DOMINO: Fast and Effective Test Data Generation for Relational Database Schemas

Abdullah Alsharif
University of Sheffield

Gregory M. Kapfhammer
Allegheny College

Phil McMinn
University of Sheffield

***Abstract*—An organization's databases are often one of its most valuable assets. Data engineers commonly use a relational database because its schema ensures the validity and consistency of the stored data through the specification and enforcement of integrity constraints. To ensure their correct specification, industry advice recommends the testing of the integrity constraints in a relational schema. Since manual schema testing is labor-intensive and error-prone, this paper presents DOMINO, a new automated technique that generates test data according to a coverage criterion for integrity constraint testing. In contrast to more generalized search-based approaches, which represent the current state of the art for this task, DOMINO uses tailored, domain-specific operators to efficiently generate test data for relational database schemas. In an empirical study incorporating 34 relational database schemas hosted by three different database management systems, the results show that DOMINO can not only generate test suites faster than the state-of-the-art search-based method but that its test suites can also detect more schema faults.**

I. INTRODUCTION

A relational database is often one of an organization's most valuable assets [1]. The integrity constraints specified as part of a schema prevent the insertion of invalid data into a relational database. For instance, "`PRIMARY KEY`" and "`UNIQUE`" constraints ensure that data values are distinct, while arbitrary "`CHECK`" constraints can impose restrictions on data values by, as an example, requiring them to be in a specific range.

Even though there are many tutorials explaining how to avoid making mistakes when designing a relational database schema (e.g., [2]–[5]), data engineers may incorrectly specify or omit integrity constraints. Since database management systems (DBMSs) often interpret the SQL standard differently, a schema may exhibit different behavior during development and after deployment. Therefore, as advocated by industrial practitioners [6], it is necessary to test a schema's integrity constraints to ensure that they behave as an engineer expects.

Since haphazard methods may overlook schema faults, prior work presented a family of coverage criteria that support systematic testing [7]. These criteria require the creation of specific rows of database data that, when featured in SQL `INSERT` statements, exercise integrity constraints as *true* or *false*, or, test some particular property of the constraint. Frequently, to satisfy these coverage requirements, certain values may need to be identical to one another, different, or `NULL`, across different rows of data. For example, to violate a primary key (i.e., to exercise it as *false*), two rows of data need to be created with the same values for the key's columns. To satisfy a `UNIQUE` constraint (i.e., exercise it as *true*), values across rows need to be different. To violate a `NOT NULL` constraint, a particular column must be `NULL`. Since it is challenging for a tester to manually cover all of these requirements, prior work presented a tool that automatically generates the tests [8].

Based on the Alternating Variable Method (AVM) [9], this state-of-the-art method for generating schema tests is a search-based one that receives guidance from a fitness function [7], [10]. Yet, the generation of schema tests with search can be slow, particularly when it must locate columns that need to have identical values and then adjust those values until they are the same. To aid the process, the AVM may be configured to start with a series of "default" values, thus ensuring that matches are likely from the outset. Yet, this can introduce a lot of similarity across the different tests in the suite, hindering both its diversity and potential fault-finding capability.

This paper shows how to improve the automated search for schema test data by using a tailored approach, called DOMINO (DOMain-specific approach to INtegrity cOnstraint test data generation), that features operators specific to the problem domain. Leveraging knowledge of the schema and a coverage requirement, it explicitly sets data values to be the same, different, or `NULL`, only falling back on random or search-based methods when it must satisfy more arbitrary constraints.

Using 34 relational database schemas hosted by three different DBMSs (i.e., HyperSQL, PostgreSQL, and SQLite) this paper experimentally compares DOMINO to both AVM and a hybrid DOMINO-AVM method. The results show that DOMINO generates data faster than both the state-of-the-art AVM and the hybrid method, while also producing tests that normally detect more faults than those created by the AVM.

The contributions of this paper are therefore as follows:

1) An informal analysis of why search-based and random methods inefficiently generate tests for the integrity constraints in relational schemas (Sections II-B and II-C).
2) The DOMINO method that incorporates domain-specific operators for finding suitable test data (Section III).
3) Experiments showing that DOMINO is both efficient (i.e., it is faster than the AVM at obtaining equivalent levels of coverage) and effective (i.e., it kills more mutants than does the AVM), while also revealing that DOMINO-AVM is not superior to DOMINO (Section IV).

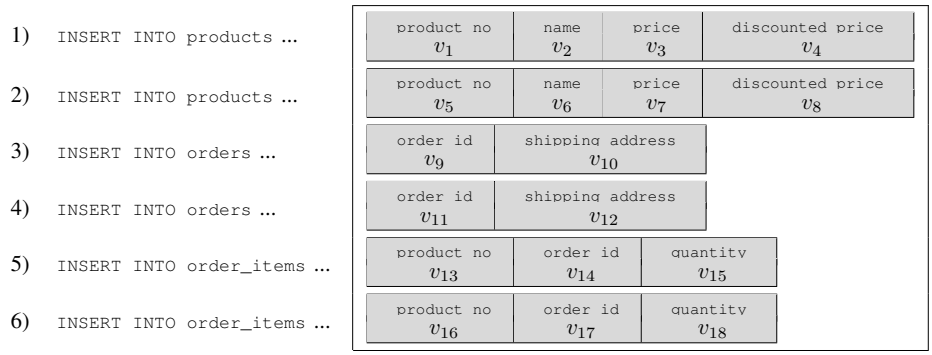
To support the replication of this paper's experimental results and to facilitate the testing of relational database schemas with DOMINO, we have integrated it into the *SchemaAnalyst* tool [8], making the complete system available at [11].

```
CREATE TABLE products (
  product_no INTEGER PRIMARY KEY NOT NULL,
  name VARCHAR(100) NOT NULL,
  price NUMERIC NOT NULL,
  discounted_price NUMERIC NOT NULL,
  CHECK (price > 0),
  CHECK (discounted_price > 0),
  CHECK (price > discounted_price));

CREATE TABLE orders (
  order_id INTEGER PRIMARY KEY,
  shipping_address VARCHAR(100));

CREATE TABLE order_items (
  product_no INTEGER REFERENCES products,
  order_id INTEGER REFERENCES orders,
  quantity INTEGER NOT NULL,
  PRIMARY KEY (product_no, order_id),
  CHECK (quantity > 0));
```

(a) A relational database schema containing three tables.

1)	`INSERT INTO products(product_no, name, price, discounted_price) VALUES(10, 'abc', 2, 1);`	✓
2)	`INSERT INTO products(product_no, name, price, discounted_price) VALUES(20, 'def', 10, 9);`	✓
3)	`INSERT INTO orders(order_id, shipping_address) VALUES(100, 'uvw');`	✓
4)	`INSERT INTO orders(order_id, shipping_address) VALUES(200, 'xyz');`	✓
5)	`INSERT INTO order_items(product_no, order_id, quantity) VALUES(10, 100, 1);`	✓
6)	`INSERT INTO order_items(product_no, order_id, quantity) VALUES(10, 100, 2);`	✗

(b) An example test case that consists of `INSERT` statements for a database specified by the relational schema in part (a). Normally inspected by a tester who is checking schema correctness, the ✓ and ✗ marks denote whether or not the data contained within each `INSERT` satisfied the schema's integrity constraints and was accepted into the database.



(c) The vector $\vec{v} = (v_1, v_2, \ldots, v_n)$ representation used by random and fitness-guided search techniques for finding the test data for each `INSERT` forming the test in part (b).

Fig. 1. The *Products* relational database schema and an example test case.

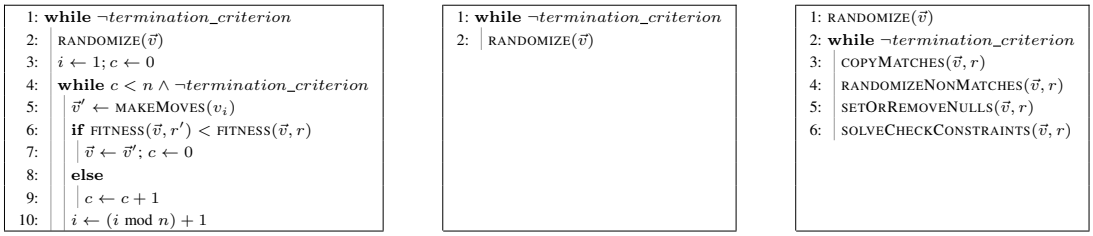
II. Background

This section introduces the problem of testing integrity constraints for relational database schemas and the different coverage criteria that have been developed for this purpose. It explains how both random and search-based methods can automatically create test data to obtain coverage, concluding with an informal analysis of why these techniques are not always efficient at generating test data for integrity constraints.

A. Schemas, Integrity Constraints, and Testing

Figure 1a shows SQL statements that create an example relational database schema specifying the structure of a database for managing a series of products and the related orders. The schema has three tables, each involving a series of columns (e.g., `product_no` and `name` for the `products` table). Every column has a specific data type, for instance `INTEGER` or `VARCHAR(100)`; the latter a variable length string of up to 100 characters in length. Every table involves the definition of *integrity constraints*, highlighted in the subfigure with a gray background. All three tables have a primary key specifying a certain set of columns that must have a distinct set of values for each row. This means, for example, that there cannot be rows with an identical combination of `product_no` and `order_no` values in the `order_items` table. Several columns are defined with a "`NOT NULL`" constraint, meaning that each row has to have an actual value for that column. Furthermore, the schemas define several "`CHECK`" constraints, or predicates that must hold over the data in each row of a table. For instance, `discounted_price` should always be less than `price` in the `products` table. Finally, the schema has two foreign keys, defined on the `order_items` table with the "`REFERENCES`" keyword. Foreign keys link rows across tables, requiring that specific non-`NULL` column values in each of the table's rows match particular column values in at least one row in the referenced table. In Figure 1a, for instance, every non-`NULL` `product_no` value in each row of the `order_items` table must match an existing `product_no` in a row of `products`, while every non-`NULL` `order_id` in each row of `order_items` must match an existing `order_id` in a row of `orders`.

The Need to Test Integrity Constraint Definitions: Integrity constraints protect the coherency, consistency, and validity of data in a database [12], [13], encoding rules such as "orders



(a) The AVM **(b)** Naive Random Search **(c)** The DOMINO Method

Fig. 2. Algorithms that automatically generate, according to some coverage criterion r, a vector $\vec{v}$ of variables appearing in the INSERT statements of a test case for database schema integrity constraints. The AVM and naive random search are general-purpose algorithms that can be adapted to a wide variety of problems, while DOMINO is an algorithm customized for automatically generating data to test the integrity constraints in a relational database schema.

must have a product" and "a product must have a non-zero or non-negative price". Any attempt to insert data into a table that violates one or more of its schema's integrity constraints is rejected and the data is not admitted into the database. Integrity constraints encode logic that is subject to faults, and therefore they need to be tested [6]. For instance, a developer could mis-specify a CHECK constraint, incorrectly formulate the columns of a PRIMARY KEY, or forget to specify certain integrity constraints altogether. These oversights could result in errors if, for example, missing constraints were intended to enforce aspects of data integrity such as unique user logins, non-NULL passwords, or prices greater than zero. To compound the problem, different DBMSs interpret the SQL standard differently (e.g., SQLite allows NULL entries in primary keys while most other DBMSs do not). This means that the behavior of a schema should be tested if the DBMS hosting a database is changed or differs between development and deployment.

Coverage Criteria for Integrity Constraint Testing: One basis for generating a test suite is to "cover" each of the integrity constraints and exercise them with test data that leads to them being evaluated as *true* (i.e., the constraint is satisfied and the data is inserted into the database) and *false* (i.e., the constraint is violated and the data is rejected). Figure 1b shows an example test case for the schema in part (a) of the figure, consisting of a series of INSERT statements. Assuming an initially empty database, the test case violates the primary key of the order_items table (i.e., it exercises it as *false*). The final INSERT statement (no. 6) duplicates the values for product_no and order_id that appeared in a previous INSERT statement (no. 5) and is rejected by the DBMS (i.e., the data is not inserted). To ensure that it is actually the primary key that is violated, and not the foreign keys defined for this table, the values used in the final INSERT must match values in the products and orders table. Hence, the test involves a series of "lead-up" INSERTs that change the initially empty database state to one that enables proper testing of the primary key.

Various coverage criteria, consisting of a number of test requirements, have been devised for testing relational schema integrity constraints [7]. For example, *Active Integrity Constraint Coverage* (AICC) demands that each integrity constraint be evaluated as *true* and *false* by the test suite while all other integrity constraints are simultaneously satisfied (similar to MC/DC coverage for testing individual conditions of decision statements in programs). *Clause-Based Active Integrity Constraint Coverage* (ClauseAICC) breaks down testing of each integrity constraint further for constraints involving multiple columns and CHECK constraints composed of subconditions. This criterion necessitates the involvement of each column in the constraint—or condition in the CHECK constraint—separately from the remaining components of the constraint.

In general, coverage criteria are good at detecting faults of *commission* but are weak at detecting faults of *omission* [14]. However, for integrity constraint testing, *Active Unique Column Coverage* (AUCC) and *Active Null Column Coverage* (ANCC) may be used to detect certain types of missing uniqueness and NOT NULL constraints. AUCC demands each column of the schema be tested with unique and non-unique values at least once, whereas ANCC requires that each column of the schema be tested with NULL/non-NULL values at least once. ClauseAICC, the strongest form of coverage criteria based on the structure of the integrity constraints themselves, is even stronger when combined with AUCC and ANCC.

While the sequence of SQL INSERT statements needed for each test requirement (such as those in Figure 1b) is straightforward to determine, generating the test values is a more complex constraint satisfaction problem. One viable solution is to use search-based techniques, as described next.

B. Search-Based Test Data Generation

The *Alternating Variable Method* (AVM) [9], [15], [16] is a local search technique that has been used to successfully generate test data for programs [17], [18] and has been applied to the automated generation of data for the INSERT statements used when testing relational database schemas [7], [10]. The AVM works to optimize a vector of test values according to a fitness function. Figure 1c shows the arrangement of the values of the test case in part (b) into the vector $\vec{v} = (v_1, v_2, \ldots, v_n)$.

The AVM's main loop is shown in Figure 2a. It starts by initializing each value in the vector at random (line 2). Next it proceeds through the vector, sequentially making a series of "moves" (adjustments) to each variable value (line 5). These moves range from small *exploratory* moves to larger *pattern* moves in the same "direction" in the fitness landscape of improvement. It performs moves until a complete cycle through the vector during which no move successfully yielded a fitness improvement. At this point the algorithm may "restart" with a new randomly initialized vector. The AVM terminates when either the required test vector has been found, or a pre-defined

resource limit has been exhausted (e.g., some number of fitness function evaluations). The AVM has been adapted in various ways to generate test data for schemas, including handling for variable-length strings, dates, and NULL values [7], [10].

Fitness Function: The focus of each AVM search is a coverage requirement for one of the aforementioned coverage criteria (e.g., the satisfaction or violation of a particular integrity constraint). The fitness function for each coverage requirement is constructed using *distance functions* similar to those employed in traditional test data generation for programs [19], [20]. For example, for satisfaction of the CHECK constraint "price > discounted_price" for INSERT statement 1 of Figure 1a, the distance function $v_4 - v_3 + K$ is applied and minimized (where K is a small positive constant value, and v_3 and v_4 are the vector values of Figure 1c). Primary key, UNIQUE, and foreign key constraints involve ensuring that certain values are the same or different to those appearing in prior INSERT statements of the test, depending on whether the constraint is to be satisfied or violated. For instance, suppose in the test of Figure 1b, the second INSERT statement was required to violate the primary key of the products table, by replicating the value for product_no in the first INSERT. In this case, the distance function for generating two equal values (i.e., $|v_1 - v_5| + K$) would be applied.

Where test requirements involve violating and satisfying different constraints at once, the fitness function is composed of individual distance functions, the outputs of which are normalized and added to form the final fitness value [7], [10].

Inefficiencies: While prior work has shown that the AVM can generate test data for relational schemas [7], it is subject to inefficiencies. It can (1) waste time cycling through column values that are not involved in any of the schema's integrity constraints; (2) get stuck in local optima, requiring restarts; and finally, (3) spend time making incremental changes to a particular column value to make it equal to some other value in the test data vector, with the purpose of satisfying or violating a primary key, UNIQUE, or foreign key constraint.

The last two issues can be mitigated by first initializing the vector to a series of default values chosen for each type (e.g., zero for integers and empty strings for VARCHAR), and only randomizing the vector on the method's restart [7]. This increases the likelihood of inducing matching column values from the outset. We refer to this variant of the AVM as "AVM-D", and the traditional randomly initialized vector version as "AVM-R". One drawback of the AVM-D approach, however, is that the individual tests of the resulting test suite share a lot of the same values, thereby lowering diversity and potentially hindering the fault-finding capability of the tests.

C. Random Test Data Generation

The AVM compares very favorably with naive random search in experiments conducted on a wide range of schemas, including those with complex integrity constraints and many tables [7], [10]. Naive random search for relational schema testing simply involves repeatedly generating vectors with random values until the required vector is found, or some resource limit is exhausted (Figure 2b). In our prior work [7], we found that the AVM can attain 100% coverage for different criteria for most schemas studied. Random search never achieved full coverage for any schema, obtaining less than 70% in some instances. In these prior experiments, the random search is not so naive: Both the AVM and random search, which we referred to as Random$^+$, make use of a pool of constants mined from the schema's CHECK constraints. When a random value is required (i.e., on line 2 of both the AVM and naive random search algorithms), a value may be selected from this pool or generated freely at random, depending on some probability. The purpose of the pool is to help each algorithm satisfy and violate CHECK constraints in the schema for some requirement of a coverage criterion. Yet, while the benefit of the constant pool was not the focus of our prior work, it seems to not vastly improve random search's effectiveness.

Inefficiencies: Random$^+$ performs poorly compared to the AVM since it receives no guidance when it comes to violating primary keys and UNIQUE constraints, and for satisfying foreign keys. Each of these three aspects of test data generation for relational database schemas involves generating identical values. Yet, random search rarely generates the same values.

III. The Domino Method for Test Data Generation

Given the inefficiencies identified in prior test data generators for integrity constraints, we developed an alternative, tailored approach to the problem that uses domain knowledge. This new approach, called "DOMINO" (DOMain-specific approach to INtegrity cOnstraint test data generation), can replicate values in the test data vector for different constraint types, depending on the coverage requirement. The DOMINO algorithm in Figure 2c begins by initializing the test data vector at random, but then, in its main loop, works according to the following intuition: Where a value needs to be the same as one of a selection of values already in the vector, choose a value from that selection at random and copy it (through the COPYMATCHES function); else randomly select a new value instead through the RANDOMIZENONMATCHES function (where the "new" value is chosen from the constant pool, as described in Section II-C, or is a freshly generated value). NOT NULL constraints and CHECK constraints are handled separately through the SETORREMOVENULLS function and the SOLVECHECKCONSTRAINTS function, respectively.

While value copying and randomization may "fix" a part of the test data vector for a particular integrity constraint, it may also invalidate some other part. For example, ensuring the distinctness of a primary key value, through RANDOMIZENONMATCHES, may destroy its foreign key reference, previously set through COPYMATCHES. To handle this concern, the functions are applied one after the other in a loop, continuing until an overall solution is found or resources (i.e., a given number of algorithm iterations) are exhausted.

We now discuss how every one of the functions in DOMINO's main loop works to generate test data for satisfying/violating each of the different types of integrity constraints.

Primary Keys and "Unique" Constraints: The functions COPYMATCHES and RANDOMIZENONMATCHES work to ensure that values in INSERT statements pertaining to primary keys/UNIQUE constraints are (a) *distinct* when such constraints need to be satisfied, else ensuring those values are (b) *identical* should the constraint need to be violated. Ensuring distinctness is not usually difficult to achieve by selecting values randomly, as the probability of choosing the same value more than once is small. Nevertheless, if two values match in the vector, the second value is regenerated by RANDOMIZENONMATCHES. Alternatively, if a primary key/UNIQUE constraint is required to be violated by the test case, the values for the columns involved in the latter, constraint-violating, INSERT statement are copied from an earlier INSERT statement to the same table appearing in the test case. Suppose the primary key of the `products` table was to be violated in the test case of Figure 1 (i.e., v_1 is required to be equal to v_5), then COPYMATCHES copies v_5's value from v_1. If there is a choice of subsequent INSERT statements from which to copy a value, COPYMATCHES selects one at uniform random. If the primary key/unique constraint involves multiple columns, then multiple values are copied together from a selected prior INSERT statement in the test case.

Foreign Keys: Compared to the previously described functions, COPYMATCHES and RANDOMIZENONMATCHES work in a reverse fashion for foreign keys in that the constraint is *satisfied* when values match for the relevant columns across INSERT statements in the test case, and *violated* when they do not. As with the previous two functions, RANDOMIZENONMATCHES generates non-matching values randomly, while COPYMATCHES copies values that are supposed to match from elsewhere in the vector. Take the example of INSERTs 5 and 6 from the test of Figure 1b and the values of `product_no` and `order_id`, which individually need to match the corresponding column in the `products` and `orders` table. In both cases, two options exist. For `product_no`, a matching value is found in INSERT statements 1 and 2 (i.e., v_1 and v_5 in the vector). For `order_no`, a matching value is found in INSERT statements 3 and 4 (i.e., v_9 and v_{11}). As before, where choices exist, COPYMATCHES selects one at uniform random.

"Not Null" Constraints: Depending on the coverage requirement, the SETORREMOVENULLS function works to overwrite values in the vector with a random value where a non-NULL value is required (e.g., to satisfy a NOT NULL constraint), and copies NULL into the vector where a NULL value is required instead (e.g., to violate a NOT NULL constraint). For instance, to violate the NOT NULL constraint on the `name` column of the `products` table, the SETORREMOVENULLS function would replace the value of either v_2 or v_6 with a NULL value.

"Check" Constraints: As they involve arbitrary predicates that need to be solved, CHECK constraints cannot generally be satisfied nor violated by copying values from elsewhere in the vector. This is the role of the SOLVECHECKCONSTRAINTS function, for which this paper presents two variants.

The first variant involves generating random values, (e.g., for `price` and `discounted_price` in the `products` table). This is the default approach taken by DOMINO, and the one employed unless otherwise specified. Values are chosen at random from the domain of the column type, or from the pool of constants mined from the schema (i.e., the mechanism described for the Random$^+$ method, introduced in Section II-C). The latter mechanism is particularly useful for constraints of the form "`CHECK a IN (x, y, z)`" where the column `a` has to be set to one of "`x`", "`y`", or "`z`" to be satisfied. These values are hard to "guess" randomly without any prior knowledge, yet since the values "`x`", "`y`", or "`z`" will have been added to the constant pool, DOMINO is able to select and use them as test data values. The second variant for solving CHECK constraints leads to a special variant of DOMINO, as described next.

The Hybrid DOMINO-AVM *Method*

DOMINO does not solve CHECK constraints with domain-specific heuristics, as with other types of constraint. Instead, its random method relies on a solution being "guessed" without any guidance. Thus, we present a hybrid version of DOMINO, called "DOMINO-AVM", that uses the AVM to handle this aspect of the test data generation problem. The AVM uses the fitness function that would have been employed in the pure AVM version of Section II-B, providing guidance to the required values that may be valuable when the constraints are complex and difficult to solve by chance selection of values.

IV. EMPIRICAL EVALUATION

The aim of this paper's empirical evaluation is to determine if DOMINO will improve the efficiency and effectiveness of test data generation for relational database schemas. Our study therefore is designed to answer these three research questions:

RQ1: Test Suite Generation for Coverage—Effectiveness and Efficiency. How effective is DOMINO at generating high-coverage tests for database integrity constraints and how fast does it do so, compared to the state-of-the-art AVM?

RQ2: Fault-Finding Effectiveness of the Generated Test Suites. How effective are the test suites generated by DOMINO in regard to fault-finding effectiveness, and how do they compare to those generated by the state-of-the-art AVM?

RQ3: The Hybrid DOMINO-AVM Technique. How do test suites generated by DOMINO-AVM compare to DOMINO's in terms of efficiency, coverage, and fault-finding capability?

A. Methodology

Techniques: To answer the RQs, we empirically evaluated DOMINO, comparing it to the AVM. We used two variants of the AVM. The first was studied by McMinn et al. [7], as discussed in Section II-B, and uses default values for the first initialization of the vector (and then random re-initialization following each restart), which we refer to as "AVM-D". For a better comparison with DOMINO we also studied a variant of the AVM where all initializations are performed randomly, which we call "AVM-R". With the exception of establishing baseline coverage levels for which to compare all techniques, we did not perform any experiments with the Random$^+$ method—as described in Section II-C we already know that it

is dominated by the AVM in terms of its effectiveness (i.e., the coverage levels and mutation score of the tests it generates) [7]. Finally, for the last research question, we compared DOMINO to DOMINO-AVM, a "hybridization" with the AVM.

Subject Schemas: We performed the experiments by using the 34 relational database schemas listed in Table I. In order to answer RQ1, and to generate test suites with which to assess fault finding capability, a coverage criterion is required. For this purpose, we adopted the combination of three coverage criteria: "ClauseAICC", "AUCC", and "ANCC", as introduced in Section II-A. The reason for using this combined coverage criterion is that it was reported as the strongest to find seeded faults [7], combining the capability to find faults of both commission and omission, as described in Section II-A.

The set of 34 relational database schemas listed in Table I is larger than that featured in previous work on testing database schemas (e.g., [7], [10], [21]). Since Houkjær et al. noted that complex real-world relational schemas often include features such as composite keys and multi-column foreign-key relationships [22], the schemas we chose for this study reflect a diverse set of features, from simple instances of integrity constraints to more complex examples involving many-column foreign key relationships. The number of tables in each relational database schema varies from 1 to 42, with a range of just 3 columns in the smallest schemas, to 309 in the largest.

Our set of subject schemas are drawn from a range of sources. *ArtistSimilarity* and *ArtistTerm* are schemas that underpin part of the Million Song dataset, a freely available research dataset of song metadata [23]. *Cloc* is a schema for the database used in the popular open-source application for counting lines of program code. While it contains no integrity constraints, test requirements are still generated since the coverage criterion we use incorporates the ANCC and AUCC criteria, discussed in Section II-A. *IsoFlav_R2* belongs to a plant compound database from the U.S. Department of Agriculture, while *iTrust* is a large schema that was designed as part of a patient records medical application to teach students about software testing methods, having previously featured in a mutation analysis experiment with Java code [24]. *JWhoisServer* is used in an open-source implementation of a server for the WHOIS protocol (http://jwhoisserver.net). *MozillaExtensions* and *MozillaPermissions* are part of the SQLite databases underpinning the Mozilla Firefox browser. *RiskIt* is a database schema that forms part of a system for modeling the risk of insuring an individual (http://sourceforge.net/projects/riskitinsurance), while *StackOverflow* is the underlying schema used by the popular programming question and answer website. *UnixUsage* is from an application for monitoring and recording the use of Unix commands and *WordNet* is the database schema used in a graph visualizer for the WordNet lexical database. Other subjects were taken from the SQL Conformance Test Suite (i.e., the six "Nist–" schemas), or samples for the PostgreSQL DBMS (i.e., *DellStore*, *FrenchTowns*, *Iso3166*, and *Usda*, available from the PgFoundry.org website). The remainder were extracted from papers, textbooks, assignments, and online tutorials in which they were provided as examples (e.g., *BankAccount*, *BookTown*, *CoffeeOrders*, *CustomerOrder*, *Person*, and *Products*). While they are simpler than some of the other schemas used in this study, they nevertheless proved challenging for database analysis tools such as the DBMonster data generator [10].

DBMSs: The HyperSQL, PostgreSQL, and SQLite DBMSs hosted the subject schemas. Each of these database management systems is supported by our *SchemaAnalyst* tool [8]; they were chosen for their performance differences and varying design goals. PostgreSQL is a full-featured, extensible, and scalable DBMS, while HyperSQL is a lightweight, small DBMS with an "in-memory" mode that avoids disk writing. SQLite is a lightweight DBMS that differs in its interpretation of the SQL standard in subtly different ways from HyperSQL and PostgreSQL. A wide variety of real-world programs, from different application domains, use these three DBMSs.

RQ1: For RQ1, we ran each test data generation method with each schema and DBMS, for each coverage requirement. Each technique moves onto the next requirement (or terminating if all requirements have been considered) if test data has been successfully found, or after iterating 100,000 times if it has not. We recorded the coverage levels obtained, and the test data generation time, for 30 repetitions of each method with each of the 34 database schemas and the three DBMSs.

RQ2: For RQ2, we studied the fault-finding strength of each test suite generated for RQ1, following standard experimental protocols that use mutation analysis [25]. We adopted Wright et al.'s procedure [26], using the same set of mutation operators that mutate the schema's integrity constraints. These operators add, remove, and swap columns in primary key, `UNIQUE`, and foreign key constraints, while also inverting `NOT NULL` constraints and manipulating the conditions of `CHECK` constraints. RQ2 deems the automatically generated test suites to be effective if they can "kill" a mutant by distinguishing between it and the original schema, leading to the formulation of the higher-is-better mutation score as the ratio between the number of killed and total mutants [14], [27], [28].

RQ3: For RQ3, we measured coverage, time taken to obtain coverage, and the mutation score of the DOMINO-AVM's tests for the schemas with `CHECK` constraints (i.e., those for which the DOMINO-AVM, which uses the AVM instead of random search to solve `CHECK` constraints, will register a difference). We compared these results to those of DOMINO, which uses the default mode of random search to solve `CHECK` constraints.

Experimentation Environment: All of our experiments were performed on a dedicated Ubuntu 14.04 workstation, with a 3.13.0–44 GNU/Linux 64-bit kernel, a quad-core 2.4GHz CPU, and 12GB of RAM. All input (i.e., schemas) and output (i.e., data files) were stored on the workstation's local disk. We used the default configurations of PostgreSQL 9.3.5, HyperSQL 2.2.8, and SQLite 3.8.2, with HyperSQL and SQLite operating with their "in-memory" mode enabled.

Statistical Analysis: Using four tables, this paper reports the mean values for the 30 sets of evaluation metrics (i.e., coverage values, time to generate test suites in seconds, and mutation scores) obtained for each schema with each DBMS.

TABLE I
THE 34 RELATIONAL DATABASE SCHEMAS STUDIED

Schema	Tables	Columns	Integrity Constraints					
			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

TABLE II
MEAN COVERAGE SCORES FOR EACH TECHNIQUE

A value annotated with the "∇" symbol means that significance tests reveal that a technique obtained a significantly lower coverage score than DOMINO (written as DOM), while "Δ" means the technique obtained a significantly higher coverage than DOMINO. A "$*$" symbol indicates that the accompanying effect size was large when comparing the technique with DOMINO. Finally, AVM-R and AVM-D are the AVM variants and R^+ is short for Random$^+$.

Schema	HyperSQL				PostgreSQL				SQLite			
	DOM	AVM-R	AVM-D	R^+	DOM	AVM-R	AVM-D	R^+	DOM	AVM-R	AVM-D	R^+
ArtistSimilarity	100	100	100	$*\nabla$59	100	100	100	$*\nabla$59	100	100	100	$*\nabla$62
ArtistTerm	100	100	100	$*\nabla$60	100	100	100	$*\nabla$60	100	100	100	$*\nabla$63
BankAccount	100	100	100	$*\nabla$85	100	100	100	$*\nabla$85	100	100	100	$*\nabla$87
BookTown	99	99	99	$*\nabla$92	99	99	99	$*\nabla$92	99	99	99	$*\nabla$92
BrowserCookies	100	$*\nabla$99	100	$*\nabla$58	100	$*\nabla$99	100	$*\nabla$58	100	$*\nabla$99	100	$*\nabla$59
Cloc	100	100	100	$*\nabla$92	100	100	100	$*\nabla$92	100	100	100	$*\nabla$92
CoffeeOrders	100	100	100	$*\nabla$58	100	100	100	$*\nabla$58	100	100	100	$*\nabla$62
CustomerOrder	100	100	100	$*\nabla$42	100	100	100	$*\nabla$42	100	100	100	$*\nabla$42
DellStore	100	100	100	$*\nabla$93	100	100	100	$*\nabla$93	100	100	100	$*\nabla$93
Employee	100	100	100	$*\nabla$89	100	100	100	$*\nabla$89	100	100	100	$*\nabla$90
Examination	100	100	100	$*\nabla$83	100	100	100	$*\nabla$83	100	100	100	$*\nabla$84
Flights	100	$*\nabla$97	100	$*\nabla$59	100	$*\nabla$97	100	$*\nabla$59	100	$*\nabla$97	100	$*\nabla$58
FrenchTowns	100	100	100	$*\nabla$35	100	100	100	$*\nabla$35	100	100	100	$*\nabla$35
Inventory	100	100	100	$*\nabla$96	100	100	100	$*\nabla$96	100	100	100	$*\nabla$96
Iso3166	100	100	100	$*\nabla$85	100	100	100	$*\nabla$85	100	100	100	$*\nabla$89
IsoFlav_R2	100	100	100	$*\nabla$88	100	100	100	$*\nabla$88	100	100	100	$*\nabla$88
iTrust	100	100	100	$*\nabla$92	100	100	100	$*\nabla$92	100	100	100	$*\nabla$92
JWhoisServer	100	100	100	$*\nabla$86	100	100	100	$*\nabla$86	100	100	100	$*\nabla$87
MozillaExtensions	100	100	100	$*\nabla$88	100	100	100	$*\nabla$88	100	100	100	$*\nabla$88
MozillaPermissions	100	100	100	$*\nabla$96	100	100	100	$*\nabla$96	100	100	100	$*\nabla$96
NistDML181	100	100	100	$*\nabla$64	100	100	100	$*\nabla$64	100	100	100	$*\nabla$65
NistDML182	100	100	100	$*\nabla$62	100	100	100	$*\nabla$62	100	100	100	$*\nabla$65
NistDML183	100	100	100	100	100	100	100	100	100	100	100	100
NistWeather	100	100	100	$*\nabla$57	100	100	100	$*\nabla$57	100	100	100	$*\nabla$75
NistXTS748	100	100	100	100	100	100	100	100	100	100	100	100
NistXTS749	100	100	100	$*\nabla$86	100	100	100	$*\nabla$86	100	100	100	$*\nabla$86
Person	100	100	100	$*\nabla$93	100	100	100	$*\nabla$93	100	100	100	$*\nabla$94
Products	98	98	98	$*\nabla$70	98	98	98	$*\nabla$70	98	98	98	$*\nabla$79
RiskIt	100	100	100	$*\nabla$68	100	100	100	$*\nabla$68	100	100	100	$*\nabla$70
StackOverflow	100	100	100	$*\nabla$96	100	100	100	$*\nabla$96	100	100	100	$*\nabla$96
StudentResidence	100	100	100	$*\nabla$70	100	100	100	$*\nabla$70	100	100	100	$*\nabla$74
UnixUsage	100	100	100	$*\nabla$50	100	100	100	$*\nabla$50	100	100	100	$*\nabla$52
Usda	100	100	100	$*\nabla$90	100	100	100	$*\nabla$90	100	100	100	$*\nabla$90
WordNet	100	100	100	$*\nabla$90	100	100	100	$*\nabla$90	100	100	100	$*\nabla$89

For reasons like those of Poulding and Clark [29], we report means instead of medians: for data that was sometimes bimodal, the median value was one of the "peaks" while the mean reported a more useful statistic between the peaks.

Using statistical significance and effect size, we further compared DOMINO pairwise with every other studied technique. Since we cannot make assumptions about the distributions of the collected data, we performed tests for statistical significance using the non-parametric Mann-Whitney U test. We performed one-sided tests (sided for each technique in each pairwise comparison) with p-value < 0.01 regarded as significant. In all of the results tables, we mark a technique's value if it was significant, using the "∇" symbol if the mean result is lower compared to DOMINO or the "Δ" symbol if the mean result is higher compared to DOMINO. In addition to significance tests, we calculated effect sizes using the $\hat{A}$ metric of Vargha and Delaney [30]. We classify an effect size as "large" if $|\hat{A} - 0.5| > 0.21$. In all of the tables, we mark a technique's result with the "$*$" symbol when DOMINO performed significantly better and with a large effect size.

Threats to Validity: To control threats of both the stochastic behavior of the techniques and the possibility of operating system events interfering with the timings, we repeated the experiments 30 times. To mitigate threats associated with our statistical analysis we (a) used non-parametric statistical tests and (b) performed all our calculations with the R programming language, writing unit tests to check our results. The diverse nature of real software makes it impossible for us to claim that the studied schemas are representative of all types of relational database schemas. Yet, we attempted to select diverse schemas that came from both open-source and commercial software systems, choosing from those used in past studies [7]. Also, our results may not generalize to other DBMSs. However, HyperSQL, PostgreSQL, and SQLite are three widely used DBMSs with contrasting characteristics—and they also implement key aspects of the SQL standard related to defining schemas with various integrity constraints. Finally, it is worth noting that, while this paper does not report the cost of running the generated tests, they normally consist of a few INSERTs whose cost is negligible and thus not of practical significance.

B. Experimental Results

RQ1: Test Suite Generation for Coverage—Effectiveness and Efficiency: Table II shows the mean coverage scores for DOMINO compared to the two AVM variants and Random$^+$.

The poor results for Random$^+$ underscore that test data generation is not a trivial task for most schemas, with the exception of *NistDML183* and *NistXTS748*. Random$^+$ is outperformed by every other method. Note that while the table only reports statistical significance and a large effect size for DOMINO pairwise with every other technique, the coverage scores for the two versions of the AVM are also significantly better with a large effect size in each case when compared to Random$^+$. Since it is dominated by the three other methods, from hereon we will discount Random$^+$ as a comparison technique for generating test suites for database schemas.

The state-of-the-art AVM-D obtains 100% coverage for each schema, except for *BookTown* and *Products*, which contain infeasible coverage requirements. DOMINO matches this effectiveness (it cannot do any better, but it does not any worse

TABLE III
MEAN TEST GENERATION TIMES (IN SECONDS)

A value annotated with a "∇" symbol means that significance tests reveal that a technique required a significantly shorter time than DOMINO (DOM), while "Δ" indicates the technique needed a significantly longer time than DOMINO. A "$*$" symbol indicates that the accompanying effect size was large when comparing the technique with DOMINO.

	HyperSQL			PostgreSQL			SQLite		
Schema	DOM	AVM-R	AVM-D	DOM	AVM-R	AVM-D	DOM	AVM-R	AVM-D
ArtistSimilarity	0.49	$*\Delta$0.96	$*\Delta$0.60	1.02	$*\Delta$1.41	$*\Delta$1.08	0.29	$*\Delta$0.72	$*\Delta$0.44
ArtistTerm	0.56	$*\Delta$1.15	$*\Delta$0.72	2.60	$*\Delta$3.10	$*\Delta$2.68	0.33	$*\Delta$0.91	$*\Delta$0.54
BankAccount	0.53	$*\Delta$0.83	$*\Delta$0.76	1.33	$*\Delta$1.62	$*\Delta$1.59	0.32	$*\Delta$0.62	$*\Delta$0.57
BookTown	1.03	$*\Delta$1.41	$*\Delta$1.09	7.18	$*\Delta$7.54	7.24	0.57	$*\Delta$0.95	$*\Delta$0.64
BrowserCookies	0.66	$*\Delta$5.76	$*\Delta$3.37	3.22	$*\Delta$8.19	$*\Delta$5.85	0.42	$*\Delta$5.97	$*\Delta$3.23
Cloc	0.51	$*\Delta$0.63	$*\Delta$0.60	1.15	$*\Delta$1.28	$*\Delta$1.19	0.30	$*\Delta$0.41	$*\Delta$0.43
CoffeeOrders	0.65	$*\Delta$1.11	$*\Delta$1.08	4.43	$*\Delta$4.90	$*\Delta$4.74	0.40	$*\Delta$0.85	$*\Delta$0.82
CustomerOrder	0.86	$*\Delta$3.36	$*\Delta$1.87	7.94	$*\Delta$10.62	$*\Delta$8.65	0.55	$*\Delta$3.22	$*\Delta$1.79
DellStore	0.83	$*\Delta$1.63	$*\Delta$1.56	4.19	$*\Delta$4.96	$*\Delta$4.84	0.48	$*\Delta$1.28	$*\Delta$1.14
Employee	0.55	$*\Delta$0.82	$*\Delta$0.90	1.05	$*\Delta$1.27	$*\Delta$1.34	0.34	$*\Delta$0.59	$*\Delta$0.70
Examination	0.78	$*\Delta$1.74	$*\Delta$1.57	4.05	$*\Delta$4.94	$*\Delta$4.84	0.49	$*\Delta$1.45	$*\Delta$1.27
Flights	0.69	$*\Delta$4.93	$*\Delta$3.99	2.48	$*\Delta$6.59	$*\Delta$5.77	0.45	$*\Delta$5.23	$*\Delta$3.90
FrenchTowns	0.68	$*\Delta$1.94	$*\Delta$1.70	3.02	$*\Delta$4.17	$*\Delta$3.86	0.43	$*\Delta$1.63	$*\Delta$1.94
Inventory	0.48	$*\Delta$0.56	$*\Delta$0.60	0.70	$*\Delta$0.75	$*\Delta$0.80	0.28	$*\Delta$0.35	$*\Delta$0.44
Iso3166	0.47	$*\Delta$0.55	$*\Delta$0.55	0.48	$*\Delta$0.54	$*\Delta$0.50	0.27	$*\Delta$0.35	$*\Delta$0.40
IsoFlav_R2	0.75	$*\Delta$1.31	$*\Delta$1.27	5.13	$*\Delta$5.69	$*\Delta$5.48	0.43	$*\Delta$0.99	$*\Delta$0.93
iTrust	4.91	$*\Delta$47.91	$*\Delta$15.99	46.95	$*\Delta$85.67	$*\Delta$55.28	4.58	$*\Delta$47.11	$*\Delta$14.12
JWhoisServer	0.89	$*\Delta$2.09	$*\Delta$1.88	4.03	$*\Delta$5.15	$*\Delta$4.87	0.55	$*\Delta$1.79	$*\Delta$1.55
MozillaExtensions	0.86	$*\Delta$2.01	$*\Delta$1.92	6.36	$*\Delta$7.62	$*\Delta$7.34	0.55	$*\Delta$1.65	$*\Delta$1.55
MozillaPermissions	0.51	$*\Delta$0.61	$*\Delta$0.66	1.08	$*\Delta$1.16	$*\Delta$1.19	0.31	$*\Delta$0.40	$*\Delta$0.49
NistDML181	0.53	$*\Delta$0.83	$*\Delta$0.71	1.55	$*\Delta$1.80	$*\Delta$1.71	0.32	$*\Delta$0.62	$*\Delta$0.54
NistDML182	0.76	$*\Delta$2.36	$*\Delta$1.94	5.74	$*\Delta$7.43	$*\Delta$6.81	0.50	$*\Delta$2.10	$*\Delta$2.09
NistDML183	0.51	$*\Delta$0.58	$*\Delta$0.64	1.32	$*\Delta$1.44	$*\Delta$1.44	0.30	$*\Delta$0.36	$*\Delta$0.48
NistWeather	0.71	$*\Delta$1.42	$*\Delta$1.31	1.93	$*\Delta$2.64	$*\Delta$2.52	0.48	$*\Delta$1.14	$*\Delta$1.22
NistXTS748	0.48	$*\Delta$0.53	$*\Delta$0.61	0.61	$*\Delta$0.66	$*\Delta$0.71	0.28	$*\Delta$0.33	$*\Delta$0.50
NistXTS749	0.55	$*\Delta$0.78	$*\Delta$0.82	1.54	$*\Delta$1.81	$*\Delta$1.77	0.33	$*\Delta$0.57	$*\Delta$0.69
Person	0.55	$*\Delta$1.05	$*\Delta$1.60	0.68	$*\Delta$1.17	$*\Delta$1.73	0.34	$*\Delta$0.87	$*\Delta$1.56
Products	0.71	$*\Delta$1.72	$*\Delta$1.71	2.30	$*\Delta$3.28	$*\Delta$3.40	0.47	$*\Delta$1.33	$*\Delta$1.38
RiskIt	1.00	$*\Delta$3.62	$*\Delta$2.31	11.70	$*\Delta$14.72	$*\Delta$12.53	0.63	$*\Delta$3.48	$*\Delta$1.99
StackOverflow	0.82	$*\Delta$1.17	$*\Delta$1.47	4.66	$*\Delta$4.83	$*\Delta$5.01	0.48	$*\Delta$0.84	$*\Delta$1.12
StudentResidence	0.59	$*\Delta$0.97	$*\Delta$0.78	1.43	$*\Delta$1.72	$*\Delta$1.54	0.38	$*\Delta$0.75	$*\Delta$0.63
UnixUsage	0.87	$*\Delta$3.48	$*\Delta$1.93	11.11	$*\Delta$13.31	$*\Delta$11.52	0.52	$*\Delta$2.99	$*\Delta$1.67
Usda	0.86	$*\Delta$1.40	$*\Delta$1.53	6.23	$*\Delta$6.40	$*\Delta$6.47	0.49	$*\Delta$1.01	$*\Delta$1.03
WordNet	0.68	$*\Delta$0.97	$*\Delta$1.13	3.64	$*\Delta$3.92	$*\Delta$3.99	0.40	$*\Delta$0.67	$*\Delta$0.84

TABLE IV
MEAN MUTATION SCORES

A value annotated with a "∇" symbol means that significance tests reveal that a technique obtained a significantly lower mutation score than DOMINO (DOM), while "Δ" indicates the technique obtained a significantly higher mutation score than DOMINO. A "$*$" symbol denotes a large effect size when comparing a technique's score with DOMINO's.

	HyperSQL			PostgreSQL			SQLite		
Schema	DOM	AVM-R	AVM-D	DOM	AVM-R	AVM-D	DOM	AVM-R	AVM-D
ArtistSimilarity	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
ArtistTerm	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
BankAccount	95.9	95.5	$*\nabla$88.5	95.9	95.5	$*\nabla$88.5	96.4	96.1	$*\nabla$86.7
BookTown	99.5	99.4	$*\nabla$97.6	99.5	99.4	$*\nabla$97.6	99.1	99.0	$*\nabla$85.5
BrowserCookies	96.3	∇95.6	$*\nabla$92.3	96.3	∇95.6	$*\nabla$92.3	95.9	96.1	$*\nabla$86.5
Cloc	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
CoffeeOrders	100.0	100.0	100.0	100.0	100.0	100.0	98.6	$*\Delta$100.0	$*\nabla$94.6
CustomerOrder	97.5	97.5	$*\nabla$94.0	97.5	97.4	$*\nabla$93.9	98.0	98.0	$*\nabla$95.2
DellStore	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Employee	97.7	97.6	$*\nabla$95.3	97.7	97.6	$*\nabla$95.3	97.3	97.4	$*\nabla$84.1
Examination	100.0	∇99.8	$*\nabla$97.3	100.0	∇99.8	$*\nabla$97.3	99.2	99.6	$*\nabla$85.8
Flights	99.8	$*\nabla$97.9	$*\nabla$95.2	99.8	$*\nabla$97.9	$*\nabla$95.2	100.0	$*\nabla$98.2	$*\nabla$84.3
FrenchTowns	94.3	94.3	$*\nabla$82.5	94.3	94.3	$*\nabla$82.5	94.6	94.6	$*\nabla$83.3
Inventory	100.0	100.0	$*\nabla$87.5	100.0	100.0	$*\nabla$88.2	100.0	100.0	$*\nabla$75.0
Iso3166	99.6	99.6	$*\nabla$77.8	99.6	99.6	$*\nabla$77.8	99.7	99.7	$*\nabla$80.0
IsoFlav_R2	99.7	99.8	$*\nabla$87.0	99.7	99.8	$*\nabla$87.0	99.7	99.8	$*\nabla$84.4
iTrust	99.7	$*\nabla$99.6	$*\nabla$95.8	99.7	$*\nabla$99.6	$*\nabla$95.8	99.2	99.2	$*\nabla$83.6
JWhoisServer	99.6	99.6	$*\nabla$78.7	99.6	99.6	$*\nabla$78.7	99.6	99.5	$*\nabla$76.6
MozillaExtensions	99.8	99.6	$*\nabla$82.1	99.8	99.6	$*\nabla$82.1	99.7	99.5	$*\nabla$71.3
MozillaPermissions	100.0	100.0	100.0	100.0	100.0	100.0	99.8	99.8	$*\nabla$76.7
NistDML181	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
NistDML182	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
NistDML183	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
NistWeather	98.2	Δ100.0	$*\nabla$93.3	98.2	Δ100.0	$*\nabla$93.3	98.4	Δ100.0	$*\nabla$93.8
NistXTS748	93.3	93.7	$*\nabla$88.2	93.3	93.7	$*\nabla$88.2	92.9	93.3	$*\nabla$87.5
NistXTS749	95.0	95.0	95.0	95.0	95.0	95.0	91.7	$*\Delta$96.0	92.0
Person	97.8	96.5	$*\nabla$81.0	97.8	96.5	$*\nabla$81.0	98.8	97.3	$*\nabla$81.8
Products	87.2	87.1	∇86.5	87.2	87.1	∇86.5	87.8	87.7	∇87.1
RiskIt	100.0	100.0	$*\nabla$99.5	100.0	100.0	$*\nabla$99.5	99.5	$*\Delta$99.9	$*\nabla$89.3
StackOverflow	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
StudentResidence	97.2	∇96.5	$*\nabla$94.4	97.2	∇96.5	$*\nabla$94.4	95.7	96.6	$*\nabla$87.2
UnixUsage	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	$*\nabla$98.2
Usda	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
WordNet	97.8	97.6	$*\nabla$93.7	97.8	97.6	$*\nabla$93.7	98.5	97.9	$*\nabla$87.4

either), while AVM-R has difficulties with *BrowserCookies* and *Flights*. For these schemas, AVM-R has trouble escaping a local optimum for a particular coverage requirement. It restarts many times, but fails to find test data before its resources are exhausted. The use of default values always provides a good starting point for AVM-D to cover the requirements concerned, and as such, it does not suffer from these problems. DOMINO does not use a fitness function, and so does not face this issue.

Thus, for coverage scores, DOMINO performs identically to AVM-D, but better than AVM-R for some schemas, and significantly better than Random$^+$ for all non-trivial schemas.

Table III gives the mean times for each technique to obtain the coverage scores in Table II, excluding Random$^+$. The results show that DOMINO outperforms both of the AVM variants, which incur significantly higher times in each case, with a large effect size. The difference is most noticeable for larger schemas (i.e., *iTrust* and *BrowserCookies*). With *iTrust*, DOMINO is approximately 40 seconds faster than AVM-R with each of the DBMSs, representing a speedup of 8–10 times for HyperSQL and SQLite. Compared to AVM-D, DOMINO is approximately 10 seconds faster for each DBMS. For smaller schemas, the differences are significant but less pronounced. Although DOMINO is faster than the AVM variants for these schemas, the practical difference is almost negligible.

Concluding RQ1, DOMINO yields the same coverage scores as the state-of-the-art AVM-D, but in less time. Compared to DOMINO, AVM-R is slower and has slightly worse coverage.

RQ2: Fault-Finding Effectiveness of the Generated Test Suites: Table IV shows the mean mutation scores obtained by each technique's generated test suites. The results show that DOMINO achieved significantly higher mutation scores than the state-of-the-art AVM-D technique for 20–23 of the 34 schemas, depending on the DBMS, with a large effect size in almost every case. AVM-R is more competitive with DOMINO, however. For these two techniques there are fewer differences in effectiveness. Therefore, it seems that developing test cases from a random starting point is important for mutation killing effectiveness. AVM-D starts from the same default values, which may remain unchanged, depending on the test requirement. Ultimately, there is less diversity across this method's test suites, leading them to kill fewer mutants.

Variations in DOMINO's effectiveness compared to AVM-R stem from differences in the approach taken for generating test data: DOMINO always copies values where it can for certain types of requirement and integrity constraint, whereas AVM-R may legitimately opt to use `NULL` instead of a matching value. For instance, DOMINO satisfies foreign keys with `NULL` values, unless there are `NOT NULL` constraints on the columns of the key. The occasional use of `NULL` leads AVM-R to kill more mutants than DOMINO for some schemas, and fewer for others. The relative advantages depend on the DBMS: For HyperSQL and PostgreSQL, DOMINO obtains a significantly higher mutation score for five schemas, while AVM-R performs better for one schema. While some of these comparisons are accompanied by a large effect size, the differences in means are usually marginal. Conversely for the SQLite DBMS, DOMINO is better for two schemas, while AVM-R is better for three. This is likely because SQLite allows the use of `NULL` values in primary key columns, giving more opportunity for `NULL` to be used as a data value in tests for schemas

TABLE V
MEAN RESULTS OF DOMINO-AVM COMPARED TO DOMINO FOR SUBJECT SCHEMAS THAT HAVE CHECK CONSTRAINTS

A value annotated with a "∇" symbol means that significance tests reveal that DOMINO (DOM) obtained a significantly lower result than DOMINO-AVM (DOM-AVM), while "△" shows that DOMINO obtained a significantly higher result than DOMINO-AVM. A "∗" symbol indicates that the effect size was large when comparing the technique with DOMINO.

	Coverage						Timing						Mutation Score					
	HyperSQL		PostgreSQL		SQLite		HyperSQL		PostgreSQL		SQLite		HyperSQL		PostgreSQL		SQLite	
Schema	DOM-AVM	DOM	DOM-AVM	DOM	DOM-AVM	DOM	DOM-AVM	DOM	DOM-AVM	DOM	DOM-AVM	DOM	DOM-AVM	DOM	DOM-AVM	DOM	DOM-AVM	DOM
BookTown	99.3	99.3	99.3	99.3	99.3	99.3	1.08	∗∇1.03	7.25	7.18	0.61	∗∇0.57	99.4	99.5	99.4	99.5	99.1	99.1
BrowserCookies	100.0	100.0	100.0	100.0	100.0	100.0	0.69	∗∇0.66	3.21	3.22	0.44	∗∇0.42	95.9	96.3	95.9	96.3	96.0	95.9
CustomerOrder	100.0	100.0	100.0	100.0	100.0	100.0	0.91	∗∇0.86	8.13	∗∇7.94	0.60	∗∇0.55	97.4	97.5	97.4	97.5	98.0	98.0
Employee	100.0	100.0	100.0	100.0	100.0	100.0	0.58	∗∇0.55	1.08	1.05	0.36	∗∇0.34	97.0	△97.7	97.0	△97.7	96.7	97.3
Examination	100.0	100.0	100.0	100.0	100.0	100.0	0.82	∗∇0.78	4.14	4.05	0.52	∗∇0.49	99.6	△100.0	99.6	△100.0	99.2	99.2
Flights	100.0	100.0	100.0	100.0	100.0	100.0	0.77	∗∇0.69	2.59	∗∇2.48	0.51	∗∇0.45	100.0	99.8	100.0	99.8	100.0	100.0
iTrust	100.0	100.0	100.0	100.0	100.0	100.0	4.82	△4.91	46.46	46.95	4.54	4.58	99.6	∗△99.7	99.6	∗△99.7	99.1	99.2
NistWeather	100.0	100.0	100.0	100.0	100.0	100.0	0.72	0.71	2.00	∇1.93	0.49	0.48	99.7	∇98.2	99.7	∇98.2	99.9	∇98.4
NistXTS748	100.0	100.0	100.0	100.0	100.0	100.0	0.49	∗∇0.48	0.61	0.61	0.29	∗∇0.28	93.3	93.3	93.3	93.3	92.9	92.9
NistXTS749	100.0	100.0	100.0	100.0	100.0	100.0	0.56	∗∇0.55	1.61	1.54	0.34	∗∇0.33	95.0	95.0	95.0	95.0	92.0	91.7
Person	100.0	100.0	100.0	100.0	100.0	100.0	0.74	∗∇0.55	0.85	∗∇0.68	0.51	∗∇0.34	97.0	97.8	97.0	97.8	97.0	△98.8
Products	97.9	97.9	97.9	97.9	98.1	98.1	0.75	∗∇0.71	2.38	∇2.30	0.50	∗∇0.47	87.1	87.2	87.1	87.2	87.7	87.8
StudentResidence	100.0	100.0	100.0	100.0	100.0	100.0	0.58	0.59	1.38	1.43	0.36	∗△0.38	96.6	△97.2	96.6	△97.2	95.0	95.7

that it hosts. AVM-R can exploit this opportunity by using NULL whereas DOMINO does not—in turn leading to more times for which using NULL can result in the killing of a mutant.

For nine schemas, a 100% mutation score was achieved regardless of technique and DBMS. Closer inspection revealed that these schemas had few or simple constraints (i.e., all NOT NULL constraints), the mutants of which were easy to kill.

The schemas with the weakest mutation score was *Products*, with a maximum of 87.8% with DOMINO and the SQLite DBMS. Closer inspection revealed that this schema had many live mutants generated as a result of CHECK constraints, thus motivating the hybrid DOMINO-AVM investigated in RQ3.

To conclude for RQ2, the results show that DOMINO is more effective at killing mutants than the state-of-the-art AVM-D technique. The results reveal few differences in the mutation score of DOMINO compared to AVM-R. Yet, RQ1 showed that DOMINO generates data significantly faster than AVM-R—with marginally better coverage as well—and therefore is the most effective *and* efficient technique of the three.

RQ3: The Hybrid DOMINO-AVM *Technique:* This research question investigates the use of the AVM as the CHECK constraint solver in DOMINO-AVM, comparing it to DOMINO's use of the random-based solver. As shown by the answer to RQ1, search-based methods excel compared to random-based approaches for relational schema test data generation, and therefore a hybrid DOMINO-AVM technique may be fruitful.

For the schemas with CHECK constraints—that is, the schemas for which DOMINO-AVM could potentially improve upon DOMINO—Table V reports the mean results of coverage, test suite generation time, and mutation scores. For ease of comparison, we re-report the results of DOMINO for these schemas alongside those obtained for DOMINO-AVM.

DOMINO-AVM achieves full coverage for all schemas, except for those that involve infeasible test requirements, as did DOMINO. Perhaps surprisingly, however, DOMINO-AVM is generally no better in terms of time to generate the test suites, and is in fact reported as significantly worse in the table for several schemas, with an accompanying large effect size. This indicates that, for this study's schemas, random search can successfully solve the CHECK constraints. Thus, using the AVM is of no additional benefit in terms of speeding up the test data generation process. When a "magic" constant is involved, DOMINO mines it from the schema and then solves the constraint by randomly selecting it from the constant pool.

In terms of mutation score, there is one schema (i.e., *NistWeather*) where DOMINO-AVM is significantly better than DOMINO for all DBMSs, and cases where the reverse is true (e.g., *Employee* and *Examination*) but for the HyperSQL and PostgreSQL DBMSs only. The actual differences in means are small, and are accounted for by the random solver's use of constants mined from the schema with DOMINO, as opposed to the search-based approach taken by DOMINO-AVM. In the cases where DOMINO does better, it is for relational constraints where a value is being compared to a constant (e.g., `x >= 0`). The use of the seeded constant (i.e., `0` for `x`) means that a boundary value is being used, which helps to kill the mutants representing a changed relational operator (e.g., from `>=` to `>`).

On the other hand, DOMINO-AVM may use any value that satisfies the constraint (e.g., `1` for `x`), according to the fitness function, that may not fall on the boundary and not kill the mutant. Conversely, not using constant seeding can help to kill other mutant types, which is what happens with *NistWeather*. Here, DOMINO only satisfies a CHECK constraint by using a value mined from the schema, leading to a repetition of the same value across different INSERT statements of a test case. In contrast, the fitness function gives guidance to different values that satisfy the CHECK constraint for DOMINO-AVM. This increased diversity helps DOMINO-AVM to consistently kill an additional mutant that DOMINO was unable to kill.

The conclusion for RQ3 is that the AVM's potential to improve the generation of data for test requirements involving CHECK constraints is only of benefit for a few cases. The use of random search, as employed by DOMINO, achieves similar results to DOMINO-AVM in a shorter amount of time.

Overall Conclusions: The results indicate that DOMINO is the best method, achieving the highest mutation scores (RQ2) and requiring the least time to generate test suites (RQ1). The coverage it obtains is optimal and is comparable with the previous state-of-the-art-technique, AVM-D. Yet, it generates test data that is more diverse, which has a positive impact on the fault-finding capability of its test suites. Given that DOMINO handles CHECK constraints randomly, while the AVM is fitness-guided, a hybrid technique would seem fruitful. However, the results from RQ3 contradict this intuition. Instead, it seems

that AVM's superiority over random search, as shown by the results for RQ1, is to do with generating test data for other types of integrity constraint. For the studied schemas, test data can be effectively generated for CHECK constraints with a random method—although DOMINO-AVM does generate tests that are better at killing mutants for one particular subject.

V. RELATED WORK

In prior work on testing databases, Bati et al., Letarte et al., and Slutz developed random and search-based methods for automatically generating structured query language (SQL) queries [31]–[33]. These queries are designed to test a DBMS, rather than the integrity constraints of a relational database schema. In terms of automated data generation for databases, other techniques have been developed, but with the specific purpose of testing the performance of a database rather than the integrity constraints of a schema. That is, the data generated is always intended to satisfy all of a schema's integrity constraints—it neither exercises them as *false* nor tests specific constraints or sub-constraints. For instance, DBMonster, studied by Kapfhammer et al., is not equipped to test schema integrity constraints, obtaining very low coverage scores [10]. Similarly, Databene Benerator [34] and DTM data generator [35] are only used to populate databases with valid test data. Moreover, all of these tools rely on random data generation or picking of values from a pre-composed library.

In contrast to the aforementioned prior work and similar to our *SchemaAnalyst* tool [8], this paper focuses on generating tests for database schemas. *SchemaAnalyst* is a tool designed to test relational database schemas using the AVM [7], [8], which is also studied as part of this paper. However, these prior works only present search-based and random methods, never considering approaches specifically tailored to the problem domain, which is the main contribution of this paper.

Critically, DOMINO copies the data values previously seen elsewhere in a test case, incorporating random search for CHECK constraints and always using values mined from the schema. Both of these strategies have parallels with the value seeding approach of Rojas et al. [36], which seeds an evolutionary algorithm with constants used in a Java program, while also re-using values previously generated during the search.

This paper also uses mutation analysis for part of its evaluation. Tuya et al. [28] developed a set of mutation operators for SQL queries such as a SELECT statement. However, since this paper concentrates on testing the integrity constraints of a relational database schema, we use the operators proposed by Kapfhammer et al. [10] and Wright et al. [26]. These operators add, swap, and remove columns from integrity constraints such as primary keys, foreign keys, and UNIQUE constraints, while also removing and adding NOT NULL constraints, and introducing small changes to the conditions of CHECK constraints.

This paper's results indicate that the diversity of the generated test data plays an important role in killing these mutants. The relationship between diversity and fault-finding capability has been studied previously. For example, Alshahwan and Harman [37] reported that the output uniqueness of a program under test positively influences test effectiveness, while Hemmati et al. [38] noted that rewarding diversity leads to better fault detection when selecting model-based tests. DOMINO can generate diverse values as it is tailored to re-use values only when needed, randomly selecting data values otherwise.

It is worth noting that there are several examples of prior work that consider the testing, debugging, and analysis of database applications. For instance, Chays et al. [39] and Zhou et al. [40] respectively created methods for automatically generating test cases and performing mutation analysis. It is also possible to use combinatorial data generators to test database applications, as was investigated by Li et al. [41]. While this paper adopts coverage criteria for database schema testing, prior work has also developed test adequacy criteria for entire database applications [42], [43]. There are also many methods customized for database applications that perform, for instance, dynamic analysis [44], regression testing [45], test execution [46], coverage monitoring [47], and fault localization [48]. All of the aforementioned methods complement this paper's domain-specific technique for schema testing.

VI. CONCLUSIONS AND FUTURE WORK

Since databases are a valuable asset protected by a schema, this paper introduced DOMINO, a method for automatically generating test data that systematically exercises the integrity constraints in relational database schemas. Prior ways to automate this task (e.g., [7], [10]) adopted search-based approaches relying on the Alternating Variable Method (AVM). Even though DOMINO is more efficient than the AVM, its domain-specific operators enable it to create tests that match the coverage of those produced by this state-of-the-art method.

DOMINO can also generate tests that are better at killing mutants than AVM-D, a version of the AVM that starts the search from a set of default values (e.g., '0' for integers or the empty string for strings). This is advantageous because the test data values generated by DOMINO, not being based on default values, have greater diversity. Following this insight, we also studied an AVM that starts with random values. Experiments show that, while AVM-R has a similar mutant killing capability to DOMINO, its overall coverage scores are not as high as the presented method's and it takes significantly longer to generate its tests. Finally, we compared DOMINO to a hybridization combining the domain-specific operators with the use of AVM for the CHECKs, finding that this alternative is less efficient that the presented method and no more effective.

Given the efficiency and effectiveness of DOMINO, we plan to experimentally compare it to methods that leverage constraint solvers [49], evolutionary algorithms [50], and other hybrid approaches [51]. Since prior work has shown the importance of human-readable test data [52], [53], we will also study whether testers understand DOMINO's data values. Building on the version of DOMINO that we have already integrated into *SchemaAnalyst* and made available at [11], we will also release all of the techniques developed as part of future work, ultimately yielding a comprehensive and publicly available approach to automatically testing database schemas.

REFERENCES

[1] P. Glikman and N. Glady, "What's the value of your data?" https://goo.gl/bFZKeR, 2015.

[2] K. Cagle, "Seven data modeling mistakes," https://goo.gl/UywFw5, 2015.

[3] L. Davidson, "Ten common database design mistakes," https://goo.gl/kYtf2z, 2007.

[4] B. Ericsson, "What's wrong with foreign keys?" https://goo.gl/zpTaYN, 2012.

[5] J. James, "Five common database development mistakes," https://goo.gl/i7X1vK, 2012.

[6] S. Guz, "Basic mistakes in database testing," https://goo.gl/AE5haq, 2011.

[7] 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.

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

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

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

[11] "SchemaAnalyst test data generation tool with the DOMINO method," https://github.com/schemaanalyst/schemaanalyst.

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

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

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

[15] J. Kempka, P. McMinn, and D. Sudholt, "Design and analysis of different alternating variable searches for search-based software testing," *Theoretical Computer Science*, vol. 605, 2015.

[16] P. McMinn, G. M. Kapfhammer, and C. J. Wright, "Virtual mutation analysis of relational database schemas," in *Proceedings of the 11th International Workshop on Automation of Software Test*, 2016.

[17] G. Fraser, A. Arcuri, and P. McMinn, "A memetic algorithm for whole test suite generation," *Journal of Systems and Software*, vol. 103, 2015.

[18] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global and hybrid search," *Transactions on Software Engineering*, vol. 36, no. 2, 2010.

[19] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, 2004.

[20] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in *Proceedings of the 13th International Conference on Automated Software Engineering*, 1998.

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

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

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

[24] B. Smith and L. Williams, "An empirical evaluation of the MuJava mutation operators," in *Proceedings of the 3rd International Workshop on Mutation Analysis*, 2007.

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

[26] 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 14th International Conference on Quality Software*, 2014.

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

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

[29] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," *Transactions on Software Engineering*, vol. 36, no. 6, 2010.

[30] 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.

[31] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A genetic approach for random testing of database systems," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007.

[32] D. Letarte, F. Gauthier, E. Merlo, N. Sutyanyong, and C. Zuzarte, "Targeted genetic test SQL generation for the DB2 database," in *Proceedings of the 5th International Workshop on Testing Database Systems*, 2012.

[33] D. R. Slutz, "Massive stochastic testing of SQL," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, 1998.

[34] "Databene Benerator," http://databene.org/databene-benerator.html, (Accessed 03/07/2012).

[35] "DTM Data Generator," http://www.sqledit.com/dg/index.html, (Accessed 04/10/2017).

[36] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *Software Testing, Verification and Reliability*, vol. 26, no. 5, 2016.

[37] N. Alshahwan and M. Harman, "Augmenting test suites effectiveness by increasing output diversity," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[38] H. Hemmati, A. Arcuri, and L. Briand, "Reducing the cost of model-based testing through test case diversity," in *Proceedings of the 22nd International Conference on Testing Software and Systems*, 2010.

[39] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "An AGENDA for testing relational database applications," *Software Testing, Verification and Reliability*, vol. 14, no. 1, 2004.

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

[41] N. Li, Y. Lei, H. R. Khan, J. Liu, and Y. Guo, "Applying combinatorial test data generation to big data applications," in *Proceedings of the 31st International Conference on Automated Software Engineering*, 2016.

[42] W. G. J. Halfond and A. Orso, "Command-form coverage for testing database applications," in *Proceedings of 21st International Conference on Automated Software Engineering*, 2006.

[43] G. M. Kapfhammer and M. L. Soffa, "A family of test adequacy criteria for database-driven applications," in *Proceedings of the 11th Symposium on the Foundations of Software Engineering*, 2003.

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

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

[46] ——, "Parallel execution of test runs for database application systems," in *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

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

[48] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold, "Localizing SQL faults in database applications," in *Proceedings of the 26th International Conference on Automated Software Engineering*, 2011.

[49] S. Khalek and S. Khurshid, "Systematic testing of database engines using a relational constraint solver," in *Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, 2011.

[50] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2004.

[51] P. McMinn and M. Holcombe, "Evolutionary testing using an extended chaining approach," *Evolutionary Computation*, vol. 14, no. 1, 2006.

[52] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proceedings of the International Workshop on Software Test Output Validation*, 2010.

[53] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, 2013.