Yang, J. C. et al. (Eds.) (2018). Proceedings of the 26th International Conference on Computers in Education. Philippines: Asia-Pacific Society for Computers in Education

Semi-Automated Assessment of SQL Schemas via Database Unit Testing

Nigel STANGER
Department of Information Science, University of Otago, New Zealand
nigel.stanger@otago.ac.nz

Abstract: A key skill for students learning relational database concepts is how to design and implement a database schema in SQL. This skill is often tested in an assignment where students derive a schema from a natural language specification. Grading of such assignments can be complex and time consuming, and novice database students often lack the skills to evaluate whether their implementation accurately reflects the specified requirements. In this paper we describe a novel semi-automated system for grading student-created SQL schemas, based on a unit testing model. The system verifies whether a schema conforms to a machine-readable specification and runs in two modes: a staff mode for grading, and a reduced functionality student mode that enables students to check that their schema meets specified minimum requirements. Analysis of student performance over the period this system was in use shows evidence of improved grades as a result of students using the system.

Keywords: Automated assessment, SQL, database schema, data definition language (DDL), student performance, unit testing

1. Introduction

Any introductory database course must cover several core concepts, including logical data models, and how to create and use databases. Such courses typically focus on the relational model and SQL database management systems (DBMSs), because the relational model provides a sound theoretical framework for discussing key concepts (Date, 2009), and because SQL DBMSs are so widely used.

Courses that teach SQL usually include assessments that test students' ability to create databases using SQL data definition (DDL) statements, and to interact with them using SQL data manipulation (DML) statements. Manually evaluating code submitted for such assessments can be slow, tedious, and error-prone, which may impact student grades. Automated or semi-automated grading has been shown to improve turnaround time and consistency, and is generally liked by students (Dekeyser, Raadt, & Lee, 2007; Douce, Livingstone, & Orwell, 2005; Prior & Lister, 2004; Russell & Cumming, 2004). If the grading can be done in real time, the grading tool can form part of a larger, interactive SQL learning environment (e.g., Kenny & Pahl, 2005; Kleiner, Tebbe, & Heine, 2013; Mitrovic, 1998; Russell & Cumming, 2004; Sadiq, Orlowska, Sadiq, & Lin, 2004).

There has been much prior work on automatically or semi-automatically assessing SQL DML (see Section 2), but relatively little work on assessing SQL *DDL*. There are generally two aspects to consider when assessing SQL DDL code. First, is the code syntactically correct? This is already handled effectively by the syntax checkers built into every SQL DBMS. A related aspect is code style (e.g., naming, formatting, indentation), but we do not consider this here.

Second, does the schema meet the requirements of the problem? A database schema is normally designed to meet a specific set of requirements, so verifying that the SQL DDL code fulfills these requirements is an effective way to assess it and to provide feedback to students. The requirements for a database schema can be loosely divided into *structure* (tables, columns, data types), *integrity* (keys, constraints), and *behavior* (sequences, triggers) (Codd, 1981).

In this paper, we describe a novel system that semi-automates the assessment of students' SQL DDL code, and present evidence that introducing the system improved student grades. The system takes as input a machine-readable specification of the assessment requirements and a submitted student schema, then verifies that the schema conforms to the specification. Rather than parsing and checking the code directly, the system verifies the schema's structure by issuing queries against the metadata (catalog) for expected values such as table names, column names, etc. It verifies integrity constraints by attempting to insert known legal and illegal values. The system currently does not check behavioral aspects. The results of these tests are compared against the machine-readable specification, thus effectively unit testing the schema using the specification as a framework. We use the PHPUnit database unit testing framework to achieve this (see Section 4).

In the next section we discuss related work, then discuss the motivation and context for our approach in Section 3. The system design is discussed in Section 4, and its impact is evaluated in Section 5. Finally, we discuss known issues and future work in Section 6, before concluding.

2. Related Work

There have been many prior systems to support students learning SQL. Early examples such as RDBI (Dietrich, 1993) and esql (Kearns, Shead, & Fekete, 1997) were essentially query execution environments that students could use to practice writing SQL queries. RDBI provided relatively little feedback about the correctness of a query, whereas esql could visualize the intermediate tables generated by each step of a query, enabling students to better understand the steps in its execution.

Later systems provided greater feedback to students. Systems like SQL-Tutor (Mitrovic, 1998, 2003), SQLator (Sadiq et al., 2004), AsseSQL (Prior & Lister, 2004), ActiveSQL (Russell & Cumming, 2004, 2005), SQLify (Dekeyser et al., 2007), ACME (Soler, Boada, Prados, Poch, & Fabregat, 2007), and aSQLg (Kleiner et al., 2013) all provided syntactic and semantic feedback. Many took a more "functional" approach to checking SQL query code, i.e., verifying that the code correctly answered the question, rather than focusing on the code itself. This was typically done by measuring the difference between the student's result set and the correct one, e.g., SQLator's "equivalence engine" (Sadiq et al., 2004). ActiveSQL could also detect "hard-coded" queries that produced the correct result, but which would fail if the data set changed (Russell & Cumming, 2005). SQL-LTM (Dollinger & Melville, 2011) used an XML representation of SQL queries to evaluate their logical correctness, while The SQL Exploratorium (Brusilovsky et al., 2010) used parameterized query templates to generate questions for students. Given the more static nature of an SQL schema, we consider this "verification" style of approach to be the most appropriate way to construct an automated assessment system for SQL DDL.

Prior systems mostly focused on SQL *queries* using the `SELECT` statement (i.e., DML) rather than SQL *schema definitions* (DDL). This is unsurprising given that `SELECT` is probably the most frequently used and most complex SQL statement. Few of the systems reviewed even mentioned schema definition. RDBI (Dietrich, 1993) supported its own non-SQL DDL, while esql (Kearns et al., 1997) simply passed anything not a `SELECT` statement directly through to the DBMS. SQL-trainer (Laine, 2001), supported "maintenance operations", but it is unclear whether this means DDL statements or DML operations like `UPDATE` and `DELETE`. Gong's (2015) "CS 121 Automation Tool" focused primarily on SQL DML statements, but appears to be extensible and could thus be modified to support SQL DDL statements. ADVICE (Cvetanović, Radivojević, Blagojević, & Bojović, 2011) and LearnSQL (Abelló et al., 2016, 2008) both supported SQL DDL, but like many prior systems, they focused on exercises involving small snippets of DDL code rather than entire database schemas.

Online SQL courses and tutorials such as SQLBolt (https://sqlbolt.com/), SQLCourse (http://www.sqlcourse.com/), and w3schools (https://www.w3schools.com/sql/) provide interactive DDL exercises, but the feedback from such systems is limited at best. For example, SQLBolt reports any errors as "incomplete query", although it does verify that the student-provided SQL meets the exercise requirements. SQLCourse accepts even syntactically incorrect code without complaint!

Many prior systems implemented some form of automated or semi-automated grading, e.g., SQLator (Sadiq et al., 2004), AsseSQL (Prior & Lister, 2004), ActiveSQL (Russell & Cumming, 2004, 2005), SQLify (Dekeyser et al., 2007), aSQLg (Kleiner et al., 2013), Gong's (2015) "CS 121 Automation Tool", ADVICE (Cvetanović et al., 2011), ACME (Soler et al., 2007), LearnSQL (Abelló et al., 2016), and XDa-TA (Bhangdiya et al., 2015; Chandra et al., 2015; Chandra, Joseph, Radhakrishnan, Acharya, & Sudarshan, 2016). Some (e.g., SQLator, AsseSQL, ADVICE) provided only correct/incorrect responses, while others (e.g., ActiveSQL, SQLify, aSQLg, XDa-TA, LearnSQL) could assign partial credit. Regardless of the grading style, these systems were often able to automatically mark a significant fraction—e.g., over a third for SQLator (Sadiq et al., 2004, p. 227)—of submitted queries as correct, thus reducing teachers' workload.

Only ADVICE and LearnSQL supported automatic grading of SQL DDL statements. Dealing with `CREATE` statements should be simpler than dealing with `SELECT` statements, and the ability to at least semi-automate the grading of SQL DDL code should reap rewards in terms of more consistent application of grading criteria, and faster turnaround time (Dekeyser et al., 2007; Douce et al., 2005; Prior & Lister, 2004; Russell & Cumming, 2004).

Another branch of related work is systems that actively aid students in learning SQL, e.g., SQL-Tutor (Mitrovic, 1998, 2003), SQLator (Sadiq et al., 2004), ActiveSQL (Russell & Cumming, 2004), and aSQLg (Kleiner et al., 2013). SQL-Tutor is typical of this category. It was an intelligent teaching system that provided students with a guided discovery learning environment for SQL queries. Kenny and Pahl (2005) described a similar system that included an assessment of a student's previous progress, enabling a more personalized and adaptive approach to student learning.

There is relatively little prior work on unit testing databases. Most authors have focused on testing database *applications*, not the database itself (e.g., Binnig, Kossmann, & Lo, 2008; Chays, Shahid, & Frankl, 2008; Haller, 2010; Marcozzi, Vanhoof, & Hainaut, 2012). Ambler (2006) discussed how to test a database's functionality, while Farré, Rull, Teniente, & Urpí (2008) described how to test the "correctness" of a schema, focusing on constraint consistency. Neither considered how to verify that a database schema conforms to its original specified requirements. Of the systems reviewed, only LearnSQL (Abelló et al., 2016) used an approach similar to unit testing.

3. Motivation

Since 1989, our department has offered some form of mandatory database concepts coverage. This most often comprised a one semester course during the second year of a three-year Bachelor's degree, building on a short introduction to data management concepts in the first year. Typical of such courses, it covered core topics such as the relational model, relational algebra, data integrity, SQL DDL and DML, and a mixture of other database topics such as transactions, concurrency control, triggers, and security. SQL skills were assessed using a mixture of assignments and tests.

The most common instrument used to assess students' SQL DDL skills was a practical assignment. Students had 3–4 weeks in which to implement a database schema based on a fictional scenario specification. The scenario posed that the student was a junior database developer in a larger project, and the provided specification was the output of the requirements analysis phase. Typical scenarios included product manufacture and sale, student records management, and used car sales.

Prior to 2001 the specifications for assignment scenarios were deliberately somewhat loosely defined and often had under-specified or ambiguous elements, enabling students to explore alternative ways of implementing a conceptual model. This of course led to significant variation across student submissions, due to differing interpretations of the under-specified elements. We therefore made no significant attempt at automated grading during this period.

From 2001 to 2003, we used a practical examination to assess students' SQL DDL skills. The primary motivation for the change was that an assignment might not truly measure a student's individual learning, as they could be assisted by others. In contrast, an examination is tightly controlled, thus providing a more objective assessment of a student's individual learning. Students were given a fictional scenario specification that tended to be tightly specified and thus less open to interpretation. Consequently, the examination was easier to grade than the earlier assignments. However, the examination format constrained the complexity of tasks that could be assessed and students found it quite stressful. We therefore reverted to a practical assignment in 2004.

A significant change from 2004 onwards is that the assignment scenario specifications were tightened up to reduce ambiguity. In 2013, we took the further step of "freezing" the specification, i.e., students were not permitted to arbitrarily change the specification without strong justification, and implementations had to preserve the "interface" presented to client programs. The in-scenario rationale was that other (fictional) developers were, in parallel, using the same specification to code end user applications, as often occurs in real world system development projects (Perry, Siy, & Votta, 2001). Any significant variation from the specification could break those end user applications. This approach meant that students could still exercise flexibility in their schema implementations where appropriate and ensured that the assignment was not just a mechanical translation exercise.

This approach seemed effective, but it was often difficult to maintain consistent grading standards across many submissions (typically about 70) due to the large number of distinct gradable items implied by the specification. This required a complex rubric so that no item was missed, and grading consequently required significant time and mental effort, especially when feedback to students was required. This raised concerns about the consistency of grades and feedback when multiple graders were involved and prompted interest in at least semi-automating the grading of this assignment. Another motivation was that it can be difficult for database novices to know whether they are on the right track while implementing a specification. A limited, student facing version of the assessment tool could be used to provide feedback on their progress before they submitted.

We decided to impose a minimum set of requirements for the assignment: students' SQL code should be syntactically correct and include all tables and columns detailed in the specification, with correct names and appropriate data types. Any student who satisfied these requirements would score at least 50%. They could check conformance with the requirements by submitting their schema through a web application, deployed in 2013. That way we (and they) could be more certain that at least the core of their schema was correct. Teaching staff graded additional aspects of the schema, such as integrity constraints, using a shell application, trialed in 2012 and fully deployed in 2013.

4. System Design

The core function of our system is to check whether a student's schema conforms to the assignment specification, by automatically comparing their submitted schema against a machine-readable version of the specification. This is essentially a unit testing approach, so we built the system around a unit testing framework. An advantage of this compared to prior systems is that we did not need to develop a dedicated "checking" or "verification" module within our system to check the student's code against the specification, as such functionality was already built into the unit testing framework.

There are few frameworks designed specifically to perform unit tests that interact with a database, probably due to the complexities involved. In conventional application unit testing it is simple to create mocked interfaces for testing purposes. With a database, however, we must create tables, populate them with test data, verify the database's state after each test, and reset the database for each new test (Bergmann, 2017). Cleaning up is crucial, as tests are executed in arbitrary order. Tests that alter the database state may affect the results of later tests in unpredictable ways.

We know of four unit testing frameworks that specifically support database unit tests: DbUnit (Java; http://dbunit.sourceforge.net/), DbUnit.NET (http://dbunit-net.sourceforge.net/), Test::DBUnit (Perl; https://metacpan.org/pod/Test::DBUnit), and PHPUnit (https://phpunit.de/). We built the system in PHP to enable quick prototyping and simplify development of the student facing web application. A similar approach could be taken with the other frameworks, however.

Our database teaching was mainly based around Oracle, but the system could be adapted for use with any DBMS supported by PHP's PDO extension. This would require an additional layer to abstract the implementation specific details of each DBMS.

The main engine of our system executes in either *student mode*, which runs only a subset of the available tests (discussed below), or *staff mode*, which runs all available tests. The mode is determined by the client application. The system generates test output in either HTML or plain text.

The assignment specification is encoded as subclasses of PHPUnit's `TestCase` class, one per database table. The methods of these subclasses return various properties of the table, for example, `getTableName` returns the expected name of the table, while `getColumnList` returns an array of column specifications, keyed by expected column name. Each column specification includes a generic data type (text, number, date, or binary), a list of corresponding acceptable SQL data types, whether nulls are permitted, and a known legal value for general testing. It may also include minimum and maximum column lengths (precision for numeric types), and the number of decimal places (scale). Underflow, overflow, and lists of known legal and illegal values can also be specified.

Each table specification also defines two sets of tests. The first verifies the table's structural elements (columns, data types, etc.), thus verifying that it meets the minimum requirements. These tests issue queries against the metadata (catalog) for elements like tables, columns, and data types. An empty data fixture (specified using a separate XML document) is required for this set of tests.

The second set of tests verifies the table's integrity constraints. Primary and foreign keys are verified using metadata queries, while nullability is tested by attempting to insert nulls. `CHECK` constraints are tested by attempting to insert lists of known legal and illegal values, consistent with normal unit testing practice. A known-legal data fixture is required for this set of tests.

The way the system runs the tests is somewhat unusual, in two ways. First, database unit testing frameworks are really designed to test database-backed *applications*, rather than the database itself. Second, in typical unit testing, tests are standalone code units executed in arbitrary order by a *driver*, which resolves dependencies among tests and handles collation of test results internally. Unit testing drivers are typically designed to be called from a specific tool or environment (e.g., the `phpunit` command line tool), rather than to be embedded as a library into arbitrary programs.

Our system effectively inverts this model. The main engine replaces the driver framework, directly creating and executing test suites itself, then listening for and collating the results. This is because we need to control the order in which tests are executed. For example, if the structural tests fail, there is little point in running the integrity tests, as they will only generate a stream of errors. (Note that while PHPUnit supports test groups, dependencies cannot be defined between groups.)

It is possible to add table properties and tests beyond those already supported, as the specification is constrained only by what can be coded in PHP. All a teacher needs to do is add custom properties to the table specification, then add tests that use those properties. Custom tests are registered with the appropriate test set (structure or integrity) using PHPUnit's `@group` annotation.

Students can check their schema by creating the tables in their personal database account, then logging in to the student mode web application, which accesses the student's schema directly. Only the structural tests are run, and the output is displayed in the web browser.

Teachers can further check a student's schema using the staff mode shell application. To ensure a clean testing environment, the teacher loads the student's submitted code into a database account used only for grading purposes, erasing the schema before moving to the next submission. If there are no syntax errors in the code, the shell application connects to the grading account, runs both the structural and integrity tests, and displays the output in the terminal window.

5. Evaluation

Unfortunately, the system was not originally conceived as a research project with formal evaluation in mind; rather it was a practical solution to a perceived teaching issue. We therefore did not carry out any evaluations with students. We were, however, able to analyze how using the system impacted on student performance, as we had extensive historical grade data. We collated data for the period 2009–2016 (there were no 2017 data because the course was discontinued; see Section 6), which encompassed several different permutations of scenario and available system modes, as summarized in Table 1. The assignment counted for 15% of a student's total grade in 2009 and 2010, and 10% in subsequent years. The grade distributions for the assignment in each year are shown in Figure 1.

The horizontal rule between 2011 and 2012 in Table 1 marks a significant reorganization of the course's curriculum and a switch from first semester (March to June) to second semester (July to October). We trialed the first prototype of staff mode in 2012, and deployed student mode in 2013. The horizontal rule between 2013 and 2014 marks a shift back to first semester. The system was not used at all in 2015 due to different staff teaching the course, and student mode was unavailable in 2016 due to technical issues. These variations provide an interesting natural experiment.

The mean assignment grade drifted slowly downwards from 2009 to 2012. This reversed dramatically in 2013, the year we first used student mode. The grades are not normally distributed (see Figure 1), so we used a Mann-Whitney *U* test. The increase in mean grade from 2012 to 2013 was highly significant ($p \approx 10^{-9}$). The 2013 mean was also significantly higher than both 2010 ($p \approx 0.0002$) and 2011 ($p \approx 10^{-6}$), but not significantly higher than 2009. The mean then decreased significantly in 2014 ($p \approx 0.0012$), the second year the system was used, and even more in 2015 ($p \approx 0.0005$), when the system was not used at all. The increase from 2015 to 2016 was not significant.

Even more interesting, if we compare performance between the years that student mode was not used (2009–2012 and 2015–2016, mean 71.6%) and the years it was (2013–2014, mean 81.7%), there is again a highly significant increase in the mean ($p \approx 10^{-8}$). When student mode was available, the lowest grade awarded was 46%, contrasted with a much longer tail of low grades when student mode was not available. This strongly suggests that introducing student mode positively impacted students' ability to complete the assignment more effectively. However, it could also be argued that this was merely a consequence of imposing minimum requirements, which we will consider shortly.

Table 1

Historical Characteristics of the Database Implementation Assignment, 2009–2016

Year	Class size	Median GPA[*]	Mean (%)	Scenario	Modes used
2009	46	–	77.5	"postgrad"	–
2010	68	3.4	73.4	"student records"	–
2011	64	3.9	71.8	"used cars"	–
2012	75	3.4	69.2	"manufacturer"	staff
2013	77	3.2	84.3	"student records"	both
2014	49	3.4	77.6	"used cars"	both
2015	71	3.0	69.2	"used cars"	neither
2016	75	3.5	71.0	"manufacturer"	staff

[*] On a 9-point scale where C− = 1, A+ = 9. Value is across all enrolled papers for the specified year only. Data for 2009 were not available.

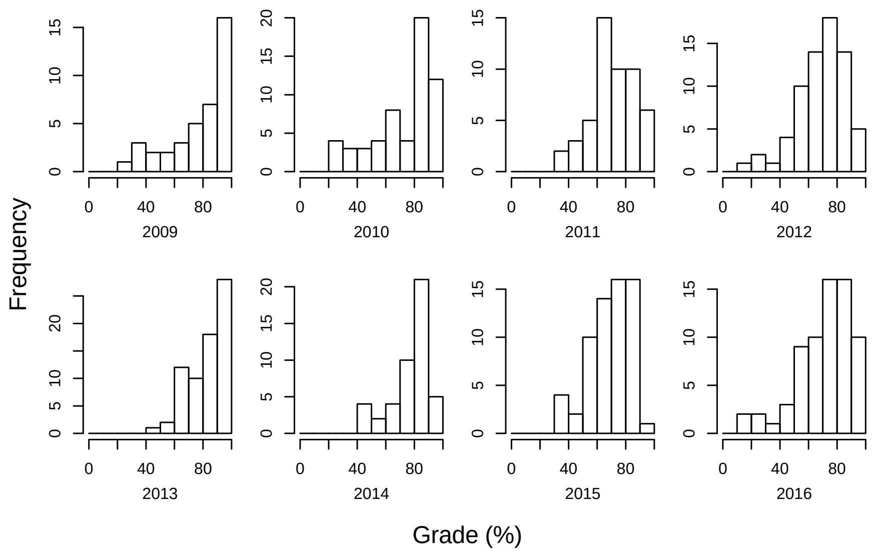

Figure 1. Grade Distributions for the Database Implementation Assignment, 2009–2016.

From the perspective of teaching staff, the system automatically ensured that all gradable items were checked, greatly improving consistency and the subjective experience of grading. The total amount of time taken to grade assignments was reduced, but the system only semi-automated the process, meaning we still needed to convert the system's output into corresponding grades and meaningful feedback (see Section 6 for further discussion). There were also a surprising number of submissions that did not meet the minimum requirements and thus still had to be manually graded.

There are several potential confounding factors to consider. First, 2013 was the first year the assignment specification was "frozen" (see Section 3). We could argue that grades improved due to students having less flexibility and thus fewer opportunities for misinterpretation. However, the specification was also "frozen" in all subsequent years, and grades vary considerably over this period, especially in 2015. It thus seems unlikely that this was a factor in improving student performance.

Second, minimum requirements were imposed from 2013 onwards. Any schema that met them scored at least 50%, which could inflate grades. If we compare 2009–2012 with 2013–2016, we do find a significant increase in the mean, from 72.4% to 75.6% ($p \approx 0.047$). We argue, however, that this effect would be minimal in the absence of a convenient mechanism for conformance checking, such as student mode. Indeed, for 2013–2014 when student mode was available, the mean (81.7%) was significantly higher ($p \approx 10^{-9}$) than for 2015–2016 (70.2%) when it was not. This is further supported by no significant difference between the means for 2009–2012 and 2015–2016.

Third, the switch to second semester in 2012–2013 could have negatively impacted students' performance by lengthening the time between learning basic data management concepts in first year and the second year database course. If so, we would expect mean grades in second semester offerings to be lower. However, mean grades for second semester (76.9%) were in fact significantly *higher* ($p \approx 0.015$) than those for first semester (72.9%). This is not surprising given that 2013 (second semester) had the highest grades overall. This rules out semester changes as a factor.

Fourth, perhaps the years with higher grades used a less complex scenario. We computed the following database complexity metrics for each of the four scenarios used (summarized in Table 2): database complexity index (DCI) (Sinha, Romney, Dey, & Amin, 2014); referential degree (RD), depth of referential tree (DRT), and number of attributes (NA) (Calero, Piattini, & Genero, 2001; Piattini, Calero, & Genero, 2001); database complexity (DC) (Pavlić, Kaluža, & Vrček, 2008); and "Software Metric Analyzer for Relational Database Systems" (SMARtS) (Jamil & Batool, 2010). All six metrics clearly show the "manufacturer", "used cars", and "student records" scenarios to be of comparable complexity, while the "postgrad" scenario is less complex on all metrics except DRT. It thus seems unlikely that scenario complexity is a factor in student performance. It is interesting to note that the "used cars" scenario was used in both 2014 and 2015, and yet the 2015 grades were significantly *lower* than 2014. The only clear difference is that our system was not used in 2015.

Table 2

Database Complexity Metrics for the Scenarios Used in the Period 2009–2016

Scenario	**DCI**	**RD**	**NA**	**DRT**	**DC**	**SMARtS**
"postgrad"	277	9	32	7	37	28.75
"student records"	367	12	43	9	61	38.25
"manufacturer"	370	11	50	8	59	40.75
"used cars"	380	13	46	7	53	36.50

Fifth, class size could be a factor. We might expect a smaller class to have a more collegial atmosphere that promotes better learning. However, the class sizes in Table 1 reveal no discernible pattern across class size and performance. Indeed, both the best (2013) and worst (2012, 2015) performances were from classes of similar size (75, 77, and 71, respectively).

Sixth, perhaps better performance occurred in years where the cohort happened to be more capable. We obtained students' annual GPA data and computed the median as an indication of each cohort's general ability. In Table 1 we immediately see that the year with the best results (2013) also had the second lowest median GPA (3.2). Contrast this with the poorer results in 2012, where the median GPA was 3.4. In both years that student mode was available, median GPA was lower than or the same as most other years, yet performance was better than in years with higher median GPA. This argues against the idea that we simply had more capable students in the better performing years.

Seventh, the timing of the assignment varied across the period, occurring either early (2012–2014), halfway (2009–2010), or late (2011, 2015–2016) in the semester. The mean grade for early timing (77.0%) was significantly higher than for both halfway (72.2%, $p \approx 0.019$) and late (71.8%, $p \approx 0.013$), while there was no significant difference between halfway and late means. This suggests that scheduling the assignment early in the semester may have a positive effect on grades, and this period does overlap with the availability of student mode. However, as noted earlier there was a highly significant difference in mean between 2012 and 2013. There was also a smaller, but still significant difference in mean between 2012 and 2014 ($p \approx 0.0015$). We therefore conclude that while scheduling the assignment early in the semester may have had some positive effect on student performance, it does not seem to explain all of the positive effect seen in 2013 and 2014.

Finally, perhaps the different assignment weightings (15% in 2009–2010 vs. 10% in 2011–2016) affected student motivation. We could argue that the higher weighting in 2009–2010 was a greater incentive for students. If so, we should expect better performance in 2009–2010. Indeed, we do find that the mean for 2009–2010 is 75.1%, while that for 2011–2016 is 73.9%, a significant decrease ($p \approx 0.034$). However, this change occurred well before our system was even conceived of, let alone deployed, so it cannot be a factor in the improved performance seen in 2013 and 2014.

The sum of this evidence provides a strong argument in favor of student mode being the primary factor in improving student grade performance in the assignment during 2013 and 2014. The most plausible explanation is that the direct feedback provided by student mode enabled students to more effectively correct structural aspects of their schema to meet the minimum requirements, giving them more time to work on the integrity aspects of their schema, and thus gain a higher grade.

6. Known Issues and Future Work

While the use of student mode in 2013 and 2014 appears to have benefited student performance in the database implementation assignment, there are outstanding issues that still need to be addressed.

Ideally, our system would automatically assign appropriate marks and generate meaningful feedback, writing both of these directly into a student management database. Currently, however, it only semi-automates the assessment process, and it is still up to the teacher to interpret the test results, assign appropriate marks, and write feedback. Automatically generating appropriate marks is not particularly difficult, and we have already implemented the core functionality required to assign mark penalties to different kinds of error. It would be a straightforward extension to calculate marks based on these penalties and write them directly into a student management database.

Automatically generating useful feedback is difficult, however, due to the way PHPUnit reports test failures. There is no way to control or suppress the messages generated by PHPUnit test assertions, most likely because PHPUnit is not intended to be embedded inside other applications (as discussed in Section 4). Individual tests can specify a meaningful message to be displayed, but PHPUnit will still also generate obscure messages like “Failed asserting that 0 matches expected 1”, which can confuse students. These insuppressible assertion messages make it much harder to provide fully automatic, meaningful feedback to students. It may be that the only way to address this issue is to change the way PHPUnit works internally, or perhaps to develop a custom testing framework.

The system currently does not support behavioral aspects such as sequences, triggers, or stored procedures, as these are usually DBMS-specific in nature. It would not be difficult to add support for these, however, in the DBMS abstraction layer proposed in Section 4.

It would also be interesting to extend our system to help facilitate the teaching of SQL DDL, rather than just focusing on the assessment and grading process. This has been done for SQL DML in several previous systems, as discussed in Section 2.

As of 2017, our department no longer offers a dedicated second year database course. The main introduction to database concepts and SQL now occurs as part of a first year “Foundations” course, which attracts about 150 students in the first semester and about 100 in the second. Classes of this size further reinforce the argument for automated assessment, and we are exploring whether our system could be used in this course. If so, we will undertake user evaluations with the students.

7. Conclusion

In this paper, we described a novel system that semi-automates the process of assessing SQL DDL code. Most prior work in this area has focused on DML statements such as `SELECT`, but even systems that do support SQL DDL focus on small, discrete snippets of code (e.g., a single `CREATE TABLE`). Our system extends the state of the art by evaluating an entire database schema as a single unit. It also takes a novel approach to checking an SQL schema: rather than attempting to parse the SQL DDL code directly, it instead verifies that the schema conforms to a machine-readable version of the original specification using a unit testing approach, which has not been used before in this context.

While no user evaluations of the system were carried out, analysis of student performance in a relevant database assessment shows a highly statistically significant increase in mean grade in the two years that the student facing component of the system was available to students. Analysis of all the factors involved strongly suggests that student use of our system had a positive impact on their performance in this assessment. This is an encouraging result, which we plan to explore further.

References

Abelló, A., Burgués, X., Casany, M. J., Martín, C., Quer, C., Rodríguez, M. E., … Urpí, T. (2016). A software tool for e-assessment of relational database skills. *International Journal of Engineering Education*, *32*(3A), 1289–1312.

Abelló, A., Rodríguez, E., Urpí, T., Burgués, X., Casany, M. J., Martín, C., & Quer, C. (2008). LEARN-SQL: Automatic assessment of SQL based on IMS QTI specification. In *Proceedings of the 8th IEEE International Conference on Advanced Learning Technologies (ICALT 2008)* (pp. 592–593). Santander, Cantabria, Spain: IEEE Computer Society. https://doi.org/10.1109/ICALT.2008.27

Ambler, S. W. (2006). Database testing: How to regression test a relational database. Retrieved August 11, 2018, from http://www.agiledata.org/essays/databaseTesting.html

Bergmann, S. (2017). PHPUnit Documentation, Chapter 8: Database Testing. Retrieved from https://phpunit.de/manual/current/en/database.html

Bhangdiya, A., Chandra, B., Kar, B., Radhakrishnan, B., Reddy, K. V. M., Shah, S., & Sudarshan, S. (2015). The XDa-TA system for automated grading of SQL query assignments. In J. Gehrke, W. Lehner, K. Shim, S. K. Cha, & G. M. Lohman (Eds.), *Proceedings of the 31st IEEE International Conference on Data Engineering* (pp. 1468–1471). Seoul, South Korea: IEEE Computer Society. https://doi.org/10.1109/ICDE.2015.7113403

Binnig, C., Kossmann, D., & Lo, E. (2008). Multi-RQP: Generating test databases for the functional testing of OLTP applications. In L. Giakoumakis & D. Kossmann (Eds.), *Proceedings of the 1st International Workshop on Testing Database Systems (DBTest 2008)* (pp. 5:1–5:6). Vancouver, British Columbia, Canada: ACM. https://doi.org/10.1145/1385269.1385276

Brusilovsky, P., Sosnovsky, S., Yudelson, M. V., Lee, D. H., Zadorozhny, V., & Zhou, X. (2010). Learning SQL programming with interactive tools: From integration to personalization. *ACM Transactions on Computing Education*, *9*(4), 19:1–19:15. https://doi.org/10.1145.1656255.1656257

Calero, C., Piattini, M., & Genero, M. (2001). Database complexity metrics. In *Proceedings of the 4th International Conference on the Quality of Information and Communications Technology (QuaTIC 2001)* (Vol. 1284, pp. 79–85). Lisbon, Portugal. Retrieved from http://ceur-ws.org/Vol-1284/paper9.pdf

Chandra, B., Chawda, B., Kar, B., Reddy, K. V. M., Shah, S., & Sudarshan, S. (2015). Data generation for testing and grading SQL queries. *The VLDB Journal*, *24*(6), 731–755. https://doi.org/10.1007/s00778-015-0395-0

Chandra, B., Joseph, M., Radhakrishnan, B., Acharya, S., & Sudarshan, S. (2016). Partial marking for automated grading of SQL queries. *Proceedings of the VLDB Endowment*, *9*(13), 1541–1544. https://doi.org/10.14778/3007263.3007304

Chays, D., Shahid, J., & Frankl, P. G. (2008). Query-based test generation for database applications. In L. Giakoumakis & D. Kossmann (Eds.), *Proceedings of the 1st International Workshop on Testing Database Systems (DBTest 2008)* (pp. 6:1–6:6). Vancouver, British Columbia, Canada: ACM. https://doi.org/10.1145/1385269.1385277

Codd, E. F. (1981). Data models in database management. *ACM SIGMOD Record*, *11*(2), 112–114. https://doi.org/10.1145/960126.806891

Cvetanović, M., Radivojević, Z., Blagojević, V., & Bojović, M. (2011). ADVICE—Educational system for teaching database courses. *IEEE Transactions on Education*, *54*(3), 398–409. https://doi.org/10.1109/TE.2010.2063431

Date, C. J. (2009). *SQL and Relational Theory: How to Write Accurate SQL Code*. Sebastopol, California, USA: O'Reilly.

Dekeyser, S., Raadt, M. de, & Lee, T. Y. (2007). Computer assisted assessment of SQL query skills. In J. Bailey & A. Fekete (Eds.), *Database Technologies 2007* (Vol. 63, pp. 53–62). Ballarat, Australia: Australian Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=1273730.1273737

Dietrich, S. W. (1993). An educational tool for formal relational database query languages. *Computer Science Education*, *4*(2), 157–184. https://doi.org/10.1080/0899340930040201

Dollinger, R., & Melville, N. A. (2011). Semantic evaluation of SQL queries. In *Proceedings of the IEEE 7th International Conference on Intelligent Computer Communication and Processing (ICCP 2011)* (pp. 57–64). https://doi.org/10.1109/ICCP.2011.6047844

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing*, *5*(3), 4:1–4:13. https://doi.org/10.1145/1163405.1163409

Farré, C., Rull, G., Teniente, E., & Urpí, T. (2008). SVTe: A tool to validate database schemas giving explanations. In L. Giakoumakis & D. Kossmann (Eds.), *Proceedings of the 1st International Workshop on Testing Database Systems (DBTest 2008)* (pp. 9:1–9:6). Vancouver, British Columbia, Canada: ACM. https://doi.org/10.1145/1385269.1385281

Gong, A. (2015, October). CS 121 Automation Tool. https://github.com/anjoola/cs12x-automate

Haller, K. (2010). The test data challenge for database-driven applications. In S. Babu & G. N. Paulley (Eds.), *Proceedings of the 3rd International Workshop on Testing Database Systems (DBTest 2010)* (pp. 6:1–6:6). Indianapolis, Indiana, USA: ACM. https://doi.org/10.1145/1838126.1838132

Jamil, B., & Batool, A. (2010). SMARtS: Software metric analyzer for relational database systems. In *Proceedings of the 2010 International Conference on Information and Emerging Technologies* (pp. 27:1–27:6). Karachi, Pakistan: IEEE Computer Society. https://doi.org/10.1109/ICIET.2010.5625716

Kearns, R., Shead, S., & Fekete, A. (1997). A teaching system for SQL. In H. Søndergaard & A. J. Hurst (Eds.), *Proceedings of the ACM SIGCSE 2nd Australasian Conference on Computer Science Education (ACSE 1997)* (pp. 224–231). Melbourne, Australia: ACM. https://doi.org/10.1145/299359.299391

Kenny, C., & Pahl, C. (2005). Automated tutoring for a database skills training environment. In W. Dann, T. L. Naps, P. T. Tymann, & D. Baldwin (Eds.), *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2005)* (pp. 58–62). St. Louis, Missouri, USA: ACM. https://doi.org/10.1145/1047124.1047377

Kleiner, C., Tebbe, C., & Heine, F. (2013). Automated grading and tutoring of SQL statements to improve student learning. In M.-J. Laakso & Simon (Eds.), *Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling '13)* (pp. 161–168). Koli, Finland: ACM. https://doi.org/10.1145/2526968.2526986

Laine, H. (2001). SQL-trainer. In E. Sutinen & M. Kuittinen (Eds.), *Kolin Kolistelut/Koli Calling, Proceedings of the First Annual Finnish/Baltic Sea Conference on Computer Science Education* (pp. 13–17). Koli, Finland. Retrieved from https://www.cs.helsinki.fi/u/laine/SQLtrainer_conf.pdf

Marcozzi, M., Vanhoof, W., & Hainaut, J.-L. (2012). Test input generation for database programs using relational constraints. In E. Lo & F. Waas (Eds.), *Proceedings of the 5th International Workshop on Testing Database Systems (DBTest 2012)* (pp. 6:1–6:6). Scottsdale, Arizona, USA: ACM. https://doi.org/10.1145/2304510.2304518

Mitrovic, A. (1998). Learning SQL with a computerized tutor. In J. Lewis, J. Prey, D. Joyce, & J. Impagliazzo (Eds.), *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'98)* (pp. 307–311). Atlanta, Georgia, USA: ACM. https://doi.org/10.1145/274790.274318

Mitrovic, A. (2003). An intelligent SQL tutor on the Web. *International Journal of Artificial Intelligence in Education*, *13*(2–4), 173–197.

Pavlić, M., Kaluža, M., & Vrček, N. (2008). Database complexity measuring method. In B. Aurer, M. Bača, & K. Rabuzin (Eds.), *Proceedings of the 19th Central European Conference on Information and Intelligent Systems (CECIIS 2008)* (pp. 577–583). Varaždin, Croatia: University of Zagreb. Retrieved from http://archive.ceciis.foi.hr/app/index.php/ceciis/2008/paper/view/84

Perry, D. E., Siy, H. P., & Votta, L. G. (2001). Parallel changes in large-scale software development: An observational case study. *ACM Transactions on Software Engineering and Methodology*, *10*(3), 308–337. https://doi.org/10.1145/383876.383878

Piattini, M., Calero, C., & Genero, M. (2001). Table oriented metrics for relational databases. *Software Quality Journal*, *9*(2), 79–97. https://doi.org/10.1023/A:1016670717863

Prior, J. C., & Lister, R. (2004). The backwash effect on SQL skills grading. In R. D. Boyle, M. Clark, & A. N. Kumar (Eds.), *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 32–36). Leeds, UK: ACM. https://doi.org/10.1145/1026487.1008008

Russell, G., & Cumming, A. (2004). Improving the student learning experience for SQL using automatic marking. In Kinshuk, D. G. Sampson, & P. T. Isaías (Eds.), *Proceedings of the IADIS International Conference on Cognition and Exploratory Learning in Digital Age (CELDA'04)* (pp. 281–288). Lisbon, Portugal: IADIS. Retrieved from http://www.napier.ac.uk/research-and-innovation/research-search/outputs/improving-the-student-learning-experience-for-sql-using-automatic-marking

Russell, G., & Cumming, A. (2005). Online assessment and checking of SQL: Detecting and preventing plagiarism. In U. O'Reilly & R. Cooper (Eds.), *Proceedings of the 3rd HEA-ICS Workshop on Teaching, Learning and Assessment in Databases (TLAD 2005)* (pp. 46–50). Sunderland, UK: LTSN-ICS. Retrieved from http://www.napier.ac.uk/research-and-innovation/research-search/outputs/online-assessment-and-checking-of-sql-detecting-and-preventing-plagiarism

Sadiq, S. W., Orlowska, M. E., Sadiq, W., & Lin, J. Y.-C. (2004). SQLator: An online SQL learning workbench. In R. D. Boyle, M. Clark, & A. N. Kumar (Eds.), *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 223–227). Leeds, UK: ACM. https://doi.org/10.1145/1026487.1008055

Sinha, B. R., Romney, G. W., Dey, P. P., & Amin, M. N. (2014). Estimation of database complexity from modeling schemas. *Journal of Computing Sciences in Colleges*, *30*(2), 95–104.

Soler, J., Boada, I., Prados, F., Poch, J., & Fabregat, R. (2007). An automatic correction tool for relational algebra queries. In *Proceedings of the 2007 International Conference on Computational Science and Its Applications (ICCSA 2007), Part II* (Vol. 4705, pp. 861–872). Kuala Lumpur, Malaysia: Springer. https://doi.org/10.1007/978-3-540-74477-1_77