Automatic Detection and Removal of Ineffective Mutants for the Mutation Analysis of Relational Database Schemas

Phil McMinn, Chris J. Wright, Colton J. McCurdy, and Gregory M. Kapfhammer

Abstract—Data is one of an organization's most valuable and strategic assets. Testing the relational database schema, which protects the integrity of this data, is of paramount importance. Mutation analysis is a means of estimating the fault-finding "strength" of a test suite. As with program mutation, however, relational database schema mutation results in many "ineffective" mutants that both degrade test suite quality estimates and make mutation analysis more time consuming. This paper presents a taxonomy of ineffective mutants for relational database schemas, summarizing the root causes of ineffectiveness with a series of key patterns evident in database schemas. On the basis of these, we introduce algorithms that automatically detect and remove ineffective mutants. In an experimental study involving the mutation analysis of 34 schemas used with three popular relational database management systems—HyperSQL, PostgreSQL, and SQLite—the results show that our algorithms can identify and discard large numbers of ineffective mutants that can account for up to 24% of mutants, leading to a change in mutation score for 33 out of 34 schemas. The tests for seven schemas were found to achieve 100% scores, indicating that they were capable of detecting and killing all non-equivalent mutants. The results also reveal that the execution cost of mutation analysis may be significantly reduced, especially with "heavyweight" DBMSs like PostgreSQL.

✦

1 INTRODUCTION

For many different organizations, including large multi-national firms such as Google and Facebook, data forms a strategic asset that must be carefully curated and protected [1]. Indeed, fields such as healthcare, science, and commerce often rely on information that is stored in databases [2]. While non-relational "NoSQL" systems have been gaining in popularity, relational databases remain pervasive. For instance, Skype, the widely used video-call software, uses the PostgreSQL database management system (DBMS) [3] while Google makes use of the SQLite DBMS in Android-based phones [4]. Moreover, relational databases form the backbone of Internet web browsers such as Chrome[1] and Firefox[2], mobile applications [5], and even software powering political campaigns [6].

According to DB-Engines.com, the three most popular storage systems are relational in nature [7]. Another way to gauge the popularity of data management technologies is through an analysis of the tags assigned to questions posted on the popular Stack Overflow question and answer web site [8]. Examining the tags attached to the questions posted to Stack Overflow from January 2008 to August 2016 reveals that, while those about relational databases (e.g., "SQL") are attached to between one and three percent of all questions on Stack Overflow, only one tag about NoSQL (i.e., "MongoDB") is assigned to more than half a percent of questions. Indeed, the sum of the percentages for the top tags about relational databases (e.g., "Database", "PostgreSQL", and "SQLite") are connected with nearly nine percent of all questions posted during the studied period. In contrast, the NoSQL tags (e.g., "Cassandra", "HBase", and

1. https://www.google.com/chrome/browser
2. http://www.mozilla.org/firefox

"CouchDB") are attached to less than one percent of Stack Overflow's questions. These results clearly indicate that relational databases, and their schemas that are the subject of this paper, are a technology that practicing programmers and database administrators frequently use and discuss.

In addition to being favored because their schema clearly documents the structure of the data [9], relational databases are also commonly adopted because a schema protects the validity and consistency of the stored data through the specification and enforcement of *integrity constraints*. Integrity constraints encode logic ensuring that the data values are: distinct as dictated by an application (e.g., usernames); not absent from a database (e.g., a part must have an identification number); maintain referential integrity with other data values (e.g., the identifier in different parts of the schema must match if they refer to the same entity); and uphold other domain-specific conditions. Prior work has shown that real-world schemas are complex and often include features such as composite keys and multi-column foreign key relationships [10]. Given the importance of data and its consistency—and the role that these complex integrity constraints play in preserving its veracity—testing database schemas is a recommended industry practice [11]. This has led to the creation of testing strategies, coverage criteria, automatic test suite generators, and mutation analysis methods tailored for database schemas [12], [13], [14], [15].

Yet, it is important to ensure that any tests are sophisticated enough to find flaws in a relational database's schema. Although there are several methods for assessing the quality of a test suite (e.g., measuring how well the tests cover the entities in the relational schema [12], [16]), many of them may be limited in their capability to characterize a test suite's fault-exposing potential. As an alternative, mutation analysis is a method that estimates the fault-finding

"strength" of a test suite by generating copies of an artifact under test and seeding small faults, known as "mutants", into those copies [17]. Mutation analysis then repeatedly runs the test suite against each mutant to see if one or more of its test cases are capable of distinguishing between the mutant and the original—that is, whether a test case fails on the mutant that passed with the original. The intuition here is that if a test suite cannot reveal the difference between the mutant and the original then it cannot detect this fault if it appears in subsequent versions of this artifact [18]. The percentage of mutants killed is known as the "mutation score" of the test suite; the higher the mutation score, the stronger the suite is judged to be at trapping real faults [17].

Nevertheless, mutation analysis can result in the generation of many mutants that are useless for the purpose of evaluating a test suite, which we refer to as "ineffective" mutants in this paper. Mutants may be generated that are invalid, known as "stillborn" mutants, or are equivalent to the original artifact or some other generated mutant, called "equivalent" and "redundant" mutants, respectively [19]. Not only can such mutants reduce the usefulness of the final mutation score, they also incur an execution time overhead that is effectively wasted [17]. Moreover, some ineffective mutants, such as those that are equivalent, 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 [17]. In the context of programs, where 45% of undetected mutants are equivalent, the manual study and classification of a mutant takes about fifteen minutes [20]. In summary, ineffective mutants have been a long-standing problem in program mutation [21], and, as this paper shows, they are also a concern for database schema mutation.

In the context of relational database schema mutation, ineffectiveness can manifest itself in a variety of ways. For instance, PRIMARY KEY constraints ensure the uniqueness of database table rows, which is also a property of UNIQUE constraints—for example, this fact leads to a source of equivalent mutants in the SQLite DBMS. In this paper, we identify a wide range of representative root causes of ineffectiveness in the mutants of relational database schemas. We summarize these root causes into a number of patterns in database schemas that can be used for ineffective mutant detection. Not only do we identify sources of stillborn, equivalent, and redundant mutants (as has been previously done for program mutants), we find and classify a new type of ineffective mutant: the "impaired" mutant. Impaired mutants are similar to stillborn mutants, in that they represent infeasible database schemas, but are not damaged to the extent that they are completely invalid and as such automatically rejected by a DBMS. They are nevertheless of little worth in mutation analysis as they are always trivially killed by test cases that attempt to interact with them.

On the basis of these representative patterns, we then present algorithms that are capable of statically analyzing mutants, identifying those that are ineffective and removing them from the mutant pool used in mutation analysis. We implemented them into our test generation and mutation analysis tool for database schema testing, the open-source system called *SchemaAnalyst* [22], and used it to perform an empirical study that incorporated 34 database schemas and three popular and widely used DBMSs—HyperSQL, PostgreSQL, and SQLite. The experiments focused on the testing of many real-world schemas, including those used in the Mozilla Firefox Internet browser and the database backend of the Stack Overflow web site. For the 34 schemas in this study, the experiments performed mutation analysis on a total of 186 tables, 1044 columns, and 590 constraints.

The results of the experimental study show that, in practice, the presented algorithms are capable of detecting and removing large numbers of ineffective mutants. Excluding ineffective mutants from the mutant pool means that mutation scores obtained for test suites become more useful, because, for instance, mutants that are the same as the original artifact—and thus cannot be killed—no longer prevent test suites from achieving 100% mutation scores. Removing ineffective mutants also ensures that redundant mutants are not double counted. In this paper's study, we found that all but one of the schemas we studied had a test suite that experienced a change in mutation score following ineffective mutant removal. The test suites for the one remaining schema always killed all mutants, and as such had already attained a "perfect" mutation score that could not be improved upon. While only 15% of the schemas that we studied had at least one test suite with a perfect score before removing ineffective mutants, a further 21% of schemas had test suites—previously thought to have suboptimal scores—that achieve 100% scores after discounting ineffective mutants, primarily due to the elimination of equivalent mutants.

We also investigated the efficiency of mutation analysis following the removal of ineffective mutants by the presented algorithms, finding that key parts of the analysis become significantly faster to run. In particular, eliminating stillborn mutants using our algorithms is always an order of magnitude faster than relying on the DBMS to "throw out" invalid schemas during the mutation analysis process. The improved efficiency of mutation analysis for other types of mutant depends on the numbers of that type of mutant involved, and whether the upfront time needed to detect and remove them is recouped by not having to consider them during mutation analysis. For instance, the time taken to identify and eliminate redundant mutants is rarely recouped, since the algorithms need to compare every mutant against every other mutant. While savings were indeed possible for several schemas, the overall process took longer for others. Nevertheless, the benefit in these cases is still, as discussed earlier in this section, the increased usefulness of the mutation score. These results also varied depending on the DBMS with which mutation analysis was performed. For fast, lightweight, and in-memory DBMSs, like SQLite, savings are harder to achieve, since tests can be processed quickly. Yet, for an enterprise, disk-based DBMS, such as PostgreSQL, significant time savings are often realizable. Therefore, the important contributions of this paper are:

1) A study and taxonomy of ineffective mutants for relational database schemas—mutants that do not make a useful contribution during mutation analysis, because they are "stillborn" (i.e., invalid), equivalent to the original schema, equivalent to some other mutant, (i.e., "redundant"), or fall into a new class of mutants, those that are "impaired". The study presents a collection of root causes that lead to ineffectiveness in database

schema mutants, explicated as a series of 10 representative patterns common to mutant schemas (Section 3).

2) A family of algorithms that statically analyze relational database schemas and remove ineffective mutants (Section 4) and are implemented as a part of our open-source testing tool called *SchemaAnalyst* (Section 5).

3) The results of an empirical study, incorporating 34 diverse schemas and three well-known and representative DBMSs, that both evaluates the efficiency and effectiveness of the methods for detecting and removing the four types of ineffective mutants and reveals how their removal influences the final mutation score for a relational database schema's test suite. The study includes a manual analysis of the generated mutants that discerns whether any of those not detected by the automated methods are actually still ineffective (Section 6).

This paper is organized as follows. We begin by detailing key background to database schemas, testing methods, and mutation analysis in Section 2. Then, Section 3 introduces a taxonomy of mutant types and a series of root causes and patterns that lead to a mutant schema being ineffective. Following this, Section 4 presents algorithms to detect each pattern of ineffectiveness, allowing these mutants to be removed from the mutant pool used in mutation analysis. Section 6 then presents the results of the empirical study, showing how the presented technique can detect large numbers of ineffective mutants, and how their removal increases the usefulness of mutation scores while potentially decreasing the execution costs of mutation analysis. Finally, we discuss related work in Section 7 and close with concluding remarks and avenues for future work in Section 8.

2 BACKGROUND

This section details the form and structure of relational database schemas, and the integrity constraints that form part of their definition. Since integrity constraints encode vital logic designed to protect the validity and authenticity of database data, it is important that they are tested. To this end, we discuss coverage criteria that have been previously proposed for this purpose, and further explain techniques for the automatic generation of a database-aware test suite. Finally, we introduce mutation analysis, initially in the context of program mutation, showing how it may be applied to relational database schemas for the purpose of estimating the "strength" of the tests used to exercise them.

2.1 Relational Schemas and Integrity Constraints

The schema of a relational database defines the structure and type of data that will reside within it, declaring any relationships between pieces of data that may exist. A relational database is composed of two-dimensional tables. Tables are organized by columns, each of which have a specified data type. 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 [2]. 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 KEYs 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. These can include boolean algebra operators like conjunction, disjunction, and negation, as well as relational operators and database operations, such as "x IS NULL", "x > y" and "x IN (y, ...)".

Figure 1 shows fragments of three different database schemas, highlighting each of the main five types of integrity constraints, and showing differences in declaration style. A segment of the relational database schema of the popular WordNet database, a large online lexical database of words in the English language[3], is shown by Figure 1(a). The snippet involves four tables (i.e., lexlinkref, linkdef, synset, and word) each declared by a separate CREATE TABLE SQL statement. Within each table declaration appear the definition of different columns (e.g., synsetid and word1id for the lexlinkref table). Each column is specified with a datatype (e.g., varchar(80), representing a variable length character string containing up to 80 characters).

The segment also shows a variety of integrity constraints declared by the relational database schema, which Figure 1 also highlights. These include several NOT NULL constraints and a PRIMARY KEY for each table. For instance, the lexlinkref table has a primary key that involves all of its columns, meaning that the combination of values for every row must be unique. Alternatively, the word table defines the uniqueness of its rows through the wordid column.

Data is inserted into relational database tables through SQL INSERT statements. Given the integrity constraints defined for the lexlinkref table, the following INSERT statement would be initially accepted by the DBMS for an empty database (i.e., the DBMS would admit the data); however, it would be rejected by the DBMS (i.e., the values would not be admitted) if it were attempted a second time, as the set of column values would no longer be unique:

```
(1) INSERT INTO lexlinkref
    (synset1id, word1id, synset2id, word2id, linkid)
    VALUES (0, 0, 0, 0, 0, 0);
```

Instead, a distinct set of values would be needed, such as in the following INSERT statement:

```
(2) INSERT INTO lexlinkref
    (synset1id, word1id, synset2id, word2id, linkid)
    VALUES (0, 0, 0, 0, 0, 1);
```

A database table can only have one primary key, so where further constraints are necessary to enforce distinctness of certain values for certain columns, the UNIQUE constraint can be used. For example, in the WordNet schema of Figure 1(a), a UNIQUE is defined on the lemma column in the word table. As such, following INSERT statement 3, INSERT statement 4 will be rejected since the value of lemma is repeated:

	Accepted?
(3) INSERT INTO word(wordid, lemma) VALUES (1, 'x');	✓
(4) INSERT INTO word(wordid, lemma) VALUES (2, 'x');	✗

3. https://wordnet.princeton.edu/

```
CREATE TABLE "lexlinkref" (
  "synset1id" decimal(9,0) NOT NULL
  "word1id" decimal(6,0) NOT NULL
  "synset2id" decimal(9,0) NOT NULL
  "word2id" decimal(6,0) NOT NULL
  "linkid" decimal(2,0) NOT NULL
  PRIMARY KEY ("word1id","synset1id",
              "word2id","synset2id","linkid")
);

CREATE TABLE "linkdef" (
  "linkid" decimal(2,0) NOT NULL
  "name" varchar(50),
  "recurses" char(1) NOT NULL,
  PRIMARY KEY ("linkid")
);

CREATE TABLE "synset" (
  "synsetid" decimal(9,0) NOT NULL
  "pos" char(1),
  "categoryid" decimal(2,0) NOT NULL
  "definition" text,
  PRIMARY KEY ("synsetid")
);

CREATE TABLE "word" (
  "wordid" decimal(6,0) NOT NULL
  "lemma" varchar(80) NOT NULL UNIQUE,
  PRIMARY KEY ("wordid")
);
```

(a) WordNet

```
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")
);
```

(b) Million Song

```
CREATE TABLE Station (
  ID INTEGER PRIMARY KEY,
  CITY CHAR(20),
  STATE CHAR(2),
  LAT_N INTEGER NOT NULL
    CHECK (LAT_N BETWEEN 0 and 90),
  LONG_W INTEGER NOT NULL
    CHECK (LONG_W BETWEEN SYMMETRIC 180 AND -180)
);

CREATE TABLE Stats (
  ID INTEGER REFERENCES STATION(ID),
  MONTH INTEGER NOT NULL CHECK (MONTH BETWEEN 1 AND 12),
  TEMP_F INTEGER NOT NULL CHECK (TEMP_F BETWEEN 80 AND 150),
  RAIN_I INTEGER NOT NULL CHECK (RAIN_I BETWEEN 0 AND 100),
  PRIMARY KEY (ID, MONTH)
);
```
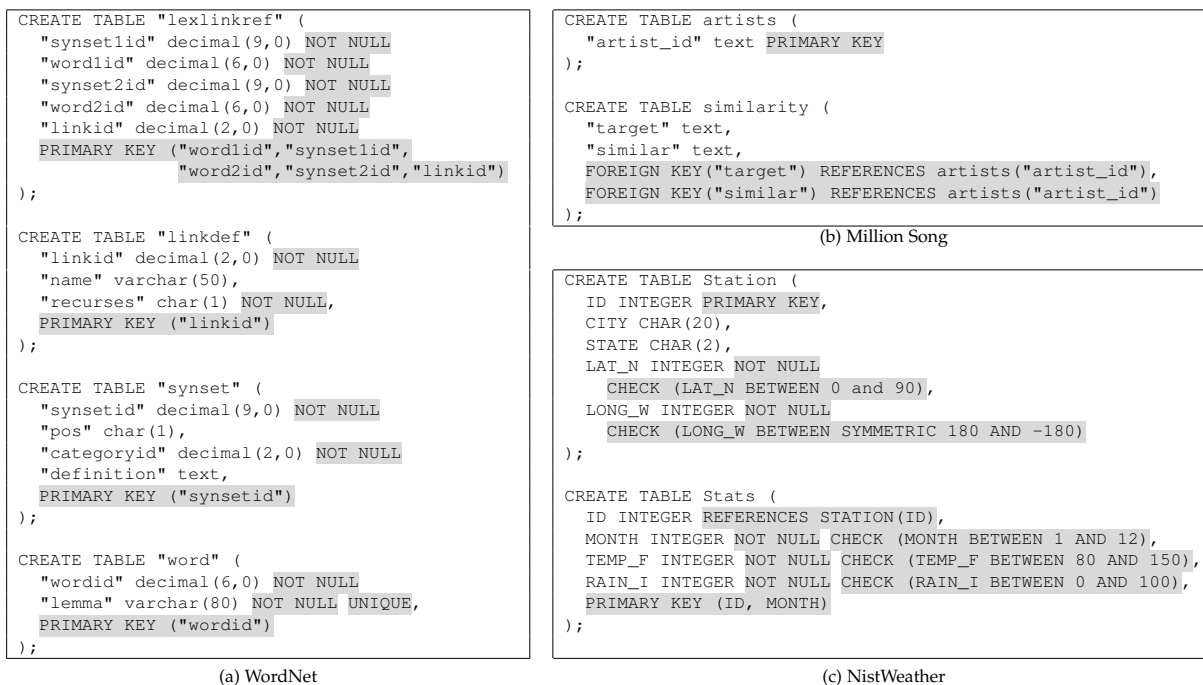
(c) NistWeather

Fig. 1. Fragments of different real-world relational database schemas, showing differences in declaration style and highlighting different integrity constraint types (e.g., a CHECK constraint in the Station table, a FOREIGN KEY in the Similarity or Stats tables, a NOT NULL in the synset or word tables, a PRIMARY KEY in the lexlinkref or linkdef tables, and a UNIQUE in the word table). Note that the creators of these schemas declared some of the columns and tables with quotation marks surrounding the variable name, which is permitted by the SQL standard.

Figure 1(b) shows a schema fragment from the freely available Million Song dataset [23], which contains 280GB of data. This fragment involves two tables, artists and similarity, that contain three integrity constraints. There is a single PRIMARY KEY, defined on the artists table—and in a different style to primary keys declared for the WordNet schema, as this time it is declared inline with the column definition for artist_id. The schema also contains two FOREIGN KEY constraints designed to ensure that each target and similar value in the "source" table, similarity, refer to an existing artist_id value in the "referenced" table, artists. With these constraints, the following INSERT statements 5 and 6 would be accepted into a empty database, while statement 7 would be rejected. Accepted statement 6 uses a value for both target and similar that has already been inserted for artist_id in the artists table. Yet, rejected statement 7 uses a value for similar that does not refer to an existing artist_id value in the referenced table:

Accepted?

```
(5) INSERT INTO artists(artist_id)
    VALUES ('x');                              ✓

(6) INSERT INTO similarity(target, similar)
    VALUES ('x', 'x');                         ✓

(7) INSERT INTO similarity(target, similar)
    VALUES ('x', 'y');                         ✗
```

Finally, Figure 1(c) shows the *NistWeather* database schema, a part of the NIST SQL conformance test suite [24]. This schema also contains a FOREIGN KEY, although declared inline to the ID column of the Stats table. This particular schema features a number of CHECK constraints. For example, the MONTH column of the Stats table has a CHECK constraint defined on it that ensures an integer MONTH value can only be between 1 and 12. Any INSERTs involving values for MONTH outside of this legal range will be rejected by the DBMS.

2.2 Mistakes Leading to Faults in Database Schemas

Given that integrity constraints encode important logic used to protect the validity and consistency of data in a database, it is also important that these constraints are properly tested, in accordance with industry advice [11]. Broadly speaking, a database designer may make mistakes when specifying a relational database schema in two different ways.

Different DBMSs have different implementations of the SQL standard and, additionally, may offer features not specifically required by the standard. A programmer moving from one DBMS to another is therefore open to making mistakes when specifying schemas, since the behavior of DBMSs varies greatly, as will be further demonstrated in Section 3. This is increasingly the case when an engineering team uses a different DBMS for development, in-house testing, and deployment. For example, programmers may prefer the speed and flexibility of a DBMS like SQLite for development, but choose a robust enterprise DBMS, such as PostgreSQL, for use with the deployed application.

One instance of differences in DBMS behavior concerns how PRIMARY KEY constraints are handled by the PostgreSQL and SQLite DBMSs. With PostgreSQL, PRIMARY KEY constraints reject NULL values (as well as ensuring column values are distinct). Yet, for SQLite, NULL values may be admitted for primary key columns. As such, a programmer familiar with PostgreSQL may reasonably expect that the specification of a primary key in SQLite will defend the database against NULL values for the key. However, unless they remember to also additionally specify NOT NULL constraints on the columns of the key, this will not be the case. Thus, the behavior of the database schema must be tested to ensure it is consistent with what the developer intended.

There are other ways in which a programmer may misunderstand SQL dialects. For instance, the treatment of

NULL values in columns denoted as UNIQUE operates in one way for PostgreSQL, SQLite, and HyperSQL and in another manner for the MS SQL Server DBMS, which only allows one instance of a NULL value in a UNIQUE column, on the basis that it is not distinct from other NULL values. Yet, the three other aforementioned DBMSs treat NULL values as meaning "unknown" and therefore still distinct from one another. As such, PostgreSQL, SQLite, and HyperSQL permit multiple instances of NULL in columns constrained by a UNIQUE.

Instead of misunderstanding the dialect of SQL that a DBMS supports, a developer may also make mistakes when specifying the schema by, for example, forgetting to add a PRIMARY KEY or UNIQUE constraint on a field for usernames that controls a system's login. If the database designer omits a PRIMARY KEY constraint on the username column in a database table, then the DBMS hosting this table would allow INSERT statements to create two users who have the same name. Or, if the designer of a schema neglects to add CHECK constraints on fields such as prices or product stock levels to ensure they can never be negative, then it may be possible for INSERT or UPDATE statements to corrupt the database's state. Finally, a designer could specify constraints on the wrong columns, thus, for instance, leading to a database table having the wrong PRIMARY KEY column.

2.3 Database Schema Testing

It is important to perform testing to identify the two broad categories of faults described in Section 2.2. The goal of prior work has been to create a test case that consists of a sequence of SQL INSERT statements that aim to fulfill a test adequacy criterion [13]. The first work on testing integrity constraints, due to Kapfhammer et al. [12], introduced a search-based technique that automatically generates data for composing tests of INSERTs that exercise a database's schema. A test case "passes" when its INSERTs are accepted by the DBMS, as expected, and the data is admitted into the database since it satisfies the constraints of the relational schema. A test case may also pass when its INSERTs are, as anticipated, rejected by the DBMS because the data was generated with the goal of violating the schema's integrity constraints.

Coverage Criteria

McMinn et al. [13] followed up the work in [12] by defining a family of coverage criteria for testing relational database schema integrity constraints. Organized into subsumption hierarchies, these criteria range from simple measures with few coverage goals to more intricate criteria with substantially more test requirements. Each criterion centers on the reformulation of the integrity constraints of a database table as a boolean predicate, referred to as the *acceptance predicate* for the table. This is because the predicate evaluates to *true* when the data in an INSERT statement will be accepted for the table by a particular DBMS (i.e., the data is admitted into the database). Conversely, an INSERT statement will be rejected by a DBMS if the data within it causes the acceptance predicate to evaluate to *false*. "Acceptance Predicate Coverage" (APC), therefore, requires the acceptance predicate for each table to have been exercised as *true* and *false* by the test suite. As such, each table should have had data in an INSERT statement admitted to it at least once, and have had an INSERT statement rejected at least once [13].
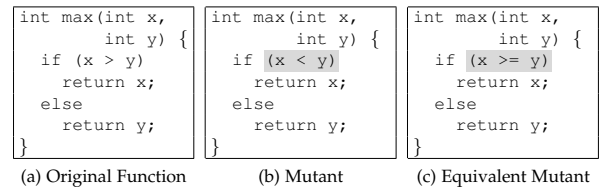

Fig. 2. An example of program mutation. This figure highlights the fact that equivalent mutants cannot be distinguished from the original program as there is no input to the max function for the mutant in part (c) that will produce a different output to the original program in part (a). Additionally, part (b) of this figure highlights a non-equivalent mutant in the max function that is semantically different from the original program.

APC does not, then, require that each particular integrity constraint has been properly exercised, because the INSERT statement may be rejected by the violation of just one of the integrity constraints defined for the table. "Active Integrity Constraint Coverage" (AICC) addresses this limitation. For this criterion, a test case is required that satisfies the acceptance predicate (i.e., all integrity constraints are satisfied), followed by tests that exercise the portion of the predicate corresponding to each integrity constraint as *false*, while ensuring the rest of the predicate evaluates to *true* (i.e., each integrity constraint is violated in isolation). "Clause-Based Active Integrity Constraint Coverage" (ClauseAICC) takes this further, requiring that each individual *clause* of the acceptance predicate be exercised as *false* [13]. A clause could correspond to a single aspect of a particular integrity constraint, for example the uniqueness of a column as part of a multi-column PRIMARY KEY constraint.

Further criteria defined by McMinn et al. include "Active Unique Column Coverage" (AUCC), which requires that test cases be produced that exercise each column of each table with unique and non-unique values, while maintaining satisfaction of the acceptance predicate. Finally, "Active Null Column Coverage" (ANCC) requires that test cases be produced that exercise each column of each database table with NULL and non-NULL values, while also maintaining satisfaction of the acceptance predicate [13].

Automatic Test Case Generation

Kapfhammer et al. [12] presented an extension of Korel's *Alternating Variable Method* (*AVM*) [25] for the automatic generation of data for INSERT statements that form part of test cases for schemas. McMinn et al. [13] extended this approach to generate test suites according to their coverage criteria. In that paper and the remainder of this one, a test suite is a collection of test cases, each of which contains its own INSERT statements designed to fulfill a testing objective. The paper also introduced a random approach, referred to as *Random$^+$*, that utilizes constants mined from the database schema's definition. An empirical study conducted by McMinn et al. revealed that the *AVM* tends to reliably generate database-aware test suites that provide full coverage of the criteria, while *Random$^+$* is more erratic and cannot guarantee such high levels of test coverage [13].

2.4 Mutation Analysis of Schema Integrity Constraints

Mutation analysis is a useful method for estimating the "strength" of a test suite—that is, its potential fault-finding capability [17]. The process of mutation works by producing copies of the artifact under test—traditionally a program—and making minor changes to them so as to simulate

```
CREATE TABLE t (        CREATE TABLE t (
    x INT,                  x INT,
    y INT,                  y INT,
    PRIMARY KEY(x)          PRIMARY KEY(x, y)
);                      );
```

(a) Original Schema   (b) Mutant

Fig. 3. An example of a relational database schema integrity constraint mutant. As seen in part (b), the PRIMARY KEY on the table is mutated from the original schema (part (a)) to include the column y as well as x.

faults. The altered copies of the original artifact are called "mutants". Figure 2 shows two examples of mutants for a max function implemented in the syntax of a Java-like programming language. Part (b) of this figure shows how the relational operator in the conditional statement of the original function in part (a) is changed, resulting in the predicate being mutated from "if x > y" to "if x < y".

If a test suite can distinguish between the original artifact and the mutant (i.e., a test case fails on the mutant that previously passed on the original), then the mutant is said to be "killed", else it is "live" [17]. For instance, the mutant in part (b) of Figure 2 is easily distinguished from the original. A test case with the inputs x=1, y=2 gives the output 2 with the original program and 1 with the mutant, and so the test case kills the mutant. A test case with the inputs x=1, y=1 would not kill the mutant, however, since the result is 1 for both versions of the program. The percentage of mutants killed by a test suite is compiled into a metric known as its "mutation score". The higher the mutation score a test suite has, the stronger it is estimated to be. A test suite is said to be mutation-adequate if it kills all mutants, that is, it achieves a "perfect" mutation score of 100% [17]. Intuitively, if a test suite cannot kill a mutant then this means that it would not be able to detect this type of programming error if it was subsequently introduced into the program under test [18].

While mutation analysis was originally proposed for traditional programs [17], it has recently been adopted for a wider range of software artifacts. For instance, Deng et al. and Lindström et al. proposed the use of mutation analysis to assess the adequacy of test suites for Android apps [26], [27]. Mutation testing has also recently been used to measure the effectiveness of test suites for web sites [28], [29], [30]. Additionally, mutation testing has been applied in other diverse domains such as mobile software agents (e.g., [31], [32]) and security policies (e.g., [33], [34]).

In the context of databases, while Bowman et al. focused on the use of mutation testing to assess test suites for an entire database management system [35], Kapfhammer et al. [12] were the first to propose and evaluate mutation operators for the integrity constraints expressed in a relational database schema. These proposed operators created mutants by adding, removing, and replacing columns in the definitions of PRIMARY KEY and UNIQUE constraints, while also adding and removing NOT NULL constraints from other columns in the schema's tables. An operator was also proposed to remove CHECK constraints from schema definitions. Wright et al. [15] extended this set by adding operators that mutate the predicates of CHECK constraints (e.g., by replacing a relational operator such as > with >=) while also adding operators to mutate the columns featuring in a relational database schema's definition of FOREIGN KEY constraints.

Figure 3 shows an example of a mutation to a PRIMARY KEY. For the solitary table of the original schema, shown by part (a), the column x is the sole primary key column. For the mutant, shown by part (b), the column y is also a part of the key. With integrity constraint mutation, a mutant is "killed" when INSERTs made to a database instantiating the mutant schema behave differently compared to a database instantiating the original schema. As highlighted by the fact that the original and mutated schemas lead to different outcomes (i.e., an ✗ indicating rejection and a ✓ meaning acceptance, respectively), the following INSERT statements are capable of distinguishing between the original and the mutant for an initially empty database:

		Original	Mutant
(8)	`INSERT INTO t(x, y) VALUES (0, 0);`	✓	✓
(9)	`INSERT INTO t(x, y) VALUES (1, 0);`	✓	✓
(10)	`INSERT INTO t(x, y) VALUES (0, 1);`	✗	✓

The data in statements (8) and (9) are successfully inserted into the database because the value of x is distinct, thereby satisfying the PRIMARY KEY constraint of the original schema, while the combination of x and y are distinct, satisfying the PRIMARY KEY of the mutant. For statement (10), however, the value for x is not distinct for the column, thereby causing the INSERT statement's rejection. The combination of x and y values is still unique for the mutated PRIMARY KEY, however, and thus statement (10) is accepted, leading to the mutant being killed, as indicated by the ✗ for the original schema and the ✓ for the mutated one.

Like program mutation, mutation analysis of a relational database schema is a costly process that takes a long time to complete, due to the many mutants that may be created, and the fact that the test suite must be run against each mutant to determine if it is killed or remains live. Furthermore, mutation can result in many "ineffective" mutants that do not contribute to the mutation score or make it less useful, while still consuming valuable execution time. The most famous of these—and most widely studied for program mutation—is the "equivalent" mutant [17]. As with all mutants, equivalent mutants correspond to seeded changes—but they do not result in any change in behavior. An example of an equivalent mutant for program mutation is shown by Figure 2(c). The relational operator of the conditional statement has been changed, but the mutant program behaves exactly the same as the original one. That is, there is no input to the mutant that will produce an output different from that of the original. Thus, as previously noted, it is impossible for a test to distinguish between them. Equivalent mutants will always remain "live" following mutation, thereby preventing the tests from achieving a perfect mutation score [17].

In addition, there may be equivalence between pairs of mutants themselves. This means that the same mutants may be considered more than once; Just et al. [36] label these mutants as "redundant" while Papadakis et al. call them "duplicate" and note that they are problematic for mutation testing [37]. Even though the removal of these equivalent and redundant mutants would make mutation testing more efficient, the detection of equivalent mutants for programs is generally undecidable due to the halting problem [17]. Finally, mutation may introduce another type of ineffective

```
CREATE TABLE t (        CREATE TABLE t (        CREATE TABLE t (
  c INT PRIMARY KEY       c INT                   c INT
  ...                     ...                     ...
                          PRIMARY KEY(c)        );
);                      );                      ALTER TABLE t
                                                  ADD PRIMARY KEY(c);
       (a)                     (b)                     (c)
```

Fig. 4. Three relational database schemas that are identical, and therefore equivalent, but declared in different, but valid, ways in the SQL.

```
CREATE TABLE t (        CREATE TABLE t (
  c INT,                  c INT,
  ...                     ...
  PRIMARY KEY (c)         UNIQUE (c)
);                      );
       (a)                     (b)
```

Fig. 5. Two relational database schemas that are different, but functionally equivalent, for SQLite, since, for this DBMS, primary keys reject non-unique values in the same way as is done by UNIQUE constraints.

mutant, known as the stillborn mutant. These mutants are ones where the seeded change has caused it to become invalid—for example, producing a program that does not compile [38]. Stillborn mutants slow down the process of mutation analysis, since there is an execution cost associated with finding them to be invalid (e.g., due to a compilation error) and removing them from the mutant pool so that they receive no further consideration. Overall, as these examples demonstrate, ineffective (i.e., equivalent, redundant, and stillborn) mutants are also a problem for the mutation of the integrity constraints in a schema. The next section explains how these mutants can arise, and, along with identifying a new type of ineffective mutant, define patterns common to schemas that are the direct cause of mutant ineffectiveness.

3 CLASSIFYING THE INEFFECTIVE MUTANTS OF INTEGRITY CONSTRAINTS IN DATABASE SCHEMAS

In this section we describe and define four different types of ineffective mutants produced during the mutation of the integrity constraints encoded in a relational database schema. We give examples of how they occur, and additionally identify common patterns in database schemas that summarize the root cause of their ineffectiveness. Three types of mutant that are ineffective for relational database schemas are also found in program mutation—equivalent, redundant and stillborn mutants [17], [36], [39], [40]. We further identify and explain a fourth type of ineffective mutant for relational database schemas, namely the impaired mutant.

3.1 Equivalent Relational Database Schema Mutants

As with program mutation, equivalent mutants for relational database schemas are mutants that have the same behavior as the original artifact, and as such cannot be distinguished by a test. In SQL, it is possible to express the same two schemas by stating their definition, at the syntactic level, in a slightly different manner. As an example, Figure 4 shows the definition of three schemas that are actually the same. Each schema consists of one table, t, with one column, c, with a PRIMARY KEY constraint defined on that column. Yet, the SQL declaration of the PRIMARY KEY constraint is expressed in three different ways. For the schema shown by part (a), the keyword "PRIMARY KEY" appears on the definition of the column. For the schema shown by part (b), the PRIMARY KEY declaration appears before the end of the table's definition. In the final schema of part (c), the PRIMARY KEY constraint definition appears after the creation of the table via an ALTER statement. We refer to schemas that are identical, but which are possibly declared in different ways, as *structurally equivalent*. We define this property as:

Def. 1 (Structural Equivalence).
  Two relational database schemas s_1 and s_2 are said to be structurally equivalent, if, following declaration, the

tables, columns, and integrity constraints that exist for schema s_1 are identical to those of schema s_2.

It is also possible to express schemas that are structurally different but are functionally equivalent, and so also indistinguishable by a test case. This is because different types of integrity constraints have similar or identical behaviors, or can be combined to have the same effect as another. Since SQLite does not enforce the standard that a PRIMARY KEY should also imply a NOT NULL [41], PRIMARY KEY and UNIQUE constraints are, for this DBMS, identical in terms of accepting and rejecting the same INSERT statements. Figure 5 shows an example of two schemas, which are the same but for the fact that one has a PRIMARY KEY constraint defined for the c column for the schema shown in part (a) of this figure, while the other, shown in part (b), has a UNIQUE constraint defined on the column instead. The functional behavior of these two schemas is the same: when distinct values for c are inserted into the table, the DBMS will accept them. Alternatively, when an INSERT statement contains a value that is already in the database for c, it will be rejected.

However, these database schemas are not equivalent for most other DBMSs (e.g., HyperSQL and PostgreSQL), where PRIMARY KEY constraints also reject the insertion of NULLs in addition to non-distinct values. As such, the two schemas shown in Figure 5 behave differently when managed by these DBMSs: one schema will be responsible for rejecting NULL values submitted for the c column (i.e., the schema in part (a) of the figure) while the other schema will admit them (i.e., the schema in part (b) of the same figure). Therefore, the equivalence of database schemas is a property that varies depending on the DBMS in question. This leads to the following definition of *behavioral equivalence*:

Def. 2 (Behavioral Equivalence).
  Two relational database schemas s_1 and s_2 are said to be behaviorally equivalent for a relational database management system D if when, following their instantiation, for two initially empty (and separate) databases d_1 and d_2, using D, no sequence of INSERT statements $I = \langle i_1, \ldots, i_q \rangle$ exists such that there is an $i_j \in \langle i_1, \ldots, i_q \rangle$ that is accepted by d_1 but rejected by d_2.

Note that, according to this previous definition, structural equivalence is a type of behavioral equivalence: all database schemas that are structurally equivalent to one another are also behaviorally equivalent. As explained in the following definition, an *equivalent mutant* therefore refers to a mutant that is behaviorally equivalent with the original database schema from which it was created:

Def. 3 (Equivalent Relational Database Schema Mutant).
  A mutant m_{eqv} of a database schema s is said to be equivalent if s and m_{eqv} are behaviorally equivalent.

Where equivalent mutants exist, mutation scores are artificially deflated [19], thus, for instance, 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 [20]. Since it is impossible to kill an equivalent mutant, such diagnostic effort on the part of testers 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 [17], an important issue for the mutation of both relational database schemas and programs. Since the large number of equivalent mutants and the high costs of human inspection make it infeasible to manually detect equivalent mutants [20], there are many approaches that attempt to automatically detect them for programs (e.g., [40], [42], [43]). This motivates our work to identify causes of equivalence for database schemas.

Structural equivalence, which occurs at the syntactic level, is one source of equivalent mutants for database schemas that we defined in Definition 1. Behavioral equivalence following integrity constraint mutation is due to the functional equivalence of integrity constraints or between combinations of integrity constraints. We now identify six representative patterns that encapsulate the ways in which behavioral equivalence can manifest.

Pattern BE-1: UNIQUE constraints and PRIMARY KEYs

Pattern BE-1 expresses the form of equivalence demonstrated in Figure 5, where, for DBMSs like SQLite, there is no behavioral difference between PRIMARY KEYs and UNIQUEs. Schemas that are identical but for a UNIQUE instead of a PRIMARY KEY defined on the same column set are equivalent.

Pattern BE-2: PRIMARY KEYs and UNIQUE constraints paired with NOT NULL constraints

For DBMSs where PRIMARY KEYs and UNIQUE constraints do not behave in the same way, because PRIMARY KEYs do not admit NULL values and UNIQUE constraints do (e.g., for DBMSs such as HyperSQL and PostgreSQL), the following two relational database schemas are behaviorally equivalent. If the columns involved in a UNIQUE constraint also have NOT NULL constraints defined on them, the combined behavior is the same as that of a PRIMARY KEY constraint:

```
CREATE TABLE t (        CREATE TABLE t
  c INT UNIQUE NOT NULL   c INT PRIMARY KEY
);                      );
```

Pattern BE-3: PRIMARY KEYs and PRIMARY KEYs paired with NOT NULL constraints

Following from the last rule, and for DBMSs where PRIMARY KEYs do not allow NULL values, NOT NULL constraints defined on primary key fields are superfluous. Thus, a schema without NOT NULL constraints on primary key fields is behaviorally equivalent to an identical schema but with additional NOT NULL constraints defined. That is, the following two database schemas are behaviorally equivalent:

```
CREATE TABLE t (        CREATE TABLE t
  c INT PRIMARY KEY       c INT PRIMARY KEY NOT NULL
);                      );
```

Pattern BE-4: Extraneous UNIQUE constraints

If a set of columns C_{sub} is declared as UNIQUE, any further UNIQUE constraints involving the same columns (i.e, a set $C_{sup}, C_{sub} \subset C_{sup}$) are extraneous. That is, the following two relational database schemas are behaviorally equivalent:

```
CREATE TABLE t (        CREATE TABLE t
  c1 INT UNIQUE,          c1 INT UNIQUE,
  c2 INT                  c2 INT,
);                        UNIQUE(c1, c2)
                        );
```

Column values for c1 will be unique, due to the "UNIQUE(c1)" declaration. Therefore the combination of *any* further column value (i.e., c2) paired with a unique value for c1 will also be unique. This means that the additional constraint "UNIQUE(c1, c2)" in the right-hand schema is superfluous, and the two database schemas are equivalent.

Note that removing "UNIQUE(c1)" from the right-hand schema would not have the same effect: The constraint "UNIQUE(c1, c2)" on its own does not guarantee that c1 is individually unique. It only guarantees that the *combination* of c1 and c2 are unique. As such, removing "UNIQUE(c1)" rather than "UNIQUE(c1, c2)" would change the behavior of the right-hand schema, and it would no longer be equivalent. It is also important to note that one of the integrity constraints could be a PRIMARY KEY (as it is not possible for a table to have two primary keys), since primary keys are equivalent to UNIQUE constraints under certain conditions, as already discussed in Patterns BE-1 and BE-2.

An exception to the rule occurs when another table in the schema has a foreign key referencing the constraint with the greater number of columns (i.e, C_{sup}). In this case, the superset constraint is not redundant, as it is preventing the schema from being invalid. We expand on the issue of foreign keys and schema validity in Section 3.3.

Pattern BE-5: NOT NULL constraints and CHECK ... IS NOT NULL constraints

The effect of a CHECK constraint of the form "CHECK c IS NOT NULL" for some column c is equivalent to defining a NOT NULL constraint on the column, as in the following example:

```
CREATE TABLE t (        CREATE TABLE t
  c INT NOT NULL          c INT,
);                        CHECK c IS NOT NULL
                        );
```

Pattern BE-6: Behaviorally equivalent CHECK constraints

Since CHECK constraints can encode arbitrary constraints, it is possible for them to be specified in different ways while being behaviorally equivalent, as in the following example.

```
CREATE TABLE t (        CREATE TABLE t
  c1 INT,                 c1 INT,
  c2 INT,                 c2 INT,
  CHECK c1 > c2           CHECK c1 >= c2 + 1
);                      );
```

3.2 Redundant Mutants

In the context of program mutation, Just et al. describe a mutant of a conditional expression with one logical operator as being redundant if it leads to the same boolean outcome as

other mutants that are better suited for efficiently assessing test suite effectiveness [36]. In this paper, we use the term more broadly: While equivalent mutants are behaviorally the same as the original artifact, a mutant is redundant with respect to another mutant if they are behaviorally equivalent to one another. This leads to the next definition:

Def. 4 (Redundant Relational Database Schema Mutant).
A mutant m_{red} of a relational database schema s is said to be redundant with respect to some other mutant m of s if m and m_{red} are behaviorally equivalent.

Patterns of redundancy are the same as for equivalence, except that the relationship holds between mutants rather than between a mutant and the original artifact. When a redundant mutant pair is found, one of the mutants may be safely discarded, as it replicates the other mutant in the pair and only serves to artificially inflate mutation scores [37].

3.3 Stillborn Mutants

In the context of program mutation, *stillborn* mutants are programs that do not compile due to a mutation operator making it syntactically or semantically invalid [38]. Such mutants cannot be used during mutation analysis, since they do not represent an artifact against which any tests can be run. Stillborn mutants are also possible for relational database schema mutation, taking the form of syntactically invalid SQL declarations and also arising from semantic invalidity. The submission of SQL statements relating to the CREATE TABLE declarations for an invalid schema, and therefore a stillborn mutant, will be rejected by these DBMSs. Thus, we define the concept of a stillborn mutant[4] as:

Def. 5 (Stillborn Relational Database Schema Mutant).
A mutant m_{stb} of a relational database schema s is said to be stillborn for a DBMS D if any SQL declaration relating to the definition of m_{stb} is rejected by D.

We now define two patterns that are a source of semantically invalid database schemas during integrity constraint mutation, thus leading to stillborn mutants.

Pattern SB-1: PRIMARY KEY and UNIQUE constraints
Some DBMSs, such as HyperSQL, do not allow UNIQUE constraints to be defined on the same column sets as the table's primary key. An attempt to submit a database schema such as the following results in an error. That is, any mutant where UNIQUE constraint columns replicate those of the primary key will be stillborn, as in the following example:

```
CREATE TABLE t (
 c INT PRIMARY KEY UNIQUE
);
```

Pattern SB-2: Foreign key misalignment
Many DBMSs require that, for foreign keys appearing in schemas, the column or columns in the referenced table

---

4. In previous work we referred to stillborn mutants as "quasi" mutants [15], since it was always the case that, in practice, a mutant that was stillborn for one DBMS was not for another. In this paper, we revert to the original "stillborn" term, since we now know that the effective/ineffectiveness status of other types of mutants—for example, equivalent mutants—also varies across different DBMSs.

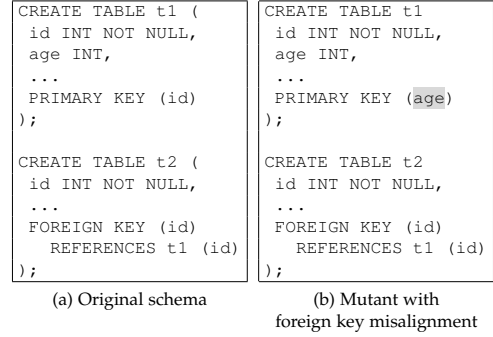(a) Original schema | (b) Mutant with foreign key misalignment

Fig. 6. A relational database schema (in part (a)) and a mutant schema (in part (b)) with foreign key misalignment. With the mutant, the primary key column for t1 has changed (as highlighted) meaning the column referenced by the foreign key for t2 is no longer distinct.

must be the primary key of that table, or be declared in a UNIQUE. We refer to this property as *foreign key alignment*:

Def. 6 (Foreign Key Alignment).
A relational database schema s for a relational database management system D is said to exhibit foreign key alignment when for each foreign key $fk = (t, \langle tc_1 \ldots tc_n \rangle, r, \langle rc_1 \ldots rc_n \rangle)$, where $tc_1 \ldots tc_n$ are columns of the table t on which the key is defined, and $rc_1 \ldots rc_n$ are the columns of the referenced table r for the key, a PRIMARY KEY or UNIQUE constraint exists on r for the columns $rc_1 \ldots rc_n$, and the pairs of columns $(tc_1, rc_1) \ldots (tc_n, rc_n)$ have compatible types for a specific relational DBMS D. A relational database schema is said to exhibit foreign key misalignment when the foreign key alignment property does not hold.

As stated by this definition, column pairs must have compatible types, a property that depends on the DBMS in use. For example, SQLite has a weak typing mechanism allowing any column type to be mapped to any other in a foreign key. In contrast, PostgreSQL is more strongly typed: It will allow a column of type INTEGER to be mapped to a column of type DECIMAL for example, but the pairing of VARCHAR and INTEGER types, for instance, is not allowed by this DBMS. An example of a database schema with correct foreign key alignment, and a mutant with *foreign key misalignment* is shown by Figure 6. The original schema (part (a) of the figure) has a foreign key defined on the table t2, mapping the column id in table t2 to the id column of table t1. Since the id column in t1 is a primary key column, the schema is correctly aligned. However, the mutated version of the schema (part (b) of the figure) has had the primary key column changed from id to age. This schema is misaligned, since the foreign key in table t2 of the mutant is still referencing the non-primary key column id.

Mutants with foreign key misalignment are problematic for most database management systems. For instance, DBMSs such as PostgreSQL and HyperSQL, will reject the second CREATE TABLE statement in Figure 6(b). Other DBMSs, such as SQLite, do not reject database schemas with foreign key misalignment, but simply reject all data that is attempted to be inserted into the table with the misaligned foreign key definition. This leads to a fourth category of ineffective mutant, heretofore not mentioned in the literature and described in the next subsection.
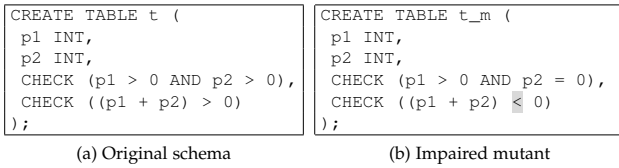
```
CREATE TABLE t (
p1 INT,
p2 INT,
CHECK (p1 > 0 AND p2 > 0),
CHECK ((p1 + p2) > 0)
);
```

(a) Original schema

```
CREATE TABLE t_m (
p1 INT,
p2 INT,
CHECK (p1 > 0 AND p2 = 0),
CHECK ((p1 + p2) < 0)
);
```

(b) Impaired mutant

Fig. 7. A schema and an impaired mutant. The mutation changes the relational operator in the second CHECK (highlighted for the mutant in part (b)), rendering the constraint infeasible. No data can be inserted into the table of the mutant, and as such we describe it as "impaired".

3.4 Impaired Mutants

For database schema mutation, impaired mutants can be created when an integrity constraint is mutated such that satisfaction of the collective system of integrity constraints for that table is not possible. That is, even though the schema will be accepted by the DBMS in use, no INSERT will ever result in a new row of data being added to the database's table. We now discuss how this can occur in terms of patterns found in a definition of a relational database schema.

Pattern IM-1: Foreign key misalignment

Database schemas with incorrect foreign key alignment, that are stillborn for most DBMSs, are impaired for others, such as for SQLite. This DBMS accepts the table definition as valid, but then refuses to accept data into the table with the misaligned foreign key when foreign keys are enabled for the DBMS. IM-1 is identical to SB-2 except that mutants are identified as impaired rather than stillborn.

Pattern IM-2: Infeasible CHECK constraints

For most DBMSs, a similar situation can occur with infeasible CHECK constraints. Infeasible CHECK constraints can occur as a result of schema mutation, as shown by the example in Figure 7. The mutation of the relational operator in the second CHECK results in an infeasible set of constraints, and as a result, every INSERT will be rejected. Since infeasibility of constraints is generally undecidable [40], [44], [45], this type of impairment is hard to detect automatically.

We name these mutants "impaired" mutants. While they are valid relational schemas as far as the DBMS is concerned—and as such do not qualify as being "stillborn"—they have been damaged by the mutation process. Impaired mutants have little use in mutation analysis, due to the ease with which they are killed—essentially any syntactically valid test case will kill this type of mutant. We therefore categorize them as ineffective, and formally define them as follows:

Def. 7 (Impaired Relational Database Schema Mutant).
A mutant m_{imp} of a relational database schema s is said to be impaired for a relational database management system D if there is some table t defined for m_{imp} for which no INSERT statements are accepted by D.

To the best of our knowledge, the concept of an "impaired" mutant has not been defined previously in the literature. An analogous ineffective mutant for program mutation might be a software component that is altered such that whenever it is used or accessed, it returns the same result or throws exceptions, and as such is trivially killed.
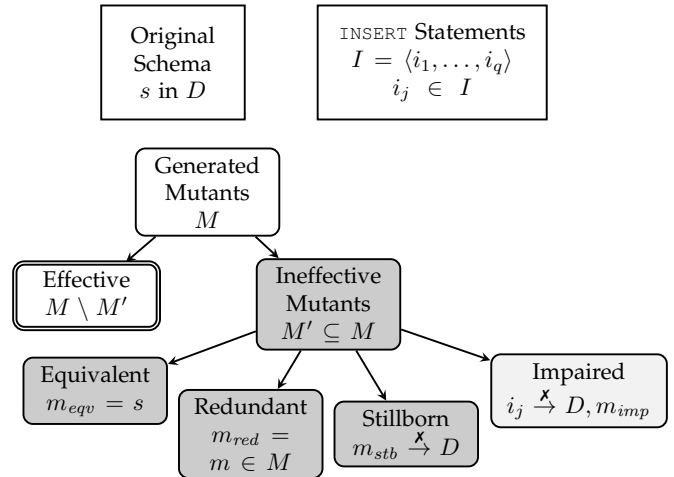
Fig. 8. A taxonomy of database schema mutant types. In this figure, boxes with rounded corners represent a type of mutant and a box with non-rounded corners denotes an artifact that plays a role in determining whether or not a mutant is ineffective. The box with a double border shows that these effective mutants will be used in a subsequent mutation analysis. The light gray box highlights the fact that this paper is the first to draw attention to this type of mutant; boxes with a dark gray background correspond to types of ineffective mutant that have been previously reported in the mutation testing literature for programs.

3.5 Ineffective Mutant Classification Summary

Figure 8 summarizes our categorization of mutants for database schemas. Out of those produced, only some will be "effective". "Ineffective" mutants are ones that are either *equivalent* to the original; *redundant*, since they are the same as an already produced effective mutant; represent an invalid schema for the DBMS concerned, that is they are *stillborn*; or, INSERTs will always fail for one or more tables of the mutant schema, that is, they are *impaired*. Additionally, this section defines representative patterns that describe the four different types of ineffective mutants: there are six patterns each for equivalent and redundant mutants and two patterns each for stillborn and impaired mutants, respectively. The next section explains how to use these patterns to automatically detect and remove ineffective mutants.

4 AUTOMATICALLY DETECTING AND REMOVING INEFFECTIVE DATABASE SCHEMA MUTANTS

Ineffective mutants decrease the usefulness of the mutation score, and may also increase the time taken to perform mutation analysis. This section describes our techniques for automatically removing certain classes of ineffective mutant that can be identified in advance of mutation analysis, thereby improving the usefulness of the mutation scores obtained and potentially decreasing analysis costs. The presented techniques rely on an abstract representation of relational database schemas, which greatly simplifies the analysis that needs to be performed, while not losing key information needed to identify the ineffective mutants. After describing this abstract representation, we then introduce our algorithms that use it when detecting and removing stillborn and impaired mutants. To avoid further unnecessary and potentially costly checks involving database schema comparisons, these types of mutants are automatically removed before applying the algorithms that identify and extract the equivalent and redundant mutants.

4.1 Abstract Representation of Database Schemas

First, our technique parses the SQL statements that declare a relational database schema and creates a model, which we refer to as the "abstract representation" of a schema. This representation abstracts away the syntactic details of an SQL definition that also make the semantic analysis of schemas for ineffectiveness harder to undertake. This step is important both because, as discussed in Section 3.1, SQL can be used to express the same schema property in a variety of ways and, furthermore, SQL dialects vary across DBMSs

In our model, a schema s is a sextuple $s = (T, CC, FK, NN, PK, UC)$, where T is a set of tables, CC is a set of CHECK constraints, FK is a set of FOREIGN KEY constraints, NN is a set of NOT NULL constraints, PK is a set of PRIMARY KEY constraints, and UC is a set of UNIQUE constraints. A table $t \in T$ is a pair (id_t, C) where id_t is a unique string identifier (i.e., $\forall t' = (id'_t, C') \in T, t \neq t', id_t \neq id'_t$) and C is a set of columns. The function COLS can be used to obtain the columns for a table (i.e., COLS$(t) = C$). A column $c \in C$ is a pair $(id_c, type)$, where id_c is a unique string identifier for the column in the table (i.e., $\forall c' = (id'_c, type') \in C, c' \neq c, id_c \neq id'_c$), and $type$ is a label indicating the data type of the column (e.g., INT).

A CHECK constraint $cc \in CC$ is a pair (t_{cc}, p), where t_{cc} is the table to which the CHECK constraint applies, $t_{cc} \in T$, and p is a predicate over the subset of columns COLS(t_{cc}).

A FOREIGN KEY constraint $fk \in FK$ is a quadruple $(t_{fk}, TC_{fk}, r_{fk}, RC_{fk})$, where $t_{fk} \in T$ is the table on which the key is defined, and $r_{fk} \in T$ is the table that it references. $TC_{fk} = \langle tc_1, ..., tc_{len} \rangle$ and $RC_{fk} = \langle rc_1, ..., rc_{len} \rangle$ are two lists of columns of equal length len, $\{tc_1, ..., tc_{len}\} \subseteq$ COLS(t_{fk}) and $\{rc_1, ..., rc_{len}\} \subseteq$ COLS(r_{fk}).

A NOT NULL constraint $nn \in NN$ is a pair (t_{nn}, c_{nn}) where t_{nn} and c_{nn} are the table and column on which the constraint is defined, $c_{nn} \in$ COLS(t_{nn}).

A PRIMARY KEY constraint $pk \in PK$ is a pair (t_{pk}, C_{pk}) where t_{pk} and C_{pk} are the table and columns on which the constraint is defined, where $C_{pk} \subseteq$ COLS(t_{pk}). Only one primary key can be specified per table, that is, $\forall pk \in PK, \nexists pk' = (t'_{pk}, C'_{pk}) \in PK$ such that $pk \neq pk' \wedge t_{pk} = t'_{pk}$.

Finally, a UNIQUE constraint $uc \in UC$ is a pair (t_{uc}, C_{uc}) where t_{uc} and C_{uc} are the table and columns on which the constraint is defined, $C_{uc} \subseteq$ COLS(t_{uc}).

4.2 Stillborn Mutants

As described in Section 3.3, stillborn mutants are mutants that will be rejected by the DBMS and may negatively influence the efficiency of mutation analysis. This paper's static analysis approach involves identifying stillborn mutants on the basis of different patterns. Following the parsing of the relational database schema into the abstract representation described in the previous subsection, the technique applies different checks to each mutant produced by each of the mutation operators. If the check passes, then the technique removes the mutant. The checks undertaken depend on the DBMS in use during mutation analysis and are as follows.

Check 1: PRIMARY KEY and UNIQUE constraints

This check applies to the mutant types characterized by Pattern SB-1 described in Section 3.3 (i.e., UNIQUE constraints defined on exactly the same column set as a PRIMARY KEY in the

---

Algorithm 1 Detecting PRIMARY KEY and UNIQUE constraints on identical column sets for a schema s

```
function UNIQUEONPRIMARYKEY(s = (..., PK, UC))
    for all pk = (t_pk, C_pk) ∈ PK do
        for all uc = (t_uc, C_uc) ∈ UC do
            if t_pk = t_uc ∧ C_pk = C_uc then
                return true
            end if
        end for
    end for
    return false
end function
```

---

same table of a schema). These mutants can be detected at the level of the abstract representation using the function UNIQUEONPRIMARYKEY shown by Algorithm 1. Mutants flagged by this detector can then be removed from the pool that is subsequently used during mutation analysis.

Check 2: Foreign key misalignment

This check investigates mutants for possible foreign key misalignment according to Pattern SB-2 (Section 3.3) for DBMSs that reject these types of schemas (i.e., HyperSQL and PostgreSQL). This check is automatically performed using the abstract representation of the mutant schema and the function DETECTFKMISALIGNMENT in Algorithm 2. If there is misalignment, then the mutant can be removed from the pool used during the subsequent mutation analysis.

4.3 Impaired Mutants

Our checks detect and remove impaired mutants according to Patterns IM-1 and IM-2. Pattern IM-1 (foreign key misalignment) is the same as Pattern SB-2, except it is applied at a different stage for DBMSs that regard mutants with malformed foreign keys as impaired rather than stillborn (e.g., SQLite). Therefore our check for IM-1 re-uses Algorithm 2.

The problem of detecting infeasible CHECKs (Pattern IM-2) is generally undecidable [40], [44], [45], although some simple analyses, with limited generality, may be possible. As stated in Section 8, this task is outside of the scope of this paper and thus we leave it for future work.

4.4 Equivalent and Redundant Mutants

Detection of equivalent and redundant mutants involves the same static analysis checks—that are applied to different mutants—since the basic problem is to detect whether two mutants behave identically (or are identical). In the case of equivalent mutants, the checks for equivalence take place between the original schema and a mutant, while for redundant mutants, the checks occur for each created mutant.

As for stillborn and impaired mutants, the presented solution for detecting equivalence involves the comparison of the abstract representation for a pair of schemas s_1 and s_2. This structural equivalence is trivial to detect as it is simply the check $s_1 = s_2$. Finding behaviorally equivalent mutants is more challenging, however. The presented method converts schemas already in the abstract representation into a normalized form, aiming to produce a single common form of the schema such that behaviorally equivalent mutants will be structurally equivalent. Equivalent mutants can then be removed from the mutant pool used in a later mutation analysis. For a pair of identical mutants, one of the mutants is redundant and can be removed from the pool.

Normalization of schemas involves a series of transformation steps, which are linked to the patterns of equivalence identified in Section 3.1. We describe these as follows, furnishing algorithms in terms of our abstract representation and illustrating the algorithms with examples, which, for ease of understanding, we demonstrate as if the abstract schema were written back out into SQL CREATE TABLEs. For clarity, we use before and after examples of database schemas to fully illustrate each of the transformation steps.

Algorithm 2 Detecting foreign key misalignment for a schema s and a DBMS D

The function COMPATIBLE returns true if two columns have compatible types for the DBMS D, else it returns $false$.
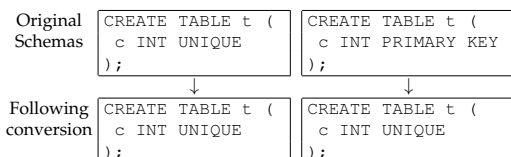
> **function** DETECTFKMISALIGNMENT$(s = (\ldots, FK, \ldots, PK, UC), D)$
>   **for all** $fk = (t_{fk}, \langle tc_1, ..., tc_{len}\rangle, r_{fk}, \langle rc_1, ..., rc_{len}\rangle) \in FK$ **do**
> $compatible \leftarrow true$
>   **for** $i = 1...len$ **do**
>   **if** $\neg$COMPATIBLE(D, tc_i, rc_i) **then**
> $compatible \leftarrow false$
>   **end if**
>   **end for**
> $C_{fk} \leftarrow \{rc_1, ..., rc_{len}\}$
> $foundPK \leftarrow false$
>   **for all** $pk = (t_{pk}, C_{pk}) \in PK$ **do**
>   **if** $r_{fk} = t_{pk} \wedge C_{fk} = C_{pk}$ **then**
> $foundPK \leftarrow true$
>   **end if**
>   **end for**
> $foundUC \leftarrow false$
>   **for all** $uc = (t_{uc}, C_{uc}) \in UC$ **do**
>   **if** $r_{fk} = t_{uc} \wedge C_{fk} = C_{uc}$ **then**
> $foundUC \leftarrow true$
>   **end if**
>   **end for**
>   **if** $\neg compatible \vee \neg(foundPK \vee foundUC)$ **then**
>   **return** $false$
>   **end if**
>   **end for**
>   **return** $true$
> **end function**

Transformation Step 1: Conversion of PRIMARY KEYs

The first transformation step corresponds to the equivalence patterns BE-1 and BE-2, converting primary keys to equivalent UNIQUEs. The transformation depends on the DBMS's "understanding" of how primary keys should behave. If, like HyperSQL and PostgreSQL, primary key column values should also be not NULL, the conversion involves also adding NOT NULL constraints to the columns concerned. If NULL values can be inserted into primary key columns, as for SQLite, this step is ignored, as Algorithm 3 shows.

The next example illustrates how, for SQLite, the following two schemas, which are behaviorally equivalent as described by pattern BE-1, are normalized into a structurally equivalent form by the transformation step. The right-hand database schema, involving a PRIMARY KEY constraint is affected by the change, and is normalized such that it is now structurally equivalent to the left-hand schema:

Original Schemas	`CREATE TABLE t (`
	`  c INT UNIQUE`
	`);`
		`CREATE TABLE t (`
		`  c INT PRIMARY KEY`
		`);`

Following conversion	`CREATE TABLE t (`
	`  c INT UNIQUE`
	`);`
		`CREATE TABLE t (`
		`  c INT UNIQUE`
		`);`

The next two examples show database schemas that will be submitted to a DBMS that mandates primary

---

Algorithm 3 The conversion of PRIMARY KEY constraints for a schema s and a DBMS D

The nature of the conversion depends on the DBMS being used. If a DBMS D—such as HyperSQL or PostgreSQL—rejects NULL as a primary key value, the function PKSARENOTNULL returns $true$ and NOT NULL constraints are added to each of the PRIMARY KEY constraints converted to UNIQUE constraints. For DBMSs that accept NULL as a primary key value (e.g., SQLite), the function PKSARENOTNULL returns $false$, and this particular step is ignored.

> **function** CONVERTPKs$(s = (\ldots, PK, UC), D)$
>   **for all** $pk = (t_{pk}, C_{pk}) \in PK$ **do**
> $PK \leftarrow PK \setminus \{pk\}$
> $UC \leftarrow UC \cup \{(t_{pk}, C_{pk})\}$
>   **if** PKSARENOTNULL(D) **then**
>   **for all** $pkc \in C_{pk}$ **do**
> $NN \leftarrow NN \cup \{(t_{pk}, pkc)\}$
>   **end for**
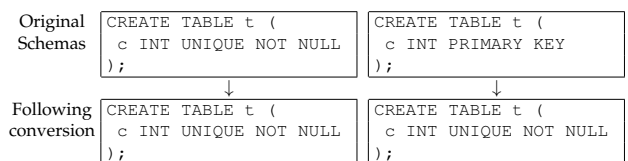>   **end if**
>   **end for**
> **end function**

Algorithm 4 Removing extraneous UNIQUE constraints involving a superset of columns for some existing UNIQUE constraint defined on some table for a schema s
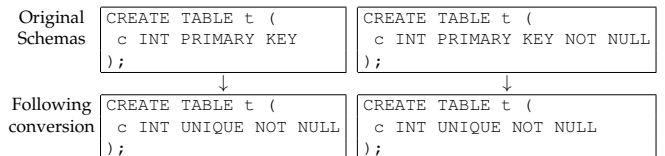
> **function** CONVERTUCs$(s = (\ldots, FK, \ldots, UC))$
>   **for all** $uc = (t_{uc}, C_{uc}) \in UC$ **do**
>   **for all** $uc' = (t'_{uc}, C'_{uc}) \in UC, uc \neq uc'$ **do**
>   **if** $\nexists fk = (t_{fk}, TC_{fk}, r_{fk}; RC_{fk}) \in FK, r_{fk} = t_{uc} \wedge RC_{fk} = C_{uc}$ **then**
>   **if** $C'_{uc} \subset C_{uc}$ **then**
> $UC \leftarrow UC \setminus \{uc'\}$
>   **end if**
>   **end if**
>   **end for**
>   **end for**
> **end function**

---

key columns should not involve NULL values (e.g., HyperSQL and PostgreSQL). In the first example, the two schemas are behaviorally equivalent according to pattern BE-2. The right-hand schema is normalized by Algorithm 3, and becomes structurally equivalent to the left-hand schema:

Original Schemas	`CREATE TABLE t (`
	`  c INT UNIQUE NOT NULL`
	`);`
		`CREATE TABLE t (`
		`  c INT PRIMARY KEY`
		`);`

Following conversion	`CREATE TABLE t (`
	`  c INT UNIQUE NOT NULL`
	`);`
		`CREATE TABLE t (`
		`  c INT UNIQUE NOT NULL`
		`);`

The second example involves a pair of relational database schemas that are behaviorally equivalent according to pattern BE-3. Again, the transformation step converts these schemas into structural equivalents. In this example, both schemas are affected. In the right-hand schema, a NOT NULL constraint is not added since one is already present for the column c in the set NN for the database schema.

Original Schemas	`CREATE TABLE t (`
	`  c INT PRIMARY KEY`
	`);`
		`CREATE TABLE t (`
		`  c INT PRIMARY KEY NOT NULL`
		`);`

Following conversion	`CREATE TABLE t (`
	`  c INT UNIQUE NOT NULL`
	`);`
		`CREATE TABLE t (`
		`  c INT UNIQUE NOT NULL`
		`);`

Transformation Step 2: Remove extraneous UNIQUE constraints

Transformation step 2 removes extraneous UNIQUEs defined on schemas—constraints that are superfluous since there is already some other UNIQUE constraint defined on the same table involving a subset of columns (Pattern BE-4 in Section 3.1). Algorithm 4 implements this step, with its third line ensuring that it does not remove a UNIQUE that is involved in a FOREIGN KEY. Note that this algorithm need not be concerned if one of the constraints is a PRIMARY KEY,

since these will already have been converted in the previous step (transformation step 1). The following example shows two schemas that are behaviorally equivalent, as described by equivalence pattern BE-4. The right-hand schema has an extraneous UNIQUE that is removed by the algorithm, such that the two schemas become structurally equivalent:
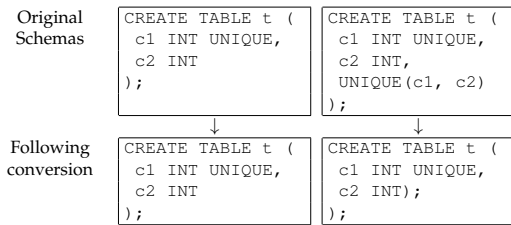
| Original Schemas | ```
CREATE TABLE t (
  c1 INT UNIQUE,
  c2 INT
);
``` | ```
CREATE TABLE t (
  c1 INT UNIQUE,
  c2 INT,
  UNIQUE(c1, c2)
);
``` |
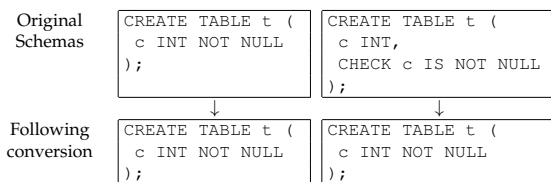|---|---|---|
| Following conversion | ```
CREATE TABLE t (
  c1 INT UNIQUE,
  c2 INT
);
``` | ```
CREATE TABLE t (
  c1 INT UNIQUE,
  c2 INT);
);
``` |

Transformation Step 3: Replace instances of CHECK ... IS NOT NULL *with* NOT NULL *constraint*

As described by equivalence pattern BE-5, CHECK constraints of the form CHECK ... IS NOT NULL are behaviorally equivalent to NOT NULL constraints. Algorithm 5 describes how they may be removed in the abstract representation.

The following example shows how such CHECK constraints are removed by the algorithm so that the two schemas involved become structurally equivalent:

| Original Schemas | ```
CREATE TABLE t (
  c INT NOT NULL
);
``` | ```
CREATE TABLE t (
  c INT,
  CHECK c IS NOT NULL
);
``` |
|---|---|---|
| Following conversion | ```
CREATE TABLE t (
  c INT NOT NULL
);
``` | ```
CREATE TABLE t (
  c INT NOT NULL
);
``` |

Finally, we do not handle behavioral equivalence pattern BE-6 in this paper, due to the undecidability of identifying equivalent constraint systems [40], [44], [45]. While simple cases of the problem could be handled by customizing the presented algorithms, we intend, as noted in Section 8, to more generally tackle this task as part of future work.

5 MUTATION ANALYSIS WITH *SchemaAnalyst*

We implemented mutation analysis (i.e., the generation of mutants and the repeated execution of the test suite to determine the mutants' kill status) and the ineffective mutant removal algorithms into our *SchemaAnalyst* tool [22], which supports the SQLite, PostgreSQL, and HyperSQL DBMSs. Although *SchemaAnalyst* also performs the automatic generation of test suites for relational schemas [22], since it is the primary focus of this paper, Figure 9 only shows the different steps involved in mutant production with *Schema-Analyst*, which we describe in the following subsections.

5.1 Automated Relational Schema Parsing

SchemaAnalyst begins by parsing the SQL declarations of the relational schema (i.e., the CREATE TABLE statements) into the abstract, DBMS-independent schema representation described in Section 4.1 (Step 1 of Figure 9). To control the threats to validity that may arise from incorrectly breaking down SQL commands, *SchemaAnalyst* performs parsing with the General SQL Parser (GSP)[5], a commercial tool that

5. General SQL Parser (GSP) is available at http://sqlparser.com.

Algorithm 5 Converting NOT NULL predicates in CHECK constraints to NOT NULL constraints for a schema s

The function assumes that the predicate of each CHECK constraint is in conjunctive normal form. The function REMOVE removes a conjunct from a predicate, returning the modified predicate or $\perp$ if no conjuncts remain.

```
function CONVERTCHECKNULLS(s = (..., CC, ..., UC, ...))
    for all cc = (t_uc, p = p_1 ∧ ... ∧ p_max) ∈ UC do
        for all c ∈ COLS(t_uc) do
            for conjunct = 1 ... max do
                if p_conjunct = "c IS NOT NULL" then
                    p ← REMOVECLAUSE(p, conjunct)
                    NN ← NN ∪ {(t_uc, c)}
                end if
            end for
            if p = ⊥ then
                CC ← CC \ {cc}
            end if
        end for
    end for
end function
```

handles SQL for a variety of database management systems, including the three used in the empirical study of Section 6.

5.2 Automated Generation of Mutants

After the schema is parsed into the abstract representation, the tool applies mutation operators to produce mutant schemas (Step 2 of Figure 9). Table 1 summarizes 13 different mutation operators that we apply in this paper and which are implemented into *SchemaAnalyst*. Designed to model the types of mistakes that database designers might make when specifying a schema, as outlined in Section 2.2, these operators were originally proposed by Kapfhammer et al. [12] and Wright et al. [15] for introducing synthetic faults into the integrity constraints of a relational schema. We designed these operators according to the following principles:

- Operators should make the smallest possible changes to the integrity constraints in a relational database schema.
- Operators should be as general as possible, applying to a wide range of DBMSs and vendor interpretations of the SQL standard. Thus, it is not an operator's responsibility to avoid the production of mutants that may be ineffective for one DBMS but effective for another.
- An operator should not create mutants that are trivially redundant with respect to its other produced mutants.
- An operator should be usable independently of other operators, thus enabling it to work in either a selective or a higher-order mutation strategy. It is therefore not an operator's concern as to whether the mutants it produces are redundant or not with respect to mutants that may or may not be produced by other operators.

While prior work has defined mutation operators for other parts of a database application (e.g., the SQL SELECT statements created by a program [46]), *SchemaAnalyst* does not incorporate them since they do not adhere to the aforementioned design principles. Notably, operators that manipulate the SELECTs cannot directly process the CREATE TABLE statements that define a relational schema. Section 8 explains that, in future work, we will customize these SELECT-based operators so that they can manipulate schemas.

For brevity and ease of identification, we assign each operator a name according to the constraint it targets 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.

The first three operators mutate CHECK constraints. The "CInListElementR" operator removes individual elements from the list of an IN expression (e.g., "CHECK month IN (1, 2, 3 ...)"). The second, "CRelOpE", produces mutants by replacing the relational operator (i.e., =, <, >, <=, and >=) in an expression of a CHECK constraint with each other possible relational operator. Finally, the third operator, called "CR", simply removes a CHECK from the definition of a schema.

The next two operators mutate FOREIGN KEY constraints. Foreign key definitions require pairs of columns that map values in a column in the table on which the FOREIGN KEY is defined, referred to as the "source" table, to a column in the "referenced" table. "FKColumnPairR", in contrast, performs the reverse operation, removing a pair of columns from an existing FOREIGN KEY constraint. The "FKColumnPairE" operator exchanges one of the columns in one of the pairs of the key (that is, the column that is changed can be on the source table side of the pair, or on the referenced table side of the pair). Wright et al. [15] also proposed an "FKColumnPairA" operator, which added a pair of columns to an existing foreign key. However, due to foreign key misalignment, this operator only produces non-stillborn or impaired mutants in a very narrow set of circumstances [47]. It does not result in any effective mutants for the representative schemas that we study in the experiments of Section 6 and thus, to forestall artificially inflating the significance of the results, we omitted it from the empirical evaluation.

Two operators mutate NOT NULL constraints. The "NNA" operator adds a NOT NULL constraint to a column that did not previously have one, while the "NNR" operator removes an existing NOT NULL constraint from a table's column.

Three additional operators mutate PRIMARY KEY constraints. The "PKColumnA" operator adds a column to an existing PRIMARY KEY constraint, or creates a new one from a column should a table not already have a primary key defined. The "PKColumnR" operator performs the reverse operation of removing a column from an existing primary key, while the "PKColumnE" operator exchanges a column in an existing key for another one in the table.

The final three operators mutate UNIQUEs in much the same way as primary keys are mutated: adding columns to an existing UNIQUE constraint or creating new constraints ("UColumnA"), removing columns from existing columns ("UColumnR"), and exchanging them ("UColumnE").

In contrast to program mutation, which makes small changes to program code at the syntactic level, these operators apply mutation at a semantic level, automatically processing the abstract representation provided by the *SchemaAnalyst* tool. This method has clear advantages as it avoids the production of many kinds of ineffective mutants from the outset. Stillborn mutants that result from syntactical issues are not possible, while structurally equivalent mutants, such as those illustrated in Figure 4, cannot be generated.

By definition, each of these operators cannot produce a mutant that is structurally equivalent to the original schema. However, some operators (i.e., "UColumnE") employ additional checks to ensure that structurally equivalent pairs of mutants (i.e., where one of the pair is redundant) are not produced. In adherence to our design principles, each

TABLE 1
The mutation operators studied in this paper

In this table, the naming convention for the mutation operators follows a system according to the constraint type being mutated (e.g., Primary Key), the aspect being mutated (that is generally a column in a table of the database), and how the aspect is being mutated (i.e., Added, Removed, or Exchanged with another).

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

operator does not know which other operators are being used together, nor does it have a notion of behavioral equivalence or invalidity as these concepts are DBMS specific. Therefore, ineffective mutants may be produced that are stillborn, equivalent, redundant, or impaired. As such, we implemented the algorithms described in the last section to automatically remove these ineffective mutants. The next subsection introduces the details of this implementation.

5.3 Automatic Removal of Ineffective Mutants

Following the tool's automatic generation of mutants, the stage that is novel to this paper removes the ineffective (i.e., the stillborn, impaired, equivalent, and redundant database schema mutants), as discussed in Section 3, and according to the algorithms described in Section 4. The algorithms detailed in that section occupy steps 3–6 of Figure 9.

Step 3 (removal of stillborn mutants), consists of applying Checks 1 and 2 (described in Section 4.2) for HyperSQL. For PostgreSQL, *SchemaAnalyst* only applies Check 2, the only relevant check for this DBMS; for SQLite, no stillborn mutants can be identified, so the tool does not perform any checks. Any mutants found to be stillborn by these checks are removed from the mutant pool. In the Figure 9 and in the experiments of Section 6, this is the set of mutants referred to as $-S$, since it contains all of the generated mutants, minus those the tool identified as stillborn.

Step 4 (removal of impaired mutants) applies to SQLite, since *SchemaAnalyst* only needs to check for mutants with foreign key misalignment for this DBMS. In Figure 9 and in Section 6's experiments, this is the set of mutants denoted $-(S+I)$, since it contains all of the generated mutants, minus those identified as being stillborn and impaired.

Step 5 (removal of equivalent mutants) normalizes the remaining mutants according to the transformation steps described in Section 4.4. It then compares mutants with the (normalized) original schema for structural equivalence. *SchemaAnalyst* removes mutants identified as equivalent from the mutant pool. In Figure 9 and in the experiments of Section 6, this is the set of mutants referred to as
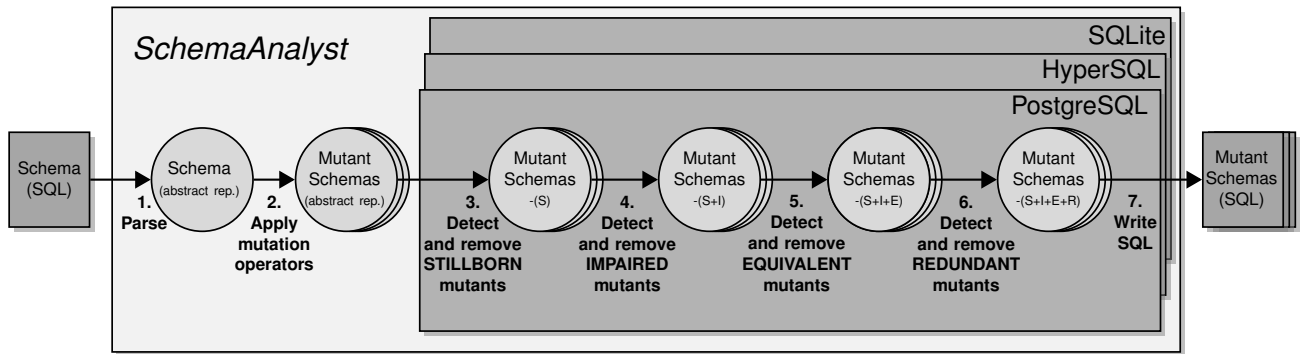
Fig. 9. The inputs and outputs of automatic mutation analysis in the *SchemaAnalyst* tool. In this figure, a dark square represents the tool and its constituent parts, an arrow stands for a process, a rectangle is a SQL representation, and circle symbolizes a relational database schema.

$-(S+I+E)$, since it contains all the mutants generated, minus those the tool identified as being stillborn, impaired as well as those equivalent to the original schema.

Step 6 (removal of redundant mutants) uses the normalized mutants in checks for equivalence between the mutants themselves. Where two mutants are found to be structurally identical, one of the mutants is marked as redundant and removed from the mutant pool. In Figure 9 and in the experiments of Section 6, this is the set of mutants referred to as $-(S+I+E+R)$, since it contains all the mutants generated, minus those automatically identified by the tool as stillborn, impaired, equivalent, and redundant with respect to some other mutant with which it is equivalent in the mutant pool.

5.4 Automated Mutation Analysis

Once it completes the phases in Figure 9, *SchemaAnalyst* then outputs the mutant schemas in the form of SQL CREATE TABLE statements, following a standardized SQL-writing process tailored to the DBMS in use in step 7. Mutation analysis can then begin, as described in Section 2.4. *SchemaAnalyst* applies its automatically generated suites of INSERT statements to the original and mutant schemas, checking whether the INSERTs are accepted or rejected by a DBMS in the same way for the two schemas. If there is any difference, then the mutant is killed, else it is deemed to be alive. Using this information, *SchemaAnalyst* computes the higher-is-better mutation score for the test suite.

6 EMPIRICAL STUDY

In order to evaluate Section 4's technique that automatically detects and removes ineffective mutants for database schemas, we designed an empirical study with the aim of answering the following three research questions:

RQ1: Ineffective Mutants Detected by Static Analysis. How many stillborn, impaired, equivalent, and redundant relational database schema mutants are detected using *SchemaAnalyst*'s automated static analysis approach? Do any ineffective mutants remain that were not identified by the presented method, and how are they characterized?

RQ2: Efficiency of the Approach. How does the up-front time cost of statically identifying and removing impaired, equivalent, and redundant database schema mutants compare to the time savings made in not having to analyze them during mutation analysis? In other words, is mutation analysis more efficient overall with or without the use of automatic ineffective mutant identification and removal?

RQ3: Impact on the Mutation Score. How does the automatic removal of impaired, equivalent, and redundant mutants influence the mutation score for the schema's tests? That is, how often does the removal of ineffective mutants cause a test suite's mutation score to increase or decrease? Does ineffective mutant removal ever enable a previously non-adequate test suite to achieve a perfect mutation score?

6.1 Methodology

We now describe the methodology that we used to conduct our experiments with *SchemaAnalyst*, beginning with our choice of database schemas to use in mutation analysis with and without the removal of the ineffective mutants.

6.1.1 Subject Schemas

In order to answer the aforementioned research questions, we constructed a representative set of 34 database schemas, over double the size of the set of subjects that featured in the conference version of this paper [15], and larger than in previous work on testing database schemas (e.g., [12], [13], [14]). Houkjær et al. notes that complex real-world relational schemas often include features such as composite keys and multi-column foreign-key relationships [10]. As such, the schemas chosen for this paper's study reflect a diverse set of features, from simple instances of each of the main types of integrity constraint (i.e., PRIMARY KEY constraints, FOREIGN KEY constraints, UNIQUE constraints, NOT NULL constraints, and CHECK constraints) to more complex examples involving many-column foreign key relationships. Additionally, the set of subjects that we used in this study involve database schemas drawn from a range of sources. Further details are shown by Table 2: the number of tables in each relational database schema varies from 1 to 42, with a range of just 3 columns in one of the smallest schema, to 309 in the largest. Collectively, the 186 tables and 1044 columns feature each of the main types of database schema integrity constraint that our mutation operators seek to manipulate.

Several schemas were taken from real-world projects. For example, *ArtistSimilarity* and *ArtistTerm* are schemas that underpin part of the Million Song dataset, a freely available research dataset of song metadata [23] (a fragment of which we introduced earlier in Figure 1(b)). *Cloc* is a schema for the database used in a popular open-source

application that counts the number of various types of lines in code for many different programming languages (http://cloc.sourceforge.net). *IsoFlav_R2* belongs to a plant compound database from the U.S Department of Agriculture. *JWhoisServer* is used in an open-source, Java-based implementation of a server for the Internet WHOIS protocol (http://jwhoisserver.net). *MozillaExtensions* and *MozillaPermissions* were both extracted from SQLite databases that are a part of the Mozilla Firefox Internet browser. *RiskIt* is a database schema that forms part of a system for modeling the risk of insuring an individual (http://sourceforge.net/projects/riskitinsurance). *StackOverflow* is the underlying schema used by a popular programming question and answer website, as previously studied in a conference data mining challenge [48]. *UnixUsage* is taken from an application for monitoring and recording the use of Unix commands, while *WordNet* is the database schema used in a graph visualizer for the WordNet lexical database (a fragment of which was introduced earlier in Figure 1(a)). While some of these database schemas are from real-world applications not used in prior experiments, we chose others because they featured in previous studies of various testing methods (e.g., *RiskIt*, *UnixUsage* [49], and *JWhoisServer* [50]).

The six "Nist–" schemas are drawn from the SQL Conformance Test Suite of the National Institute of Standards and Technology (NIST) [24], and have featured in past studies such as those conducted by Tuya et al. [46] (the *NistWeather* schema in particular is shown by Figure 1(c)). *DellStore*, *FrenchTowns*, *Iso3166*, and *Usda* were taken from the samples for the PostgreSQL DBMS, available from the PgFoundry.org website. *iTrust* is a large schema that was designed as part of a patient records medical application to teach students about software testing methods. It previously featured in a mutation analysis experiment of Java code [51]. The remaining schemas (e.g., *BankAccount*, *BookTown*, *CoffeeOrders*, *CustomerOrder*, *Person*, and *Products*) were extracted from the textbooks, assignments, and online tutorials in which they were provided as examples. While simpler than some of the other schemas used in our study, they nevertheless proved challenging for open-source database analysis tools such as the *DBMonster* data generator [12].

Since many of the database schemas studied in this paper's experiments contain many lines of complex SQL code, we do not include them in this paper. However, all of the schemas used as subjects are available from the web site for the *SchemaAnalyst* tool [22]. Moreover, the *SchemaAnalyst* tool parsed the SQL for each of these schemas into the abstract representation that was previously described in Section 5. Once in this abstract form, the tool wrote the SQL out again for each of the particular DBMSs featured in our study, regardless of minor differences in the version of SQL used; the abstract representation of each schema is also available for download from *SchemaAnalyst*'s web site [22].

6.1.2 Subject DBMSs

We performed experiments using the HyperSQL, PostgreSQL, and SQLite DBMSs. Each of these database management systems is supported by our *SchemaAnalyst* tool [22]; they were chosen for their performance differences and varying design goals. PostgreSQL is a full-featured, extensible, and scalable DBMS, while HyperSQL is

TABLE 2
The 34 relational database schemas studied in this paper

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
BrowserCookies	2	13	2	1	4	2	1	10
Cloc	2	10	0	0	0	0	0	0
CoffeeOrders	5	20	0	4	10	5	0	19
CustomerOrder	7	32	1	7	27	7	0	42
DellStore	8	52	0	0	39	0	0	39
Employee	1	7	3	0	0	1	0	4
Examination	2	21	6	1	0	2	0	9
Flights	2	13	1	1	6	2	0	10
FrenchTowns	3	14	0	2	13	0	9	24
Inventory	1	4	0	0	0	1	1	2
Iso3166	1	3	0	0	2	1	0	3
IsoFlav_R2	6	40	0	0	0	0	5	5
iTrust	42	309	8	1	88	37	0	134
JWhoisServer	6	49	0	0	44	6	0	50
MozillaExtensions	6	51	0	0	0	2	5	7
MozillaPermissions	1	8	0	0	0	1	0	1
NistDML181	2	7	0	1	0	1	0	2
NistDML182	2	32	0	1	0	1	0	2
NistDML183	2	6	0	1	0	0	1	2
NistWeather	2	9	5	1	5	2	0	13
NistXTS748	1	3	1	0	1	0	1	3
NistXTS749	2	7	1	1	3	2	0	7
Person	1	5	1	0	5	1	0	7
Products	3	9	4	2	5	3	0	14
RiskIt	13	57	0	10	15	11	0	36
StackOverflow	4	43	0	0	5	0	0	5
StudentResidence	2	6	3	1	2	2	0	8
UnixUsage	8	32	0	7	10	7	0	24
Usda	10	67	0	0	31	0	0	31
WordNet	8	29	0	0	22	8	1	31
Total	186	1044	38	49	357	122	24	590

a lightweight, small DBMS that supports an "in-memory" mode that avoids disk writing. SQLite is a lightweight DBMS that differs in its interpretation of the SQL standard in subtly different ways from HyperSQL and PostgreSQL. All three of these DBMSs are used in a wide variety of real-world programs from many diverse application domains.

6.1.3 Automatic Generation of the Example Test Suites

To study the effect of removing ineffective mutants on the usefulness and cost of mutation analysis, we needed example test suites on which to perform mutation analysis. Since none of the chosen schemas are accompanied by a suite of tests that contain a sequence of INSERT statements, we generated suites with *SchemaAnalyst* using the approach described in our prior work [13]. In that paper, we detailed a series of coverage criteria and automated techniques that aim to generate tests to fulfill them. Importantly, the number of tests generated by the techniques from our prior work is a function of the chosen coverage criteria and not a parameter whose values are controlled by the empirical study's design.

We used *Random*$^+$ and *AVM* to automatically generate test cases with the aim of satisfying the coverage criteria combination of "ClauseAICC", "AUCC", and "ANCC", as previously introduced in Section 2.3. Since our previous work showed that these two test generators satisfy these particular criteria to different degrees [13], resulting in tests with medium to strong mutant killing power, we deemed them highly suited to the task of assessing the relative usefulness and costs of mutation analysis with and without ineffective mutant removal. Since both methods rely on random number generation, we generated 30 test suites for

each database schema using each of the two techniques and while always employing a different random seed [52].

6.1.4 Experimental Procedure

RQ1: Ineffective Mutants Detected by Static Analysis. To answer RQ1, we ran the automated static analysis approach for detecting ineffective mutants on each of the schemas when hosted by every DBMS, recording the numbers of mutants detected for each of the four category types—stillborn, impaired, equivalent, and redundant. Our static analysis procedure follows rules consistent with each DBMS such that false positives are unlikely to occur, unless there are bugs in our implementation of *SchemaAnalyst* or in the database management system itself (see Section 6.2 for more details about how we addressed the former in mitigating the threats to the validity of our experimental study).

Although we judge that *SchemaAnalyst* is capable of identifying large numbers of mutants as ineffective, its checks are not exhaustive, and as such false negatives are still possible—that is, there may be mutant schemas that are ineffective, yet missed by our approach. For stillborn mutants, false negatives are easy to find in the course of standard mutation analysis—any mutant not identified as stillborn is rejected by the DBMS. For equivalent mutants, we manually analyzed the live mutants following mutation analysis, as a mutant that is killed by a test suite cannot be equivalent. For this purpose, a mutant is counted as "live" if it was not killed by *any* test suite. Since the number of live mutants is relatively few in number, an exhaustive manual analysis of these mutants is possible. For impaired and redundant mutants, however, no automated DBMS checks exist, nor is the set of mutants naturally reduced to a tractable number for manual checking through our implemented and tested mutation analysis procedure. Thus, we further checked our results through an intensive manual spot-check of the database schema mutants.

To do so, we selected a subset of non-stillborn mutants produced by *SchemaAnalyst* for each schema and DBMS and manually generated INSERT statements with the aim of checking the classification of each mutant by our tool as either equivalent, redundant, impaired, or normal (i.e., effective). We selected an initial 50 mutants at random. We then added a further seven mutants to this pool to ensure that it contained at least one representative mutant for each schema, DBMS, mutation operator, and ineffective mutant pattern, as detailed in Section 3. Where possible, we selected these additional mutants at random from a constrained set (e.g., all mutants produced by a particular operator or for a schema, if fixing one of these aspects was an important property). Finding an exemplar mutant for Pattern IM-2 (i.e., infeasible CHECK constraints) was less straightforward, however, since we were unaware if such mutants existed in the set of mutants *SchemaAnalyst* generated for our schemas, and if they did, which ones they were. The process we adopted was therefore as follows: We manually reasoned about the 13 subject schemas involving CHECK constraints listed in Table 2, firstly concluding that removing elements of IN expressions through the CInListElementR operator, or complete constraints through the CR operator, would not result in infeasible constraints for

any of these schemas. This left mutants produced by the CRelOpE operator, which was applicable to eight of our schemas (i.e., *BookTown*, *BrowserCookies*, *Employee*, *Examination*, *NistXTS748*, *NistXTS749*, *Products*, and *StudentResidence*), as they involve CHECK constraints with relational expressions. Two of these schemas (i.e., *BrowserCookies* and *Products*) have expressions of sufficient complexity that they could be sources of infeasibility following mutation. The rest involve expressions that simply compare a column with a constant. We performed an exhaustive manual analysis of the mutants produced by CRelOpE for *BrowserCookies* (15 mutants) and *Products* (20 mutants). We found three mutants for *Products* with infeasible CHECKs following mutation and selected one to use in our manual spot-check of mutants.

The first author then produced a JUnit test suite for each of the 57 mutants for our manual spot-check analysis. Each test suite consisted of INSERT statements that could be automatically checked against mutants with the intention of asserting whether that mutant was correctly classified as "effective" or "ineffective", and, if ineffective, what type of ineffective mutant it was. To rule out the possibility of a mutant being "impaired", the first author devised INSERTs to show that data could be added to each table. To eliminate the possibility of manually classifying a mutant as equivalent to the original schema, INSERTs were crafted to show a difference in behavior for the original schema and the selected mutant (that is, a difference in the acceptance/rejection pattern of the INSERTs with the mutant compared to the original schema for the DBMS concerned).

Finally, to rule out the conclusion that the mutant was equivalent to another mutant (i.e., it was redundant) further INSERT statements were written to ensure a difference between the mutant and each other mutant produced for the schema in question. To assist this process, the first author wrote utility methods that could be used by each JUnit test suite to automatically instantiate databases with the original and mutant schemas, submit INSERT statements, and compare the DBMS responses. If we could not construct INSERT statements to refute a particular type of ineffectiveness, the mutant was labeled accordingly, and the manually derived conclusion cross-checked against the mutant's classification as automatically computed by *SchemaAnalyst* using the static detection routines. As noted in Section 6.2, all of the aforementioned test suites, classifications, and crosschecks produced by the first author are available for download from a replication package accompanying this paper.[6]

RQ2: Efficiency of the Approach. To answer RQ2, we split up our analysis to specifically investigate (a) stillborn mutants and (b) impaired, equivalent, and redundant mutants. We treat these two sets of mutants separately since stillborn mutants may also be identified using the database management system, while the other three types of ineffective mutants cannot. If a mutant is stillborn, the DBMS will reject its CREATE TABLE statements. There are no such DBMS checks for impaired, equivalent, and redundant mutants. So, we compare the performance of our algorithms against the use of the DBMS for stillborn mutants, while for the other types of mutants, we compare the execution cost of mutation analysis with and without their inclusion. As background

---

6. https://github.com/schemaanalyst/ineffectivemutants

processes on the workstation could lead to small differences in the timings collected to answer this research question, we always ran 30 repeat trials for each experiment [52].

(a) Stillborn Mutants. For the first part of our investigation, relating specifically to stillborn mutants, we ran three experiments. First, we recorded the time taken to submit each of the CREATE TABLE statements for each mutant for each subject schema to every DBMS (i.e., HyperSQL, PostgreSQL, and SQLite). We then verified whether the DBMS accepted the mutant schema or rejected it as invalid. This particular scenario represents the simplest method of performing mutation analysis for database schemas, since schemas are identified during the process as stillborn by relying on the DBMS to report an error when the schema is invalid.

Secondly, we recorded the time taken to perform the same process, but this time by wrapping the CREATE TABLEs for each mutant schema inside SQL transactions. This represents a potentially faster method of detecting stillborn mutants using the DBMS. Transactions leverage the "roll back" feature of a DBMS to remove any successfully created tables in the event of DBMS rejection of some later CREATE TABLE [53], rather than individually removing the parts of the schema that were successfully created. This is important since all fragments of a schema need to be removed from the DBMS in preparation for the analysis of the next mutant.

Finally, we recorded the time taken to perform the stillborn mutant checking process using *SchemaAnalyst*'s automated static analysis approach. That is, the tool identified stillborn mutants ahead of the mutation analysis process, and then removed them from the mutant pool. Since, to our knowledge, stillborn mutants for database schema integrity constraints cannot be created for SQLite with the operators studied—and as a result there is no need for static analysis checks for this particular DBMS—we ran *SchemaAnalyst* for this analysis with only HyperSQL and PostgreSQL.

(b) Impaired, Equivalent, and Redundant Mutants. To address the second part of the investigation, relating to impaired, equivalent, and redundant mutants, we first studied the time taken to run the detection and removal algorithms with each combination of schema and DBMS to achieve the four sets of mutants introduced in Section 5.3: "$-S$", which corresponds to all mutants that are produced except those identified by the algorithms as stillborn; "$-(S+I)$", which further excludes mutants identified as impaired; "$-(S+I+E)$", which additionally excludes equivalent mutants; and finally "$-(S+I+E+R)$", which also excludes redundant mutants, and as such excludes *all* ineffective mutants found by the algorithms. Figure 9 showed the sequencing of these removals implemented in *SchemaAnalyst*.

To find the times taken to produce every set of ineffective mutants, we timed how long *SchemaAnalyst* took to execute each of the steps 4–6 as described in Section 5.3 and depicted in Figure 9. The time needed to produce the $-(S+I+E+R)$ set of mutants corresponds to the complete time (i.e., for steps 4–6 inclusively). To obtain the time for $-(S+I+E)$, we subtracted the time spent in step 6. For $-(S+I)$, we subtracted the time spent in steps 5 and 6; while for $-S$ the time is zero, since mutation analysis will always take place with stillborn mutants removed, regardless of which of the three different methods studied in part

(a) of this research question is used to remove them.

We ran *SchemaAnalyst* to perform mutation analysis with all non-stillborn mutants produced by its operators (i.e., the set "$-S$"), recording the time taken to evaluate each individual mutant. Then it calculated the time taken to perform mutation analysis for each of the four different sets of mutants (i.e., $-S$, $-(S+I)$, $-(S+I+E)$, and $-(S+I+E+R)$) by summing the evaluation times for each of the mutants in each of those particular sets. We repeated mutation analysis 30 times for each combination of schema, DBMS (i.e., HyperSQL, PostgreSQL, and SQLite) and test generation method (i.e., the *AVM* and *Random$^+$*) using different tests generated with a different random seed, thus minimizing the possibility of random chance, during test generation, affecting the results [52]. To produce a total time to perform mutation analysis with each of the four mutant sets, we added the time to produce each respective set of mutants with the time needed to perform mutation analysis with it.

RQ3: Impact on the Mutation Score. To answer RQ3, we used *SchemaAnalyst* to compute the mutation scores for each of the four sets of mutants (i.e., $-S$, $-(S+I)$, $-(S+I+E)$, and $-(S+I+E+R)$) as evaluated during mutation analysis for the experiments that we conducted to answer RQ2.

We performed all of the experiments with our *SchemaAnalyst* tool [12], [13], [15], [22], as described in Section 5, 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 a dedicated Ubuntu 14.04 workstation, with a 3.13.0-44 GNU/Linux 64-bit kernel, a quad-core 2.4GHz CPU, and 12GB RAM. All input (i.e., relational database schemas) and output (i.e., data files) were stored on the workstation's local disk. We used the default configuration of PostgreSQL version 9.3.5, HyperSQL version 2.2.8, and SQLite 3.8.2. HyperSQL and SQLite were used with "in-memory" mode enabled.

6.1.5 Evaluating the Impact on Timing and Mutation Score

For each experiment, we computed the means of mutation scores and timings, over each of the experiment's 30 repetitions. To gauge the efficiency implications of ineffective mutant removal, we compared the time taken, and the mutation scores obtained, for mutation analysis with and without ineffective mutants. For timing data, where a type of mutant was removed from the mutant pool, we include the time required for the static analyses to run, detect, and remove ineffective mutants, thereby producing the different mutant sets described in Section 5.3 (i.e., "$-S$" and "$-(S+I)$").

Given two sets of data (obtained for either timing or mutation score, one set with an ineffective mutant type and one without), we checked for statistical significance with the Wilcoxon Rank-Sum test, using $p < 0.05$ as the significance threshold [52]. Then, we calculated the Vargha-Delaney $\hat{A}$ statistic to measure effect size, thereby determining the average probability that one approach outperforms another [54]. In the tables of timing data and mutation scores (i.e., Tables 9–14), we annotate large effect sizes (that is, $\hat{A} < 0.29$ or > 0.71) with a "$\star$". Statistically significant decreases are annotated by a "$\triangledown$" symbol, while statistically significant increases are annotated by a "$\triangle$" symbol. If timings are subject to a significant decrease, this means

that the process is more efficient with the removal of the ineffective mutants. Conversely, if timings are subject to a significant increase, the exclusion of ineffective mutants is slower than when it includes them. With mutation scores, a significant increase means that test suites killed a more favorable percentage of mutants following the removal of an ineffective mutant, while a significant decrease indicates that test suites killed a less favorable percentage of mutants.

For assessing the implications of removing stillborn mutants, we only present standard deviations computed for the 30 runs of each experiment, as shown by Table 8. Due to the large differences in the means, and the relatively small standard deviations involved, the trend was clearly evident and thus further statistical analysis was not necessary.

6.2 Threats to Validity

We now detail some threats to the validity of our empirical study, and explain how we sought to mitigate them as part of our experimental design. At the outset, it is important to note that the *SchemaAnalyst* tool and all of the relational database schemas used in this paper's study are available from the tool's web site.[7] The availability of the data generation and mutation analysis tools, in addition to the SQL source code for each schema listed in Table 2, permits both the replication of this paper's experiments and the external confirmation that we correctly controlled many of the threats to validity discussed in this subsection.

Also, all of the data sets and each of the data manipulation, statistical analysis, and table-creation routines—implemented separately by two different authors of the paper in two distinct programming languages—are available for download from the web site for this paper's replication package.[8] Along with supporting the external confirmation that we appropriately controlled some of the validity threats mentioned in the remainder of this subsection, the availability of this replication package enables the recreation of all of the paper's data tables and statistical analyses [55]. In summary, along with releasing all of the software used to arrive at this paper's conclusions, we identified and handled the following validity threats for the experimental study.

- **The schemas used may not generalize.** While it impossible for us to claim that our schemas are representative of all of the characteristics of all possible relational database schemas, the set of subjects we have collected is larger than previously considered [12], [13], [14], [15] and contains schemas drawn from a wide range of sources, including the production systems detailed in Section 6.1.1. Table 2 shows the diversity captured by the 34 schemas that vary in size and their coverage of each of the main types of integrity constraint.

- **The DBMSs used are not representative.** While it is the case that there are some popular DBMSs that we did not include in the experiments, we note that our choice of

DBMSs provides a good coverage of the different design goals (i.e., high performance through in-memory data storage or stability by keeping data on disk) and adherence to the SQL standard of many DBMSs used in practice, as we explained in Section 6.1.2. Although the results may vary for different DBMSs, the patterns observed for other management systems are likely to be similar to those seen for the chosen DBMSs as long as their features are similar—which several recent comparisons suggest is, in fact, largely the case.[9]

- **The test suites used may bias the results.** To ensure a diverse set of tests in each of our test suites, we chose to generate test suites with two different test data generation techniques—the *AVM* and *Random*+— with their differing approaches to obtaining coverage, as explained in Section 6.1.3. These two methods are stochastic, and so further diversity can be achieved by repeating experiments using a different random seed, which we did for each experiment and test data generator. Finally, since none of the chosen database schemas were accompanied by tests, we could not study how these types of test influenced the detection and removal of ineffective mutants; Section 8 notes that this may be a promising area for future work as more database designers start to test relational schemas.

- **The mutation operators may not generalize.** Since prior work has shown that real-world relational schemas are complex and often include features such as composite keys and multi-column foreign-key relationships [10], our operators specifically target these aspects of relational database schemas. Yet, different results may be obtained with different types of operators, and our results may not generalize to those operators. For instance, this paper does not focus on the identification and removal of ineffective higher-order mutants. However, Section 8 notes that we plan, as part of future work, to further control this threat by extending the set of mutation operators used by the *SchemaAnalyst* tool.

- **The mutants are not representative of real faults.** According to the "competent programmer" hypothesis [56], programmers are likely to produce programs that are nearly correct, implying that real faults will frequently be the result of small mistakes. By making small changes to each type of constraint, the mutation operators that we used were designed to model such faults in the context of relational database schemas. They implement operators for both the addition and removal of columns, and as such model faults of both omission and commission, further improving the range of mistakes in database schema that they can represent.

- **Background tasks interfering with timings.** The timing of the processes for detecting and removing ineffective mutants, and performing the mutation analysis itself, are subject to interference from background tasks. To minimize the impact of background tasks, we repeated the experiments and recorded all timings.

---

7. The web site https://github.com/schemaanalyst/schemaanalyst features a Git version control repository containing all of the relational database schemas used in the experiments in addition to the documented version of *SchemaAnalyst*'s source code and test suite.

8. Along with providing the source code for all of the data analysis and manipulation routines and all of the raw data files, https://github.com/schemaanalyst/ineffectivemutants also furnishes the manually created JUnit tests used in answering RQ1.

9. The web site available at http://goo.gl/7pzxeV provides a regularly-updated "DBMS comparison" table revealing that different database management systems now offer many of the same features.

- **Defects in the** *SchemaAnalyst* **tool.** To mitigate this threat, we have implemented a JUnit test suite in parallel with the development of the *SchemaAnalyst* tool itself. Furthermore, we have extensively hand checked the results obtained to ensure that they are correct. In addition, as part of the methodology of the experiments, we manually-checked the classification of 57 mutants, further confirming the tool's correctness.

- **Mistakes made as part of the manual analysis.** Although the INSERT statements and JUnit tests used to find false positives in RQ1 of the experimental study were manually written by an author of this paper, they were automatically checked against the behavior of the DBMS and other mutants, and in each case our conclusions agreed with the result produced by the static analysis algorithms for ineffective mutant detection.

- **The statistical tests used.** We cannot be certain that our data is normally distributed, and as a result, we used non-parametric statistical tests, including the Wilcoxon Rank-Sum (Mann-Whitney U) Test and the Vargha-Delaney $\hat{A}$ statistic for measuring effect size. These two statistical tests are commonly adopted for analysing results arising from the study of software engineering methods that employ randomness [52], thereby mitigating concerns that our conclusions are incorrect.

- **Defects in the statistical analysis tools.** Since it is possible that we made a mistake during the manipulation and statistical analysis of the empirical results, we took several steps to control this threat to validity. For instance, two authors of this paper separately implemented the data analysis routines and then compared the outputs, ultimately finding agreement in the final data tables and outcomes of the statistical tests.

6.3 Characterizing the Test Suites

Since none of the database schemas came with a test suite, we automatically generated tests using the *AVM* and *Random*$^+$ methods provided by the *SchemaAnalyst* tool. Table 3 characterizes the test suites created by both of these techniques, revealing that it is rare for the *AVM* to not achieve full coverage of the test requirements. In fact, in cases where *AVM* does not cover all of the requirements, we found that this was due to an infeasibility in the constraints that the test data generator must cover. This table also shows that test suites created by *AVM* are of a higher coverage—and often comprised of more tests—than those created by *Random*$^+$, thereby suggesting that they will also have a higher mutation score and a longer mutation analysis time than those that are produced by the random method.

6.4 Answers to Research Questions

RQ1: Ineffective Mutants Detected by Static Analysis

Table 4 shows the number of mutants produced for each database schema, and the ineffective mutants identified for each. Table 5 breaks the data down by mutation operator.

The number of mutants produced for each database schema depends on the number and type of integrity constraints it has, all information that is shown in Table 2. If a type of integrity constraint is not present for a schema,

then certain operators cannot be applied (e.g., the NNR operator cannot be used to produce mutants with NOT NULL constraints removed, if there are no NOT NULL constraints in the first instance). Certain types of integrity constraints will yield more mutants with certain operators than others. For example, the PKColumnA operator will produce four different mutants for a table with a single column primary key with five columns, where each mutant is the original primary key with another column in the table added to it.

Responding to this research question, we now discuss the results for each of the types of ineffective mutant.

Stillborn Mutants

The tables show that *SchemaAnalyst*'s use of the abstract representation and static analysis checks leads to the identification of many stillborn mutants for the 34 schemas and the HyperSQL and PostgreSQL DBMSs. For stillborn mutants, it is possible to automatically verify the results by submitting each mutant to the DBMS and checking to see if was rejected as invalid. This process confirmed that the static analysis correctly identified all stillborn mutants. The stillborn mutants found are identified as a result of the foreign key misalignment rule (as discussed in Section 3.3) and the rule for HyperSQL that detects PRIMARY KEY constraints and UNIQUE constraints with identical column sets, which are disallowed for this DBMS. Since neither issue affects SQLite, no stillborn mutants were found for this DBMS.

Table 4 shows that the *NistDML182* schema resulted in the most stillborn mutants with the mutation operators. This schema has one foreign key with 15 columns, leading to many instances of foreign key misalignment when *SchemaAnalyst* applies the mutation operators. The *RiskIt*, *UnixUsage*, and *CustomerOrder* schemas also have high numbers of stillborn mutants. As Table 2 shows, these schemas have the highest number of foreign keys (10, 7, and 7, respectively), which again causes foreign key misalignment. As seen from Table 5, approximately 80% of mutants produced by the FKColumnPairE operator are stillborn. In order to create a mutant that maintains a correctly aligned foreign key, this operator has to exchange a column where the modified column set in the referenced table corresponds to an existing PRIMARY KEY or UNIQUE constraint so that the mutant is valid. However, this happened relatively infrequently. Moreover, approximately a third of mutants are stillborn for the FKColumnPairR operator. Since FKColumnPairR removes a pair of columns from the foreign key (i.e., a column in the foreign key table and its associated column in the referenced table), valid mutants tend not to be produced except for when the foreign key involves a single column pair, in which case the entire constraint is removed.

Impaired mutants

For SQLite, mutant schemas with foreign key misalignment are impaired, rather than stillborn. Accordingly, many mutants that would have been classified as stillborn for HyperSQL and PostgreSQL are identified as impaired for SQLite. An example of this phenomenon is given in Table 5, where, for SQLite, operators like FKColumnPairE and PKColumnA, and PKColumnE produce no stillborn mutants and, respectively, 415, 102, and 111 impaired mutants. Also, there are fewer impaired mutants for SQLite than

TABLE 3
Mean coverage and size of the test suites that were automatically generated by *Random+* and the *AVM*
In this table, the abbreviation "Cov." stands for the higher-is-better coverage score of the suite created by the test generator, while "# Tests" is the number of test cases.

| | HyperSQL | | | | PostgreSQL | | | | SQLite | | | |
| | Random+ | | AVM | | Random+ | | AVM | | Random+ | | AVM | |
Schema	Cov.	# Tests	Cov.	# Tests	Cov.	# Tests	Cov.	# Tests	Cov.	# Tests	Cov.	# Tests
ArtistSimilarity	59.3	10.7	100.0	18.0	59.3	10.7	100.0	18.0	61.2	11.7	100.0	19.0
ArtistTerm	59.5	23.9	100.0	40.0	59.5	23.9	100.0	40.0	62.0	26.9	100.0	43.0
BankAccount	84.6	26.4	100.0	31.0	84.6	26.4	100.0	31.0	87.0	32.4	100.0	37.0
BookTown	91.8	247.4	99.0	266.0	91.8	247.4	99.0	266.0	91.8	250.4	99.0	269.0
BrowserCookies	57.8	42.0	100.0	72.0	57.8	42.0	100.0	72.0	58.6	42.0	100.0	71.0
Cloc	92.1	36.9	100.0	40.0	92.1	36.9	100.0	40.0	92.1	36.9	100.0	40.0
CoffeeOrders	57.1	44.4	100.0	77.0	57.1	44.4	100.0	77.0	61.6	55.7	100.0	90.0
CustomerOrder	41.1	49.9	100.0	120.0	41.1	49.9	100.0	120.0	41.3	52.7	100.0	126.0
DellStore	93.0	165.4	100.0	177.0	93.0	165.4	100.0	177.0	93.0	165.4	100.0	177.0
Employee	88.8	31.3	100.0	35.0	88.8	31.3	100.0	35.0	89.9	34.3	100.0	38.0
Examination	82.4	85.4	100.0	103.0	82.4	85.4	100.0	103.0	83.3	89.6	100.0	107.0
Flights	58.4	38.8	100.0	66.0	58.4	38.8	100.0	66.0	57.8	36.1	100.0	62.0
FrenchTowns	34.1	18.4	100.0	53.0	34.1	18.4	100.0	53.0	34.1	18.4	100.0	53.0
Inventory	95.3	15.3	100.0	16.0	95.3	15.3	100.0	16.0	96.0	17.3	100.0	18.0
Iso3166	84.5	7.7	100.0	9.0	84.5	7.7	100.0	9.0	88.5	10.7	100.0	12.0
IsoFlav_R2	87.1	155.0	100.0	177.0	87.1	155.0	100.0	177.0	87.1	155.0	100.0	177.0
iTrust	91.1	1334.8	100.0	1458.0	91.1	1334.8	100.0	1458.0	91.6	1395.4	100.0	1517.0
JWhoisServer	85.9	131.5	100.0	152.0	85.9	131.5	100.0	152.0	86.5	137.5	100.0	158.0
MozillaExtensions	87.3	198.5	100.0	226.0	87.3	198.5	100.0	226.0	87.5	201.5	100.0	229.0
MozillaPermissions	95.3	30.7	100.0	32.0	95.3	30.7	100.0	32.0	95.3	31.7	100.0	33.0
NistDML181	63.1	23.6	100.0	37.0	63.1	23.6	100.0	37.0	64.2	24.7	100.0	38.0
NistDML182	62.0	110.0	100.0	176.0	62.0	110.0	100.0	176.0	65.0	124.0	100.0	190.0
NistDML183	100.0	34.0	100.0	34.0	100.0	34.0	100.0	34.0	100.0	34.0	100.0	34.0
NistWeather	56.8	29.8	100.0	52.0	56.8	29.8	100.0	52.0	75.2	42.3	100.0	56.0
NistXTS748	100.0	16.0	100.0	16.0	100.0	16.0	100.0	16.0	100.0	16.0	100.0	16.0
NistXTS749	85.0	31.7	100.0	37.0	85.0	31.7	100.0	37.0	85.9	30.2	100.0	35.0
Person	92.8	17.7	100.0	19.0	92.8	17.7	100.0	19.0	93.7	18.7	100.0	20.0
Products	69.3	32.8	97.0	46.0	69.3	32.8	97.0	46.0	78.3	41.9	98.0	52.0
RiskIt	67.9	167.8	100.0	245.0	67.9	167.8	100.0	245.0	69.7	175.2	100.0	250.0
StackOverflow	95.5	164.2	100.0	171.0	95.5	164.2	100.0	171.0	95.5	164.2	100.0	171.0
StudentResidence	70.0	21.0	100.0	30.0	70.0	21.0	100.0	30.0	73.2	25.1	100.0	34.0
UnixUsage	49.6	73.6	100.0	147.0	49.6	73.6	100.0	147.0	51.8	76.6	100.0	147.0
Usda	89.6	222.4	100.0	247.0	89.6	222.4	100.0	247.0	89.6	222.4	100.0	247.0
WordNet	89.5	114.4	100.0	127.0	89.5	114.4	100.0	127.0	88.6	105.5	100.0	118.0

```
(1)   CHECK (price > 0)
(2)   CHECK (discounted_price > 0)
(3)   CHECK (price > discounted_price)
```

Fig. 10. CHECK constraints of the *Products* schema

stillborn mutants for HyperSQL and PostgreSQL because SQLite does not regard mutants with UNIQUE constraints and PRIMARY KEYs on the same sets of columns as invalid, as does HyperSQL, and type mismatches between columns in mutated, yet correctly-aligned, FOREIGN KEYs are of no concern for this DBMS. (As discussed in Section 3.3, SQLite has a weak typing mechanism where, for example, a column of type TEXT can form a foreign key with a column of type INTEGER.) Since there is no need for static checks for impaired mutants with HyperSQL and PostgreSQL, no impaired mutants were identified for these DBMSs.

Yet, our manual analysis did reveal impaired mutants that escaped the automated analysis. For the mutants of schemas with CHECKs, we found three that had constraints that were infeasible, and are therefore impaired. Involving *Products*, these mutants were produced by CRelOpE, which is responsible for changing the relational operator in a CHECK. This database schema has a table involving the three CHECK constraints shown by Figure 10. The operator mutated the first constraint to price = 0, price < 0, and price <= 0, respectively. Because of the second constraint, mandating that discounted_price be greater than zero, the third constraint price > discounted_price can never be true.

It is worth noting that our manual analysis involved an exhaustive search for mutants with infeasible CHECK constraints, and resulted in us finding only three mutants—a very small percentage of the total number of mutants produced for the subject schemas. As reported in Tables 4 and 5, a total of 5223 mutants were produced, meaning that only $\frac{3}{5233} = 0.06\%$ of mutants escaped the automated analysis.

Nevertheless, as discussed in Section 8, future work will automatically identify cases of infeasibility in CHECKs and remove them from the subsequent mutation analysis.

Equivalent mutants

Tables 4 and 5 show that a significant number of mutants were identified as equivalent to the original schema by the automated analysis. Following the removal of stillborn and impaired mutants from the total of 5223 mutants produced for all subject schemas, *SchemaAnalyst* identified 162 (3%), 271 (5%), and 115 (2%) of these as equivalent for the HyperSQL, PostgreSQL, and SQLite DBMSs, respectively.

Using the mutation operators with SQLite results in the fewest equivalent mutants of all three DBMSs detected by the static analysis checks (115 mutants). As shown by Table 5, the NNA and NNR operators do not produce equivalent mutants for SQLite—even though they do so with PostgreSQL and HyperSQL. This is because of the difference in PRIMARY KEY behavior between SQLite and the other two DBMSs. For PostgreSQL and HyperSQL, a NOT NULL can be added to or removed from a column that is already part of a PRIMARY KEY, and it will have no effect on the behavior of the PRIMARY KEY constraint, since for these DBMSs, primary key columns also have an implicit NOT NULL defined on them. As such, the NNA and NNR operators produce mutants that are indistinguishable in behavior from the original schema. However, for SQLite, values in primary key columns may be NULL, so adding and removing NOT NULL constraints on these columns changes the behavior of the schema.

As shown in Table 4, the number of equivalent mutants detected for HyperSQL (162 mutants) is lower than that for PostgreSQL (271 mutants). This phenomenon is evident because many mutants that are equivalent for the PostgreSQL DBMS are stillborn for HyperSQL, and thus they were previously removed from the mutant pool. This set of

TABLE 4
Ineffective mutants by database schema

In this table, "Produced" is the number of mutants produced for a relational database schema, while "Stillborn", "Impaired", "Equivalent", and "Redundant" indicate the numbers of these mutants that are ineffective by their differing type. The "Ineffective" column denotes the total number of ineffective mutants, while "Effective" indicates the total number of remaining effective mutants. Finally, the "Reduction" columns indicates the overall reduction in the number of mutants needed for mutation analysis following the removal of ineffective mutants. In this table, "H" = HyperSQL, "P" = PostgreSQL, and "S" = SQLite.

Schema		Produced	Stillborn	Impaired	Equivalent	Redundant	Ineffective	Effective	Reduction (%)
ArtistSimilarity	H	13	2	0	1	2	5	8	38.5
	P	13	1	0	2	2	5	8	38.5
	S	13	0	1	1	4	6	7	46.2
ArtistTerm	H	29	6	0	3	0	9	20	31.0
	P	29	3	0	6	0	9	20	31.0
	S	29	0	3	3	4	10	19	34.5
BankAccount	H	42	14	0	2	0	16	26	38.1
	P	42	12	0	4	0	16	26	38.1
	S	42	0	10	2	0	12	30	28.6
BookTown	H	235	11	0	13	2	26	209	11.1
	P	235	0	0	24	2	26	209	11.1
	S	235	0	0	13	29	42	193	17.9
BrowserCookies	H	114	30	0	3	3	36	78	31.6
	P	114	29	0	4	3	36	78	31.6
	S	114	0	21	1	3	25	89	21.9
Cloc	H	30	0	0	0	0	0	30	0.0
	P	30	0	0	0	0	0	30	0.0
	S	30	0	0	0	10	10	20	33.3
CoffeeOrders	H	101	45	0	5	0	50	51	49.5
	P	101	40	0	10	0	50	51	49.5
	S	101	0	40	5	0	45	56	44.6
CustomerOrder	H	183	92	0	7	0	99	84	54.1
	P	183	87	0	14	0	101	82	55.2
	S	183	0	71	7	0	78	105	42.6
DellStore	H	156	0	0	0	39	39	117	25.0
	P	156	0	0	0	39	39	117	25.0
	S	156	0	0	0	52	52	104	33.3
Employee	H	45	1	0	1	0	2	43	4.4
	P	45	0	0	2	0	2	43	4.4
	S	45	0	0	1	0	1	44	2.2
Examination	H	138	26	0	2	0	28	110	20.3
	P	138	24	0	4	0	28	110	20.3
	S	138	0	16	2	0	18	120	13.0
Flights	H	84	36	0	4	2	42	42	50.0
	P	84	36	0	4	2	42	42	50.0
	S	84	0	31	0	2	33	51	39.3
FrenchTowns	H	128	30	0	0	35	65	63	50.8
	P	128	22	0	8	35	65	63	50.8
	S	128	0	18	8	36	62	66	48.4
Inventory	H	21	3	0	1	1	5	16	23.8
	P	21	0	0	2	2	4	17	19.0
	S	21	0	0	1	4	5	16	23.8
Iso3166	H	11	1	0	1	0	2	9	18.2
	P	11	0	0	2	0	2	9	18.2
	S	11	0	0	1	0	1	10	9.1
IsoFlav_R2	H	219	3	0	0	0	3	216	1.4
	P	219	0	0	0	3	3	216	1.4
	S	219	0	0	3	37	40	179	18.3
iTrust	H	1458	51	0	44	22	117	1341	8.0
	P	1458	21	0	74	22	117	1341	8.0
	S	1458	0	19	30	46	95	1363	6.5

Schema		Produced	Stillborn	Impaired	Equivalent	Redundant	Ineffective	Effective	Reduction (%)
JWhoisServer	H	190	6	0	6	0	12	178	6.3
	P	190	0	0	12	0	12	178	6.3
	S	190	0	0	6	0	6	184	3.2
MozillaExtensions	H	364	4	0	2	12	18	346	4.9
	P	364	0	0	4	13	17	347	4.7
	S	364	0	0	2	28	30	334	8.2
MozillaPermissions	H	31	1	0	1	0	2	29	6.5
	P	31	0	0	2	0	2	29	6.5
	S	31	0	0	1	0	1	30	3.2
NistDML181	H	33	13	0	2	0	15	18	45.5
	P	33	13	0	2	0	15	18	45.5
	S	33	0	11	0	4	15	18	45.5
NistDML182	H	351	255	0	15	0	270	81	76.9
	P	351	270	0	15	0	285	66	81.2
	S	351	0	240	0	17	257	94	73.2
NistDML183	H	31	11	0	0	0	11	20	35.5
	P	31	11	0	0	0	11	20	35.5
	S	31	0	11	0	6	17	14	54.8
NistWeather	H	48	14	0	3	1	18	30	37.5
	P	48	13	0	4	1	18	30	37.5
	S	48	0	13	1	2	16	32	33.3
NistXTS748	H	19	1	0	0	1	2	17	10.5
	P	19	0	0	0	2	2	17	10.5
	S	19	0	0	1	2	3	16	15.8
NistXTS749	H	38	13	0	3	2	18	20	47.4
	P	38	12	0	4	2	18	20	47.4
	S	38	0	10	1	2	13	25	34.2
Person	H	23	1	0	1	0	2	21	8.7
	P	23	0	0	2	0	2	21	8.7
	S	23	0	0	1	0	1	22	4.3
Products	H	67	16	0	4	0	20	47	29.9
	P	67	14	0	6	0	20	47	29.9
	S	67	0	14	2	2	18	49	26.9
RiskIt	H	347	148	0	12	4	164	183	47.3
	P	347	138	0	22	4	164	183	47.3
	S	347	0	123	10	8	141	206	40.6
StackOverflow	H	129	0	0	0	5	5	124	3.9
	P	129	0	0	0	5	5	124	3.9
	S	129	0	0	0	43	43	86	33.3
StudentResidence	H	45	7	0	2	0	9	36	20.0
	P	45	5	0	4	0	9	36	20.0
	S	45	0	4	2	0	6	39	13.3
UnixUsage	H	192	98	0	8	3	109	83	56.8
	P	192	92	0	14	3	109	83	56.8
	S	192	0	67	6	7	80	112	41.7
Usda	H	201	0	0	0	31	31	170	15.4
	P	201	0	0	0	31	31	170	15.4
	S	201	0	0	0	67	67	134	33.3
WordNet	H	107	6	0	16	6	28	79	26.2
	P	107	0	0	20	8	28	79	26.2
	S	107	0	0	4	8	12	95	11.2
Total	H	5223	945	0	162	171	1278	3945	24.5
	P	5223	843	0	271	179	1293	3930	24.8
	S	5223	0	723	115	423	1261	3962	24.1

mutants corresponds to schemas where a PRIMARY KEY and a UNIQUE constraint involve an identical set of columns, and, as Table 5 shows, is largely the result of the UColumnA operator, where a column is added to an existing UNIQUE constraint or a new single-column UNIQUE is created that is identical to the database table's primary key.

With the goal of finding equivalent mutants that were not detected by our static analysis approach, we investigated mutants not killed following all of the mutation analysis runs, in adherence to the methodology detailed in Section 6.1.4. Table 6 summarizes this data, showing the numbers of remaining live mutants for each schema and operator after mutation analysis with each of the three chosen DBMSs. We manually studied each live mutant to try and ascertain whether it was a genuine equivalent mutant that

was missed by our automated analysis, or whether the test suites used had simply failed to kill it. Following this investigation, we found that only three mutants were genuinely equivalent for each of the DBMSs. The first equivalent mutant is the one produced by the CR operator for *Products*, as listed in Table 6. This operator mutated the CHECK constraints shown by Figure 10. For this schema, the first constraint is actually superfluous, since price must be greater than zero, if, according to constraint (2), discounted_price is greater than zero, and price must be greater than discounted_price as per constraint (3). Therefore, when the CR operator produces a mutant by removing constraint (1), the mutant is equivalent to the original. Two further equivalent mutants occur with the *Products* schema and CRelOpE, accounting for two of the three mutants listed for CRelOpE in Table 6.

TABLE 5
Ineffective mutants by mutation operator
(Please refer to Table 4 for a description of each heading)

Operator		Produced	Stillborn	Impaired	Equivalent	Redundant	Ineffective	Effective	Reduction (%)
CInListElementR	H	282	0	0	0	0	0	282	0.0
	P	282	0	0	0	0	0	282	0.0
	S	282	0	0	0	0	0	282	0.0
CR	H	38	0	0	0	0	0	38	0.0
	P	38	0	0	0	0	0	38	0.0
	S	38	0	0	0	0	0	38	0.0
CRelOpE	H	110	0	0	0	0	0	110	0.0
	P	110	0	0	0	0	0	110	0.0
	S	110	0	0	0	0	0	110	0.0
FKColumnPairE	H	643	518	0	0	0	518	125	80.6
	P	643	535	0	0	0	535	108	83.2
	S	643	0	415	0	0	415	228	64.5
FKColumnPairR	H	67	23	0	0	2	25	42	37.3
	P	67	23	0	0	2	25	42	37.3
	S	67	0	23	0	2	25	42	37.3
NNA	H	687	0	0	71	0	71	616	10.3
	P	687	0	0	71	0	71	616	10.3
	S	687	0	0	2	0	2	685	0.3
NNR	H	357	0	0	91	0	91	266	25.5
	P	357	0	0	91	0	91	266	25.5
	S	357	0	0	0	0	0	357	0.0
PKColumnA	H	884	114	0	0	99	213	671	24.1
	P	884	102	0	8	103	213	671	24.1
	S	884	0	102	12	338	452	432	51.1
PKColumnE	H	568	114	0	0	0	114	454	20.1
	P	568	111	0	0	0	111	457	19.5
	S	568	0	111	0	0	111	457	19.5
PKColumnR	H	160	51	0	0	23	74	86	46.2
	P	160	51	0	0	24	75	85	46.9
	S	160	0	51	0	33	84	76	52.5
UColumnA	H	1167	109	0	0	28	137	1030	11.7
	P	1167	8	0	0	28	137	1030	11.7
	S	1167	0	8	101	28	137	1030	11.7
UColumnE	H	223	12	0	0	14	26	197	11.7
	P	223	9	0	0	14	23	200	10.3
	S	223	0	9	0	14	23	200	10.3
UColumnR	H	37	4	0	0	5	9	28	24.3
	P	37	4	0	0	8	12	25	32.4
	S	37	0	4	0	8	12	25	32.4
Total	H	5223	945	0	162	171	1278	3945	24.5
	P	5223	843	0	271	179	1293	3930	24.8
	S	5223	0	723	115	423	1261	3962	24.1

TABLE 6
Live mutants following mutation analysis with all test suites

A mutant is said to be "live" if it was not killed by any of the test suites generated by *SchemaAnalyst* as part of the experimental study. In this table, "H" = HyperSQL, "P" = PostgreSQL, and "S" = SQLite.

Schema	CR			CRelOpE			PKColumnA			UColumnA			Total		
	H	P	S	H	P	S	H	P	S	H	P	S	H	P	S
BankAccount	0	0	0	0	0	0	3	3	4	0	0	0	3	3	4
BrowserCookies	0	0	0	1	1	1	0	0	6	4	4	4	5	5	11
CoffeeOrders	0	0	0	0	0	0	0	0	3	0	0	0	0	0	3
CustomerOrder	0	0	0	0	0	0	4	4	4	0	0	0	4	4	4
FrenchTowns	0	0	0	0	0	0	0	0	0	6	6	6	6	6	6
iTrust	0	0	0	0	0	0	2	2	1	0	0	0	2	2	1
NistWeather	0	0	0	0	0	0	2	2	2	0	0	0	2	2	2
Products	1	1	1	2	2	2	0	0	0	0	0	0	3	3	3
RiskIt	0	0	0	0	0	0	0	0	9	0	0	0	0	0	9
UnixUsage	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1
WordNet	0	0	0	0	0	0	2	2	4	0	0	0	2	2	4
Total	1	1	1	3	3	3	13	13	34	10	10	10	27	27	48

The first mutant changes the expression of CHECK constraint (1) to `price != 0` while the second changes it to `price >= 0`. Again, these constraints add nothing further to constraints (2) and (3), and are thus equivalent to the original schema.

Manual analysis of the remaining live mutants revealed that the automatically generated test suites were incapable of distinguishing each of these mutants from their corresponding original schema. That is, the mutants were in theory killable by a test suite, and they were not actually equivalent. This is a shortcoming of the generated test suites and not the technique for detecting equivalent mutants. We refer the reader to our prior work on test data generation for relational database schemas [13] for a discussion of why the test suites generated with the chosen coverage criteria cannot kill all of the mutants that the operators produce.

In summary, the automated analysis approach detects a significant number of equivalent mutants. However, due to the arbitrary nature of CHECK constraints, some mutants related to this type of constraint are not detected. As for impaired mutants, this is a small number (i.e., 3 of 5223 mutants), and is related to the fact that the current implementation does not analyze CHECK constraints. As mentioned in Section 8, we will address this issue in future work.

Redundant mutants

Tables 4 and 5 show the number of schema mutants found to be redundant using the automated analysis. These tables show that, following the removal of stillborn, impaired, and equivalent mutants from the initial total of 5223 mutants produced for all subject schemas, *SchemaAnalyst* identified 171 (3%), 179 (3%) and 423 (7%) of these as redundant for the HyperSQL, PostgreSQL, and SQLite DBMSs, respectively.

In practice, redundant mutants can be caused by the same or two different operators producing two identical mutants. One mutant is kept, while the other is removed from the mutant pool. Table 5 lists the mutants removed according the operator that produced them. Table 7 gives another view of redundant mutants, showing the pairs of operators that were responsible for producing the identical mutant pairs. The operators listed on each row are the ones that produced the mutant that was removed, while the operators listed on each column are those that produced the identical mutant that *SchemaAnalyst* retained. The cells of the table contain numbers of identical mutant pairs produced by each pair of operators for a particular DBMS.

Tables 4 and 5 show that more redundant mutants were produced for SQLite than for HyperSQL and PostgreSQL. Table 7 reveals that this was due to the overlapping effects of PKColumnA and UColumnA and the fact that SQLite does not force primary key values to also not be NULL. As such, these two operators can produce schemas with the same behavior for this DBMS. Since HyperSQL and PostgreSQL do require primary key values to not be NULL, the same effect does not occur, except when primary key columns are also declared as NOT NULL—that is, the addition of either a PRIMARY KEY or a UNIQUE constraint to a column would have had an identical effect in terms of the schema's behavior.

Table 7 shows that redundant mutants can be placed into two categories: redundant mutants that were caused by two operators that mutate the (a) same or (b) a different type of integrity constraints. In category (a) are redundant mutants caused by operators that add, remove, and exchange columns from foreign keys, primary keys, and UNIQUE constraints have overlapping effects that are detected using the automated analysis. An example of this can occur when a table has two single-column UNIQUE constraints. The column of the first constraint is exchanged with that of the second by the UColumnE operator, which effectively removes the first constraint. This replicates UColumnR when it produces a mutant that removes the same constraint, resulting in behaviorally identical schemas. Yet, these mutants are a small proportion of the original pool of 5223, with the exact numbers depending on the DBMS used in each case.

TABLE 7
Redundant mutants by pairs of operators that produced them
In this table, "H" = HyperSQL, "P" = PostgreSQL, and "S" = SQLite.

Operator		FKColumnPairE	NNA	PKColumnE	PKColumnR	UColumnA	UColumnE	UColumnR
FKCColumnPairR	H	2	0	0	0	0	0	0
	P	2	0	0	0	0	0	0
	S	2	0	0	0	0	0	0
PKCColumnA	H	0	0	0	1	98	0	0
	P	0	4	0	1	98	0	0
	S	0	0	0	3	335	0	0
PKCColumnR	H	0	0	0	0	23	0	0
	P	0	0	1	0	23	0	0
	S	0	0	3	0	30	0	0
UCColumnA	H	0	0	0	0	9	3	16
	P	0	0	0	0	9	3	16
	S	0	0	0	0	9	3	16
UCColumnE	H	0	0	0	0	0	14	0
	P	0	0	0	0	0	14	0
	S	0	0	0	0	0	14	0
UCColumnR	H	0	0	0	0	0	5	0
	P	0	0	0	0	0	8	0
	S	0	0	0	0	0	8	0

In category (b) are redundant mutants generated by PKColumnR and UColumnA. Again, these mutants happen in relatively rare situations (i.e., 23 to 30 mutants, depending on the DBMS), as Table 7 indicates. An example of such a situation is when a mutant is produced that removes a column from a multi-column primary key on columns (A, B) with PKColumnR, making it a primary key on just A. A behaviorally identical mutant is produced with UColumnA by adding a UNIQUE constraint to the column A. Although this mutant still has the primary key on (A, B), it now behaves the same as the UNIQUE constraint on A, for the reasons explained in Section 3's presentation of Pattern BE-4. Other mutants in this category are created by PKColumnA and NNA for PostgreSQL. This occurs when the operators add a primary key and a NOT NULL, respectively, to a column already declared as UNIQUE. (For HyperSQL, adding a primary key to a UNIQUE column makes it stillborn, whereas for SQLite, primary keys are not also required to be not NULL—hence the two schemas have different behaviors for this DBMS).

Finally, our careful manual analysis of the 57 mutants chosen to verify *SchemaAnalyst*'s automated approach revealed no redundant mutants not already found by our tool.

Conclusion for RQ1

The automated static analyses detected many ineffective mutants, the majority of which are stillborn or impaired, accounting for as many as 18% of mutants with HyperSQL. The tool also detected significant numbers of equivalent and redundant mutants. Our manual analysis of these mutants revealed that there were some ineffective mutants that our approach did not detect, but that they were relatively few in number and were associated with complex CHECKs. Analysis of arbitrary constraints for infeasibility and equivalence is undecidable in general [40], [44], [45], rendering these types of ineffective mutant hard to detect automatically. Yet, if the schema is free of CHECKs, the static analyses can reliably detect all ineffective mutants. If the schema does involve CHECKs, then some ineffective mutants may be missed, particularly where the constraints are relational expressions and there are multiple constraints involving the same columns, both of which contributed to mutants with infeasible constraints when applying the operators. Overall,

our technique was able to identify approximately 24% of mutants as ineffective, regardless of the DBMS being used.

RQ2: Efficiency of the Approach

Stillborn Mutants

Table 8 shows the times taken for each of the three methods devised to identify stillborn mutants for HyperSQL and PostgreSQL (i.e., "DBMS", "DBMS-Transacted", and "Static"), as detailed by the methodology described in Section 6.1.4. Since no stillborn mutants are produced by our operators for SQLite—as confirmed by the answer to the last research question—there is no need for static analysis checks, and thus there are no results to report for this DBMS.

This table shows that using a DBMS is significantly more time consuming than using static analysis (in fact, due to the clarity of this result, we do not furnish a statistical analysis of these data points). This is the case even when attempting to use the DBMS to check schemas as efficiently as possible, by wrapping CREATE TABLE statements in transactions. Static analysis only takes a fraction of a second for any of the schema and DBMS combinations. For HyperSQL, it requires two milliseconds or less for four schemas (i.e., *Cloc*, *DellStore*, *StackOverflow*, and *Usda*) and, with PostgreSQL, less than one millisecond for 15 schemas (i.e., for just under half of the schemas). The longest time was recorded for *NistDML182* with HyperSQL, at just 265 milliseconds. In contrast, relying on the DBMS to reject schemas takes several orders of magnitude longer, with the use of transactions only marginally decreasing the time overhead. The longest time recorded is with the non-transacted method for *iTrust*, which requires over eight seconds to process with HyperSQL, and over one hour with PostgreSQL. As Table 8 shows, more processing time was required for the database schemas when used in conjunction with PostgreSQL as opposed to HyperSQL, whether it be with the DBMS method (i.e., non-transacted) or with the DBMS-Transacted version.

Further analysis of each technique's longest processing times sheds light on how schema characteristics influence running time. For the static analysis checks, the schemas with the longest processing times (for HyperSQL and PostgreSQL, respectively) are *NistDML182* (257ms and 265ms), *RiskIt* (208ms and 199ms), *UnixUsage* (149ms and 141ms), and *CustomerOrder* (144ms and 138ms). These are also the schemas with the greatest number of foreign keys (c.f. Table 2) or the most complex foreign key relationships—as discussed in the answer to the last research question—and which, therefore, require the most foreign key misalignment checks needed for detecting the stillborn mutants.

For the DBMS-based methods, the schemas with lengthy processing times tend to be those that are the largest and/or result in the greatest number of mutants—that is, the ones that will require the most setup on the host DBMS and/or the most DBMS-based validity checks. For example, *iTrust*, *MozillaExtensions*, *RiskIt*, and *BookTown* produce the longest times for the (non-transacted) DBMS method with PostgreSQL (46, 2.9, 2.6, and 2.4 $\times 10^5$ ms respectively). These are the schemas that produce the most mutants (first is *iTrust*, with 1458 mutants; second is *MozillaExtensions* with 364, as shown by Table 4), or are the largest in terms of the number of tables (first is *iTrust*, with 42 tables; second is *BookTown*

TABLE 8
Mean times taken to detect stillborn database schema mutants (in milliseconds)

This table compares the times taken to detect stillborn mutants by using the DBMS (i.e., the columns labeled "DBMS") versus using the static analysis approach (i.e., the columns labeled "Static"), which makes a series of checks on each mutated schema. The data for the columns labeled "DBMS-Transacted" are for the approach that also uses the DBMS to identify stillborn mutants, but groups SQL statements in transactions in order to speed up the process. "SD" refers to the standard deviation for each set of 30 times recorded. Since our mutation operators do not produce stillborn mutants with SQLite, only figures for HyperSQL and PostgreSQL are recorded.

| | HyperSQL | | | | | | PostgreSQL | | | | | |
| | DBMS | | DBMS-Transacted | | Static | | DBMS | | DBMS-Transacted | | Static | |
Schema	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
ArtistSimilarity	461	62	449	5	28	< 1	2811	312	2563	126	27	4
ArtistTerm	537	8	531	2	37	< 1	15059	769	13351	453	32	< 1
BankAccount	550	7	546	3	43	1	6261	387	5594	257	39	< 1
BookTown	2072	12	2069	13	40	1	244112	15297	185449	1335	< 1	< 1
BrowserCookies	768	4	766	4	63	1	33613	1133	31620	655	59	< 1
Cloc	477	4	476	2	1	< 1	2345	161	2123	179	< 1	< 1
CoffeeOrders	1000	10	991	11	87	2	35357	1571	28307	545	79	1
CustomerOrder	1906	21	1905	57	144	1	95346	3393	75280	1191	138	1
DellStore	1574	13	1571	12	1	1	38694	826	28500	668	< 1	< 1
Employee	499	3	497	2	27	< 1	3746	234	3665	269	< 1	< 1
Examination	856	4	853	6	59	1	21451	812	18810	547	54	< 1
Flights	729	4	724	6	67	1	12764	556	10821	462	65	< 1
FrenchTowns	1087	11	1080	9	72	1	63723	1601	59590	768	62	1
Inventory	446	3	444	2	29	< 1	2569	151	2600	221	< 1	< 1
Iso3166	442	4	442	3	26	< 1	986	107	984	62	< 1	< 1
IsoFlav_R2	1056	11	1054	15	30	1	153139	4083	137529	1646	< 1	< 1
iTrust	8313	83	8258	104	124	2	4625197	18668	3785693	60134	67	2
JWhoisServer	1796	21	1796	15	37	1	111755	3703	97565	1233	< 1	< 1
MozillaExtensions	1278	21	1274	13	35	1	286327	4926	259820	2211	< 1	< 1
MozillaPermissions	464	1	463	3	26	< 1	4104	235	4234	281	< 1	< 1
NistDML181	486	3	482	3	41	< 1	6775	381	6128	214	40	< 1
NistDML182	1011	6	978	10	257	2	30688	1352	24817	712	265	2
NistDML183	485	3	481	2	39	1	3382	257	2864	126	38	< 1
NistWeather	628	2	626	5	43	1	7450	543	6552	396	40	1
NistXTS748	452	4	450	4	27	1	1641	120	1707	175	< 1	< 1
NistXTS749	523	6	517	3	42	2	7514	403	6842	287	39	1
Person	471	3	470	3	27	< 1	1927	168	1970	142	< 1	< 1
Products	738	3	735	3	48	< 1	18198	884	16054	459	44	< 1
RiskIt	2309	17	2282	23	208	2	256171	5611	203354	2340	199	2
StackOverflow	872	6	872	3	1	< 1	35407	1619	30343	732	< 1	< 1
StudentResidence	565	3	563	3	35	< 1	6801	410	6025	284	31	1
UnixUsage	1467	54	1442	9	149	1	91778	3526	71117	1148	141	2
Usda	1761	10	1760	9	2	< 1	68873	1296	50764	925	< 1	< 1
WordNet	1169	12	1169	12	34	< 1	79787	2986	68046	993	< 1	< 1

with 22; third is *RiskIt* with 13, as shown by Table 2). While the times required for the DBMS-based methods are related to the schema's size or the number of mutants which result from it, the cost of the static checks is more closely related to the number of stillborn mutants produced by the operators.

Impaired, Equivalent, and Redundant Mutants

Following the notational conventions given in Section 6.1.4, Tables 9 and 10 report mean mutation analysis times with tests generated using *AVM* and *Random$^+$*, respectively, with the exclusion of different sets of mutants to form the mutant pool used in mutation analysis. Each subsequent column in the table for a DBMS involves the removal of a particular type of ineffective mutant, and, for the purposes of statistical testing and effect size computation, can be compared with the left-most adjacent column for timing differences. For example, the $-(S+I+E+R)$ column shows mean times when all ineffective mutants have been removed, and can be contrasted with the $-(S+I+E)$ column to draw conclusions about the effect of removing redundant mutants. As given in Table 11, we summarize this information in the following discussion by counting the number of database schemas where times significantly improve (i.e., times decrease) or become significantly worse (i.e., times increase), highlighting the greatest increases and decreases in mutation analysis time for a database schema as appropriate.

Impaired Mutants. Since only SQLite has automatic checks for impaired mutants, as originally discussed in Section 5.3, Tables 9 through 11 report figures for the $-(S+I)$ set of mutants (i.e., the set of mutants following removal of impaired mutants) for this DBMS. To analyze the effect of removing impaired mutants, we contrast times in this column with the preceding $-S$ column of the same table.

For *Random$^+$*, *SchemaAnalyst* performs mutation analysis significantly faster for 14 schemas after removing impaired mutants, with a large effect size in each case. With the *AVM*, mutation analysis is significantly faster overall for the same 14 schemas as *Random$^+$*, and additionally, *FrenchTowns*. The effect size is large for 12 of these 15 schemas, of which *RiskIt* sees the greatest performance improvement. With SQLite, a comparison of the value for *RiskIt* in Table 9's "Impaired" column to its "Stillborn" column shows that *SchemaAnalyst* achieved a mean saving of 6 seconds when using *Random$^+$*. The mean saving for the *AVM* is just under 8 seconds, as observed from the corresponding values in Table 10.

Table 4 shows that 19 schemas have impaired mutants, meaning that there were four schemas with impaired mutants (i.e., *ArtistSimilarity*, *ArtistTerm*, *iTrust*, and *StudentResidence*) for which mutation analysis did not become significantly faster following their removal. In fact, Tables 9 and 10 reveal that mutation analysis time increased, in a statistically significant fashion and with large effect size, for two schemas (i.e., *ArtistSimilarity* and *StudentResidence*) with *Random$^+$* and also for *ArtistSimilarity* with *AVM*. Notably for *ArtistSimilarity*, the cost of finding and removing its single impaired mutant leads to a time increase. It is clear that these schemas did not have a sufficient number of impaired mutants to make mutation analysis significantly faster.

Equivalent Mutants. In contrast to impaired mutants, *SchemaAnalyst* supports, with the methods described in Sections 4 and 5, the removal of equivalent mutants for all three DBMSs. The $-(S+I+E)$ columns of Tables 9 and 10 show mean mutation analysis times for the mutant pool with equivalent mutants removed. We compare this column with the preceding $-S$ column (for HyperSQL and PostgreSQL)

TABLE 9
Mean mutation analysis times (in milliseconds) for test suites generated with $Random^+$

This table reports the mean mutation times for mutation analysis following the removal of a type of ineffective mutant from the mutant pool (resulting in one of the different sets of mutants used in our experiments, i.e., "$-S$, $-(S+I)$ ...", etc., as described in Section 5.3). A figure with an accompanying "▽" symbol denotes that mutation analysis times significantly decreased (i.e., mutation analysis became faster) with the removal of the ineffective mutant type, while "△" denotes that mutation analysis times significantly increased (i.e., mutation analysis became slower). A "★" indicates that the effect size was large. (See Section 6.1.5 for more information about the computation of the statistical significance and effect size measures reported in this table.)

	HyperSQL			PostgreSQL			SQLite			
	Stillborn	Equivalent	Redundant	Stillborn	Equivalent	Redundant	Stillborn	Impaired	Equivalent	Redundant
Schema	$-S$	$-(S+I+E)$	$-(S+I+E+R)$	$-S$	$-(S+I+E)$	$-(S+I+E+R)$	$-S$	$-(S+I)$	$-(S+I+E)$	$-(S+I+E+R)$
ArtistSimilarity	125	★△ 144	★△ 153	5190	★▽ 4348	★▽ 3532	61	★△ 84	★△ 114	★△ 137
ArtistTerm	714	718	★△ 840	30256	★▽ 23422	23546	397	407	★△ 473	★△ 552
BankAccount	725	★△ 753	★△ 899	24367	★▽ 21226	21371	456	★▽ 419	★△ 500	★△ 694
BookTown	22063	★▽ 21164	★△ 25154	1529847	★▽ 1374219	▽ 1365462	30572	30572	★▽ 29311	29877
BrowserCookies	1786	★△ 1880	★△ 2251	109460	★▽ 104543	▽ 100974	1216	★▽ 1118	★△ 1270	★△ 1693
Cloc	875	★△ 939	★△ 1058	25883	25948	26067	325	325	★△ 390	★△ 416
CoffeeOrders	2182	★▽ 2135	★△ 2511	85352	★▽ 71440	71828	1709	★▽ 1324	★△ 1429	★△ 1888
CustomerOrder	3144	3190	★△ 3896	190678	★▽ 162563	163239	3576	★▽ 2951	★△ 3116	★△ 4440
DellStore	7393	★△ 7604	★▽ 5986	539668	539878	★▽ 405741	7121	7121	7336	★▽ 6213
Employee	785	★△ 855	★△ 1054	24892	★▽ 23662	23860	348	348	★△ 452	★△ 649
Examination	3110	★△ 3251	★△ 3801	246915	★▽ 237505	238059	3313	★▽ 3135	3275	★△ 3906
Flights	1426	1434	★△ 1646	44565	40840	39227	936	★▽ 704	★△ 854	★△ 1199
FrenchTowns	1565	★△ 1725	1616	96648	★▽ 89527	★▽ 57964	844	829	★△ 954	★△ 1204
Inventory	222	★△ 245	★△ 311	9509	★▽ 8670	★▽ 7863	104	104	★△ 162	★△ 227
Iso1	71	★△ 84	★△ 109	2512	★▽ 2084	2109	35	35	★△ 77	★△ 101
IsoFlav_R2	6735	★△ 6929	★△ 9008	941703	941922	931542	8349	8349	8431	8702
iTrust	1384595	★▽ 1346515	★△ 2641232	81471300	★▽ 77147636	77233244	2061300	2045496	★▽ 2008114	★△ 3195904
JWhoisServer	7228	★▽ 7318	★△ 9287	578919	★▽ 542861	544821	7349	7349	7450	★△ 9346
MozillaExtensions	14287	14465	★△ 19777	1969632	1948320	★△ 1883958	21881	21881	22050	★△ 25958
MozillaPermissions	633	★△ 671	★△ 790	24046	★▽ 22519	22639	274	274	★△ 349	★△ 468
NistDML181	442	450	★△ 560	14919	★▽ 13468	13577	268	★▽ 239	★△ 318	★△ 427
NistDML182	2674	★▽ 2492	★△ 3175	207688	★▽ 169772	170305	9040	★▽ 4579	★△ 4805	★△ 7215
NistDML183	630	★△ 686	★△ 784	19524	19579	19677	332	★▽ 292	★△ 365	★△ 399
NistWeather	868	903	★△ 1059	23592	★▽ 20981	20495	569	★▽ 485	★△ 600	★△ 799
NistXTS748	228	★△ 263	★△ 320	6636	6672	★▽ 5955	76	76	★△ 132	★△ 200
NistXTS749	719	718	★△ 785	24891	★▽ 21032	★▽ 19256	374	★▽ 341	★△ 429	★△ 579
Person	350	★△ 389	★△ 499	8529	★▽ 7829	7940	132	132	★△ 206	★△ 318
Products	1384	1426	★△ 1759	49173	★▽ 43066	43400	827	★▽ 768	★△ 898	★△ 1253
RiskIt	11389	★▽ 11072	★△ 14924	1141541	★▽ 1020890	★▽ 1003204	28729	★▽ 22702	★▽ 22096	★△ 30407
StackOverflow	5074	★△ 5221	★△ 5438	497848	497998	★▽ 478979	5069	5069	★△ 5218	★▽ 4007
StudentResidence	681	★△ 749	★△ 955	21297	★▽ 18904	19112	349	★△ 357	★△ 453	★△ 677
UnixUsage	3613	3659	★△ 4259	278588	★▽ 239667	★▽ 232038	5640	★▽ 4761	△ 4885	★△ 6364
Usda	10889	★△ 11092	★△ 11388	983728	983933	★▽ 834074	13225	13225	13432	★▽ 11439
WordNet	3788	★▽ 3647	△ 3980	324839	★▽ 264274	★▽ 240764	3131	3131	★△ 3270	★△ 3681

or the $-(S+I)$ column (for SQLite) of the same data table to study the effect of removing equivalent mutants from the mutant pool on the time taken for mutation analysis.

The summary information in Table 11 reveals that, for HyperSQL, mutation analysis times significantly improve for 6 schemas with test suites generated using $Random^+$, and an additional 6 with test suites generated by the AVM (i.e., 12 in total), with large effect sizes in each case. When SchemaAnalyst uses either $Random^+$ or the AVM, the iTrust schema sees the greatest decrease in mutation analysis time for HyperSQL, with time savings of about 38 and 42 seconds, respectively. Yet, for HyperSQL with both the AVM and $Random^+$, times are significantly worse for 19 schemas.

For PostgreSQL, mutation analysis times never become significantly worse, and become significantly better for 25 schemas with test suites generated by $Random^+$, and 27 schemas with the tests from AVM. For each configuration, significance is coupled with a large effect size. As with the HyperSQL DBMS, the iTrust schema demonstrates the greatest decrease in mutation analysis time, with SchemaAnalyst saving 72 and 89 minutes when it uses tests by $Random^+$ and the AVM, respectively. Finally, for SQLite, mutation analysis times become significantly better for 3 schemas with tests from AVM and $Random^+$, but significantly worse for 25. Yet, all three cases of significant improvement are coupled with a large effect size. Notably, SchemaAnalyst experiences the greatest decrease in mutation analysis time with the iTrust schema, respectively saving about 37 and 41 seconds with the test suites generated by $Random^+$ and the AVM.

Removing equivalent mutants involves comparing each mutated schema against the original schema. The cost of this comparison depends on the complexity of the schema under analysis, and tends to be slower than the automated checks that identify impaired mutants. As with impaired mutants, the differences by schema are explained by the number of equivalent mutants removed from the pool. Therefore, the timings vary depending on how many mutants SchemaAnalyst removes. When equivalent mutants are abundant, the overall time required for SchemaAnalyst to perform mutation analysis is reduced significantly. In fact, the schemas that experienced significant improvement in times for all three DBMSs (i.e., BookTown, iTrust, and RiskIt) had some of the greatest numbers of equivalent mutants identified out of all the subjects (c.f. Table 2). Yet, when equivalent mutants are few in number, the mutation analysis is significantly slower.

Yet, the choice of DBMS has the greatest effect on the mutation analysis times with or without the equivalent mutants. For PostgreSQL, mutation analysis times never become significantly worse, whereas for the other two DBMSs it depends on how many equivalent mutants are produced by the operators. This is primarily due to the way that the DBMSs are designed and work: PostgreSQL is an enterprise DBMS that uses disk-based storage, thus making it slow at evaluating mutants. The cost of evaluating extra ineffective mutants dominates that of detecting and eliminating them. In contrast, HyperSQL and SQLite store databases in memory, allowing for mutants to be evaluated quickly—meaning that the numbers of equivalent mutants involved must be high before the cost of running the removal algorithms may be recouped and additional time saved.

Redundant Mutants. The $-(S+I+E+R)$ columns of Tables 9 and 10 show the mean times for mutation analysis when the pool of mutants excludes those that are redundant. We compare this column with the preceding $-(S+I+E)$

TABLE 10
Mean mutation analysis times (in milliseconds) for test suites generated with the *AVM*
(Please see the caption of Table 9 for a description of this table's headings)

Schema	HyperSQL			PostgreSQL			SQLite			
	Stillborn $-S$	Equivalent $-(S+I+E)$	Redundant $-(S+I+E+R)$	Stillborn $-S$	Equivalent $-(S+I+E)$	Redundant $-(S+I+E+R)$	Stillborn $-S$	Impaired $-(S+I)$	Equivalent $-(S+I+E)$	Redundant $-(S+I+E+R)$
ArtistSimilarity	237	★ △ 246	★ ▽ 232	8721	★ △ 7295	★ ▽ 5862	118	★ △ 138	★ △ 164	★ △ 167
ArtistTerm	1128	★ △ 1095	★ △ 1216	49333	★ ▽ 38090	38214	636	637	★ △ 690	★ △ 710
BankAccount	847	★ △ 867	★ △ 1013	29174	★ ▽ 25335	25480	549	★ ▽ 487	★ △ 566	★ △ 760
BookTown	23488	★ ▽ 22507	★ △ 26486	1646118	★ ▽ 1478688	★ ▽ 1468870	32808	32808	★ △ 31432	★ ▽ 30957
BrowserCookies	2128	★ △ 2229	★ △ 2601	204301	★ ▽ 194419	★ ▽ 187244	2098	★ ▽ 1936	★ △ 2080	★ △ 2475
Cloc	968	★ △ 1032	★ △ 1151	27872	27936	28056	381	381	★ △ 445	★ △ 450
CoffeeOrders	2598	★ ▽ 2517	★ △ 2893	187003	★ ▽ 156504	156892	2899	★ ▽ 2370	★ △ 2395	★ △ 2854
CustomerOrder	5146	★ ▽ 5090	★ △ 5795	542150	★ ▽ 461932	462608	9495	★ ▽ 8041	7913	★ △ 9238
DellStore	7448	★ △ 7658	★ ▽ 6069	589502	589711	★ ▽ 447365	7622	7622	★ △ 7836	★ ▽ 6398
Employee	899	★ △ 966	★ △ 1165	28533	★ ▽ 27102	27300	409	409	★ △ 512	★ ▽ 709
Examination	3566	★ △ 3700	★ △ 4250	312234	★ ▽ 300155	300708	4244	★ ▽ 4027	★ △ 4158	★ △ 4789
Flights	1918	1919	★ △ 2080	88541	★ ▽ 80846	★ ▽ 77390	1594	★ ▽ 1225	★ △ 1374	★ △ 1701
FrenchTowns	2439	★ △ 2599	★ ▽ 2317	244147	★ ▽ 225622	★ ▽ 144442	2385	★ ▽ 2288	2298	★ ▽ 2099
Inventory	231	★ △ 253	★ △ 318	9656	★ △ 8748	★ △ 7895	114	114	★ △ 172	★ △ 234
Iso3166	85	★ △ 96	★ △ 121	2762	★ ▽ 2291	2317	51	51	★ △ 93	★ △ 117
IsoFlav_R2	7413	★ △ 7607	★ △ 9686	1045369	1045588	★ ▽ 1033437	8999	8999	9041	★ △ 9141
iTrust	1512846	★ ▽ 1470576	★ △ 2763257	91935913	★ ▽ 86593001	86648519	2222324	2208819	★ ▽ 2167725	★ △ 3350112
JWhoisServer	7722	★ △ 7793	★ △ 9761	674101	★ ▽ 630688	632648	8640	8640	8641	★ △ 10538
MozillaExtensions	14909	15086	★ △ 20372	2142353	★ ▽ 2117573	★ ▽ 2044416	22814	22814	23004	★ △ 26736
MozillaPermissions	670	★ △ 706	★ △ 825	24469	★ ▽ 22795	22916	301	301	★ △ 375	★ △ 494
NistDML181	697	★ ▽ 676	★ △ 785	23293	★ ▽ 21012	21121	444	★ ▽ 377	★ △ 456	★ △ 539
NistDML182	3296	★ ▽ 3029	★ △ 3712	362090	★ ▽ 293206	293739	12802	★ ▽ 6710	★ △ 6935	★ △ 9048
NistDML183	644	★ △ 700	★ ▽ 798	19906	19961	20059	350	★ ▽ 306	★ △ 379	★ △ 408
NistWeather	1340	★ ▽ 1331	★ △ 1472	48448	★ ▽ 42817	★ ▽ 41644	775	★ ▽ 656	★ △ 769	★ △ 957
NistXTS748	212	★ △ 246	★ △ 305	6465	6501	★ ▽ 5873	84	84	★ △ 140	★ △ 206
NistXTS749	853	★ ▽ 837	★ △ 894	29727	★ ▽ 24974	★ ▽ 22826	459	★ ▽ 412	★ △ 498	★ △ 642
Person	392	★ △ 429	★ △ 539	8934	★ △ 8176	8287	162	162	★ △ 235	★ △ 347
Products	1765	★ △ 1789	★ △ 2122	76111	★ ▽ 66182	66516	1038	★ ▽ 968	★ △ 1088	★ △ 1437
RiskIt	15333	★ ▽ 14744	★ △ 18509	1920017	★ ▽ 1715760	★ ▽ 1682656	41185	★ ▽ 33360	★ ▽ 32111	★ △ 40123
StackOverflow	5269	★ △ 5416	★ △ 5615	511428	511578	★ ▽ 492259	5436	5436	★ △ 5585	★ ▽ 4175
StudentResidence	959	★ △ 1013	★ △ 1220	31906	★ ▽ 28314	28522	524	523	★ △ 610	★ △ 835
UnixUsage	5523	★ ▽ 5387	★ △ 5971	723963	★ ▽ 622293	★ ▽ 601248	11805	★ ▽ 10293	10164	★ △ 11316
Usda	11797	★ △ 12001	★ △ 12179	1111570	1111775	★ ▽ 942597	13889	13889	★ △ 14096	★ ▽ 11629
WordNet	4058	★ ▽ 3870	★ △ 4193	358214	★ ▽ 290945	★ ▽ 264849	3780	3780	★ △ 3903	★ △ 4268

column of the corresponding table to study the effect of removing redundant mutants from the mutant pool on the time taken to perform mutation analysis with a schema.

The summary information in Table 11 shows that, for HyperSQL and the removal of redundant mutants, only one schema (i.e., *DellStore*) experienced a significant decrease in mutation analysis time with tests generated by either *Random$^+$* or *AVM*. While almost all of the other schemas saw a significant increase in mutation analysis time (i.e., 32 with *Random$^+$* and 31 with *AVM*), the effect size for *DellStore* is large, representing a time savings of about one second for HyperSQL and the tests from either *Random$^+$* or *AVM*. For PostgreSQL, 14 schemas experienced a significant decrease in mutation analysis time with *Random$^+$* and 17 with the *AVM*. In contrast to mutation analysis with HyperSQL, no schemas are subject to a significant increase in time for this DBMS. With PostgreSQL, *Usda* saw the greatest reduction in time, saving about two minutes when it used tests from either *Random$^+$* or *AVM*. Finally, for SQLite, three schemas exhibit a significant improvement in mutation analysis time for *Random$^+$* (all with a large effect size), with five for *AVM* (again, all with a large effect size). Most of the remaining schemas (i.e., 29) saw a significant increase in time. Again, *Usda* sees the greatest reduction in mutation analysis time, with savings of about one and two seconds when it uses SQLite and tests from *Random$^+$* and *AVM*, respectively.

The identification of redundant mutants is potentially more costly than for equivalent mutants: each mutant must be compared not just against one other schema (i.e., the original schema under test), but against every other mutant. Therefore, overall savings are less frequent with this type of ineffective mutant, since more mutants need to be removed to recoup the upfront cost of the analysis. For the majority of schemas, the overall mutation analysis process becomes significantly slower, except for when PostgreSQL is used.

Here, as for equivalent mutants, the use of disk-based storage makes this DBMS slow at evaluating mutants, and thus the cost of performing the static analysis never leads to a significantly negative effect on mutation times.

Finally, it is worth noting that when mutation analysis uses tests created by *Random$^+$*, *SchemaAnalyst* has fewer significant timing improvements than it does when it leverages *AVM*'s tests. This trend is evident because *Random$^+$* makes smaller test suites that cover fewer test coverage requirements than those suites created by the *AVM*, as shown in Table 3. Consequently, mutants are faster to evaluate with *Random$^+$* than *AVM*, which in turn renders the removal of ineffective mutants less beneficial—particularly for faster, memory-based DBMSs like HyperSQL and SQLite.

Conclusion for RQ2

Checking for stillborn mutants by submitting mutated schemas to the DBMS is a time-consuming process, even when using transactions. In contrast, static analysis checks for invalid schemas are fast, taking on the order of milliseconds, rather than seconds, minutes, or even hours. The removal of impaired mutants, a step that the algorithms only perform for SQLite, is similarly fast. When mutation operators produce many impaired mutants for a schema, the speed of mutation analysis generally improves significantly when *SchemaAnalyst* removes them. Table 11 illustrates this trend, revealing that, for test suites created by either *Random$^+$* or *AVM*, the detection and removal of impaired mutants respectively reduces mutation analysis times, with a large effect size, for 14 and 15 database schemas.

For tests generated by both *Random$^+$* and *AVM*, the summary in Table 11 makes it clear that, when using PostgreSQL, the removal of equivalent mutants decreases mutation analysis time, with a large effect size, for 25 and 27 of the 34 schemas. While the further removal of redundant

TABLE 11
Summary of significance and effect size results for mutation analysis times

This table gives the number of database schemas for which mutation analysis became significantly faster (i.e., times decreased) when a type of impaired mutant was removed consideration in our experiments, denoted by columns headed "▽", while columns headed "△" denote the number of schemas for which mutation analysis became significantly slower (i.e., times increased). Figures in parentheses indicate the number of times that significance was accompanied by a large effect size. (See Section 6.1.5 for more information about the computation of the statistical significance and effect size measures reported in this table.) Note that there is no data for the removal of impaired mutants for the HyperSQL and PostgreSQL DBMSs, as they do not need automated checks to identify this type of mutant.

| | $Random^+$ | | | | | | AVM | | | | | |
| | Impaired $-(S+I)$ | | Equivalent $-(S+I+E)$ | | Redundant $-(S+I+E+R)$ | | Impaired $-(S+I)$ | | Equivalent $-(S+I+E)$ | | Redundant $-(S+I+E+R)$ | |
	▽	△	▽	△	▽	△	▽	△	▽	△	▽	△
HyperSQL	-	-	6 (6)	19 (18)	1 (1)	32 (32)	-	-	12 (12)	19 (19)	3 (3)	31 (31)
PostgreSQL	-	-	25 (25)	0 (0)	14 (12)	0 (0)	-	-	27 (27)	0 (0)	17 (17)	0 (0)
SQLite	14 (14)	2 (2)	3 (3)	25 (24)	3 (3)	29 (29)	15 (15)	1 (1)	3 (3)	25 (25)	5 (5)	29 (29)

mutants is less beneficial, this table shows that, for $Random^+$ and AVM respectively, 12 and 17 schemas still see mutation analysis times drop with a large effect size. These trends are evident because PostgreSQL is a disk-based DBMS, meaning that the cost of the static checks for ineffective mutant detection is outweighed by the expense of creating the mutants and running tests during mutation analysis.

The picture for equivalent mutants, and the HyperSQL and SQLite DBMSs, is mixed and depends on whether the mutation of a schema generates enough ineffective mutants so that the time taken to check for and remove them from the pool is recouped by not having to consider them later in mutation analysis. Table 11 reveals that, while there are more schemas that do not benefit from removing equivalent mutants than do, 6 and 12 schemas, respectively, see a decrease in mutation analysis time with HyperSQL and tests from $Random^+$ and AVM. Yet, only three schemas, for tests generated by both $Random^+$ and AVM, experience a reduction in mutation analysis time with SQLite.

Table 11 shows that the further exclusion of redundant mutants, for both HyperSQL and SQLite, leads to few decreases in mutation analysis time. While Tables 9 and 10 point out that small reductions in mutation analysis time are possible, Table 11 reveals, for tests generated by both $Random^+$ and AVM, that this happens for relatively few schemas. For instance, only 1 and 3 schemas, respectively, see decreased mutation analysis time with tests from $Random^+$ when run with HyperSQL and SQLite. The trend is similar for tests generated by the AVM, with only 3 and 5 schemas, respectively, seeing time reductions with these two DBMSs. Overall, these results are evident since redundant mutant removal requires the comparison of a mutant to all others in the pool, making it expensive and thus less likely to decrease the overall cost of mutation analysis.

In summary, Tables 9 through 11 point out that, while the removal of impaired and equivalent mutants often speeds up mutation analysis, there is a point of diminishing returns. Yet, importantly, there are other benefits to the removal of ineffective mutants. As explained in the answer to the next research question, finding and removing these mutants can also lead to desirable changes in the mutation score.

RQ3: Impact on the Mutation Score

Tables 12 and 13 show how the mean mutation score changes as a result of removing different types of ineffective mutants for the tests generated by $Random^+$ and the AVM, respectively. The AVM achieves higher mutation scores than does $Random^+$. On average, its mutation score never drops below 85% over all schemas when stillborn mutants are not

considered, regardless of the DBMS. In contrast, the average mutation score of $Random^+$'s test suites, over all schemas, never increases beyond 75%, regardless of the DBMS.

We now discuss the effect of removing different types of ineffective mutant on the mutation scores for the database schemas, leveraging Table 14's summary data. At the outset, it is worth noting that a decrease in a test suite's mutation score is not a negative outcome per se—the lower score now gives a more useful understanding of its effectiveness. Since stillborn mutants do not contribute to mutation scores, we start by analyzing the effect of removing impaired mutants.

Impaired Mutants

Because impaired mutants artificially inflate the mutation scores, their removal can only lead to a decrease in the resulting scores. The algorithms are only designed to detect impaired mutants for the SQLite DBMS, and 19 of the subject schemas feature them, as shown by Table 4.

With tests generated by the AVM, the mutation scores of 16 of these 19 significantly decreased (with large effect sizes) following *SchemaAnalyst*'s removal of impaired mutants, as summarized by Table 14 and seen in Tables 12 and 13 by comparing the $-S$ (i.e., stillborn) and $-(S+I)$ (i.e., impaired) columns for this DBMS. The mutation scores of the remaining three schemas with impaired mutants (i.e., *NistDML181*, *NistDML182*, and *NistDML183*) did not change, because their test suites killed all other mutants, and as such had perfect mutation scores (i.e., 100%) before (and after) their removal. The killed-to-total proportion remained the same for the other schemas without any impaired mutants.

Tests suites generated by $Random^+$ for *NistDML181* and *NistDML182* did not obtain perfect mutation scores before *SchemaAnalyst* removed impaired mutants (as can be seen by comparing their mutation scores in Tables 12 and 13 under the $-S$ column for SQLite). Overall, 18 of the 19 schemas with impaired mutants saw a decrease in mutation score with this test generator. Of these 18 schemas, 16 saw significant decreases, with a large effect size.

Equivalent Mutants

As equivalent mutants are impossible to kill, and thus artificially deflate mutation scores, schemas with these mutants saw their mutation scores increase for all of the test suites generated by both the AVM and $Random^+$. This effect can be seen by looking at Tables 12 and 13 and, for SQLite, comparing the scores in the $-(S+I)$ (i.e., impaired) column to the $-(S+I+E)$ (i.e., equivalent) column. Additionally, for HyperSQL and PostgreSQL, the comparison is between

TABLE 12
Mean mutation scores obtained with test suites generated by *Random*$^+$

Mutation scores are reported for each schema with each DBMS following the removal of a type of impaired mutant from mutation analysis (resulting in one of the different sets of mutants used in our experiments, i.e., "$-S$, $-(S+I)$ ...", etc., as described in Section 5.3). A figure with an accompanying "▽" symbol denotes that mutation scores significantly decreased with the removal of the ineffective mutant type, while "△" denotes that mutation score significantly increased. A "★" indicates that the effect size was large. (See Section 6.1.5 for more information on the statistical tests and effect size measures used.)

	HyperSQL			PostgreSQL			SQLite			
Schema	Stillborn $-S$	Equivalent $-(S+I+E)$	Redundant $-(S+I+E+R)$	Stillborn $-S$	Equivalent $-(S+I+E)$	Redundant $-(S+I+E+R)$	Stillborn $-S$	Impaired $-(S+I)$	Equivalent $-(S+I+E)$	Redundant $-(S+I+E+R)$
ArtistSimilarity	74.5	★△ 82.0	★▽ 77.5	68.3	★△ 82.0	★▽ 77.5	64.6	★▽ 61.7	★△ 67.3	★△ 74.3
ArtistTerm	69.9	★△ 80.3	80.3	61.8	★△ 80.3	80.3	62.5	★▽ 58.2	★△ 65.8	★△ 79.3
BankAccount	69.2	★△ 74.5	74.5	64.6	★△ 74.5	74.5	68.3	★▽ 58.4	★△ 62.3	62.3
BookTown	84.5	★△ 89.7	89.9	80.5	★△ 89.7	89.9	80.9	80.9	★△ 85.6	△ 86.9
BrowserCookies	51.8	★△ 53.7	★△ 55.8	51.2	★△ 53.7	★△ 55.8	55.8	★▽ 45.8	46.3	★△ 47.9
Cloc	89.8	89.8	89.8	89.8	89.8	89.8	79.6	79.6	79.6	★△ 84.7
CoffeeOrders	42.0	★△ 46.1	46.1	38.5	★△ 46.1	46.1	66.6	★▽ 44.8	★△ 48.8	48.8
CustomerOrder	34.0	△ 36.9	36.9	31.9	★△ 37.3	37.3	59.9	★▽ 34.6	36.9	36.9
DellStore	87.1	87.1	★△ 90.1	87.1	87.1	★△ 90.1	85.2	85.2	85.2	★△ 88.9
Employee	87.0	89.0	89.0	85.0	★△ 89.0	89.0	86.4	86.4	88.4	88.4
Examination	87.4	89.0	89.0	85.9	★△ 89.0	89.0	78.9	76.2	77.4	77.4
Flights	40.8	44.5	46.3	40.8	44.5	46.3	66.5	★▽ 47.0	47.0	48.6
FrenchTowns	23.0	23.0	26.0	21.3	23.0	26.0	32.6	★▽ 21.6	23.3	26.6
Inventory	79.6	★△ 84.3	★△ 87.5	74.6	★△ 82.5	★△ 88.2	79.4	79.4	★△ 83.3	★△ 87.5
Iso3166	69.7	77.4	77.4	63.3	★△ 77.4	77.4	61.5	61.5	67.7	67.7
IsoFlav_R2	56.3	56.3	56.3	56.9	56.9	56.9	49.3	49.3	50.0	★▽ 47.2
iTrust	87.1	★△ 90.0	90.1	85.3	★△ 90.0	90.1	85.5	85.4	★△ 87.2	87.4
JWhoisServer	85.4	★△ 88.3	88.3	82.7	★△ 88.3	88.3	85.9	85.9	★△ 88.7	88.7
MozillaExtensions	60.2	60.5	61.3	59.9	60.6	61.5	59.7	59.7	60.1	60.7
MozillaPermissions	92.4	★△ 95.6	95.6	89.5	★△ 95.6	95.6	92.7	92.7	★△ 95.8	95.8
NistDML181	64.7	★△ 71.9	71.9	64.7	★△ 71.9	71.9	73.2	★▽ 59.8	59.8	★△ 72.0
NistDML182	66.0	★△ 78.2	78.2	60.5	★△ 74.2	74.2	89.9	★▽ 68.2	68.2	★△ 80.5
NistDML183	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
NistWeather	44.9	49.2	50.7	43.6	49.2	50.7	72.7	★▽ 62.6	★△ 64.4	★△ 67.6
NistXTS748	94.3	94.3	★▽ 93.9	94.6	94.6	★▽ 93.9	89.3	89.3	★△ 94.3	★▽ 93.5
NistXTS749	75.2	★△ 85.5	86.7	72.3	★△ 85.5	86.7	82.3	★▽ 76.0	78.8	79.2
Person	88.0	92.2	92.2	84.2	★△ 92.2	92.2	88.6	88.6	92.6	92.6
Products	55.8	★△ 60.5	60.5	53.6	★△ 60.5	60.5	76.9	★▽ 70.8	73.6	74.8
RiskIt	63.5	★△ 67.6	68.3	60.5	★△ 67.6	68.3	76.5	★▽ 63.6	★△ 66.5	★△ 67.9
StackOverflow	94.8	94.8	94.5	94.8	94.8	94.5	89.5	89.5	89.5	★△ 92.1
StudentResidence	83.1	★△ 87.7	87.7	78.9	★△ 87.7	87.7	76.2	★▽ 73.9	★△ 77.7	77.7
UnixUsage	45.4	★△ 49.7	★△ 51.4	42.7	★△ 49.7	★△ 51.4	69.4	★▽ 53.0	★△ 55.7	★△ 59.1
Usda	80.9	80.9	★△ 85.5	80.9	80.9	★△ 85.5	75.6	75.6	75.6	★△ 81.7
WordNet	69.8	★△ 83.0	82.4	67.3	★△ 82.8	82.4	76.5	76.5	★△ 79.4	78.8

the scores in $-S$ (i.e., stillborn) and $-(S+I+E)$ (i.e., equivalent) columns. The summary in Table 14 shows that the tests for 15 to 23 schemas experienced a significant increase in mutation score when they were generated by *Random*$^+$, while test scores for 26 to 27 schemas significantly increased if the *AVM* generated them. The removal of equivalent mutants had a further interesting effect: when using the *AVM*, tests for seven schemas with HyperSQL and PostgreSQL, and five schemas with SQLite, which were previously thought to have suboptimal mutation scores, changed to scores of 100%. That is, every mutant was killed after removing equivalent mutants from the mutant pool.

Redundant Mutants

The removal of redundant mutants cannot cause test suites to change to 100% mutation scores, even if all other mutants are killed. Either the redundant mutant pair is killed, and thus the score must already be 100%, or the pair is not killed, in which case one member of the pair will still remain alive in the mutant pool. When the pair is *not* killed, the mutation score always increases when one of the mutants in the pair is removed, as the total number of mutants (i.e., the denominator of the mutation score equation) is now less. When the pair is killed, the mutation score always decreases when a mutant in the pair is removed, as the numerator and denominator always decrease by one.

Since the *AVM*'s tests are good at killing mutants, suites generated by this technique tend to experience a decrease in mutation score when redundant mutants are removed from the pool. Table 14 shows that test suites for 12 to 15 schemas experience a significant drop, depending on the DBMS in use. In contrast to the *AVM*, *Random*$^+$ generates tests that are not as good at killing mutants and thus suites for 2 schemas exhibited a significant decrease in their score, with 5 to 14 seeing a significant increase in their score.

Conclusion for RQ3

As shown in Table 14, the removal of impaired, equivalent, and redundant mutants generally changes the mutation score of a test suite. In particular, removing equivalent mutants always affects test suites, and can cause a test suite to "gain" a perfect mutation score, where previously it was thought to have one that was suboptimal, due to a mutant that was impossible to kill. In summary, this paper's technique can help testers of database schemas achieve a more precise understanding of the quality of their test suites.

The effect of removing impaired and redundant mutants on a test suite's mutation score depends on the initial strength of the generated tests. If a test suite originally achieves a 100% score, then removing these mutants cannot improve the score any further; moreover, their removal cannot make it any worse. However, if a test suite has a sub-100% score, then it is likely to experience a change in its score. This means that the removal of these types of ineffective mutants has more of an effect on the weaker test suites generated by the *Random*$^+$ test data generator than it does for the *AVM*'s test suites, which are stronger and more likely to therefore achieve 100% mutation scores.

Overall, automatically finding and removing ineffective mutants lead to 33 of the 34 schemas having a significantly changed mutation score for at least one DBMS and one test generation method. The only schema that didn't undergo a significant change—*NistDML183*—had a consistent 100% score before any ineffective mutants were removed.

TABLE 13
Mean mutation scores obtained with test suites generated by the *AVM*
(Please see the caption of Table 12 for a description of this table's headings)

Schema	HyperSQL Stillborn $-S$	Equivalent $-(S+I+E)$	Redundant $-(S+I+E+R)$	PostgreSQL Stillborn $-S$	Equivalent $-(S+I+E)$	Redundant $-(S+I+E+R)$	SQLite Stillborn $-S$	Impaired $-(S+I)$	Equivalent $-(S+I+E)$	Redundant $-(S+I+E+R)$
ArtistSimilarity	90.9	★ △ 100.0	100.0	83.3	★ △ 100.0	100.0	92.3	★ ▽ 91.7	★ △ 100.0	100.0
ArtistTerm	87.0	★ △ 100.0	100.0	76.9	★ △ 100.0	100.0	89.7	★ ▽ 88.5	★ △ 100.0	100.0
BankAccount	82.1	★ △ 88.5	88.5	76.7	★ △ 88.5	88.5	85.7	★ ▽ 81.2	★ △ 86.7	86.7
BookTown	92.0	★ △ 97.6	★ ▽ 97.6	87.7	★ △ 97.6	★ ▽ 97.6	82.6	82.6	★ △ 87.4	★ ▽ 85.5
BrowserCookies	89.3	★ △ 92.6	★ ▽ 92.3	88.2	★ △ 92.6	★ ▽ 92.3	88.6	★ ▽ 86.0	★ △ 87.0	★ ▽ 86.5
Cloc	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
CoffeeOrders	91.1	★ △ 100.0	100.0	83.6	★ △ 100.0	100.0	92.1	★ ▽ 86.9	★ △ 94.6	94.6
CustomerOrder	86.8	★ △ 94.0	94.0	80.2	★ △ 93.9	93.9	93.4	★ ▽ 89.3	★ △ 95.2	95.2
DellStore	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Employee	93.2	★ △ 95.3	95.3	91.1	★ △ 95.3	95.3	82.2	82.2	★ △ 84.1	84.1
Examination	95.5	★ △ 97.3	97.3	93.9	★ △ 97.3	97.3	86.2	★ △ 84.4	★ △ 85.8	85.8
Flights	87.5	★ △ 95.5	★ ▽ 95.2	87.5	★ △ 95.5	★ ▽ 95.2	90.5	★ ▽ 84.9	84.9	★ ▽ 84.3
FrenchTowns	88.8	88.8	★ ▽ 82.5	82.1	★ △ 88.8	★ ▽ 82.5	85.2	★ ▽ 82.7	★ △ 89.2	★ ▽ 83.3
Inventory	83.3	★ △ 88.2	★ ▽ 87.5	81.0	★ △ 89.5	★ ▽ 88.2	76.2	76.2	★ △ 80.0	★ ▽ 75.0
Iso3166	70.0	★ △ 77.8	77.8	63.6	★ △ 77.8	77.8	72.7	72.7	★ △ 80.0	80.0
IsoFlav_R2	87.0	87.0	87.0	87.2	87.2	★ ▽ 87.0	85.8	85.8	★ △ 87.0	★ ▽ 84.4
iTrust	92.8	★ △ 95.8	★ ▽ 95.7	90.9	★ △ 95.8	★ ▽ 95.7	82.6	★ ▽ 82.4	★ △ 84.2	★ ▽ 83.6
JWhoisServer	76.1	★ △ 78.7	78.7	73.7	★ △ 78.7	78.7	74.2	74.2	★ △ 76.6	76.6
MozillaExtensions	82.2	★ △ 82.7	★ ▽ 82.1	81.9	★ △ 82.8	★ ▽ 82.1	73.1	73.1	★ △ 73.5	★ ▽ 71.3
MozillaPermissions	96.7	★ △ 100.0	100.0	93.5	★ △ 100.0	100.0	74.2	74.2	★ ▽ 76.7	76.7
NistDML181	90.0	★ △ 100.0	100.0	90.0	★ △ 100.0	100.0	100.0	100.0	100.0	100.0
NistDML182	84.4	★ △ 100.0	100.0	81.5	★ △ 100.0	100.0	100.0	100.0	100.0	100.0
NistDML183	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
NistWeather	85.3	★ △ 93.5	★ ▽ 93.3	82.9	★ △ 93.5	★ ▽ 93.3	93.8	★ ▽ 91.4	★ △ 94.1	★ ▽ 93.8
NistXTS748	88.9	88.9	★ ▽ 88.2	89.5	89.5	★ ▽ 88.2	84.2	84.2	★ △ 88.9	★ ▽ 87.5
NistXTS749	84.0	★ △ 95.5	★ ▽ 95.0	80.8	★ △ 95.5	★ ▽ 95.0	92.1	★ ▽ 89.3	★ △ 92.6	★ ▽ 92.0
Person	77.3	★ △ 81.0	81.0	73.9	★ △ 81.0	81.0	78.3	78.3	★ △ 81.8	81.8
Products	79.7	★ △ 86.5	86.5	76.7	★ △ 86.5	86.5	87.6	★ △ 87.6	★ △ 87.6	★ ▽ 87.1
RiskIt	93.5	★ △ 99.5	★ ▽ 99.5	89.0	★ △ 99.5	★ ▽ 99.5	90.8	★ ▽ 85.7	★ △ 89.7	★ ▽ 89.3
StackOverflow	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
StudentResidence	89.5	★ △ 94.4	94.4	85.0	★ △ 94.4	94.4	84.4	★ ▽ 82.9	★ △ 87.2	87.2
UnixUsage	91.5	★ △ 100.0	100.0	86.0	★ △ 100.0	100.0	95.8	★ ▽ 93.6	★ △ 98.3	★ ▽ 98.2
Usda	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
WordNet	79.2	★ △ 94.1	★ ▽ 93.7	76.6	★ △ 94.3	★ ▽ 93.7	85.0	85.0	★ △ 88.3	★ ▽ 87.4

7 RELATED WORK

Much like this paper, there has been an extensive amount of work that aims to make mutation analysis more efficient and/or more useful. Due to the voluminous nature of work in this area, we survey some representative papers that are related to this paper's technique and its experimental study; a more comprehensive treatment can be found in the survey by Jia and Harman [17]. To start, Wong and Mathur presented an early empirical study that evaluated different strategies for reducing the cost of mutation testing by randomly selecting certain mutants for analysis. Mresa and Bottaci [57] and Offutt et al. [58] followed up this study with new experiments that investigated how selective mutation could speed-up mutation testing by reducing the number of mutants subject to analysis; McCurdy et al. subsequently released an open-source tool to support further experimentation with these techniques [59]. Finally, Ma and Kim showed how clustering can reduce the cost of mutation testing by identifying similar mutants, facilitating the analysis of only the representative ones [60]. While all of these prior papers present ways to improve the efficiency of mutation analysis by discarding mutants, they may yield a mutation score that differs from the original; in contrast, the method presented in this paper will never compromise the mutation score because it only removes ineffective mutants.

Many related papers have attempted to improve the performance of mutation analysis by using either specialized computer hardware or integrated software tools. In early work, both Offutt et al. [61] and Byoungju and Mathur [62] proposed a technique for high-performance mutation testing on a parallel computer. Attempting to improve the software that performs mutation analysis, both DeMillo et al. [63] and Just et al. [64] developed methods that were directly integrated into the compiler for a specific pro-gramming language. In an effort to make the generation of program mutants faster, Untch et al. [65], Ma et al. [66], and Just et al. [67] explored the use of configurable "templates" when manipulating the source code of program mutants. While each of these prior papers improves the efficiency of mutation testing, it does so at the expense of requiring either specialized hardware (e.g., a parallel computer) or customized software (e.g., a Java compiler). This paper's method is distinguished from these prior works in that it creates and executes mutants without needing to modify the DBMS or any other systems software. It is also worth noting that other prior work, like that of Zhang et al. [68] and Just et al. [69], obviate the need for a customized execution environment by applying regressing testing methods [70] that improve the efficiency of mutation testing by reordering the tests. While these methods could be customized for relational database schemas, to date and to the best of our knowledge, none of them can yet handle this new domain.

In the context of using mutation analysis to compare test suite quality, the detection of equivalent program mutants is known to be generally undecidable [71]. Since there is a considerable human and computational cost associated with deciding if a mutant is equivalent [20], prior work has developed approaches that use genetic algorithms [42], compiler optimization [37], [39], constraint-based testing [40], and coverage analysis [20] to detect and remove some equivalent mutants. In addition, Hierons et al. explained how to use program slicing to reduce the computational and human effort needed to determine if a mutant is equivalent [72]. Applying it to the equivalent mutant problem, Hierons and Merayo also presented an algorithm for detecting equivalence between pairs of probabilistic stochastic finite state machines [73]. While these methods may be adapted for databases, none of them currently handle database schema

TABLE 14
Summary of mutation score changes

This table reports the numbers of schemas whose tests, generated by either *Random*[+] or the *AVM*, experienced a decrease or increase for a specific DBMS following the removal of a type of impaired mutant from mutation analysis (resulting in one of the different sets of mutants used in the experiments—for instance, "$-S$, $-(S+I)$ ..."—described in Section 5.3). "▽" denotes the number of schemas for which the mutation score significantly decreased following the removal of an impaired mutant type, while "△" denotes the numbers of schemas for which the mutation score significantly increased. The value following this number in parentheses is the number of schemas for which the significant change was accompanied by a large effect size. "○" denotes the number of schemas that experienced a 100% mutation score after the removal of the impaired mutants. Of the 34 schemas used in the experiments, the percentage of schemas with a 100% score is shown in parentheses. Since there are no automated checks to identify impaired mutants for HyperSQL and PostgreSQL, there is no data for this mutant type and these DBMSs.

| | *Random*[+] | | | | | | | | | | *AVM* | | | | | | | | | |
| | Stillborn $-S$ | Impaired $-(S+I)$ | | | Equivalent $-(S+I+E)$ | | | Redundant $-(S+I+E+R)$ | | | Stillborn $-S$ | Impaired $-(S+I)$ | | | Equivalent $-(S+I+E)$ | | | Redundant $-(S+I+E+R)$ | | |
	○	▽	△	○	▽	△	○	▽	△	○	○	▽	△	○	▽	△	○	▽	△	○
HyperSQL	1 (2.9%)	-	-		0 (0)	19 (18)	1 (2.9%)	2 (2)	5 (5)	1 (2.9%)	5 (14.7%)	-	-		0 (0)	26 (26)	12 (35.3%)	12 (12)	0 (0)	12 (35.3%)
PostgreSQL	1 (2.9%)	-	-		0 (0)	23 (23)	1 (2.9%)	2 (2)	5 (5)	1 (2.9%)	5 (14.7%)	-	-		0 (0)	27 (27)	12 (35.3%)	13 (13)	0 (0)	12 (35.3%)
SQLite	1 (2.9%)	16 (16)	0 (0)	1 (2.9%)	0 (0)	15 (15)	1 (2.9%)	2 (2)	14 (13)	1 (2.9%)	7 (20.6%)	16 (16)	0 (0)	7 (20.6%)	0 (0)	26 (26)	9 (26.5)	15 (15)	0 (0)	9 (26.5%)

mutants. Moreover, the term redundant mutant was previously used by Just et al. [36] to describe Java program mutants that should be removed because they are subsumed by other mutants. We use this term more generally to mean all mutants that are equivalent to other mutants. Finally, Papadakis et al. point out that ineffective mutants—like the ones that this paper's methods can identify and remove—are a validity threat for experiments using mutation analysis to assess the effectiveness of testing techniques [21].

As previously mentioned in Section 2, most work involving the implementation, improvement, and evaluation of mutation analysis methods was originally focused on traditional programs, like those written in programming languages such as Fortran, C, and Java [66], [74], [75]. However, mutation has recently been adopted for a wider range of software artifacts. For instance, the technique developed by Gligoric et al. considers concurrent programs [76]. Moving beyond traditional programs, work such as that of Deng et al. and Lindström et al., proposed the use of mutation analysis to assess the adequacy of test suites for Android apps [26], [27]. Others have recently applied mutation analysis to the measurement of test suite effectiveness for web sites [28], [29], [30]. Mutation testing has additionally been applied in other diverse domains such as mobile software agents (e.g., [31], [32]) and security policies (e.g., [33], [34]). Like these examples of related work, this paper considers mutation testing for a new domain—in this case, relational database schemas. Yet, unlike the aforementioned papers, this one's focus is on the automatic identification and removal of the ineffective mutants that may result in misleading mutation scores and an inefficient mutation analysis.

Since many organizations maintain large databases [2] and the quality of the data in these databases is highly valued by consumers [77], it is worth noting that several examples of prior work have motivated the need for efficient and effective mutation analysis methods for relational database schemas. Experimentally observing that the schema of the database in real-world applications changes frequently, Qui et al. both demonstrated the important role that the relational database schema plays in ensuring the correctness of an application and motivated the need for extensive schema testing [78]. 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" [11]. These aforementioned papers also stressed the importance of efficiently testing relational database schemas with adequate tests; this paper's automated technique for identifying and removing ineffective

mutants help testers achieve this goal by making database schema mutation analysis both faster and more useful than it was when performed with prior methods (e.g., [12]).

While Bowman et al. focused on using mutation analysis to assess test suites for an entire database management system [35], Kapfhammer et al. [12] were the first to propose mutation operators for the integrity constraints expressed in a relational database schema. These suggested operators created mutants by adding, removing, and replacing columns in the definitions of PRIMARY KEY and UNIQUE constraints, while also adding and removing NOT NULL constraints from other columns in the schema's tables. An operator was also proposed to remove CHECK constraints from database schema definitions. Wright et al. [15] extended this set with operators that mutated the predicates of CHECK constraints (e.g., by replacing a relational operator such as > with >=), while also introducing operators to mutate the columns in the definition of FOREIGN KEY constraints.

Other prior work by this paper's authors furnished methods, such as mutant schemata and parallel execution, for speeding up the mutation analysis of database schemas [14]. Wright's dissertation presented a unified treatment of these approaches to efficient schema mutation testing [47]. While this dissertation, and the author's aforementioned work (e.g., [12], [14], [15]), focused on mutating the CREATE TABLE statements that produce the schema, other prior work has proposed mutation operators for the SQL SELECT statements used by applications to retrieve data stored in a database [46], [79]. The idea of mutation analysis for database queries was later incorporated into a tool for instrumenting and testing database applications written in the Java programming language, potentially mutating any executed SELECT statement [80]. Chan et al. also proposed mutation operators for the entity-relationship model managed by a database application [81]. Yet, unlike these aforementioned papers, this paper's methods concentrate on the database schema and are designed to remove the ineffective mutants that make mutation testing slower and less useful.

8 CONCLUSIONS AND FUTURE WORK

Since data is a key driver in business and science, its integrity is of obvious importance [53]. Relational database schemas help to ensure the validity of data through integrity constraints [1]. However, mistakes can be made while specifying schemas, or by misunderstanding the dialect of SQL understood by the DBMS of concern. Therefore, it is important to test relational schemas, as has been recently

recommended by industrial practitioners [11]. Since test cases for database schemas may not be equally capable at finding faults, mutation analysis offers a way to evaluate the "strength" of test suites by inserting potential defects and then checking to see if the tests can find them [12], [13].

Although mutation analysis is known to effectively characterize the quality of tests for programs [82], it is subject to certain concerns [17], to which mutation analysis for schemas is also vulnerable. One issue is the production of useless, ineffective mutants. For instance, a mutant is ineffective if it is equivalent to the original program under test [83]. In the context of using mutation analysis to assess the quality of a program's tests, the detection of equivalent program mutants is known to be generally undecidable [71] and costly from a human perspective [20]. Since these concerns for program mutation also apply to the mutation analysis of database schemas, in this paper we have identified patterns of ineffectiveness in database schemas that lead to equivalent, redundant, and stillborn mutants. We have also discovered a new type of ineffective mutant not heretofore observed in program mutation: the impaired mutant. These impaired mutants are similar to stillborn mutants in that the schema is infeasible. However, instead of being rejected by the database management system (or failing to compile, as would be the case for stillborn mutants with program mutation), they are live until trivially killed by a test.

This paper presented general-purpose algorithms, designed to be run before mutation analysis and implemented in the *SchemaAnalyst* tool, that statically analyze the mutants of database schemas to check if they are ineffective. In an empirical study, focusing on 34 representative database schemas comprising a total of 186 tables, 1044 columns, and 590 constraints that were hosted by the well-known HyperSQL, PostgreSQL, and SQLite DBMSs, we found that a significant number of ineffective mutants could be identified with this automated approach. We also discovered that removing them from the mutant pool often significantly decreased the time needed to perform mutation analysis.

In particular, the prior identification and removal of stillborn mutants was shown to be an order of magnitude faster than relying on the DBMS to reject them during mutation analysis. The efficiency benefits of removing other types of mutant depended on their numbers, and whether the time taken to detect and eliminate them was regained in the course of not having to analyze significant numbers of them later, which could lead to further time savings. Finally, the results also revealed that the removal of ineffective mutants generally changed the mutation score, making it more useful to testers assessing the quality of their tests. In particular, the removal of equivalent mutants sometimes lead to a test suite achieving a perfect mutation score.

Although this paper presents and empirically evaluates a comprehensive suite of methods for automatically detecting and removing ineffective mutants in database schemas, several avenues for future work remain. Yet, these methods could not identify some equivalent and infeasible impaired mutants due to the existence of arbitrary predicates in the database schema's CHECK constraints. Even though the equivalence and infeasibility of predicates is a generally undecidable problem [17], [40], future work will develop methods that can automatically detect simple forms and use them as the basis for removing such mutants—for example, by using a constraint solver. Moreover, even after the removal of ineffective mutants, many mutants remain that are costly to analyze. Future work needs to develop approaches that can speed up their analysis (e.g., through further investigation of virtual execution approaches [84]); or with techniques to reduce the number of mutants that need to be considered by selecting a representative sample, as has been previously proposed for program mutation (e.g., [42], [57], [58]) and preliminarily developed and evaluated for schema mutation [59]. Additionally, we will investigate how this paper's presented methods for the identification and removal of ineffective mutants could be applied to other domains, such as traditional programs and web sites.

There are also many ways in which we intend to improve the empirical study presented in this paper. For instance, we will extend the experiments by considering new database schemas and database-aware mutation operators. This second extension will involve, along with the development of higher-order mutation operators for schemas, the customization of the mutation operators for SQL SELECTs [46] so that they ultimately work for database schemas. We will also more thoroughly investigate how both automatically and manually created tests influence the detection and removal of ineffective mutants. These new experimental configurations will serve to further control the threats to the validity of this paper's experimental study, leading to, for instance, further confirmation of the generalizability of the results.

Once completed, we will integrate all of the new techniques into the existing repository for the *SchemaAnalyst* tool[10]. Overall, the combination of this paper's automated method for handling ineffective mutants, and the improvements completed during future work, will yield an effective way to assess the quality of the test suites for the integrity constraints in a database schema. Ultimately, the use of the algorithms presented in this paper will support the production of better tests suites for schemas, leading to the creation of high-quality relational databases that store the data sets arising in fields such as science and business.

References

[1] P. Glikman and N. Glady, "What's the value of your data?" (Accessed 12/6/2016). [Online]. Available: https://goo.gl/bFZKeR

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

[3] "PostgreSQL featured users," (Accessed 12/6/2016). [Online]. Available: https://www.postgresql.org/about/users/

[4] "Well-known users of SQLite," (Accessed 12/10/2016). [Online]. Available: https://www.sqlite.org/famous.html

[5] K. Roukounaki, "Five popular databases for mobile," (Accessed 12/6/2016). [Online]. Available: https://goo.gl/rAUAe0

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

[7] "DB-Engines DBMS ranking," 2016. [Online]. Available: http://db-engines.com/en/ranking/

[8] D. Robinson, "Releasing the StackLite dataset of Stack Overflow questions and tags," (Accessed 12/6/2016). [Online]. Available: http://varianceexplained.org/r/stack-lite/

[9] C. Marisic, "With the recent prevelance of NoSQL databases why would I use a SQL database?" [Online]. Available: https://goo.gl/ivFu8h
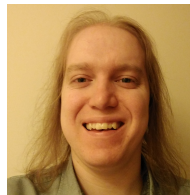
10. https://github.com/schemaanalyst/schemaanalyst

[10] K. Houkjær, K. Torp, and R. Wind, "Simple and realistic data generation," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006.

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

[12] G. M. Kapfhammer, P. McMinn, and C. J. Wright, "Search-based testing of relational schema integrity constraints across multiple database management systems," in *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, 2013.

[13] P. McMinn, C. J. Wright, and G. M. Kapfhammer, "The effectiveness of test coverage criteria for relational database schema integrity constraints," *Transactions on Software Engineering and Methodology*, vol. 25, no. 1, 2015.

[14] C. J. Wright, G. M. Kapfhammer, and P. McMinn, "Efficient mutation analysis of relational database structure using mutant schemata and parallelisation," in *Proceedings of the 8th International Workshop on Mutation Analysis*, 2013.

[15] ——, "The impact of equivalent, redundant and quasi mutants on database schema mutation analysis," in *Proceedings of the 14th International Conference on Quality Software*, 2014.

[16] G. M. Kapfhammer and M. L. Soffa, "Database-aware test coverage monitoring," in *Proceedings of the 1st India Software Engineering Conference*, 2008.

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

[18] G. M. Kapfhammer, "Software testing," in *The Computer Science Handbook*, 2004.

[19] B. Grun, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *Proceedings of the 4th International Workshop on Mutation Analysis*, 2009.

[20] D. Schuler and A. Zeller, "(un-)covering equivalent mutants," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, 2010.

[21] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2016.

[22] P. McMinn, C. J. Wright, C. Kinneer, C. J. McCurdy, M. Camara, and G. M. Kapfhammer, "SchemaAnalyst: Search-based test data generation for relational database schemas," in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, 2016.

[23] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval*, 2011.

[24] "NIST SQL conformance suite," (Accessed 12/14/2016). [Online]. Available: http://www.itl.nist.gov/div897/ctg/sql_form.htm

[25] B. Korel, "Automated software test data generation," *Transactions on Software Engineering*, vol. 16, no. 8, 1990.

[26] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing Android apps," *Information and Software Technology*, vol. 81, 2017.

[27] B. Lindström, S. F. Andler, J. Offutt, P. Pettersson, and D. Sundmark, "Mutating aspect-oriented models to test cross-cutting concerns," in *Proceedings of the 11th International Workshop on Mutation Testing*, 2015.

[28] U. Praphamontripong, J. Offutt, L. Deng, and J. Gu, "An experimental evaluation of web mutation operators," in *Proceedings of the 12th International Workshop on Mutation Testing*, 2016.

[29] S. Mirshokraie, "Effective test generation and adequacy assessment for JavaScript-based web applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2014.

[30] T. A. Walsh, P. McMinn, and G. M. Kapfhammer, "Automatic detection of potential layout faults following changes to responsive web pages," in *Proceedings of the 30th International Conference on Automated Software Engineering*, 2015.

[31] A. A. Saifan and H. A. Wahsheh, "Mutation operators for JADE mobile agent systems," in *Proceedings of the 3rd International Conference on Information and Communication Systems*, 2012.

[32] S. Savarimuthu and M. Winikoff, "Mutation operators for cognitive agent programs," in *Proceedings of the International Conference on Autonomous Agents and Multi-agent Systems*, 2013.

[33] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *Proceedings of the 16th International World Wide Web Conference*, 2007.

[34] T. Mouelhi, F. Fleurey, and B. Baudry, "A generic metamodel for security policies mutation," in *Proceedings of the 3rd International Workshop on Mutation Testing*, 2008.

[35] I. T. Bowman, "Mutatis mutandis: Evaluating DBMS test adequacy with mutation testing," in *Proceedings of the 6th International Workshop on Testing Database Systems*, 2013.

[36] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *Proceedings of the 7th International Workshop on Mutation Analysis*, 2012.

[37] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[38] L. Bottaci, "Type sensitive application of mutation operators for dynamically typed programs," in *Proceedings of the 5th International Workshop on Mutation Analysis*, 2010.

[39] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Journal of Software Testing, Verification, and Reliability*, vol. 4, 1994.

[40] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, 1997.

[41] "SQL as understood by SQLite," https://sqlite.org/lang_createtable.html, [Online; accessed 27-May-2017].

[42] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Genetic and Evolutionary Computation*, 2004, vol. 3103.

[43] M. Kintis and N. Malevris, "MEDIC: A static analysis framework for equivalent mutant identification," *Information and Software Technology*, vol. 68, 2015.

[44] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *Transactions on Software Engineering*, vol. 17, no. 9, 1991.

[45] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, 1994.

[46] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Mutating database queries," *Information and Software Technology*, vol. 49, no. 4, 2007.

[47] C. Wright, "Mutation analysis of relational database schemas," Ph.D. dissertation, University of Sheffield, 2015.

[48] A. Bacchelli, "Mining challenge 2013: Stack overflow," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013.

[49] K. Pan, X. Wu, and T. Xie, "Generating program inputs for database application testing," in *Proceedings of the 26th International Conference on Automated Software Engineering*, 2011.

[50] J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold, "Dynamic invariant detection for relational databases," in *Proceedings of the 9th International Workshop on Dynamic Analysis*, 2011.

[51] B. Smith and L. Williams, "An empirical evaluation of the MuJava mutation operators," in *Proceedings of the Testing: Academic and Industrial Conference - Practice and Research Techniques and the International Workshop on Mutation Analysis*, 2007.

[52] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, 2014.

[53] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 6th ed., 2010.

[54] 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.

[55] G. M. Kapfhammer, P. McMinn, and C. J. Wright, "Hitchhikers need free vehicles! Shared repositories for statistical analysis in SBST," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, 2016.

[56] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, 1978.

[57] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Software Testing, Verification and Reliability*, vol. 9, no. 4, 1999.

[58] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th International Conference on Software Engineering*, 1993.

[59] C. J. McCurdy, P. McMinn, and G. M. Kapfhammer, "mrstudyr: Retrospectively studying the effectiveness of mutant reduction techniques," in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, 2016.

[60] Y.-S. Ma and S.-W. Kim, "Mutation testing cost reduction by clustering overlapped mutants," *Journal of Systems and Software*, vol. 115, 2016.

[61] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation testing of software using a MIMD computer," in *Proceedings of the International Conference on Parallel Processing*, 1992.

[62] C. Byoungju and A. P. Mathur, "High-performance mutation testing," *Journal of Systems and Software*, vol. 20, no. 2, 1993.

[63] R. DeMillo, E. Krauser, and A. Mathur, "Compiler-integrated program mutation," in *Proceedings of the 15th Annual International Computer Software and Applications Conference*, 1991.

[64] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *Proceedings of the 26th International Conference on Automated Software Engineering*, 2011.

[65] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proceedings of the International Symposium on Software Testing and Analysis*, 1993.

[66] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: A mutation system for Java," in *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[67] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using conditional mutation to increase the efficiency of mutation analysis," in *Proceedings of the 6th International Workshop on Automation of Software Test*, 2011.

[68] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2013.

[69] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *Proceedings of the 23rd International Symposium on Software Reliability Engineering*, 2012.

[70] G. M. Kapfhammer, "Regression testing," in *The Encyclopedia of Software Engineering*, 2010.

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

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

[73] R. M. Hierons and M. G. Merayo, "Mutation testing from probabilistic and stochastic finite state machines," *Journal of Systems and Software*, vol. 82, no. 11, 2009.

[74] K. N. King and A. J. Offutt, "A Fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, 1991.

[75] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language," in *Proceedings of the Testing: Academic and Industrial Conference - Practice and Research Techniques and the International Workshop on Mutation Analysis*, 2008.

[76] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2013.

[77] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," *Journal of Management Information Systems*, vol. 12, no. 4, 1996.

[78] D. Qiu, B. Li, and Z. Su, "An empirical analysis of the co-evolution of schema and code in database applications," in *Proceedings of the 21st International Symposium on the Foundations of Software Engineering*, 2013.

[79] G. Kaminski, U. Praphamontripong, P. Ammann, and J. Offutt, "A logic mutation approach to selective mutation for programs and queries," *Information and Software Technology*, vol. 53, no. 10, 2011.

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

[81] W. K. Chan, S. C. Cheung, and T. H. Tse, "Fault-based testing of database application programs with conceptual data model," in *Proceedings of the 5th International Conference on Quality Software*, 2005.

[82] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proceedings of the 39th International Conference on Software Engineering*, 2017.

[83] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[84] P. McMinn, G. M. Kapfhammer, and C. J. Wright, "Virtual mutation analysis of relational database schemas," in *Proceedings of the 11th International Workshop on Automation of Software Test*, 2016.
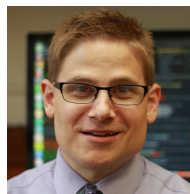
Phil McMinn is a Reader (Associate Professor) in the Department of Computer Science at the University of Sheffield, where he has been researching and teaching software engineering since 2006. His main research interests include search-based software engineering, software testing, program transformation, and reverse engineering.

Chris J. Wright received the PhD degree in computer science in 2016 from the University of Sheffield, United Kingdom. His research interests include mutation analysis and search-based software engineering, with a focus on the use of automated software testing techniques.

Colton J. McCurdy received the BSc degree in computer science in 2017 from Allegheny College. He is a software engineer at StockX with research interests in search-based software engineering and selective mutation testing.

Gregory M. Kapfhammer is an Associate Professor in the Department of Computer Science at Allegheny College. In addition to teaching courses in many technical areas, he conducts research and develops useful tools in the fields of software engineering and software testing.