Virtual Mutation Analysis of Relational Database Schemas

Phil McMinn
University of Sheffield, UK

Gregory M. Kapfhammer
Allegheny College, USA

Chris J. Wright
University of Sheffield, UK

ABSTRACT

Relational databases are a vital component of many modern software applications. Key to the definition of the database schema — which specifies what types of data will be stored in the database and the structure in which the data is to be organized — are integrity constraints. Integrity constraints are conditions that protect and preserve the consistency and validity of data in the database, preventing data values that violate their rules from being admitted into database tables. They encode logic about the application concerned, and like any other component of a software application, need to be properly tested. Mutation analysis is a technique that has been successfully applied to integrity constraint testing, seeding database schema faults of both omission and commission. Yet, as for traditional mutation analysis for program testing, it is costly to perform, since the test suite under analysis needs to be run against each individual mutant to establish whether or not it exposes the fault. One overhead incurred by database schema mutation is the cost of communicating with the database management system (DBMS). In this paper, we seek to eliminate this cost by performing mutation analysis *virtually* on a local model of the DBMS, rather than on an actual, running instance hosting a real database. We present an empirical evaluation of our virtual technique revealing that, across all of the studied DBMSs and schemas, the virtual method yields an average time saving of 51% over the baseline.

1. INTRODUCTION

Relational databases provide a reliable way to store and retrieve data, forming a critical component of a wide range of different software applications, from domains such as Internet browsers (e.g., Chrome[1] and Firefox[2]) to applications powering political campaigns [2]. Despite the recent wave of interest in "NoSQL" technologies, relational databases are still important, relevant and popular in modern software application design, as evidenced by the $876,022$ questions on the popular StackExchange technical question and answer website tagged with labels devoted to the topic.[3]

Key benefits to using a relational database include the good performance of database management systems (DBMSs) [1] — such as PostgreSQL and SQLite, two popular and free DBMSs — and their reliability. Developers often prefer relational databases to other storage and retrieval systems due to the availability of a clear database *schema*, which specifies the data to be stored and how it is structured into tables, serving as easy-to-refer-to documentation.[4]

A relational database schema further involves the definition of a series of *integrity constraints* that guard the validity and consistency of stored data. Integrity constraints ensure that certain data values are unique, through `PRIMARY KEY` constraints and `UNIQUE` constraints; maintain referential integrity with other data values, through `FOREIGN KEY` constraints; are actually present, through `NOT NULL` constraints; and are subject to other arbitrary domain-specific conditions, through `CHECK` constraints. Integrity constraints prevent invalid values being admitted into the database via SQL `INSERT` statements, by causing the DBMS to reject the statement with an error. Integrity constraints encode key application logic, providing the broader software application with a last line of defense against malformed data entries. As such, and in accordance with industry advice [4], they require thorough testing.

To this end, previous work has devised coverage criteria and automated test suite generation approaches for relational database schema integrity constraint testing [9, 11]. Past work has also proposed mutation analysis techniques to seed faults in database schemas that simulate faults of commission and omission, which may be used to evaluate test suites for integrity constraints [8, 17]. However, as with most forms of mutation — including traditional program mutation analysis — these techniques are costly to execute, since the test suite needs to be evaluated against every mutant to determine whether it is capable of exposing the seeded fault.

One cost incurred in evaluating relational database schema test suites is the overhead of communicating with the DBMS that hosts each of the mutant databases, which forms a significant component of the overall time needed to perform mutation analysis. In this paper, we propose to perform mutation analysis *virtually*, on a local model of the DBMS, rather than an actual running instance that incurs constant and costly interaction. Since different DBMSs interpret the SQL standard differently, and often have unique implementation "quirks", a model is required for each different DBMS. In this paper we utilize models for three popular and widely-used DBMSs: HyperSQL, PostgreSQL and SQLite. We empirically show how using virtual mutation instead of the standard method can cut the costs of analysis by 51% on average across all of the studied configurations. Thus, the contributions of this paper are:

1. Avoiding costly DBMS interactions, a new method for performing mutation analysis *virtually* using a local model of three popular and widely-used DBMSs: HyperSQL, PostgreSQL and SQLite.

2. Demonstrating the efficiency trade-offs associated with virtual mutation analysis, an empirical study incorporating nine relational database schemas and the three representative DBMSs.

2. BACKGROUND

Relational Database Schemas. Figure 1 shows SQL `CREATE TABLE` statements for the "NistWeather" schema, which is a part of the NIST SQL conformance test suite[5]. The schema defines

[1]https://www.google.com/chrome/browser
[2]http://www.mozilla.org/firefox
[3]http://goo.gl/F3Tiax
[4]http://goo.gl/v03nUr
[5]http://www.itl.nist.gov/div897/ctg/sql_form.htm

AST'16, May 14-15, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4151-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2896921.2896933

```
 1 CREATE TABLE Station (
 2   ID INTEGER PRIMARY KEY,
 3   CITY CHAR(20),
 4   STATE CHAR(2),
 5   LAT_N INTEGER NOT NULL
 6     CHECK (LAT_N BETWEEN 0 and 90),
 7   LONG_W INTEGER NOT NULL
 8     CHECK (LONG_W BETWEEN SYMMETRIC 180 AND -180)
 9 );
10 CREATE TABLE Stats (
11   ID INTEGER REFERENCES STATION(ID),
12   MONTH INTEGER NOT NULL
13     CHECK (MONTH BETWEEN 1 AND 12),
14   TEMP_F INTEGER NOT NULL
15     CHECK (TEMP_F BETWEEN 80 AND 150),
16   RAIN_I INTEGER NOT NULL
17     CHECK (RAIN_I BETWEEN 0 AND 100),
18   PRIMARY KEY (ID, MONTH)
19 );
```

Figure 1: The NistWeather database schema.

two tables. The "Stats" table (lines 10–19) is for storing rainfall and temperature statistics for a given month pertaining to a particular weather station, the details of which are to be stored in the "Station" table (lines 1–9). Each table involves a number of different columns, each with an associated data type, and a series of integrity constraints, as highlighted in the figure.

Defining integrity constraints protects the validity and consistency of data stored in the tables of the database. For instance, the MONTH column of the Stats table has a "CHECK" constraint defined on it (line 13) that ensures an integer MONTH value can only be between 1 and 12. Further CHECK constraints defined on both tables ensure that other column values are within certain valid ranges (lines 6, 8, 15 and 17). Relational databases allow columns to have missing or unknown values (denoted by the "NULL" marker). To prevent inconsistency (for instance, with the MONTH column) several columns have a "NOT NULL" constraint defined on them, enforcing values to be present for those columns in all rows of the table concerned. Furthermore, the Stats table involves a "foreign key", defined on line 11. Here, "ID" column values in the Stats table must match some value for the ID column in a row of the Station table. Finally, both tables have "primary keys" defined on them (lines 2 and 18). A primary key specifies a set of columns in the table that must have distinct sets of values for each row, and ensures the row is uniquely identifiable. The primary key of the Stats is multicolumn, involving the ID and MONTH columns.

Testing Integrity Constraints. Relational database schemas are an important artifact in a software application, and integrity constraints are a key part of their definition. Poorly or incorrectly specified integrity constraints for a schema may leave an application open to a range of serious failures — for example, non-unique login IDs or negative values for prices or stock levels. For this reason, testing the integrity constraints of a database schema is an important activity that is recommended by industry practitioners [4].

In our previous work, we defined coverage criteria for the integrity constraints of a relational database schema [11]. These coverage criteria mandate the creation of test cases with the aim of demonstrating that the schema has been correctly specified for the purpose of admitting valid values into the database, while also rejecting invalid ones. Each test case is designed to exercise a specific integrity constraint defined for the schema, causing it to either be (a) satisfied, by submitting rows of database values that are valid; or, (b) violated, by submitting rows of values that are invalid.

In practice, each test case takes the form of a minimal number of SQL INSERT statements that encode the rows of values with which the schema will be tested. For instance, the following INSERT statement could be used to check that the integrity constraints of the "Station" table admits certain values as expected. Given an empty table, the values embodied in the following SQL statement should be correctly inserted into the table:

```
INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W)
VALUES(1, 'Austin', 'TX', 30, 98);
```

Further INSERT statements could then be used to test that the table *rejects* certain values as expected, for example using NULL or out of range values for the LAT_N or LONG_W columns, or by attempting to use a value for ID that has already been inserted into the table. It will of course transpire that each of these INSERTs *will* be rejected, since the table has correctly defined NOT NULL, CHECK and PRIMARY KEY constraints that cover each of these cases. Whereas traditional program testing involves assertions over values outputted from a program, database schema testing involves checking that INSERT statements were accepted or rejected as expected. If the acceptance-rejection pattern for a series of INSERT statements differs from that which was expected, a specification error may exist in the definition of the schema.

Mutation Analysis of Relational Database Schemas. Once a test suite has been created, its strength — that is, its potential fault-finding capability — can be estimated using mutation analysis [6]. Mutation analysis involves seeding small faults into the artifact under test to create *mutants* and then checking to find if the test suite behaves differently with the mutant compared with the original artifact. If a difference in behavior is found, the test suite is capable of distinguishing the faulty artifact from the original.

For instance, seeding a fault into the NistWeather schema could take the form of removing the NOT NULL constraint on the MONTH column of the Stats table. In this faulty version, NULL values would be admitted into a database table for the MONTH column that would previously have been rejected, due to violation of the integrity constraint. Thus, an INSERT statement with otherwise valid values for the Stats table, but a NULL for the MONTH column, would be *accepted* with the mutant schema — and yet *rejected* for the original schema. A test suite with a test involving such an INSERT is therefore capable of detecting such a fault, and the mutant is said to be "killed". However, if the test suite did not involve any INSERT statements with NULL for MONTH, the difference between the mutant and the original schema would not be exposed, and the mutant would still be classified as "alive". Such a test suite would have a lower *mutation score* — the number of mutants killed divided by the total number of mutants — and would therefore be regarded as having a weaker fault detection capability.

Previous work [9, 17, 18] has developed a series of mutation operators for the integrity constraints of relational database schemas. These include operators that add, remove and replace columns for constraints defined over one or more columns (e.g., PRIMARY KEY and FOREIGN KEY constraints), add and remove NOT NULL constraints for columns, remove CHECK constraints and alter the relational operators used within them (e.g., substituting ">" for ">=").

Execution Cost of Mutation Analysis. A longstanding problem with mutation analysis is the issue of its high execution cost. The test suite needs to be executed with each mutant, and mutation analysis tends to produce large numbers of mutants due to the large number of potential ways in which faults can be seeded into software artifacts, such as programs and relational database schemas. This problem has prompted several researchers to develop techniques to reduce its execution costs. These were categorized into three groups by Offutt and Untch [13]: "do fewer", "do smarter" and "do faster". As its name suggests, the "do fewer" category involves evaluating a subset of the complete set of mutants according to some selection strategy, while the latter two categories involve

```
function obtain_primary_key_constraint_predicate($pkc(tbl, CL), nr$)
    where
        • $pkc(tbl, CL), nr$ is a PRIMARY KEY constraint
            • $tbl$ is the table on which the constraint is defined
            • $CL = cl_1...cl_n$ is the set of columns of the constraint
        • $nr$ be a new row of data to be inserted into $tbl$
    // The test for null values:
    let $nc_{pk} \leftarrow (nr(cl_1) \neq \texttt{NULL} \wedge \ldots \wedge nr(cl_n) \neq \texttt{NULL})$
    // The test for uniqueness of values:
    let $cc_{pk} \leftarrow (\forall er \in tbl : nr(cl_1) \neq er(cl_1) \vee \ldots \vee nr(cl_n) \neq er(cl_n))$
    // The complete integrity constraint predicate:
    let $icp_{pk} \leftarrow nc_{pk} \wedge cc_{pk}$
    return $icp_{pk}$
end function

function obtain_primary_key_constraint_predicate_for_SQLite ($pkc(tbl, CL), nr$)
    where
        • $pkc(tbl, CL)$ is a PRIMARY KEY constraint for SQLite
            • $tbl$ is the table on which the constraint is defined
            • $CL = cl_1...cl_n$ is the set of columns of the constraint
        • $nr$ be a new row of data to be inserted into $tbl$
    // The test for null values:
    let $nc_{pk} \leftarrow (nr(cl_1) = \texttt{NULL} \vee \ldots \vee nr(cl_n) = \texttt{NULL})$
    // The test for uniqueness of values:
    let $cc_{pk} \leftarrow (\forall er \in tbl : nr(cl_1) \neq er(cl_1) \vee \ldots \vee nr(cl_n) \neq er(cl_n))$
    // The complete integrity constraint predicate:
    let $icp_{pk} \leftarrow nc_{pk} \vee cc_{pk}$
    return $icp_{pk}$
end function
```

Figure 2: Two example integrity constraint predicate functions. This figure gives two functions for obtaining PRIMARY KEY integrity constraint predicates. While the first function is a general definition, the second function applies to the specific behavior of the SQLite database management system. (Adapted from McMinn et al. [11].)

techniques that approximate the result of standard mutation analysis, or otherwise reduce its overall execution cost.

Like mutation analysis for programs, database schema mutation analysis is a time-consuming process. With database schema mutation analysis, one significant addition to the time cost is the overhead of communicating with a DBMS over a network socket. This can be a bottleneck even when the DBMS is running on the same machine as the process conducting the analysis and submitting the relevant SQL commands. In previous work, we sought to minimize the amount of communication that needed to take place, for example by combining all mutants into a single database schema [17]. This approach, however, does not completely eliminate communication costs. In this paper, we present a technique that substantially reduces its execution costs — by evaluating mutants *virtually*.

3. VIRTUAL MUTATION ANALYSIS

Virtual Mutation Analysis of relational database schemas is the use of a DBMS model to perform mutation analysis rather than communicating with a real instance of a DBMS.

In this paper, we derive a model for the HyperSQL, PostgreSQL, and SQLite DBMSs based on previous work [11] in which we modeled the behavior of the integrity constraints of different DBMSs. This model was originally motivated by the desire to derive different coverage criteria for testing the integrity constraints of a relational database schema. In this paper, we show how this same model can be used to evaluate mutants, thus removing the need to communicate with a real instance of a DBMS and so potentially speeding up the time needed to perform mutation analysis.

Modeling the Integrity Constraints of a DBMS. McMinn et al. introduced *integrity constraint predicates* to model the integrity constraints that can be specified as part of a relational database schema for a database management system [11]. Integrity constraint predicates evaluate to *true* when a row of data in an INSERT statement satisfies the constraint, and false when it does not.

Table 1: Schemas analysed in the empirical study.

Schema	Tables	Columns	Checks	Foreign Keys	Not Nulls	Primary Keys	Uniques	$\sum$ Constraints
CoffeeOrders	5	20	0	4	10	5	0	19
Employee	1	7	3	0	0	1	0	4
Inventory	1	4	0	0	0	1	1	2
Iso3166	1	3	0	0	2	1	0	3
JWhoisServer	6	49	0	0	44	6	0	50
MozillaPermissions	1	8	0	0	0	1	0	1
NistWeather	2	9	5	1	5	2	0	13
Person	1	5	1	0	5	1	0	7
Products	3	9	4	2	5	3	0	14
Total	21	114	13	7	71	21	1	113

As per the relational model, originally due to Codd [3], a database table consists of a set of rows with identical column names. We express an individual row as $r = (cl_1 : v_1, \ldots, cl_{ncl} : v_{ncl})$ for a table with ncl columns, where $cl_{1\ldots ncl}$ are the column names and $v_{1\ldots ncl}$ are the values for each column. As a shorthand, we use the notation $r(cl)$ to refer to the value of a column cl for a row r.

Figure 2 shows functions for acquiring predicates for primary keys; "obtain_primary_key_constraint_predicate" gives a predicate for the standard DBMS implementation of a primary key, while "obtain_primary_key_constraint_predicate_for_SQLite" provides an especially customized version for SQLite. SQLite differs from most other DBMSs in that NULL is a legal entry for a primary key column. Both functions take the configuration of a primary key represented as a set of columns CL for a table tbl and a row of data values nr. The predicate returned by the function can then be used to decide, given the current state of the table tbl, whether the values in nr conform to the primary key of tbl or not. For example, the predicate returned for the PRIMARY KEY of the Station table for a non-SQLite DBMS such as PostgreSQL is:

$$(nr(\texttt{id}) \neq \texttt{NULL}) \wedge (\forall er \in \texttt{Station} : nr(\texttt{id}) \neq er(\texttt{id}))$$

That is, the value of the id column in the row nr must not be NULL, and it must not be identical to some other value for id appearing in a row already present in the Station table.

Once predicates have been obtained for all integrity constraints pertaining to a database table[6], an *acceptance predicate* can be formed. An acceptance predicate describes whether a row of data (such as that which forms part of an INSERT statement in a test case) conforms to all of the integrity constraints defined on a table, and as such, whether that row of data will be admitted into the database. An acceptance predicate is formed by the conjunction of each of the individual integrity constraint predicates.

Performing Virtual Mutation Analysis. Once acceptance predicates have been obtained for each of the tables of a database schema, they can be used to perform virtual mutation analysis. Rather than submitting the rows of data in the INSERT statements of a test case directly to the DBMS, rows of data can instead be evaluated by the acceptance predicate relevant to the table that is the subject of the INSERT. With standard mutation analysis, monitoring focuses on the differences in the acceptance and rejection behavior of the DBMS with respect to the INSERT statements submitted as part of each test case. With virtual mutation analysis, however, the monitoring instead considers the difference in truth values of acceptance predicates when evaluated with the data contained within an individual INSERT statement. A difference in truth value of an acceptance predicate with the original schema compared to a mutant indicates that the mutant is killed by the test suite.

[6]See McMinn et al. [11] for a full list of functions for obtaining predicates for integrity constraints used by the schemas and DBMSs that we study in this paper.

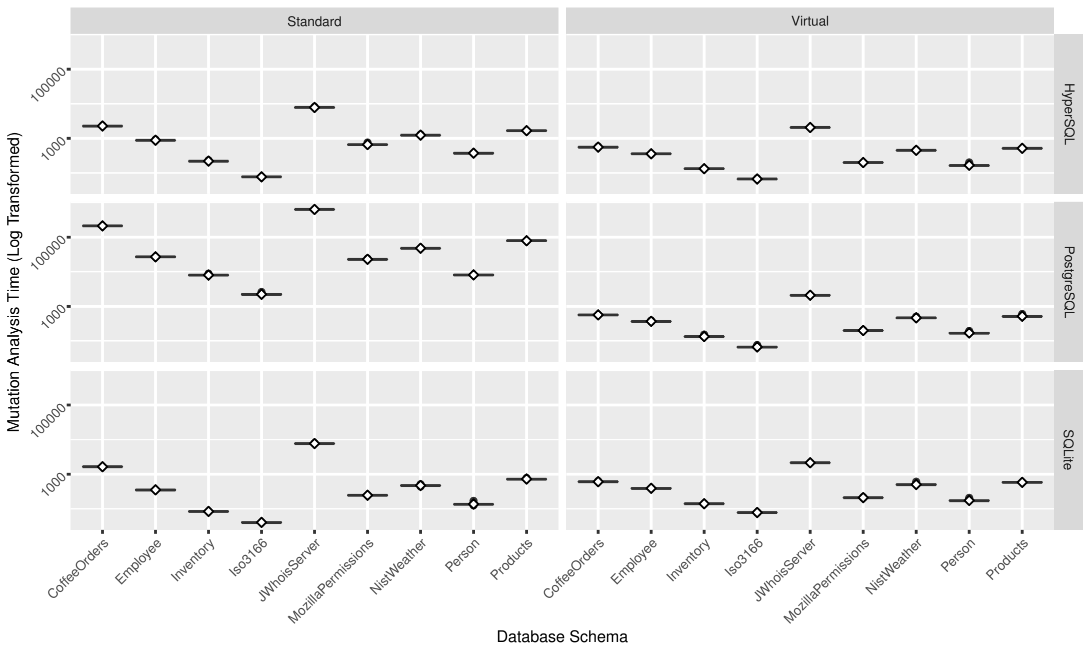

Figure 3: Box plot of the execution time for the standard and virtual mutation analysis techniques. For a detailed description of the meaning of the boxes in this plot, please refer to Section 4. The boxes in this plot are noticeably compressed because there is little variability in the timings for each of the various configurations. Since the results from running the standard method on the PostgreSQL DBMS differ substantially from those with the other techniques and databases, all of the data values were log-transformed, thereby best revealing the relevant trends.

It is worth noting that mutation analysis of schemas involves the repeated execution of tests normally containing a minimal number of `INSERT`s. Yet, even when running these tests, virtual mutation analysis always avoids the need to communicate with an instance of a DBMS that hosts the mutant databases. This saves the cost of setting up the tables of each database for the original schema and each of the subsequent mutants (through SQL `CREATE TABLE`s), executing the `INSERT` statements in the test suite against the database, and then, finally, restoring the state of the DBMS by removing the database tables (through SQL `DROP TABLE` statements).

One disadvantage of virtual mutation analysis is that a model of the operation of the integrity constraints is needed for the DBMS with which we wish to test schemas and perform mutation analysis. While integrity constraints tend to operate in broadly the same way across most DBMSs there is the potential for subtle variations due to differing interpretations of the SQL standard (as shown with the primary key example with SQLite). So, while we have models that are accurate for HyperSQL, PostgreSQL and SQLite, which we use in this paper, new models may be required for other DBMSs.

Related Work. Although this paper is the first to present the idea of virtual mutation, there is a lot of prior work focused on improving the efficiency of mutation analysis; due to space constraints we briefly survey some of the most related approaches. For instance, Just et al. introduced a compiler-integrated method to make mutation testing faster [7]. Also, Tokumoto et al. showed how various techniques, such as the use of virtual machines, can improve the efficiency of mutation testing [14]. Finally, Tuya et al. applied selective mutation to the mutation testing of SQL `SELECT`s [15].

4. EXPERIMENTAL SETUP

To experimentally evaluate the presented virtual mutation analysis technique, we investigate three research questions, comparing it against the standard approach to mutation analysis for relational database schemas, which uses an actual instance of a DBMS (and is hereafter simply referred to as the "standard" method).

RQ1 (Efficiency): How does the time overhead of virtual mutation analysis compare to the standard technique's cost, and how does this vary depending on the DBMS in use?

RQ2 (Time Savings): How do the time savings from using virtual mutation analysis vary when increasing either the number of analysed mutants or the number of executed tests?

RQ3 (Mutation Score): How does the mutation score of virtual mutation analysis compare to the score of a selective method that runs randomly chosen mutants for as long as the virtual one?

Methodology. To answer the first and second research questions, we recorded the time needed to run the standard approach and virtual mutation analysis 30 times each, for each of the subject schemas listed in Table 1 and with each of the three representative DBMSs — HyperSQL, PostgreSQL and SQLite. Two schemas we used appear in open-source projects (i.e., JWhoisServer and MozillaPermissions), while others appear in SQL conformance suites and DBMS sample sets (i.e., NistWeather and Iso3166, respectively). Previous studies have shown that the remaining schemas were challenging to handle for random test generators [11] and the open-source DBMonster tool [9]. Between 9 and 184 mutants were generated for each schema and between 426 and 449 in total for the three DBMSs. These totals correspond to numbers following the removal of certain ineffective mutants — mutants found to be equivalent to the original or some other mutant, or "stillborn" [18]. Due to differing interpretations of the SQL standard, the notion of "equivalence" differs from DBMS to DBMS, leading to a different number of mutants for some of the schemas (e.g., JWhoisServer yields 178 for HyperSQL and PostgreSQL and 184 for SQLite).

For each run, we used a test suite that was automatically generated by a search-based method with a unique random seed. Details of the specific generation algorithms used are given by McMinn et al. [11]. We used the alternating variable method, or *AVM*, since past experiments have shown it to be the most reliable automated method for generating test suites that achieve high levels of test coverage [11]. The coverage criterion we used was a combination of "ClauseAICC", "AUCC" and "ANCC", thus merging the strongest criteria for testing the integrity constraints of schemas.

When the mutation analysis of any part of a database application is too expensive — and a "virtualised" method is unavailable — prior work has performed selective mutation in an attempt to reduce overall analysis costs by, for instance, only evaluating a subset of the mutants [15]. Therefore, to answer the third research question, we developed a strategy like Tuya et al.'s [15] and ran the standard technique 30 times, performing a mutation analysis that randomly selected mutants until the time taken for the corresponding run of the 30 repetitions of virtual mutation analysis was exhausted.

We implemented our virtual model into our *SchemaAnalyst* tool [9, 11, 18], which we also used to perform all of the experiments. *SchemaAnalyst* was compiled with the JDK 7 compiler and executed with the Linux version of the 64-bit Oracle Java 1.7 virtual machine. Experiments were executed on an Ubuntu 14.04 workstation, with a 3.13.0-44 Linux 64-bit kernel, a quad-core 2.4GHz CPU and 12GB of RAM. All input (i.e., the database schemas) and output (i.e., the result 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.

Analysis Methods. Figures 3 and 5 furnish box and whisker plots. In these plots the box itself represents the interquartile range (IQR), or the measure of statistical dispersion that is the difference between the first and third quartiles. Moreover, the upper whisker extends from the top of the box to the highest value that is within 1.5 times the IQR, the lower whisker goes from the bottom of the box to the lowest value within 1.5 times the IQR, and the thick horizontal line represents the median value. Also, these box plots use a filled circle for an outlier and an open diamond for the mean value.

To statistically analyze the trends in Figure 3 we conducted tests for significance with the nonparametric Wilcoxon rank-sum test, using the sets of 30 execution times obtained with a specific DBMS and the standard and virtual mutation analysis techniques. A p-value less than 0.05 is deemed significant. To complement significance tests, the nonparametric $\hat{A}_{12}$ statistic of Vargha and Delaney [16] was used to compute effect sizes, which determine the average probability that one approach "outperforms" another. We followed the guidelines of Vargha and Delaney in that an effect size is deemed to be "large" if the value of $\hat{A}_{12}$ is < 0.29 or > 0.71, "medium" if $\hat{A}_{12}$ is < 0.36 or > 0.64 and "small" if $\hat{A}_{12}$ is < 0.44 or > 0.56. Values of $\hat{A}_{12}$ close to the 0.5 value are viewed as showing no effect. When discussing effect sizes for the execution times of the two methods, we follow Neumann et al. [12] and say that a value of $\hat{A}_{12}$ closer to zero indicates that virtual is the preferred technique while a value near one shows that standard is faster.

As the number of mutants subject to analysis and the number of generated tests increases, Figure 4 plots the percentage of mean time saved from using virtual mutation analysis instead of the standard method. This value is determined by first calculating the mean execution time from the 30 trials of both the standard and the virtual techniques. If T_s denotes the mean time taken by the standard method and T_v is the mean time needed for the virtual one, then we calculated the percentage of mean time saved by $(T_s - T_v)/T_s$.

We employed a correlation statistic to determine how the mutation scores of the selective mutation method correspond to those of virtual mutation analysis. Due to the possibility of rank ties, which are not supported by a number of correlation measures, we chose to use the tie-aware Kendall's τ_b coefficient, as provided by the "Kendall" R package [10]. Kendall's τ_b provides a measurement of correlation between -1 and 1, representing a strong negative and strong positive association, respectively, with 0 indicating that there is no correlation. Following Inozemtseva and Holmes, we adopt the Guildford scale to describe the correlation values, with the absolute value of a coefficient being described as "low" when it is less than 0.4, "moderate" when it is between 0.4 and 0.7, "high" when ranging from 0.7 to 0.9, and "very high" when it is greater than 0.9 [5].

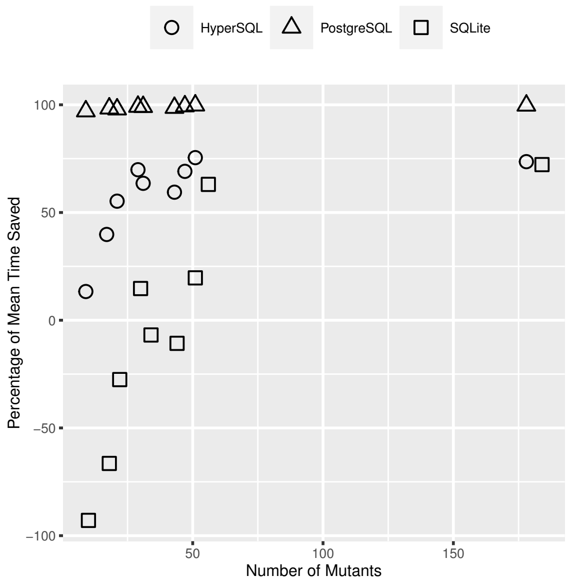

Figure 4: Scatter plot of the percentage of mean time saved for the number of schema mutants subject to analysis.
In these plots, a specific point corresponds to the percentage of mean time saved for a given number of mutants, across all three of the database management systems (a similar graph with the number of tests on the horizontal axis shows the same trend as this graph). For a detailed description of how to calculate the values on the vertical axis, please refer to Section 4.

Threats to Validity. There are several threats to the validity of the empirical results presented in this paper. First, several of the techniques that we used (e.g., the test data generator in *SchemaAnalyst* and the choice of mutants by the selective mutation analysis method) employ randomness. Additionally, background processes running during experimentation may introduce small random variations in the timings for the mutation analysis methods. To control for these threats we ran 30 trials and used box and whisker plots and statistical tests to analyze the results. Adhering to the advice of Neumann et al. [12], we disregarded all timing differences of 100 ms — as they would not be perceived by users of our tool — to ensure that we did not misapply the Vargha-Delaney effect size.

Another threat to the validity of our results is potential defects in the data generator or one of the mutation analysis methods. We controlled these threats by implementing and regularly applying an automated test suite for all of these software tools; to ensure their correct operation we also manually performed "spot checks" on small schemas. In addition, we verified that the virtual mutation analysis always yielded the same mutation score as the standard one. Finally, since possible defects in our results analysis routines would compromise the conclusions from our experiments, we created and regularly ran tests for the R code that manipulated the data, carried out the statistical analyses, and visualised the results.

While the rich and diverse nature of real software systems makes it impossible for us to claim that our schemas are representative of all the characteristics of all possible relational database schemas, we endeavored to select schemas from a wide variety of sources, comprising open-source programs, conformance suites for the SQL

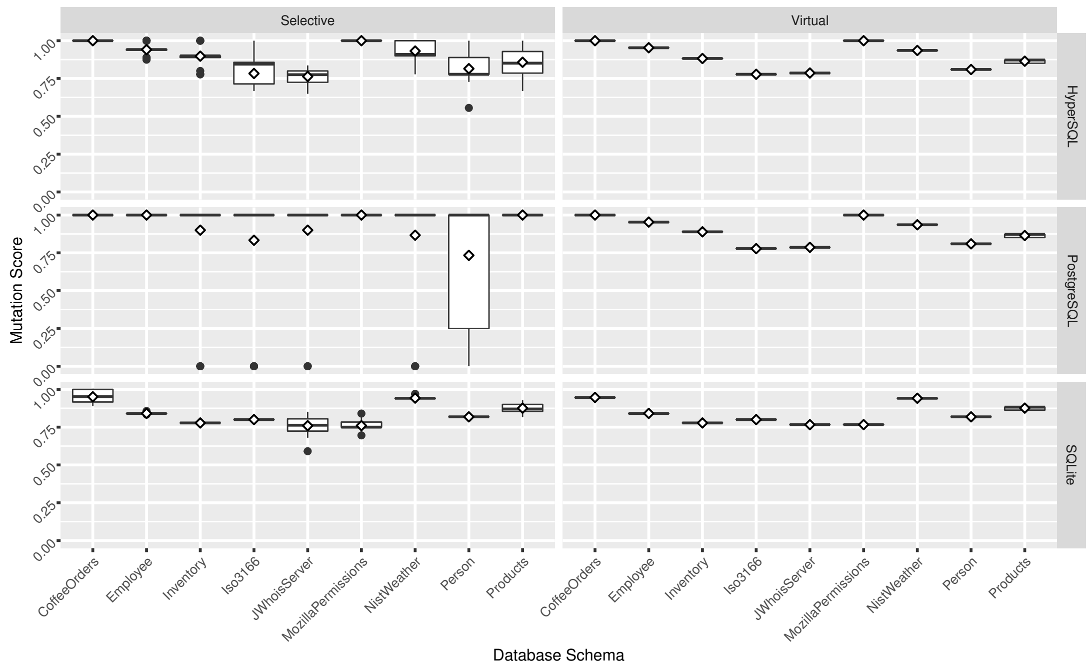

Figure 5: Box plot of the mutation score for the selective and virtual mutation analysis techniques.
Involving test suites from 30 separate runs of the search-based test data generation method developed by McMinn et al. [11], this plot shows the variation in the mutation score for the selective and virtual methods and for all of the chosen relational schemas and the three database management systems.

standard and schemas from databases that were used in previous studies. Table 1 shows the variety captured by the schemas, with tables that contain 1–50 constraints, including `CHECK`s, `FOREIGN KEY`s, `PRIMARY KEY`s, `NOT NULL`s and `UNIQUE`s.

5. EMPIRICAL RESULTS

Comparing Standard and Virtual Mutation. The six box plots in Figure 3 show the mutation analysis time for the two techniques across all of the relational schemas and the three DBMSs. This plot reveals that, when using the HyperSQL DBMS, the virtual method is faster than the standard one, especially for large schemas such as JWhoisServer. Since PostgreSQL is a "heavyweight" disk-based DBMS, virtual mutation analysis — which avoids database interactions — demonstrates much lower execution times with it than does the standard method. Yet, these plots show that the performance of the virtual approach is similar to the standard one when mutation analysis runs on the high-performance SQLite DBMS.

The statistical tests and effect size calculations confirm the trends evident in Figure 3. When comparing the timings for the two mutation analysis methods on the HyperSQL and PostgreSQL DBMSs, the Wilcoxon rank-sum test reveals, with a p-value near zero, that virtual is faster than standard in a statistically significant fashion. Moreover, the $\hat{A}_{12}$ values of 0.26 and 0.0008 for the timings for HyperSQL and PostgreSQL, respectively, show that there is a large effect size evident in the timings and thus sustain virtual mutation analysis as the clear winner for efficiency. Returning a p-value of 0.905, the Wilcoxon rank-sum test confirms that there is no statistical difference between the standard and virtual methods when mutation analysis runs on SQLite. An effect size of 0.503, indicating that the two techniques are stochastically equivalent, further shows that a fast DBMS obviates the benefits of virtual mutation.[7]

[7] Following the suggestions of Neumann et al. [12], we also transformed the effect size values by disregarding all timings differences of 100 milliseconds, ultimately yielding the same conclusions as reported for the untransformed data values.

Saving Time with Virtual Mutation. As evident by the scatter plot in Figure 4, it is worthwhile to see how the time savings associated with using virtual mutation analysis varies as the number of mutants and tests increases. Since PostgreSQL is a heavyweight DBMS relative to HyperSQL and SQLite, this scatter plot reveals that, by avoiding database interactions, the virtual method yields substantial savings regardless of the number of mutants subject to analysis or the number of tests run. Figure 4 also affirms that using virtual mutation on HyperSQL saves time, albeit in a way that is gradual and tapering off as there are more mutants and tests.

The scatter plots also highlight the fact that, when run on SQLite, virtual only improves the performance of mutation analysis for four of the nine schemas. While these larger schemas see reduced overheads with the virtual technique, the 5 smaller schemas do not benefit from the decrease in database interactions afforded by the presented method, thus leading to the negative values of the percentage of mean time saved seen in Figure 4. Yet, even in these cases in which a small schema and a fast DBMS should outperform virtual mutation analysis, we found that the difference in execution time was always less than 100 milliseconds, a negligible amount that experts agree is not perceivable by users of a software tool [12].

Selective and Virtual Mutation. Since the experiments revealed that virtual mutation analysis is faster than the standard one in 22 out of the 27 studied configurations — and competitive with the DBMS-based method in the other 5 — it is useful to ascertain whether the presented technique might yield more accurate mutation scores in some circumstances. To this end, Figure 5 presents the mutation score of both the virtual approach and a time-limited selective analysis in which the standard technique randomly analyzes mutants for as long as virtual would normally take. These box plots show that the selective technique results in mutation scores that are often highly variable. This result can be attributed to the randomness inherent in selectively running mutation analysis under a time limit that will not permit the examination of every mutant.

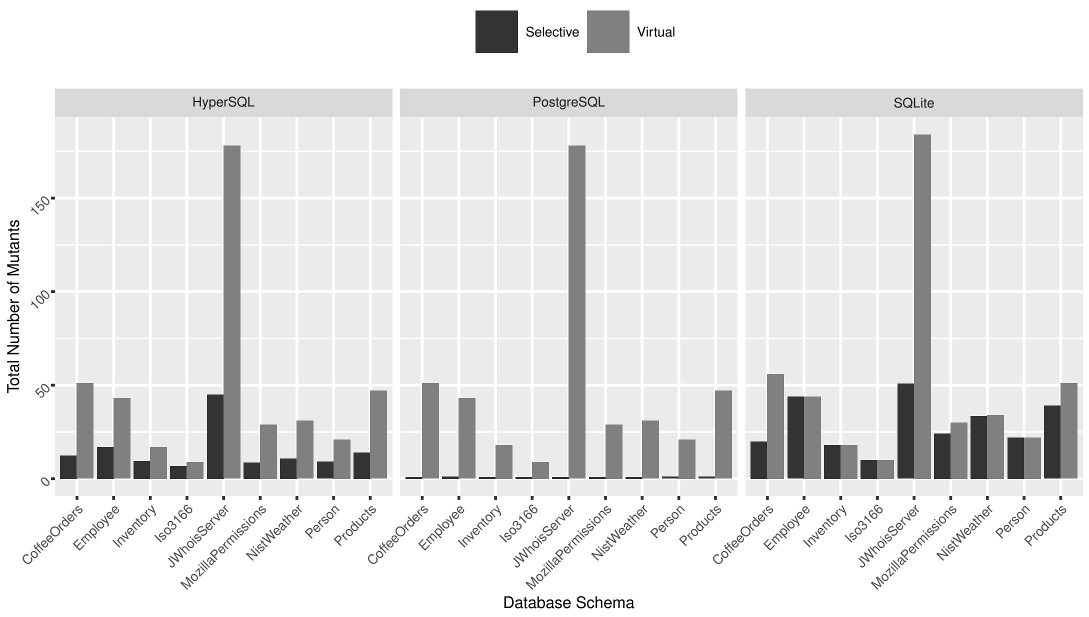

Figure 6: Bar plot of the mutant count for both the selective and virtual mutation analysis techniques. In this plot the height of the bar corresponds to the number of mutants subject to analysis by the selective and virtual methods, reported for all of the schemas and the three DBMSs. Since the selective technique employs randomness to pick mutants that can be run within a specified time limit, the height of a dark grey bar is the average across a total of thirty runs; virtual mutation is deterministic and thus the height of the light grey bar is a direct count.

For instance, the noticeable variability in mutation score when the Person schema is run on PostgreSQL is due to the high probability of selecting a mutant that cannot be killed. The bar chart in Figure 6 further shows the benefit of virtual by revealing that the random selective method rarely analyzes as many mutants as it does, especially for large schemas like JWhoisServer.

Bearing in mind that the virtual method produces mutation scores that are always equal to those achieved by the standard technique, it is also important to observe that mutation analysis through random selection leads to overly high mutation scores. Yet, at least for the HyperSQL and SQLite DBMSs, the box plots in Figure 5 suggest that the mutation scores are roughly similar for virtual mutation analysis and the time-constrained selective method. To rigorously establish this correlation, we calculated Kendall's τ_b for the two techniques on each of the DBMSs, arriving at the values of 0.561 (moderate), 0.132 (low) and 0.756 (high) for HyperSQL, PostgreSQL and SQLite, respectively. These correlations suggest that virtual mutation is the best option when highly accurate scores are needed and there is limited time for mutation analysis of a schema.

6. CONCLUSIONS AND FUTURE WORK

This paper introduces a cost-effective and accurate technique that performs mutation analysis for relational database schemas. Virtual mutation analysis executes test suites virtually against a model of the mutated schema. This novel approach removes the need to setup an instance of a database with the mutated schema on a real DBMS, communicate with the DBMS over a socket connection to set up the database, execute the SQL `INSERT`s of a test case against it, and then tear down the database to prepare the DBMS for the next test. Incorporating nine representative schemas and three industry-standard DBMSs, this paper's experiments reveal that virtual mutation analysis is better than the standard technique in 22 of the 27 configurations studied, yielding a time savings ranging from 13 to 99% — and proving to be competitive with the standard method in the other 5 cases. Given the promise of these results, in future work we plan to ensure our models support other DBMSs and additionally extend the empirical study to include new schemas, thereby furnishing further confirmation of the extensibility and efficiency of virtual mutation analysis for database schemas.

7. REFERENCES

[1] Y. Abrahami. Scaling to 100M: MySQL is a better NoSQL. http://goo.gl/Q9fW5P. (Accessed 11/12/2015).

[2] B. Butler. Amazon: Our cloud powered Obama's campaign. *Net. Wor.*, 2012.

[3] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13, 1970.

[4] S. Guz. Basic mistakes in database testing. http://goo.gl/ByifeQ. (Accessed 24/01/2014).

[5] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. of 36th ICSE*, 2014.

[6] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5), 2011.

[7] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proc. of 6th AST*, 2011.

[8] G. M. Kapfhammer. *A Comprehensive Framework for Testing Database-Centric Applications*. PhD thesis, University of Pittsburgh, 2007.

[9] G. M. Kapfhammer, P. McMinn, and C. J. Wright. Search-based testing of relational schema integrity constraints across multiple database management systems. In *Proc. of 6th ICST*, 2013.

[10] A. McLeod. *Kendall: Kendall rank correlation and Mann-Kendall trend test*, 2011. R package version 2.2.

[11] P. McMinn, C. J. Wright, and G. M. Kapfhammer. The effectiveness of test coverage criteria for relational database schema integrity constraints. *TOSEM*, 25(1), 2015.

[12] G. Neumann, M. Harman, and S. Poulding. Transformed Vargha-Delaney effect size. In *Proc. of 7th SBSE*. 2015.

[13] A. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. *Proc. of Mutation*, 2001.

[14] S. Tokumoto, H. Yoshida, K. Sakamoto, and S. Honiden. MuVM: Higher order mutation analysis virtual machine for C. In *Proc. of 9th ICST*, 2016.

[15] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva. Mutating database queries. *IST*, 49(4), 2006.

[16] A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *EBS*, 25(2), 2000.

[17] C. J. Wright, G. M. Kapfhammer, and P. McMinn. Efficient mutation analysis of relational database structure using mutant schemata and parallelisation. In *Proc. of 8th Mutation*, 2013.

[18] C. J. Wright, G. M. Kapfhammer, and P. McMinn. The impact of equivalent, redundant and quasi mutants on database schema mutation analysis. In *Proc. of 14th QSIC*, 2014.