Self-Paced, Instructor-Assisted Approach to Teaching SQL

Pavel Solin[a,b]

[a] *University of Nevada, Reno, 1664 N Virginia St, Reno, NV 89557, USA*
[b] *NCLab, 450 Sinclair St, Reno, NV 89501, USA*

Abstract

We present a novel approach to teaching Structured Query Language (SQL) suitable for both college classroom environment and asynchronous remote instruction. Students do not sit passively, listening to lectures or watching tutorial videos. Instead, they spend their time (including lecture time) working actively at their own pace through bite-sized tutorials, examples, exercises, practical tasks, and quizzes. They get much more hands-on practice than in the traditional lecture + homework model. Their work is checked in real time by an AI-based software platform which also provides instant help and guidance as needed. Students must prove mastery of each concept before being able to move on to the next one. The instructor does not lecture in the traditional way. Instead, s/he interacts with students individually, and provides personalized help and assistance as needed. Students not only enjoy the one-to-one interaction with their instructor much more than listening to a lecture, but they also greatly benefit from it. In this paper we provide a concise overview of the teaching method, and then we focus on automated server-side analysis of SQL queries, which is the cornerstone of the self-paced SQL course. We introduce a number of Python-based SQL analyzers for various types of queries, and present sample code. Our SQLGrader library is freely available on Github under an open source license.

Keywords: Structured Query Language (SQL), Competency-based education (CBE), Online learning, Self-paced learning, Asynchronous learning, Learning-by-doing, NCLab

Email address: solin@unr.edu, pavel@nclab.com (Pavel Solin)

1. Introduction

SQL (Structured Query Language) is a domain-specific language designed for managing data held in a relational database management system (RDBMS) or for stream processing in a relational data stream management system (RDSMS). With the growing importance of data in all areas of human activity, SQL has become an essential job skill in a number of occupations including but not limited to data engineers, data scientists, data analysts, database administrators, software developers, web designers, and many others.

Various approaches to teaching SQL have been discussed by both educators and practitioners. The following representative papers, and the references therein, offer a broad view of modern approaches to teaching SQL, including gamification, visualization tools, and flipped classroom methods:

In [1] the authors compare traditional teaching methods of SQL with gamified approaches, analyzing the effectiveness of each method in student engagement and learning outcomes. The study [2] explores the impact of interactive visual learning tools in teaching SQL, highlighting how visual aids improve students' comprehension of database concepts and query construction. The paper [3] presents a teaching method where students learn SQL by mapping natural language queries to semantic structures, focusing on improving their query formulation skills. In [4] the authors discuss the application of flipped classrooms in teaching SQL, assessing both the students' perspectives and the quantitative outcomes of using this approach. The paper [5] discusses a teaching experiment using collaborative problem solving in online environments to teach SQL, comparing this method's efficiency with traditional individual problem-solving techniques.

The present study builds on our recent work [6] and [7] where the self-paced, instructor-assisted teaching method was introduced and discussed in the context of Linear Algebra and Python programming. The outline of the present paper is as follows: A brief overview of SQL is presented in Section 2. The difference between skills and knowledge is discussed in Section 3. Section 4 describes the teaching method, and Sections 5 - 8 discuss automated server-side analysis and grading of SQL queries. This is a cornerstone of the self-paced SQL course because it validates the students' understanding of the material, giving them confidence in what they have learned, and allowing them to progress through the course efficiently. The SQLGrader library presented in this paper is freely available on Github under an open source

license. In Section 9 we share repository and license information, and discuss code integration and adaptability. Conclusions are presented in Section 10, and sample code is shown in Appendix A.

In particular, our previous study [7] applies to teaching SQL which, like Python, is a programming language. It would not be possible to provide a detailed discussion of the teaching methodology here without creating a significant overlap with our previous papers. Therefore, we recommend that the reader retrieves [7] and goes through Sections 1 (Introduction), 3 (Lectures, Homework, and Flipped Classroom), 4 (Self-Paced Learning by Doing), 5 (Interactive Course Materials), 6 (Importance of Real-Time Feedback), 7 (Real-Time Progress Monitoring), 8 (Results of a Student Survey) and 9 (Representative Student Testimonials) and Appendix A (Traditional Instruction Versus Self-Paced Learning By Doing From a Student's Point of View). We also recommend that the reader retrieves the original paper [6] and goes at least through Section 1 (Introduction) and Subsections 3.1 (Self-Paced, Instructor-Assisted Learning vs. Flipped Classroom), 3.3 (Benefits for the Students), 3.4 (Benefits for the Instructor), 3.6 (Student Comments), and Appendix A (Tips, Tricks, and Lessons Learned).

2. Brief Overview of SQL

The origins of SQL date back to the early 1970s when it was developed by Donald D. Chamberlin and Raymond F. Boyce at IBM. The language was originally called SEQUEL (Structured English Query Language), but it was later abbreviated to SQL due to trademark issues. The primary goal of SQL was to provide a more intuitive and accessible way to interact with databases, allowing users to query, insert, update, and delete data without needing to understand the underlying structure of the database.

The significance of SQL lies in its standardization and widespread adoption. SQL became the standard language for relational database management after it was standardized by the American National Standards Institute (ANSI) in 1986 and by the International Organization for Standardization (ISO) in 1987. The language's ability to handle large datasets, its declarative nature (where users specify what data to retrieve rather than how to retrieve it), and its use of simple, English-like syntax contributed to its success. Today, SQL is used in almost every application that deals with structured data, from simple web applications to complex enterprise systems, making it a cornerstone of modern data management and analysis.

Academic literature has often highlighted the evolution and importance of SQL. For instance, Melton and Simon [8] provided a comprehensive overview of SQL's evolution and its impact on database technology. They discuss how SQL has expanded from a simple query language into a robust tool for defining, controlling, and manipulating data across various industries. Codd's seminal work [9] on relational databases, which laid the groundwork for SQL, is another critical reference in the academic study of databases. SQL's ability to provide a common language for diverse database systems has been crucial in enabling interoperability between systems, thus driving innovation and collaboration across different technological platforms.

Moreover, in more recent research, the scalability and optimization of SQL queries have been a significant area of focus. Researchers like Chaudhuri [10] have explored the ways in which SQL query optimization can improve the performance of large-scale databases, a crucial consideration as data volumes continue to grow. The language's adaptability and continuous improvement have ensured its relevance in an ever-evolving technological landscape, making SQL not just a tool of the past, but a key player in the future of data management.

3. Knowledge, Skills, and the Skills Gap

In order to fully appreciate the teaching method presented here, one needs to acknowledge the difference between learning new knowledge and acquiring new skills. The distinction between knowledge and skills is foundational in understanding how people learn and perform tasks.

Knowledge refers to the theoretical or factual information that someone possesses. It's what one knows, including concepts, facts, principles, and theories. For example, knowledge might include knowing the laws of physics, understanding historical events, or being aware of mathematical formulas or SQL keywords. It is often acquired through reading, studying, or being taught.

Skills, on the other hand, refer to the ability to apply knowledge to perform tasks effectively. Skills are what one can do, and they often require significant amounts of both practice and time to develop. For instance, being able to solve complex math problems, conduct scientific experiments, or write complex SQL queries. Skills involve practical application and are typically developed through hands-on experience, repetition, and practice.

In summary, knowledge is about comprehension and awareness, while skills are about action and application. Knowledge can often be acquired passively (through listening, reading, or watching), whereas skills usually require active practice. Knowledge is the foundation upon which skills are built. For instance, knowing a language (knowledge) is different from being able to speak it fluently (skill). In many cases, both are needed to be competent in a particular field: knowledge provides the necessary understanding, and skills enable you to apply that understanding effectively.

Nowadays, most colleges and universities place emphasis on knowledge, and students are expected to acquire the corresponding skills by completing a limited number of homework assignments. However, this approach does not work, and employers consistently report that graduates do not possess sufficient skills when entering the workforce. The gap between the skills of graduates and the expectations of employers, referred to as the "skills gap", is a well-documented issue in academic and industry literature.

For instance, [11] highlights the concern of employers that graduates often lack key skills necessary for the workplace, despite having academic qualifications. The study [12] discusses the discrepancy between the skills that graduates possess and those that employers find necessary, with a particular emphasis on the lack of "soft skills". The report [13] from McKinsey highlights the global issue where many employers feel that graduates are not adequately prepared for the workforce, particularly in terms of practical skills. In [14] the authors discuss the challenges universities face in helping students develop the generic skills that are highly valued by employers.

In the context of SQL, the paper [15] addresses the skills gap among graduates and suggests curriculum improvements to better prepare students for industry needs. The study [16] discusses the challenges students face in learning SQL and the subsequent gap in skills when they enter the workforce. In [17], the authors explore the effectiveness of different teaching approaches to SQL and database design and highlights the skill deficiencies observed in graduates. Last but not least, the paper [18] examines the SQL skills gap and proposes experiential learning as a method to enhance SQL competencies among students.

4. Brief Overview of Self-Paced, Instructor-Assisted Learning

In traditional lectures, students are asked to sit, listen, take notes, and stay focused over long periods of time such as 75 or even 90 minutes. How-

ever, it is challenging for the students to keep their attention alive for so long. Furthermore, they are not able to practice and master each concept before proceeding to the next one. They are also not able to obtain personalized attention of the instructor. This model was established during the 2nd industrial revolution, and in [19] it is called "assembly line education".

Some authors present the so-called "flipped classroom" as a remedy (see [6, 7] for references). In this model, students are asked to learn the new material on their own, outside of class, typically through videos, readings, or online lectures. They may also be asked to complete assignments or quizzes to ensure they have understood the material. Then, class time is used for engaging in activities that apply the concepts learned in the pre-class materials. This can include problem-solving, discussions, group work, or hands-on projects. Our problem with the flipped classroom is that the instructor is not available to help students when they are learning the new material and actually need help.

For this reason, initially in the context of Linear Algebra [6], we introduced an alternative model where students work actively 100% of the time, using an advanced interactive learning platform combined with real-time individual assistance of their instructor. The methodology met with enormously positive student feedback. It was equally successful when it was extended to teaching Python programming in [7]. Below we present a brief summary of the teaching method, to avoid excessive overlap with our previous papers.

Instead of sitting and listening, or learning on their own without help, students work actively 100% of the time under the supervision of an instructor. They use their own devices and work at their own pace through bite-sized tutorials, examples, exercises, and graded practical tasks and quizzes. Their work is checked by a cloud platform in real-time, instantly validating their understanding of the material and providing help where needed. Students must prove mastery of each concept before being allowed to tackle the next one. Instead of lecturing, the instructor assists each student individually. This makes a huge positive difference for the students, and it also allows the instructor to understand much better how each student performs and where they need help. After 10+ years of using this approach, we are convinced that nothing can replace a one-to-one interaction between the student and his/her instructor.

5. Automated Analysis of SQL Queries

A cornerstone of the self-paced, instructor-assisted teaching method is an advanced software platform such as [20] which is capable of checking students' solutions to exercises and tasks in real time, and providing them with detailed instant feedback. Students need this feedback in order to validate their understanding of the material, gain confidence, and progress through the coursework efficiently. We discussed this aspect of the methodology in detail in the context of Linear Algebra in [6] and in the context of Python programming in [7]. In the context of SQL, automated real-time feedback is equally important.

When checking students' queries, we first analyze their output and then also their SQL code. To begin with, the following three functions compare the output of the student's query with the output of the master solution:

Column Presence Analyzer

Function `sql_column_presence_analyzer` identifies columns which are correctly included, missing, or unexpectedly present in the student's output compared to the output of the master solution. It provides specific feedback about which columns are correctly included, which are missing, and which should not have been included.

Column Order Analyzer

Function `sql_column_order_analyzer` ensures that the order of columns in the output of the student's query matches the order of columns in the output of the master solution. If the columns are out of order, it provides feedback indicating which columns are not correctly ordered.

Row Analyzer

Function `sql_row_analyzer` checks if the rows in the output of the student's query match those in the output of the master solution, in terms of both content and ordering. It points out discrepancies, whether there are missing rows, excess rows, or issues with the ordering of rows. Its functionality is as follows:

- **Convert Rows to Hashable Objects**: The function first converts the rows in the outputs of both the master solution and the student's solution into tuples, allowing them to be used in set operations.

- **Compare Rows Via Set Operations**: It performs set operations to identify rows that are correct (appear in both sets), missing (appear only in the solution), and excessive (appear only in the student's result).
- **Detect Discrepancies**: The function also checks for discrepancies by comparing the rows in the solution output and student's output. If there are discrepancies, it generates corresponding error messages.
- **Check Ordering**: The function checks whether the rows in the student's output are ordered correctly. If not, it reports the rows that are out of order.
- **Return Value**: The function returns a list of error messages (as text strings) if problems are found, or `False` if there were no errors.

For illustration, the source code for the `sql_row_analyzer` function is presented in Appendix A.

Code Analyzer

In contrast to the above functions which analyze the output of SQL queries, the `sql_code_analyzer` function analyzes the SQL code of the student's query, and points out problems in case the output differs from the output of the master solution. This function does the following:

- **Normalization and Sanitation**: It standardizes SQL code by removing non-essential elements such as comments and excess formatting.
- **Keyword Analysis**: The function evaluates the presence and correctness of SQL keywords. It identifies missing or excessive keywords and checks their syntactical order.
- **Dynamic Element Handling**: Adapting to queries involving dynamic SQL elements like `CURRENT_USER`, the function personalizes the grading process by replacing these elements with specified user names, enhancing the relevance of feedback.
- **Feedback Generation**: For any identified errors, `sql_code_analyzer` provides detailed, actionable feedback, pinpointing the exact nature of the mistakes and suggesting how to correct them.

6. SELECT Statement Grader

The elementary analyzer functions described above are used to build more complex grader functions for individual SQL statements. To begin with, the SELECT Statement Grader (function `sql_select_grader`) is designed to assess the correctness of `SELECT` statements written by students. This function does several things:

Initial Tests and Code Sanitation

The analysis begins with checking for any critical issues such as missing output. If the student's query does not produce any results, immediate feedback is provided, urging the student to review his or her query and try again. Both the solution and student codes are then sanitized using the function `sql_code_sanitizer` which cleans up the code by removing non-essential elements such as comments or excessive white spaces. This step is done to provide a cleaned student and solution queries suitable for further inspection with the function `sql_code_analyzer`.

Analyzing Columns

The function checks whether all required columns are present in the output of the student's query. It uses the function `sql_column_presence_analyzer` to perform this check, ensuring that no required columns are missing and no extra columns are included. If the query involves aggregate functions such as `MAX`, `COUNT`, etc., then the grader ensures that these are used correctly. If the student incorrectly uses or omits necessary aggregate functions, then specific feedback is provided. The order of columns is also checked using the function `sql_column_order_analyzer`.

Analyzing Rows

Once the columns in the output of the student's query are correct, function `sql_row_analyzer` is used to check the rows. This part of the grading ensures that the `SELECT` statement not only selects the correct columns but also fetches the right set of rows according to the conditions specified in the query. All missing or excess rows are reported.

Analyzing SQL Code

If the output of the student's query is not correct, function `sql_code_analyzer` is called to check the SQL code of the query. This includes checking its syntactical and functional correctness, ensuring that it aligns with SQL standards and the specific requirements of the assignment. The Code Analyzer is able to precisely pinpoint the error in the student's query.

Error Reporting

If any discrepancies are found in the columns or rows of the output, or in the SQL query itself, then detailed feedback is provided. This feedback includes specific errors, corrective suggestions, and useful hints that help the student understand what went wrong and how to correct it.

The SELECT Statement Grader plays a fundamental role in teaching students the intricacies of writing correct and efficient SQL `SELECT` queries. By providing multi-faceted, detailed feedback on every aspect of the query, the grader not only helps students to improve their incorrect solutions, but it also aids in their understanding of how SQL queries should be structured and executed. This comprehensive approach ensures that students learn SQL in a manner that emphasizes both accuracy and efficiency, making them well-prepared for real-world data manipulation and analysis tasks.

7. Additional SQL Statement Graders

In addition to the SELECT Statement Grader described above, we also use the following Statement Graders for other SQL statements:

CREATE/ALTER Table Grader

Function `sql_create_or_alter_table_grader` grades SQL statements for creating or altering tables. It verifies that all required columns, their types, and any modifications are correctly defined. This function provides detailed error messages related to column definitions and table alterations.

Table Constraint Grader

Function `sql_table_constraints_grader` assesses the implementation of constraints in table definitions. It ensures that primary, foreign keys, and other constraints are correctly applied. This function identifies and reports missing or unnecessary constraints.

INSERT Grader

Function `sql_insert_grader` evaluates the correctness of `INSERT` operations. It confirms that inserted data adheres to defined constraints and correctly populates tables. This function highlights discrepancies in data entries, particularly focusing on `NOT NULL` constraints and field accuracy.

DELETE Grader

Function `sql_delete_grader` grades `DELETE` operations. It verifies that only the appropriate records are removed based on the task definition. This function provides a detailed error message when incorrect records are affected or conditions are not applied correctly.

DROP TABLE Grader

Function `sql_drop_table_grader` assesses the `DROP TABLE` command. It ensures that the specified tables are correctly dropped from the database. This function alerts students if a table that should have been dropped remains.

Function Grader

Function `sql_function_grader` analyzes SQL functions written by students. It checks for immutability, handling of `NULL` values, and the correct data type of the function as defined in the function's metadata. If discrepancies are found between the student's function and the solution, then detailed feedback is provided, highlighting the parameter in question and the expected value.

Function Parameters Grader

Function `sql_function_parameters_grader` specifically checks the parameters used in defining SQL functions. It verifies that all parameters of the student's function match those of the solution in both type and order. This function returns detailed feedback if any parameters are missing, misplaced, or incorrectly defined.

Function Tests Grader

Function `sql_function_tests_grader` checks the correctness of SQL functions by running them with test inputs and comparing the outputs against expected results. It evaluates whether the function returns the correct data for given test cases and also checks the correctness of the function's code itself if discrepancies are found. This function provides a combination of row-based and code-based feedback, making it clear where the function fails – whether the problem is in the logic, output, or syntax.

Type Grader

Function `sql_type_grader` checks user-defined types. It compares the attributes of types defined by students against those specified in the solution, including data type, nullability, and other constraints. A detailed error message is provided if any discrepancies are found.

8. Example

We will illustrate the usage of the SELECT Statement Grader on a concrete example which involves the well-known database (schema) `world`. This schema is part of the MySQL database system [21], and it is often used for learning and practicing SQL queries. It contains data about many countries, cities, and languages of the world, providing a realistic dataset for working with relational databases. This schema is commonly used in tutorials, SQL exercises, and courses because it presents a practical, structured dataset suitable for educational purposes.

Readers who know SQL will benefit from this example the most, but even readers who aren't familiar with SQL will get an idea of what the SELECT Statement Grader does and how the instant feedback benefits the students. Let's begin with a brief overview of the schema `world`.

Schema `world`

The schema `world` consists of three tables `city`, `country` and `countrylanguage`. To begin with, table `country` has 239 rows (one per country) and the following 15 columns:

- `code`: a unique country code (e.g., `'USA'` for the United States).
- `name`: the name of the country.

- `continent`: the corresponding continent.
- `region`: more specific geographic region.
- `surfacearea`: the surface area of the country in square kilometers.
- `population`: the population of the country.
- `gnp` (Gross National Product) and `gnp_old`: economic data.
- Other fields related to government form, capital, life expectancy, etc.

To get a better idea of how the data looks like, let's display four selected columns of this table by typing a simple query,

```
SELECT code, name, continent, surfacearea FROM world.country;
```

The output of this query is

```
code | name                  | continent     | surfacearea
-----+-----------------------+---------------+------------
AFG  | Afghanistan           | Asia          | 652090.0
EGY  | United Arab Republic | Africa        | 1001450.0
ANT  | Netherlands Antilles | North America | 800.0
ALB  | Albania               | Europe        | 28748.0
DZA  | Algeria               | Africa        | 2381740.0
...
```

The second table `city` has 4079 rows (one per city) and the following five columns:

- `id`: a unique identifier for the city.
- `name`: the city's name.
- `countrycode`: a foreign key linking the city to the country it belongs to.

- `district`: the district or state the city is located in.
- `population`: the population of the city.

For illustration, let's display a few rows of this table by typing

```
SELECT * FROM world.city;
```

Here is the corresponding output:

```
id | name           | countrycode | district      | population
---+----------------+-------------+---------------+-----------
1  | Kabul          | AFG         | Kabol         | 1780000
2  | Qandahar       | AFG         | Qandahar      | 237500
3  | Herat          | AFG         | Herat         | 186800
4  | Mazar-e-Sharif | AFG         | Balkh         | 127800
5  | Amsterdam      | NLD         | Noord-Holland | 731200
...
```

Finally, the third table `countrylanguage` stores data about languages spoken in the 239 countries. This table has 984 rows and four columns:

- `countrycode`: A foreign key linking to the `country` table.
- `language`: The name of the language.
- `isofficial`: Whether the language is an official language of the country (Boolean).
- `percentage`: The percentage of the population that speaks the language.

The data can be viewed by typing the query

```
SELECT * FROM world.countrylanguage;
```

The output of this query is

```
countrycode | language   | isofficial | percentage
------------+------------+------------+-----------
AFG         | Pashto     | True       | 52.4
NLD         | Dutch      | True       | 95.6
ANT         | Papiamento | True       | 86.2
...
```

With the knowledge of the schema `world`, we can now proceed to our actual example.

Sample Task and Master Solution

Let's assume that the student's task is to find the five largest cities in Africa where the official language is French, and display the city name, country name, and city population. The three columns in the resulting table should be named `name`, `country` and `population`. The table should be sorted by city population in descending order.

The correct query (master solution) is

```
SELECT city.name, country.name AS country, city.population
  FROM world.city
    JOIN world.country
      ON world.city.countrycode = world.country.code
    JOIN world.countrylanguage
      ON world.countrylanguage.countrycode = world.country.code
  WHERE continent = 'Africa'
    AND world.countrylanguage.language = 'French'
    AND world.countrylanguage.isofficial = 'True'
  ORDER BY city.population DESC LIMIT 5;
```

And, this is the corresponding output:

```
name         | country   | population
-------------+-----------+-----------
Antananarivo | Malagasy  | 675669
Bujumbura    | Burundi   | 300000
Kigali       | Rwanda    | 286000
Toamasina    | Malagasy  | 127441
Antsirabé    | Malagasy  | 120239
```

It is left up to the course creator to decide whether or not to share the expected output, or its part, with the student. Based on our experience, sharing the expected output makes it easier for the students to debug their queries.

Checking the Student's Solution

The query includes a `SELECT` statement, therefore the SELECT Statement Grader is used to check the result. The grader compares the output of the student's query with the output of the master solution. If they are identical then the student's solution is accepted. If not, then the SELECT Statement Grader points out the discrepancies, and even provides hints as to what part of the student's query should be improved.

For illustration, suppose that the student submits the following incorrect query,

```
SELECT city.name, country.name, city.population
  FROM world.city
    JOIN world.country
      ON world.city.countrycode = world.country.code
    JOIN world.countrylanguage
      ON world.countrylanguage.countrycode = world.country.code
  WHERE continent = 'Africa'
    AND world.countrylanguage.language = 'French'
    AND world.countrylanguage.isofficial = 'True';
```

whose output is

```
name         | name       | population
-------------+------------+-----------
Bujumbura    | Burundi    | 300000
Antananarivo | Malagasy   | 675669
Toamasina    | Malagasy   | 127441
Antsirabé    | Malagasy   | 120239
Mahajanga    | Malagasy   | 100807
Fianarantsoa | Malagasy   | 99005
Mamoutzou    | Mayotte    | 12000
Kigali       | Rwanda     | 286000
Victoria     | Seychelles | 41000
```

This output is obviously incorrect (wrong column headers, wrong number of rows, table not sorted correctly). The SELECT Statement Grader will not list all errors at once, because their number would be overwhelming. In the first step, it will just point out the wrong column names:

```
Your result correctly contains the following columns:
    name, population
Your result is missing the following column:
    country
Your result contains one column which is not expected:
    name
```

If the number or ordering of the columns was not correct, then this would be reported as well.

Let's assume that the student adjusts his or her query based on this feedback, and submits an updated query with column names corrected:

```
SELECT city.name, country.name AS country, city.population
  FROM world.city
    JOIN world.country
      ON world.city.countrycode = world.country.code
    JOIN world.countrylanguage
      ON world.countrylanguage.countrycode = world.country.code
```

```
  WHERE continent = 'Africa'
    AND world.countrylanguage.language = 'French'
    AND world.countrylanguage.isofficial = 'True';
```

The corresponding output is:

```
name         | country    | population
-------------+------------+-----------
Bujumbura    | Burundi    | 300000
Antananarivo | Malagasy   | 675669
Toamasina    | Malagasy   | 127441
Antsirabé    | Malagasy   | 120239
Mahajanga    | Malagasy   | 100807
Fianarantsoa | Malagasy   | 99005
Mamoutzou    | Mayotte    | 12000
Kigali       | Rwanda     | 286000
Victoria     | Seychelles | 41000
```

The number, names, and ordering of the columns are now correct, but as the reader can see, the rows are not correct. Also, the last column is not sorted in descending order. Therefore the student's query is not accepted, and the SELECT Statement Grader provides the following additional feedback:

```
5 rows are correct (rows 1, 2, 3, 4, 8).
4 rows are not expected (rows 5, 6, 7, 9).
Your SQL code is missing these keywords:
    ORDER BY, DESC, LIMIT
```

Let's assume that the student adds the required keywords and submits the updated query:

```
SELECT city.name, country.name AS country, city.population
  FROM world.city
    JOIN world.country
```

```
    ON world.city.countrycode = world.country.code
  JOIN world.countrylanguage
    ON world.countrylanguage.countrycode = world.country.code
WHERE continent = 'Africa'
  AND world.countrylanguage.language = 'French'
  AND world.countrylanguage.isofficial = 'True'
ORDER BY country.population DESC LIMIT 5;
```

The corresponding output is

```
name         | country   | population
-------------+-----------+-----------
Antananarivo | Malagasy | 675669
Toamasina    | Malagasy | 127441
Antsirabé    | Malagasy | 120239
Mahajanga    | Malagasy | 100807
Fianarantsoa | Malagasy | 99005
```

This result is still wrong because the resulting table is sorted based on country population and not city population. The SELECT Statement Grader will guide the student to find the problem:

```
3 rows are correct (rows 1, 2, 3).
2 rows are not correct (rows 4, 5).
Check the 2nd value population near:
    ORDER BY country.population DESC LIMIT 5
```

Based on the feedback, the student now realizes where the final mistake was, corrects it, and submits the correct query:

```
SELECT city.name, country.name AS country, city.population
  FROM world.city
    JOIN world.country
      ON world.city.countrycode = world.country.code
```

```
    JOIN world.countrylanguage
      ON world.countrylanguage.countrycode = world.country.code
  WHERE continent = 'Africa'
    AND world.countrylanguage.language = 'French'
    AND world.countrylanguage.isofficial = 'True'
ORDER BY city.population DESC LIMIT 5;
```

The query finally passes all tests, the student's solution is accepted, and the student is allowed to move on in the coursework.

9. The SQLGrader Library

The SQLGrader library discussed in this paper is available on Github,

`https://github.com/femhub/sqlgrader`

under the Creative Commons Attribution-NonCommercial 4.0 International license. The library was originally developed for NCLab [20], but it can be used within other online learning platforms as well. It supports multiple SQL dialects.

The library should be installed on the server, where all grading takes place. Depending on the online learning platform, this might require using a suitable technology such as JSON to transfer SQL code from the client to the server. When the grading is finished, the feedback should be transferred back to the client, for the student to read.

Using the library involves the following typical steps:

Step 1: Cleaning SQL Code

Both the master solution and the student's solution are cleaned using the function `sql_clean_and_divide`. This function is built upon the open-source library SQLParse [22]. It removes comments and unnecessary white spaces, formats the code, and divides it into individual SQL statements. These statements are then analyzed using the corresponding statement graders.

Step 2: Executing SQL Code

After cleaning the master solution and the student's solution, both queries are executed, and the output is saved for further use.

Step 3: Grading

All statements in the student's query are checked with the help of the corresponding statement graders.

Step 4: Providing Feedback

If the student's solution passes all tests, then positive feedback is provided. If any of the tests fail, the feedback from graders is passed to the student, so that s/he can improve the solution.

10. Conclusion

We presented a novel self-paced, instructor-assisted approach to teaching SQL. The general methodology was described rather briefly, because it was already discussed in our previous papers. The emphasis of the present paper was on automated analysis and grading of SQL queries. Receiving instant feedback is extremely important for students, because this is what validates their understanding of the material and allows them to learn efficiently at their own pace. We introduced a number of SQL output analyzers and code graders, and explained how they are used to build Statement Graders for individual SQL statements. Finally, we presented the SQLGrader library which is freely available on Github, designed to automate the analysis and grading of complex SQL queries.

Appendix A. Source Code of the Row Analyzer

For illustration, below we present the source code of the functions `get_row_numbers` and `sql_row_analyzer`. This is a relatively small part of the SQLGrader library, but it gives the reader an idea of how the code looks like:

```
def get_row_numbers(student_rows, set_of_rows):
    """Get row numbers of a set of rows in a student result
    :param student_rows:
    student rows
    :param set_of_rows:
    set of rows to be found
    :return:
    string - list of row numbers where the set of rows is
```

```
            found in the student result
    """

    row_numbers = []
    # We update rows to hashable objects (tuple)
    student_rows_list = list(map(tuple, student_rows))

    for i in range(len(student_rows_list)):
        if student_rows_list[i] in set_of_rows:
            row_numbers.append(i+1)

    # Get only the first three items from the list
    sliced_list = row_numbers[:5]

    # Convert the sliced list to a comma-separated string
    result = ', '.join(map(str, sliced_list))

    # Add ellipsis if there are more than three items
    if len(row_numbers) > 5:
        result += ', ...'

    return result


def sql_row_analyzer(sol_rows, student_rows, sol_cols,
                     ordering):
    """Serves to grade rows
    :param sol_rows:
    solution rows
    :param student_rows:
    student rows
    :param sol_cols:
    solution column names
    :param ordering:
    should we match the order
    :return:
    list - Errors are returned or False if no problem is
           found
```

```
"""

# We use Python collections to very effectively do
# set operations with rows.
# First, we update rows to hashable objects (tuple),
# then create collections.
sol_rows_c = c(list(map(tuple, sol_rows)))
student_rows_c = c(list(map(tuple, student_rows)))

# Correct columns
rows_correct = sol_rows_c & student_rows_c
# Columns missing in the student result
rows_missing = sol_rows_c - student_rows_c
# Excess columns in the student result
rows_excess = student_rows_c - sol_rows_c

rows_text = []
if rows_missing or rows_excess:
    if rows_correct:
        pl = len(rows_correct) > 1
        rows_correct_text = str(len(rows_correct))
        rows_correct_numbers_text = \
        get_row_numbers(student_rows, rows_correct)
        rows_text.append(
            [
                True,
                "{} {} correct ({} {}). ".format(
                    rows_correct_text,
                    ("rows are" if pl else "row is"),
                    ("rows" if pl else "row"),
                    rows_correct_numbers_text
                )
            ]
        )

    # If the same number of rows is missing and not
    # expected this number of rows is "incorrect"
    if len(rows_missing) == len(rows_excess) > 0:
```

```
        pl = len(rows_missing) > 1
        rows_missing_text = str(len(rows_missing))
        rows_missing_numbers_text = \
        get_row_numbers(student_rows, rows_excess)
        rows_text.append(
            [
                False,
                "{} {} not correct ({} {}). ".format(
                    rows_missing_text,
                    ("rows are" if pl else "row is"),
                    ("rows" if pl else "row"),
                    rows_missing_numbers_text
                )
            ]
        )

    else:
        if rows_missing:
            pl = len(rows_missing) > 1
            rows_missing_text = str(len(rows_missing))
            rows_text.append(
                [
                    False,
                    "%s %s missing. "
                    % (rows_missing_text, ("rows are" \
                    if pl else "row is")),
                ]
            )

        if rows_excess:
            pl = len(rows_excess) > 1
            rows_excess_text = str(len(rows_excess))
            rows_excess_numbers_text = \
            get_row_numbers(student_rows, rows_excess)
            rows_text.append(
                [
                    False,
                    "{} {} not expected ({} {}). "\
```

```
                    .format(
                        rows_excess_text,
                        ("rows are" if pl else \
                        "row is"),
                        ("rows" if pl else "row"),
                        rows_excess_numbers_text
                    )
                ]
            )

rows_set_problem = False
if rows_text:
    rows_set_problem = True

# Should there be any differences in collections we
# know about it already.
# We still have no information about ordering.
# We do not have to comply to any specific ordering
# unless ordering == True
if not rows_set_problem and not ordering:
    return False

if not rows_set_problem and ordering:
    # We need to check ordering
    rows_unordered = [j for i, j in zip(sol_rows, \
    student_rows) if i != j]
    if rows_unordered:
        num_row_unordered = len(rows_unordered)
        if num_row_unordered > 3:
            return [
                [False, "%s rows are not ordered \
                correctly. " % (str(num_row_unordered))]
            ]
        else:
            rows_unordered_text = "\n".join(str(c) \
            for c in rows_unordered)
            return [
                [
```

```
                    False,
                    "These rows are not ordered \
                    correctly: <pre>%s</pre>"
                    % (rows_unordered_text),
                ]
            ]
    else:
        return False

# The message currently in the rows_text is adjusted
# if only one row is to be returned
if rows_set_problem and len(sol_rows) == 1:
    if len(student_rows) > 1:
        rows_text = [
            [
                False,
                "Only one row is expected. \
                The result has %s rows. "
                % (str(len(student_rows))),
            ]
        ]
    elif len(student_rows) == 1 and len(sol_rows) == 1:
        # Student has only one row, but the result
        # is different.
        student_row = student_rows[0]
        sol_row = sol_rows[0]
        row_differences = [i != j for i, j in \
        zip(sol_row, student_row)]
        row_different_vals = [i for i, j in \
        zip(student_row, row_differences) if j]
        cols_different = [i for i, j in zip(sol_cols, \
        row_differences) if j]
        pl = len(cols_different) > 1
        rows_text = [
            [
                False,
                "%s <code>%s</code> in the column%s \
                <code>%s</code> %s incorrect. "
```

```
                % (
                    ("Values" if pl else "The value"),
                    "(" + ", ".join(str(c) for c in \
                    row_different_vals) + ")",
                    ("s" if pl else ""),
                    ", ".join(str(c) for c in \
                    cols_different),
                    ("are" if pl else "is"),
                ),
            ]
        ]

    return rows_text
```

References

[1] D. Katsaros, M. Katsanou, T. P.I., A comparative study on teaching SQL programming through traditional and gamified approaches, Journal of Educational Computing Research 57 (8) (2020) 2028—2047.

[2] P. Visual, M. Sibley, H. S.P., Interactive visual learning tools for teaching SQL and database concepts, Education and Information Technologies 27 (2022) 527–543.

[3] M. Sabin, V. Kafali, Teaching SQL query formulation through semantic mapping and visualization, Journal of Computer Science Education 29 (4) (2019) 155–164.

[4] M. B. Olivier, P. L. Matray, Flipped classrooms in teaching SQL: Student perceptions and learning outcomes, Journal of Information Systems Education 29 (3) (2018) 155–164.

[5] J. Krogstie, E. A. Christensen, An experiment in teaching SQL using collaborative problem solving in online learning environments, Computers & Education 112 (2017) 92–105.

[6] P. Solin, Self-paced, instructor-assisted approach to teaching linear algebra, Mathematics in Computer Science 15 (4) (2021). doi:DOI:10.1007/s11786-021-00499-z.

[7] P. Solin, A. Freyer, Self-paced, instructor-assisted approach to teaching python programming, Mathematics in Computer Science 17 (2) (2023). doi:DOI:10.1007/s11786-023-00560-z.

[8] J. Melton, A. R. Simon, Understanding the new SQL: A complete guide, Morgan Kaufmann (1993).

[9] E. F. Codd, A relational model of data for large shared data banks, Communications of the ACM 13 (6) (1970) 377–387.

[10] S. Chaudhuri, An overview of query optimization in relational systems, Proceedings of the ACM SIGMOD International Conference on Management of Data (1998) 34–43.

[11] W. Archer, J. Davison, Graduate employability: The view of employers, London: The Council for Industry and Higher Education (CIHE) (2008).

[12] J. Andrews, H. Higson, Graduate employability, 'soft skills' versus 'hard' business knowledge: A european study, Higher Education in Europe 33 (4) (2008) 411–422.

[13] M. Mourshed, D. Farrell, D. Barton, Education to employment: Designing a system that works, McKinsey & Company (2012).

[14] D. Bennett, S. Richardson, P. MacKinnon, Enacting strategies for graduate employability: How universities can best support students to develop generic skills, National Centre for Student Equity in Higher Education (NCSEHE), Curtin University (2016).

[15] T. W. Burns, W. Yeoh, Bridging the sql skills gap: Curriculum and pedagogical improvements in an introductory database course, Proceedings of the EDSIG Conference on Information Systems and Computing Education (2015).

[16] J. F. Pane, B. A. & Myers, Improving the learnability of sql, Proceedings of the ACM CHI 2000 Conference on Human Factors in Computing Systems 12 (1) (2000) 53–60.

[17] T. M. Connolly, C. E. Begg, A constructivist-based approach to teaching database analysis and design, Journal of Information Systems Education 17 (1) (2006) 43–53.

[18] R. Radford, C. Lansley, Addressing the SQL skills gap: An experiential learning approach, International Journal of Database Management Systems (IJDMS) 8 (4) (2016) 1–16.

[19] A. Levine, S. Van Pelt, The great upheaval: Higher education's past, present, and uncertain future, Johns Hopkins University Press, Baltimore (2021).

[20] NCLab, `http://nclab.com/` (Accessed December 20, 2024).

[21] MySQL World Sample Database, `https://dev.mysql.com/doc/index-other.html` (Accessed December 20, 2024).

[22] SQLParse Library, `https://github.com/andialbrecht/sqlparse/` (Accessed December 20, 2024).