The Impact of Equivalent, Redundant and Quasi Mutants on Database Schema Mutation Analysis

Chris J. Wright
Dept. of Computer Science
University of Sheffield

Gregory M. Kapfhammer
Dept. of Computer Science
Allegheny College

Phil McMinn
Dept. of Computer Science
University of Sheffield

Abstract—**Since the relational database is an important component of real-world software and the schema plays a major role in ensuring the quality of the database, relational schema testing is essential. This paper presents methods for improving both the efficiency and accuracy of mutation analysis, an established method for assessing the quality of test cases for database schemas. Using a DBMS-independent abstract representation, the presented techniques automatically identify and remove mutants that are either equivalent to the original schema, redundant with respect to other mutants, or undesirable because they are only valid for certain database systems. Applying our techniques for ineffective mutant removal to a variety of schemas, many of which are from real-world sources like the U.S. Department of Agriculture and the Stack Overflow website, reveals that the presented static analysis of the DBMS-independent representation is multiple orders of magnitude faster than a DBMS-specific method. The results also show increased mutation scores in 75% of cases, with 44% of those uncovering a mutation-adequate test suite. Combining the presented techniques yields mean efficiency improvements of up to 33.7%, with averages across schemas of 1.6% and 11.8% for HyperSQL and PostgreSQL, respectively.**

I. Introduction

An essential part of many real-world software systems, the relational database is usually one of an organization's most valuable assets. The structure of a relational database is defined in a schema that is normally expressed in the structured query language (SQL). The schema describes the types of data to be stored in a database, the organization of that data into tables and the constraints to which the data must conform [1].

Since it is crucial to maintain the integrity of the database's contents, the correctness of the database schema is of the utmost importance. Without a correctly defined schema, an application may, for instance, incorrectly create two users with the same login ID or products with prices that are less than zero. Even though schema correctness is essential, practitioners report that database schema testing is a frequently overlooked aspect of software quality assurance [2].

While several techniques (e.g., [3], [4], [5]) exist to generate data for a database that can be used as "test cases" as part of a schema testing strategy, a method is required to assess the quality of those generated test cases. An effective way to perform such an evaluation is through mutation analysis [6], [7]. For example, by employing mutation analysis, Kapfhammer et al. [8] found that the *SchemaAnalyst* tool was more effective at testing database constraints than the popular *DBMonster* tool. Mutation analysis involves repeatedly replicating a software artifact and systematically injecting hypothesized faults. If a test case behaves differently with the mutant compared to the original schema, the mutant is said to be "killed". The more mutants a test suite kills, the more likely it is to reveal real faults that are similar to the mutants that it kills.

Yet, mutation analysis is often a slow and expensive process, requiring the generation and evaluation of many mutants, not all of which make a useful contribution to the analysis. For instance, some mutants may represent invalid schemas. Other mutants may be equivalent to the original schema or the same as other mutants—not only consuming unnecessary computational and human resources, but potentially affecting the accuracy of the analysis. The automatic detection and removal of these types of mutants has been a topic of concern for program mutation (e.g., [9], [10]), but it has not, hitherto, been studied in the context of database schema mutation.

This paper proposes techniques for removing these "ineffective" mutants during database schema mutation analysis. The initial step involves parsing schemas into an abstract, DBMS-independent representation. Mutation is then performed at the SQL-semantic level, in contrast to the traditional approach for program mutation that creates mutants by syntactically manipulating the program's source code. Since there are many different ways in which the same schema may be expressed in the numerous and complex database-specific variants of SQL, a standard syntactic approach can generate many mutants that are trivially equivalent to the original, non-mutant schema. While the abstract representation enables the static detection of equivalent mutants, it also supports the identification of redundant mutants that are semantically the same as one or more other mutants. In addition, while some mutants are valid schemas for certain DBMSs, they may be invalid for others. The DBMS-independent representation also supports the static identification of these mutants, allowing them to be culled before starting the expensive process of mutant evaluation.

Including sixteen database schemas that vary in their source (e.g., the U.S. Department of Agriculture and the Stack Overflow website) and the number of tables (2 to 22), columns (3 to 67), and constraints (0 to 50), the empirical study shows that, without compromising the resulting mutation score, the static analysis of the DBMS-independent abstract representation is multiple orders of magnitude faster than a DBMS-specific method. The experiments also demonstrate that the removal of ineffective mutants improves both the efficiency

```
CREATE TABLE artists (
 artist_id text PRIMARY KEY
);
CREATE TABLE similarity (
 target text,
 similar text,
 FOREIGN KEY(target) REFERENCES artists(artist_id),
 FOREIGN KEY(similar) REFERENCES artists(artist_id)
);
```

(a) Schema definition

```
INSERT INTO artists VALUES ("A");
INSERT INTO artists VALUES ("A");
-- "A" already exists
```

(b) `INSERT` statements violating the schema's `PRIMARY KEY`

Fig. 1. Fragment of the MillionSong schema [11]

and effectiveness of mutation analysis, reducing both the computational and human costs of inspecting live mutants. The study also shows that ineffective mutant removal increases the mutation score, often revealing a mutation-adequate test suite.

To conclude, the contributions of this paper are as follows:

1) An abstract, DBMS-independent representation of relational database schemas that supports the accurate and efficient identification of ineffective mutants during the mutation analysis process (Section III);
2) Automated techniques for finding and removing the ineffective mutants of database schemas (Section IV);
3) Experiments that determine the efficiency improvement attributable to the use of the DBMS-independent representation and the impact that ineffective mutant removal has on both the execution time of mutation analysis and a schema's resultant mutation score (Section V).

II. Background

A. Relational Database Schemas

When creating a relational database, it is necessary to specify a schema, which defines its structure in terms of tables, columns and columnar data types. Optionally, the schema may also include further restrictions on what data can be added to the database, expressed as one or more integrity constraints.

There are five common types of constraints expressed in a schema [1]. `PRIMARY KEY` constraints ensure that the values in the given column(s) are unique, such that they individually identify each row. As only one `PRIMARY KEY` can be declared per table, `UNIQUE` constraints can also enforce additional row-uniqueness properties. A `NOT NULL` constraint specifies that a `NULL` value cannot be stored in a specific column. `FOREIGN KEY` constraints enforce that each row in one table must have a matching row in another table, connected according to the values in one or more corresponding pairs of columns. Lastly, `CHECK` constraints provide a means of defining arbitrary predicates that each row must satisfy to be accepted into the database. These may include boolean algebra operators, such as conjunction, disjunction and negation, as well as relational operators and database operations, such as '`x IS NULL`', '`x BETWEEN y AND z`' and '`x IN (y, ...)`'.

Figure 1a shows a schema fragment from the freely available MillionSong dataset [11], which contains 280GB of data. The fragment involves two tables, `artists` and `similarity`, and involves three integrity constraints: one `PRIMARY KEY` and two `FOREIGN KEY`s. The `PRIMARY KEY` on `artist_id` means that each `INSERT` statement adding a row to the database must contain a different value. Figure 1b demonstrates a sequence of statements, including one `INSERT` that would be rejected due to violating this constraint. The `FOREIGN KEY`s ensure that each `target` and `similar` value refers to an existing `artist_id` value in `artists`.

B. Testing and Mutating Schemas

When the integrity constraints of a schema are not specified correctly, erroneous data may be accepted into the database. For instance, if the `PRIMARY KEY` for the `artists` table of Figure 1a was accidentally omitted the DBMS would erroneously accept duplicate rows of data, causing inconsistency in the stored information. Alternatively, if a `NOT NULL` constraint was incorrectly added to the `target` attribute of the `similarity` table, then the DBMS would not accept the `NULL` values that are permitted for this attribute.

Testing the database schema is therefore a crucial quality assurance activity. Despite industrial practitioners who advocate it [2], this is a stage of testing that is often overlooked. Since a database schema is one of the first artifacts created during software development, mistakes made in its definition can have far-reaching effects that are costly to fix.

In previous work [8], we proposed techniques, implemented in a tool called *SchemaAnalyst*, that test database schemas by generating a test suite of `INSERT` statements. Using test data that will be accepted or rejected by the DBMS, the purpose of these `INSERT` statements is to satisfy and negate each of the schema's constraints, By running these tests and studying their results, missing or incorrectly specified schema constraints manifest as wrongly accepting or rejecting data.

The quality of these test suites can be evaluated using mutation analysis, a process that involves modifying the artifact-under-test in small ways to produce so-called mutants. Intuitively, the mutation analysis of a relational database schema works according to the following steps:

1) Produce mutants by applying mutation operators.
2) Execute `INSERT` statements with the original schema.
3) Execute `INSERT` statements with the mutant schemas.
4) Classify a mutant as *killed* if the results differ compared to the original, otherwise categorize it as *alive*.

As an example of mutation analysis for schemas, Figure 2a shows a mutant of the MillonSong schema of Figure 1a. A `NOT NULL` constraint is added to the `target` field of the `similarity` table. Figure 2b shows a test case involving `INSERT` statements that demonstrate differing behavior between the original schema and the mutant.

The more mutants a test suite kills, the more types of faults represented by the mutants it is able to kill, and the stronger the suite is deemed to be [6]. A test suite is assigned a mutation score, which is simply the number of mutants killed divided by the total number of mutants created from the original schema.

```
CREATE TABLE artists_M ( ... );
CREATE TABLE similarity_M (
 target text NOT NULL,
 similar text,
 FOREIGN KEY(target) REFERENCES artists_M(artist_id),
 FOREIGN KEY(similar) REFERENCES artists_M(artist_id)
);
```

(a) A mutant with an additional NOT NULL constraint.
(Mutant tables are suffixed with _M for ease of identification.)

```
INSERT INTO similarity VALUES (NULL,NULL);

INSERT INTO similarity_M VALUES (NULL,NULL);
-- NULL not allowed
```

(b) A test case that kills the mutant in Figure 2(a). NULL values are accepted for the original schema (first line), but the same INSERT statement is rejected for the mutated schema (highlighted second line).

Fig. 2. Example of a mutant for the MillionSong schema fragment of Figure 1

By applying mutation analysis, Kapfhammer et al. [8] found that *SchemaAnalyst*-generated test suites had a higher mutation score than data generated by the open-source *DBMonster* tool [12]. As with program mutation [6], [13], [14], however, the mutation analysis process is often very computationally expensive due to the large number of mutants that need to be analysed by executing, in the worst case, the full test suite for each mutant. Furthermore, mutation scores can be inaccurate due to certain types of mutants that have the effect of skewing the overall score; we now discuss these mutant types.

Equivalent Mutants. When applying mutation operators to create mutants, it is possible that some mutants may be functionally identical to the original artifact, despite their syntactic differences. This implies that no test case can differentiate between the original and the mutant, and thus the mutant cannot be killed. In the context of schema mutation, Figure 3 furnishes an example of an equivalent mutant involving the addition of UNIQUE constraint to a table where a PRIMARY KEY is defined for the same column. Since primary keys require the column to contain distinct values, the addition of the UNIQUE constraint has no additional effect. The mutant, therefore, behaves exactly the same as the original.

When equivalent mutants exist, the mutation score may be inaccurate [15], thus potentially compromising the comparison of different data generation techniques through mutation analysis. Equivalent mutants also have an associated human cost: following mutation analysis, testers often have to manually inspect test cases, mutants and the original schema to determine why a mutant is still alive. In the context of programs, where 45% of undetected mutants are equivalent, the manual study and classification of a mutant takes about fifteen minutes [16]. Since it is impossible to kill an equivalent mutant, such diagnostic effort is essentially wasted.

Combined with the execution cost per mutant, this makes the detection and discarding of these mutants, known as the equivalent mutant problem [6], an important issue for database schemas. Since the large number of equivalent mutants and the high costs of human inspection make it infeasible to manually detect equivalent mutants, there are numerous approaches that attempt to automatically detect them (e.g., [9], [16], [17]).

```
CREATE TABLE artists_M (
  artist_id text PRIMARY KEY UNIQUE
);
CREATE TABLE similarity_M ( ... );
```

Fig. 3. Example of a mutant equivalent to the original schema in Figure 1

```
CREATE TABLE artists_M ( ... );
CREATE TABLE similarity_M (
 ...
 FOREIGN KEY(target) REFERENCES artists_M(artist_id),
 FOREIGN KEY(similar) REFERENCES artists_M(artist_id)
);
```

Fig. 4. A mutation that may be produced by multiple operators, leading to redundant mutants

Prior to this paper, no methods existed for detecting the equivalent mutants of relational database schemas.

Redundant Mutants. In the context of program mutation, Just et al. describe a mutant as redundant if it is always subsumed by other mutants [10]. In this paper, we use the term more broadly: While equivalent mutants are semantically the same as the original artifact, we say that mutants are redundant if they are the same as one or more, already created, mutants. Figure 4 furnishes an example of a mutation that may be caused by more than one operator, thus leading to redundant mutants. The mutation could be produced by either removing the FOREIGN KEY directly or exchanging the similar column for target, thus overwriting the existing FOREIGN KEY constraint. These mutations would produce mutants that are equivalent to each other—and therefore redundant as well.

Quasi-Mutants. For traditional program mutation, a syntactically invalid mutant is classified as being still-born [18]. Still-born mutants can also be created for database schema mutation. However, in contrast to program mutation, it is possible that some schema mutants are still-born with respect to one DBMS but valid for another DBMS. This is due to inconsistencies between the relational database models—for example, PostgreSQL requires the columns of a FOREIGN KEY to be part of a UNIQUE or PRIMARY KEY constraint, while HyperSQL does not. These mutants are referred to as quasi-mutants [8]. Not removing quasi-mutants ahead of time precludes further optimization techniques for mutation analysis [7], thus decreasing the efficiency of this process.

Summary. This paper explains how these—fundamentally ineffective—equivalent, redundant and quasi-mutants can be identified and removed before mutation analysis, thus decreasing computational costs, saving human inspection time and improving the accuracy of mutation scores. The next section introduces the mutation operators for database schemas and the DBMS-independent representation used to create the mutants, thereby paving the way for the presentation of methods for removing equivalent, redundant and quasi-mutants in Section IV.

III. Mutant Generation with *SchemaAnalyst*

Figure 5 shows the different steps taken by *SchemaAnalyst* during mutant production, beginning with the parsing of SQL into an abstract, DBMS-independent schema representation, described in Section III-A. SQL parsing is performed using

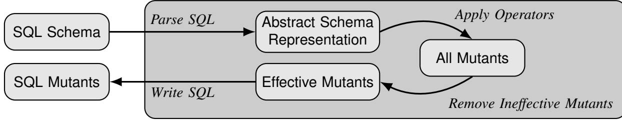

Fig. 5. The mutant generation pipeline

```
CREATE TABLE similarity_M1(
 target text PRIMARY_KEY,
 similar text,
 ...
);

CREATE TABLE similarity_M2(
 target text,
 similar text,
 ...
 PRIMARY_KEY(target)
);
```

```
CREATE TABLE similarity_M3(
 target text,
 similar text,
 ...
);
ALTER TABLE similarity_M3
ADD PRIMARY KEY(target);
```

Fig. 6. Three equivalent (syntactic) mutants, avoided by the use of the abstract representation of the relational database schema

the General SQL Parser (GSP)[1], a commercial tool handling SQL for a variety of DBMSs. After the schema is parsed into the abstract representation, mutation operators are applied to produce mutant schemas; the mutation operators presented in this paper are documented in Section III-B. Following this, the stage novel to this paper removes the ineffective equivalent, redundant and quasi mutants, as discussed in Section IV. Mutation analysis then begins, with mutants written to SQL through an SQL-writing process tailored to the DBMS in use. The SQL is then submitted to the DBMS and the test suite is run against it to determine how many mutants the tests killed.

A. Abstract Schema Representation

In contrast to program mutation, which makes small changes to program code at the syntax level, *SchemaAnalyst* applies mutation operators to an abstract representation of the relational schema. The representation is a model that abstracts over the various DBMS-specific dialects of SQL. Currently, *SchemaAnalyst* supports the oft-used SQLite, PostgreSQL and HyperSQL dialects. Along with modelling tables, columns and integrity constraints, the representation abstracts the plethora of data types made available by the different DBMSs down to seven key types (e.g., `boolean` and `string`) [8].

The use of an abstract representation allows the technique to mutate schemas at a semantic level, rather than a syntactic one. For instance, Figure 6 shows three different ways a table may be mutated to add a `PRIMARY KEY` to the same column, where all three mutants are equivalent to one another. Mutation at the semantic level avoids this issue. In the abstract model, adding a column to a `PRIMARY KEY` is one operation and the abstract representation of a schema is then written out as SQL for a specific DBMS. The abstract representation also enables a straightforward and effective static analysis of a schema to efficiently find patterns corresponding to the equivalent, redundant and quasi mutants, as explained in Section IV.

B. Mutation Operators

We now detail the mutation operators used in this paper. We apply ten mutation operators, originally described by

[1]General SQL Parser (GSP) is available at http://sqlparser.com

TABLE I
MUTATION OPERATORS
The first ten (above the dashed line) are originally due to Kapfhammer et al. [8], the remainder (below the dashed line) are new to this paper. The naming scheme follows a system according to the constraint type being mutated (e.g., Primary Key), the aspect being mutated (generally a column), and how the aspect is being mutated (i.e., Added, Removed or Exchanged with another).

Operator Name	Description
PKColumnA	Adds a column to a `PRIMARY KEY`
PKColumnR	Removes a column from a `PRIMARY KEY`
PKColumnE	Exchanges a column in a `PRIMARY KEY`
FKColumnPairR	Removes a column pair from a `FOREIGN KEY`
NNA	Adds a `NOT NULL` to a column
NNR	Removes a `NOT NULL` from a column
UColumnA	Adds a column to a `UNIQUE`
UColumnR	Removes a column from a `UNIQUE`
UColumnE	Exchanges a column in a `UNIQUE`
CR	Removes a `CHECK`
FKColumnPairA	Adds a column pair to a `FOREIGN KEY`
FKColumnPairE	Exchanges a column pair in a `FOREIGN KEY`
CInListElementR	Removes an element from an IN (...) of a `CHECK`
CRelOpE	Exchanges a relational operator in a `CHECK`

Kapfhammer et al. [8], along with four more introduced in this paper. The mutation operators, which are listed with brief descriptions in Table I, target schema constraints that guard the consistency of the data stored in the database.

For brevity and ease of identification, we assign each operator a name according to the constraint it affects and the modification it makes. For example, the "`PRIMARY KEY` Column Addition" operator is abbreviated to "PKColumnA". The "addition" and "removal" operators add and remove components, respectively, while the "exchange" operators swap some component for another. We refer the reader to [8] for an explanation of the first ten operators.

Focusing on `FOREIGN KEY` and `CHECK` constraints, our four additional operators address gaps in the initial ten operators. The first new operator, "CInListElementR", removes elements from the list of a '`CHECK COL IN (list)`' in turn. The second, "CRelOpE", replaces the relational operator ($=,<,>,\leq,\geq$) in a `CHECK` constraint with each other relational operator. The third operator, "FKColumnPairA", adds a pair of columns with matching data types to a `FOREIGN KEY` and the fourth, "FKColumnPairE", exchanges a pair of columns from a `FOREIGN KEY` with a pair that have the same data type.

Comparing all 14 operators to the "atomic changes" identified in a study of changes made between different versions of database schemas [19] confirms that the operators model realistic refactorings of integrity constraints. Therefore, when applied to a real-world schema the mutants generated are likely to reveal constraints omitted or misspecified by the developer.

IV. REMOVING INEFFECTIVE MUTANTS

As previously mentioned in Section II, the mutation operators can generate ineffective (i.e., equivalent, redundant and quasi) mutants that may decrease the accuracy of the mutation score and/or increase the time taken to perform mutation analysis. This section describes our techniques for automatically removing these ineffective mutants, thereby both decreasing analysis costs and improving mutation score accuracy. Integrated into *SchemaAnalyst*'s mutant generation pipeline, as shown in Figure 5, these techniques work at the level of the abstract representation, as described in Section III-A, thus greatly

```
function SCHEMAEQUIV(a, b)
    if NAME(a) ≠ NAME(b)  return F
    else if |TABLES(a)| ≠ |TABLES(b)|  return F
    else if ¬TABLESEQUIV(TABLES(a),TABLES(b))  return F
    else if ¬PKEYSEQUIV(PKEYS(a),PKEYS(b))  return F
    else if ¬FKEYSEQUIV(FKEYS(a),FKEYS(b))  return F
    else if ¬UNIQUESEQUIV(UNIQUES(a),UNIQUES(b))  return F
    else if ¬CHECKSEQUIV(CHECKS(a),CHECKS(b))  return F
    else return ¬NOTNULLSEQUIV(NOTNULLS(a),NOTNULLS(b))

function TABLESEQUIV(a, b)
    if NAME(a) ≠ NAME(b)  return F
    else return COLUMNSEQUIV(COLUMNS(a),COLUMNS(b))

function COLUMNSEQUIV(a, b)
    if NAME(a) ≠ NAME(b)  return F
    else return DATATYPE(a) = DATATYPE(b)

function PKEYSEQUIV(a, b)
    return MULTICOLUMNCONSTRAINTEQUIV(a,b)

function FKEYSEQUIV(a, b)
    if ¬MULTICOLUMNCONSTRAINTEQUIV(a,b)  return F
    else if NAME(REFTABLE(a)) ≠ NAME(REFTABLE(b))  return F
    else if |REFCOLUMNS(a)| ≠ |REFCOLUMNS(b)|  return F
    else return REFCOLUMNS(a) ⊆ REFCOLUMNS(b)

function UNIQUESEQUIV(a, b)
    return MULTICOLUMNCONSTRAINTEQUIV(a,b)

function CHECKSEQUIV(a, b)
    if NAME(a) ≠ NAME(b)  return F
    else if NAME(TABLE(a)) ≠ NAME(TABLE(b))  return F
    else return EXPRESSION(a) = EXPRESSION(b)

function NOTNULLSEQUIV(a, b)
    if NAME(a) ≠ NAME(b)  return F
    else if NAME(TABLE(a)) ≠ NAME(TABLE(b))  return F
    else return NAME(COLUMN(a)) = NAME(COLUMN(b))

function MULTICOLUMNCONSTRAINTEQUIV(a, b)
    if NAME(a) ≠ NAME(b)  return F
    else if NAME(TABLE(a)) ≠ NAME(TABLE(b))  return F
    else if |COLUMNS(a)| ≠ |COLUMNS(b)|  return F
    else return COLUMNS(a) ⊆ COLUMNS(b)
```

Fig. 7. Rules for detecting structural equivalence

simplifying the analysis that needs to be performed without losing the key information required to identify the ineffective mutants. Using this abstract representation therefore avoids the need for more computationally expensive techniques such as genetic algorithms [9], constraint-based testing [17] or coverage analysis [16]. In addition, the mutant generation process produces mutants directly in this abstract representation, facilitating the automatic, accurate, and efficient comparison of the original and mutated database schemas.

A. *Equivalent Mutants*

This section defines two types of equivalence: structural and behavioral. When *SchemaAnalyst* used detectors for these two kinds of equivalence, as described in Figures 7 through 9 and the remainder of this section, this led to the removal of 123 equivalent mutants during the mutation analysis of 16 schemas, as noted in the empirical study of Section V.

Structural Equivalence. Structural equivalence is where two schemas are identical except for possible syntactic differences in their SQL definition. Along with factoring away these syntactic differences, Figure 7 also reveals that the abstract representation makes it easy to define equivalence detection.

Behavioural Equivalence. Two schemas may not be structurally equivalent, but yet still have equivalent behavior, due to functional overlap between different SQL expressions and operators (which may pertain to certain DBMSs only). We describe three patterns of behavioural equivalence, with Figure 8 furnishing functions for detecting them in schemas. These patterns were discovered during a thorough study of over 60 schemas running on HyperSQL, PostgreSQL and SQLite. While additional patterns may also exist for other DBMSs and new schemas, we judge these to be both representative and powerful and we further note that it is easy to integrate new patterns into *SchemaAnalyst*'s equivalence detectors.

```
function NOTNULLEQUIVTOCHECK(a, b)
    unmatched ← ∅
    for all notNullA in NOTNULLS(a) do
        found ← F
        for all notNullB in NOTNULLS(b) do
            if notNullA = notNullB
                found ← T
                break
        if found = F
            unmatched ←unmatched ∪ {notNullA}
    if |unmatched| = 0  return T
    else
        for all notNull in unmatched do
            if ¬EXISTSCHECK(notNull, b)  return F
    return T

function NOTNULLONPRIMARYKEY(schema)
    for all notNull in NOTNULLS(schema) do
        if EXISTSPRIMARYKEY(COLUMN(notNull))
            REMOVE(notNull)

function UNIQUEONPRIMARYKEY(schema)
    for all unique in UNIQUES(schema) do
        if EXISTSPRIMARYKEY(COLUMNS(unique))
            REMOVE(unique)
```

Fig. 8. Rules for detecting patterns of behavioral equivalence

Pattern 1: NOT NULLs in CHECK constraints. Defining a column as `NOT NULL` is behaviorally equivalent to defining a `CHECK(...IS NOT NULL)`. This can be implemented in the SCHEMAEQUIV function of Figure 7 by looking for an equivalent `CHECK(...IS NOT NULL)` for each `NOT NULL` that doesn't have a matching `NOT NULL` constraint, and vice versa. If no such constraint exists, then the schemas are not equivalent. The first function in Figure 8, NOTNULLEQUIVTOCHECK, shows the algorithm used for this check.

Pattern 2: NOT NULLs in PRIMARY KEY constraints. When using PostgreSQL or HyperSQL, the columns of a `PRIMARY KEY` implicitly cannot be `NULL`. This differs from the SQLite DBMS, which requires an additional `NOT NULL` constraint to match this behaviour. Consequently, when using either PostgreSQL or HyperSQL, a mutation operator adding a `NOT NULL` to the `artist_id` column in Figure 1 would produce a behaviourally equivalent mutant. This means there exists no input that would be accepted by the table in the original and rejected by the table in the mutant, or vice versa.

The second function of Figure 8, NOTNULLONPRIMARYKEY, shows the detection function for this pattern. Applied to each mutant in turn, it identifies the matching mutants so that they can be modified, in the case of either PostgreSQL or HyperSQL, by removing any extraneous `NOT NULL` constraints. Removing these `NOT NULL` constraint(s) later causes the structural equivalence detection function, as given in Figure 7, to discard this type of ineffective mutant.

Pattern 3: UNIQUEs and PRIMARY KEYs with shared column sets. Finally, as primary keys can be considered a stricter form of the `UNIQUE` constraint, a mutant adding a `UNIQUE`

```
function DETECTQUASI(schema)
    for all fk in FKEYS(schema) do
        if ¬EXISTPRIMARYKEY(schema,COLUMNS(fk))
            if ¬EXISTUNIQUE(schema,COLUMNS(fk))  return F
    return T
function EXISTPRIMARYKEY(schema, columns)
    for all pk in PKEYS(schema) do
        if columns = COLUMNS(pk)  return T
    return F
function EXISTUNIQUE(schema, columns)
    for all uc in UNIQUES(schema) do
        if columns = COLUMNS(uc)  return T
    return F
```

Fig. 9. Rules for detecting quasi-mutants

constraint to a primary key column will be behaviourally equivalent to the original. Tables `artists` and `artists_m2` from Figures 1a and 3 show an example of two schemas that are equivalent according to this pattern. Using pattern three in a detection rule will drop the `UNIQUE` constraint in the `artists_M` mutant of Figure 3, thus ensuring that it will be identified as directly equivalent and discarded. Figure 8 reveals the implementation of this function, UNIQUEONPRIMARYKEY.

B. Redundant Mutants

While equivalent mutants can be discarded for being structurally or behaviourally equivalent to the original, non-mutant schema, this paper's methods also handle the problem of redundant mutants, which Section II-B defined as mutants that are equivalent to one or more other mutants. To both reduce the cost of mutation analysis and increase the accuracy of the resulting mutation score, a mutant is discarded if it is redundant with respect to an already generated mutant.

Our approach to redundant mutant detection leverages the technique for equivalent mutant detection, as previously described in Section IV-A. Instead of checking each mutant for equivalence with respect to the original schema, the same check is effectively applied between every pair of mutants. Adding redundant mutant removal to *SchemaAnalyst* led to the discarding of eight mutants during the mutation analysis of 16 schemas, as presented in the empirical study of Section V.

C. Quasi-Mutants

As described in Section II-B, quasi-mutants will be rejected by at least one DBMS and may negatively influence the efficiency of mutation analysis; although, unlike equivalent and redundant mutants, they should only be removed when using a DBMS that will reject them.

This paper focuses on one major source of quasi-mutants that pertains to the PostgreSQL and HyperSQL DBMSs. If a `FOREIGN KEY` is defined on a table, the columns in the referenced table must have a `PRIMARY KEY` or a `UNIQUE` constraint defined on them as well. Otherwise, these DBMSs will reject the schema. Even though the SQLite DBMS would accept a `CREATE TABLE` statement not adhering to this rule, PostgreSQL will reject it with the error message:

"There is no unique constraint matching given keys for referenced table (Error 42830 (invalid_foreign_key))"

...while HyperSQL produces the error message:

"a UNIQUE constraint does not exist on referenced columns (Error 5529)"

TABLE II
SCHEMAS ANALYSED IN THE EMPIRICAL STUDY

Schema	*Tables*	*Columns*	*Checks*	*Foreign Keys*	*Not Nulls*	*Primary Keys*	*Uniques*	$\sum$ *Constraints*
ArtistSimilarity	2	3	0	2	0	1	0	3
ArtistTerm	5	7	0	4	0	3	0	7
BankAccount	2	9	0	1	5	2	0	8
BookTown	22	67	2	0	15	11	0	28
Cloc	2	10	0	0	0	0	0	0
CoffeeOrders	5	20	0	4	10	5	0	19
Flights	2	13	1	1	6	2	0	10
IsoFlav_R2	6	40	0	0	0	0	5	5
JWhoisServer	6	49	0	0	44	6	0	50
NistDML183	2	6	0	1	0	0	1	2
NistWeather	2	9	5	1	5	2	0	13
NistXTS749	2	7	1	1	3	2	0	7
RiskIt	13	57	0	10	15	11	0	36
StackOverflow	4	43	0	0	5	0	0	5
UnixUsage	8	32	0	7	10	7	0	24
WordNet	8	29	0	0	22	8	1	31
Total	91	401	9	32	140	60	7	248

Although a quasi-mutant schema can be identified by submitting it to the DBMS as a series of `CREATE TABLE` statements and then checking for DBMS rejection, this can be very costly due to the number of tables and mutants combined with the time taken to test each using the database. In addition, this approach must run `DROP TABLE` statements to clear the database between each mutant. An alternative to the individual submission of SQL commands is to surround all of the `CREATE TABLE` statements of each mutant in an SQL transaction, leveraging a database's "roll back" feature to remove the tables in the event of DBMS rejection, thereby preparing for the next mutant. Avoiding DBMS interaction altogether, we introduce the discarding of quasi-mutants in the *SchemaAnalyst* mutant generation pipeline using a detection function, invoked when using an applicable DBMS. This function, shown as DETECTQUASI in Figure 9, is used to statically analyse each mutant and discard any not satisfying the conditions. These three techniques (i.e., direct DBMS interaction with and without transactions and static quasi-mutant detection) are evaluated and compared in the empirical study of Section V, revealing the detection of 567 quasi-mutants across 16 schemas.

Interestingly, the existence of quasi-mutants demonstrates that mutation analysis for relational schemas cannot be viewed as a DBMS-independent process. It is also important to note that, while additional kinds of quasi-mutant may exist for other types of DBMSs, the experimental results in Section V show that our current emphasis on one type of representative quasi-mutant yields a substantial efficiency improvement. Moreover, the use of the DBMS-independent abstract model makes it is easy to add detectors for new types of quasi-mutants.

V. EMPIRICAL STUDY

A. Experimental Setup

We selected 16 schemas as case studies to use in our experiments, aiming for a representative range in the number of tables (2 to 22), columns (3 to 67) and constraints (0 to 50), as detailed in Table II. This assortment of properties helps to ensure the results are broadly applicable to many different schemas. Of the chosen schemas, 10 were taken from real-world sources. For instance, JWhoisServer and

IsoFlav_R2 belong to a WHOIS server application and a plant compound database from the U.S. Department of Agriculture, respectively. Other examples include UnixUsage, from a Unix command history monitoring application, and WordNet, a schema used in a visualiser for the WordNet lexical database.

Since the choice of DBMS may affect the results, experiments were performed using both PostgreSQL and HyperSQL, DBMSs chosen for their performance differences and varying design goals. PostgreSQL is a full-featured, extensible and highly scalable DBMS, while HyperSQL is a small, lightweight DBMS supporting an "in-memory" mode that avoids disk writing; we enabled this HyperSQL feature to ensure that the two DBMSs have different performance profiles. Although *SchemaAnalyst* also supports SQLite, this paper does not report on results with this DBMS due to space constraints and the fact that it leads to empirical trends like HyperSQL's.

We performed the experiments with the *SchemaAnalyst* tool [8], compiled with the Java Development Kit 7 compiler and executed with the Linux version of the 64-bit Oracle Java 1.7 virtual machine. Experiments were executed on an Ubuntu 12.04 workstation, with a 3.2.0-27 64-bit Linux kernel, a quad-core 2.4GHz CPU and 12GB of RAM. All input (i.e., schemas) and output (i.e., results files) were stored on the local disk. We used the default configuration of PostgreSQL version 9.1.9 and HyperSQL version 2.2.8 running in the "in-memory" mode.

Threats to Validity. Since background processes may lead to small differences in the timings of results, we ran 15 repeat trials per schema for each quasi-mutant detection approach and each mutation analysis task. This allowed for means and standard deviations to be calculated, statistical tests to be performed, and for the results to be visualised with box plots, providing further confidence that the results are accurate and representative. Finally, we controlled the threats arising from defects in *SchemaAnalyst* by both carefully testing the tool and manually checking results on simple schemas. For instance, we verified that *SchemaAnalyst* correctly applied the right mutation operators, removed the anticipated ineffective mutants and correctly calculated the mutation score.

Quasi-mutant Detection Metrics. To evaluate our static-analysis approach to quasi-mutant detection, we compared it to the simple and SQL transaction-optimised techniques described in Section IV-C. We measured the number of quasi-mutants detected by the three methods for each of the 16 schemas to determine their effectiveness, with the technique that performs simple DBMS interactions acting as the "gold standard". To compare efficiency, we recorded the time taken by each approach to identify the quasi-mutants.

Ineffective Mutant Removal Metrics. To determine the effect of removing equivalent, redundant and quasi-mutants (using the static analysis approaches described in Sections IV-A, IV-B and IV-C, respectively) we compared mutation analysis performed with and without ineffective mutant removal.

To measure the efficacy of ineffective mutant removal, we determined the number of each type of mutant detected per schema, allowing us to calculate the proportion of mutants discarded by Figure 5's mutant generation pipeline. We classified the influence on the mutation score in terms of increase, decrease or none, and tested for statistical significance using the Wilcoxon Rank-Sum test, where a value <0.05 is significant.

TABLE III
INEFFECTIVE MUTANTS REMOVED

Number of equivalent, redundant and quasi-mutants that were automatically removed. As all quasi-mutant detection techniques identified the same number of mutants, the reported values are representative regardless of the chosen method. Savings is the percentage of mutants that have been removed; total savings is calculated in a row-wise fashion.

(A) GROUPED BY SCHEMA

Schema	Produced	Equivalent	Redundant	Quasi-Mutant	∑Ineffective	∑Effective	Savings (%)
ArtistSimilarity	13	2	2	1	5	8	38.5
ArtistTerm	29	6	0	3	9	20	31.0
BankAccount	51	4	0	21	25	26	49.0
BookTown	235	22	0	0	22	213	9.4
Cloc	30	0	0	0	0	30	0.0
CoffeeOrders	115	10	0	54	64	51	55.7
Flights	70	4	3	19	26	44	37.1
IsoFlav_R2	219	0	0	0	0	219	0.0
JWhoisServer	190	12	0	0	12	178	6.3
NistDML183	40	0	2	18	20	20	50.0
NistWeather	58	3	0	23	26	32	44.8
NistXTS749	33	4	0	7	11	22	33.3
RiskIt	503	22	0	297	319	184	63.4
StackOverflow	129	0	0	0	0	129	0.0
UnixUsage	220	14	0	124	138	82	62.7
WordNet	107	20	1	0	21	86	19.6
Total	2042	123	8	567	698	1344	34.2

(B) GROUPED BY OPERATOR

Operator	Produced	Equivalent	Redundant	Quasi-Mutant	∑Ineffective	∑Effective	Savings (%)
CInListElementR	4	0	0	0	0	4	0.0
CR	9	0	0	0	0	9	0.0
CRelOpE	10	0	0	0	0	10	0.0
FKColumnPairA	188	0	0	188	188	0	100.0
FKColumnPairE	287	0	5	222	227	60	79.1
FKColumnPairR	34	0	2	4	6	28	17.6
NNA	261	13	0	0	13	248	5.0
NNR	140	60	0	0	60	80	42.9
PKColumnA	327	0	0	61	61	266	18.7
PKColumnE	201	0	0	66	66	135	32.8
PKColumnR	74	0	0	21	21	53	28.4
UColumnA	427	50	0	1	51	376	11.9
UColumnE	68	0	0	2	2	66	2.9
UColumnR	12	0	1	2	3	9	25.0
Total	2042	123	8	567	698	1344	34.2

We gauged the efficiency implications of ineffective mutant removal by comparing the time taken for mutation analysis with and without ineffective mutants, including the time taken for static analysis in the latter case. We tested the change for significance using the Wilcoxon Rank-Sum test and the Vargha-Delaney $\hat{A}_{12}$ effect size measure [20], where the difference is categorised as "large" if <0.29 or >0.71, "medium" if <0.36 or >0.64 and "small" if <0.44 or >0.56. Otherwise, the effect size is "none". An effect size >0.5 represents a time decrease, while <0.5 represents a time increase.

B. Empirical Results

Summary of Results. Table IIIa shows the number of mutants generated for each schema; how many equivalent, redundant and quasi-mutants were discarded as ineffective; and the number of mutants remaining (retained). Table IIIb presents the same data, but instead grouped by mutation operator. For both, the savings value is the percentage of mutants removed

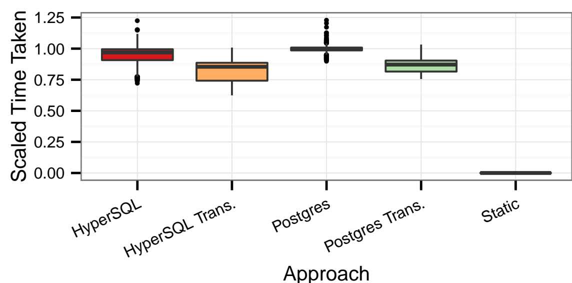

Fig. 10. Quasi-mutant Detection – Time Taken

The results for each schema were scaled according to the maximum mean from any approach for that schema, thereby allowing them to be combined without being affected by scale differences attributed to the wide range of times taken.

for each schema. The total rows provides column-wise totals except the savings, which we calculated across the total rows themselves according to the following equation:

$$\text{Total Savings} = (1 - \sum \text{Effective} / \sum \text{Produced}) \times 100$$

The results in Table IV detail the mean and standard deviation of the time taken to detect the quasi-mutants of each schema for the three techniques. Since the time taken for "Static" is the same for both PostgreSQL and HyperSQL, it only appears in the table once. As we observed that the number of quasi-mutants detected by each technique was consistent for every case study and repeat trial, we omit these results.

Figure 10 shows the results from Table IV, with the values for each schema scaled according to the maximum mean from any approach. This allows them to be combined without being affected by the range of times taken caused by the differences in the schemas (e.g., the number of tables and constraints).

Table V shows the time taken for mutation analysis with and without ineffective mutants, including the cost of static analysis in the latter case. We summarise these values in Table VI. The results in Table VII show the significance and effect size for changes in the time overhead of mutation analysis. Finally, we classify the change in mutation score caused by ineffective mutant removal by direction of change and report the significance of this change in Table VIII.

Research Questions.

RQ1: *Is the detection of quasi-mutants using static analysis more efficient than the DBMS-based approaches?*

As shown in Table IV, the static analysis method for detecting quasi-mutants is markedly quicker than the DBMS-based approaches—regardless of whether the transaction-based optimisation is used—across all schemas and both DBMSs. Figure 10 shows the time taken across all schemas in a scaled format. This graph reveals that, while using the transaction-based optimisation improves the performance of quasi-mutant detection for both HyperSQL and PostgreSQL, the time taken by the static analysis approach is still substantially lower than the others. Additionally, although not included in the table due to space constraints, each technique correctly identifies the same number of quasi-mutants, as reported in Table III. We therefore conclude that the static analysis approach represents a vast efficiency improvement, reducing the time taken by multiple orders of magnitude, while maintaining an identical effectiveness when compared to the "gold standard" approach.

RQ2: *How does ineffective mutant removal affect the time taken for mutation analysis, and is it cost-effective?*

As evidenced by the results in Tables V, VI and VII, the time efficiency of removing ineffective mutants varies significantly depending on both the schema and the DBMS.

When using HyperSQL, the mean time difference ranges from -824 to 718ms and the mean percentage difference from -9.71 to 23.05%, with mean values across all schemas of 7.5ms and 1.6%, respectively. For 9 out of 16 schemas, removal of ineffective mutants produces a mean time saving, all of which are statistically significant (p-value $<$0.05) and have a "large" effect size. For all of the 6 schemas where the time taken increases, the changes are also significant and have a "large" effect size. In summary, while the impact of ineffective mutant removal is not consistent across schemas when using HyperSQL, it provides a small time saving on average (mean 1.6%, median 1.4%), and produces a statistically significant decrease in time taken for the majority of schemas (56%).

When using PostgreSQL, the mean time difference ranges from -3,086 to 317,208ms and the mean percentage difference from -0.33 to 33.71%, with mean values across all schemas of 50,880ms and 12.7%, respectively. Of the 16 schemas, removal of ineffective mutants produces a mean time saving in 14 cases, where the difference is significant and has a "large" effect size in 13 cases. In the two cases where the mean time taken increases, both are statistically significant, with "medium" and "large" effect sizes. However, as a proportion of the overall time taken, these only represent increases of 0.33% and 0.30%, or approximately 1.7 and 3.1 seconds in real terms. To summarise, the impact of ineffective mutant removal when using PostgreSQL is a statistically significant decrease in time taken for 81% of the chosen schemas, reducing the time taken for mutation analysis by over 5 minutes in the best case.

In conclusion, although the results differ according to the DBMS used, in both cases ineffective mutant removal provides savings on average, with mean values of 1.6% and 11.8% for HyperSQL and PostgreSQL, respectively. We propose that this difference is caused by the varying performance of HyperSQL and PostgreSQL. As HyperSQL is much faster than PostgreSQL, the cost of static analysis forms a larger proportion of the overall time taken; for those schemas with few or no ineffective mutants, this causes a statistically significant increase in time overhead. Yet, given the relative efficiency of mutation analysis using HyperSQL, an in-memory DBMS, we argue that this difference is not of practical significance. Thus, our results show that ineffective mutant removal is cost-effective, yielding a practically significant reduction in the duration of mutation analysis, especially when using PostgreSQL.

RQ3: *Does removal of ineffective mutants significantly affect the mutation score?*

The results in Table VIII show that, regardless of the DBMS used, for 75% of the chosen schemas the mutation score increases when removing ineffective mutants—and in no case

TABLE IV
MEAN AND STANDARD DEVIATION OF TIME TAKEN TO DETECT QUASI-MUTANTS

Schema	Time Taken (ms)									
	HyperSQL		HyperSQL Trans.		PostgreSQL		PostgreSQL Trans.		Static	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
ArtistSimilarity	2203	117	2057	148	2811	154	2562	101	<1	<1
ArtistTerm	14852	372	13373	307	14889	331	13377	364	3	<1
BankAccount	7248	244	6362	136	7372	278	6360	177	2	<1
BookTown	227657	1934	177102	1012	238193	1495	185695	1135	<1	<1
Cloc	2361	177	2114	130	2321	145	2052	112	<1	<1
CoffeeOrders	39086	635	32108	590	39071	785	32062	561	11	<1
Flights	10113	251	9059	272	10480	211	9355	192	2	<1
IsoFlav_R2	151500	4475	137992	961	150570	1225	138145	970	<1	<1
JWhoisServer	107537	1187	96748	908	110345	1144	100450	801	<1	<1
NistDML183	3511	145	2957	113	4143	306	3445	198	2	<1
NistWeather	8643	385	7483	313	8536	198	7450	253	2	<1
NistXTS749	6633	206	6138	231	6676	213	6064	201	1	<1
RiskIt	258236	1754	210010	696	334288	1874	269025	1217	57	3
StackOverflow	34724	682	30633	510	34843	625	30796	726	<1	<1
UnixUsage	82574	1186	65771	756	102872	1341	79872	586	21	<1
WordNet	74003	852	65714	891	77998	1153	68160	529	<1	<1

TABLE V
MUTATION ANALYSIS TIME WITH AND WITHOUT INEFFECTIVE MUTANTS

Schema	HyperSQL				PostgreSQL			
	Mean Time (ms)		Mean Diff.		Mean Time (ms)		Mean Diff.	
	Without	With	(ms)	(%)	Without	With	(ms)	(%)
ArtistSimilarity	185	240	55	23.05	5693	8588	2895	33.71
ArtistTerm	1178	1302	124	9.54	42668	55905	13238	23.68
BankAccount	1025	1071	47	4.35	30741	35458	4717	13.30
BookTown	26283	27001	718	2.66	1710434	1908186	197752	10.36
Cloc	1100	1027	-73	-7.15	28797	28838	41	0.14
CoffeeOrders	2979	3016	37	1.22	217683	257678	39995	15.52
Flights	2187	2161	-27	-1.24	79073	90303	11230	12.44
IsoFlav_R2	9316	8492	-824	-9.71	1032813	1029727	-3086	-0.30
JWhoisServer	9401	9104	-297	-3.26	646683	686089	39407	5.74
NistDML183	755	737	-17	-2.37	23262	24378	1116	4.58
NistWeather	1546	1541	-4	-0.29	44922	48610	3688	7.59
NistXTS749	892	928	36	3.83	26214	31125	4911	15.78
RiskIt	17555	17831	276	1.55	2117462	2434670	317208	13.03
StackOverflow	6415	6007	-408	-6.79	515815	514126	-1689	-0.33
UnixUsage	5661	5955	294	4.93	885947	1042460	156513	15.01
WordNet	3668	3851	183	4.76	115310	141511	26201	18.51

TABLE VI
SUMMARY OF TIME SAVED BY REMOVING INEFFECTIVE MUTANTS

DBMS	Time Difference (ms)		Time Difference (%)	
	Median	Mean	Median	Mean
HyperSQL	36.2	7.5	1.4	1.6
Postgres	8071.0	50880.0	12.7	11.8
Both	229.9	25450.0	4.7	6.7

did the score decrease. Practically, this means that following mutation analysis there are fewer mutants remaining that would require human inspection, reducing the effort required to produce a mutation-adequate test suite. In 44% of the cases where the score changed, the mutation score increased to 1, meaning the test suite was in fact mutation adequate—and thus required no manual inspection—but would be misclassified without the use of ineffective mutant removal. The change in mutation score across all schemas is also shown to be statistically significant (p-value <0.05) in Table VIII.

VI. RELATED WORK

Experimentally observing that the schema of the database in real-world applications changes frequently, Qui et al. both demonstrate the important role that the relational database schema plays in ensuring the correctness of an application and motivate the need for extensive schema testing [19]. The empirical results of Qui et al. are amplified by Guz's remark that one of the key mistakes in testing database applications is "not testing [the] database schema" [2]. Moreover, the mutation operators employed by the methods in this paper are similar to the real-world schema faults found by Qui et al.

While we focus on mutating the `CREATE TABLE` statements that produce the schema, previous work has proposed and evaluated mutation operators for the SQL `SELECT` statements used by applications to retrieve data stored in a database [21]. This was later incorporated into a tool for instrumenting and testing database applications written in the Java programming language, potentially mutating any executed `SELECT` statement [22]. Chan et al. propose some operators for mutating schemas [23]; however, unlike this paper, they provide neither an implementation nor an evaluation. Even though practitioners emphasise the importance of schema testing [2], they neither recommend specific approaches nor suggest a means of comparing different techniques, in contrast to this paper.

TABLE VII
MUTATION ANALYSIS TIME – SIGNIFICANCE

The significance of the change in the time taken for mutation analysis time when removing ineffective mutatants, calculated with the Wilcoxon Rank-Sum test and effect size calculated using the Vargha-Delaney $\hat{A}_{12}$ measure.

Schema	HyperSQL		PostgreSQL	
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
ArtistSimilarity	<0.01	1.00 (large)	<0.01	1.00 (large)
ArtistTerm	<0.01	1.00 (large)	<0.01	1.00 (large)
BankAccount	<0.01	1.00 (large)	<0.01	1.00 (large)
BookTown	<0.01	0.87 (large)	<0.01	1.00 (large)
Cloc	<0.01	0.00 (large)	0.61	0.54 (none)
CoffeeOrders	<0.01	0.75 (large)	<0.01	0.97 (large)
Flights	<0.01	0.15 (large)	<0.01	1.00 (large)
IsoFlav_R2	<0.01	0.00 (large)	<0.01	0.28 (large)
JWhoisServer	<0.01	0.01 (large)	<0.01	0.97 (large)
NistDML183	<0.01	0.00 (large)	<0.01	0.96 (large)
NistWeather	<0.01	0.25 (large)	<0.01	1.00 (large)
NistXTS749	<0.01	0.98 (large)	<0.01	1.00 (large)
RiskIt	<0.01	0.84 (large)	<0.01	1.00 (large)
StackOverflow	<0.01	0.00 (large)	0.01	0.30 (med)
UnixUsage	<0.01	1.00 (large)	<0.01	1.00 (large)
WordNet	<0.01	1.00 (large)	<0.01	1.00 (large)

TABLE VIII
MUTATION SCORE – CHANGE SUMMARY

The summary of how mutation scores change when removing ineffective mutants. Increase, decrease and no change are the percentage of scores that increased, decreased or did not change, respectively. Adequate is the percentage of scores that changed from a value <1 to 1 (mutation adequate). The significance of the change in mutation score when removing ineffective mutants is calculated with the Wilcoxon Rank-Sum test.

DBMS	Increase (%)	Decrease (%)	No Change (%)	Adequate (%)	Significance
HyperSQL	75	0	25	44	0.001
PostgreSQL	75	0	25	44	0.001
Both	75	0	25	44	<0.001

In the context of using mutation analysis to compare test data, the detection of equivalent program mutants is considered to be generally undecidable [24]. Approaches using genetic algorithms [9], constraint-based testing [17] and coverage analysis [16] have been applied to detect some equivalent mutants. In addition, Hierons et al. explained how to use program slicing to reduce the effort needed to determine if a mutant is equivalent [25]. Applying it to the equivalent mutant problem, Hierons and Merayo have also presented an algorithm for detecting equivalence between pairs of probabilistic stochastic finite state machines [26]. Yet, while these methods may be adapted for databases, none of them currently handle relational database schema mutants. Finally, the term redundant mutant was previously used by Just et al. [10] to describe mutants that should be removed because they are always subsumed by other mutants. We use this term more generally, encompassing all of the mutants that are equivalent to other mutants.

VII. Conclusions and Future Work

While mutation analysis can assess the quality of a test suite, mutation operators may generate ineffective (i.e., equivalent, redundant and quasi) mutants that both reduce the accuracy of the resulting mutation score and increase the associated computational and human costs. This paper presents methods that statically analyse a DBMS-independent abstract representation of a relational schema to identify and remove these mutants. To handle equivalent and redundant mutants, the presented technique looks for structural and behavioral equivalence patterns in the abstract representation. Removing quasi-mutants also involves static analysis of the representation to find foreign key constraints rejected by certain DBMSs.

Using 16 relational schemas, the two-phase empirical study highlighted the positive impact of removing ineffective mutants. The first phase showed that detecting quasi-mutants using static analysis, rather than direct interaction with the DBMS, was many orders of magnitudes faster for both HyperSQL and PostgreSQL, with savings of over 5 minutes in the best case. The second phase revealed that ineffective mutant removal is beneficial for the majority of schemas, with a mean savings of 1.6% and 11.8% for HyperSQL and PostgreSQL, respectively. In addition, removing ineffective mutants improves the accuracy of mutation analysis, increasing the mutation score in 75% of the cases, and revealing a mutation-adequate test suite in 44% of those cases, thus obviating the need for human inspection of the live mutants.

Aiming to further improve the efficiency and accuracy of mutation anlaysis, we intend, as part of future work, to investigate whether additional mutant equivalence patterns exist. In an effort to identify additional types of quasi mutants, we will also continue to study how different DBMSs implement the structured query language. Since direct DBMS interaction is useful for validating the presented static analyses, we will also explore optimisations for quasi-mutant detection that involve submitting only the mutated section of the schema. Using more relational schemas, we also plan to conduct additional experimental studies. Finally, we will explore the use of selective mutation to determine which operators may be omitted, thereby reducing the number of mutants to analyse without adversely affecting the accuracy of the mutation score. Overall, the combination of this paper's ineffective mutant removal methods and the improvements completed during future work will yield an accurate and efficient way to assess the quality of relational schema test suites through mutation analysis.

References

[1] G. M. Kapfhammer, "A comprehensive framework for testing database-centric applications," Ph.D. dissertation, University of Pittsburgh, 2007.

[2] S. Guz, "Basic mistakes in database testing," http://java.dzone.com/articles/basic-mistakes-database, (Accessed 24/01/2014).

[3] C. Binnig, D. Kossmann, and E. Lo, "Multi-RQP: Generating test databases for the functional testing of OLTP applications," in *Proc. of TDS*, 2008.

[4] N. Bruno and S. Chaudhuri, "Flexible database generators," in *Proc. of VLDB 2005*.

[5] 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.

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

[7] C. J. Wright, G. M. Kapfhammer, and P. McMinn, "Efficient mutation analysis of relational database structure using mutant schemata and parallelisation," in *Proc. of Mutation*, 2013.

[8] 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.

[9] K. Adamopoulos, M. Harman, and R. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Proc. of GECCO*, 2004.

[10] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *Proc. of Mutation*, 2012.

[11] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *Proc. of ISMIR*, 2011.

[12] A. Ominiyi, "Playing with a DBMonster," http://java.dzone.com/tips/playing-dbmonster, (Accessed 24/01/2014).

[13] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using conditional mutation to increase the efficiency of mutation analysis," in *Proc. of AST*, 2011.

[14] 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.

[15] B. Grun, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *Proc. of Mutation*, 2009.

[16] D. Schuler and A. Zeller, "(Un-)covering equivalent mutants," in *Proc. of ICST*, 2010.

[17] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *JSTVR*, vol. 7, no. 3, 1997.

[18] 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.

[19] D. Qiu, B. Li, and Z. Su, "An empirical analysis of the co-evolution of schema and code in database applications," in *Proc. of FSE*, 2013.

[20] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Jour. of Educ. and Behav. Stat.*, vol. 25, no. 2, 2000.

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

[22] C. Zhou and P. Frankl, "JDAMA: Java database application mutation analyser," *JSTVR*, vol. 21, no. 3, 2011.

[23] 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.

[24] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Inform.*, vol. 18, no. 1, 1982.

[25] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *JSTVR*, vol. 9, no. 4, 1999.

[26] R. M. Hierons and M. G. Merayo, "Mutation testing from probabilistic and stochastic finite state machines," *JSS*, vol. 82, no. 11, 2009.