What Factors Make SQL Test Cases Understandable For Testers? A Human Study of Automated Test Data Generation Techniques

Abdullah Alsharif
University of Sheffield

Gregory M. Kapfhammer
Allegheny College

Phil McMinn
University of Sheffield

***Abstract*—Since relational databases are a key component of software systems ranging from small mobile to large enterprise applications, there are well-studied methods that automatically generate test cases for database-related functionality. Yet, there has been no research to analyze how well testers — who must often serve as an "oracle" — both understand tests involving SQL and decide if they reveal flaws. This paper reports on a human study of test comprehension in the context of automatically generated tests that assess the correct specification of the integrity constraints in a relational database schema. In this domain, a tool generates INSERT statements with data values designed to either satisfy (i.e., be accepted into the database) or violate the schema (i.e., be rejected from the database). The study reveals two key findings. First, the choice of data values in INSERTs influences human understandability: the use of default values for elements not involved in the test (but necessary for adhering to SQL's syntax rules) aided participants, allowing them to easily identify and understand the important test values. Yet, negative numbers and "garbage" strings hindered this process. The second finding is more far reaching: humans found the outcome of test cases very difficult to predict when NULL was used in conjunction with foreign keys and CHECK constraints. This suggests that, while including NULLs can surface the confusing semantics of database schemas, their use makes tests less understandable for humans.**

I. Introduction

Often considered one of an organization's most valuable resources [1], a relational database is the backbone of many software applications. As such, software engineering practitioners advocate the testing of relational databases [2], which frequently have a complex and hard-to-specify schema that defines how the database's data is structured. A schema encodes integrity constraints that safeguard the database's state [3]. For instance, a record in a database may be identified through an attribute that is always unique, specified in a UNIQUE constraint. Since a UNIQUE ensures that data values are distinct, its accidental omission from the schema can compromise database correctness and increase future maintenance costs.

Prior work presented a family of coverage criteria aiding the systematic testing of database schemas [4]. These criteria require the creation of data rows in INSERT statements that exercise integrity constraints as *true* or *false*. The testing goal is to run INSERTs with data values that either satisfy (i.e., are accepted into the database) or violate the schema (i.e., are rejected from the database). For instance, a test can violate a UNIQUE constraint (i.e., exercise it as *false*) by populating a database and then trying to add an identical value. Or, a test can satisfy the UNIQUE (i.e., exercise it as *true*) by always using different data values. Since schemas often contain many tables and integrity constraints, manually writing most of these tests is tedious and error-prone. As such, prior work presented automated test data generators that output a test suite of INSERTs that effectively covers the test requirements [5], [6].

It is challenging to create test cases that are understandable and maintainable [7], [8] — especially when the tests use complex and inter-dependent INSERT statements to populate a relational database [9]. While automated test data generators can create test cases that aid systematic database schema testing [10], the human cost associated with inspecting test output and understanding test outcomes is often overlooked [11].

When database schemas evolve [12], their automatically generated tests should be understandable by humans. Source code understandability is subjective, with developers having different views of automatically generated tests [13]. For example, if testers are deciding whether or not the database will reject a test, some may prefer English-like strings, while others may appreciate simple values such as empty strings. Yet, since understandable test inputs support human comprehension of test outcomes and may expedite the process of finding and fixing faults [14], it is critical to identify the general-purpose characteristics of understandable database schema tests.

With the goal of identifying the factors that make SQL tests understandable for human testers, this paper uses several automated test data generation methods to create tests for database schemas. We place these techniques into four categories according to the data that they generate: (1) random values; (2) default values that use empty strings for characters and constants for numeric values; (3) values from a language model used by Afshan et al. [15], combined with a search-based technique, Alternating Variable Method (AVM); and (4) reused values derived from either column names or a library of readable values. To evaluate the understandability of the data generated by these techniques we conducted a human study. The human participants in this experiment were tasked with explaining test outcomes for data arising from the five data generators. The study's participants were asked to identify which INSERT statement, if any, would be rejected by the database because it violated a schema's integrity constraint.

This paper highlights two key findings. The first is that the data values in INSERTs influence human understandability:

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

(a) The *BrowserCookies* relational database schema

AVM-D

```
1)  INSERT INTO places(host, path, title, visit_count, fav_icon_url)
    VALUES ('', '', '', 0, '')
2)  INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)
    VALUES (0, '', '', 0, 0, 0, '', '')
3)  INSERT INTO places(host, path, title,visit_count, fav_icon_url)
    VALUES ('a', '', '', 0, '')
4)  INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)
    VALUES (1, '', '', 0, 0, 0, '', '')
```

DOM-RND

```
1)  INSERT INTO places(host, path, title, visit_count, fav_icon_url)
    VALUES ('xuksiu', 'fwkjy', 'bmmniu', -53, 'f')
2)  INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)
    VALUES (0, 'iywt', 'ryl', 0, -357, -877, 'xuksiu', 'fwkjy')
3)  INSERT INTO places(host, path, title,visit_count, fav_icon_url)
    VALUES ('lmm', 'j', 'w', 907, NULL)
4)  INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)
    VALUES (131, 'iywt', 'mdofmfl', NULL, NULL, 106, 'xuksiu', 'fwkjy')
```

(b) Automatically generated test cases using AVM-D and DOM-RND that violates a `UNIQUE` constraint,

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

using default values for elements not involved in the test — but necessary for adhering to SQL's syntax rules — aided participants, allowing them to easily identify and understand the important values. Yet, negative numbers and "garbage" strings hindered a human's ability to reason about the rejection of `INSERT` statements. The second finding is more far reaching and in confirmation of prevailing wisdom among database developers: humans found the outcome of tests very difficult to predict when `NULL` was used in conjunction with foreign keys and `CHECK` constraints. Even though `NULL`s limit test understandability for humans, this result suggests that `NULL` use in tests can surface the confusing semantics of database schemas. Overall, this paper makes the following contributions:

1) A human study that assesses the understandability of automatically generated test data by using a realistic task in which participants must determine which `INSERT`, if any, would be rejected by a relational database (Sections III – IV).

2) Readability guidelines for schema tests, derived from quantitative and qualitative feedback from industrial and academic experts in the human study, directing both manual testers and creators of automated testing tools (Sections V – VII).

II. Background and Motivation

This section introduces the state-of-the-art methods for automatically generating test data for relational database schemas. It also explains the challenges often confronting humans when they attempt to understand and maintain this type of test data.

Relational Database Schemas. A relational database schema describes how data is structured when it is stored in a database. Including tables with columns that are associated with data types and integrity constraints that afford data protection, schemas are normally specified by `CREATE TABLE` statements, like the ones in Figure 1(a). For instance, the `PRIMARY KEY` in the `cookies` table of Figure 1(a) requires that each record must have a unique `id` attribute, while the `NOT NULL` constraint for `id` and `name` stipulates that these attributes cannot store a `NULL` value. This table also features a `UNIQUE` constraint on the `name`, `host`, and `path` columns, stipulating that no rows inserted into the database can have exactly the same values for these three attributes. Furthermore, the table has two `CHECK` constraints that restrict the range of values that `expiry`, `last_accessed`, and `creation_time` can store. This schema also illustrates how a `FOREIGN KEY` on the `host` and `path` columns of `cookies` links the records of that table to those in the `places` table.

Test Case Generation for Database Schemas. Integrity constraints preserve the consistency and validity of the data in a database [16]. Since a database administrator may either incorrectly specify or omit an integrity constraint [10], it is important to use coverage criteria to guide their systematic testing [4]. Intuitively, testing an integrity constraint involves exercising it as *true* (i.e., *satisfying* the constraint) and *false* (i.e., *violating* the constraint). For example, if testers want to manually ensure the correctness of the `UNIQUE` constraint in the *BrowserCookies* schema, then assuming an initially blank database, they should `INSERT` a row of valid data into the `cookies` table. To assess the correctness of the `UNIQUE` they would then try to insert a row of data where a value for at least one of the three columns in the `UNIQUE` constraint is distinct from those previously entered. To check that the schema correctly rejects invalid data, they should also attempt to insert a row of data in which the values are the same as those already entered for those columns, as shown in Fig 1(b).

Since it may be challenging for testers to identify attribute values that are tailored to the constraints in a schema, test data generation techniques support this process by automatically creating tests that aim to cover all the integrity constraints. For schemas, there are many techniques for test data generation, such as Random$^+$ [10], the Alternating Variable Method (AVM) [4], [10], and DOMINO [6], [17], with prior work showing that AVM and DOMINO are often the most effective [6].

The AVM is a local search technique that uses guidance from a fitness function to generate the tests that cover the integrity constraints for a given schema [10]. Leveraging test adequacy criteria, formulations of distance functions, and different ways to restart, this method optimizes a vector of test values that will appear in an `INSERT` statement [4]. Operating with the same goal as the AVM, DOMINO is a random technique that uses domain-specific operators to generate test data for schema testing. After creating random values, it uses operators that perform value copying and randomization, the setting or removal of NULLs, and the solving of CHECK constraints [6]. The copying operator helps to match values depending on the coverage criteria (e.g., it copies values to violate a UNIQUE constraint or to link a FOREIGN KEY). The randomizing operator aids when DOMINO must generate distinct values to satisfy the coverage criteria. When solving CHECKs, DOMINO uses randomization to select values from a constant pool containing values mined from the schema [6].

Figure 1(b) gives examples of tests, produced by both aforementioned techniques, that violate the UNIQUE constraint of the cookies table. (This example refers to the AVM as "AVM-D" and DOMINO as "DOM-RND" to distinguish them from new variations of these techniques presented in Section III-A.) Both AVM/AVM-D and DOMINO/DOM-RND assume an empty database, building up the sequence of INSERTs required to first populate the database with valid values, so that the constraint can be tested with identical values for the columns focused on by the final INSERT of each test. The sequence of statements also involves inserting data into the places table so that the foreign key of the cookies table is not violated instead of the UNIQUE constraint, which is the ultimate target of this test case.

Automated test data generators can help testers to avoid the tedious and error-prone task of manually writing tests for a database schema. Prior work has shown that automatically generated tests can effectively cover the schema and detect synthetic schema faults [4], [18], [19]. Yet, testers must still act as an "oracle" for a test when they judge whether it passed or failed [14], a challenging task that is often overlooked.

Human Oracle Costs. The effort expended by a human acting as an oracle for a test suite — that is, understanding each test case and its outcomes, reasoning about whether a test should pass or fail and whether the observed behavior is correct or otherwise — is referred to as the "human oracle cost" [20]. Human oracle costs are either quantitative or qualitative. It is possible to decrease the quantitative costs by, for instance, reducing the number of tests in a suite or the length of the individual tests. Strategies to reduce the qualitative costs often involve modifying the test data generators so that they create values that are more meaningful to human testers [15], [21]. With the ultimate goal of reducing human oracle costs, this paper identifies the factors that influence test understandability.

Test Understandability Factors. Although human oracle costs can be ameliorated by creating automated test data generation methods that consider readability (e.g., [13], [22]), to the best of our knowledge there is no prior work aiming to characterize and limit the qualitative human oracle costs associated with the automated testing of a database schema. As a first step, we must determine how generated test data affects a human's understanding of a test's behavior. Thus, before focusing on generating tests that limit human oracle costs, it is prudent to identify the characteristics that make test cases easy for testers to understand and reason about. This paper reports on a human study performing this important task.

As an example, even though each of the tests in Figure 1(b) successfully violate the intended UNIQUE, they employ different values because they were created with the two previously described automated test data generation techniques. Depending on the generated test data, it may be more or less challenging for a tester to effectively reason about test outcomes [14] and determine whether or not the tests achieved the goal of creating inputs that do not satisfy an integrity constraint. For instance, the second and fourth INSERT statements from AVM-D assign empty strings for the values of the UNIQUE constraint, while the second and fourth INSERTs from the DOM-RND technique use randomly generated strings. Since every data generator works differently, each created test may have varying values — all of which may differ in their human understandability and support of effective testing — for both those attributes involved in testing an integrity constraint and the other schema attributes.

Knowing that the readability of test inputs influences test case understandability [14], we created variants of AVM and DOMINO that generate more readable data values. This enables us to characterize the factors involved in the comprehension of tests for relational database schemas, which is the focus of this paper's study, the design of which the next section describes.

III. METHODOLOGY

In order to act as a human oracle, testers must understand the behavior of a test. The aim of our study, therefore, was to find out what properties of relational schema tests, comprising SQL INSERTs, make them easy for humans to understand.

We studied five different ways to automatically generate tests, based on the two main techniques, AVM and DOMINO, as introduced in the last section. Each technique embodies a different strategy for producing the test inputs (i.e., the values within the INSERTS) that may affect the human comprehension of those tests. These involve the use of default values, random values, pre-prepared data such as dictionary words, or data specifically generated to have English-like qualities.

A. Automated Test Case Generation Techniques

We first introduce each automated method with example test cases for the *NistWeather* schema shown in Figure 2. The test cases generated by each method, featured in part (b) of this figure, aim to satisfy the CHECK constraint on the MONTH column of the Stats table, starting from an initially empty database. In order to insert a valid row in the Stats table, a row must first be inserted into the Station table, thereby ensuring that the foreign key declared in the Stats table is not violated. Thus, each test case consists of two INSERT statements.

The first two test data generators, AVM-D and AVM-LM, are based on the alternating variable method from Section II.

AVM-D is a version of the AVM that starts by initializing each data value to a default (e.g., a zero for a numeric type or empty string for a string type). AVM-D was chosen for this study as it has featured in a number of prior papers devoted to testing relational database integrity constraints (e.g., [4], [10]). An example test case generated by AVM-D is shown in Figure 2(b). The default values — empty strings and zeros

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

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

(a) The *NistWeather* relational database schema.

AVM-D	1)	`INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W) VALUES (0, '', '', 0, 0);`
	2)	`INSERT INTO Stats(ID, MONTH, TEMP_F, RAIN_I) VALUES (0, 1, 127, 0);`
AVM-LM	1)	`INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W) VALUES (100, 'Thino', 'jo', 0, 0);`
	2)	`INSERT INTO Stats(ID, MONTH, TEMP_F, RAIN_I) VALUES (100, 6, 127, 1);`
DOM-RND	1)	`INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W) VALUES (100, 'ivjyv', 'jr', 0, 0);`
	2)	`INSERT INTO Stats(ID, MONTH, TEMP_F, RAIN_I) VALUES (100, 12, 90, 40);`
DOM-COL	1)	`INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W) VALUES (100, 'CITY_0', 'ST', 2, 0);`
	2)	`INSERT INTO Stats(ID, MONTH, TEMP_F, RAIN_I) VALUES (100, 12, 90, 1);`
DOM-READ	1)	`INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W) VALUES (100, 'sidekick', 'ba', 90, 150);`
	2)	`INSERT INTO Stats(ID, MONTH, TEMP_F, RAIN_I) VALUES (100, 12, 80, 12);`

(b) Generated test cases with multiple techniques that satisfies a `CHECK` constraint for column `MONTH`.

Fig. 2. The *NistWeather* relational database schema with examples of automatically generated test case data.

— are shown in each `INSERT` statement and are used when AVM-D did not need to modify the data values to fulfill the test requirement. The values of `1` and `127` are needed to satisfy the `CHECK` constraints on `MONTH` and `TEMP_F`, respectively.

AVM-LM is the basic AVM algorithm described in Section II but with an additional post-processing step. Following the generation of data using the AVM (this time, starting with random, rather than default values), the strings in a test case are optimized for "English-likeness" using a language model, similar to that employed by Afshan et al. [15]. This method replaces every instance of a string in each `INSERT` statement of a test case with a new string generated using the language model. The algorithm generates 10,000 strings of the same length and picks the one with the best language model score. We included AVM-LM because, in Afshan et al.'s study of automated test data generation for C programs, the incorporation of a language model as an extra fitness component in the search-based method helped to produce more readable strings that made tests easier and quicker for human testers to understand [15]. Figure 2(b) shows an example of a test case generated with AVM-LM. The test case does not use default values, but rather starts with a sequence of data values that are either randomly generated or randomly selected from constants used in the schema itself. This method creates English-like words for test strings (i.e., "`Thino`" and "`jo`").

The next three methods, DOM-RND, DOM-COL, and DOM-READ are variants of the DOMINO from Section II.

DOM-RND is the basic form of DOMINO, which a prior study found to obtain the highest mutation scores out of all studied testing methods [6]. Figure 2(b) gives an example of a test in which this method generated all values randomly or randomly selected from constants mined from the schema.

DOM-COL is a variant of DOMINO that, instead of using a randomly generated value for a string, uses the value's associated column name with a sequential integer suffix. The motivation behind DOM-COL is the intuition that, if a data value embodies the column name, testers should easily match data values in an `INSERT` with their columns. Since this is only viable with strings, for integer data DOM-COL attempts to use sequentially generated integers instead of random values. DOM-COL's example test in Figure 2(b) shows how "`CITY_0`" is used as one of the values. Since the `STATE` column has a two character limit, the chosen value is a random subsequence of the column name, which here is the first two characters.

DOM-READ is another variant of DOMINO that selects values from a database that is used separately from the testing process and is populated for the schema. The motivation for this customization of DOMINO is similar to that of AVM-LM: readable values from an existing database should make test data values easier to follow in the `INSERT` statements that contain them. For the purposes of this study, we made the populated databases with a Java library called DataFactory [23], which fills string fields with English words. The test for DOM-READ in Figure 2(b) is similar to that of DOM-RND's except that it features either English words or word-like subsequences for length-constrained fields (i.e., "`sidekick`" and "`ba`").

While AVM-D and DOM-RND have previously appeared in the literature [4], [6], AVM-LM, DOM-COL, and DOM-READ are new techniques designed for this study of test input comprehension. We do not include a purely random test data generator because its values are, from an understandability perspective, nearly identical to those generated by DOM-RND.

B. Measuring Comprehension

Program comprehension, the task of reading and understanding programs, is a complex cognitive task [24]. It often involves understanding a system's behavior through the development of either general-purpose or application-specific software knowledge [25]. This paper uses multiple-choice questions to measure this human knowledge and identify comprehension factors. While some studies use multiple-choice questions to assess problem-solving skill [26], others report that performance on a multiple-choice quiz correlates with knowledge of a written text [27]. In comparison to open-ended short-answer essays, multiple-choice questions are normally more reliable because they constrain the responses [28]. Overall, this prior work shows that multiple-choice questions can surface a human's understanding and problem-solving skills.

C. Research Questions

With the goal of identifying the factors that make SQL test cases understandable, we designed a human study to focus on answering the following two research questions:

RQ1: Success Rate in Comprehending the Test Cases. How successful are testers at correctly comprehending the behavior of schema test cases generated by automated techniques?

RQ2: Factors Involved in Test Case Comprehension. What are the factors of automatically generated SQL `INSERT` statements that make them easy for testers to understand?

D. Experimental Set-up

Schemas and Generators. To generate tests we used the publicly available *SchemaAnalyst* tool [5], which already provides an implementation of the AVM-D and DOM-RND techniques for database schema testing. We added DOM-COL and DOM-READ (and their value-initializing libraries) and AVM-LM (and its language model) to *SchemaAnalyst*, making the enhanced tool, as shown in Figure 3, available for download at https://github.com/schemaanalyst/schemaanalyst. Using *SchemaAnalyst*, we generated tests for the schemas from Section II (i.e., *BrowserCookies* in Figure 1 and *NistWeather* in Figure 2), applying each of the five test generation techniques. We chose these database schemas because, taken together, they have the five main types of integrity constraint (i.e., primary keys, foreign keys, `CHECK`, `NOT NULL`, and `UNIQUE`) and different data types (e.g., integers, text, and constrained strings).

Test Cases. We configured *SchemaAnalyst* to generate test suites by fulfilling a coverage criterion that produces tests that exercise each integrity constraint of the schema with `INSERT` statements that are (a) accepted, because the test data in the `INSERT` statements *satisfies* the integrity constraint along with any other constraints that co-exist in the same table, and (b) contains an `INSERT` statement that is rejected, because test data in it *violates* the integrity constraint (while satisfying all other constraints) [4]. We selected one example of a test that satisfies each different type of integrity constraint (e.g., primary keys and foreign keys) and one example of a test case that violates each type of integrity constraint for each relational schema.

When there were multiple test cases to choose from (because, for example, the schema involves multiple `CHECK` constraints), we selected one at random. *BrowserCookies* involves at least one of each of the main five types of integrity constraint, while *NistWeather* involves all the main types of integrity constraint except a `UNIQUE`. As such, the set of test cases used for the questionnaire consisted of ten test cases for *BrowserCookies* and a further eight for *NistWeather* — to satisfy and violate each of the integrity constraint types — generated by each of the five techniques, resulting in a total of 90 test cases overall. We configured *SchemaAnalyst* to generate test cases suitable for database schemas hosted by the PostgreSQL DBMS. We chose PostgreSQL as its behavior is generally accepted as closest to the SQL standard [29], [30].

Each test case starts from a blank relational database, building up the state needed to test a constraint through a series of initial `INSERT`s. For instance, the adequate testing of a primary key involves an `INSERT` adding at least one row into a table; to test a foreign key, data must be inserted into the referenced table. We then incorporated the generated test cases in a comprehension task delivered by a web-based questionnaire system, as further described in Section III-E.

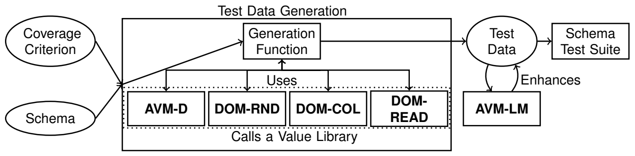


Fig. 3. The inputs and outputs of the enhanced *SchemaAnalyst* tool.

Pilot Trial. The number of questions, test cases, and schemas were carefully chosen using a pilot trial. The trial revealed that when participants were given more than two schemas they got confused and could not remember schema properties, which is not realistic. We also noted that humans completed the tasks in less than an hour when given tests covering all of the integrity constraints in a schema like the ones in Figures 1 and 2.

E. Design of the Human Study

Web-Based Questionnaire Supporting Two Studies. We created a web application to allow human participants to answer questions about the automatically generated test cases. Each question has its own individual web page featuring a specific test. The web page shows the schema for which *SchemaAnalyst* generated the test case at the top of the page, with the `INSERT` statements making up the test underneath.

The questionnaire then required the participants to select the first `INSERT` statement of the test case, if any, that is rejected by the DBMS because it violates one of more integrity constraints of the schema. If the test is designed to satisfy all of the integrity constraints, none of the `INSERT` statements will fail, whereby participants should select "None of them". The goal is for participants to focus on the test inputs, acting as oracles for these tests that do not have assertion statements. When a participant could not decide on the answer, we also provided an "I don't know" option for them to select, thereby preventing them from having to select a response at random to continue to the next question. Importantly, adding the "I don't know" option helped to prevent guessing from influencing the results.

To answer our RQs, we designed a human study based on this questionnaire. In the first part, referred to as the "silent" study, participants answered the questionnaire under "exam conditions" (i.e., they were not allowed to interrupt other participants or confer). This allowed us to obtain a relatively large set of quantitative data from the questionnaire in a short amount of time. The second part took the form of a "think aloud" study in which we collected more detailed and qualitative information from a smaller number of participants. The participants did not receive the correct answers to any of the questions, which might have influenced their answers to questions involving later test cases. Importantly, this type of mixed design is often used to validate quantitative results [31].

The Silent Study (SS). Designed to answer RQ1, this study involved 25 participants recruited from the student body at the University of Sheffield, studying Computer Science (or a related degree) at either the undergraduate or PhD level.

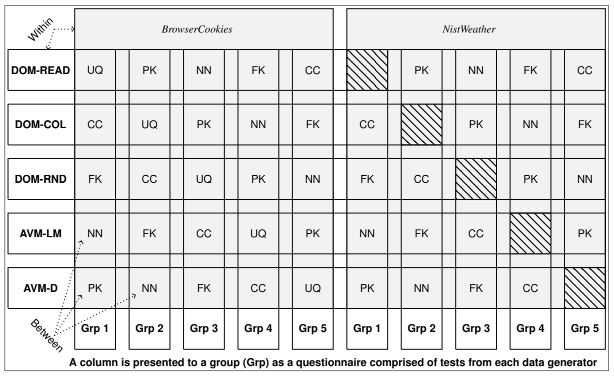
As part of our recruitment and sign-up process, potential participants completed an assessment in which they had to say whether four `INSERT` statements would be accepted or rejected for a table with three constraints. We did not invite anyone to participate if they got more than one answer wrong, ensuring that we included capable participants with adequate SQL knowledge. The web-based questionnaire asked the level of SQL experience of each participant, which varied between less than a year for nine participants to over five years for two. We designed this quiz to focus on the understandability of test inputs and not the understandability of basic SQL commands.

We assigned each participant to one of five groups randomly, such that there were five participants in each group. The study had two within-subject variables (i.e., the database schemas and the test case generation techniques) and one between-subject variable (i.e., the specific test cases themselves) [32], as shown in Figure 4. That is, all groups answered questions involving an adequate test case created by each test generation technique for each of the two schemas and a specific integrity constraint. We assigned a test made by a generator to precisely one group, resulting in five responses per test. Since each cell in Figure 4 represents a separate test for satisfying and violating each constraint, this means that there were 450 data points in total, with 250 for *BrowserCookies* and 200 for *NistWeather*. Although we added two questions at the start of a question set so that participants could practice and get familiar with each schema, we did not analyze the responses to these questions. Each participant was financially compensated with £10, encouraging them to do their best to understand the schema tests and complete the questionnaire in under an hour.

The Think Aloud Study (TAS). We designed this study to answer RQ2, recruiting five new individuals to complete the questionnaire, assigning each to their own group and allowing full coverage of the questions in the questionnaire. Participants were asked to say their thought processes aloud, a technique commonly used in the HCI research community for studying human cognitive processes in problem-solving tasks [33]. This protocol allows for the inferences, reasons, and decisions made by participants to be surfaced when they complete an assignment [34]. The first author performed this study, prompting participants to say why they had chosen an answer if they had not already verbalized their reasoning.

The first author made an audio recording of each participant's session, manually transcribing it to text afterwards. Following this, we analyzed all of these statements. When at least three of the five participants said the same thing, this paper reports it as a "key observation" in the answer to RQ2.

The five participants comprised three additional Computer Science PhD students from the University of Sheffield and two industry participants who each had two years of experience. With these five participants, we restricted ourselves to prompting them with a "why?" question to get them to reveal their thought processes, without any further interactions. We also recruited a sixth participant, with whom we performed the TAS in a randomly assigned group. In contrast to the first five par-


Within
Between

	BrowserCookies					*NistWeather*				
DOM-READ	UQ	PK	NN	FK	CC	▨	PK	NN	FK	CC
DOM-COL	CC	UQ	PK	NN	FK	CC	▨	PK	NN	FK
DOM-RND	FK	CC	UQ	PK	NN	FK	CC	▨	PK	NN
AVM-LM	NN	FK	CC	UQ	PK	NN	FK	CC	▨	PK
AVM-D	PK	NN	FK	CC	UQ	PK	NN	FK	CC	▨
	Grp 1	Grp 2	Grp 3	Grp 4	Grp 5	Grp 1	Grp 2	Grp 3	Grp 4	Grp 5

A column is presented to a group (Grp) as a questionnaire comprised of tests from each data generator

Fig. 4. The mixed study design with two within-subjects variables (i.e., a schema and a data generator) and one between-subjects variable (i.e., a test case). Each test case represents a question and is denoted by an integrity constraint's test (i.e., a Primary Key ("PK"), Foreign Key ("FK"), `UNIQUE` constraint (UQ), `NOT NULL` constraint (NN), or `CHECK` constraint (CC)). The hashed box shows that this schema did not have a `UNIQUE` test.

ticipants, we probed the sixth participant with direct questions inspired by comments that others made. This sixth participant was a developer from a large multi-national corporation with over 10 years of software development experience, including with the SQL. As such, we refer to him in the answer to RQ2 as the "experienced industry engineer". We do not count him among the official TAS participants. Instead, our answer to RQ2 uses the expert as an additional source of comments that we report alongside those from the first five participants.

F. Threats to Validity

External Validity. The selection of schemas for this paper's study is a validity threat because those chosen may yield results that are not be evident for real schemas. To mitigate this threat we picked two schemas that feature all of the integrity constraints and data types commonly evident in schemas [12]. Since they may not represent those often used in practice, the tests used in the study are also a validity threat. To address this matter, we used an open-source automated test data generation tool, *SchemaAnalyst* [5], and configured it to create effective tests according to a recommended adequacy criterion [4]. This decision guaranteed that the study's participants considered tests that can exercise all of a schema's integrity constraints as both *true* and *false*. The use of a small number of relational schemas and tests is also a validity threat. It is worth noting that we purposefully limited the number of these artifacts to ensure that participants could complete the questionnaire in a reasonable amount of time, thereby mitigating the potentially negative effects of fatigue. Since no previous human studies have been done in this area, we began with a small-scale experiment using a small number of participants. Given the relatively small number of total data points, we used a statistical power calculation to see the percentage chance of detecting differences in the human responses to the questionnaire.

Internal Validity. The potential for a learning effect is a validity threat that could arise when participants become better at answering questions as the questionnaire progresses, due to their experience with prior tasks. We mitigated this threat by randomizing the presentation order for questions and schemas.

TABLE I
CORRECT AND INCORRECT ANSWERS FOR THE SILENT STUDY

Technique	Correct Responses	Incorrect Responses	Percentage Correct	Rank
AVM-D	76	14	84%	1
AVM-LM	65	25	72%	=3
DOM-COL	67	23	74%	2
DOM-RND	55	35	61%	5
DOM-READ	65	25	72%	=3

The "think aloud" (TAS) experiment also had threats that we attempted to mitigate. To ensure that all study participants had a uniform experience, the people in the TAS had to abide by a restricted form of interaction with the first author, ensuring that they did not inappropriately discover facets of the comprehension task. Since participants in a think aloud may be naturally reluctant to verbalize their thought process, we instructed each person to "stream" their thoughts during their completion of the questionnaire. Another potential validity threat is that the majority of the participants in the studies were students. However, the TAS included two industrialists and an expert who had technique rankings that were similar to those arising from the silent studies with the students. This trend suggests that it is acceptable to use students to identify the factors that make SQL tests understandable, in broad confirmation of prior results in software engineering [35].

Construct Validity. The measurement of a subjective concept like understandability is also a validity threat. To assess test understandability, we determined how successful human testers were at identifying which `INSERT` statement, if any, would be rejected by the database because it violated an integrity constraint — a viable proxy to understandability that we could accurately calculate. Yet, a study of this nature raises other concerns since participants might not be accustomed to using the questionnaire application to determine the outcome of a SQL test case. It is also possible that testers might have better knowledge of a database schema that they designed. To overcome both of these final concerns, the study included two practice questions with responses that were not recorded.

IV. ANSWERS TO THE RESEARCH QUESTIONS

RQ1: Success Rate in Comprehending the Test Cases. Table I shows the number of correct and incorrect responses for RQ1. A response is correct if a participant successfully selected the first `INSERT` that was rejected by the DBMS, or the "None" option, if all the `INSERT` statements are accepted. The "I do not know" option was not selected by participants in response to any of the questions in the silent study (SS).

Tests generated by AVM-D were most easily comprehended: participants correctly responded 84% of the time. Conversely, tests produced by DOM-RND were the most misunderstood: participants only correctly responded 61% of the time for this method. AVM-LM, DOM-COL, and DOM-READ, which all employ operations to produce more readable strings, achieved similar numbers of correct responses between 72 and 74%.

We performed a Fisher Exact test on the results of each pair of techniques, which revealed a statistically significant difference between AVM-D and DOM-RND, with a p-value < 0.001. However, at the same alpha-level of 0.05, there were no statistically significant differences between the other techniques. We also performed a post-hoc test called "Power of Fisher's Exact Test for Comparing Proportions" to compute the statistical power of Fisher's Exact test [36]. This test shows that, with 90 responses each for DOM-RND and AVM-D, there will be a 93% chance of detecting a significant difference at the 0.05 significance level, assuming that the response score is 84% and 61% for AVM-D and DOM-RND, respectively. For the other test data generators, a post-hoc test calculates that there is a 50% or less chance of detecting a significant difference, suggesting the need for more human participants.

Figure 5 shows the numbers of correct and incorrect responses for each test case. This plot reveals that participants had particular trouble with DOM-RND and identifying test cases where there was no rejected `INSERT` statement for the *BrowserCookies* schema, as shown in the figure by the bars labeled with the "BC-S-" prefix. These are test cases designed to exercise an integrity constraint such that all data in the `INSERT` statements is successfully entered into the database. All of the questions involving these test cases were answered incorrectly for DOM-RND. Similarly, participants struggled with these types of test cases for AVM-LM, DOM-COL, and DOM-READ: they correctly answered 5, 9, and 6 questions out of 25, respectively. However, for AVM-D, participants did not encounter the same issues, answering 18 out of 25 questions correctly. The ratio of correct/incorrect answers is more or less similarly evenly distributed for other test types, although even for these remaining types of tests, DOM-RND remains the weakest performer in terms of correct responses.

In conclusion for RQ1, the SS showed that participants seem to most easily comprehend the behavior of the test cases generated by AVM-D, as evidenced by the fact that they answered the most questions correctly for test cases generated by this technique. In contrast, the most difficult test cases to understand were those generated by DOM-RND. The other techniques, that fall in between these two extremes, have a similar influence on the human comprehension of schema tests.

We designed the TAS with the aim of finding out more about these potential differences in the minds of the human participants, the results of which we discuss next.

RQ2: Factors Involved in Test Case Comprehension. The TAS resulted in fewer overall responses as there were only five participants. Yet, Table II shows the recorded answers follow a similar pattern to those given by participants for RQ1: AVM-D produces tests that are understood the best, with DOM-RND the worst, AVM-LM and DOM-COL falling between the two, and DOM-READ tying AVM-D in this study. The main purpose of the TAS was to surface what participants thought about the tests for which they answered questions. There were seven key observations (KOs) made by three or more of the five participants, each of which we discuss next.

■ *Confusing Behavior of Foreign Keys* (KO1) *and* `CHECK` *Constraints* (KO2) *with* `NULL`. When `NULL` is used on columns without `NOT NULL` constraints but with other integrity constraints,

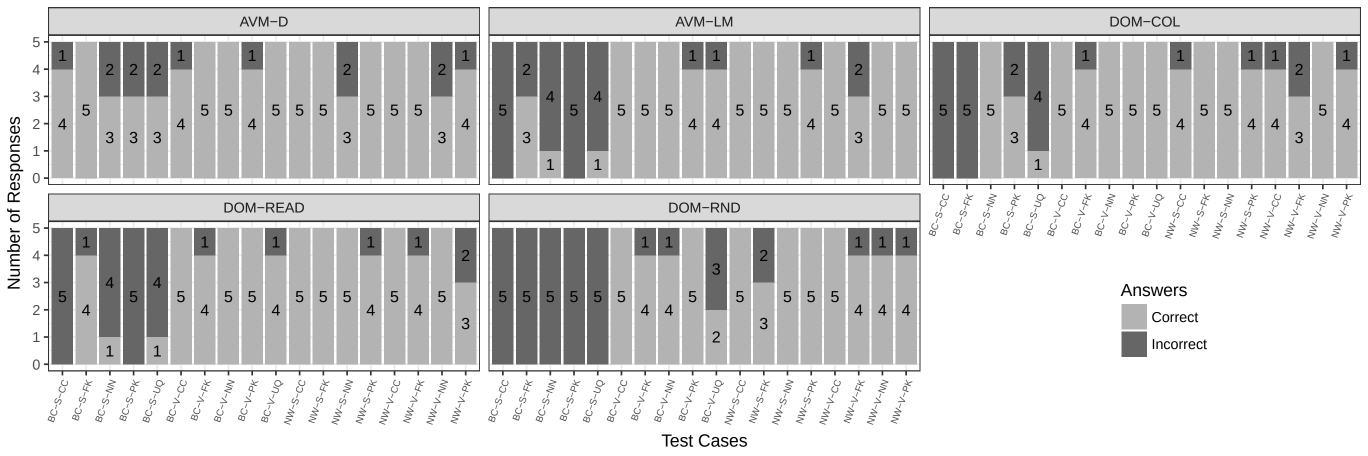


Fig. 5. A stacked bar plot that shows the count of correct and incorrect responses for each automated test generation technique. Each stacked bar corresponds to a specific test case. The horizontal-axis labels designate which schema the test case was generated for (either *BrowserCookies*, denoted by the "BC" prefix, or *NistWeather*, denoted by the "NW" prefix). These are suffixed by the test case type – either the satisfaction ("S") or violation ("V") of a specific integrity constraint (a primary key ("PK"), foreign key ("FK"), UNIQUE constraint (UQ), NOT NULL constraint (NN), or CHECK constraint (CC)).

participants tended to think that the INSERT statement should be rejected. All five stated this for foreign keys, while four commented they thought this was true of CHECK constraints. Yet, this is not the behavior defined by the SQL standard [37].

One participant admitted that they "think it is easier to just look at the ones that have a NULL to see if they are rejected first". While it was easy for the participants to spot NULLs, they found it confusing to judge how they would behave when interacting with the schema's other integrity constraints. For example, one participant stated that "the path [a FOREIGN KEY column in the *BrowserCookies* schema] is NULL which is not going to work, so I will stop thinking there and judge [INSERT statement] four to be the faulty statement." Another participant said that a "CHECK constraint should be a NOT NULL by default" even when the constraint involved columns that could be NULL.

The experienced industry engineer stated the following when he encountered a NULL on a FOREIGN KEY column: "the schema does not allow it" and on another question that "it should fail the FOREIGN KEY because of the NULL [in one of the compound foreign key columns] and the fact that value does not exist [the other foreign key column value in the referenced table]". He also debated with himself on the issue of whether CHECK constraints should not allow a NULL as he was "not sure about the boolean logic around NULLs— I do not think NULL is equal to zero and I do not think NULL is greater than NULL". He asked himself "can I treat a NULL as zero?". After answering the question with "I do not know", we asked him "Do you think NULLs in CHECK constraints are a bit confusing?". He answered "Yes, I am very wary with NULL". After completing the survey, he made the following observation: "In a work situation, I would have looked up how NULL is interpreted in a logical constraint. I did not find them hard to read but I do not know how the DBMS is going to interpret a NULL".

To conclude this KO, NULL is confusing for testers, and the frequency of its use in tests is a factor for comprehension.

■ *Negative Numbers Require More Comprehension Effort When Used in* CHECK *Constraints* (KO3). Negative numbers confused four participants when the column is numeric and used within a boolean logic of a CHECK. Participants repetitively checked negative numbers when they were compared together. A participant reported that negative numbers were more difficult than positive numbers because "it takes more time to do mental arithmetic" when they are in comparisons. Another participant said negative numbers "are not realistic".

The experienced industry engineer also commented on negative numbers when we prompted him after answering a survey question with them. He stated "they are harder, slightly, to think about but it is OK and I can reason about them". For negative numbers with primary keys he said: "It feels that you would not use a negative value on a primary key".

To conclude this KO, the use of negative numbers increases the comprehension effort for database schema test cases.

■ *Randomly Generated Strings Require More Comprehension Effort to Compare* (KO4). Four think-aloud participants said that randomly-generated strings are harder to work with than readable or empty strings. One participant referred to such strings as "garbage data". They went on to say that random strings are "harder when you are thinking of primary and foreign key [string columns], as you had to combine them, and there will be one letter difference, and it will be easier if it is real words". In particular, DOM-RND generates random string values, as shown in Figures 1(b) and 2(b). Comparing the similarity of values that have small differences requires more attention. One participant stated that small differences with characters are "trickier" when trying to review duplicates and references. After completing the survey, the experienced industry engineer said "the one I liked least is random values" (i.e., data generated by DOM-RND). Of the data generated, he stated "these are horrible, they are more distinct …but they do not mean anything. At least [readable strings], I can understand. But for this I had to compare each character".

Because they are both "more readable" and "pronounceable", participants also preferred non-random strings (e.g., those produced by DOM-COL, DOM-READ, and AVM-LM).

Concluding this KO, humans prefer readable, realistic strings to randomly-generated ones when understanding schema tests.

■ *It is Easy to Identify When* NULL *Violates* NOT NULL *Constraints* (KO5). NULL was confusing for participants when used with foreign keys and CHECK constraints, but as would be expected, their behavior is straightforward to identify when used with NOT NULL constraints. Three participants made this comment. One participant stated after he finished the questions that "the NOT NULL constraints are the easiest to spot [violation of NOT NULL], followed by PRIMARY KEY constraints". Another participant commented on a test case that did not involve NULL: "nothing is NULL, so it is easy to see the ones [INSERT statements] that are NULL to see if they will be rejected".

To conclude this KO, it is clear that NULL has differing effects on test case comprehension, depending on the context in which it appears. When used with NOT NULL constraints, human testers thought that the behavior of a test was obvious.

■ *Empty Strings Look Strange* (KO6)*, But They Are Helpful* (KO7). The AVM-D technique uses empty strings as the initial value for string columns in INSERT statements, only modifying them as required by the goal of the test case, as illustrated by the examples in Figures 1(b) and 2(b).

When a question involving a test case generated by AVM-D was revealed to one of the participants, he said "this is difficult". However, he changed his mind afterward, saying that one could see the "differences and similarities between INSERTs", which helped him to identify parts of the INSERT statements that affected the behavior of the overall test case.

Another participant stated that a test case with default values was "a good one" because "zeros are easy to read". However, when the same participant first encountered empty strings he said that they were "weird". The experienced industry engineer liked empty strings because "they are easy to skip over to get to the important data". Reflecting on test effectiveness, he also said "empty strings are boundary values that need to be tested".

To conclude this KO, empty strings help to denote unimportant data, an crucial cue in SQL test comprehension.

Our answer to RQ2 includes many thought-provoking observations. Participants raised issues concerning the use of NULL (KOs 1, 2 and 5), suggesting its judicious use in test data generation. There were positive comments about default values (KO7), readable strings (KO4), and unenthusiastic comments about negative numbers (KO3) and random strings (KO4). We explore these factors in the subsequent discussion section.

V. Discussion

There are several factors that influence the understanding of automatically generated SQL tests, as evident from the think aloud study. This section investigates the frequency of these factors in the test cases generated by each method, explaining whether they aid or hinder successful test comprehension.

Frequency of NULL. Table III shows the median, mean, and total occurrences of NULL in the 18 test cases generated by each technique for the two schemas used in the study. Test cases generated by AVM-D did not have many NULLs (5 in total) compared to the other techniques, which involved 20 or more occurrences, with 39 for DOM-READ. AVM-D's tests had the highest comprehension rate in the SS: 84% of questions involving them were answered correctly, as shown in the table. Conversely, test generation techniques leading to many occurrences of NULL (e.g., DOM-RND and DOM-READ) had the lowest comprehension rates. The TAS revealed that participants got confused with NULLs on columns involving integrity constraints, but which did not also have NOT NULL constraints defined on them. This suggests two strategies: (1) generate NULLs in these scenarios, helping testers to understand the behavior of NULL in schemas and test edge cases that detect more faults, as reported in prior work [6]; or (2) limit the use of NULL in order to expedite the human oracle process.

TABLE II
Correct and Incorrect Answers for the Think Aloud Study

Technique	Correct Responses	Incorrect Responses	Percentage Correct	Rank
AVM-D	16	2	89%	=1
AVM-LM	14	4	78%	4
DOM-COL	15	3	83%	3
DOM-RND	12	6	67%	5
DOM-READ	16	2	89%	=1

Negative Numbers. Table III shows the median, mean, and total occurrences of negative numbers in test cases generated by each technique. DOM-COL generates numeric values through the use of sequential integers, and therefore did not produce test cases with any negative numbers. AVM-D's test cases only contained two occurrences of negative numbers, while other techniques involved 20 or more occurrences. AVM-D and DOM-COL were two of the best performers in terms of test case comprehension for RQ1, but there is not a significant difference in the number of questions that participants correctly answered between DOM-COL and the other techniques. Therefore, the data gives weak evidence that negative numbers affect test case comprehension; however, negative numbers are important to test boundaries, and as such the decision to include them needs to balance thoroughness of the testing process with human comprehension of test cases.

Repetitious Values. TAS participants commented that the AVM-D's use of many empty strings helped them to identify the important parts of the test case. Critically, the smaller the number of distinct values in a test case, the smaller the amount of information the human had to understand. Table III shows that AVM-D involved the smallest number of distinct values in the test cases generated (e.g., 68), while the number of distinct values for the other techniques was more or less similar (e.g., approximately 200). The frequency of string values follows an inverse pattern. AVM-D's test cases received the highest percentage of correctly comprehended test cases. Once again, the results suggest that repetitious values are a positive factor for the database schema tests. Moreover, unlike the other factors (i.e., the use of NULL or negative numbers), repeating values or using suitable defaults (e.g., empty strings or zero values) for unimportant aspects of a test case may not limit a human tester's ability to understand a schema test's behavior.

TABLE III
TEST CASE FACTORS BY TECHNIQUE, INCLUDING THE PERCENTAGE THAT SILENT STUDY PARTICIPANTS CORRECTLY COMPREHENDED

Technique	**NULLs**			**Negatives**			**Word Frequency**		**Distinct Values**			**Percentage Correct**
	Median	Mean	Total	Median	Mean	Total	Median	Mean	Median	Mean	Total	
AVM-D	0	0.3	5	0	0.1	2	4	5.1	4	3.8	68	84%
AVM-LM	1	1.1	21	1	1.4	25	1	1.2	10.5	12	216	72%
DOM-COL	1.5	1.4	26	0	0	0	1	1.2	10.5	11.3	203	74%
DOM-RND	2	1.5	27	0.5	1.1	19	1	1.1	10	11.3	203	61%
DOM-READ	2	2.7	39	1	1.4	26	1	1.2	10.5	11.3	204	72%

Readable Values. We developed three of our techniques to generate non-random, human-readable strings (e.g., AVM-LM, DOM-COL, and DOM-READ). The results for RQ1 do not suggest that this was the most important factor in test case comprehension. While these techniques did not produce test cases with the highest comprehension rate, they were also not the worst. In the TAS of RQ2, participants agreed that random strings were hard to understand, and therefore preferred readable strings. We asked the experienced industry engineer about the different types of strings produced by AVM-LM, DOM-COL, and DOM-READ. He said the following of DOM-READ: "…easy to compare them because I can read them. [I see] distinct values, but I prefer nouns and adverbs"; of strings generated by the AVM-LM: "nice because they are pronounceable"; of strings generated by DOM-COL: "[values are] easy to correlate" with column names. However, he also stated that DOM-COL should have "visually different words" to help distinguish between different values. Overall, while human-readable values seem helpful, the results suggest that they are not critical to SQL test case comprehension.

The responses to RQ1 and RQ2 and the results in Table III highlight the factors that influence human comprehension of schema tests. The results suggest that the frequency of `NULL`s, existence of negative numbers, repetition of data values, and presence of readable values can influence the understandability of automatically generated tests. This means that both manual testers and the creators of automated testing tools should consider these issues as they may influence whether humans can understand tests and effectively complete testing tasks.

VI. RELATED WORK

Test comprehension is a frequently studied issue. For instance, Li et al. surveyed 212 developers and more than half reported difficulty with understanding unit tests [38]. The studies of Daka et al. [14] and Rojas et al. [39] show that professional testers find automatically generated tests hard to understand and difficult to maintain. Grano et al. reported that manually written tests are more readable than automated tests [40]. Different approaches have been created to address this issue: test visualization [41], [42], automatic test documentation [38], [43], [44], and test readability improvement [14], [15], [22].

Yet, these papers examined test comprehension in the context of traditional programs. While prior work studied human errors in the context of database query languages [45]–[48], to the best of our knowledge this paper is the first to study test comprehension in the domain of database schemas and SQL statements. This is surprising, since there are many prior methods for automatically testing and debugging a database.

Examples of prior work range from testing `SELECT` queries with constraint solvers or specifications (e.g., [49]–[51]) or by using randomized techniques (e.g., [52]–[54]); to testing database functionality in database-centric applications (e.g., [55]–[59]); to automatically populating databases as a testing enabler (e.g., [60]); to testing and debugging database schemas (e.g., [4], [6], [61]–[63]). While these papers often experimentally evaluate the effectiveness of the generated tests and data, they do not, like this paper, investigate whether the constructed tests are understandable for and useful to humans.

This paper also features a think aloud study (TAS), which has been previously employed by software testing researchers. For instance, Rojas et al. used a TAS, featuring five participants, to investigate the usefulness of automated Java testing tools compared to manually engineered tests [39]. Itkonen et al. used a TAS to understand how professional testers write tests manually [64]. While this paper's methodology is similar to these prior papers, its TAS aimed to surface insights from humans who performed database schema testing tasks.

One of the findings of this paper is the confusing nature of `NULL` for testers. This finding is consistent with the fact that the use of `NULL` has been the source of debate and chagrin for DBMS designers as well [65]–[68], leading to defects and inconsistency in the DBMS engines themselves [69], [70].

VII. CONCLUSIONS AND FUTURE WORK

Presenting a study of the factors that make SQL test cases understandable for human testers, this paper reveals that:

1. `NULL` is confusing for testers. Testers find the behavior of `NULL` difficult to predict when used in conjunction with different integrity constraints such as foreign keys and `CHECK` constraints, suggesting the need for their judicious use.

2. Negative numbers require testers to think harder. Testers prefer positive numeric values, although, from a testing perspective, negative numbers should not be avoided altogether.

3. Simple repetitions for unimportant test values help testers. If only the important data values in the test case vary, while all others are held constant, a tester can easily focus on the non-trivial aspects of a test case to understand its behavior.

4. Readable string values. Testers prefer to work with human-readable, rather than randomly generated, string values.

We plan to use these findings as guidelines for the development of new test data generators for database schemas and, when appropriate, traditional programs. Our goal is to develop tools that automatically generate tests containing data values that are both understandable to humans and effective at finding faults.

References

[1] P. Glikman and N. Glady, "What's the value of your data?" 2015. [Online]. Available: https://techcrunch.com/2015/10/13/whats-the-value-of-your-data/

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

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

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

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

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

[7] E. Dustin, T. Garrett, and B. Gauf, *Implementing automated software testing: How to save time and lower costs while raising quality*, 2009.

[8] M. Olan, "Unit testing: Test early, test often," *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, 2003.

[9] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2007.

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

[11] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Proceedings of the 3rd International Workshop on Search-Based Software Testing*, 2010.

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

[13] J. M. Rojas and G. Fraser, "Is search-based unit test generation research stuck in a local optimum?" in *Proceedings of the 10th International Workshop on Search-Based Software Testing*, 2017.

[14] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, 2015.

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

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

[17] A. Alsharif, G. M. Kapfhammer, and P. McMinn, "Generating test suites with DOMINO," in *Proceedings of the 11th International Conference on Software Testing, Verification and Validation – Demonstrations Track*, 2018.

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

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

[20] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *Transactions on Software Engineering*, vol. 41, no. 5, 2015.

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

[22] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proceedings of International Symposium on Software Testing and Analysis*, 2017.

[23] A. Gibson, "Generate test data with DataFactory," 2011. [Online]. Available: https://www.andygibson.net/blog/article/generate-test-data-with-datafactory/comment-page-1/

[24] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and Software*, vol. 7, no. 4, 1987.

[25] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, 1995.

[26] A. A. Rupp, T. Ferne, and H. Choi, "How assessing reading comprehension with multiple-choice questions shapes the construct: A cognitive processing perspective," *Language Testing*, vol. 23, no. 4, 2006.

[27] Y. Ozuru, S. Briner, C. A. Kurby, and D. S. McNamara, "Comparing comprehension measured by multiple-choice and open-ended questions." *Canadian Journal of Experimental Psychology*, vol. 67, no. 3, 2013.

[28] D. R. Bacon, "Assessing learning outcomes: A comparison of multiple-choice and short-answer questions in a marketing context," *Journal of Marketing Education*, vol. 25, no. 1, 2003.

[29] "SQL conformance." [Online]. Available: https://www.postgresql.org/docs/9.5/static/features.html

[30] DigitalOcean, "SQLite vs MySQL vs PostgreSQL: A comparison of relational database management systems," Jul 2017. [Online]. Available: https://goo.gl/mrZSG4

[31] J. W. Creswell, R. Shope, V. L. Plano Clark, and D. O. Green, "How interpretive qualitative research extends mixed methods research," *Research in the Schools*, vol. 13, no. 1, 2006.

[32] G. Charness, U. Gneezy, and M. A. Kuhn, "Experimental methods: Between-subject and within-subject design," *Journal of Economic Behavior & Organization*, vol. 81, no. 1, 2012.

[33] J. Nielsen, T. Clemmensen, and C. Yssing, "Getting access to what goes on in people's heads? Reflections on the think-aloud technique," in *Proceedings of the 2nd Nordic Conference on Human-Computer Interaction*, 2002.

[34] M. W. van Someren, Y. F. Barnard, and J. A. Sandberg, *The think aloud method: A practical approach to modelling cognitive*, 1994.

[35] M. Höst, B. Regnell, and C. Wohlin, "Using students as subjects: A comparative study of students and professionals in lead-time impact assessment," *Empirical Software Engineering*, vol. 5, no. 3, 2000.

[36] Y. Xia, J. Sun, and D.-G. Chen, "Power and sample size calculations for microbiome data," in *Statistical Analysis of Microbiome Data with R*, 2018.

[37] "Database language SQL part 2: Foundation (SQL/foundation)," ANSI/ISO/IEC International Standard – ISO/IEC 9075-2:2011, 2011.

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

[39] J. M. Rojas, G. Fraser, and A. Arcuri, "Automated unit test generation during software development: A controlled experiment and think-aloud observations," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2015.

[40] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *Proctedings of the 26th International Conference on Program Comprehension*, 2018.

[41] B. Cornelissen, A. Van Deursen, L. Moonen, and A. Zaidman, "Visualizing testsuites to aid in software understanding," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, 2007.

[42] A. M. Smith, J. J. Geiger, G. M. Kapfhammer, M. Renieris, and G. E. Marai, "Interactive coverage effectiveness multiplots for evaluating prioritized regression test suites," in *Compendium of the 15th Information Visualization Conference*, 2009.

[43] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, "Documenting database usages and schema constraints in database-centric applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.

[44] B. Li, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Aiding comprehension of unit test cases and test suites with stereotype-based tagging," in *Proceedings of the 26th International Conference on Program Comprehension*, 2018.

[45] A. Ahadi, J. Prior, V. Behbood, and R. Lister, "Students' semantic mistakes in writing seven different types of SQL queries," in *Proceedings of the Conference on Innovation and Technology in Computer Science Education*, 2016.

[46] P. Reisner, "Human factors studies of database query languages: A survey and assessment," *Computing Surveys*, vol. 13, no. 1, 1981.

[47] T. Taipalus, M. Siponen, and T. Vartiainen, “Errors and complications in SQL query formulation,” *Transactions on Computing Education*, vol. 18, no. 3, 2018.

[48] J. B. Smelcer, “User errors in database query composition,” *International Journal of Human-Computer Studies*, vol. 42, no. 4, 1995.

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

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

[51] C. Binnig, D. Kossmann, and E. Lo, “Multi-RQP: Generating Test Databases for the Functional Testing of OLTP Applications,” in *Proceedings of the International Workshop on Testing Database Systems*, 2008.

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

[53] D. R. Slutz, “Massive stochastic testing of SQL,” in *Proceedings of the International Conference on Very Large Data Bases*, 1998.

[54] J. Castelein, M. Aniche, M. Soltani, A. Panichella, and A. van Deursen, “Search-based test data generation for SQL queries,” in *Proceedings of the International Conference on Software Engineering*, 2018.

[55] 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 International Conference on Automated Software Engineering*, 2011.

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

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

[59] Gregory M. Kapfhammer and Mary Lou Soffa, “A family of test adequacy criteria for database-driven applications,” in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2003.

[60] K. Houkjær, K. Torp, and R. Wind, “Simple and realistic data generation,” in *Proceedings of the International Conference on Very Large Data Bases*, 2006.

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

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

[63] 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.

[64] J. Itkonen, M. V. Mantyla, and C. Lassenius, “How do testers do it? An exploratory study on manual testing practices,” in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2009.

[65] “NULL handling in SQLite versus other database engines.” [Online]. Available: https://www.sqlite.org/nulls.html

[66] R. Sheldon, “How to get NULLs horribly wrong in SQL Server,” 2016. [Online]. Available: https://www.red-gate.com/simple-talk/sql/t-sql-programming/how-to-get-nulls-horribly-wrong-in-sql-server/

[67] W. Lam, “The behavior of NULLs in SQL.” [Online]. Available: http://www-cs-students.stanford.edu/~wlam/compsci/sqlnulls

[68] M. Winand, “Modern SQL: NULL — purpose, comparisons, NULL in expressions, mapping to/from NULL.” [Online]. Available: https://modern-sql.com/concept/null

[69] SQLite Tutorial Website, “SQLite primary key: The ultimate guide to primary key.” [Online]. Available: http://www.sqlitetutorial.net/sqlite-primary-key/

[70] S. Stein, G. Milener, C. Guyer, and R. Byham, “Unique Constraints and Check Constraints.” [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/tables/unique-constraints-and-check-constraints?view=sql-server-2017