of database application programs with conceptual data model," in *Proc. of QSIC*, 2005.