STICCER: Fast and Effective Database Test Suite Reduction Through Merging of Similar Test Cases

Abdullah Alsharif
University of Sheffield

Gregory M. Kapfhammer
Allegheny College

Phil McMinn
University of Sheffield

***Abstract*—Since relational databases support many software applications, industry professionals recommend testing both database queries and the underlying database schema that contains complex integrity constraints. These constraints, which include primary and foreign keys, `NOT NULL`, and arbitrary `CHECK` constraints, are important because they protect the consistency and coherency of data in the relational database. Since testing integrity constraints is potentially an arduous task, human testers can use new tools to automatically generate test suites that effectively find schema faults. However, these tool-generated test suites often contain many lengthy tests that may both increase the time overhead of regression testing and limit the ability of human testers to understand them. Aiming to reduce the size of automatically generated test suites for database schemas, this paper introduces STICCER, a technique that finds overlaps between test cases, merging database interactions from similar tests and removing others. By systematically discarding and merging redundant tests, STICCER creates a reduced test suite that is guaranteed to have the same coverage as the original one. Using thirty-four relational database schemas, we experimentally compared STICCER to two greedy test suite reduction techniques and a random method. The results show that, compared to the greedy and random methods, STICCER is the most effective at reducing the number of test cases and database interactions while maintaining test effectiveness as measured by the mutation score.**

I. Introduction

Many software applications rely on relational databases for critical data storage, thereby making their correctness paramount and leading industry experts to advise that they be rigorously tested [1]–[3]. Developing a relational database involves the design of a schema that defines its data structures, relationships, and integrity constraints [4]. While a correct schema design ensures the integrity of the data within the database, inadvertent definitions of it (i.e., omitting constraints or adding the wrong constraints) can manifest in a failure that corrupts the data [5]. Testers can use automated test data generators for the integrity constraints in a relational database to support the schema testing process [6], [7]. These generators try to ameliorate the task of writing tests so that developers can focus on developing new features, thereby speeding up the process of testing and producing a more reliable database.

Many real-world database applications contain complex and rapidly evolving database schemas [8]–[10], suggesting the need for efficient approaches to regression testing. Since these schemas contain many tables, columns, and integrity constraints, state-of-the-art automated test data generators for schemas can generate numerous tests because they use local search techniques to cover the test requirements [6], [7]. One approach to the regression testing of a database schema involves re-running the automatically generated tests after a schema modification, with follow-on steps to ensure the test suite's continued effectiveness by adequacy assessment through both coverage and mutation analysis [11], [12]. The prohibitive cost of repeated test suite execution and test adequacy assessment suggests the need for test suite reduction methods that can distill the suite to those tests that are essential for maintaining the schema's correctness during its evolution.

Automatically generated schema tests construct complex database states that are often intertwined, thereby leading to test dependencies that are not explicitly captured by the requirement that the test was designed to cover. Because traditional test suite reduction methods (e.g., [13]–[15]) discard tests only when they cover the same requirements, they are not well-suited to reducing test suites for database schemas. Since Section V's results show that these traditional reducers overlook up to 539 opportunities for test data merging in a complex schema like *iTrust*, this paper presents a novel approach to test suite reduction, called Schema Test Integrity Constraints Combination for Efficient Reduction (STICCER), that discards tests that redundantly cover requirements while also merging those tests that produce similar database states. STICCER creates a reduced test suite, thus decreasing both the number of database interactions and restarts and lessening the time needed for test suite execution and mutation analysis.

Using 34 relational database schemas and test data generated by two state-of-the-art methods, we experimentally compared STICCER to two greedy test suite reduction techniques and a random method. The results show that reduced test suites produced by STICCER are up to 5X faster than the original test suite and 2.5X faster than reduced suites created by the traditional methods, often leading to significant decreases in mutation analysis time. STICCER's tests always preserve the coverage of the test suite and rarely lead to a drop in the mutation score, with a maximum decrease in fault detection of 3.2%. In summary, this paper's contributions are as follows:

1) A novel test suite reduction method, called STICCER, that quickly and effectively reduces database test suites by discarding and merging redundant schema tests.
2) Leveraging 34 relational database schemas running in a popular database engine, an empirical study of STICCER's effectiveness when it reduces test suites from two state-of-the-art test data generators, as compared to two traditional reducers and a random baseline. The results highlight: (i) the limitations of existing methods, (ii) the decrease in tests and database interactions, (iii) the impact on the mutation score, and (iv) the change in the time taken for test execution and mutation analysis.

II. Background

This section introduces relational database schemas, integrity constraints, and the issues associated with testing them. It overviews automated test suite generation methods for integrity constraint testing, and the types of generated tests. It then reviews some traditional methods for test suite reduction and their potential uses and limitations in decreasing the size of the test suite for relational database schema testing.

A. Relational Databases, Schemas and Integrity Constraints

Relational database management systems (DBMSs), such as SQLite [16] or Postgres [17], are enterprise software components that administer one or more of relational databases, each specified by a *schema*. Acting as a guard for the database's correctness, a relational database schema describes how data in a database is structured, what relationships exist between the data, and what types of data values are permissible [4].

Figure 1 highlights one aspect of a relational database: the SQL `CREATE TABLE` statements that are involved in constructing the database's schema. This particular database stores information about website cookies for a web browser, involving two tables. The first `CREATE TABLE` statement describes a table called `cookies`, for storing rows of data where each row corresponds to an individual cookie. Nested within the `CREATE TABLE` statement is a list of *columns* ("`id`" to "`path`") that describe the data to be stored in each row about each cookie, including its name, value, and when it expires. Each column has an associated type, such as a string (using the SQL "`TEXT`" type) or an integer (using the SQL "`INTEGER`").

Underneath the definition of columns (and occasionally inline with a column definition — for example the primary key on the `ID` column) are the declaration of a series of *integrity constraints* on the data in the table. For example, the `name`, `host`, and `path` data elements must be unique for each row of the table when taken in combination and appear as part of a `UNIQUE` integrity constraint definition. Several columns are marked as `NOT NULL`, meaning they cannot be left undefined using `NULL`. There is a foreign key constraint that links the `cookies` table to another table in the schema — `places`, which stores details about websites from which cookies originated. The foreign key declaration specifies that each `host` and `path` pair in the `cookies` table should appear as part of a row in the `places` table. Following the foreign key definition is the declaration of two `CHECK` constraints, involving predicates over the `expiry`, `last_accessed`, and `creation_time` values in each row. Finally, both tables involve primary keys, which like `UNIQUE` constraints, require certain columns to be unique, but with the specific purpose of identifying rows for fast retrieval.

B. The Need for Testing Database Schemas

Despite the large body of work on program testing, there has been much less work devoted to the testing of database-related artefacts that underpin software. One particularly neglected aspect is the database schema, which is often implicitly assumed to be correct [3], but yet is frequently subject to modifications throughout an application's lifetime [2], [8], [9].

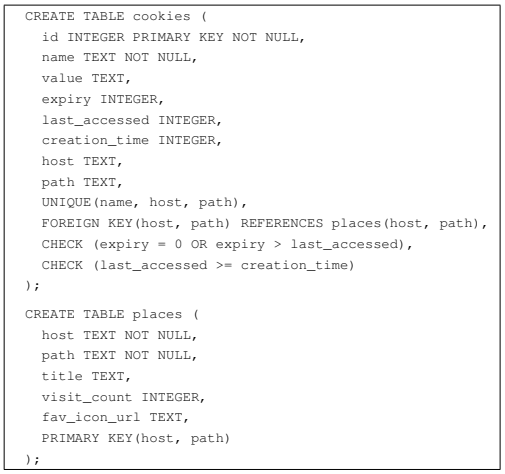
Mistakes made when developing a database's schema can be particularly costly, requiring regression changes to both program code and SQL queries. Additionally, different DBMSs have different interpretations of the SQL standard that developers need to be aware of, thus requiring further testing. For example, SQLite allows `NULL` to be inserted for a primary key in certain circumstances [18], while PostgreSQL forbids this behavior. Most DBMSs do not constrain how many times `NULL` can appear in a column designated as "`UNIQUE`", even though Microsoft SQL Server will only allow it to appear once [19]. Migrating to different DBMS engines is a common task in industry [20]. The use of a fast, lightweight DBMS for development (e.g., SQLite) and the switch to a more robust, enterprise DBMS for deployment (e.g., Postgres) is also a common practice positively encouraged by the Django framework [21] in use for Python-based web development at major companies such as Instagram and Pinterest [22].

One important aspect of a database's schema is the integrity constraints that defend and preserve the consistency and coherency of data — for example, preventing duplicate usernames and non-sensical values such as negative stock levels and prices. Integrity constraints often form the last line of defense against values that might compromise a database's contents, since the database itself may be accessed from a number of applications that may themselves fail to properly implement guards against malformed or incorrect values.

Like other software artefacts, integrity constraints are subject to errors of omission and commission [23]. An example of an omission error is a developer forgetting to add a constraint on a column, such as not defining a `UNIQUE` constraint on a `username` column. Conversely, a commission error would be a developer unintentionally adding an integrity constraint, such as a `UNIQUE` constraint on a column representing somebody's first name (these mistakes may happen in combination, as the unique constraint may have been intended for a column representing some distinctly identifiable information, such as an identification number). For these reasons, industry experts recommend thorough testing of integrity constraints [1]–[3].

C. Testing Integrity Constraints in a Relational Schema

McMinn et al. defined a series of coverage criteria for testing relational database schema integrity constraints [5]. These criteria are based on the idea of testing integrity constraints much like conditions in a program, that is, they are exercised as *true* and *false* by the test suite. In practice, this means designing test cases of SQL `INSERT` statements with data such that an integrity constraint is *satisfied* (i.e., the DBMS admits the data into the database) and *violated* (i.e., the DBMS rejects the data and prevents it from being inserted into the database). For instance, a test that satisfies a `CHECK` constraint would seek to insert data that makes the predicate of the constraint true and a test that seeks to violate the constraint would attempt to insert values that make it false. A test that satisfies a `NOT NULL` constraint would be an `INSERT` statement involving an actual (non-`NULL`) value for the column with the constraint declared on it. Conversely, a violating test would attempt to insert `NULL`.



```
CREATE TABLE cookies (
  id INTEGER PRIMARY KEY NOT NULL,
  name TEXT NOT NULL,
  value TEXT,
  expiry INTEGER,
  last_accessed INTEGER,
  creation_time INTEGER,
  host TEXT,
  path TEXT,
  UNIQUE(name, host, path),
  FOREIGN KEY(host, path) REFERENCES places(host, path),
  CHECK (expiry = 0 OR expiry > last_accessed),
  CHECK (last_accessed >= creation_time)
);
CREATE TABLE places (
  host TEXT NOT NULL,
  path TEXT NOT NULL,
  title TEXT,
  visit_count INTEGER,
  fav_icon_url TEXT,
  PRIMARY KEY(host, path)
);
```

(a) The *BrowserCookies* relational database schema

(i) AVM-D generated test case to satisfy the compound UNIQUE key

A-S_1	`INSERT INTO places(host, path, title, visit_count, fav_icon_url) VALUES ('', '', '', 0, '')`
A-S_2	`INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path) VALUES (0, '', '', 0, 0, 0, '', '')`
A-A_1	`INSERT INTO places(host, path, title,visit_count, fav_icon_url) VALUES ('a', '', '', 0, '')`
A-A_2	`INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path) VALUES (1, '', '', 0, 0, 0, 'a', '')`

(ii) DOMINO generated test case to violate the compound UNIQUE key

D-S_1	`INSERT INTO places(host, path, title, visit_count, fav_icon_url) VALUES ('aqrd', 'xj', 'vnobtpvl', 0, 'dmnofpe')`
D-S_2	`INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path) VALUES (0, 'ddfvkxnjg', '', 0, -801, -890, 'aqrd', 'xj')`
D-A_1	`INSERT INTO places(host, path, title,visit_count, fav_icon_url) VALUES ('vkjdkfc', 'xxfp', 'tp', -640, 'mdewsfaw')`
D-A_2	`INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path) VALUES (261, 'ddfvkxnjg', 'euer', NULL, NULL, NULL, 'aqrd', 'xj')`

(b) Automatically generated test cases using (i) AVM-D and (ii) DOMINO to test the compound UNIQUE constraint on the columns `name`, `host`, and `path`. The test case generated by AVM-D aims to satisfy the constraint, while the one generated by DOMINO violates it.

Fig. 1. The *BrowserCookies* relational database schema with examples of automatically generated test case data.

McMinn et al. described further, finer grained, coverage criteria (e.g., "Clause-Based Active Integrity Constraint Coverage", or ClauseAICC for short) that test individual clauses of CHECK constraints and the presence of certain columns for the integrity constraints that can accommodate them [5]. In experiments with mutation analysis, ClauseAICC was found to be the coverage criterion most sensitive to schema faults when combined with two other criteria, namely "Active Unique Column Coverage" (AUCC) and "Active Null Column Coverage" (ANCC), which exercise the uniqueness of all columns in a database and their NULL/not NULL status, respectively [5]. As such, it is this combination of coverage criteria that the experiments in Section V use to generate the test suites.

D. Automatic Test Data Generation for Relational Schemas

SchemaAnalyst [24] is a tool for automatically generating test cases to satisfy coverage criteria for integrity constraints. *SchemaAnalyst* implements two state-of-the-art test generation algorithms called AVM-D [5] and DOMINO [7]. Both of these algorithms populate a sequence of INSERT statements with test data designed to satisfy a test coverage requirement.

DOMINO works by initializing the test data to random values and then adjusting those values to meet the test requirement [7]. For example, if the test requirement involves violating a primary key, two INSERT statements will be required with the same value for the primary key column. If those values are different, DOMINO will overwrite one of the values so that it matches the other. DOMINO performs similar operations to generate test data for UNIQUE constraints, foreign keys, and NOT NULL constraints. It attempts to satisfy/violate CHECK constraints through pure random generation of values, with the assistance of constant values mined from the schema.

AVM-D is a search-based method that is as a variant of Korel's Alternating Variable Method [25]–[27]. In contrast to DOMINO, it initializes data to pre-defined default values (e.g., zero for integers and empty strings for text fields). It then uses a fitness function to generate values to meet a coverage requirement using traditional search-based distance metrics [28]. For example, if two values x and y are required to be the same (e.g., to violate a uniqueness property), the distance metric is $f = |x - y|$, where smaller values of f denote closeness to the required test data, with zero indicating that the search has found identical values of x and y.

Figure 1b shows examples of test cases generated by AVM-D (part b(i)) and DOMINO (part b(ii)). The test requirement for the AVM-D test case is to satisfy the UNIQUE constraint of the schema involving `name`, `host`, and `path`, while the goal of DOMINO's requirement is to violate it. This figure shows how the test data generated by AVM-D contain repeated "default" values that did not need to be manipulated to meet the test requirement, while DOMINO's test data is varied and random. For each test, assuming an initially empty `cookies` table, "setup" INSERT statements (*-S_1 and *-S_2) are needed to put the database into the required state for testing the constraint — since uniqueness cannot be tested unless there is already some data in the database for comparison purposes.

In both cases, the S_2-suffixed statement inserts data for the `cookies` table, but since it has a foreign key to `places`, a prior INSERT (*-S_1) must first be made to that table. This is so that the test does not fail before the UNIQUE constraint can be tested, as violation of the foreign key is not the focus of these particular tests. Following these "setup" INSERTs are statements referred to as the "action" INSERTs, since they perform the actual test (*-A_2) — or support it through ensuring that the foreign key relationship to the `places` table can be maintained (*-A_1). A-A_2 inserts a different combination of values for `name`, `host`, and `path` to A-S_2, ensuring that the integrity constraint is satisfied, while D-A_2 inserts the *same* values as D-S_2, so that the constraint is ultimately violated.

The coverage criteria for testing integrity constraints necessitate the coverage of many test requirements. Although *SchemaAnalyst* automates the process of generating test cases to satisfy those test requirements, the resultant test suites can still be lengthy, often manipulating database state in an intertwined fashion. This paper investigates ways to reduce the size of these test suites while maintaining their coverage.

	r_1	r_2	r_3	r_4	r_5	r_6
t_1	X	X	X			
t_2	X			X		
t_3		X			X	
t_4			X			X
t_5					X	
	T_1	T_2	T_3	T_4	T_5	T_6

Fig. 2. Example of test cases $\{t_1, \ldots, t_5\}$ and test requirements $\{r_1, \ldots, r_6\}$, an input for test suite reduction methods (excerpted from [29]).

E. Traditional Test Suite Reduction

The task of producing a reduced test suite is equivalent to the minimal set cover problem, which is NP-complete [30]. However, several techniques are capable of effectively reducing test suite size in a way that is useful to developers. We now introduce three such techniques that are implemented as baselines to which, in Section V, we compare the presented approach for reducing relational database schema test suites. We do this by way of the example in Figure 2, which shows a test suite with five test cases $\{t_1 \ldots t_5\}$ and five test requirements $\{r_1 \ldots r_6\}$, where the test cases have different, yet overlapping, coverage of the test requirements.

Random is a technique that is often effective at reducing test suites [31]. This reduction method starts with an empty test suite, adding test cases from the original test suite so long as they cover new test requirements, and continuing until all test requirements are covered. Greedy test suite reduction works in a similar loop to produce a smaller test suite, but instead of selecting test cases at random from the original test suite, it selects the next previously unconsidered test case that covers the most uncovered test requirements [31]. In the example from Figure 2, Greedy selects t_1 first. Since the remaining test cases all cover one remaining requirement, this reduction method will select them at random until all requirements are covered, yielding a reduced test suite of four test cases.

Another well-known approach, called HGS, was developed by Harrold, Gupta, and Soffa [13]. It works by creating test suites containing test cases that cover each test requirement — $T_1 = \{t_1, t_2\}$, covering r_1; $T_2 = \{t_1, t_3\}$ covering r_2, up to $T_6 = \{t_4\}$, covering r_6. HGS starts by adds test cases to the reduced test suite from the test suites $T_1 \ldots T_n$ with cardinality 1. In the example, test suites with cardinality 1 are T_4 and T_6, involving test cases t_2 and t_4, which result in the coverage of $\{r_1, r_4\}$ and $\{r_3, r_6\}$, respectively. HGS then "marks" test suites that also cover these requirements (i.e., T_1 and T_3) so they are not considered by further steps of the algorithm. HGS then repeatedly selects the test cases in unmarked test suites of increasing cardinality. In the example, unmarked test suites of cardinality 2 are T_2 and T_5, with t_3 the only test case to occur in both, and thus added to the reduced test suite. Since t_3 covers r_2 and r_5, all test requirements are now covered, and the algorithm terminates with the reduced test suite containing three tests — one fewer than Greedy. HGS avoids selecting t_1, which is challenging for Greedy, thus leading to Greedy being less successful than HGS at reducing this example test suite. Since HGS has been extended, the interested reader is referred to Yoo and Harman's survey [31] for more details; in this paper we only consider the original HGS algorithm.

III. The STICCER Approach

The more fault-finding and powerful a coverage criteria is, the more test requirements it involves, and the test suites needed to satisfy all of the those test requirements become larger as a result [23]. For example, the *BrowserCookies* schema in Figure 1a has ten integrity constraints. A basic coverage criterion that simply satisfies and violates each constraint would therefore have 20 test requirements, The more complex combination of ClauseAICC, ANCC and AUCC, with higher fault revealing power [5], has 71 requirements. Although *SchemaAnalyst* automates the generation of tests, the test suites can become large since there are many requirements to satisfy. This happens because, unless a test requirement is a duplicate or subsumed by some other, *SchemaAnalyst* treats it as a separate "target" during test case generation [6].

As explained in the last section, and as seen in Figure 1, each test incurs a setup "cost" from running the `INSERT`s that get the database into some state needed to exercise the test requirement. For instance, in Figure 1, the database needs to have some data in it for the `UNIQUE` constraint to be properly tested, as otherwise there will be nothing in the database to test the "uniqueness" of the inserted data forming the last `INSERT` statement of the test. Since the table of interest involves a foreign key, data must also be added to the referenced table to ensure that the test does not fail for reasons other than the `UNIQUE` constraint that it is supposed to test. Furthermore, the database state must be reset following each test so that it does not "pollute" any following tests and introduce unintended behavior or flakiness that might compromise testing [32].

The *first observation* we make, therefore, is that integrity constraint tests often share common sequences of setup-focused `INSERT` statements that could be shared across different test cases to reduce setup/teardown time when running the test suite, resulting in fewer overall `INSERT` statements for a human to understand when maintaining the test suite. For example, Figure 3 shows two test cases, t_1 and t_2, designed to test the inclusion of the two columns of a compound primary key belonging to the `places` table of the *BrowserCookies* schema in Figure 1. The first, t_1, tests the uniqueness of the `host` column while t_2 tests the uniqueness of the `path` column. As the figure shows, the setup part of t_2 can be thrown away (statement t_2S_1), with the "action" part (statement t_2A_1) appended to the end of t_1. The new, merged test case covers both test requirements of the original two tests.

The *second observation* we make is that some `INSERT` statements pertaining to foreign keys in a test case are redundant and can be removed, as in the example of Figure 1b(ii) and the test case generated by DOMINO. Here, the test requirement is to violate the `UNIQUE` constraint of the schema. This involves two `INSERT` statements to the `cookies` table (D-S_2 and D-A_2), with the second `INSERT` (D-A_2) replicating the data values for the columns involved in the constraint of the first (D-S_2), so that they clash with those already in the database. Because the `cookies` has a foreign key to the `places` table, the test case involves `INSERT` statements to that table also (D-S_1

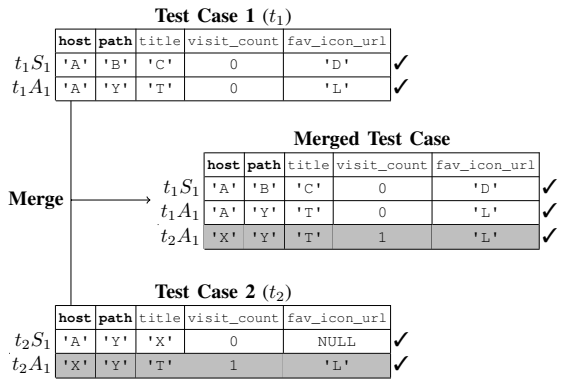


Fig. 3. An example of how STICCER merges two test cases t_1 and t_2 into one test case with the same test coverage behavior. The test cases involve the `places` table of the *BrowserCookies* schema of Figure 1 and aim to satisfy the primary key of the table by exercising the uniqueness of different columns (t_1, the `path` column; t_2, the `host` column). The checkmarks indicate whether a row of data (wrapped into `INSERT` statements in the fully elaborated set of test cases) is successfully admitted into the database.

and D-A_1) — one to support each `INSERT` to `cookies`. This is to ensure that data is present to satisfy the foreign key relationship (since violating this constraint would mean that the test requirement for this test would not be fulfilled). Yet, in this particular case D-A_1 is redundant. Since the columns of the `UNIQUE` constraint are also involved in the foreign key, and both `INSERT` statements to `cookies` must have the same data for those columns, those `INSERT` statements both rely on D-S_1 to fulfil the foreign key relationship. Note that this is not always the case, since the corresponding `INSERT` to the `places` table is needed for the AVM-D test case in part b(i), as the test requirement there is to satisfy the unique constraint, and as such D-S_2 and D-A_2 need to be distinct, which in turn means that the foreign key values must also be distinct.

To handle both of these issues as part of an automated approach to reduce the size of relational database schema integrity constraint tests, we present a technique called "STICCER", which stands for "Schema Test Integrity Constraints Combination for Efficient Reduction". STICCER builds on the standard greedy approach to test suite reduction by merging tests (or "sticking" them together) — thereby sharing setup statements (and the associated teardown costs following the test) — and by also removing redundant `INSERT` statements.

The overall algorithm for STICCER works as follows. The first step removes redundant `INSERT` statements from the existing set of test cases, through a function we refer to as REMOVEREDUNDANTINSERTS. This function checks all `INSERT` statements made to foreign key tables, and ensures that the foreign keys are actually referenced by other `INSERT`s in the test. If they are not, those `INSERT` statements are redundant, and therefore are cut out of the test case under consideration.

STICCER then performs a greedy reduction on the test suite, before moving into the test case merging stage. STICCER iterates through the test suite comparing each test t_1 with each other test t_2. STICCER then checks for a potential merge through a function called CHECKMERGE. The primary role of CHECKMERGE is to assess if the proposed test for merging will cover the same test requirements as the original two tests. If the first test, t_1, leaves the database state in such a way that the behavior of the "action" `INSERT` statements (i.e., t_2A_1 in the example of Figure 3) will behave differently after merging (i.e., the merged test does not cover the same test requirements as t_1 and t_2 combined), CHECKMERGE will reject the potential merge. STICCER continues iterating through the test suite, checking the remaining tests with the newly created test case for further merge possibilities. To better ensure that STICCER does not produce overly long, unwieldy tests that are difficult to understand and maintain, CHECKMERGE will only merge tests if the following, further conditions are met:

1. *The coverage requirements involve the same database table.* CHECKMERGE will not merge two tests if they are designed to test integrity constraints belonging to different tables. Each test must focus on the integrity constraints of one table only.

2. *The tests are intended to both satisfy or both violate aspects of the schema.* To simplify the intentions of each test, and make it easier to maintain and understand, each merged test will only attempt to satisfy or to violate the integrity constraints of a particular table in the schema. Under this condition, the human tester/maintainer knows what type of behavior to expect from the `INSERT` statements of each of the final tests — that is, whether the data in them is intended to be accepted by the DBMS, or whether they are all supposed to be rejected.

3. *The tests both involve database setup, or they both do not.* Some tests do not involve any database setup at all (e.g., `CHECK` constraints, where the predicate is only concerned with the current row of data, rather than the state of the database), and these tests should not be merged together with those that do.

If CHECKMERGE permits a merge of tests, a function called MERGE then actually performs the merge, removing the setup `INSERT` statements from t_2, and appending it to the end of t_1. This test may undergo further merges with other tests in the suite as they are considered in turn by STICCER.

We implemented STICCER into *SchemaAnalyst*, which has functionality to statically analyze what test requirements are covered by a test case [24], and we made use of this to check merged tests when implementing the presented technique. Also, we integrated the Random, Greedy, and HGS reduction techniques into *SchemaAnalyst*, making the complete system available at https://github.com/schemaanalyst/schemaanalyst.

IV. EMPIRICAL EVALUATION

This section evaluates STICCER by comparing it to other test suite reduction techniques in an empirical study that seeks to answer the following three research questions:

RQ1: Reduction Effectiveness. How effective is STICCER at reducing the number of test cases and `INSERT`s within each test case, compared to other test suite reduction techniques, while preserving the test requirements of the original suite?

RQ2: Impact on Fault Finding Capability. How is the fault-finding capability of the test suites affected following the use of STICCER and other test suite reduction techniques?

RQ3: Impact on Test Suite and Mutation Analysis Runtime. How are the running times of the reduced test suites and subsequent mutation analysis affected by test suite reduction?

A. Methodology

1) Schema Subjects: We performed our experiment on a diverse set of 34 schemas listed by Table I, which have featured in previous research on schema testing [5]–[7], [12], and are drawn from several sources, including open-source programs (e.g., *JWhoisServer*, *MozillaExtensions*, and *WordNet*), government projects (e.g., *IsoFlav_R2*, *NistDML181*, and *Usda*), and examples from websites (e.g., *DellStore*, *FrenchTowns* and *Iso3166*). The set of relational schemas varies in both size and complexity, with schemas ranging from having 1–42 tables, 3–309 columns, and 1–134 constraints. In particular, all types of integrity constraint are represented in this set of schemas, including those with compound primary keys, `UNIQUE` constraints and foreign keys, and `CHECK` constraints.

2) Experiments: To study STICCER, we generated test suites using the AVM-D and DOMINO test generation algorithms (introduced in Section II-D) for the 34 schemas using the ClauseAICC+ANCC+AUCC coverage criterion combination (introduced in Section II-C as the coverage criterion found to have the strong fault-finding capability in previous work [5]). We used the implementations of these techniques found in the publicly available *SchemaAnalyst* tool (introduced in Section II-D), and configured *SchemaAnalyst* to generate the test suites with the well-known SQLite as the target DBMS. Due to the stochastic nature of AVM-D and DOMINO, we repeated the generation of the test suites 30 times. We set *SchemaAnalyst* to use a maximum of 100,000 iterations for the test data search for each test requirement.

To answer RQ1 we compare STICCER at reducing the test suites generated by *SchemaAnalyst* with implementations of the Random, Greedy, and HGS methods (introduced in Section II-E). To ensure fairness of comparison with STICCER, our implementations of these techniques also removed redundant `INSERT` statements from test suites (using the REMOVEREDUNDANTINSERTS function discussed in Section IV-A). We calculated the effectiveness of reduction for the test suite size and number of `INSERT`s using Equation 1.

$$(1 - \frac{\textit{No. of test cases in the reduced test suite}}{\textit{No. of test cases in the original test suite}}) \times 100 \quad (1)$$

We report the median values of this equation for each reduction method for the 30 test suites generated for each schema with the two test generation methods, and calculate statistical significance and effect sizes as detailed in the next subsection. We also report the number of merges STICCER could perform while reducing the test suites created by the two different test generation techniques, AVM-D and DOMINO.

To answer RQ2, we investigate the fault-finding capability of the reduced test suites using mutation analysis. We used the mutation analysis techniques implemented into *SchemaAnalyst*, which adopt Wright et al.'s [33] mutation operators for integrity constraints. These operators add, remove, and exchange columns in primary key, unique, and foreign key constraints, invert `NOT NULL` constraints, remove `CHECK` constraints and mutate their relational operators. We calculated the mutation score for each reduced test suite, a percentage of mutants that are "killed" (i.e., detected) by the tests.

TABLE I
THE RELATIONAL DATABASE SCHEMAS STUDIED

			Integrity Constraints					
Schema	Tables	Columns	Check	Foreign Key	Not Null	Primary Key	Unique	Total
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

To answer RQ3, we tracked the times needed by each reduction algorithm to reduce test suites, and the time needed to perform mutation analysis using the reduced suites.

We performed all experiments on a Linux workstation running Ubuntu 14.04 with a 3.13.0–44 GNU/Linux 64-bit kernel, quad-core 2.4GHz CPU, and 12GB of RAM. We used SQLite version 3.8.2 with "in-memory" mode enabled.

B. Statistical Analysis

Because the presented techniques are stochastic, and since recording the wall-clock timings for the experiments is potentially subject to interferences that we cannot control (e.g., operating system interrupts), we repeated our experiments 30 times. As we cannot make any assumptions about the normality of the resulting distributions, we apply non-parametric statistical measures, including the Mann-Whitney U-test and the $\hat{A}$ effect size metric of Vargha and Delaney [34], as recommended by Arcuri and Briand in their guide to statistics and software engineering experiments [35]. In our tables we report if a technique was statistically better or worse to some other by formatting the statistics for those techniques in bold, using the "▼" symbol if the secondary technique was *significantly worse* and the "▲" symbol if it was *significantly better*, and an asterisk if the effect size was large (i.e., $\hat{A} < 0.29$ or > 0.71), following the suggested cut-offs from Vargha and Delaney [34]. Supporting the replication of our analyses, we performed these calculations using the R language for statistical computation, making the scripts and raw data publicly available at https://github.com/schemaanalyst/sticcer-replicate.

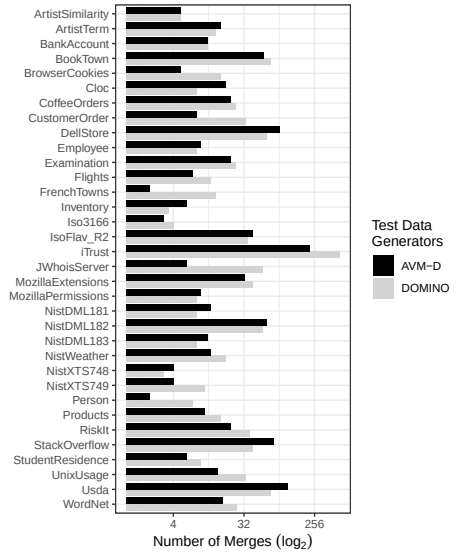


Fig. 4. Median number of test case merges made by STICCER (log_2 scale)

C. Threats to Validity

Our empirical study used a diverse set of schemas taken from past studies [5], [7], [36], [37]. While these schemas support the claims we make in the following sections, it is impossible to claim that they are representative of all schemas, and obtaining such a suitably representative set is equally difficult. However, our set of schemas are derived from a variety of sources, with a broad range of sizes and complexities. Other validity threats include the stochastic behavior of the test data generators (and the possibility our results are obtained by chance and are thus unrepresentative) and the use of wall-clock timings, which are subject to interferences that are out of our control. To mitigate both of these issues, we repeated the experiments 30 times, following the advice of Arcuri and Briand [35] to mitigate errors in the statistical analysis of our results, for example by using non-parametric hypothesis tests.

We implemented these analyses in R, using unit tests to gain confidence in their correctness. Finally, we controlled threats arising from defects in our implementation of STICCER and the reduction techniques by checking the results on selected schemas and, where appropriate, by writing unit tests. For instance, since the presented methods aim to reduce test suites while maintaining test coverage, we confirmed that all reduction was achieved without lowering the test coverage.

V. Experimental Results

RQ1: Reduction Effectiveness

Figure 4 shows the median number of test case merges that STICCER made with tests generated by either AVM-D or DOMINO. The highest count is 539 merges for *iTrust*, with test suites generated by DOMINO. The *iTrust* schema is the largest we studied (see Table I), with the greatest number of integrity constraints (134). The schema with the next greatest number of merges is *BookTown*, with 70 merges, for test suites generated by DOMINO. *BookTown*'s original test suite size is 269. The smallest number of merges was 2 for *Person*, for test suites generated by AVM-D, which has an original test suite size of 20. Figure 4 shows the distribution of merge opportunities that were unavailable to the other, traditional reduction techniques we applied to database schema test suites, which only remove test cases on the basis of overlapping coverage requirements.

Table II shows the median effectiveness of each of the reduction techniques at decreasing the number of test cases for each schema in each of the 30 test suites generated by AVM-D and DOMINO respectively. Table III shows the effectiveness of each reduction technique at reducing the overall number of SQL statements (i.e., `INSERT`s) in those reduced suites.

As the summary statistics show, STICCER was the most effective at reducing the test suites, achieving up to a 93% reduction for the *StackOverflow* schema and a minimum 37% for *JWhoisServer* for tests generated by AVM-D. With tests generated by DOMINO, STICCER achieved a maximum reduction of 89% with *NistDML182* and a minimum of 58% for *Iso3166*. It is worth noting that this minimum figure is greater than the maximum achievable with Random for DOMINO-produced test suites and only 6% lower than the maximums achieved by Greedy and HGS, respectively. STICCER created reduced test suites that were statistically significantly smaller than the original test suites and those reduced by the other methods for all database schemas and with a large effect size.

Table III shows that the reduction in test cases achieved by STICCER was not the result of it naively concatenating SQL statements from each constituent test case — STICCER also reduced the number of overall `INSERT` statements in the resulting test suites. As Table III shows, the average reduction in the number of `INSERT`s (i.e., the constituent statements making up each test case) that STICCER achieved was greater than that of any of the other three reduction techniques studied.

On average, STICCER is more effective with test suites generated by DOMINO than AVM-D, a fact also shown by Figure 4. This is because the default values used by AVM-D are repeated across `INSERT` statements, which makes it more difficult to merge them together across different test cases. The repeated values inadvertently trigger primary key and `UNIQUE` constraint violations when the same values appear for different `INSERT` statements from different test cases for particular columns. This results in a combined test case with different coverage requirements compared to its constituent originals — test cases that will be disregarded by STICCER.

Comparing the average of the median reduction scores for each schema, HGS is the next most effective reduction technique following STICCER. Notably, STICCER achieves an average reduction of 66% and 74% for test suites generated by AVM-D and DOMINO, respectively, while HGS achieves comparatively lower scores of 46% and 50%. HGS also performed worse overall with test suites generated by AVM-D compared to DOMINO, but the differences we observed were not as marked as those with STICCER. It seems that DOMINO is capable of producing test cases that cover more distinct sets of requirements than AVM-D, and with less of an intersection with other tests, thereby making them more amenable for reduction techniques that aim to remove redundant test cases.

TABLE II
MEDIAN REDUCTION EFFECTIVENESS FOR TEST SUITES

Test suite reduction effectiveness is calculated using the higher–is-better formula of Equation 1. The "▼" symbol indicates that a technique's reduction score was significantly less than STICCER's, while "▲" indicates that it was significantly greater. The "∗" symbol denotes a large effect size for a technique when compared with STICCER. The numbers in brackets denote the median number of test cases in the reduced suites as a fraction of those in the original, unreduced test suite generated by either AVM-D or DOMINO.

	AVM-D								DOMINO							
Schema	STICCER		Random		Greedy		HGS		STICCER		Random		Greedy		HGS	
ArtistSimilarity	63%	(7/19)	∗▼**32%**	(13/19)	∗▼**32%**	(13/19)	∗▼**42%**	(11/19)	63%	(7/19)	∗▼**32%**	(13/19)	∗▼**37%**	(12/19)	∗▼**32%**	(13/19)
ArtistTerm	65%	(15/43)	∗▼**28%**	(31/43)	∗▼**30%**	(30/43)	∗▼**37%**	(27/43)	65%	(15/43)	∗▼**28%**	(31/43)	∗▼**30%**	(30/43)	∗▼**30%**	(30/43)
BankAccount	68%	(12/37)	∗▼**35%**	(24/37)	∗▼**38%**	(23/37)	∗▼**43%**	(21/37)	70%	(11/37)	∗▼**38%**	(23/37)	∗▼**43%**	(21/37)	∗▼**49%**	(19/37)
BookTown	58%	(113/269)	∗▼**35%**	(175/269)	∗▼**38%**	(167/269)	∗▼**46%**	(144/269)	68%	(87/269)	∗▼**37%**	(168/269)	∗▼**42%**	(156/269)	∗▼**49%**	(138/269)
BrowserCookies	62%	(27/71)	∗▼**51%**	(35/71)	∗▼**56%**	(31/71)	∗▼**59%**	(29/71)	76%	(17/71)	∗▼**46%**	(38/71)	∗▼**55%**	(32/71)	∗▼**56%**	(31/71)
Cloc	90%	(4/40)	∗▼**43%**	(23/40)	∗▼**48%**	(21/40)	∗▼**50%**	(20/40)	85%	(6/40)	∗▼**48%**	(21/40)	∗▼**51%**	(20/40)	∗▼**57%**	(17/40)
CoffeeOrders	64%	(32/90)	∗▼**37%**	(57/90)	∗▼**41%**	(53/90)	∗▼**42%**	(52/90)	74%	(23/90)	∗▼**39%**	(55/90)	∗▼**44%**	(50/90)	∗▼**47%**	(48/90)
CustomerOrder	40%	(76/126)	∗▼**32%**	(86/126)	∗▼**33%**	(84/126)	∗▼**37%**	(79/126)	63%	(46/126)	∗▼**33%**	(85/126)	∗▼**35%**	(82/126)	∗▼**39%**	(77/126)
DellStore	86%	(24/177)	∗▼**33%**	(118/177)	∗▼**35%**	(115/177)	∗▼**38%**	(110/177)	71%	(52/177)	∗▼**36%**	(113/177)	∗▼**41%**	(105/177)	∗▼**42%**	(102/177)
Employee	74%	(10/38)	∗▼**46%**	(20/38)	∗▼**50%**	(19/38)	∗▼**50%**	(19/38)	80%	(8/38)	∗▼**53%**	(18/38)	∗▼**61%**	(15/38)	∗▼**61%**	(15/38)
Examination	72%	(30/107)	∗▼**50%**	(54/107)	∗▼**51%**	(52/107)	∗▼**52%**	(51/107)	87%	(14/107)	∗▼**57%**	(46/107)	∗▼**64%**	(38/107)	∗▼**64%**	(38/107)
Flights	69%	(19/62)	∗▼**49%**	(32/62)	∗▼**58%**	(26/62)	∗▼**58%**	(26/62)	71%	(18/62)	∗▼**45%**	(34/62)	∗▼**52%**	(30/62)	∗▼**53%**	(29/62)
FrenchTowns	40%	(32/53)	∗▼**32%**	(36/53)	∗▼**36%**	(34/53)	∗▼**34%**	(35/53)	60%	(21/53)	∗▼**32%**	(36/53)	∗▼**32%**	(36/53)	∗▼**34%**	(35/53)
Inventory	67%	(6/18)	∗▼**33%**	(12/18)	∗▼**39%**	(11/18)	∗▼**44%**	(10/18)	78%	(4/18)	∗▼**44%**	(10/18)	∗▼**56%**	(8/18)	∗▼**56%**	(8/18)
IsoFlav_R2	75%	(45/177)	∗▼**45%**	(96/177)	∗▼**49%**	(90/177)	∗▼**50%**	(88/177)	81%	(34/177)	∗▼**52%**	(85/177)	∗▼**60%**	(70/177)	∗▼**62%**	(66/177)
Iso3166	58%	(5/12)	∗▼**25%**	(9/12)	∗▼**33%**	(8/12)	∗▼**33%**	(8/12)	58%	(5/12)	∗▼**29%**	(8/12)	∗▼**33%**	(8/12)	∗▼**42%**	(7/12)
iTrust	57%	(646/1517)	∗▼**38%**	(934/1517)	∗▼**43%**	(872/1517)	∗▼**44%**	(847/1517)	85%	(235/1517)	∗▼**44%**	(849/1517)	∗▼**49%**	(776/1517)	∗▼**50%**	(754/1517)
JWhoisServer	37%	(100/158)	∗▼**31%**	(109/158)	∗▼**33%**	(106/158)	∗▼**35%**	(103/158)	70%	(48/158)	∗▼**32%**	(107/158)	∗▼**35%**	(103/158)	∗▼**37%**	(99/158)
MozillaExtensions	75%	(57/229)	∗▼**51%**	(112/229)	∗▼**60%**	(92/229)	∗▼**50%**	(115/229)	85%	(35/229)	∗▼**55%**	(102/229)	∗▼**63%**	(84/229)	∗▼**64%**	(83/229)
MozillaPermissions	73%	(9/33)	∗▼**42%**	(19/33)	∗▼**48%**	(17/33)	∗▼**48%**	(17/33)	88%	(4/33)	∗▼**55%**	(15/33)	∗▼**58%**	(14/33)	∗▼**64%**	(12/33)
NistDML181	79%	(8/38)	∗▼**47%**	(20/38)	∗▼**53%**	(18/38)	∗▼**53%**	(18/38)	76%	(9/38)	∗▼**45%**	(21/38)	∗▼**55%**	(17/38)	∗▼**58%**	(16/38)
NistDML182	89%	(20/190)	∗▼**53%**	(90/190)	∗▼**57%**	(82/190)	∗▼**57%**	(81/190)	89%	(21/190)	∗▼**52%**	(90/190)	∗▼**60%**	(76/190)	∗▼**62%**	(73/190)
NistDML183	82%	(6/34)	∗▼**44%**	(19/34)	∗▼**47%**	(18/34)	∗▼**53%**	(16/34)	76%	(8/34)	∗▼**41%**	(20/34)	∗▼**50%**	(17/34)	∗▼**53%**	(16/34)
NistWeather	64%	(20/56)	∗▼**39%**	(34/56)	∗▼**45%**	(31/56)	∗▼**43%**	(32/56)	82%	(10/56)	∗▼**39%**	(34/56)	∗▼**45%**	(31/56)	∗▼**46%**	(30/56)
NistXTS748	62%	(6/16)	∗▼**34%**	(10/16)	∗▼**38%**	(10/16)	∗▼**44%**	(9/16)	69%	(5/16)	∗▼**41%**	(10/16)	∗▼**50%**	(8/16)	∗▼**50%**	(8/16)
NistXTS749	57%	(15/35)	∗▼**43%**	(20/35)	∗▼**46%**	(19/35)	∗▼**51%**	(17/35)	69%	(11/35)	∗▼**40%**	(21/35)	∗▼**49%**	(18/35)	∗▼**49%**	(18/35)
Person	50%	(10/20)	∗▼**30%**	(14/20)	∗▼**40%**	(12/20)	∗▼**40%**	(12/20)	80%	(4/20)	∗▼**30%**	(14/20)	∗▼**30%**	(14/20)	∗▼**35%**	(13/20)
Products	67%	(17/52)	∗▼**40%**	(31/52)	∗▼**44%**	(29/52)	∗▼**48%**	(27/52)	69%	(16/52)	∗▼**38%**	(32/52)	∗▼**44%**	(29/52)	∗▼**46%**	(28/52)
RiskIt	51%	(122/250)	∗▼**41%**	(148/250)	∗▼**43%**	(142/250)	∗▼**48%**	(130/250)	63%	(92/250)	∗▼**44%**	(140/250)	∗▼**48%**	(129/250)	∗▼**52%**	(120/250)
StackOverflow	93%	(12/171)	∗▼**46%**	(92/171)	∗▼**48%**	(89/171)	∗▼**50%**	(86/171)	84%	(28/171)	∗▼**53%**	(80/171)	∗▼**60%**	(68/171)	∗▼**60%**	(68/171)
StudentResidence	62%	(13/34)	∗▼**38%**	(21/34)	∗▼**41%**	(20/34)	∗▼**44%**	(19/34)	74%	(9/34)	∗▼**41%**	(20/34)	∗▼**47%**	(18/34)	∗▼**47%**	(18/34)
UnixUsage	52%	(70/147)	∗▼**41%**	(86/147)	∗▼**43%**	(84/147)	∗▼**47%**	(78/147)	73%	(40/147)	∗▼**44%**	(83/147)	∗▼**48%**	(76/147)	∗▼**50%**	(73/147)
Usda	88%	(30/247)	∗▼**39%**	(152/247)	∗▼**40%**	(147/247)	∗▼**44%**	(139/247)	78%	(54/247)	∗▼**44%**	(139/247)	∗▼**49%**	(125/247)	∗▼**51%**	(121/247)
WordNet	58%	(50/118)	∗▼**36%**	(76/118)	∗▼**42%**	(69/118)	∗▼**44%**	(66/118)	64%	(43/118)	∗▼**35%**	(77/118)	∗▼**40%**	(71/118)	∗▼**44%**	(66/118)
Minimum	37%		25%		30%		33%		58%		28%		30%		30%	
Average	66%		39%		43%		46%		74%		42%		48%		50%	
Maximum	93%		53%		60%		59%		89%		57%		64%		64%	

Greedy was the third best performer, achieving marginally less reduction in overall test suite size than HGS. As expected, the baseline Random technique was the worst performer.

In conclusion for RQ1, STICCER is the most effective at reducing the number of test cases and the overall number of `INSERT` statements in a test suite. Importantly, STICCER does this while preserving the coverage of the original test suite.

RQ2: Impact on Fault Finding Capability

Table IV shows the median mutation scores for all the reduction and test generation techniques where a difference was recorded for one of the reduction techniques with respect to the original test suite (OTS). Perhaps surprisingly, differences were only observed for one of the reduction techniques and eight of the 34 schemas — for the rest, the same mutation score was recorded. For each of these schemas, differences were only experienced with test suites generated by AVM-D.

For test suites generated by AVM-D, STICCER's reduced test suites had a mutation score significantly worse than the OTS for five schemas. Although statistically significant, the difference does not register to the first decimal point for two schemas (i.e., *BrowserCookies* and *iTrust*) and the difference is at most 3.2% for *NistWeather*. Random and Greedy were also significantly worse for five schemas with AVM-D's tests. Greedy performed almost identically to STICCER. Given that STICCER performs greedy reduction as one of its initial steps, this points to the loss of fault detection capability being down to test cases that were removed as duplicate due to their coverage of test requirements, rather than test merging. Finally, HGS was statistically significantly worse compared to the OTS for the highest number of schemas, namely eight in total.

In all cases, DOMINO-generated suites are more robust to the reduction, likely because of the diversity of test values that it generates. Conversely, the re-use of "default" values for AVM-D means that the loss of test cases and `INSERT`s through reduction results in a small loss of fault-finding capability.

In conclusion for RQ2, mutation scores of test suites were more or less preserved following reduction. While some test suites experienced a drop in mutation score, the difference was not substantial (3.2% maximum). Test suites generated by DOMINO did not experience any loss of mutation score following the application of any of the reduction techniques.

RQ3: Impact on Test Suite and Mutation Analysis Runtime

Table V shows the median time for performing reduction with STICCER and mutation analysis with the resulting test suites, compared to performing mutation analysis with just the OTS. In general, STICCER is capable of substantial time savings for large schemas with large numbers of integrity constraints and test requirements. The largest saving is for the largest schema, *iTrust*, with a saving of approximately 16 minutes. Savings of 20 seconds are possible with *BookTown* and *RiskIt*. For the smallest five schemas by the number of columns, STICCER performed statistically worse. Yet, in practice the difference is negligible and always under a second.

Table VI compares STICCER with the other reduction techniques. The results show that STICCER's test suites generated with DOMINO ran significantly faster than all reduced test suite by other techniques. STICCER was over 2.5 times faster

TABLE III
MEDIAN REDUCTION EFFECTIVENESS FOR SQL INSERT STATEMENTS IN REDUCED TEST SUITES

Test suite reduction effectiveness is calculated as in the higher–is-better formula of Equation 1. The "▼" symbol indicates that a technique's reduction score was significantly less than STICCER, while "▲" indicates that it was significantly higher. The "∗" symbol denotes a large effect size for a technique when compared with STICCER. The numbers in brackets denote the median number of `INSERT` statements in reduced suites as a fraction of those in the original, unreduced test suite generated by either AVM-D or DOMINO.

	AVM-D								DOMINO							
Schema	STICCER		Random		Greedy		HGS		STICCER		Random		Greedy		HGS	
ArtistSimilarity	48%	(23/44)	∗▼**34%**	(29/44)	∗▼**34%**	(29/44)	∗▼**45%**	(24/44)	50%	(22/44)	∗▼**32%**	(30/44)	∗▼**39%**	(27/44)	∗▼**35%**	(28/44)
ArtistTerm	56%	(54/124)	∗▼**34%**	(82/124)	∗▼**36%**	(79/124)	∗▼**45%**	(68/124)	56%	(54/124)	∗▼**33%**	(84/124)	∗▼**36%**	(79/124)	∗▼**36%**	(79/124)
BankAccount	56%	(35/80)	∗▼**36%**	(52/80)	∗▼**38%**	(50/80)	∗▼**44%**	(45/80)	57%	(34/80)	∗▼**39%**	(49/80)	∗▼**44%**	(44/80)	∗▼**50%**	(40/80)
BookTown	44%	(225/403)	∗▼**35%**	(262/403)	∗▼**38%**	(250/403)	∗▲**47%**	(215/403)	49%	(206/403)	∗▼**37%**	(254/403)	∗▼**42%**	(233/403)	∗▼**48%**	(208/403)
BrowserCookies	64%	(63/175)	∗▼**56%**	(77/175)	∗▼**62%**	(66/175)	∗▼**64%**	(63/175)	69%	(55/175)	∗▼**50%**	(87/175)	∗▼**59%**	(71/175)	∗▼**60%**	(70/175)
Cloc	62%	(23/60)	∗▼**42%**	(35/60)	∗▼**48%**	(31/60)	∗▼**50%**	(30/60)	61%	(24/60)	∗▼**45%**	(33/60)	∗▼**48%**	(31/60)	∗▼**55%**	(27/60)
CoffeeOrders	61%	(107/273)	∗▼**41%**	(162/273)	∗▼**45%**	(149/273)	∗▼**48%**	(143/273)	65%	(96/273)	∗▼**43%**	(156/273)	∗▼**49%**	(139/273)	∗▼**51%**	(132/273)
CustomerOrder	39%	(290/475)	∗▼**33%**	(317/475)	∗▼**36%**	(305/475)	∗▲**39%**	(288/475)	56%	(208/475)	∗▼**35%**	(310/475)	∗▼**36%**	(302/475)	∗▼**41%**	(279/475)
DellStore	56%	(123/281)	∗▼**37%**	(177/281)	∗▼**39%**	(172/281)	∗▼**42%**	(162/281)	50%	(141/281)	∗▼**40%**	(169/281)	∗▼**44%**	(156/281)	∗▼**47%**	(150/281)
Employee	60%	(21/53)	∗▼**42%**	(30/53)	∗▼**47%**	(28/53)	∗▼**47%**	(28/53)	62%	(20/53)	∗▼**49%**	(27/53)	∗▼**57%**	(23/53)	∗▼**57%**	(23/53)
Examination	66%	(78/229)	∗▼**49%**	(118/229)	∗▼**50%**	(114/229)	∗▼**52%**	(111/229)	74%	(60/229)	∗▼**55%**	(102/229)	∗▼**63%**	(86/229)	∗▼**64%**	(83/229)
Flights	70%	(41/137)	∗▼**57%**	(59/137)	∗▼**66%**	(46/137)	∗▼**66%**	(46/137)	58%	(58/137)	∗▼**47%**	(73/137)	∗▼**55%**	(62/137)	∗▼**55%**	(62/137)
FrenchTowns	50%	(81/161)	∗▼**41%**	(95/161)	∗▼**47%**	(86/161)	∗▼**43%**	(91/161)	52%	(77/161)	∗▼**40%**	(97/161)	∗▼**40%**	(96/161)	∗▼**43%**	(92/161)
Inventory	54%	(13/28)	∗▼**32%**	(19/28)	∗▼**39%**	(17/28)	∗▼**43%**	(16/28)	57%	(12/28)	∗▼**43%**	(16/28)	∗▼**54%**	(13/28)	54%	(13/28)
IsoFlav_R2	62%	(104/274)	∗▼**46%**	(148/274)	∗▼**50%**	(136/274)	∗▼**50%**	(136/274)	64%	(100/274)	∗▼**49%**	(138/274)	∗▼**58%**	(116/274)	∗▼**59%**	(111/274)
Iso3166	47%	(10/19)	∗▼**26%**	(14/19)	∗▼**37%**	(12/19)	∗▼**37%**	(12/19)	42%	(11/19)	∗▼**32%**	(13/19)	∗▼**37%**	(12/19)	47%	(10/19)
iTrust	56%	(978/2204)	∗▼**43%**	(1261/2204)	∗▼**48%**	(1142/2204)	∗▼**50%**	(1103/2204)	57%	(940/2204)	∗▼**45%**	(1212/2204)	∗▼**50%**	(1101/2204)	∗▼**52%**	(1064/2204)
JWhoisServer	38%	(158/256)	∗▼**36%**	(163/256)	38%	(158/256)	∗▲**40%**	(153/256)	47%	(136/256)	∗▼**38%**	(159/256)	∗▼**41%**	(152/256)	∗▼**43%**	(145/256)
MozillaExtensions	71%	(105/356)	∗▼**53%**	(168/356)	∗▼**63%**	(130/356)	∗▼**50%**	(179/356)	69%	(110/356)	∗▼**52%**	(170/356)	∗▼**60%**	(144/356)	∗▼**61%**	(140/356)
MozillaPermissions	62%	(19/50)	∗▼**42%**	(29/50)	∗▼**48%**	(26/50)	∗▼**48%**	(26/50)	66%	(17/50)	∗▼**51%**	(24/50)	∗▼**54%**	(23/50)	∗▼**60%**	(20/50)
NistDML181	68%	(25/78)	∗▼**50%**	(39/78)	∗▼**55%**	(35/78)	∗▼**53%**	(37/78)	65%	(28/78)	∗▼**45%**	(43/78)	∗▼**54%**	(36/78)	∗▼**56%**	(34/78)
NistDML182	73%	(102/384)	∗▼**56%**	(168/384)	∗▼**61%**	(151/384)	∗▼**61%**	(150/384)	74%	(98/384)	∗▼**53%**	(180/384)	∗▼**61%**	(149/384)	∗▼**63%**	(144/384)
NistDML183	65%	(24/68)	∗▼**46%**	(37/68)	∗▼**46%**	(37/68)	∗▼**53%**	(32/68)	62%	(26/68)	∗▼**41%**	(40/68)	∗▼**50%**	(34/68)	∗▼**51%**	(33/68)
NistWeather	61%	(47/120)	∗▼**45%**	(66/120)	∗▼**52%**	(58/120)	∗▼**49%**	(61/120)	63%	(44/120)	∗▼**45%**	(66/120)	∗▼**51%**	(58/120)	∗▼**52%**	(58/120)
NistXTS748	48%	(12/23)	∗▼**28%**	(16/23)	∗▼**35%**	(15/23)	∗▼**39%**	(14/23)	52%	(11/23)	∗▼**37%**	(14/23)	∗▼**48%**	(12/23)	∗▼**48%**	(12/23)
NistXTS749	53%	(34/73)	∗▼**45%**	(40/73)	∗▼**47%**	(39/73)	53%	(34/73)	58%	(30/73)	∗▼**41%**	(43/73)	∗▼**49%**	(37/73)	∗▼**49%**	(37/73)
Person	53%	(14/30)	∗▼**37%**	(19/30)	∗▼**50%**	(15/30)	∗▼**50%**	(15/30)	47%	(16/30)	∗▼**37%**	(19/30)	∗▼**37%**	(19/30)	∗▼**43%**	(17/30)
Products	65%	(50/144)	∗▼**47%**	(76/144)	∗▼**53%**	(68/144)	∗▼**56%**	(63/144)	62%	(54/144)	∗▼**46%**	(78/144)	∗▼**51%**	(70/144)	∗▼**53%**	(68/144)
RiskIt	50%	(342/687)	∗▼**44%**	(384/687)	∗▼**47%**	(366/687)	∗▲**51%**	(336/687)	54%	(318/687)	∗▼**46%**	(373/687)	∗▼**50%**	(346/687)	∗▼**53%**	(322/687)
StackOverflow	64%	(93/257)	∗▼**46%**	(139/257)	∗▼**48%**	(134/257)	∗▼**50%**	(129/257)	66%	(88/257)	∗▼**51%**	(126/257)	∗▼**57%**	(111/257)	∗▼**57%**	(110/257)
StudentResidence	56%	(32/72)	∗▼**39%**	(44/72)	∗▼**42%**	(42/72)	∗▼**46%**	(39/72)	60%	(29/72)	∗▼**42%**	(42/72)	∗▼**49%**	(37/72)	∗▼**49%**	(37/72)
UnixUsage	58%	(252/595)	∗▼**45%**	(328/595)	∗▼**47%**	(313/595)	∗▼**51%**	(293/595)	70%	(178/595)	∗▼**47%**	(318/595)	∗▼**52%**	(287/595)	∗▼**54%**	(274/595)
Usda	59%	(157/381)	∗▼**40%**	(227/381)	∗▼**42%**	(220/381)	∗▼**46%**	(206/381)	59%	(157/381)	∗▼**44%**	(212/381)	∗▼**50%**	(190/381)	∗▼**52%**	(181/381)
WordNet	50%	(96/192)	∗▼**40%**	(116/192)	∗▼**47%**	(102/192)	∗▼**49%**	(98/192)	49%	(97/192)	∗▼**39%**	(117/192)	∗▼**45%**	(106/192)	∗▼**49%**	(98/192)
Minimum	38%		26%		34%		37%		42%		32%		36%		35%	
Average	57%		42%		46%		49%		59%		43%		49%		51%	
Maximum	73%		57%		66%		66%		74%		55%		63%		64%	

TABLE IV
MEDIAN MUTATION SCORES EXPRESSED AS PERCENTAGES

Scores are expressed as a higher-is-better percentage of mutants killed by the test suites concerned. In this table, "OTS" refers to the original, unreduced test suites, while the other scores were obtained by test suites reduced by each of the respective techniques.

The "▼" symbol means that the statistical significance test reveals that a technique's test suite obtained a significantly lower mutation score compared to the OTS, while "▲" indicates the technique obtained a significantly higher mutation score. The "∗" symbol denotes a large effect size when comparing the tests from the technique with the OTS.

	AVM-D					DOMINO				
Schemas	OTS	STICCER	Random	Greedy	HGS	OTS	STICCER	Random	Greedy	HGS
BrowserCookies	86.5	**▼86.5**	86.5	**▼86.5**	**▼86.5**	96.6	96.6	96.6	96.6	96.6
FrenchTowns	83.3	∗**▼80.3**	∗**▼80.3**	∗**▼80.3**	∗**▼81.8**	95.5	95.5	95.5	95.5	95.5
iTrust	83.6	∗**▼83.6**	∗**▼83.6**	∗**▼83.6**	∗**▼83.6**	99.2	99.2	99.2	99.2	99.1
NistWeather	93.8	∗**▼90.6**	93.8	∗**▼90.6**	93.8	100.0	100.0	100.0	100.0	100.0
NistXTS749	92.0	92.0	**▼92.0**	92.0	∗**▼88.0**	94.0	94.0	94.0	94.0	94.0
RiskIt	89.3	89.3	**▼89.3**	89.3	∗**▼88.8**	99.5	99.5	99.5	99.5	99.5
UnixUsage	98.2	98.2	98.2	98.2	∗**▼97.3**	100.0	100.0	100.0	100.0	100.0
WordNet	87.4	∗**▼86.3**	**▼87.4**	∗**▼86.3**	∗**▼86.3**	99.0	99.0	99.0	99.0	99.0

than the other techniques for the *iTrust* schema. STICCER was generally, but not always, faster than the other reduction techniques for tests generated by AVM-D. Since the reduction techniques were less successful at reducing AVM-D-generated test suites, there is less to differentiate their performance.

In conclusion for RQ3, the results show that, in general, test suites reduced by STICCER are fast to run compared to the OTS and those reduced by other techniques. For the largest schema, STICCER-reduced test suites were up to five times faster to run mutation analysis with, as compared to the OTS.

VI. RELATED WORK

Many researchers have studied test suite reduction with the motivation of reducing the regression testing or human oracle costs (e.g., [13], [29]–[31], [38], [39]). Yoo and Harman [31] surveyed prior work on ways to reduce suites by selecting a representative subset of test cases. These included Greedy [40], HGS [13], Greedy Essential (GE), and Greedy Redundant Essential (GRE) [41]. The GE algorithm first selects the essential, or "irreplaceable" tests, next applying the standard greedy algorithm. Conversely, GRE removes any redundant tests that the essential tests already covered and then applies the greedy algorithm. Since experimental studies showed that HGS and Greedy significantly reduced the size of test suites [42], others have used these methods as building blocks for new reducers (e.g., [43]–[45]). For instance, some applied integer linear programming (e.g., [46]) or evolutionary algorithms (e.g., [47]) to test suite reduction. Finally, Yoo and Harman used multi-objective search with test reduction, test coverage, and past fault-detection history as goals [48].

While some work found that reduced test suites negatively influence fault detection [15], [49], others found that fault detection effectiveness was more or less maintained [50], [51]. It is worth noting that all of these reduction techniques were focused on reducing test suites for traditional programs rather than database schema testing, as presented in this paper.

In the context of database testing, there are several studies of database schema evolution (e.g., [8]–[10]), thereby motivating the need for efficient regression testing methods. Kapfhammer reduced test suites for database application tests using a greedy algorithm [52]. Similarly, Tuya et al. used a greedy algorithm to reduce the amount of data within databases for testing SQL `SELECT` queries [53], [54]. Haftmann et al. used a slicing technique to prioritize tests and reduce the number of database resets to improve the efficiency of regression testing [55].

TABLE V
MEDIAN MUTATION ANALYSIS TIMES FOR STICCER-GENERATED TESTS VERSUS THE ORIGINAL TEST SUITE

Times for STICCER are broken down into "RT" (i.e., time to reduce test suites), and "MT" (i.e., time for mutation analysis with STICCER), with the total compared to the original test suite (OTS) for a fair comparison. The "▲" symbol means that STICCER required a statistically significantly longer time to run than the OTS, while "▼" denotes the opposite. The "∗" symbol indicates a large effect size when comparing the two.

	AVM-D				DOMINO			
	OTS	STICCER			OTS	STICCER		
Schemas	Total	RT	MT	Total	Total	RT	MT	Total
ArtistSimilarity	0.09	0.12	0.05	∗**▲0.17**	0.10	0.13	0.05	∗**▲0.18**
ArtistTerm	0.54	0.14	0.22	∗**▼0.37**	0.54	0.14	0.22	∗**▼0.36**
BankAccount	0.45	0.17	0.19	∗**▼0.36**	0.46	0.16	0.19	∗**▼0.34**
BookTown	36.19	0.94	16.17	∗**▼17.11**	36.50	1.07	10.85	∗**▼11.92**
BrowserCookies	2.21	0.37	0.94	∗**▼1.31**	2.34	0.37	0.73	∗**▼1.09**
Cloc	0.27	0.15	0.07	∗**▼0.22**	0.28	0.15	0.09	∗**▼0.25**
CoffeeOrders	2.96	0.31	1.20	∗**▼1.51**	2.98	0.29	1.05	∗**▼1.34**
CustomerOrder	9.74	0.80	6.68	∗**▼7.47**	9.78	0.84	4.56	∗**▼5.40**
DellStore	8.04	1.42	1.93	∗**▼3.35**	8.40	1.46	3.22	∗**▼4.68**
Employee	0.34	0.20	0.13	**▼0.32**	0.36	0.18	0.12	∗**▼0.30**
Examination	4.42	1.06	1.39	∗**▼2.45**	4.48	1.11	1.00	∗**▼2.12**
Flights	1.60	0.30	0.61	∗**▼0.91**	1.68	0.41	0.68	∗**▼1.09**
FrenchTowns	2.43	0.23	1.46	∗**▼1.69**	2.44	0.23	1.23	∗**▼1.46**
Inventory	0.10	0.12	0.05	∗**▲0.17**	0.10	0.12	0.04	∗**▲0.17**
IsoFlav_R2	9.80	0.70	2.83	∗**▼3.52**	10.20	0.75	2.72	∗**▼3.47**
Iso3166	0.04	0.11	0.02	∗**▲0.13**	0.05	0.11	0.03	∗**▲0.13**
iTrust	2297.86	150.17	959.31	∗**▼1109.48**	2330.02	157.23	428.57	∗**▼585.80**
JWhoisServer	8.91	1.64	5.62	∗**▼7.25**	9.34	1.86	3.88	∗**▼5.74**
MozillaExtensions	25.52	2.02	6.62	∗**▼8.64**	26.61	2.38	5.39	∗**▼7.78**
MozillaPermissions	0.23	0.15	0.08	0.23	0.24	0.15	0.07	∗**▼0.22**
NistDML181	0.36	0.15	0.11	∗**▼0.26**	0.38	0.15	0.12	∗**▼0.28**
NistDML182	14.60	1.93	2.22	∗**▼4.15**	14.99	2.00	2.43	∗**▼4.43**
NistDML183	0.28	0.14	0.09	∗**▼0.23**	0.29	0.14	0.10	∗**▼0.25**
NistWeather	0.74	0.22	0.29	∗**▼0.51**	0.76	0.23	0.25	∗**▼0.48**
NistXTS748	0.08	0.12	0.04	∗**▲0.16**	0.08	0.11	0.04	∗**▲0.15**
NistXTS749	0.39	0.17	0.20	**▼0.37**	0.40	0.17	0.16	∗**▼0.33**
Person	0.16	0.10	0.09	∗**▲0.19**	0.16	0.15	0.07	∗**▲0.22**
Products	1.04	0.14	0.42	∗**▼0.57**	1.06	0.18	0.44	∗**▼0.62**
RiskIt	44.40	1.41	23.15	∗**▼24.57**	45.91	1.59	18.90	∗**▼20.49**
StackOverflow	5.58	0.93	0.97	∗**▼1.90**	5.76	1.16	1.42	∗**▼2.58**
StudentResidence	0.43	0.16	0.20	∗**▼0.36**	0.44	0.14	0.17	∗**▼0.32**
UnixUsage	12.25	0.92	6.21	∗**▼7.13**	12.34	0.95	3.83	∗**▼4.79**
Usda	15.55	1.39	2.78	∗**▼4.17**	16.44	1.52	4.15	∗**▼5.68**
WordNet	3.89	0.37	1.86	∗**▼2.23**	3.93	0.38	1.69	∗**▼2.07**

However, unlike these examples of prior work, this paper focuses on database schema testing and the reduction of test suites using greedy techniques and the merging of test cases.

VII. CONCLUSIONS AND FUTURE WORK

Since many real-world database applications contain complex and rapidly evolving database schemas [8]–[10], there is the need for an efficient way to regression test these systems. While the automatically generated test suites created by tools like *SchemaAnalyst* [24] mitigate the challenges associated with manually testing a database schema's integrity constraints, the numerous, often interwoven, generated tests make repeated testing and test adequacy assessment time consuming. This paper presents a test suite reduction technique, called Schema Test Integrity Constraints Combination for Efficient Reduction (STICCER), that systematically discards and merges redundant tests, creating a reduced test suite that is guaranteed to have the same coverage as the original one. STICCER advances the state-of-the-art in test suite reduction because, unlike traditional approaches such as Greedy and HGS, it identifies and reduces both the overlap in test requirement coverage and the database state created by the tests.

Using 34 relational database schemas and test data created by two test generation methods, this paper experimentally compared STICCER to Greedy, HGS, and a Random method. These results show that STICCER significantly outperforms the other techniques at decreasing test suite size, while also lessening the overall number of database interactions (i.e., the SQL `INSERT` statements) performed by the tests. The

TABLE VI
MEDIAN MUTATION ANALYSIS TIMES FOR STICCER-GENERATED TESTS VERSUS THE OTHER REDUCTION TECHNIQUES

The "▼" symbol means that the statistical significance test reveals that the technique's reduced tests required a significantly longer time than those generated STICCER, while "▲" indicates a technique's tests needed a significantly shorter time than STICCER's. The "∗" symbol means the effect size was large when comparing the technique to STICCER.

	AVM-D				DOMINO			
Schemas	STICCER	Random	Greedy	HGS	STICCER	Random	Greedy	HGS
ArtistSimilarity	0.05	∗**▼0.07**	∗**▼0.07**	∗**▼0.06**	0.05	∗**▼0.07**	∗**▼0.07**	∗**▼0.07**
ArtistTerm	0.22	∗**▼0.38**	∗**▼0.38**	∗**▼0.33**	0.22	∗**▼0.39**	∗**▼0.37**	∗**▼0.37**
BankAccount	0.19	∗**▼0.32**	∗**▼0.31**	∗**▼0.28**	0.19	∗**▼0.29**	∗**▼0.28**	∗**▼0.25**
BookTown	16.17	∗**▼22.89**	∗**▼22.05**	∗**▼19.10**	10.85	∗**▼22.37**	∗**▼20.75**	∗**▼19.11**
BrowserCookies	0.94	∗**▼1.12**	∗**▼1.02**	∗**▼0.98**	0.73	∗**▼1.27**	∗**▼1.12**	∗**▼1.04**
Cloc	0.07	∗**▼0.17**	∗**▼0.16**	∗**▼0.15**	0.09	∗**▼0.18**	∗**▼0.17**	∗**▼0.15**
CoffeeOrders	1.20	∗**▼1.86**	∗**▼1.88**	∗**▼1.68**	1.05	∗**▼1.81**	∗**▼1.81**	∗**▼1.57**
CustomerOrder	6.68	∗**▲6.47**	∗**▼7.13**	∗**▲5.88**	4.56	∗**▼6.32**	∗**▼7.02**	∗**▼5.68**
DellStore	1.93	∗**▼5.27**	∗**▼5.15**	∗**▼4.68**	3.22	∗**▼5.02**	∗**▼4.71**	∗**▼4.53**
Employee	0.13	∗**▼0.21**	∗**▼0.20**	∗**▼0.21**	0.12	∗**▼0.19**	∗**▼0.16**	∗**▼0.16**
Examination	1.39	∗**▼2.34**	∗**▼2.32**	∗**▼2.16**	1.00	∗**▼2.08**	∗**▼1.80**	∗**▼1.69**
Flights	0.61	∗**▼0.85**	∗**▼0.72**	∗**▼0.71**	0.68	∗**▼0.95**	∗**▼0.87**	∗**▼0.84**
FrenchTowns	1.46	∗**▼1.59**	∗**▼1.52**	∗**▼1.55**	1.23	∗**▼1.63**	∗**▼1.62**	∗**▼1.54**
Inventory	0.05	∗**▼0.07**	∗**▼0.07**	∗**▼0.06**	0.04	∗**▼0.07**	∗**▼0.06**	∗**▼0.06**
IsoFlav_R2	2.83	∗**▼5.51**	∗**▼5.24**	∗**▼5.01**	2.72	∗**▼5.04**	∗**▼4.33**	∗**▼4.04**
Iso3166	0.02	∗**▼0.03**	∗**▼0.03**	∗**▼0.03**	0.03	∗**▼0.03**	∗**▼0.03**	∗**▼0.03**
iTrust	959.31	∗**▼1367.85**	∗**▼1251.95**	∗**▼1223.85**	428.57	∗**▼1273.12**	∗**▼1155.09**	∗**▼1117.88**
JWhoisServer	5.62	∗**▼5.88**	∗**▼5.73**	∗**▲5.48**	3.88	∗**▼5.81**	∗**▼5.56**	∗**▼5.29**
MozillaExtensions	6.62	∗**▼12.35**	∗**▼10.22**	∗**▼12.74**	5.39	∗**▼11.75**	∗**▼9.89**	∗**▼9.81**
MozillaPermissions	0.08	∗**▼0.16**	∗**▼0.15**	∗**▼0.15**	0.07	∗**▼0.14**	∗**▼0.13**	∗**▼0.12**
NistDML181	0.11	∗**▼0.20**	∗**▼0.19**	∗**▼0.18**	0.12	∗**▼0.22**	∗**▼0.18**	∗**▼0.18**
NistDML182	2.22	∗**▼7.14**	∗**▼6.50**	∗**▼6.31**	2.43	∗**▼7.54**	∗**▼6.24**	∗**▼5.85**
NistDML183	0.09	∗**▼0.17**	∗**▼0.17**	∗**▼0.17**	0.10	∗**▼0.18**	∗**▼0.16**	∗**▼0.17**
NistWeather	0.29	∗**▼0.45**	∗**▼0.41**	∗**▼0.41**	0.25	∗**▼0.46**	∗**▼0.42**	∗**▼0.41**
NistXTS748	0.04	∗**▼0.06**	∗**▼0.05**	∗**▼0.05**	0.04	∗**▼0.05**	∗**▼0.05**	∗**▼0.05**
NistXTS749	0.20	∗**▼0.24**	∗**▼0.24**	0.21	0.16	∗**▼0.25**	∗**▼0.22**	∗**▼0.22**
Person	0.09	∗**▼0.12**	**▼0.10**	∗**▼0.10**	0.07	∗**▼0.11**	∗**▼0.11**	∗**▼0.10**
Products	0.42	∗**▼0.64**	∗**▼0.61**	∗**▼0.55**	0.44	∗**▼0.68**	∗**▼0.63**	∗**▼0.59**
RiskIt	23.15	∗**▼25.53**	∗**▼25.79**	∗**▲22.77**	18.90	∗**▼26.43**	∗**▼24.60**	∗**▼21.52**
StackOverflow	0.97	∗**▼3.15**	∗**▼3.00**	∗**▼2.97**	1.42	∗**▼2.73**	∗**▼2.53**	∗**▼2.48**
StudentResidence	0.20	∗**▼0.28**	∗**▼0.27**	∗**▼0.26**	0.17	∗**▼0.28**	∗**▼0.25**	∗**▼0.25**
UnixUsage	6.21	∗**▼6.97**	∗**▼7.52**	∗**▼6.30**	3.83	∗**▼7.57**	∗**▼6.96**	∗**▼5.91**
Usda	2.78	∗**▼8.89**	∗**▼8.60**	∗**▼9.34**	4.15	∗**▼9.72**	∗**▼7.50**	∗**▼7.33**
WordNet	1.86	∗**▼2.45**	∗**▼2.21**	∗**▼2.07**	1.69	∗**▼2.44**	∗**▼2.28**	∗**▼2.09**

results further reveal that the reduced test suites produced by STICCER are up to 5X faster than the original test suite and 2.5X faster than reduced suites created by Greedy, HGS, and Random, often leading to significant decreases in mutation analysis time. STICCER's tests always preserve the coverage of the test suite and rarely lead to a drop in the mutation score, with a maximum decrease in fault detection of 3.2%.

Given these promising results, as part of future work we will enhance STICCER so that it operates in a multi-objective fashion, explicitly balancing testing goals like decreasing the test suite size while maximizing its mutation score. We will also directly integrate STICCER into the test generation loop, allowing the reduction method to work in tandem with the test data generators. After finishing these improvements to the presented technique, we will conduct more experiments with new relational database schemas. Since STICCER has proven effective at reducing database schema test suites, we will also investigate ways in which we can adapt the presented approach to the reduction of the test suites for traditional programs that manipulate complex state in other formats. Our goal is to develop an automated method that quickly generates focused and effective tests for a wide variety of data-driven programs, like those that use relational databases or NoSQL data stores.

REFERENCES

[1] S. Guz, "Basic mistakes in database testing," https://dzone.com/articles/basic-mistakes-database, 2011.

[2] S. Ambler and P. J. Sadalage, *Refactoring Databases: Evolutionary Database Design*, 2006.

[3] S. Ambler, "Database testing: How to regression test a relational database," http://www.agiledata.org/essays/databaseTesting.html.

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

[5] 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.

[6] 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 International Conference on Software Testing, Verification and Validation*, 2013.

[7] A. Alsharif, G. M. Kapfhammer, and P. McMinn, "DOMINO: Fast and effective test data generation for relational database schemas," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2018.

[8] C. A. Curino, L. Tanca, H. J. Moon, and C. Zaniolo, "Schema evolution in Wikipedia: Toward a web information system benchmark," in *International Conference on Enterprise Information Systems*, 2008.

[9] 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 International Symposium on the Foundations of Software Engineering*, 2013.

[10] S. Ambler, "Adopting evolutionary/agile database techniques," http://www.agiledata.org/essays/adopting.html.

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

[12] 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 International Workshop on Mutation Analysis*, 2013.

[13] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *Transactions on Software Engineering and Methodology*, vol. 2, no. 3, 1993.

[14] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *Proceeding of the 12th International Conference on Testing Computer Software*, 1995.

[15] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, 2002.

[16] "The SQLite DBMS," http://www.sqlite.org/.

[17] "The PostgreSQL DBMS," http://www.postgresql.org/.

[18] "SQL as understood by SQLite," https://sqlite.org/lang_createtable.html.

[19] "Why does a UNIQUE constraint allow only one NULL?" https://dba.stackexchange.com/questions/80514/why-does-a-unique-constraint-allow-only-one-null.

[20] "Database Migration: What It Is and How to Do It," https://rollout.io/blog/database-migration/.

[21] "Which database should I use in production?" https://djangodeployment.com/2016/12/23/which-database-should-i-use-on-production/.

[22] "Top 10 sites built with Django framework," http://ddi-dev.com/blog/programming/top-sites-built-django-framework/.

[23] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2016.

[24] 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 International Conference on Software Maintenance and Evolution*, 2016.

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

[26] J. Kempka, P. McMinn, and D. Sudholt, "A theoretical runtime and empirical analysis of different alternating variable searches for search-based testing," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2013.

[27] P. McMinn and G. M. Kapfhammer, "AVMf: An open-source framework and implementation of the alternating variable method," in *Proceedings of the International Symposium on Search-Based Software Engineering*, 2016.

[28] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in *Proceedings of the International Conference on Automated Software Engineering*, 1998.

[29] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," in *Proceedings of the 6th Workshop on Program Analysis for Software Tools and Engineering*, 2005.

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

[31] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, 2012.

[32] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2015.

[33] C. J. Wright, G. M. Kapfhammer, and P. McMinn, "The impact of equivalent, redundant and quasi mutants on database schema mutation analysis," in *Proceedings of the International Conference on Quality Software*, 2014.

[34] 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.

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

[36] P. McMinn, G. M. Kapfhammer, and C. J. Wright, "Virtual mutation analysis of relational database schemas," in *Proceedings of the International Workshop on Automation of Software Test*, 2016.

[37] P. McMinn, C. Wright, C. McCurdy, and G. M. Kapfhammer, "Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas," *Transactions on Software Engineering*, 2019.

[38] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, no. 3, 1996.

[39] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," in *Proceedings of the International Conference on Software Engineering*, 2018.

[40] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, vol. 4, no. 3, 1979.

[41] T. Y. Chen and M. F. Lau, "A new heuristic for test suite reduction," *Information and Software Technology*, vol. 40, no. 5-6, 1998.

[42] H. Zhong, L. Zhang, and H. Mei, "An experimental comparison of four test suite reduction techniques," in *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[43] D. Jeffrey and N. Gupta, "Improving fault detection capability by selectively retaining test cases during test suite reduction," *Transactions on Software Engineering*, vol. 33, no. 2, 2007.

[44] C.-T. Lin, K.-W. Tang, and G. M. Kapfhammer, "Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests," *Information and Software Technology*, vol. 56, no. 10, 2014.

[45] C.-T. Lin, K.-W. Tang, J.-S. Wang, and G. M. Kapfhammer, "Empirically evaluating greedy-based test suite reduction methods at different levels of test suite complexity," *Science of Computer Programming*, vol. 150, 2017.

[46] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *Proceeding of the International Conference on Software Engineering*, 2004.

[47] N. Mansour and K. El-Fakih, "Simulated annealing and genetic algorithms for optimal regression testing," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 1, 1999.

[48] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software*, vol. 83, no. 4, 2010.

[49] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in *Proceedings of the International Conference on Software Maintenance*, 1998.

[50] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, "Test set size minimization and fault detection effectiveness: A case study in a space application," *Journal of Systems and Software*, vol. 48, no. 2, 1999.

[51] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Software: Practice and Experience*, vol. 28, no. 4, 1998.

[52] G. Kapfhammer, "Towards a method for reducing the test suites of database applications," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2012.

[53] J. Tuya, C. de la Riva, M. J. Suarez-Cabal, and R. Blanco, "Coverage-aware test database reduction," *Transactions on Software Engineering*, vol. 42, no. 10, 2016.

[54] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Query-aware shrinking test databases," in *Proceedings of the International Workshop on Testing Database Systems*, 2009.

[55] F. Haftmann, D. Kossmann, and E. Lo, "A framework for efficient regression tests on database applications," *The VLDB Journal*, vol. 16, no. 1, 2007.