An Analysis of the Effectiveness of Different Coverage Criteria for Testing Relational Database Schema Integrity Constraints*

Phil McMinn†, Chris J. Wright* and Gregory M. Kapfhammer‡

Abstract

Despite industry advice to the contrary, there has been little work that has sought to test that a relational database's schema has correctly specified integrity constraints. These critically important constraints ensure the coherence of data in a database, defending it from manipulations that could violate requirements such as "usernames must be unique" or "the host name cannot be missing or unknown". This paper is the first to propose coverage criteria, derived from logic coverage criteria, that establish different levels of testing for the formulation of integrity constraints in a database schema. These range from simple criteria that mandate the testing of successful and unsuccessful `INSERT` statements into tables to more advanced criteria that test the formulation of complex integrity constraints such as multi-column `PRIMARY KEY`s and arbitrary `CHECK` constraints. Due to different vendor interpretations of the structured query language (SQL) specification with regard to how integrity constraints should actually function in practice, our criteria crucially account for the underlying semantics of the database management system (DBMS). After formally defining these coverage criteria and relating them in a subsumption hierarchy, we present two approaches to automatically generating tests that satisfy the criteria. We then describe the results of an empirical study that uses mutation analysis to investigate the fault-finding capability of data generated when our coverage criteria are applied to a wide variety of relational schemas hosted by three well-known and representative DBMSs — HyperSQL, PostgreSQL and SQLite. In addition to revealing the complementary fault-finding capabilities of the presented criteria, the results show that mutation scores range from as low as just 12% of mutants being killed with the simplest of criteria to 96% with the most advanced.

1 Introduction

It is often said that the data in an organization's database is one of the most valuable assets that it can own [1]. Having recently found use in application areas ranging from politics and government [2] to the simulation of astrophysical phenomenon [3], the relational database is a prevalent data management technology. In the context of relational databases, *integrity constraints* are tasked with protecting the coherency and consistency of the managed data. Responsible for ensuring that the database management system (DBMS) rejects structured query language (SQL) statements that would introduce non-complying data, the integrity constraints prevent the database from being "corrupted" with potentially invalid entries.

Integrity constraints are part of the definition of a *schema* for a relational database [1]. A relational database schema defines what types of data will be stored in the database and how they are grouped into tables. Through integrity constraints, a developer can specify which columns of which tables should have unique data values (through `PRIMARY KEY` constraints and `UNIQUE` constraints), which data values should not be marked as "unknown" or "missing" (through `NOT NULL` constraints), which values should be related to values in other tables (through `FOREIGN KEY` constraints) and which values should satisfy domain-specific predicates (through the specification of `CHECK` constraints). Surprisingly, and despite industry advice to the contrary [4], there has been very little work on testing relational database schemas to ensure that integrity constraints have been specified correctly. This could lead to serious bugs in a database application, if, for example, two users are allowed to have the same identification value, or if invalid monetary or stock values are permitted into the database.

Test suites for database schemas are not only valuable for checking the correct formulation of integrity constraints for a new application, they are also an important resource for regression testing and for when the underlying database management system (DBMS) is changed. This is because different DBMS vendors have different interpretations of the SQL standard, and thus implement integrity constraints differently. For example, for most DBMSs, a `PRIMARY`

*Please cite the journal version of this paper: *Phil McMinn, Chris J. Wright and Gregory M. Kapfhammer, "The Effectiveness of Test Coverage Criteria for Relational Database Schema Integrity Constraints", ACM Transactions on Software Engineering and Methodology, To Appear.*

†Dept. of Computer Science, University of Sheffield, UK

‡Dept. of Computer Science, Allegheny College, USA

KEY constraint declared over columns of a database implicitly means that those columns should also not be NULL. This is not the case for SQLite. For most DBMSs, NULL values are freely allowed for columns declared as part of UNIQUE constraints. Microsoft SQL Server, however, will reject NULL values after the first, on the basis that further NULL values are not "unique". These subtle differences can cause problems when an application is developed using a local lightweight DBMS such as SQLite, but then deployed to an enterprise DBMS such as PostgreSQL. It is therefore important to verify, through a series of systematically designed and highly effective tests, both that the schema is behaving the same across the different DBMSs with which the database is intended to be used and correct after its initial specification and subsequent modifications.

In this paper, we propose a family of coverage criteria for testing the specification of integrity constraints in a relational database schema. We define two flavors of criterion: "constraint coverage" criteria and "column coverage" criteria. With constraint coverage, we show how integrity constraints can be expressed as predicates, and how work on logic coverage [5] can be used as the foundation for a set of criteria that handles three-valued logic, dependencies on existing table data and testing at different levels of thoroughness. With the "column coverage" flavor of criteria, we show how test requirements can be generated that may check for integrity constraints potentially omitted from the schema's definition. Since these criteria can support the identification of both faults of omission and commission, they contrast with most other work on test coverage, which has historically been largely restricted to testing "what's there" and normally limited to only detecting faults of commission [6, 7].

Following this, we propose a framework for automatically generating concrete test cases for the test requirements necessitated by our criteria. Currently, this framework employs two types of search methods for generating the test cases. The first is an augmented random approach, seeded with constants mined from the databases' relational schema. Since this approach is known to struggle with difficult test requirements whose test data values cannot be found easily by "chance" [8], we also use a directed search-based approach based on Korel's Alternating Variable Method (*AVM*) [9, 10]. Search-based approaches use a fitness function to guide the search towards required test data. Then, our framework formulates the generated test values into SQL INSERT statements. By checking whether the DBMS accepts (i.e., the data is entered into the database) or rejects (i.e., the data is not added to the database) the INSERT statements, the tester can ascertain the correctness of the integrity constraint formulation in the schema.

Finally, we present the results of an empirical study in which we evaluate our criteria using the test suites that the two data generators created for thirty-two relational schemas hosted by three real-world DBMSs: HyperSQL, PostgreSQL and SQLite. To evaluate the quality of the generated tests, we perform a mutation analysis, simulating "faults" by making small changes to the original schemas under test. The percentage of mutants that were detected by our test suites ranged widely depending on the criteria used. For very simple criteria — which are arguably only marginally better than a series of ad-hoc sanity checks that a tester might manually perform — only low detection rates are achievable, where just 12% of mutants are "killed". For the most advanced criteria that engender the most thorough set of test requirements, higher detection rates are possible where up to 96% of the mutants can be killed. Since the experiments consider both a wide variety of schemas — many of which are from real-world databases — and three representative database management systems, we judge that they reveal the benefit of using the presented coverage criteria to guide the testing of relational database integrity constraints.

Therefore, the contributions of this paper are as follows:

1. The definition of nine coverage criteria that are organized in subsumption hierarchies; these criteria aim to detect both faults of commission and omission in the schema of a relational database that is hosted by three representative and widely used database management systems (Section 3).

2. A framework that uses the presented coverage criteria to guide the generation of test cases that are formulated as complete SQL INSERT statements; currently, this automated and extensible framework supports both a random and a search-based method for test suite generation (Section 4)

3. The results of an empirical study that uses mutation analysis to assess the testing strength of our coverage criteria. Incorporating thirty-two diverse schemas and three well-known and representative DBMSs, the study finds that there is a broad range of mutation scores from the weakest criteria to the strongest. The experiments also reveal that criteria in different subsumption hierarchies are more suited to killing different types of mutants and that criteria can be combined to give the highest fault-finding capability (Section 5).

```
1  CREATE TABLE places (
2   host TEXT NOT NULL,
3   path TEXT NOT NULL,
4   title TEXT,
5   visit_count INTEGER,
6   fav_icon_url TEXT,
7   PRIMARY KEY(host, path)
8  );
9
10 CREATE TABLE cookies (
11  id INTEGER PRIMARY KEY NOT NULL,
12  name TEXT NOT NULL,
13  value TEXT,
14  expiry INTEGER,
15  last_accessed INTEGER,
16  creation_time INTEGER,
17  host TEXT,
18  path TEXT,
19  UNIQUE(name, host, path),
20  FOREIGN KEY(host, path) REFERENCES places(host, path),
21  CHECK (expiry = 0 OR expiry > last_accessed),
22  CHECK (last_accessed >= creation_time)
23 );
```

Figure 1: The example *BrowserCookies* schema, for managing cookie information in an Internet browser

2 Background

2.1 Relational Database Schemas and Integrity Constraints

Databases form the backbone of many different types of software applications, powering everything from operating systems, mobile devices, content delivery networks, and websites, to large organizations including banks and stock exchanges [1, 11]. They range from very large databases that serve multiple applications, often concurrently, to smaller databases embedded within programs to manage runtime information and achieve persistence of data.

Despite the age of "big data", and the existence of approaches to handle either semi-structured or unstructured data (e.g., [12, 13]), relational databases are still a very popular method for organizing data in a fashion that supports fast and reliable manipulation and retrieval. Since databases offer features that are often complementary to big data solutions [14] and many individuals and organizations do not currently need to manage large-scale datasets [15], the relational database is a well-established and commonly used data management technology [1].

In order to store data in a relational database, a *schema* must be defined to specify how information will be structured into *tables* [1]. Each table has a number of *columns* that contain specific types of data. Data entries into a table are referred to as records or *rows*, where each row involves data elements for each column of the table. In addition, a number of *integrity constraints* may be specified over columns of each table. Designed to preserve the consistency of the database, integrity constraints place restrictions on data that the DBMS should allow into it.

Users interact with relational databases using SQL statements. A schema is specified with SQL statements submitted to a database management system (DBMS) such as PostgreSQL[1] or SQLite[2]. Figure 1 shows an example of a schema declaration involving two `CREATE TABLE` SQL statements, hereafter referred to as the *BrowserCookies* example, inspired by the SQLite database that Mozilla Firefox uses to manage cookies. It distills many aspects of schemas that are both observed in the real subjects used in our empirical study in Section 5 and necessary for understanding the techniques developed in this paper. These features include different column types, examples of the different types of integrity constraint, and the various ways in which those constraints have been declared.

The schema involves two tables: `cookies` and `places`. Each table declaration involves the definition of individual columns (lines 2–6 and 11–18, respectively), with associated data types. The `cookies` table stores information about each distinct cookie, including its name and value, represented as textual strings. Additional columns record the time (in integer format) when the cookie was created, last accessed, and when it expires. Each cookie also has an associated host and path, which links it to an entry in the `places` table.

Figure 2 depicts these tables conceptually, containing data as a result of various SQL `INSERT` statements submitted to the DBMS. While `INSERT` statements 1 and 3 were accepted by the DBMS, statements 2 and 4 were rejected, since they failed to satisfy the restrictions specified by integrity constraints declared on the schema.

[1] http://www.postgresql.org/
[2] http://sqlite.com/

```
1) INSERT INTO places(host, path, title, visit_count, fav_icon_url)                                  ✓
   VALUES('amazon.com', '/login.html', 'Amazon', 1, NULL);

2) INSERT INTO places(host, path, title, visit_count, fav_icon_url)                                  ✗
   VALUES('amazon.com', '/login.html', 'Amazon.co.uk', 10, 'fav.ico');

3) INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)            ✓
   VALUES(1, 'session_id', 'xyz' , NULL, 2000, 1000, 'amazon.com', '/login.html' );

4) INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)            ✗
   VALUES(2, 'text_size_pref', '2' , 3000, 1000, 2000, 'www.bbc.co.uk', '/' );
```

(a) `INSERT` SQL statements for tables of the *BrowserCookies* schema

host	path	title	visit_count	fav_icon_url
'amazon.com'	'/login.html'	'Amazon'	1	NULL

(b) Final state of the `places` table following issuing of the `INSERT` statements in part (a)

id	name	value	expiry	last_accessed	creation_time	host	path
1	'session_id'	'xyz'	NULL	2000	1000	'amazon.com'	'/login.html'

(c) Final state of the `cookies` table following issuing of the `INSERT` statements in part (a)

Figure 2: `INSERT` statements and final table data for the *BrowserCookies* schema of Figure 1

Integrity constraints help to define the structure of a database and work to stop invalid data values from being entered into database tables. `PRIMARY KEY` constraints define subsets of columns for which the sets of row values should be unique, so that specific rows can be easily retrieved later. Our example illustrates two different styles of primary key. The `cookies` table uses the integer `id` column as its primary key (declared on line 11 of Figure 1). This type of primary key is called a *surrogate* key, since a specially generated value is used in preference to actual application data (e.g., the column `name` coupled with `host` and `path`). In contrast, the `places` table uses a *natural* key that is the composite of two columns, `host` and `path` (declared on line 7) [16]. `INSERT` statements 1 and 3 in Figure 2 (a) both pass the primary key constraint that the elements on the columns for which they are specified are unique. Yet, statement 2 contains an identical pair of values for the primary key pair of the `places` table, resulting in its rejection by the DBMS. Additionally, several columns involve `NOT NULL` constraints, which ensure that values for a table column are never "`NULL`" — a special marker normally used to denote an unknown or non-existent value.

Columns that do not form primary keys, but for which row values must be unique (sometimes referred to as *candidate keys* [1]) may be declared using a `UNIQUE` constraint. In the `cookies` table the triple of values for the `name`, `host` and `path` columns must be unique, enforcing the rule that the database should not store a cookie with the same name originating from the same place; this is defined with the `UNIQUE` constraint declaration on line 19.

`FOREIGN KEY` constraints link the rows in one table to rows in some other referenced table. A foreign key relationship exists between the `cookies` table and the `places` table through the `host` and `path` columns, specified by the `FOREIGN KEY` constraint on line 20 of Figure 1. Figure 2 shows how this constraint is successfully enforced: the foreign key values for `host` and `url` for `INSERT` statement 3 match the values already inserted into the `places` table by `INSERT` statement 1. However, there is no matching pair of values in `places` for the values of `host` and `url` to be inserted into `cookies` by `INSERT` statement 4, thus resulting in the rejection of this statement.

Finally, `CHECK` constraints involve the declaration of arbitrary predicates over table data. In Figure 1, there are two `CHECK` constraints, declared on lines 21 and 22. Either `expiry` should be zero (signifying the cookie expires when the browser closes) or greater than `last_accessed`, which in turn should be greater than or equal to `creation_time`. The latter constraint is not true of the values in `INSERT` statement 4 of Figure 2, giving a secondary cause for the DBMS to reject it.

2.1.1 Testing the Specification of Integrity Constraints in Schemas

Despite the large body of work on the testing of programs, there has been relatively little work devoted to the testing of the artefacts related to the databases that often drive these programs. One neglected aspect in the testing of database-centric applications is that of the database schema, which is often implicitly assumed to be correct and yet frequently subject to extensive modification throughout the lifetime of an application [17]. However, mistakes made in the design and implementation of the schema can have a far reaching, and often costly, negative impact on the quality of the rest of a software application. Schema mistakes may, for example, require the need for regression changes to be made to program code and to the program's complex SQL queries that currently interact with the

database on the assumption of a correct schema having been designed first [18].

A database schema is often an application's last line of defense against data that compromises the integrity of the database's contents. Without a properly defined schema, an application may, for instance, incorrectly create two users with the same login ID or products with prices that are less than zero. Additionally, the correctness of the schema may be tied to the underlying DBMS. Often, programmers will use a different DBMS during the development and deployment phrases of an application. For instance, SQLite may be used in development, since it is fast and may be used "in-memory", whereas PostgreSQL may be used during deployment, since it is more suited to managing large data sets and handling concurrent accesses.

However, as a consequence of the many different interpretations of the SQL standard [11], a schema developed for one DBMS may vary in behavior when it is used with another. For example, SQLite allows primary key columns to be `NULL`, whereas for most other DBMSs (e.g., PostgreSQL), `NULL` is not allowed for primary keys, in accordance with the SQL standard. Even though most DBMSs do not constrain the appearance of `NULL` in columns in `UNIQUE` constraints, Microsoft SQL Server will only allow `NULL` to appear once, on the basis that the secondary `NULL` is not distinct from the existing `NULL` in a column. These are just two of the, potentially many, varied and nuanced cross-DBMS differences that may be missed as the schema is developed. It is therefore important to have a test suite asserting that the schema's behavior is the same after database deployment as it was during development.

In summary, the relational database schema is a complex software artefact whose correctness can be imperiled during creation, subsequent modification, or DBMS migration, thus degrading the quality of an entire application. As such, testing the database schema is an important activity that is advocated by industry practitioners [4]. Yet, there has not, hitherto, been a well-founded basis on which to conduct this testing. In response to the dearth of a foundation for systematic relational schema testing, this paper develops coverage criteria for the logic encoded by the integrity constraints in a database's schema, as introduced in the previous section. Our coverage criteria, as later defined in Section 3, extend prior work on *logic coverage*, which we introduce in the next subsection.

2.2 Predicates, Clauses and Logic Coverage

In the following content, we summarize Ammann and Offutt, who reviewed the prior work in the literature devoted to logic coverage and the relevant test criteria (e.g., [19, 20]), and clarified some confusion that has arisen amongst researchers [5]. As we define and compare these logic coverage criteria, we also employ the popular understanding of criteria subsumption dictating that a criterion C subsumes, or is stronger than, criterion C' if and only if every test suite that satisfies C also satisfies C' [21].

A *predicate* is defined as an expression that evaluates to a boolean value, for instance $a = b \vee x > y$. A predicate is a top-level structure that assembles a series of atomic boolean expressions, or *clauses*, through the logical connectives ($\neg, \wedge, \vee, \rightarrow, \oplus$, and $\leftrightarrow$). That is, $a = b$ and $x > y$ are clauses for $a = b \vee x > y$. The simplest type of test coverage for predicates is *Predicate Coverage*, which involves exercising the predicate as true and false. Alternatively, *Clause Coverage* works at the lower level of clauses, mandating that each one is independently exercised as true and false. Clause Coverage does not subsume Predicate Coverage: in general, ensuring each clause is evaluated as true and false does not guarantee the overall predicate will also have been evaluated with both true and false outcomes [5].

To address this, different types of coverage criteria have been developed that test both clauses and predicates. The perhaps simplest "brute force" coverage criterion of this type is *Combinatorial Coverage*, also known as *Multiple Condition Coverage*. By requiring that the clauses of a predicate are evaluated to each possible combination of truth values, Combinatorial Coverage subsumes both Predicate and Clause Coverage. However, even though testers normally like to concentrate on one particular clause of the predicate at a time, checking its contribution to the overall predicate, Combinatorial Coverage does not support this strategy. Another drawback of Combinatorial Coverage is the explosion of tests that it causes — which is 2^n tests in the worst case for n clauses.

Active Clause Coverage (ACC), a criterion designed to address these shortcomings, is "almost identical" [5] to the *Modified Condition Decision Coverage* criterion previously defined by Chilenski [19] and adopted in support of many safety-critical testing environments [20]. ACC requires the test cases for individual clauses to also influence the truth evaluation of the predicate, thereby exploiting an overlap of test requirements such that a predicate and all its clauses are exercised as true and false, but in a smaller number of test cases.

ACC takes each clause of a predicate in turn as the focus of a subset of test requirements. The current clause under consideration is referred to as the *major clause*, and the remaining clauses the *minor clauses*. Let p be the predicate that we wish to test, and $c_1 \ldots c_n$ the clauses it contains. The fundamental principle behind ACC is *determination*: the circumstances under which a particular clause has the authority to establish the truth value of the overall predicate. Determination ensures that the effect of a clause on a predicate is isolated for the purposes of testing, thus allowing a tester to check how different truth values for the clause induce different truth values for the predicate. More formally, given a major clause c_i in p, c_i *determines* p if the minor clauses $c_j \in p, j \neq i$ have

	$a = b$	$x > y$	$a = b \vee x > y$
1)	T	F	T
2)	F	F	F
3)	F	T	T
4)	F	F	F

Figure 3: Deriving Active Clause Coverage test requirements for the predicate $a = b \vee x > y$. Truth values for a major clause appear in shaded cells: $a = b$ is the major clause first, followed by $x > y$. Two test requirements are generated per clause, with three distinct requirements overall, since requirements 2 and 4 are identical.

truth values such that changing the value of c_i changes the truth value of p [5]. The truth value of c_i need not be the same as p, so long as p's truth value changes as c_i also changes. For example, with the predicate $a = b \vee x > y$, the clause $a = b$ determines p when $x > y$ is false.

ACC requires that each clause of a predicate is exercised as true and false, where the major clause is also responsible for determining the overall predicate. Figure 3 shows how test requirements are generated for $a = b \vee x > y$. With this predicate, the minor clause must be false for the major clause to determine the predicate. Taking $a = b$ as the major clause and $x > y$ as the minor clause results in two test requirements (1 and 2 in Figure 3) as $x > y$ is set as false and the truth value of $a = b$ is flipped from true to false. The same procedure but with $x > y$ as the major clause and $a = b$ as the minor clause results in two further requirements (3 and 4 in the figure). Since requirements 2 and 4 are identical, three distinct test requirements are generated overall.

As Ammann and Offutt discuss, there are three different instantiations of ACC that are possible in practice: *General Active Clause Coverage (GACC)*, *Correlated Active Clause Coverage (CACC)* and *Restricted Active Clause Coverage (RACC)* [5]. These differences are only actually observable for certain types of predicates (for $a = b \vee x > y$, all three types of ACC result in an identical set of test requirements). In this paper, we ignore GACC, because in general, it does not subsume Predicate Coverage since, unlike CACC and RACC, there is no explicit requirement for the top level predicate to have been evaluated as true and false. The difference between CACC and RACC centers on the treatment of the minor clauses. RACC demands that the truth values of minor clauses are identical as the truth value of the major clause is alternated. With CACC, there is no such restriction — minor clauses are free to change truth value, so long as the major clause still determines the overall predicate.

For many types of predicate ($a = b \vee x > y$ being a case in point) CACC and RACC produce identical test requirements. In these cases, minor clauses must be fixed while flipping the major clause, otherwise the major clause would cease to determine the overall predicate. However, in other instances, fixing minor clauses can lead to additional test requirements that are infeasible or hard to satisfy [5]. Moreover, in the context of relational database schemas, the distinction between CACC and RACC is rarely evident because a schema represents the logical conjunction of predicates. For these reasons, where a predicate entails a difference between CACC and RACC, the coverage criteria defined in this paper are based on the CACC variant of ACC.

3 Coverage Criteria for Integrity Constraints

Integrity constraints may be tested by attempting to insert a new row of data into a database table and checking whether the row was accepted into the database by the DBMS (i.e., the data satisfied the integrity constraints) or was rejected (i.e., the data did not conform to the integrity constraints). By verifying that acceptance and rejection of data was as expected, a tester can ascertain whether the integrity constraints are correctly specified in the schema. For instance, it could be that a `NOT NULL` constraint was omitted from a column definition, as evidenced by a `NULL` value being inserted into a table when it was supposed to be rejected. Or, a tester could observe that the same username was entered twice into a table for two different users, suggesting that the schema is missing a `UNIQUE` constraint.

The key challenge associated with the aforementioned testing strategy is systematically running the right `INSERT` statements so as to ensure that a large class of database schema faults can be reliably detected. To this end, the rest of this section details coverage criteria for the methodical testing of the integrity constraints in a relational database schema. Given a schema s, each criterion involves the production of a set of test requirements TR. In the next subsection, we first begin with the preliminaries for a model of relational databases.

3.1 Preliminaries

In the relational model, due to Codd [22], a database table is a *relation*, which is a set of *tuples* (i.e., table rows) with identical *labels* (i.e., column names). A tuple is written $r = (cl_1 : v_1, \ldots, cl_{ncl} : v_{ncl})$ for a relation with ncl labels, where $cl_{1\ldots ncl}$ are the labels and $v_{1\ldots ncl}$ are the values of the tuples corresponding to each label. In our formal definitions, we adopt the conventions of Maier [23], using the notation $r(cl)$ to obtain the value of a tuple r with the label cl, and the use of $\perp$ as a shorthand for `NULL`. In the interest of reducing potential confusion, we standardize by using the DBMS terms for their relational algebra equivalents, so in the following content we will use *table* when referring to a relation, *row* when referring to a tuple, and *column* when referring to a label.

3.2 Integrity Constraint Predicates

Integrity constraints may be formulated into predicates that evaluate to true when the values in a row are judged admissible into the database with respect to that particular constraint, and false when they do not.

Definition 1 (Integrity Constraint Predicate) *An* integrity constraint predicate *icp, for some integrity constraint ic, is a predicate that evaluates to true when data in some new row nr conforms to ic and false when it does not.*

As already discussed in Section 2.1.1, integrity constraint behavior can vary across different DBMSs. We handle this in our approach by defining *integrity constraint functions*, which formulate a predicate for an integrity constraint with a particular DBMS. Different DBMSs can be accommodated by simply using an alternative version of the function that is tailored for that database.

Figures 4 and 5 show definitions of functions for HyperSQL, PostgreSQL and SQLite, which are the three commonly used DBMSs that we focus on in our empirical study (a function is suitable for use with all three DBMSs, unless otherwise specified). In general, each function involves some constraint declared on a table *tbl* and evaluates the data for some new row *nr* to be inserted into *tbl*. Each function derives a predicate on the basis of a pair of conditions. The first is the *null condition*, which evaluates whether the data in *nr* is admissible to *tbl* on the basis of the `NULL` values that *nr* potentially contains. Different integrity constraints make allowances for `NULL` values in different ways. `FOREIGN KEY` and `UNIQUE` constraints will accept rows involving `NULL` for any columns in *nr* for which they are defined (regardless of values for other columns that make up the constraint). `PRIMARY KEY` constraints may reject `NULL` values for all primary key columns depending on the DBMS. We therefore define two types of functions for `PRIMARY KEY`s in Figure 5: one covering the case for HyperSQL and PostgreSQL, where `NULL` values are rejected for `PRIMARY KEY`s and an alternative version for SQLite, with which `NULL` values are accepted. Finally, `CHECK` constraints admit `NULL` values for columns that result in their expression evaluating to unknown.

The second condition is the *constraint condition*, which evaluates whether the data in *nr* conforms to the rationale of the constraint in question. `PRIMARY KEY` and `UNIQUE` constraint conditions evaluate to "true" when the values in *nr* are unique with respect to other rows in *tbl* for columns on which the constraint is defined. For `FOREIGN KEY` constraints, the values in *nr* for columns on which it is defined must match those in some row of the table referenced by the key. `CHECK` constraints check that some expression holds over the data in *nr*.

For `NOT NULL` constraints, there is no null condition — since the very purpose of the constraint is to reject row values that are `NULL` — so the integrity constraint predicate is a solitary constraint condition. Where the constraint accepts `NULL` values, the integrity constraint predicate is formed from the disjunction of the null condition and the constraint condition (e.g., `UNIQUE` constraints and `CHECK` constraints), whereas for constraints that reject `NULL` values the integrity constraint predicate is a conjunction of the null condition and the constraint condition (e.g., `PRIMARY KEY`s for non-SQLite DBMSs).

Figure 6 gives examples of concrete integrity constraint predicates for constraints declared for the `cookies` table of the *BrowserCookies* example given in Figure 1.

3.3 Acceptance Predicates

By forming a conjunction of the predicates for each integrity constraint involving a table, we can form a complete predicate that states whether the data in a new row *nr* should be accepted into the table or rejected. We refer to these predicates as *acceptance predicates*.

Definition 2 (Acceptance Predicate) *An* acceptance predicate *ap for a table tbl is a boolean predicate over values in a new row nr to be inserted into tbl that specifies when data in nr will be successfully admitted into a database for tbl. An acceptance predicate ap is a conjunction of integrity constraint predicates for the integrity constraints defined for tbl.*

function get_check_constraint_predicate($cc(tbl, exp), nr$)
where:
- $cc(tbl, exp)$ is a `CHECK` constraint
  - tbl is the table on which the `CHECK` constraint is defined
  - exp is the `CHECK` expression
- nr is a new row of data to be inserted into tbl

// null condition
let $nc_c \leftarrow (exp = unknown)$
// constraint condition
let $cc_c \leftarrow (exp = true)$
// integrity constraint predicate
let $ipc_c \leftarrow nc_c \vee cc_c$
return icp_c
end function

function get_foreign_key_constraint_predicate($fkc(tbl, CL, rtbl, RCL), nr$)
where
- $fkc(tbl, CL, rtbl, RCL)$ is a `FOREIGN KEY` constraint
  - tbl is the table on which the constraint is defined
  - $CL = cl_1...cl_{nclc}$ is the set of columns of the constraint
  - $rtbl$ is the table referenced in the foreign key
  - $RCL = rcl_1...rcl_{nclc}$ is the set of columns in the referenced table $rtbl$
- nr be a new row of data to be inserted into tbl

// null condition
let $nc_{fk} \leftarrow (nr(cl_1) = \bot \vee \ldots \vee nr(cl_{nclc}) = \bot)$
// constraint condition
let $cc_{fk} \leftarrow (\exists er \in rtbl : nr(cl_1) = er(rcl_1) \wedge \ldots \wedge nr(cl_{nclc}) = er(rcl_{nclc}))$
// integrity constraint predicate
let $ipc_{fk} \leftarrow nc_{fk} \vee cc_{fk}$
return icp_{fk}
end function

function get_not_null_constraint_predicate($nnc(tbl, cl), nr$)
where
- $nnc(tbl, cl)$ is a `NOT NULL` constraint
  - tbl is the table on which the constraint is defined
  - cl is the column of the constraint
- nr be a new row of data to be inserted into tbl

// constraint condition
let $cc_{nn} \leftarrow (nr(cl) \neq \bot)$
// integrity constraint predicate
let $icp_{nn} \leftarrow cc_{nn}$
return icp_{nn}
end function

function get_unique_constraint_predicate($uc(tbl, CL), nr$)
where
- $uc(tbl, CL)$ is a `UNIQUE` constraint
  - tbl is the table on which the constraint is defined
  - $CL = cl_1...cl_{nclc}$ is the set of columns of the constraint
- nr be a new row of data to be inserted into tbl

// null condition
let $nc_u \leftarrow (nr(cl_1) = \bot \vee \ldots \vee nr(cl_{nclc}) = \bot)$
// constraint condition
let $cc_u \leftarrow (\forall er \in tbl : nr(cl_1) \neq er(cl_1) \vee \ldots \vee nr(cl_{nclc}) \neq er(cl_{nclc}))$
// integrity constraint predicate
let $icp_u \leftarrow nc_u \vee cc_u$
return icp_u
end function

Figure 4: Functions for obtaining integrity constraint predicates.

function get_primary_key_constraint_predicate($pkc(tbl, CL), nr$)
where
- $pkc(tbl, CL), nr$ is a `PRIMARY KEY` constraint
  - tbl is the table on which the constraint is defined
  - $CL = cl_1...cl_n$ is the set of columns of the constraint
- nr be a new row of data to be inserted into tbl

// null condition
let $nc_{pk} \leftarrow (nr(cl_1) \neq \bot \wedge \ldots \wedge nr(cl_n) \neq \bot)$
// constraint condition
let $cc_{pk} \leftarrow (\forall er \in tbl : nr(cl_1) \neq er(cl_1) \vee \ldots \vee nr(cl_n) \neq er(cl_n))$
// integrity constraint predicate
let $icp_{pk} \leftarrow nc_{pk} \wedge cc_{pk}$
return icp_{pk}
end function

function get_primary_key_constraint_predicate_for_SQLite ($pkc(tbl, CL), nr$)
where
- $pkc(tbl, CL)$ is a `PRIMARY KEY` constraint for SQLite
  - tbl is the table on which the constraint is defined
  - $CL = cl_1...cl_n$ is the set of columns of the constraint
- nr be a new row of data to be inserted into tbl

// null condition
let $nc_{pk} \leftarrow (nr(cl_1) = \bot \vee \ldots \vee nr(cl_n) = \bot)$
// constraint condition
let $cc_{pk} \leftarrow (\forall er \in tbl : nr(cl_1) \neq er(cl_1) \vee \ldots \vee nr(cl_n) \neq er(cl_n))$
// integrity constraint predicate
let $icp_{pk} \leftarrow nc_{pk} \vee cc_{pk}$
return icp_{pk}
end function

Figure 5: Functions for obtaining `PRIMARY KEY` integrity constraint predicates. The first function applies to the PostgreSQL and HyperSQL DBMSs, the secondary function applies to the specific behavior of SQLite.

`PRIMARY KEY` constraint on line 11:
$nc_1 \leftarrow (nr(\texttt{id}) \neq \bot)$
$cc_1 \leftarrow (\forall er \in \texttt{cookies} : nr(\texttt{id}) \neq er(\texttt{id}))$
$icp_1 \leftarrow nc_1 \wedge cc_1$

`NOT NULL` constraint on line 11:
$cc_2 \leftarrow (nr(\texttt{id}) \neq \bot)$
$icp_2 \leftarrow cc_2$

`NOT NULL` constraint on line 12:
$cc_3 \leftarrow (nr(\texttt{name}) \neq \bot)$
$icp_3 \leftarrow cc_3$

`UNIQUE` constraint on line 19:
$nc_4 \leftarrow (nr(\texttt{name}) = \bot \vee nr(\texttt{host}) = \bot \vee nr(\texttt{path}) = \bot)$
$cc_4 \leftarrow (\forall er \in \texttt{cookies} : nr(\texttt{name}) \neq er(\texttt{name}) \vee nr(\texttt{host}) \neq er(\texttt{host}) \vee nr(\texttt{path}) \neq er(\texttt{path}))$
$icp_4 \leftarrow nc_4 \vee cc_4$

`FOREIGN KEY` constraint on line 20:
$nc_5 \leftarrow (nr(\texttt{host}) = \bot \vee nr(\texttt{path}) = \bot)$
$cc_5 \leftarrow (\exists er \in \texttt{places} : nr(\texttt{host}) = er(\texttt{host}) \wedge nr(\texttt{path}) = er(\texttt{path}))$
$icp_5 \leftarrow nc_5 \vee cc_5$

`CHECK` constraint on line 21:
$nc_6 \leftarrow ((nr(\texttt{expiry}) = 0 \vee nr(\texttt{expiry}) > nr(\texttt{last\_accessed})) = unknown)$
$cc_6 \leftarrow ((nr(\texttt{expiry}) = 0 \vee nr(\texttt{expiry}) > nr(\texttt{last\_accessed})) = true)$
$icp_6 \leftarrow nc_6 \vee cc_6$

`CHECK` constraint on line 22:
$nc_7 \leftarrow ((nr(\texttt{last\_accessed}) \geq nr(\texttt{creation\_time})) = unknown)$
$cc_7 \leftarrow ((nr(\texttt{last\_accessed}) \geq nr(\texttt{creation\_time})) = true)$
$icp_7 \leftarrow nc_7 \vee cc_7$

Figure 6: Integrity constraint predicates for the `cookies` table of Figure 1, for some new row nr to be inserted into the table. The `PRIMARY KEY` constraint predicate was formed using Figure 5's '"get_primary_key_constraint_predicate" function for the PostgreSQL/HyperSQL DBMSs.

3.4 Minimality of Integrity Constraint Declarations

Schemas may involve the declaration of more constraints than necessary to restrict the types of data that may be accepted into a database table. For instance, the same integrity constraint may be mistakenly specified twice — especially given the redundant nature of some SQL features [11]. More subtle types of redundancy are also possible, depending on the DBMS. One example of this is the declaration of `NOT NULL` constraints on `PRIMARY KEY` columns for HyperSQL and PostgreSQL. Since for these DBMSs, a `PRIMARY KEY` column value must implicitly not be `NULL`, the inclusion of further `NOT NULL` constraints on columns that are also a part of a primary key is not necessary. We refer to a schema as *IC-Minimal* when it does not involve extraneous integrity constraints for a particular DBMS. We define IC-Minimality in terms of acceptance predicates.

Definition 3 (IC-Minimality) *For each table tbl of a schema s, form an acceptance predicate ap from the integrity constraint predicates for constraints declared on s involving tbl, ensuring that ap is in conjunctive normal form. We say that s is* IC-Minimal *for a DBMS when each ap for each tbl does not contain any duplicated conjuncts.*

For example, a fragment of the acceptance predicate for the `cookies` table is

$$icp_1 \wedge icp_2 \wedge \ldots$$

That is, the conjunction of icp_1, the integrity constraint predicate for the `PRIMARY KEY` of the `cookies` table, and icp_2, the predicate for the `NOT NULL` constraint declared on the `id` field, and so on. Each conjunct can be expanded into full predicates, which are given in Figure 6:

$$nr(\texttt{id}) \neq \bot \wedge \forall er \in \texttt{cookies} : nr(\texttt{id}) \neq er(\texttt{id}) \wedge nr(\texttt{id}) \neq \bot \wedge \ldots$$

Note that the third conjunct is a repetition of the first. That is, the constraint condition cc_2 of icf_2 is a repetition of the null condition nc_1 of icf_1. In order for the *BrowserCookies* example to be IC-Minimal, the `NOT NULL` constraint on the `id` field must be removed. A simple algorithm for deriving an IC-Minimal schema from a non-IC-Minimal schema is as follows: take each integrity constraint in turn, remove it, and observe its effect on the form of each *ap*. If any originally non-duplicated conjuncts disappear from tbl_p, reinstate the constraint, else permanently remove it.

It is important to note that IC-Minimality is dependent on the DBMS that hosts the relational schema. In the preceding example, the primary key predicate was formed using the "get_primary_key_predicate" function of Figure 5 for the PostgreSQL/HyperSQL DBMS. Yet, for SQLite, primary key columns may be `NULL`: so for that DBMS, no conjuncts are duplicated in the acceptance predicate, and thus no constraints need to be removed since the schema is already IC-Minimal for that particular DBMS.

3.5 Existing Sufficient Data for Testing

Figures 4–6 show how the evaluation of certain integrity constraints depends on *existing* rows of data in the database. For instance, a `UNIQUE` constraint can never be violated unless there is already data in the database. A test involving a `FOREIGN KEY` constraint is trivial unless there are already keys in the referenced table. In order to ensure the effectiveness of a test, a database of the schema under test needs to be in some initial state that prevents tests from being infeasible or trivial. We refer to this state as being *T-Sufficient*:

Definition 4 (T-Sufficiency) *The contents of some database d for some schema under test s is said to be* T-Sufficient *with respect to some test requirement $tr \in TR$ iff: 1) tr cannot be trivially satisfied by the insertion of an arbitrary row of data into d; and 2) The contents of d do not render tr infeasible.*

For example, the data in the table for Figure 2 (c) is T-Sufficient for testing satisfaction and violation of the `UNIQUE` constraint in the `cookies` table. From a satisfaction point of view, any new row entered into `cookies` must have a unique triple of values for `name`, `host` and `path`, while from a violation point of view, a new row must have the same triple. With an empty database, any row could be entered to satisfy the constraint, while violation would have been infeasible. In the definition for each of the coverage criterion in the following subsections, we assume a T-Sufficient database state for each $tr \in TR$.

```
1) INSERT INTO places(host, path, title, visit_count, fav_icon_url)                                  ✗
   VALUES('amazon.com', '/login.html', '', 0, '');
2) INSERT INTO places(host, path, title, visit_count, fav_icon_url)                                  ✓
   VALUES('www.bbc.co.uk', '/', '', 0, '');
3) INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)            ✗
   VALUES(2, 'last_search', '3', 0, 2, 1, 'www.google.com', '/')
4) INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)            ✓
   VALUES(2, 'textsizepref', '3', 0, 2, 1, 'www.bbc.co.uk', '/')
```

Figure 7: Example `INSERT`s for obtaining full Acceptance Predicate Coverage of the *BrowserCookies* schema, using the database state of Figure 2. `INSERT` statements 1 and 2 are for the `places` table: statement 1 is rejected by the DBMS (as denoted with a cross symbol) because it uses identical primary key values to the row already in the table, while statement 2 is accepted by the DBMS (as denoted by a tick symbol). Statements 3 and 4 are for the `cookies` table: statement 3 is rejected by the DBMS due to a non-existent foreign key reference (the values for `host` and `path` do not match the pair of values in the row for the `places` table). Finally, statement 4 is accepted by the DBMS.

3.6 Simple Coverage Criteria

The very minimum for testing the integrity constraints of a schema is that we attempt to insert data into tables and test for successful acceptance and rejection of that data. We formulate this notion into a coverage criterion called *Acceptance Predicate Coverage (APC)*, based on the concept of acceptance predicates.

Criterion 1 (Acceptance Predicate Coverage (APC)) *For each table tbl of the schema-under-test s, let ap be the acceptance predicate, and add two test requirements to TR: one where ap evaluates to true, one where ap evaluates to false.*

APC results in $2 \times |TBL|$ test requirements, where TBL is the set of tables in the schema-under-test s. Assuming the database state of Figure 2, the four `INSERT` statements of Figure 7 fulfill the test requirements of APC.

APC, however, does not ensure that each individual integrity constraint is exercised. As seen from the preceding examples, rejection of an `INSERT` is due to the violation of one particular integrity constraint. To this end, we define *Integrity Constraint Coverage (ICC)*:

Criterion 2 (Integrity Constraint Coverage (ICC)) *For each integrity constraint ic of s, two test requirements are added to TR, one where the integrity constraint predicate icp for ic evaluates to true, and one where it evaluates to false.*

ICC ensures that for each integrity constraint, there is at least one test case where the `INSERT` statement conforms to the constraint, and one that causes the constraint to be violated. This is achieved by ensuring an evaluation of the constraint's condition as true and false, and resulting in an upper bound of $2 \times |IC|$ test cases for a schema, where IC is the set of integrity constraints defined for the schema.

Note that ICC does not subsume APC, because ICC can be satisfied for each integrity constraint involving some table *tbl* without evaluating *tbl*'s acceptance predicate as true. For a test requirement involving conformance of `INSERT` statement data for an integrity constraint, it is not mandated that the data also conforms to all other constraints defined for the table, such that the data is ultimately accepted into the database and the table's acceptance predicate is true. This means that it is hard for the tester to isolate testing of a particular constraint and reason about its potential conformance or violation — if the data in some new row *nr* does not conform to the remaining integrity constraints, *nr* could be rejected for a number of reasons unrelated to the current constraint of interest. We therefore define further criteria, inspired by Active Clause Coverage, as introduced in Section 2, that aim to isolate the testing of each individual integrity constraint.

3.7 Active Coverage Criteria

Active Integrity Constraint Coverage (AICC) takes an acceptance predicate for a table and produces test requirements by considering each integrity constraint predicate in turn as the *major integrity constraint predicate* and the remaining integrity constraint predicates as the *minor predicates*, such that the major integrity constraint predicate *determines* the top-level acceptance predicate. The major predicate is manipulated such that its truth value is changed from true to false. Since acceptance predicates are conjunctions, minor integrity constraint predicates need

	PRIMARY KEY (11) id	NOT NULL (12) name	UNIQUE (19) name, host path	FOREIGN KEY (20) host, path last_accessed	CHECK (21) expiry, creation_time	CHECK (22) last_accessed,	
	icp_1	icp_3	icp_4	icp_5	icp_6	icp_7	ap
1)	T	T	T	T	T	T	T
2)	F	T	T	T	T	T	F
3)	T	T	T	T	T	T	T
4)	T	F	T	T	T	T	F
5)	T	T	T	T	T	T	T
6)	T	T	F	T	T	T	F
7)	T	T	T	T	T	T	T
8)	T	T	T	F	T	T	F
9)	T	T	T	T	T	T	T
10)	T	T	T	T	F	T	F
11)	T	T	T	T	T	T	T
12)	T	T	T	T	T	F	F

(a) Test requirements. Each column corresponds to an integrity constraint predicate, defined in Figure 6, with the acceptance predicate *ap* forming the right-most column. Above each integrity constraint predicate are details of the original constraint — its type, line number in Figure 1 in brackets, and affected columns. Each row forms a test requirement, showing the required truth values of each integrity constraint predicate. Shaded cells indicate an integrity constraint predicate is a major predicate for that particular test requirement. Requirements 3, 5, 7, 9 and 11 are duplicates of requirement 1, leaving seven distinct test requirements overall.

```
INSERT INTO cookies(
id,      name,     value, expiry, last_accessed, creation_time,      host,           path
) VALUES (
1)  2,   'session_id', NULL,  NULL,      NULL,         NULL,          NULL,          NULL          ✓
2)  1,   'session_id', NULL,  NULL,      NULL,         NULL,          NULL,          NULL          ✗
4)  2,      NULL,      NULL,  NULL,      NULL,         NULL,          NULL,          NULL          ✗
6)  2,   'session_id', NULL,  NULL,      NULL,         NULL,      'amazon.com',  '/login.html'     ✗
8)  2,   'session_id', NULL,  NULL,      NULL,         NULL,     'facebook.com', '/login.html'     ✗
10) 2,   'session_id', NULL,    5,        10,          NULL,          NULL,          NULL          ✗
12) 2,   'session_id', NULL,  NULL,       10,           20,           NULL,          NULL          ✗
);
```

(b) Example values for `INSERT` statements for each of the non-duplicated test requirements, assuming the database is reset to the state of Figure 2 before each `INSERT` for T-Sufficiency. The right-most column shows whether the `INSERT` statement is expected to be accepted (tick mark) or rejected (cross mark) by the DBMS (correlating with the truth value of *ap* in part (a)).

Figure 8: AICC test requirements and example test `INSERT` statements for the `cookies` table of Figure 1. (We assume the use of HyperSQL/PostgreSQL, so icp_2 is ignored so that the table is IC-Minimal.)

to evaluate to true so that the major integrity constraint predicate determines the top-level acceptance predicate. That is, the effect of the integrity constraint is isolated with respect to the acceptance predicate.

Criterion 3 (Active Integrity Constraint Coverage (AICC)) *For each table tbl of s, let ap be the acceptance predicate and ICP the integrity constraint predicates involved in ap. Take each integrity constraint predicate $icp_i \in ICP$ as the* major integrity constraint predicate*, ensuring that each remaining* minor integrity constraint predicate *$icp_j \in ICP, j \neq i$ evaluates to true, so that icp_i determines ap. TR contains the following requirements: icp_i evaluates to true, and icp_i evaluates to false, causing ap to also evaluate as true and false, respectively.*

Recall the concept of IC-Minimality from Section 3.4. Non-IC-Minimal schemas result in infeasible test requirements with AICC. For instance, with the `cookies` table, it impossible to evaluate the `NOT NULL` integrity constraint predicate icp_2 to false — and thus requiring `id` to be `NULL` — while simultaneously evaluating the `PRIMARY KEY` constraint icp_1 to true, this mandating a non-`NULL` value for `id`. Similarly, for a pair of duplicated constraints, one of the constraints cannot be false while the other, identical, constraint is true.

AICC subsumes both APC and ICC for IC-Minimal schemas: As part of the AICC criterion, *ap* for each table takes on true and false evaluations, and each integrity constraint predicate is also evaluated as true and false. For non-IC-Minimal schemas, AICC only "weakly" subsumes APC and ICC, since although the set of test requirements for AICC is essentially a superset of those for APC and ICC, some of the requirements for AICC can be shown to be infeasible — for which the equivalent requirements with APC and ICC will not. Therefore, since it is generally

undecidable if a test requirement is infeasible [5, 21], we contend that it is preferable to establish an IC-Minimal version of the schema before testing it, a practice that we follow for the experiments of Section 5.

In Figure 8 (a), we demonstrate the generation of the test requirements for the IC-Minimal version of the `cookies` table of Figure 1 (i.e., with the `NOT NULL` constraint on the `id` column, icf_2, removed). Twelve test requirements are generated, yet five requirements are duplicates of the first requirement, where *ap* is satisfied and an `INSERT` is made successfully into the table; this leaves seven distinct requirements. In part (b) of Figure 8, we show example values that might be used in the `INSERT` statements that form the test cases for satisfying each requirement.

AICC is based on inducing one of two truth values for each integrity constraint predicate, and as such, the tests may not fully exercise the null and constraint conditions embedded within them. This potentially leads to a superficial test suite. For example, the `PRIMARY KEY` constraint is never exercised with a `NULL` value by the concrete tests of Figure 8, leaving untested the scenario when the null condition is false. In many cases, the true evaluation of an integrity constraint predicate is delivered through the selection of `NULL` values, (i.e., exercising the null condition as true), but not testing satisfaction of the constraint condition. As such, for instance, the expression of the `CHECK` constraint `expiry = 0 OR expiry > last_accessed` is never actually exercised as true by any of the test cases.

The next coverage criterion aims to address this deficiency, by mandating that null and constraint conditions are fully exercised with both true and false evaluations.

Criterion 4 (Condition-Based Active Integrity Constraint Coverage (CondAICC)) *For each table tbl of s, let ap be the acceptance predicate and ICP the integrity constraint predicates involved in ap. Take each integrity constraint predicate $icp_i \in ICP$ as the* major integrity constraint predicate*, ensuring that each remaining* minor integrity constraint predicate *$icp_j \in ICP, j \neq i$ evaluates to true, so that icp_i determines ap.*

For each icp_i, the null condition and constraint condition take turns to become the major condition *$cond_{maj}$ with the remaining condition $cond_{min}$ evaluating to a truth value such that the $cond_{maj}$ determines icp_i. TR contains the following requirements: $cond_{maj}$ evaluates to true, and $cond_{maj}$ evaluates to false. As the truth value of $cond_{maj}$ flips, the truth values of icp_i and ap also flip.*

CondAICC begins in the same way as AICC, isolating the effect of the integrity constraint predicate on the acceptance predicate. It then goes a step further, isolating and testing the consequences of changing the truth value of the null condition and the constraint condition embedded in the major integrity constraint predicate.

Figure 9 shows the test requirements that would be created for the IC-Minimal version of the `cookies` table. The table shows that, during the derivation of the test requirements, constraint conditions can potentially evaluate to unknown, due to the presence of `NULL`s. The null condition always ensures, however, that the overall integrity constraint predicate, and consequently the acceptance predicate as well, can only ever be two-valued.

Since integrity constraint predicates for `PRIMARY KEY`s are conjunctions, the constraint condition must be "not-false" (i.e., true or unknown) when the null condition is the major condition, as shown in requirements 1 and 2. In practice the choice of truth values is fixed. When the null condition is true, such that no columns are `NULL` and thus the constraint condition cannot be unknown, then it can only be true in order for the null condition to determine the integrity constraint predicate. Conversely, when the null condition is false, columns must be `NULL`, so the constraint condition necessarily evaluates to unknown. When the constraint condition is the major condition, the null condition must always evaluate to true for the constraint condition to determine the integrity constraint predicate, as evident in requirements 3 and 4. `NOT NULL` constraints have no null condition, and thus the derivation of requirements here is identical to AICC.

For all other constraints, the integrity constraint predicate is a disjunction of the null condition and the constraint condition. Thus the constraint condition must be "not-true" (i.e., false or unknown) for the null condition, as the major condition, to determine the integrity constraint predicate. Again, the truth value of the null condition necessarily decides the truth value of the constraint condition. When the null condition is true, columns are `NULL`, so the constraint condition must be unknown. When the null condition is false, the constraint condition must also be false. When the constraint condition is the major condition, the null condition must evaluate to false. While 21 requirements are produced, there are duplicates: Requirements 3 and 5 are duplicates of requirement 1. Requirement 10 is a duplicate of 8, 14 is a duplicate of 12, 18 a duplicate of 16, and 22 a duplicate of 20. Following the removal of duplicates there are 16 test requirements overall.

CondAICC subsumes AICC, since, for each integrity constraint *ic*, the process of ensuring each condition evaluates to true and false causes the integrity constraint predicate to evaluate to true and false.

Null and constraint conditions are themselves made up of individual clauses. For example, the first `CHECK` constraint for the `cookies` table (line 12 of Figure 1), $nr(\texttt{expiry}) = 0 \vee nr(\texttt{expiry}) > nr(\texttt{last\_accessed})$, is made up of two disjuncts. While CondAICC involves more stringent test requirements than AICC, it considers only truth evaluations of overall conditions, rather than their individual clauses. The following coverage criterion, *Clause-Based Active Integrity Constraint Coverage (ClauseAICC)*, expands testing to the clauses of each condition.

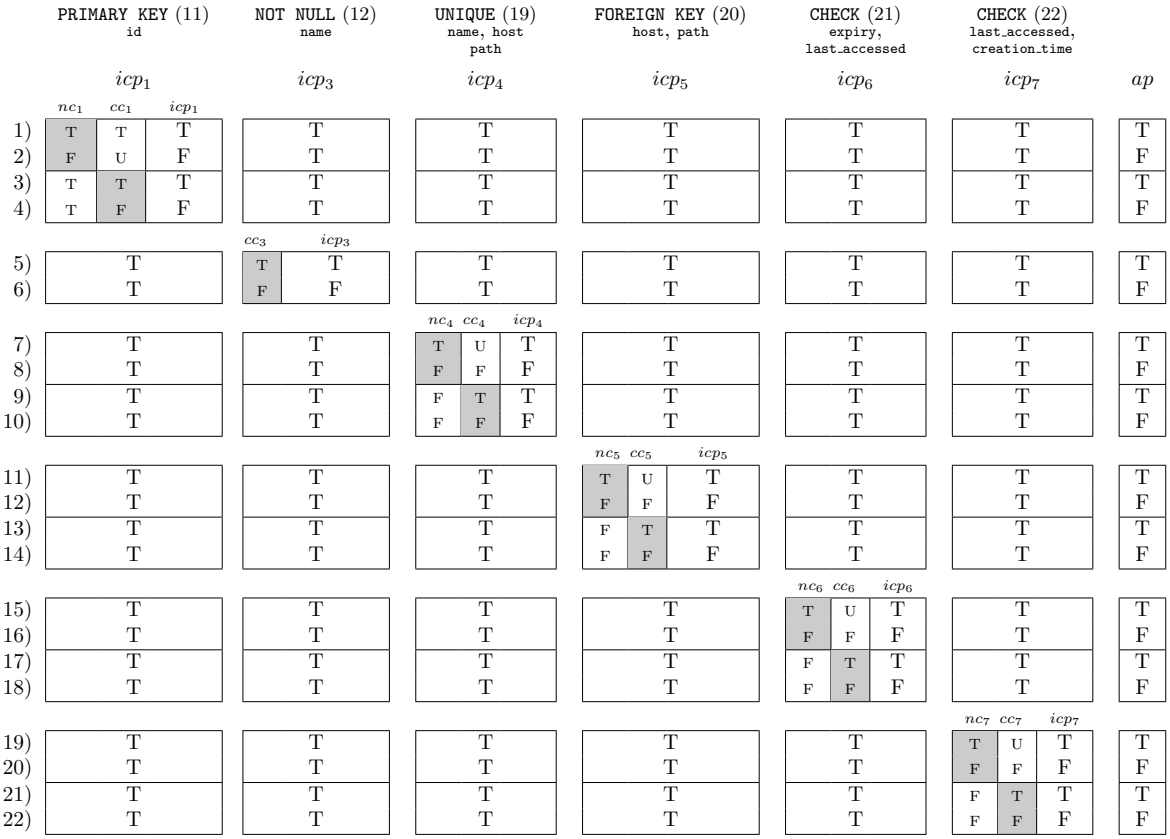


Figure 9: CondAICC test requirements for the `cookies` table of Figure 1. Each column corresponds to an integrity constraint predicate, defined in Figure 6, with the acceptance predicate *ap* forming the right-most column. Above each integrity constraint predicate are details of the original constraint — its type, line number in Figure 1 in brackets, and affected columns. Each row forms a test requirement, showing the required truth values for each integrity constraint predicate. Shaded cells indicate the major predicate and condition for that particular test requirement. (We assume the use of HyperSQL/PostgreSQL, so icp_2 is ignored and thus the table is IC-Minimal.) Requirements 3, 5, 10, 14, 18 and 22 are duplicates, resulting in 16 distinct test requirements overall.

Criterion 5 (Clause-Based Active Integrity Constraint Coverage (ClauseAICC)) *For each table tbl of s, let ap be the acceptance predicate and ICP the integrity constraint predicates involved in ap. Take each integrity constraint predicate* $icf_i \in ICF$ *as the* major integrity constraint predicate*, ensuring that each remaining* minor integrity constraint predicate $icp_j \in ICP, j \neq i$ *evaluates to true, so that* icp_i *determines ap.*

Let C be the set of atomic clauses of icp_i*, that is, the subexpressions of* icp_i *joined through the logical connectives* $\wedge$ *and* $\vee$*. Take each* $c_k \in C$ *as the* major clause *and ensure truth values for each remaining* minor clause $c_l \in C$ *such that* c_k *determines ip. TR contains requirements such that each major clause* c_k *evaluates to true and false. As* c_k *changes truth value from true to false,* icp_i *also changes truth value along with ap.*

Similar to the simpler active coverage criteria defined in this section, ClauseAICC first ensures an integrity constraint predicate determines the overall acceptance predicate. The secondary step then involves taking the major integrity constraint predicate and making each of its clauses the focus of an individual test. Using concrete examples, we now describe in detail how this process works for each type of constraint.

`UNIQUE` constraints. The process of generating test requirements for `UNIQUE` constraints has the effect of producing tests that explicitly check (1) what happens when each individual column is `NULL`, and (2) what happens when each column is individually unique. The second aspect effectively tests the inclusion of each individual column in the constraint, and for potential errors when forming the constraint from several columns.

As an example of how test requirements are derived, take the `UNIQUE` constraint on line 19 of Figure 1, for which the integrity constraint predicate is:

$$nr(\texttt{name}) = \bot \vee nr(\texttt{host}) = \bot \vee nr(\texttt{path}) = \bot \vee$$

$$\forall er \in \texttt{cookies} : nr(\texttt{name}) \neq er(\texttt{name}) \vee nr(\texttt{host}) \neq er(\texttt{host}) \vee nr(\texttt{path}) \neq er(\texttt{path})$$

Figure 10 (a) shows the test requirements that would be generated. Due to the potential involvement of `NULL` values, clauses of the constraint condition are three-valued and may evaluate to unknown. Since the intermediate predicate is a disjunction, minor clauses need to be "not-true", that is false or unknown, for the major clause to determine the predicate. In practice, there is no choice with respect to constraint condition truth values, which are necessarily fixed as false or unknown by the earlier null condition clauses.

Test requirements 1, 3 and 5 ensure that each column of the constraint is independently tested with `NULL`. For test requirements 7, 9 and 11, each column takes the turn of being unique with respect to values already in `cookies`, whereas values for the other columns are non-unique. This helps the tester make individual decisions about whether the column should be included in a `UNIQUE` constraint. If the value should not be capable of making a row uniquely-identifiable on its own, it should not be part of the constraint. Test 2 ensures that the testing of the overall acceptance predicate will set it to false, checking what happens when the key values clash with values already present in a row of the database table.

`PRIMARY KEY` constraints. In a similar fashion to `UNIQUE` constraints, ClauseAICC for primary keys tests what happens when each column involved is `NULL`, and what happens when each column is individually unique. For primary keys, however, `NULL` should be rejected. The `PRIMARY KEY` defined on the `cookies` table is made up of a single column. For single column `PRIMARY KEY` constraints, `UNIQUE` constraints and `FOREIGN KEY` constraints the test requirements created for ClauseAICC are identical to CondAICC. For instance, with the `cookies` table, the primary key has the integrity constraint predicate $nr(\texttt{id}) \neq \bot \wedge (\forall er \in \texttt{cookies} : nr(\texttt{id}) \neq er(\texttt{id}))$, or, in other words, a conjunction of the null condition and the constraint condition. So that the major clause determines the predicate, minor clauses need to be "not-true" (i.e., false or unknown). Because each condition is made up of a single clause, in following the ClauseAICC procedure, we effectively end up with the same tests as with CondAICC.

`FOREIGN KEY` constraints. ClauseAICC for foreign keys has the effect of testing 1) each column involved in the `FOREIGN KEY` as `NULL`, and 2) for its correct inclusion in the key. Following the `cookies` example, the integrity constraint predicate of the `FOREIGN KEY` is:

$$(nr(\texttt{host}) = \bot \vee nr(\texttt{path}) = \bot) \vee (\exists er \in \texttt{places} : nr(\texttt{host}) = er(\texttt{host}) \wedge nr(\texttt{path}) = er(\texttt{path}))$$

Figure 10 (b) shows the derivation of the test requirements. Since it is a disjunction composed of a further disjunction (the null condition) and a conjunction (the constraint condition), the predicate has a more complex structure than those previously considered. The two disjuncts at the top level need to be "not-true" for the lower level major clauses to determine the predicate. In the null condition disjunct, minor clauses then need to be not-true. In the constraint condition conjunct, at most one minor clause can be true, the other (or both) not-true.

In test requirements 1-4, one of the null condition clauses is the major clause. Since tests 2 and 4 are subsumed by later test requirements, the process leaves tests 1 and 3, which test that the `host` and `path` columns are individually `NULL`. In tests 5–8, the constraint condition clauses take the turn of being the major clause. Clause 7 is a duplicate of 5, so the process leaves three distinct tests, in which the foreign key column match a row in the referenced table (requirement 5), `host` does not match but `path` does (requirement 6) and `host` matches but `path` does not. The latter two tests check for correct inclusion of the foreign key columns in the key.

`CHECK` constraints. `CHECK` constraints with multiple clauses lead to further test requirements with ClauseAICC. The integrity constraint predicate for the `CHECK` constraint on line 21 in Figure 1 is as follows:

$$\begin{array}{c}((nr(\texttt{expiry}) = 0 \vee nr(\texttt{expiry}) > nr(\texttt{last\_accessed})) = unknown) \vee \\ ((nr(\texttt{expiry}) = 0 \vee nr(\texttt{expiry}) > nr(\texttt{last\_accessed})) = true)\end{array}$$

Essentially, an unknown or true evaluation of the `CHECK` expression leads to the constraint being satisfied. In the previously stated predicate, the evaluation of the two disjuncts to a specific truth value effectively converts clauses in three-valued logic to a two-valued logic.

Figure 10 (c) shows the derivation of test requirements according to ClauseAICC, in which each of the two original `CHECK` clause takes the turn of being unknown, true or false. One test requirement (number 1), however is infeasible. For $nr(\texttt{expiry}) = 0$ to evaluate to unknown, `expiry` must be `NULL`, which automatically results in $nr(\texttt{expiry}) > nr(\texttt{last\_accessed})$ evaluating to unknown rather than false, as required.

`NOT NULL` constraints. Test requirements with `NOT NULL` constraints are identical for ClauseAICC and CondAICC, since `NOT NULL` constraint expressions can only consist of one clause.

	$nr(\texttt{name}) = \bot$	$nr(\texttt{host}) = \bot$	$nr(\texttt{path}) = \bot$	$nr(\texttt{name}) \neq er(\texttt{name})$	$nr(\texttt{host}) \neq er(\texttt{host})$	$nr(\texttt{path}) \neq er(\texttt{path})$	icf_4
1)	T	F	F	U	F	F	T
2)	F	F	F	F	F	F	F
3)	F	T	F	F	U	F	T
4)	F	F	F	F	F	F	F
5)	F	F	T	F	F	U	T
6)	F	F	F	F	F	F	F
7)	F	F	F	T	F	F	T
8)	F	F	F	F	F	F	F
9)	F	F	F	F	T	F	T
10)	F	F	F	F	F	F	F
11)	F	F	F	F	F	T	T
12)	F	F	F	F	F	F	F

(a) Test requirements for the `UNIQUE` constraint. Requirements no.s 4, 6, 8, 10 and 12 are duplicates of requirement 2, leaving seven distinct requirements in total for this integrity constraint.

	$nr(\texttt{host}) = \bot$	$nr(\texttt{path}) = \bot$	$nr(\texttt{host}) = er(\texttt{host})$	$nr(\texttt{path}) = er(\texttt{path})$	icf_5
1)	T	F	U	F	T
2)	F	F	*At most one clause is* T		F
3)	F	T	F	U	T
4)	F	F	*At most one clause is* T		F
5)	F	F	T	T	T
6)	F	F	F	T	F
7)	F	F	T	T	T
8)	F	F	T	F	F

(b) Test requirements for the `FOREIGN KEY` constraint. Requirement 7 is a duplicate of 5, while requirements 2 and 4 are subsumed by both requirements 6 and 8, leaving five distinct requirements in total for this integrity constraint function.

	$(nr(\texttt{expiry}) = 0) = unknown$	$nr(\texttt{expiry}) > nr(\texttt{last\_accessed}) = unknown$	$nr(\texttt{expiry}) = 0 = true$	$nr(\texttt{expiry}) > nr(\texttt{last\_accessed}) = true$	icf_6
1)	T	F	F	F	*Inf.*
2)	F	F	F	F	F
3)	F	T	F	F	T
4)	F	F	F	F	F
5)	F	F	T	F	T
6)	F	F	F	F	F
7)	F	F	F	T	T
8)	F	F	F	F	F

(c) Test requirements for the `CHECK` constraint. Requirements 4, 6 and 8 are duplicates of requirement 2. Requirement 1 is infeasible, leaving four distinct requirements in total for this integrity constraint function.

Figure 10: Dissecting test requirements for ClauseAICC for the `cookies` table

3.8 Column Coverage Criteria

The previously-described "Constraint Coverage" criteria test the logic of existing integrity constraints specified in the schema. They do not test, however, for integrity constraints that may have been omitted from the schema definition: For instance, a "usernames" column not being declared `UNIQUE` or a surname column not being declared as `NOT NULL`. The following criteria set out to test for such missing constraints by testing each column of each table of the schema. *Unique Column Coverage (UCC)* tests each column with unique and non-unique values, while *Null Column Coverage (NCC)* tests columns with `NULL` and not-`NULL` values.

Criterion 6 (Unique Column Coverage (UCC)) *For each table tbl of a schema s, let CL be the set of columns. Let nr be a new row to be inserted into tbl. For each $cl \in CL$, let $ucl \leftarrow \forall er \in tbl : nr(cl) \neq er(cl)$. TR contains two requirements for each cl, one in which $ucl = true \wedge nr(cl) \neq \perp$, and one where $ucl = false \wedge nr(cl) \neq \perp$.*

Criterion 7 (Null Column Coverage (NCC)) *For each table tbl of a schema s, let CL be the set of columns. Let nr be a new row to be inserted into tbl. For each $cl \in CL$, let $nncl \leftarrow nr(cl) \neq \perp$. TR contains two requirements for each cl, one in which $nncl = true$, and one where $nncl = false$.*

Note that UCC demands tests for truly-unique values, by ensuring the column value cannot be set to `NULL`. While UCC demands each column is not-`NULL`, it does not subsume NCC, since it does not mandate each column should be individually `NULL` also. Yet, these simple criteria lead to the same kind of problems as ICC, in that the columns are tested independently of the rest of the integrity constraints. An `INSERT` statement for an ICC test requirement is likely to be rejected on the basis that one or more of the integrity constraints of the target table are not satisfied. This makes the cause of any potential fault hard to discern, since rejection of the statement could have been for a number of reasons related to particular integrity constraints. We therefore present "active" versions of each criteria, which involve respecting the integrity constraints on the table while alternating the unique/not-unique and `NULL`/not-`NULL` status of each column, with *Active Unique Column Coverage (AUCC)* and *Active Null Column Coverage (ANCC)* respectively defined in the following fashion:

Criterion 8 (Active Unique Column Coverage (AUCC)) *For each table of a schema s, let tbl be the current table under consideration and CL be tbl's set of columns. For each $cl \in CL$, let nr be a new row to be inserted into tbl, and let $ucl \leftarrow \forall er \in tbl : nr(cl) \neq er(cl)$. Let ap_{aucc} be the acceptance predicate for tbl that does not account for integrity constraints that require cl to be individually unique (i.e.,* `UNIQUE` *constraints and* `PRIMARY KEY` *constraints defined on cl). TR contains two requirements for each cl, one in which $ucl = true \wedge nr(cl) \neq \perp \wedge ap_{aucc} = true$, and one where $ucl = false \wedge nr(cl) \neq \perp \wedge ap_{aucc} = true$.*

Criterion 9 (Active Null Column Coverage (ANCC)) *For each table of a schema s, let tbl be the current table under consideration and CL be tbl's set of columns. For each $cl \in CL$, let nr be a new row to be inserted into tbl, and let $ancl \leftarrow nr(cl) = \perp$. Let ap_{ancc} be the acceptance predicate for tbl that does not account for integrity constraints that require cl to be individually* `NULL` *(i.e., a* `NOT NULL` *constraint on cl; or a* `PRIMARY KEY` *constraint defined for cl only, in the case of a non-SQLite database). TR contains two requirements for each cl, one in which $ancl = true \wedge ap_{ancc} = true$, and one where $ancl = false \wedge ap_{ancc} = true$.*

Note that the "active" criteria exclude certain existing integrity constraints that are defined on the current column of interest — that is, `NOT NULL` constraints for ANCC and single-column `UNIQUE` constraints for AUCC. This is so that the column can be properly tested as needed: If a `NOT NULL` constraint exists on some column *cl* that must be respected, the test requirement involving making *cl* `NULL` would be infeasible. Likewise for AUCC, for the current column of interest *cl*, single column `PRIMARY KEY` constraints or `UNIQUE` constraints defined on that column are ignored. (Note that multi-column `PRIMARY KEY` constraints or `UNIQUE` constraints involving *cl* do not need to be ignored, as *cl* can be independently unique/non-unique, even in the presence of those constraints.) Notice further that AUCC and ANCC do not subsume APC. While the respective criteria guarantee the inclusion of a test requirement involving a row of data being accepted into each table of the schema, there is no reverse guarantee that a test requirement will involve a new row of data being rejected. Of course, this is unless the table has `UNIQUE` or `PRIMARY KEY` constraints, in the case of AUCC, and `NOT NULL` constraints in the case of ANCC — but the presence of such integrity constraints cannot be guaranteed for every database table.

In general, the number of test requirements generated for column coverage criteria is twice the number of columns in the tables of the schema-under-test. However, for ANCC, duplicate test requirements may be created when a table has more than one `NOT NULL` constraint defined for it. This is because mandating a column *cl* be not-`NULL` (as demanded by the criterion) is the same as *cl* having an actual `NOT NULL` constraint defined on it. Suppose a table

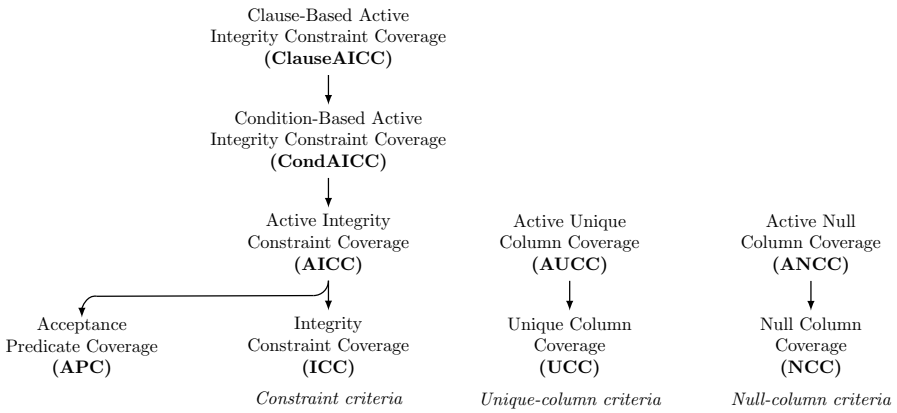


Figure 11: The coverage criteria subsumption hierarchy for testing relational database schemas.

t has two columns cl_1 and cl_2 with `NOT NULL` constraints defined on them. ANCC will generate a test requirement where cl_1 should be not-`NULL`, while cl_2 should already be not-`NULL` because of its `NOT NULL` constraint. This is identical to the test requirement for when cl_2 is mandated to be not-`NULL` while cl_1 must be not-`NULL` because of its integrity constraint.

There is no similar situation with AUCC, because mandating that a column be unique (as demanded by the criterion) is not the same as the column having a `UNIQUE` constraint defined on it, because a `UNIQUE` constraint can be satisfied by a `NULL` value *or* a unique value (i.e., the integrity constraint predicate also involves the null condition, in addition to the constraint condition that specifies uniqueness). For two columns cl_1 and cl_2 involved in two `UNIQUE` constraints, the test requirement that cl_1 must be unique cannot be satisfied by a `NULL` value, whereas the `UNIQUE` constraint defined for cl_2 means that the value generated for cl_2 can be unique or `NULL`. This test requirement, therefore, is not identical to the reverse case where cl_1 may be `NULL` *or* unique, while cl_2 *must* be unique.

3.9 Summary

In this section we have defined different coverage criteria for database schemas, which can be organized into three different subsumption hierarchies, as shown in Figure 11. In essence, there are two flavors of criteria — "constraint coverage" criteria, comprising ICC up to ClauseAICC (the leftmost subsumption hierarchy in the figure) and "column coverage" criteria; consisting of the "unique-column coverage" criteria, UCC and AUCC (the middle subsumption hierarchy); and the "null-column coverage" criteria, NCC and ANCC (the rightmost subsumption hierarchy). APC does not consider specific constraints and as such is not formally a part of the constraint coverage tree, although criteria from AICC upwards subsume it. Intuitively, the criteria at the top of the subsumption hierarchy in Figure 11 are "stronger" than those at the bottom, thus indicating that, for instance, a test suite satisfying ClauseAICC will also satisfy all of the criteria below it in the hierarchy.

4 Automatic Test Case Generation

In this section, we describe a framework that is capable of taking a test requirement, created by one of the coverage criteria introduced in the last section, and turning it into a concrete test case. These concrete generated test cases may then be used with a real database that instantiates the schema under test while managed by a specific DBMS.

The coverage criteria in the last section describe test requirements that formulate predicates for testing integrity constraints and acceptance predicates in which a new row of data nr is to be inserted into some table *tbl* of a database. This section describes how the data in nr is generated. Furthermore, recall that data may already be needed in the database before nr is inserted, in order ensure the database is in a "T-Sufficient" state, as per Definition 4 (page 9). This is so that the test does not trivially "pass" with any values for nr, or the test is infeasible from the outset. Our framework handles the problem of T-Sufficiency by generating test cases under the assumption that the database is empty (or can be emptied [24]), and that the test case itself is responsible for putting the database into the required T-Sufficient state, before the attempted insertion of nr. This also ensures that test cases can be executed independently and in isolation of other tests.

A test case, therefore, consists of a sequence of SQL `INSERT` statements:

Definition 5 (Test Case) *A* test case *tc is a sequence of* `INSERT` *statements* $\langle I_0 ... I_{tl} \rangle$ *designed to fulfill some test requirement tr. The "length" of the test case, denoted tl, is the number of* `INSERT` *statements that tc contains, subject to the restriction that* $tl > 0$.

The initial subsequence of `INSERT`s, up to but not including the final `INSERT` statement, are responsible for putting an empty database into the T-Sufficient state required for the test. We refer to this initial subsequence as the "presequence", as explained in the following definition.

Definition 6 (Presquence of a Test Case) *The* presequence *of a test case tc is the series of* `INSERT` *statements* $\langle I_0 ... I_{tl-1} \rangle$ *that put the database in the required T-Sufficient state so that the test requirement tr can be fulfilled. All* `INSERT` *statements in the presequence of a test case should be accepted by the underlying DBMS for which tc is generated.*

As the definition states, since the presequence is intended to modify the state of the database in order to ensure T-Sufficiency, each of its `INSERT` statement should be accepted by the DBMS. That is, the data contained in each statement must be generated such that it conforms to the acceptance predicate of the table concerned, thus allowing the data to become part of the database.

The final `INSERT` statement, involving the new row of data *nr*, may be accepted or rejected by the DBMS, depending on the test requirement. We therefore refer to it as the *decisive* `INSERT` of the test case:

Definition 7 (Decisive `INSERT` statement of a Test Case) *The* decisive `INSERT` *of a test case tc is the last* `INSERT` *statement* I_{tl} *of a test case that involves the new row of data nr to be submitted to the DBMS, in order to fulfill the test requirement tr on which tc is based.* I_{tl} *may be accepted or rejected by the DBMS, depending on the nature of tr.*

The first step in generating test cases is establishing the actual sequence of `INSERT` statements that is required, and the tables into which they need to insert data. At this point the data values of each `INSERT` are "blank". It is the responsibility of the second step of the test case generation process to then fill in those blanks with appropriate data values that fulfill the test requirement. The following two subsections details these two steps.

4.1 Step 1 – Determining the Necessary Sequence of `INSERT` Statements

Step 1 of the test case generation algorithm is itself subdivided into two phases. The first phase establishes a skeleton presequence of `INSERT`s needed for T-Sufficiency. The second phase then modifies the presequence to ensure that all foreign key relationships in the schema are accounted for, so that `INSERT`s in the test do not violate any `FOREIGN KEY` constraints and thus lead to rejection by the DBMS no matter what data they contain.

Phase 1. Skeleton Test Case. The first steps towards establishing T-Sufficiency in the presequence depends on what the test requirement is testing:

- **For `CHECK` constraints and test requirements that involve making columns `NULL`/not-`NULL` (e.g., for NCC and ANCC)**, an empty database is already T-Sufficient, so no presequence of `INSERT`s is required in the test case.
- **For uniqueness constraints and test requirements that involve making columns unique/not-unique (i.e., for UCC and AICC)**, an empty database is not T-Sufficient, since any value(s) for the column(s) that need to be unique will also be trivially accepted. Therefore, an `INSERT` statement for *tbl* must be added to the presequence so that the decisive `INSERT` may be potentially accepted or rejected, based on the data values contained within the two respective `INSERT`s (i.e., a "comparison" row of data in the presequence `INSERT` and the values of *nr* in the decisive `INSERT`).
- **For `FOREIGN KEY` constraints**, an empty database is not T-Sufficient. Since there is no data in the database for the decisive `INSERT` to potentially reference, it will always be rejected by default. Thus the presequence needs to have an `INSERT` to the table referenced by the `FOREIGN KEY` so that the decisive `INSERT` may potentially be accepted or rejected, based on the data values contained within the two respective `INSERT`s.

Example. As part of the ClauseAICC criterion the `UNIQUE` constraint on line 19 for the Cookies example of Figure 1 is tested, involving the `name`, `host` and `path` columns. One test requirement involves all `name` and `host` being *non-unique* and `path` *unique* (i.e., requirement 11 of Figure 10 (a)). The test case therefore requires an `INSERT`

to the `cookies` table (i.e., the decisive `INSERT` statement). Assuming all other integrity constraints are satisfied, this `INSERT` will be trivially accepted by the DBMS, no matter what the values in the `INSERT` statement are for `name`, `host` and `path`. Therefore, a prior `INSERT` is required to the `cookies` table to establish T-Sufficiency. This statement forms the initial presequence. Accordingly, the skeleton test case established as a result of Phase 1 is one that contains two `INSERT` statements, where both `INSERT`s are for the `cookies` table as shown in the following. The decisive `INSERT` is denoted D and the initial presequence `INSERT` statement is called P_A:

P_A	`INSERT INTO cookies VALUES(...)`
D	`INSERT INTO cookies VALUES(...)`

Phase 2. Satisfying Foreign Key Relationships. Whatever the test requirement, following the first phase, the test case will consist of at most two `INSERT` statements. The subject tables of these `INSERT`s may involve foreign key relationships that also need to be satisfied by the presequence of the test case, otherwise the `FOREIGN KEY` constraints concerned will be violated, and one or more `INSERT`s of the test case will always be rejected, regardless of the purpose of the original test requirement. With the example presented for Phase 1, for instance, the `cookies` table has a foreign key relationship with the `places` table. If the values for `host` and `path` inserted into the `cookies` table do not already appear in the `places` table, then both of those `INSERT` statements will fail.

In order to ensure all foreign key relationships are satisfied as part of the test case, additional `INSERT`s need to be injected into the presequence. These `INSERT`s ensure that data is in the database for reference by later `INSERT`s that might depend on them, thus preventing `FOREIGN KEY` violations. The algorithm for ensuring that this is the case takes each existing `INSERT` in the test case in turn. Let tbl_{target} be the table of an `INSERT` statement in the test case currently under consideration. The foreign key structure of the schema is explored, starting with tbl_{target} and analyzing tables directly and transitively linked to it through `FOREIGN KEY` constraints, in a depth-first fashion. Each new table encountered in the exploration process results in the injection of a new `INSERT` statement for that table into the presequence, either directly before the last injection (if one has been made) or before the existing `INSERT` from which the analysis began. A limitation of this process is that it cannot handle cyclic foreign key relationships between tables. If such a cycle is detected, the algorithm terminates in failure. Our test case generation algorithm cannot handle such schemas unless constraint enforcement is switched off in the DBMS. This, however, defeats the purpose of our technique. Ideally, the cyclic dependency — a hallmark of a schema that may be poorly designed — needs to be broken or otherwise reviewed and refactored [25]. As mentioned in Section 7, the complete handling of cyclic dependencies during test generation is an area for future work; with that said, it is important to note that our current approach works correctly for the 32 schemas used in the empirical study.

A fresh `INSERT` statement will not be injected into the presequence if an `INSERT` for that table already appears at some prior point in the sequence. Multiple `INSERT`s to the same table are not generally needed to satisfy foreign key relationships, since one row in the foreign key table may be referenced by multiple rows in one or more other tables. There is one exception to this rule, however. This is when a uniqueness constraint or property in *tbl* is being tested, and the columns involved are also part of a `FOREIGN KEY` constraint for *tbl*. Since the column values involved may need to be unique over two rows (nr and the "comparison" row in the presequence added in phase 1), the referenced values will also need to be unique, mandating two rows to the referenced table rather than just one.

Example. Moving forward with the example presented for Phase 1, the algorithm encounters the first `INSERT`, P_A. This has a `FOREIGN KEY` constraint referencing the `places` table. An `INSERT` to `places`, denoted P_B, is injected into the presequence before P_A:

P_B	`INSERT INTO places VALUES(...)`
P_A	`INSERT INTO cookies VALUES(...)`
D	`INSERT INTO cookies VALUES(...)`

The `places` table has no `FOREIGN KEY` constraints defined for it, so no prior `INSERT`s need to be injected for further tables. The algorithm moves to the statement D, also directed towards the `cookies` table, with the `FOREIGN KEY` constraint to `places`. The presequence already contains an `INSERT` to `places`, however this single `INSERT` in the presequence is insufficient. There is no way to have two different values for `path` in two `INSERT`s to `cookies` — to test `path` as unique, as per the original test requirement — without two rows in the `places` table with each of these two values. Therefore, the test must perform another `INSERT` on the `places` table, which is injected before D. The final test case therefore consists of four `INSERT`s, as follows:

P_B	`INSERT INTO places VALUES(...)`
P_A	`INSERT INTO cookies VALUES(...)`
P_C	`INSERT INTO places VALUES(...)`
D	`INSERT INTO cookies VALUES(...)`

Table 1: Generic column types used in test data generation, with example mappings to real DBMS types

Universal Type	General Type	Default Value	Initial Range	Examples of DBMS Types
Boolean	Atomic	False	False, True	`BOOLEAN`
DateTime	(Fixed) Compound	2000/1/1 00:00:00	1990/1/1 00:00:00–2020/1/1 23:59:59	`DATETIME`
Date	(Fixed) Compound	2000/1/1	1990/1/1–2020/1/1	`DATE`
Numeric	Atomic	0	-1000–1000 (to some number of decimal places)	`DECIMAL`, `DOUBLE`, `FLOAT`, `INTEGER`, `NUMERIC`, `REAL` ...
String	(Flexible) Compound	Empty string	characters: 'a'–'Z', string length 0–10	`CHAR`, `VARCHAR`, `TEXT` ...
Timestamp	Atomic	0	631152000–1577836800 (equivalent to 1990/1/1 00:00:00–2020/1/1 23:59:59)	`TIMESTAMP`
Time	(Fixed) Compound	00:00:00	00:00:00–23:59:59	`TIME`

4.2 Step 2 – Generation of Test Data Values for the `INSERT` Statements

Once the "blank" series of `INSERT` statements making up a test case has been determined, data values need to be generated. In the running example, the following data values would satisfy the test requirement (i.e., unique values for `path` in P_A and D, non-unique values for `name` and `host`, and with foreign key relationships satisfied such that the prior `INSERT`s to `places` contain data values referenced by later statements):

P_B	`INSERT INTO places(host, path, ...) VALUES('b', '1', ...)`
P_A	`INSERT INTO cookies(..., name, host, path, ...) VALUES(..., 'a', 'b', '1', ...)`
P_C	`INSERT INTO places(host, path, ...) VALUES('b', '2', ...)`
D	`INSERT INTO cookies(..., name, host, path, ...) VALUES(..., 'a', 'b', '2', ...)`

We present two algorithms for generating data values. First, however, we discuss how we handle the plethora of data types that a column can have, describing a solution that supports types in different real-world DBMSs.

4.2.1 Generating Test Values for Different Column Types

Since each DBMS has its own diverse set of column types, we developed an abstraction so that our techniques can map a specific DBMS column type to one of seven "universal" types. Each universal type is capable of encoding key concrete DBMS-specific properties, for example a particular range of values for an integer type. We list each of the universal types, along with example mappings, in Table 1.

We further distill the seven universal types into two more general types: *atomic types* and *compound types*. Atomic types are decimal numbers, with some specified minimum and maximum value, and a number of decimal places. For example, values of the "Boolean" type are either true or false; while values of the "Timestamp" type are integers. Values of an atomic type are encoded as decimal numbers. Compound types are formed from values of an atomic type joined together. Whether the type contains a definite number of values depends on whether the type is classed as *fixed* or *flexible*. The "Date" type is an example of a fixed compound type; consisting of three integers representing day, month and year values. The "String" type is a flexible compound type, consisting of a variable number of characters, represented by values. During the search, strings can shrink and grow in length (up to a pre-determined maximum) by having characters added to or removed from the end of the sequence.

4.2.2 Formulating the Goal Predicate

In order to establish when suitable test values have been found, the test generation approach formulates a "goal predicate" for each `INSERT` statement of the test case. For every `INSERT` statement of the presequence, the goal predicate is simply formed from the acceptance predicate of the table to which the `INSERT` is to be made. Recall the acceptance predicate is created from the conjunction of integrity constraint predicates, which are conditions over a new row of data nr to be inserted into the database, and each existing row er already in the database. The goal predicate for presequence `INSERT`s is therefore assessed on the basis of the data in the statement itself (i.e., nr), and each row of data er in the `INSERT`s to the relevant tables before it. Additionally, the goal predicate for the presequence `INSERT`s makes the further stipulation that each data value is not-`NULL`. The absence of `NULL` values in the database is important for guaranteeing T-Sufficiency, particularly in regard to testing `UNIQUE` and `FOREIGN KEY` constraints, which cannot be negated if the database state consists entirely of `NULL` values.

Given a means of representing data values in `INSERT` statements, and a predicate that the data values must satisfy, the next two subsections describe two algorithms for finding those values. We apply a search-based test

data generation approach [26], following the advice of Clark et al. [27] to apply local search to the problem first and then compare it to random search. We begin by introducing the random approach and then explain the local search approach that is based on Korel's Alternating Variable Method (*AVM*) for test data generation [10]. While more complex methods (e.g., Genetic Algorithms [26]) may be applicable to the task of generating data for the `INSERT` statements — and may even work better — we leave this for future research, as discussed in Section 7.

4.3 The *Random*$^+$ Method

Our implementation of the random data generation of values involves selecting values at random for each value of each `INSERT`. The column type is found for the value and mapped to one of the types listed in Table 1. A value is then selected from the type's range. With a probability of p_{null}, `NULL` is used instead. With a probability of p_{lib}, a value appropriate to the column's type is selected from a special "library" of values. This library is generated through mining the schema for constants found in any `CHECK` constraints it may have. Such constants appear, for example, on one side of an inequality or in a list of values used with an `IN` operator. Mining is performing by parsing the SQL `CREATE TABLE` statements used to construct the schema, extracting constants from the parse tree and inserting them into the library for later use by the random data generator.

Once all values have been selected, the test case is checked against the test requirement, using the goal predicate. If the requirement is fulfilled, then the process terminates with the test case, else random selection repeats.

We refer to this method as "*Random*$^+$", because the library of values enhances it over a pure random technique, and helps it satisfy test requirements involving arbitrary `CHECK` constraints more easily. However, finding certain data values through random search is still problematic for some types of test requirements. A more directed approach is needed, which is why we also developed the search-based technique that we describe in the next section.

4.4 The *AVM* (Alternating Variable Method)

Search-based data generators explore the domain of variables for appropriate test values [26]. Instead of being either exhaustive or completely random, the search is heuristic, using a fitness function to guide it to the required values. In the following content, we explain the *AVM*'s representation, fitness function and search strategy.

Representation. In order to apply search-based techniques, we need to encode actual solutions in the problem context (i.e., the initially "blank" values of the `INSERT` statements that form the test case), to some lower-level data structure that a search technique can utilize, which we call the problem *representation*. In this structure, the values for every `INSERT` that needs to be generated are mapped to a linear sequence, where each entry in the sequence corresponds to a particular column value of an `INSERT`. We refer to each entry of the sequence as a "cell".

A cell can either be `NULL` or encode an actual data value. Values are encoded by mapping the column's original data type to one of the types listed in Table 1. Atomic values are encoded with a single decimal number (e.g., 0 for false and 1 for true for a "Boolean" type), or a sequence of decimal values in the case of a compound type. Figure 12 shows how the values of two `INSERT` statements are encoded using our representation. Integer values map directly to the values used in the list, while strings appear as sequences of ASCII numbers.

Fitness Function. The process of forming a fitness function involves reformulation of a goal predicate into a distance function, which indicates "how far away" the existing test data values are from those that are needed for the data values appearing in the `INSERT` statements of the test case. Our distance functions are inspired by those used in structural testing [26], where if, for example, a predicate "`a == b`" needs to evaluated as true, the function $|\texttt{a} - \texttt{b}|$ is applied. Values of `a` and `b` that are closer together receive smaller distance values. Our atomic distance metrics, for comparing two data values (in potentially different `INSERT` statements), follow the same pattern. We extend these to compare atomic types with compound types and `NULL` values.

Figure 13 gives our distance functions. In addition to conjunctions and disjunctions, the *and_dist* and *or_dist* are used for handling universal and existential quantifiers, respectively. Each individual clause involves the comparison of two entities a and b, handled by *value_dist*. Depending on the types of a and b, and whether either is `NULL`, further calls may be required to compare atomic types using *atomic_dist* or compound types with *compound_dist*. In order to ensure that no one part of the goal predicate dominates the others in terms of the "proportion" of the final fitness value, distance values are normalized in the range [0,1] using Arcuri's function $norm(d) = \frac{d}{d+1}$ [28].

For instance, consider Figure 12's example involving the insertion of rows into the `places` and `cookies` tables of the *BrowserCookies* example and the respective encoding of values. The goal predicate is the conjunction of the acceptance predicate for `places` and `cookies`. For ease of understanding we limit our focus to one conjunct of the goal predicate, the `FOREIGN KEY` integrity constraint predicate of the `cookies` table, which is as follows:

SQL Test Case

```
INSERT INTO places VALUES(
   'host',    -- index: 1 (host: TEXT)
   'path',    -- index: 2 (path: TEXT)
   'title',   -- index: 3 (title: TEXT)
   1,         -- index: 4 (visit_count: INTEGER)
   'url'      -- index: 5 (fav_icon_url: TEXT)
);

INSERT INTO cookies VALUES(
   0,         -- index: 6 (id: INTEGER)
   'name',    -- index: 7 (name: TEXT)
   NULL,      -- index: 8 (value: TEXT)
   0,         -- index: 9 (expiry: INTEGER)
   1,         -- index: 10 (last_accessed: INTEGER)
   0,         -- index: 11 (creation_time: INTEGER)
   'host',    -- index: 12 (host: TEXT)
   'path',    -- index: 13 (path: TEXT)
);
```

Encoding

Cell Index	Encoded Value
1	$\langle 104, 111, 115, 116\rangle$
2	$\langle 112, 97, 116, 104\rangle$
3	$\langle 116, 105, 116, 108, 101\rangle$
4	1
5	$\langle 117, 114, 108\rangle$
6	0
7	$\langle 110, 97, 109, 101\rangle$
8	NULL
9	0
10	1
11	0
12	$\langle 104, 111, 115, 116\rangle$
13	$\langle 112, 97, 116, 104\rangle$

Figure 12: Encoding a test case, involving the insertion of two rows of data, into the representation used by the search. The `INSERT` statements to the left of the figure are annotated with indexes next to each data value, which map into the sequence of "cells" used by the search.

value_dist(a, op, b)

a	b	
NULL	NULL	return 0
NULL	*any*	return 1
any	NULL	return 1
Atomic	Atomic	return $norm(atomic\_dist(a, op, b))$
Compound	Compound	return $norm(compound\_dist(a, op, b))$

atomic_dist(a, op, b)

op	
$=$	if $(\lvert a-b\rvert = 0)$ then return 0 else return $\lvert a-b\rvert + 1$
$\neq$	if $(\lvert a-b\rvert \neq 0)$ then return 0 else return 1
$<$	if $(a-b<0)$ then return 0 else return $(a-b)+1$
$\leq$	if $(a-b\leq 0)$ then return 0 else return $(a-b)+1$
$>$	if $(b-a<0)$ then return 0 else return $(b-a)+1$
$\geq$	if $(b-a\leq 0)$ then return 0 else return $(b-a)+1$

compound_dist$((a_1, \ldots a_p), op, (b_1, \ldots b_q))$

op	
$=$	return $\lvert p-q\rvert + \sum_{i=1}^{min(p,q)} norm(atomic\_dist(a_i, =, b_i))$
$\neq$	if $(p \neq q)$ then return 0 else return $min_{i=1}^{min(p,q)} norm(atomic\_dist(a_i, \neq, b_i))$

other	$d \leftarrow 0$
	while $(i \leq min(p,q) \wedge d = 0)$
	  if $(a_i \neq b_i)$ then $d \leftarrow norm(atomic\_dist(a_i, op, b_i))$
	  $i \leftarrow i + 1$
	end while
	return $d + atomic\_dist(p, op, q)$

and_dist$(d_1, \ldots, d_n)$
return $norm(\sum_{i=0}^{i=n} d_i)$

or_dist$(d_1, \ldots, d_n)$
return $min_{i=0}^{i=n} d_i$

Figure 13: Distance functions for the formation of complete fitness functions

$$(nr(\texttt{host}) = \bot \vee nr(\texttt{path}) = \bot) \wedge (\exists er \in \texttt{places} : nr(\texttt{host}) = er(\texttt{host}) \wedge nr(\texttt{path}) = er(\texttt{path}))$$

This predicate can be transformed into a distance function, with references to the cells of the encoding, as previously described:

$$\begin{aligned} &and\_dist(or\_dist(value\_dist(cell(12), =, \texttt{NULL}), value\_dist(cell(13), =, \texttt{NULL})), \\ &\quad and\_dist(value\_dist(cell(1), =, cell(12), value\_dist(cell(2), =, cell(13))) \end{aligned}$$

Search Using the Alternating Variable Method. We adapt Korel's *Alternating Variable Method* [10] (*AVM*) as the search technique to minimize the fitness function, with the values of each cell initialized to Table 1's defaults.

The *AVM* sequentially makes adjustments to each cell in sequence, referred to as "moves". After each move, the list of values is evaluated according to the fitness function. If a move leads to an improvement in fitness, the new adjusted value is kept, else the value reverts to its previous state. The initial set of moves attempted for a cell are referred to as "exploratory" moves. The cell first has its `NULL` status flipped. If, following this move, the value is not `NULL`, then further moves are performed depending on the value's general type.

If the value is *atomic*, two moves are attempted, one which decreases the value, and one that increases the value. If either move is found to improve fitness, a series of "pattern" moves are made, which accelerate modifications to the value in the direction of improvement. Pattern move steps are computed using the function $step_m = 2^m \cdot 10^{-d} \cdot dir$ [8], where $step_m$ is the m^{th} successive move, dir is the direction of improvement, $dir \in \{-1, 1\}$, and d is the number of decimal places specified for the type. Pattern moves continue until a move is made that no longer improves fitness.

If the value is of a *compound* type, cells are simply treated as subsequences of further cells; with each element of the subsequence subjected to an exploratory move, followed by a pattern move in the case of progress. Flexible compound types (i.e., the "String" type, as shown by Table 1) have additional exploratory moves performed on their length, with characters added and removed to their end of their sequences.

An iteration of the *AVM* completes when either a fitness of zero has been reached, indicating that the required data values have been found for the test requirement, or when a complete cycle of exploratory moves has been made through the entire sequence of cells without any improvement in fitness. If the latter occurs, the *AVM* is restarted but with random values for each value — which are generated as with the $Random^+$ search described in Section 4.3. The whole algorithm terminates in failure if a certain number of fitness evaluations have been used and the required data values have not been found.

5 Empirical Study

We designed an empirical study to assess the effectiveness of our coverage criteria, with the aim of answering the following four research questions:

RQ1: Coverage. How do the number of test requirements generated by the coverage criteria differ depending on the criterion, the DBMS and the data generation technique being used, and how successfully can test cases be automatically generated to satisfy them?

RQ2: Effectiveness at Finding Faults. How effective are the test suites generated for each coverage criteria at finding faults? How does fault finding effectiveness vary depending on the combination of criterion, DBMS and data generation technique used?

RQ3: Coverage Criteria and Types of Faults. Are certain types of faults more easily found with certain criteria? If so, what patterns emerge?

RQ4: Combining Criteria and Fault-Finding Effectiveness. Does combining criteria from different subsumption hierarchies increase fault-finding capability compared to using individual criterion to derive requirements?

We implemented the coverage criteria and data generation techniques discussed in this paper into a tool called *SchemaAnalyst* [9, 29, 30], and used it to perform the experiments needed to answer our research questions. This also necessitated the collection of relational database schema subjects, as discussed in the next subsection.

5.1 Subject Relational Database Schemas Studied

As shown in Table 2, we gathered 32 schemas that contain different types of integrity constraints of varying levels of complexity, thus making the results of our study as generalizable as is possible. This set was compiled from a

Table 2: Schemas used in the empirical study. Figures in brackets indicate the number of "multi-clause" constraints that result in additional test requirements for ClauseAICC compared to CondAICC (i.e., multi-column `PRIMARY KEY`, `FOREIGN KEY` and `UNIQUE` constraints; and `CHECK` constraints made up of `AND`s, `OR`s, `BETWEEN`s or `IN`s).

Schema	Tables	Columns	Total Constraints	CHECK Constraints	FOREIGN KEY Constraints	NOT NULL Constraints	PRIMARY KEY Constraints	UNIQUE Constraints
ArtistSimilarity	2	3	3 (0)	0 (0)	2 (0)	0	1 (0)	0 (0)
ArtistTerm	5	7	7 (0)	0 (0)	4 (0)	0	3 (0)	0 (0)
BankAccount	2	9	8 (0)	0 (0)	1 (0)	5	2 (0)	0 (0)
BookTown	22	67	28 (1)	2 (1)	0 (0)	15	11 (0)	0 (0)
BrowserCookies	2	13	10 (4)	2 (1)	1 (1)	4	2 (1)	1 (1)
Cloc	2	10	0 (0)	0 (0)	0 (0)	0	0 (0)	0 (0)
CoffeeOrders	5	20	19 (0)	0 (0)	4 (0)	10	5 (0)	0 (0)
CustomerOrder	7	32	42 (1)	1 (1)	7 (0)	27	7 (0)	0 (0)
DellStore	8	52	39 (0)	0 (0)	0 (0)	39	0 (0)	0 (0)
Employee	1	7	4 (0)	3 (0)	0 (0)	0	1 (0)	0 (0)
Examination	2	21	9 (0)	6 (0)	1 (0)	0	2 (0)	0 (0)
Flights	2	13	10 (4)	1 (1)	1 (1)	6	2 (2)	0 (0)
FrenchTowns	3	14	24 (1)	0 (0)	2 (0)	13	0 (0)	9 (1)
Inventory	1	4	2 (0)	0 (0)	0 (0)	0	1 (0)	1 (0)
Iso3166	1	3	3 (0)	0 (0)	0 (0)	2	1 (0)	0 (0)
iTrust	42	309	134 (15)	8 (8)	1 (0)	88	37 (7)	0 (0)
JWhoisServer	6	49	50 (0)	0 (0)	0 (0)	44	6 (0)	0 (0)
MozillaExtensions	6	51	7 (4)	0 (0)	0 (0)	0	2 (0)	5 (4)
MozillaPermissions	1	8	1 (0)	0 (0)	0 (0)	0	1 (0)	0 (0)
NistDML181	2	7	2 (2)	0 (0)	1 (1)	0	1 (1)	0 (0)
NistDML182	2	32	2 (2)	0 (0)	1 (1)	0	1 (1)	0 (0)
NistDML183	2	6	2 (2)	0 (0)	1 (1)	0	0 (0)	1 (1)
NistWeather	2	9	13 (6)	5 (5)	1 (0)	5	2 (1)	0 (0)
NistXTS748	1	3	3 (0)	1 (0)	0 (0)	1	0 (0)	1 (0)
NistXTS749	2	7	7 (1)	1 (0)	1 (0)	3	2 (1)	0 (0)
Person	1	5	7 (1)	1 (1)	0 (0)	5	1 (0)	0 (0)
Products	3	9	14 (1)	4 (0)	2 (0)	5	3 (1)	0 (0)
RiskIt	13	57	36 (1)	0 (0)	10 (0)	15	11 (1)	0 (0)
StackOverflow	4	43	5 (0)	0 (0)	0 (0)	5	0 (0)	0 (0)
StudentResidence	2	6	8 (0)	3 (0)	1 (0)	2	2 (0)	0 (0)
UnixUsage	8	32	24 (1)	0 (0)	7 (0)	10	7 (1)	0 (0)
Usda	10	67	31 (0)	0 (0)	0 (0)	31	0 (0)	0 (0)
Total	172	975	554 (47)	38 (18)	49 (5)	335	114 (17)	18 (7)

variety of sources, including databases used in production and in open-source software. Houkjær et al. [31] notes that real-world complex relational schemas often include features such as composite keys and multi-column foreign-key relationships. As such, our set of schemas reflects a diverse set of features from simple instances of each of the main types of integrity constraint (i.e., `PRIMARY KEY` constraints, `FOREIGN KEY` constraints, `UNIQUE` constraints, `NOT NULL` constraints and `CHECK` constraints) to more complex examples involving many-column foreign key relationships.

Several schemas were taken from real-world database-driven applications: *Cloc* is used as a data repository for a popular open-source application to count the number of various types of lines in code for a large range of programming languages (`http://cloc.sourceforge.net`). *JWhoisServer* is used in an open-source, Java-based implementation of a server for the internet "WHOIS" protocol (`http://jwhoisserver.net`). Both *MozillaExtensions* and *MozillaPermissions* were extracted from SQLite databases that are a part of the Mozilla Firefox Internet browser. *RiskIt* is part of system for modeling the risk of insuring individuals (`http://sourceforge.net/projects/riskitinsurance`), adjusting their premium based on their likelihood of making a claim. *StackOverflow* is the schema used by a popular programming question and answer website, as previously studied in a conference data mining challenge [32], while *UnixUsage* is taken from an application for monitoring and recording the Unix commands used by a group of students. Some of these schemas have featured in previous studies of various testing methods (e.g., *RiskIt* and *UnixUsage* [33], and *JWhoisServer* [34]).

ArtistSimilarity and *ArtistTerm* are part of the "Million Song" dataset, a database of song metadata [35].

The six "Nist" schemas are from the SQL Conformance Test Suite of the National Institute of Standards and Technology (NIST) (`http://www.itl.nist.gov/fipspubs/fip193.htm`), and have featured in past studies such as as those conducted by Tuya et al. [36], while several schemas were taken from the samples for the PostgreSQL DBMS (e.g., *DellStore*, *FrenchTowns*, *Iso3166* and *Usda*), available from the `PgFoundry.org` website. *BrowserCookies* is the schema used in this paper to illustrate challenges with schema testing, introduced in Figure 1.

The remainder (e.g., *BankAccount*, *BookTown*, *CoffeeOrders*, *CustomerOrder*, *Person* and *Products*) were extracted from textbooks, laboratory assignments, and online tutorials, where they were provided as examples. Nonetheless, it is important to note that many of these are sufficiently complex for the well-established *DBMonster* tool, an open-source SQL data generator, to have difficulties handling them, as shown in our previous work [9]. *iTrust*, in particular, is a large schema designed for the scenario of a medical application for teaching students about software testing methods; it previously was featured in a mutation analysis experiment of Java code [37].

The original schemas were intended to be used with one of the DBMSs studied in this paper (i.e., PostgreSQL, HyperSQL and SQLite), or were in suitably generic SQL such that they could be imported easily into one of those DBMSs. The SQL for each schema was parsed into an abstract object representation in our *SchemaAnalyst* tool,

Table 3: Mutation operators for relational database schemas

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

Original Schema

```
1 CREATE TABLE places (
2    host TEXT NOT NULL,
3    path TEXT NOT NULL,
4    title TEXT,
5    visit_count INTEGER,
6    fav_icon_url TEXT,
7    PRIMARY KEY(host, path)
8 );
```

Mutant 1

```
1 CREATE TABLE places (
2    host TEXT NOT NULL,
3    path TEXT NOT NULL,
4    title TEXT,
5    visit_count INTEGER,
6    fav_icon_url TEXT,
7    PRIMARY KEY(path)
8 );
```

Mutant 2

```
1 CREATE TABLE places (
2    host TEXT NOT NULL,
3    path TEXT NOT NULL,
4    title TEXT,
5    visit_count INTEGER,
6    fav_icon_url TEXT,
7    PRIMARY KEY(host)
8 );
```

Figure 14: The original SQL `CREATE TABLE` statement for the `places` table of the *BrowserCookies* example of Figure 1 and two mutants produced by the PKColumnR (`PRIMARY KEY` Column Removal) operator. The operator produces mutants by systematically removing columns from `PRIMARY KEY` constraints.

using the General SQL Parser[3], as described in our prior work [30]. Once parsed into this representation, the schema could be written out in the SQL suitable for the other DBMSs, such that we could use each schema with each DBMS, regardless of subtle syntactic differences in SQL used by the creators of each DBMS.

5.2 Assessing the Fault-Finding Capability of Test Suites Using Mutation Analysis

In order to provide answers to research questions 2–4, we apply *mutation analysis*, a technique for estimating the fault-finding capability of test suites where particular types of faults are concerned [38]. Mutation analysis works by generating a series of *mutants* for some artifact under test — in this case, the integrity constraint specification portion of a database schema [29]. Mutants are copies of the original artifact but with small modifications, or *mutations*, intended to alter the behavior of the artifact, and thus model a fault that might be made by a real software developer. A mutant is said to be "killed" when a test case exposes differences in the behavior of one or more test cases when applied using the mutant and the original artifact under test. The more mutants a test suite is capable of detecting, the more discerning it is likely to be in terms of exposing real faults in practice [38].

Mutants are produced by *mutation operators*, which are responsible for altering the artifact-under-test in a certain systematic way. In our previous work on database schema testing [9, 30], we described 14 different mutation operators for mutating the integrity constraints of a schema. These operators are listed in Table 3. Each operator is named according to the constraint that it affects (i.e., `PRIMARY KEY` constraint, `FOREIGN KEY` constraint, `NOT NULL` constraint, `UNIQUE` constraint, `CHECK` constraint) and the type of modification they make (e.g., Addition of an element such as a column, Removal of an element, or Exchanging an element with another). For instance, the UColumnA operator mutates a `UNIQUE` constraint by adding a column, while the FKColumnPairR modifies a `FOREIGN KEY` constraint by removing a column from the source table and its associated column in the referenced table. An example of the two mutants produced by the PKColumnR operator for the `places` table of the *BrowserCookies* schema are shown in Figure 14. The `PRIMARY KEY` constraint of this table involves the column pair `host` and `path`. In each of these mutants, one of these columns is removed.

Not all mutants produced by a mutation operator are useful. For instance, operators may produce an "equivalent" mutant, which is the result of some mutation that actually results in the same behavior as the original artifact, thus making it indistinguishable from the original in terms of its operation. We apply techniques, implemented in the *SchemaAnalyst* tool and described in our previous work [30], for removing certain types of "ineffective" mutants — including classes of equivalent mutants, certain types of "redundant" mutant that are functionally equivalent to some other mutant already generated, and mutants that represent invalid or infeasible schemas (e.g., so-called

[3]http://www.sqlparser.com

"still-born" mutants). Whether a mutant is classed as "ineffective" or not depends on the DBMS, and as such, the final set of mutants removed varies depending on the DBMS of current interest. (We discuss these issues in more detail in Wright et al. [30], and refer the reader to that reference for more information.)

As with traditional mutation analysis for programs, mutation analysis for database schemas makes use of a test suite to be executed against the original non-mutated artifact-under-test, and each individual mutant created by one of the mutation operators. When executing the series of `INSERT` statements comprising a test case for a schema, we record whether each `INSERT` was accepted by the DBMS, or rejected, due to an integrity constraint violation. In the context of integrity constraint testing, a mutant is *killed* if a test case (as defined in Definition 5 (page 17)) registers a difference in the sequence of acceptances and rejections made by the DBMS with the mutant compared to when the original schema is used.

5.3 Experimental Procedure

Our experimental procedure involved generating test suites for each of the 32 schemas with each combination of:

- Coverage criterion (i.e., the "constraint coverage" criteria *APC*, *ICC*, *AICC*, *CondAICC*, *ClauseAICC* and the "column coverage" criteria *UCC*, *AUCC*, *NCC* and *ANCC*), as detailed in Section 3; and
- DBMS (i.e., PostgreSQL, HyperSQL and SQLite); and
- Data generation technique (i.e., $Random^{+}$ and the *AVM*, as explained in Section 4);

Due to the stochasticity inherent in each data generation technique, we repeated test suite generation 30 times with a different random seed. This was done to reduce the possibility of our empirical results being produced by mere "chance". We then applied the generated test suites to mutant versions of each schema according to the mutation analysis approach detailed in the last subsection. For the *AVM* and $Random^{+}$, we set $p_{null} = 0.1$ and $p_{lib} = 0.25$, which, as introduced in Section 4.3, are the probabilities associated with using a `NULL` value or a constant mined from a schema's `CHECK` constraints, respectively. We found that, so long as the probabilities were not greater than 0.5, the actual values used did not influence the effectiveness of the test data generator. The maximum number of fitness evaluations was set to $100,000$ — a standard termination limit from the literature on search-based test data generation [8]. As detailed in Section 7, we plan, as part of future work, to conduct additional experiments to further discern how the data generators are sensitive to the tuning of these parameters. With that said, it is worth noting that prior empirical studies in the context of search-based test data generation for Java programs (e.g., [39, 40]) suggest that parameter tuning rarely improves the effectiveness of the data generators.

To best ensure that Section 5.5's data visualizations do not obscure the most noteworthy empirical trends, we always show the value of the evaluation metrics over all relational schemas (e.g., in Figure 15 we plot the percentage of test requirements covered over all of the 32 schemas). When appropriate to do so, we also comment on data points for specific schemas, normally with either a focus on the largest and smallest of the schemas or, alternatively, on the database schema that best illustrates a fundamental trade-off in the empirical results.

5.4 Threats to Validity

The following threats to validity are inherent in our empirical study. We now discuss how we mitigated their possible effects from the outset.

1. **The schemas are not representative of real-world schemas.** While the rich and diverse nature of real software systems makes it impossible for us to claim that our schemas are representative of all the characteristics of all possible relational database schemas, we endeavored to select schemas from a wide variety of sources, comprising real-world applications, conformance suites, textbook examples and schemas from databases that were used in previous studies, as explained in Section 5.1. Furthermore, Table 2 shows the diversity captured by our 32 schemas, with 1–42 tables, 3–309 columns, and 0–134 constraints, including `CHECK`s, `FOREIGN KEY`s, `PRIMARY KEY`s, `NOT NULL`s and `UNIQUE`s.

2. **The mutation operators are not representative of real faults.** According to the "competent programmer" hypothesis [41], programmers are likely to produce programs that are nearly correct, implying that real faults will frequently be the result of small mistakes. Our mutation operators are designed to model such faults, in the context of relational database schemas, by making small changes to each type of constraint. By implementing operators for both the addition and removal of columns we model faults of both omission and commission, further improving the range of faults our operators can represent. It is worth noting that prior empirical studies have demonstrated that mutation faults are indeed a valid substitute for experimentation in

Table 4: Total numbers of test requirements derived for different schema for each coverage criterion. Since HyperSQL and PostgreSQL share the same model of integrity constraints, test requirement numbers are identical. Numbers in brackets correspond to the final number of test requirements used in the experiments after duplicate and trivially infeasible requirements are removed.

(a) Constraint Coverage Criteria

	PostgreSQL/HyperSQL		SQLite	
APC	316	(316)	316	(316)
ICC	958	(958)	1108	(1108)
AICC	958	(637)	1108	(712)
CondAICC	1177	(893)	1327	(1000)
ClauseAICC	1571	(1288)	1721	(1378)

(b) Column Coverage Criteria

	PostgreSQL/HyperSQL		SQLite	
UCC	1950	(1950)	1950	(1950)
AUCC	1950	(1950)	1950	(1950)
NCC	1950	(1950)	1950	(1950)
ANCC	1950	(1694)	1950	(1739)

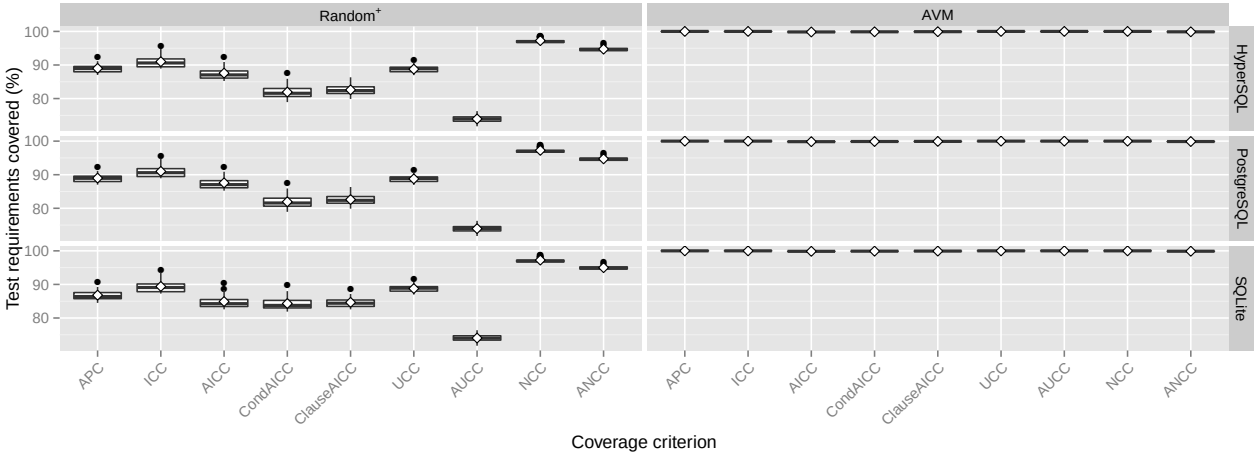


Figure 15: The percentage of test requirements covered for each coverage criterion over all schemas with each DBMS, using *Random*$^+$ and the *AVM* test data generators. Each box spans from the 1st to 3rd quartile, with whiskers extending up to 1.5× the inter-quartile range. The line across the middle of the box marks the median. The white diamond denotes the mean, while filled circles correspond to outliers.

the absence of real-world faults [42, 43, 44]. While we cannot, strictly speaking, argue that these results also apply to the use of mutation for database schemas, it is possible that they do hold since our schema mutants are very similar in spirit to those used in these past experiments with Java and C programs.

3. **The statistical analysis used.** Where not obvious from the box plots accompanying answers to each research question, statistical tests were performed to ascertain statistical significance of sample means. We used the Mann–Whitney U test (Wilcoxon rank-sum test), a nonparametric test, since the normality of the sample means could not be guaranteed as required by parametric statistical tests such as the t-test.

4. **Defects in our *SchemaAnalyst* tool leading to incorrect results.** It is possible that defects are present in our implementation; however, we have a suite of comprehensive, and frequently executed, unit tests for our tool. In addition, empirical results were cross-checked whenever possible to ensure the absence of errors.

5.5 Answers to Research Questions

RQ1: Coverage

Table 4 summarizes the total number of test requirements generated for all of the schemas featuring in our study, following the removal of trivially infeasible requirements. Trivially infeasible requirements are where a goal predicate mandates that a particular table column be both `NULL` and not `NULL` simultaneously. This form of infeasibility is easy to identify and remove. However, more complex forms of infeasible requirements are possible and yet not as easily identifiable — usually as the result of arbitrary `CHECK` constraints — and as such our *SchemaAnalyst* tool did not remove them in advance.

For the "constraint coverage" criteria, the number of requirements increase moving up through the subsumption hierarchy, since the criteria become more complex and add test requirements for the more fine-grained aspects of integrity constraints that can be tested. In general, the number of requirements is twice the number of columns for the "column coverage" criteria. However, for ANCC, duplicate test requirements may be created when a table has

more than one `NOT NULL` constraint defined for it (as described in Section 3.8), leaving the final test requirement count for ANCC lower than NCC.

As well as considering the number of test requirements derived for each criterion, we are interested in how successful our data generation techniques are at "covering" them, that is, finding data to complete the `INSERT` statements for each concrete test case designed to fulfill each individual test requirement. Figure 15 shows box and whisker plots of the percentage of test requirements covered for all schemas with a particular a criterion, for the 30 repetitions of the experiments. The plots clearly show the *AVM* to be more successful at generating data to cover test requirements than *Random*$^+$. With the *AVM*, all test requirements are successfully covered with the exception of a small number of infeasible test requirements that remained following the initial filtering of trivially-infeasible ones. These more complex forms of infeasibility occurred for the *Products* schema and the AICC, CondAICC and ClauseAICC constraint coverage criteria, and also for the *BookTown* schema with the ANCC column coverage criterion.

The *Random*$^+$ data generator fails to consistently achieve 100% coverage for any of the criteria. In general, *Random*$^+$'s performance gets worse the higher a criterion is in its particular subsumption hierarchy. The only exception is moving from CondAICC to ClauseAICC, which seems to add test requirements that are easier for *Random*$^+$ to cover. However, Wilcoxon Rank-Sum tests comparing the respective sets of repetitions for the two criteria are not significant at the 0.05 level — the p-value is 0.09 for HyperSQL and PostgreSQL, and 0.2 for SQLite.

In terms of variation across DBMSs, no variation is observed with the *AVM* due to perfect coverage scores being obtained in every instance. When comparing *Random*$^+$ test data generation across the three DBMSs, no variation is observed between PostgreSQL and HyperSQL. This is because the two DBMSs implement integrity constraints in the same way, giving rise to identical test requirements for all criteria, resulting in test cases that were the same (aside from differences in the way data for columns types across the two DBMSs are expressed). As discussed in Section 3, SQLite varies in its implementation of `PRIMARY KEY` constraint, thus leading to slightly different test requirements that manifest in a slight variation in coverage score when compared to HyperSQL and PostgreSQL.

Conclusion for RQ1: It is possible to reliably generate test suites with full coverage for each of the coverage criteria, so long as the *AVM* technique is used. Differences between DBMSs (if any) can be accounted for by the variations in behavior between those DBMSs in terms of their implementation of their integrity constraints.

RQ2: Effectiveness at Finding Faults

To assess fault-finding capability we applied mutation analysis for relational database schemas, as described in Section 5.2. Application of our complete set of mutation operators to all of our schemas, following the removal of "ineffective" mutants, totaled 3,775 with HyperSQL and PostgreSQL, and 3,915 with SQLite. Since HyperSQL and PostgreSQL share the same model of integrity constraint behavior, the number of mutants produced in each case is identical, while there is a small amount of variation between these two DBMSs and SQLite.

Figure 16 shows the percentage of these mutants killed with the test cases generated for a particular coverage criterion, over the 30 repetitions of the experiment. The plots reveal a high degree of consistency across each of the 30 repetitions of the experiment, with a maximum inter-quartile range (represented by the length of the box) of 1.7% and 0.2% for *Random*$^+$ and the *AVM*, respectively.

The percentage of mutants killed consistently improves when moving from one particular coverage criterion to the next further up in a specific subsumption hierarchy. For the "constraint coverage" criteria, this is partly due to an increasing number of test requirements, and therefore the number of test cases generated. However, adding test cases does not result in an automatic increase in the number of mutants killed — it matters what test requirements are necessitated by the coverage criterion. For instance, following the removal of duplicate and infeasible test requirements, the AICC criterion produces fewer test requirements than does ICC. However, AICC leads to tests that kill a significantly higher percentage of mutants. This is easily seen in Figure 17, a scatter plot of the mean percentage of mutants killed against the mean number of test cases for each coverage criterion. The data in this figure also shows that, for the "column coverage" criteria, there is no correlation between test suite size and mutation score, particularly when *Random*$^+$ is used as the data generator.

One reason for the higher percentages obtained by column coverage criteria compared to constraint coverage criteria is the highly productive nature of mutation operators for `UNIQUE` and `NOT NULL` constraints, which column criteria are well suited to killing. These operators produce 52% of the total mutants when aggregated over all schemas and DBMSs, following the removal of "useless" mutants. As Figure 18 shows, the "null-column" and "unique-column" criteria are able to kill a high proportion of mutants related to `UNIQUE` and `NOT NULL` constraints, explaining why the high percentages for these criteria are evident in the results.

Figure 16 shows that for "constraint" coverage criteria, the percentage of mutants killed with test suites generated by *Random*$^+$ search are consistently poorer than those obtained with the *AVM*. This is likely due to the fact that

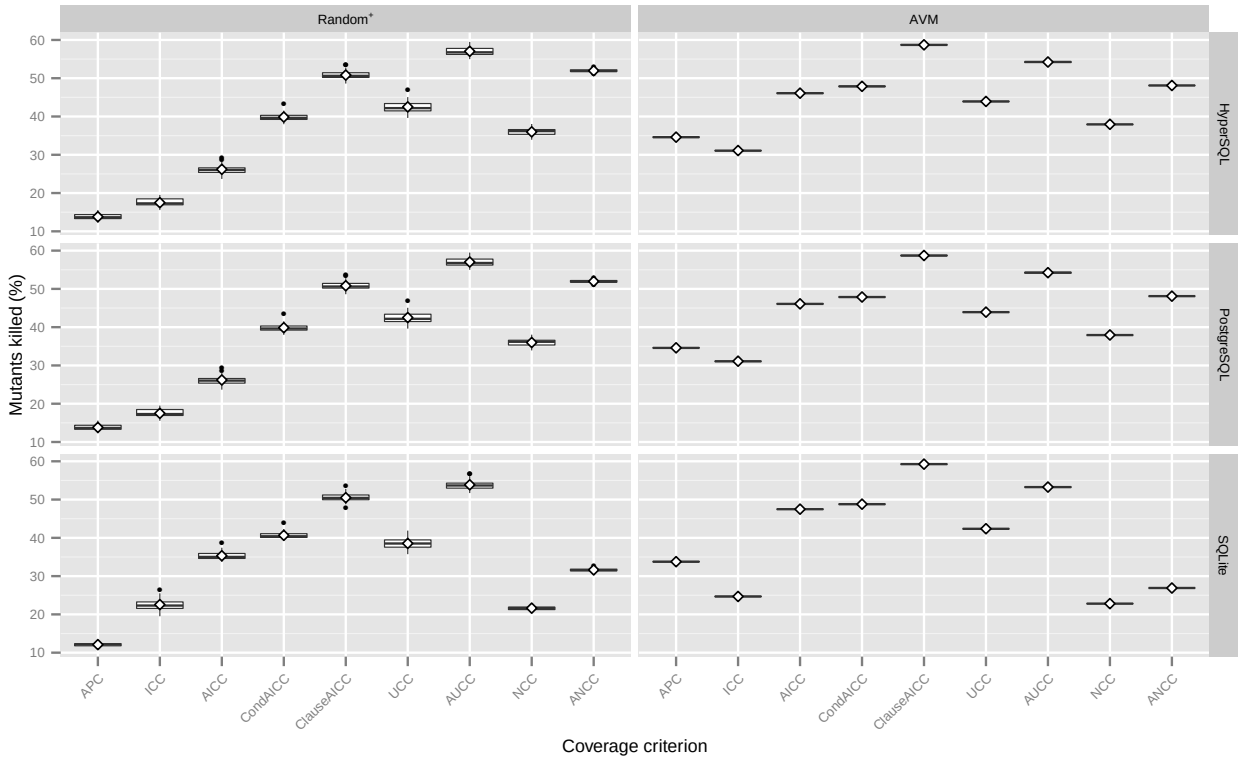


Figure 16: The percentage of mutants produced over all schemas that were killed using test cases generated for each coverage criterion with each DBMS (i.e., HyperSQL, PostgreSQL and SQLite), using both of the data generators (i.e., *Random*$^+$ and the *AVM*). (Box plots should be interpreted as for Figure 15.)

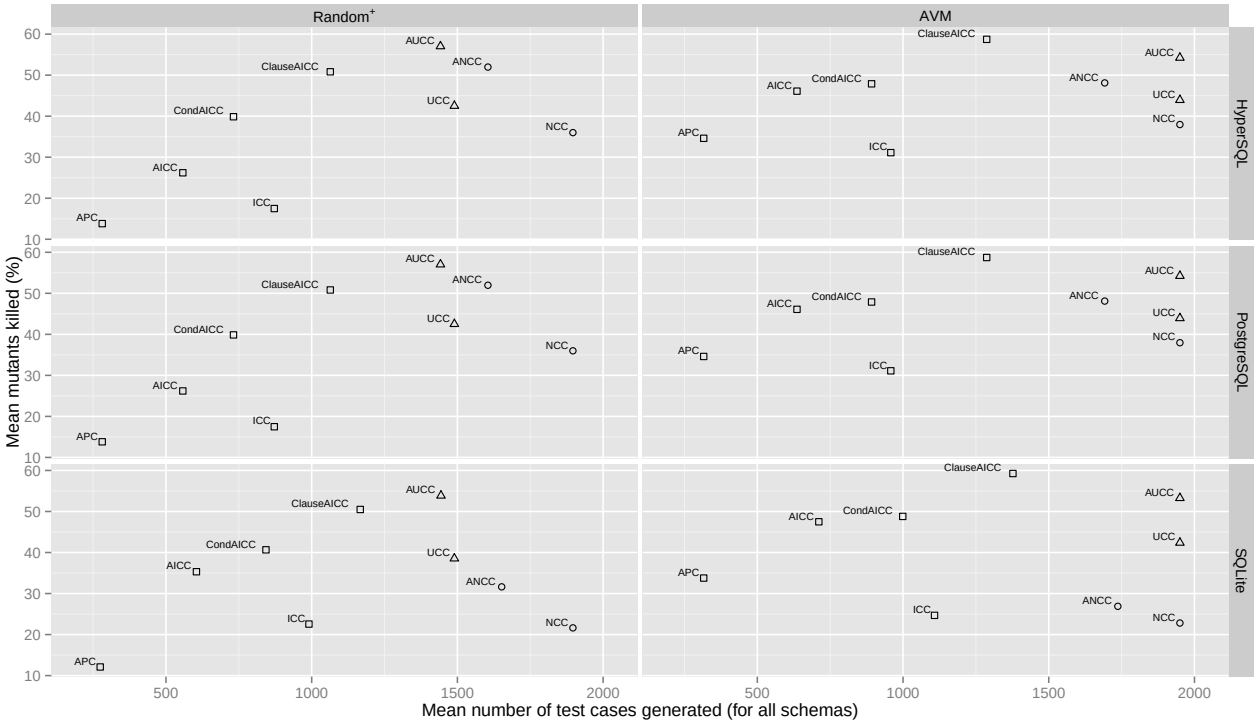


Figure 17: The mean percentage of mutants killed against mean number of test cases generated over all schemas. Constraint criteria, unique-column criteria and null-column criteria are plotted as square, triangular and circular points, respectively.

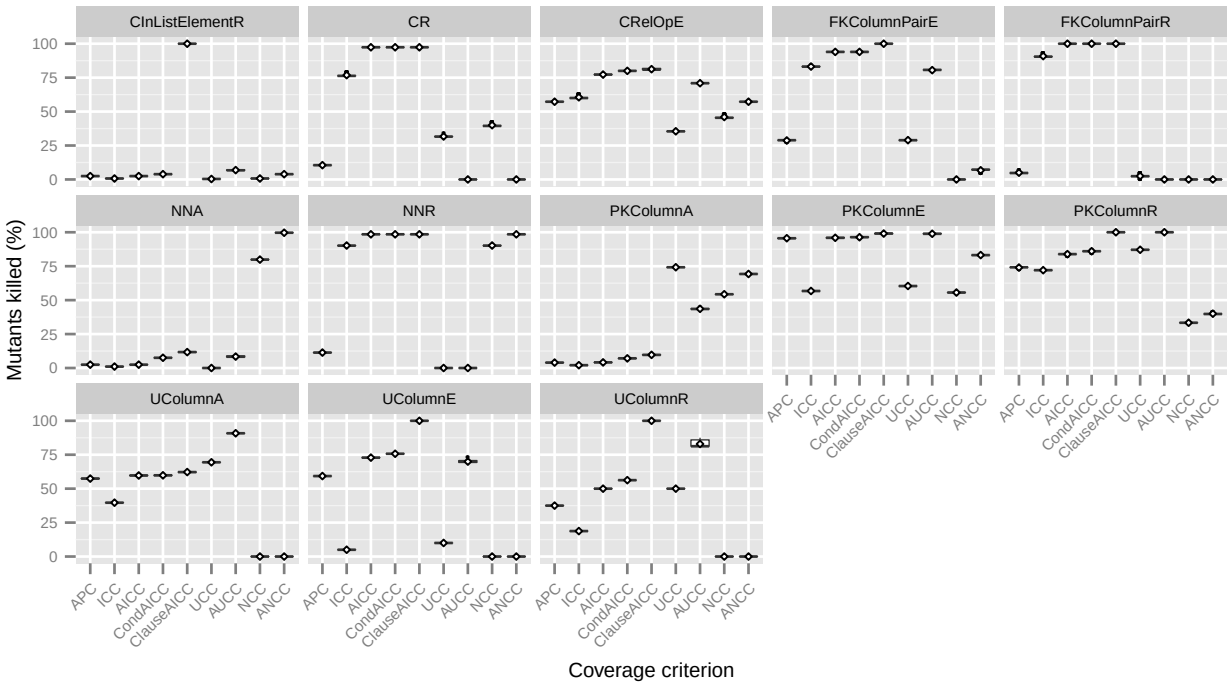


Figure 18: The percentage of mutants produced by a particular mutation operator for all of the schemas that were killed by test cases generated when the *AVM* used different coverage criteria with the HyperSQL/PostgreSQL DBMS. (Box plots should be interpreted as for Figure 15.)

fewer test requirements are covered by *Random*$^+$ search, resulting in fewer opportunities with which to kill the same types of mutants. Interestingly, for "column" coverage criteria, *Random*$^+$'s tests often outperform those produced by the *AVM*. This seems to be due to the increased diversity inherent in the test cases generated by *Random*$^+$ search. The *AVM* generates tests from the same default starting point, and unless a random restart is required, many test cases share similar characteristics, potentially lowering the likelihood of killing as many mutants.

For the *AVM*, ClauseAICC is the coverage criterion for which test cases obtained the highest mutant-killed percentages, for all three DBMSs. For *Random*$^+$, it is AUCC. However, ClauseAICC test suites generated with the *AVM* killed a significantly higher percentage of mutants than AUCC test cases generated with *Random*$^+$. For SQLite, this is clearly seen from the Figure 16, since the distributions do not overlap. For PostgreSQL and HyperSQL, the situation is less judicable from the figure alone; however, the Wilcoxon Rank-Sum test reveals a highly significant p-value of less than 10^{-8}. Overall, therefore, we judge that test suites generated according to the ClauseAICC criterion with the *AVM* are likely to have the best fault-finding capabilities.

With respect to cross-DBMS comparisons, there is no difference between the results obtained with HyperSQL and PostgreSQL, as found in the answer to the last research question regarding coverage. There are some differences between those DBMSs and SQLite, due to the difference in the way it implements `PRIMARY KEY` constraints. The most noticeable difference is the performance of NCC and ANCC. For HyperSQL and PostgreSQL, these criteria help kill `PRIMARY KEY` mutants, since columns must implicitly be not `NULL` for primary key columns. Thus NCC and ANCC perform better for these DBMSs compared to SQLite, for which primary key columns may be `NULL`.

Conclusion for RQ2: The ClauseAICC criterion produces test suites with the highest mutation killing power when test suites are generated with the *AVM*. AUCC is the strongest criterion when test cases are generated with *Random*$^+$. Yet, ClauseAICC tests produced with the *AVM* are stronger than AUCC-based tests produced by *Random*$^+$.

RQ3: Coverage Criteria and Types of Faults

Figures 18 and 19 show the percentages of mutants killed broken down by the mutation operator that produced them for the HyperSQL/PostgreSQL DBMSs. As we previously observed in the answer to the last research question the box plots reveal high consistency across trials, even with *Random*$^+$ test data generation. There are some broad trends for the operators that hold across data generation technique and DBMS. The first is that the column coverage criteria, unsurprisingly, tend to be amongst the best criteria at killing mutants to do with changing the `UNIQUE` and

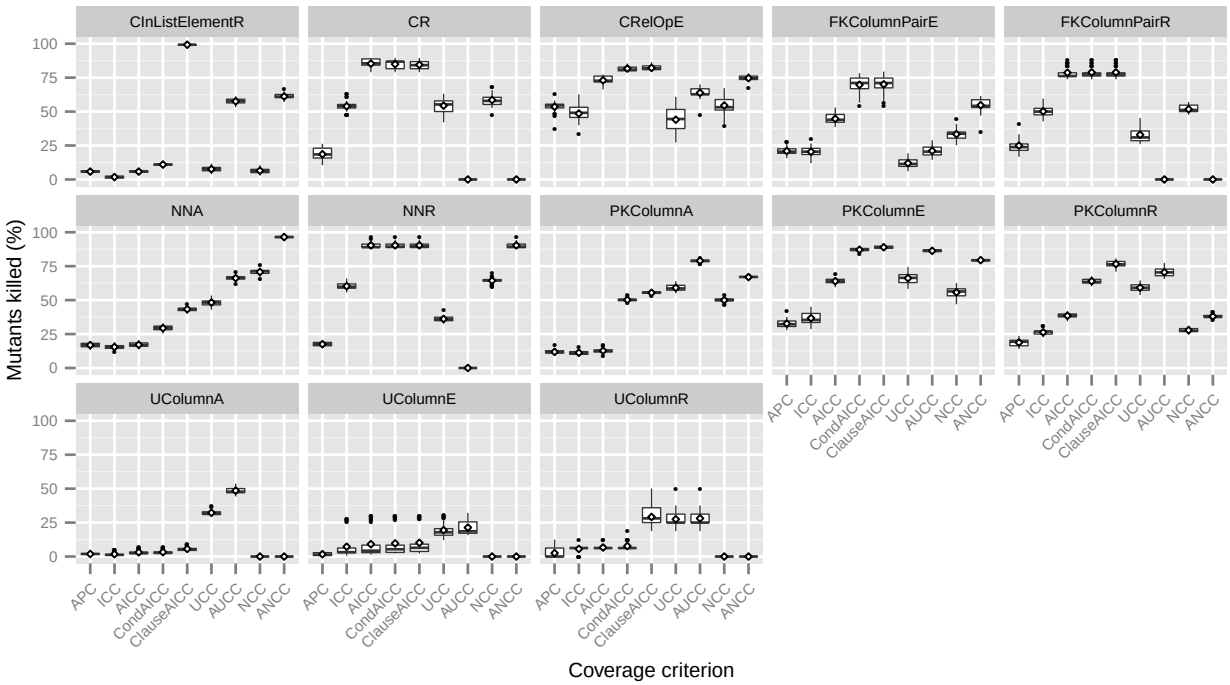


Figure 19: The percentage of mutants produced by a particular mutation operator for all of the schemas that were killed by test cases generated using *Random*$^+$ with different coverage criteria and the HyperSQL/PostgreSQL DBMS. (Box plots should be interpreted as for Figure 15.)

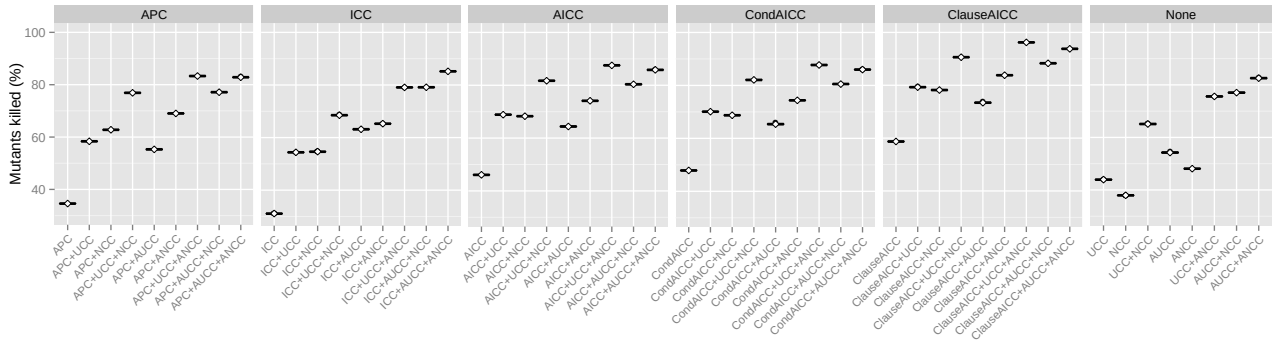
`NULL` status of columns. AUCC outperforms all other criteria for mutants produced by the UColumnA operator, where columns are given `UNIQUE` status through the addition of a column to an existing or new `UNIQUE` integrity constraint. (One of the reasons behind AUCC's success, as found in the answer to the last research question, seems to be to do with its strong ability to kill mutants from this operator and the high volume of mutants it produces.) ANCC outperforms all other criteria for mutants produced by the NNA operator, where columns are given `NULL` status through the addition of a `NULL` integrity constraint.

These patterns exist because the constraint criteria only test integrity constraints that already exist as part of the schema, not for those that may have been omitted (or have had additional columns omitted from their declaration, for example in the case of `UNIQUE` constraints). Column criteria therefore seem well-suited to testing for faults of "omission" whereas constraint criteria are stronger at testing for faults of "commission". The latter is evident by the fact that criteria such as ClauseAICC are good at producing test cases for detecting the removal of columns from `UNIQUE` constraints and the removal of `NOT NULL` constraints as seen in plots for the UCColumnR (which removes columns from `UNIQUE` constraints) and NNCR (which removes `NOT NULL` constraints) operators. They also tend to frequently detect changes in `CHECK` constraints and `FOREIGN KEY` constraints, especially those generated with ClauseAICC.

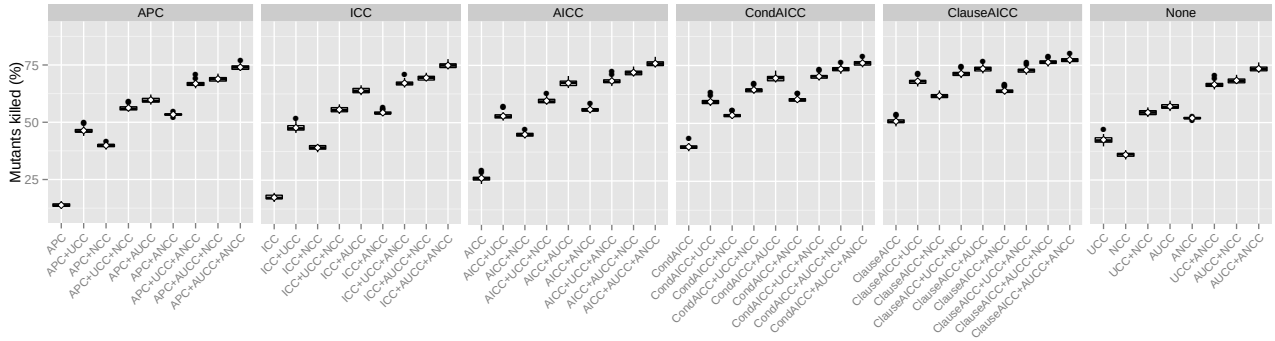
Some aspects of the results vary depending on the data generation technique. For instance, ClauseAICC does not detect columns being swapped into `UNIQUE` constraints with the UCColumnE operator anywhere near as well with *Random*$^+$ data generation as with the *AVM*. This, however, is due to the way that test suites are generated, rather than the criterion. The *AVM*'s test suites tend not be as diverse as *Random*$^+$, as the search starts from the same default initial vector of data values. If these values remain unchanged, the *AVM* can detect exchanges of columns from `UNIQUE` constraints more easily, since they will more readily generate non-unique column values for over consecutive `INSERT` statements. *Random*$^+$, in contrast, will generate diverse and more likely unique values that fail to detect the change.

For space reasons, we do not show the results for the SQLite DBMS, which are broadly similar to those for HyperSQL/PostgreSQL. As found previously, any differences between DBMSs tend to be due to the difference in the way that `PRIMARY KEY` constraints work for HyperSQL and PostgreSQL compared to SQLite. The `NULL` coverage criteria NCC and ANCC tend to perform well for primary key operators for HyperSQL and PostgreSQL but not SQLite, since primary keys in SQLite do not implicitly include a `NOT NULL` constraint on the columns involved.

Conclusion for RQ3: Constraint coverage criteria and column coverage criteria have some complementary effects



(a) With the *AVM*



(b) With *Random*$^+$

Figure 20: The percentage of mutants killed over all schemas with different constraint and column coverage criteria, when using the PostgreSQL DBMS with different data generators. (Box plots should be interpreted as for Figure 15.)

in revealing different types of faults. Constraint coverage criteria, ClauseAICC in particular, tend to better at trapping faults of commission while column constrain criteria, AUCC and ANCC in particular, tend to be good at trapping certain types of omission faults.

RQ4: Combining Criteria and Fault-Finding Effectiveness

Given that the coverage criteria in our three different subsumption hierarchies have very different aims, and given the answer to the last research question — suggesting that different mutants are killed by different criteria from these hierarchies — this final research question assesses how the combination of test cases generated for different criteria may affect the percentage of mutants killed.

Figure 20 shows box plots of the percentage of mutants killed by using all of the test cases generated for different 2-way and 3-way combinations of criteria. The figure shows the results for the *AVM* (part (a) of the figure) and *Random*$^+$ data generator (part (b)) when used with PostgreSQL only (however the trends are similar for other DBMSs). For the *AVM*, the ClauseAICC + UCC + ANCC combination achieves the highest mutants-killed percentage (mean average 96%). This combination involves UCC rather than AUCC, the former being different from the other two elements of the combination (ClauseAICC and ANCC) in that it is not at the top of its particular subsumption hierarchy. Rather, ClauseAICC + AUCC + ANCC comes second for the *AVM* with a mean average of 94% mutants killed. However, the success of UCC in combination with ClauseAICC and ANCC may be an artefact of the *AVM* technique, as the best combination *does* involves AUCC when *Random*$^+$ is used as the data generation technique, since for *Random*$^+$, ClauseAICC + AUCC + ANCC is the combination that kills the most mutants. Since the *AVM* uses default values, non-unique values may be generated for all columns at once when generating tests with UCC, and it is this characteristic of the data generator that seems to be responsible for collaterally killing more mutants, not because of the specific strengths of UCC in particular.

Given that such a high percentage of mutants were killed with the top-scoring combinations of criteria, particularly when considering the use of the *AVM*, we investigated the constitution of the mutants that remained alive with that technique. We found that these mutants were those produced by the CRelOpE, PKColumnA and UColumnA operators. CRelOpE produces mutants for relational expressions in `CHECK` constraints where the relational operator is changed, while PKColumnA and UColumnA add columns to the existing column set of `PRIMARY KEY` and `UNIQUE` constraints respectively. These column sets may be empty in the case that the schema is mutated by adding a

PRIMARY KEY constraint or UNIQUE constraint. It appears that our criteria lead to tests that still have trouble in killing mutants produced in the alternative case, where the operator is adding a column to an existing constraint.

Conclusion for RQ4: Combining coverage criteria from each of the three subsumption hierarchies improves fault-finding capability, due to the different types of faults each of the criteria in those hierarchies is good at targeting. Combining criteria at the top of each hierarchy results in the best fault-finding capability, although the use of UCC "beats" AUCC when the *AVM* is used to generate test cases.

6 Related Work

Since this paper presents coverage criteria that support the testing of the integrity constraints encoded in the schema of a relational database, it is related to prior work in the area of adequacy criteria and testing methods for both database management systems and database applications. Kapfhammer and Soffa presented what is, to the best of our knowledge, the first family of test adequacy criteria for database applications [45]; it was later extended by Wilmor and Embury [46]. Like our paper, Kapfhammer and Soffa's work formally defined adequacy criteria and organized them into a subsumption hierarchy. Yet, in contrast to our paper's focus on the schema, their criteria consider the flow of data between the program and the database. Halfond and Orso also introduced a criterion called "command form" coverage; instead of concentrating on the database's schema, it uses a test coverage monitor, like the one developed by Kapfhammer and Soffa [47], to assess whether or not the various database commands that could be generated by an application are actually exercised [48]. Next, Tuya et al. proposed a form of predicate coverage, based on the masking modified condition decision coverage criterion [19], that, unlike our paper, tracks the coverage of SQL queries instead of the database schema [49]. Finally, Pan et al. presented a method that calculates the coverage of a database application given a fixed state for a relational database [33]. It is important to note that, since the aforementioned criteria each have a different emphasis than those presented in this paper, they are ultimately complementary approaches to evaluating the quality of tests for database-related software.

In recent years, researchers have developed a wide variety of methods that automatically generate various types of data in support of testing database management systems and database applications; we survey the most directly related work in this section. Slutz presented the random generation of SQL (RAGS) tool that creates SQL queries that can support the testing of a DBMS [50]. Later, Bati et al. implemented a genetic algorithm-based system that was more effective than RAGS at generating queries in support of DBMS testing [51]. Similar to the two aforementioned papers, Bruno et al. developed a different type of generator that also enabled DBMS testing by creating queries designed to produce result sets of a desired size [52]. In the most recent work on this topic, Khalek et al. also created a data generator that can support the testing of a DBMS [53]. All of these papers are different than ours because, firstly, they generate SQL queries instead of INSERT statements and, secondly, they primarily focus on the testing of a database management system instead of a relational schema.

In addition to methods for generating queries, there exist many data generators that can populate the database. The first notable system, called UpSizeR, can "scale" a database by a specified factor, thus making it larger or smaller to facilitate activities like performance testing [25]. Although UpSizeR does technically generate data, it is different than our approach because it does not focus on correctness and it cannot generate "negative cases" designed specifically to violate the schema's integrity constraints. There are also several manual or semi-automated methods that can generate test data. For instance, Bruno and Chaudhuri presented a specification language that helps a tester define a flexible data generator, with the primary aim of avoiding the proliferation of bespoke data creation schemes implemented in a myriad of languages [54]. Chays et al. described one of the best known data generators, called AGENDA, that asks the tester to employ the category-partition method to create data in a semi-automated fashion [55]. Finally, Willmor and Embury proposed an "intensional" approach that also supports the manual specification of test suites for database applications [46]. Since they are completely automated and require no guidance from the tester, the two data generators presented in this paper are distinct from these three previously described systems. Moreover, while our two data generators are systematically guided by test adequacy criteria, this is not the case for the systems presented in the three aforementioned papers.

Similar to our generators, Khalek et al., Lo et al., and de la Riva et al. presented automated methods for generating test data [56, 57, 58]. Additionally, Shah et al. described the X-Data tool for automatically generating test data designed to kill the mutants of SQL queries [59]. Although these four examples of prior work all operate automatically, they create data in support of testing SQL queries, while we focus on testing the integrity constraints in the schema. Even though the X-Data system also uses mutation analysis, it does so to guide the generator towards effective data while we use it as a way to evaluate the effectiveness of test data generated according to logic coverage criteria. Ultimately, these previously mentioned systems could be used, in conjunction with our test adequacy criteria and data generators, as part of a complete system for testing database applications. Finally, Houkjær et al.

introduced a method for automatically generating data while taking into account both the database's schema and any pre-existing data [31]. In contrast to this paper's methods, their focus is on performance testing and, as such, their experiments do not evaluate the effectiveness of the generated data at finding schema faults. There is another difference between this paper and the one by Houkjær et al.: while we demonstrate that our techniques support the testing of 32 relational databases schemas (many of which are from real-world databases), their experiments only consider databases that are part of benchmarks from the Transaction Processing Performance Council.

Since search-based test data generators, like the ones employed in this paper, are complex software artifacts, it is often important to evaluate their efficiency [60]. Even though performance is not the focus of this paper, it is worth noting that Kinneer et al. [61, 62] developed and used automated methods for empirical assessments regarding worst-case time complexity, applying them to the data generation techniques proposed in this paper. They found that data generation scales linearly with the size of most types of schemas, yet for others is quadratic, or in rare cases, cubic or worse; they also noted that the stronger coverage criteria always necessitate more time for test data generation [61]. Since they were produced with sizable schemas that contained hundreds or thousands of tables, constraints, and attributes, these results suggest that this paper's criteria support efficient test data generation.

This paper also presents logic coverage criteria that support the systematic testing of relational database schemas. There is a long history of work in logic testing that, according to Amman and Offutt, stretches back to 1979 [5]. With the exception of Tuya et al.'s development of a predicate coverage criterion for SQL queries [49], none of the prior work has, to the best of our knowledge, considered any aspect of relational databases. While the idea of condition and decision coverage has been used in practice for many years, it was formalized by Zhu et al. [21]. More germane to this paper and its focus on active coverage criteria, is Chilenski and Miller's proposal of modified condition decision coverage (MC/DC) [19], Chilenski and Richey's development of masking MC/DC [63], Dupuy and Leveson's empirical evaluation of MC/DC's effectiveness, and, more broadly, Amman and Offutt's unified explanation of the logic coverage criteria [5]. Work on the development of logic criteria continues: recently Kaminski et al. proposed a new criterion, called minimal-MUMCUT, that has been shown to find more faults than MC/DC [64]. Since the focus of this paper is on the creation and evaluation of logic testing criteria for integrity constraints in relational database schemas, it may be possible to follow the overall strategy of the aforementioned paper to develop "database-aware" versions of these recently proposed criteria.

Since this paper employs mutation analysis to evaluate the quality of the automatically generated test data, we briefly survey related work in this area. In our own prior work, we developed and empirically evaluated various methods for improving the efficiency of mutation analysis [29] and studied the impact that different types of schema mutants can have on the mutation score [30]. Since this paper is not concerned with improving or better understanding the mutation analysis of schemas, we simply incorporate the best practices and empirical findings from our prior work into the experiments of this paper. In contrast to this paper's focus on the relational schema, the prior work of others has proposed and evaluated mutation operators for the SQL `SELECT` statements used by applications to retrieve data stored in a database [36]. This approach was later incorporated into a tool for instrumenting and testing database applications written in the Java programming language, potentially mutating any executed `SELECT` statement [65]. Finally, Chan et al. proposed some operators for mutating schemas [66]; yet, unlike this paper, they provide neither an implementation nor an evaluation. All of this prior work is similar to this paper in that each of these mutation analysis methods make small changes to components of database applications.

7 Conclusions and Future Work

The data in a relational database is often described as an organization's most important asset, with the database's schema expressing the integrity constraints that protect this valuable data. Since, despite industry advice to the contrary, there has been little work focused on assessing the correctness of the integrity constraints in a relational database schema, this paper presented nine different coverage criteria for testing the design and implementation of these critically important constraints. "Constraint Coverage" criteria, based on logic coverage criteria [5], test the formulation of integrity constraints that have actually been specified in the schema, making them well-suited to finding faults of commission, while "Column Coverage" criteria test the unique and `NULL` status of columns in tables, making them more suited to finding faults of omission. After explaining how our criteria account for the idiosyncrasies associated with the underlying DBMS that hosts the data and the schema, this paper formally defined the test adequacy criteria and related them in three subsumption hierarchies.

We presented two approaches for generating test data to satisfy the test requirements of each criterion. The first, *Random*$^+$, generates data randomly with a bias towards certain constants appearing in the `CHECK` constraints of the schema. The second is a search-based method based on Korel's Alternating Variable Method (*AVM*) [10], which uses a fitness function to guide it to the required test data. The test data generated becomes part of a series

of SQL `INSERT` statements. Our testing procedure then checks whether these `INSERT`s succeed or fail as expected.

Incorporating 32 schemas — including some real-world ones from databases in, for instance, the Mozilla Firefox Internet Browser and the StackOverflow website — and three representative and frequently used database management systems (i.e., HyperSQL, PostgreSQL and SQLite), this paper reported on an empirical study investigating four research questions. The experiments for the first question revealed that it is possible to reliably generate test suites that achieve full coverage for each of the criteria in the subsumption hierarchies. To answer the second research question, we evaluated our coverage criteria using mutation analysis. The results showed that, in general, the "higher" a criterion was in the subsumption hierarchy, the more mutants it killed. As an answer to the third research question, the experiments pointed out that different subsumption hierarchies were indeed more suited to killing certain types of mutants than others. Yet, we also discovered that, in answer to research question four, even higher mutation scores could be obtained by combining criteria across the subsumption hierarchies.

As part of future work, we intend to develop new methods for testing data types in schemas — for example, by constructing `INSERT` statements with values that are in-range and out of bounds. We also intend to apply different types of search techniques, such as Genetic Algorithms and hybrids of the random, local, and global search methods. Moreover, we plan general improvements to the data generation process so that it both handles more types of relational schemas and kills more mutants. For instance, we are working on improvements to the data generator that can efficiently and effectively identify and manage cyclic dependencies between tables in a database.

To best ensure that we can complete future experiments with relational schemas larger than those used in this paper, we plan to enhance our prior work in mutation analysis (e.g., [29, 30]) to further increase the efficiency of this process. In addition to future experimentation with larger schemas and more database management systems, we also intend to conduct many new experiments. Following the experimental protocols established by Arcuri and Fraser [39] and Kotelyanskii and Kapfhammer [40], we will perform experiments to determine whether tuning parameters can affect the performance of the search process for generating data. In adherence to the guidelines set by Fraser et al. in their study of test generation for Java classes [67], we will also empirically compare human-produced tests with those generated by automated methods like the ones in this paper. We also intend to improve the human-readability of our tests, particularly in the area of string values [68, 69, 70]. Finally, following the experimental protocol established by Just et al. [43], we want to conduct experiments to conclusively determine if schema mutants are a valid substitute for real-world faults in relational database schemas.

Given that different DBMS vendors interpret the SQL standard differently, thus providing different implementations of integrity constraints, in future work we also intend to investigate the possibility of producing integrity constraint mutants that are targeted towards the differences across DBMSes — a form of "semantic mutation" [71] for DBMSs. Finally, we will integrate the presented adequacy criteria with different database-aware testing methods (e.g., test case prioritization [24], test suite reduction [72] and automated fault localization [73]), to ensure that they benefit from the enhanced guidance often afforded by a systematic testing strategy. Ultimately, combining this paper's test adequacy criteria and data generators with the improvements completed during future work will yield an effective and efficient way to thoroughly test the integrity constraints in a relational database schema.

For more information about the *SchemaAnalyst* tool, please visit our website (`http://www.schemaanalyst.org`).

Acknowledgements. Phil McMinn was supported in part by EPSRC grant EP/I010386/1.

References

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

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

[3] S. Loebman, D. Nunley, Y. Kwon, B. Howe, M. Balazinska, and J. P. Gardner, "Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help?" in *Proceedings of the Workshop on Interfaces and Abstractions for Scientific Data Storage*, 2009.

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

[5] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[6] R. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE Software*, vol. 18, no. 3, 2001.

[7] J. Ploski, M. Rohr, P. Schwenkenberg, and W. Hasselbring, "Research issues in software fault categorization," *SIGSOFT Software Engineering Notes*, vol. 32, no. 6, 2007.

[8] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, 2010.

[9] 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.

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

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

[12] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A distributed, searchable key-value store," in *Proceedings of the International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2012.

[13] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, "TIMBER: A native XML database," *The VLDB Journal*, vol. 11, no. 4, 2002.

[14] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "MapReduce and parallel DBMSs: Friends or foes?" *Communications of the ACM*, vol. 53, no. 1, 2010.

[15] B. Howe and D. Halperin, "Advancing declarative query in the long tail of science," *Data Engineering Bulletin*, vol. 35, no. 3, 2012.

[16] F. Pascal, *Practical Issues in Database Management: A Reference for the Thinking Practitioner*. Addison-Wesley, 2000.

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

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

[19] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, 1994.

[20] A. Dupuy and N. Leveson, "An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software," in *Proceedings of the 19th Digital Avionics Systems Conference*, 2000.

[21] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, 1997.

[22] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, 1970.

[23] D. Maier, *The Theory of Relational Databases*. Computer Science Press, 1983.

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

[25] Y. C. Tay, B. T. Dai, D. T. Wang, E. Y. Sun, Y. Lin, and Y. Lin, "UpSizeR: Synthetically scaling an empirical relational database," *Information Systems*, vol. 38, no. 8, 2013.

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

[27] J. A. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. F. Jones, M. Lumkin, B. S. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. J. Shepperd, "Reformulating software engineering as a search problem," *IEE Proceedings — Software*, vol. 150, 2003.

[28] A. Arcuri, "It does matter how you normalise the branch distance in search based software testing," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, 2010.

[29] 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.

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

[31] 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.

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

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

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

[35] 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.

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

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

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

[39] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, 2013.

[40] A. Kotelyanskii and G. M. Kapfhammer, "Parameter tuning for search-based test-data generation revisited: Support for previous results," in *Proceedings of the 14th International Conference on Quality Software*, 2014.

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

[42] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, 2006.

[43] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd International Symposium on the Foundations of Software Engineering*, 2014.

[44] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[45] G. M. Kapfhammer and M. L. Soffa, "A family of test adequacy criteria for database-driven applications," in *Proceedings of the 9th European Software Engineering Conference and the 11th Symposium on the Foundations of Software Engineering*, 2003.

[46] D. Willmor and S. M. Embury, "An intensional approach to the specification of test cases for database applications," in *Proceedings of the 28th International Conference on Software Engineering*, 2006.

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

[48] 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.

[49] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Full predicate coverage for testing SQL database queries," *Software Testing, Verification and Reliability*, vol. 20, no. 3, 2010.

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

[51] 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.

[52] N. Bruno, S. Chaudhuri, and D. Thomas, "Generating queries with cardinality constraints for DBMS testing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 12, 2006.

[53] S. Abdul Khalek and S. Khurshid, "Automated SQL query generation for systematic testing of database engines," in *Proceedings of the 25th International Conference on Automated Software Engineering*, 2010.

[54] N. Bruno and S. Chaudhuri, "Flexible database generators," in *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

[55] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weber, "A framework for testing database applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2000.

[56] S. Khalek, B. Elkarablieh, Y. Laleye, and S. Khurshid, "Query-aware test generation using a relational constraint solver," in *Proceedings of the 23rd International Conference on Automated Software Engineering*, 2008.

[57] E. Lo, C. Binnig, D. Kossmann, M. Tamer Özsu, and W.-K. Hon, "A framework for testing DBMS features," *The Very Large Data Bases Journal*, vol. 19, no. 2, 2010.

[58] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya, "Constraint-based test database generation for SQL queries," in *Proceedings of the 5th International Workshop on the Automation of Software Test*, 2010.

[59] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. Gupta, and D. Vira, "Generating test data for killing SQL mutants: A constraint-based approach," in *Proceedings of the 27th International Conference on Data Engineering*, 2011.

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

[61] C. Kinneer, G. M. Kapfhammer, C. J. Wright, and P. McMinn, "Automatically evaluating the efficiency of search-based test data generation for relational database schemas," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, 2015.

[62] ——, "expOse: Inferring worst-case time complexity by automatic empirical study," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, 2015.

[63] J. J. Chilenski and L. A. Richey, "Definition for a masking form of modified condition decision coverage," Boeing, Tech. Rep., 1997.

[64] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *Journal of Systems and Software*, vol. 86, no. 8, 2013.

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

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

[67] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? A controlled empirical study," *ACM Transactions on Software Engineering Methodology*, To Appear.

[68] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *International Conference on Software Testing, Verification and Validation (ICST 2013)*, 2013.

[69] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-based test input generation for string data types using the results of web queries," in *International Conference on Software Testing, Verification and Validation (ICST 2012)*, 2012.

[70] M. Shahbaz, P. McMinn, and M. Stevenson, "Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions," *Science of Computer Programming*, vol. 97, no. 4, 2015.

[71] J. A. Clark, H. Dan, and R. M. Hierons, "Semantic mutation testing," *Science of Computer Programming*, vol. 78, no. 4, 2013.

[72] G. Kapfhammer, "Towards a method for reducing the test suites of database applications," in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, 2012.

[73] 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.