# Automated SQL Query Evaluations in Massive Database Courses

Ljiljana Brkić*, Igor Mekterović ** and Melita Fertalj***
*, **, *** University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia
Croatia
* ljiljana.brkic@fer.hr
** igor.mekterovic@fer.hr
*** melita.fertalj@fer.hr

***Abstract*** **- When assessing submissions in a massive course, using an Automated Programming Assessment System (APAS), can benefit both students and teachers. Students can expect fast and consistent assessment, while teachers benefit from a reduced workload. Acquiring proficiency in SQL is one of the core goals of any introductory or advanced database course. Evaluation of students' SQL queries differs from a general-purpose code evaluation, such as that for C or Java, by requiring a database on which the query will be evaluated and parameterized comparisons of the obtained recordsets. The evaluation using APAS is typically performed in such a way that the system executes two queries: the student query and the correct query provided by the course staff and compares the resulting datasets in terms of accuracy and completeness. When comparing the obtained datasets, there are a number of factors to consider, including the importance of tuple ordering and the relevance of column names. Moreover, some SQL statements create, alter, or delete database objects such as tables and indexes, and their correctness cannot be determined using a predefined output dataset. In order to do so, every SQL question in APAS must reference some test database (populated with data). With hundreds of students enrolled, it becomes technically challenging to execute and evaluate their queries in real time, especially as these databases begin to pile up due to courses development and evolution. In this paper, we comment on a possible solution and present our approach with APAS that uses multiple cloned instances of the test database while supporting the aforementioned specifics of SQL query evaluation.**

***Keywords - Automated Programming Assessment System, Dataset Comparison, Multiple Database Instances, SQL***

## I. INTRODUCTION

Along with database schema modeling, learning to use the SQL query language correctly and efficiently is one of the most important goals of an introductory database course. Although the quality of modeling (at least at a conceptual level) seems almost impossible to assess in an automated or at least computer-aided way, the correctness of SQL statements seems more suitable for automated or computer-aided assessment. The traditional approach to assessing student proficiency in database courses has been hampered by the challenges of grading large volumes of SQL queries manually. This issue is especially noticeable in higher education environments, where the increasing number of students can overwhelm the capacity of instructors to provide timely and accurate feedback.

To address the challenges associated with teaching and assessing SQL statements, automated SQL query evaluation (AQE) tools have emerged. Examples include eSQL, one of the earliest tools proposed for teaching query processing concepts but not utilized for evaluation [1]. Another tool is SQL-Tutor, developed at the University of Canterbury, offering semantic feedback, also not employed for assessment [2]. AsseSQL, created at the University of Technology in Sydney and similar tool, named SQLator, developed by the University of Queensland [3] provide binary grading for queries submitted by students. Additionally, SQLify incorporates computer assisted learning and assessment using a sophisticated automatic grading system in combination with peer review [4]. Finally, an example of a tool that can serve as a tutoring tool and in assignment (or even exam) assessment is a tool named aSQLg [5]. These tools represent efforts to address the complexities of teaching and assessing SQL in higher education.

The automation of the grading process through AQE systems offers advantages for both students and educators. Students can expect timely and consistent grading potentially accompanied by feedback that guides students towards deeper understanding and improved query-writing skills. By reducing or eliminating the need for manual grading, AQE systems contribute to improved teacher efficiency, freeing up time for activities such as refining course materials, developing richer assignments, or engaging in more meaningful student interactions [6].

Despite the potential of AQE systems, their implementation in higher education courses has faced several challenges. One such challenge involves the creation of a reliable AQE system that can assess a variety of SQL queries—from simple to complex—and handle different data structures and formats. Moreover, to help students develop their query-writing abilities and to help them understand the underlying ideas, AQE systems must not only identify errors but also provide insightful feedback. that should be accessible regardless of location and time. Integrating additional data sources, such as student enrollment records for specific time periods, details on instructors and staff, and a comprehensive database of test questions and answers, becomes imperative for enhancing the functionality of AQE systems.

Many higher education institutions have profited from the use of Automated Programming Assessment System (APAS) for evaluating students' assignments, especially in

massive courses. However, APASs often lack support for evaluating SQL queries. Instead of developing the AQE component as an inseparable, integral part of APAS that is used for automatic evaluation of different types of assignments, a possible solution is to develop the AQE tool independently as a component that can be plugged in and integrated into the existing APAS. Such an approach was, for example, used in the integration of the aSQLg tool in APAS WebCAT [7], as well as in the CMS of the University of Applied Sciences and Arts in Hannover.

At the University of Zagreb Faculty of Electrical Engineering and Computing (FER), we have been developing and actively using an APAS called Edgar for eight years. Contrary to many other APASs Edgar was initially developed to automate the assessment of SQL programming assignments, and this component is inseparable from the rest of the system. Edgar extends its support to the evaluation of assignments written in any programming language executable on a Linux platform (C, C#, C++, Java, Python, etc). More details can be found on our previous papers [8][9][10].

The distinction between evaluating procedural code, such as C or Python, and unprocedural code like SQL queries lies in their assessment methodologies. In procedural code evaluation, the process often involves the use of test cases, predefined inputs that are fed into the program, and the expected outputs are known in advance. The assessment boils down to comparing the actual outputs generated by the code with the expected outputs specified by the test cases. On the other hand, unprocedural code, like SQL queries, follows a different assessment paradigm. Instead of relying on predefined test cases, the evaluation focuses on comparing datasets. SQL queries retrieve and manipulate data from databases, and the assessment involves examining the results produced by these queries. This comparison is often parametrized, considering factors such as the order of rows, the significance of column names, and other contextual considerations.

In this paper, we present the approach that Edgar uses to define SQL assignments and the corresponding environment needed to carry out and evaluate submissions from a course with more than 500 students enrolled each academic year.

## II. APAS SETUP FOR A DATABASE COURSES

SQL query evaluation is unique in that it always executes in the context of a database. Therefore, to serve as a repository for these databases, a database server (or servers) must be included in the architecture of an APAS that supports the automatic assessment of SQL queries.

### *A. What type of database is suitable for teaching and examining courses related to databases?*

SQL assignments are created keeping in mind the underlying database schema that the queries are run against. In terms of the number and types of objects it contains, the database needs to correspond with the course content. If it is used in an introductory course, relations with attributes of basic data types and defined common constraints (primary and foreign keys or CHECK constraints) are sufficient. However, in advanced courses that teach topics like full text search or geospatial concepts, the schema must include, for example object-relational data types and accompanying functions/operations. The issue of selecting an appropriate Database Management System (DBMS) inevitably comes up here. Almost all relational DBMSs will be suitable to host an introductory database. However, when it comes to advanced concepts, one should carefully compare the requested with the available functionalities.

Although it may seem that a database of suitable schema, topic matter and content can easily be found online, our experience is that this is not the case. Almost all databases we use in classes were created in the following way. The process would start by creating a database schema considering the concepts we teach in the course (e.g. recursive queries are easier to teach and test if there is at least one reflexive relationship in the schema). We would then identify constraints that can be declaratively defined as well as business rules that need to be respected when generating data (e.g. a student cannot pass a course before enrolling it). After that, we would determine how many tuples each relation should have and finally generate the tuples programmatically.

The introductory databases course, in our institution, is attended by fourth-semester undergraduate students pursuing a bachelor's degree in computer science, while advanced courses are reserved for those enrolled in the master's degree program. For the past eight years, we have been using PostgreSQL RDBMS [11] to teach relational database concepts in the introductory course. Additionally, for an extensive period, we have utilized a database named *studAdmin* for homework assignments and laboratory exercises. This database includes data relevant to actual processes within student and academic administration. From the very beginning of the course, students are introduced to the *studAdmin* database schema. Detailed explanations are provided that clarify the roles of individual relations, their mutual connections, and the meanings of attributes. While derived from our real-world information system, the schema is adapted for educational purposes, featuring simplified structures, reduced data volumes, and anonymized attribute values to maintain data confidentiality.

A list of the *studAdmin* database tables is included in TABLE I, along with metadata detailing the number of attributes and different attribute data types, as well as the count of constraints and tuples. Ensuring an adequate number of tuples within the relations is imperative to forestall the possibility of predicting query results. However, it is equally crucial to avoid performance-related concerns. The size of the datasets requiring comparison and display in the browser escalates proportionally with the number of tuples.

In addition to the previously mentioned, we are currently using nine other databases for teaching and assessing one introductory and one advanced course. While some databases are utilized in homework and lab exercises, others are used in exams, and most frequently, new database versions are made specifically for exams.

TABLE I. STUDADMIN DATABASE TABLE METADATA

| Table name | Num. of attrib. | Num. of diferent datatypes | Num. of constraints | Num. of tuples |
|---|---|---|---|---|
| attendance | 4 | 2 | 3 | 11338 |
| classroom | 2 | 2 | 1 | 46 |
| county | 2 | 2 | 1 | 22 |
| course | 4 | 4 | 1 | 66 |
| courseacyear | 3 | 2 | 2 | 167 |
| coursegroup | 5 | 3 | 5 | 810 |
| enrolledcourse | 4 | 3 | 3 | 3579 |
| exam | 6 | 4 | 4 | 4325 |
| examterm | 4 | 3 | 2 | 324 |
| orgunit | 3 | 2 | 1 | 134 |
| student | 8 | 3 | 4 | 529 |
| studentgroup | 3 | 2 | 3 | 135 |
| teacher | 8 | 4 | 3 | 335 |
| town | 3 | 3 | 2 | 275 |

### *B. How to mitigate performance issues when evaluating SQL queries in courses with several hundred students*

TABLE II presents the data for two courses heavily reliant on Edgar APAS for instructional purposes and assessments during the academic year 2022/2023. The assessments encompass a comprehensive range of evaluation methods, spanning homework assignments, laboratory exercises, projects, midterm examinations, final examinations, and regular assessments. SQL questions constitute only a fraction of the total number of student question-exam instances, accounting for 27.32% in the Databases course and 27.47% in the Advanced Databases course. This distribution reflects the versatility of Edgar, which accommodates various question types, including multi-correct multiple-choice questions, free-format text responses, and peer assessments. Peer assessment is particularly valuable for evaluating open-ended assignments related to Entity-Relationship (ER) modelling and relational modelling.

TABLE II. STATISTICAL DATA ON COURSE EXAM-INSTANCES

| Course name | Databases. | Advanced Databases |
|---|---|---|
| **Study level** | Bachelor | Master |
| **No of tutorials** | 13 | 8 |
| **No of exams** | 41 | 22 |
| **No. of students enroled** | 567 | 155 |
| **No of exam instances** | 10607 | 2707 |
| **No of question-exam instances** | 109570 | 14406 |
| **No of SQL question-exam instances** | 29936 | 3958 |

Among all assessments administered via Edgar, the midterm and final exams impose the most strain on the computer and software infrastructure due to their concurrent completion by all students within a short timeframe (typically 120 minutes). These exams are conducted under supervised conditions in designated faculty building classrooms, where access is restricted to internet locations essential for Edgar's operation and exam administration via network configuration. This setup effectively prevents students from collaborating or exchanging solutions during the exam. In contrast, other exam types, such as unsupervised assignments completed at home over a span of seven days or laboratory exercises involving a smaller number of students completing tasks simultaneously (up to 170 students), pose less demand on resources.

To minimize the common issues that arising from concurrent read/write operations during large-scale exams, we opted to employ multiple databases for evaluating student solutions. Currently, there are 10 instances of this database, so rather than having 567 student queries evaluated on one database, an equal distribution allows for around 57 queries to be assessed per database.

Due to the current setup, which consists of 10 instances of the same database, as opposed to 567 student submissions being evaluated in the same database, with an even distribution, approximately 57 student queries are evaluated on a single database at a time. This setup, with 10 instances of the same database, allows for a more balanced distribution of workload during assessments.

Moreover, our utilization of 10 databases for the two mentioned courses, each with 10 instances of distinct databases, increases the total number of instances to 100, presenting significant maintenance challenges. To address this, we implemented a solution wherein we segregated the relations of each of the 10 unique databases into separate schemas, consolidating all 10 schemas within a single database. This approach facilitates the accommodation of tables with identical names across different schemas, a beneficial feature considering that many of our databases feature tables named "student" or "person." Subsequently, we replicated an additional nine identical databases using a straightforward SQL command derived from the template database.

Figure 1 depicts the PostgreSQL database server on the left, showing 10 instances of databases utilized within Edgar for evaluating SQL queries (*examdb01* through *examdb10*). On the right, the schemas within the *examdb04* database are illustrated, comprising five schemas employed in the Databases course and an additional five utilized in the Advanced Databases course. Although the schema names in the illustration are generic and do not align with actual names, our operational system indeed encompasses 10 schemas, one of which is *studAdmin*, containing the database described earlier in this chapter, where the tables from TABLE 1 were created.

The SQL questions within Edgar requires the use of objects, specifically tables, from a single schema which now consolidates all tables previously distributed across separate databases.

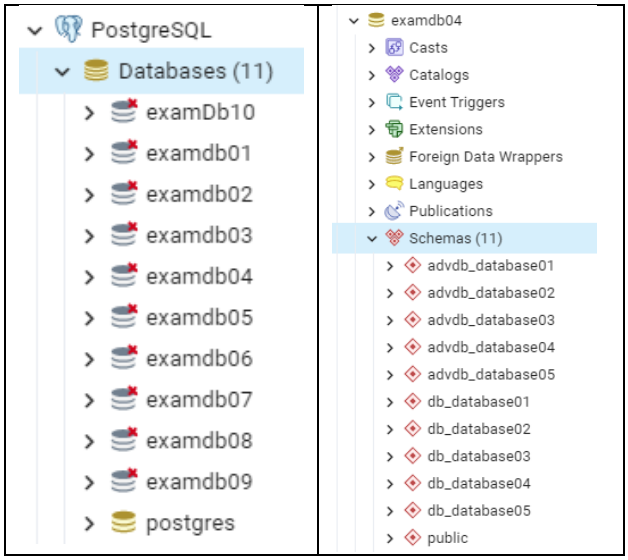


Figure 1. Database instances used to validate SQL queries on database courses

We don't find this limitation to be an issue because, in the past, we didn't use tables from distinct databases in the same query. We don't even require these capabilities on advanced courses so far.

Consequently, each question in Edgar mandates the definition of a schema within the execution environment of the SQL query. For queries involving SELECT, INSERT, UPDATE, or DELETE statements, the tables referenced belong to the specified schema, while CREATE [TABLE/INDEX/FUNCTION/...] statements generate objects within the schema associated with the question. PostgreSQL, akin to certain other DBMSs, employs the concept of a search path, dictating the order in which schemas are consulted during query evaluation. Prior to query execution, Edgar configures the SEARCH_PATH to prioritize the schema relevant to the question, followed by the public schema housing common objects. A typical statement defining the SEARCH_PATH for SQL queries utilizing *studAdmin* database tables looks like:

```
SET SEARCH_PATH TO studAdmin, public;
```

This configuration avoids the need for qualified names for tables (include the schema name as a prefix) within the SQL query solution, which are tedious to construct.

## III. SQL QUESTION DEFINITION AND EVALUATION IN EDGAR

In Edgar, each programming question, whether it pertains to SQL, Java, C, or any other language, is structured with three code snippets: the required source code (called *SQL Answer* in SQL questions), an optional prefix, and an optional suffix. These components are combined to form the complete program. When completing the assignment, students are solely responsible for providing the required source code. However, the question author can add the optional prefix and suffix, which are incorporated before (prefix) and after (suffix) the required source code, respectively. Despite being part of the overall solution, these optional segments are always authored by the question creator.

Sometimes, when it is acceptable, optional components are even given in the assignment text and can be used when solving it. This approach provides flexibility for teachers to creatively design scenarios and support diverse programming assignments. We have addressed this in more detail in our earlier work [10]. For instance, in an Introduction to Programming course, assignment may involve the evaluation of function that students should write for a provided prototype and prescribed functionality. In some cases, students are not required to write the main program themselves. Instead, teachers compose the main program and include it as a suffix component of the solution, which is then provided to students within the assignment text.

In SQL questions, the implementation of the prefix-source-suffix scheme makes it easier to evaluate record changes, deletions, and even Data Manipulation Language (DML) expressions like CREATE INDEX and ALTER TABLE in addition to SELECT statements. In the code's suffix section, a query to the system catalogue is used to do this.

Consider the exam from the Figure 2 which consists of one SQL assignment that expects a single SELECT statement in response. The question refers to the *studAdmin* database and students know this in advance. By pressing the Run button, the student initiates the execution of his own SQL code. The program determines the database in which the query will be executed among the 10 available databases, considering an even distribution of the load across all databases. The configuration of the question includes information about the schema to which the query pertains.

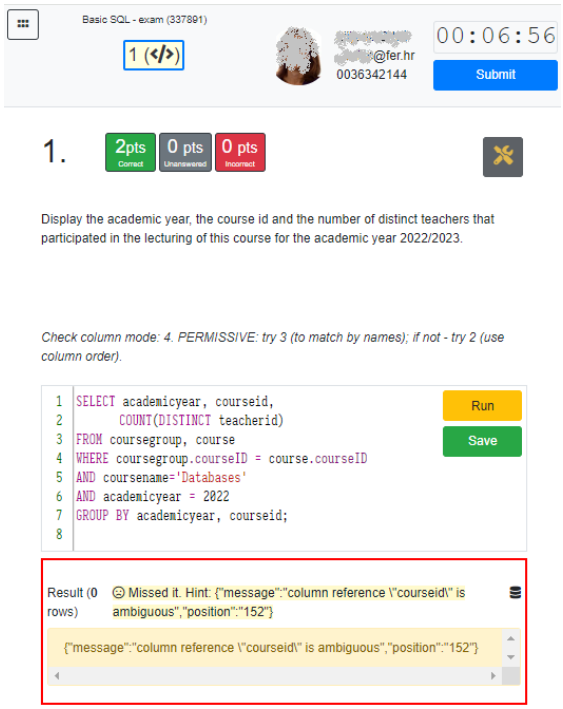


Figure 2. An exam with a single SQL question in Egar that expects a single SELECT statement as a solution.

On the lower part of the screen, syntactical errors in the query are visible, enabling the student to correct the error based on feedback.

Figure 3 displays the screen for question definition used by the course teacher. The correct solution is provided under the SQL Answer section, while (this time) the SQL prefix, SQL suffix, and Presentation query parts remain empty.

TABLE III shows the complete code constructed based on SQL prefix - SQL Answer - SQL suffix and Presentation query parts. The SQL Answer section provided by both the teacher and the student (highlighted in light blue) is inserted between the prefix and suffix sections (highlighted in yellow), which are left empty in this case. When the complete SQL code is assembled for the teacher and the student, record sets presenting the correct solution and the student's solution are acquired and subsequently compared using options visible on Figure 3 above the SQL Answer label. In this particular case, the student's solution ended with an error, so it will not produce a recordset, and there will be no comparison of the recordsets.

In Edgar, the SQL-runner operates within a transaction that is consistently rolled back, effectively creating a sandbox environment. This approach does not limit the execution solely to read-only SELECT statements. Rather, it allows for the execution of various DML statements such as INSERT, DELETE, UPDATE, as well as DDL statements like ALTER TABLE, CREATE TABLE, INDEX, and so forth. The SQL-runner executes these statements, retrieves the necessary datasets, and subsequently rolls back the transaction.

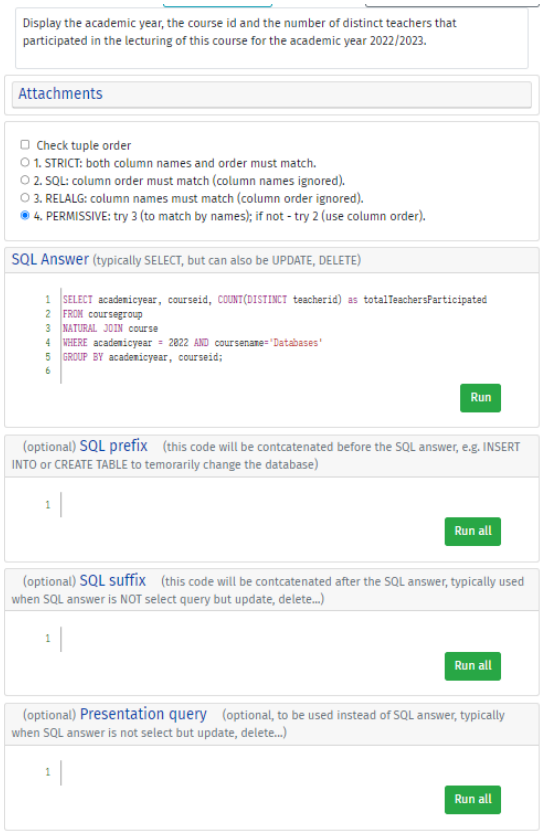


Figure 3. Definition of the question from the Figure 2 by the teacher

TABLE III. VALUATING CORRECTNESS FOR THE EXAMPLE ON FIGURE 2 AND FIGURE 3

| | Assembled code to execute: |
|---|---|
| Teacher | BEGIN WORK;<br>SET SEARCH_PATH TO studAdmin, public; |
| | |
| | SELECT academicyear, courseid,<br>COUNT(DISTINCT teacherid) as<br>totalTeachersParticipated<br>FROM coursegroup<br>NATURAL JOIN course<br>WHERE academicyear = 2022<br>AND coursename='Databases'<br>GROUP BY academicyear, courseid; |
| | |
| | ROLLBACK WORK; |
| Student | BEGIN WORK;<br>SET SEARCH_PATH TO studAdmin, public; |
| | |
| | SELECT academicyear, courseid,<br>COUNT(DISTINCT teacherid)<br>FROM coursegroup, course<br>WHERE coursegroup.courseID =<br>course.courseID<br>AND coursename='Databases'<br>AND academicyear = 2022<br>GROUP BY academicyear, courseid; |
| | |
| | ROLLBACK WORK; |

It is important to emphasize that, unlike code questions or multiple-choice questions, the correctness of SQL questions is binary - 0% or 100%. Teachers can subsequently correct the points if necessary, and we regularly do this for submissions graded with 0%.

Figure 4 shows the definition of a question whose answer is to create a constraint in the database. In this question, in addition to the SQL Answer, which is expected from the student in that or a similar form, the SQL prefix and the SQL suffix part were also used.

TABLE IV outlines the complete SQL code that is evaluated for the teacher's and student's solution. Since the primary and foreign key constraints have already been created for the tables in the studAdmin database, before the SQL code written under SQL Answer starts creating them, they should be dropped first (statements under SQL prefix). After the constraints are created (commands under SQL Answer), the SELECT command (SQL suffix) is used to check whether the constraints were created correctly. Edgar_keys is a view created in the databases examdb01-examdb10 in the schema public, which obtains all primary and foreign key constraints. SQL prefix and SQL suffix part are inserted into the complete solution before (prefix) and after (suffix) the teacher's correct answer and the student's answer. The only variable part is the SQL code shown in the table in light blue.

The student will be presented with a recordSet returned with the SQL suffix statement.

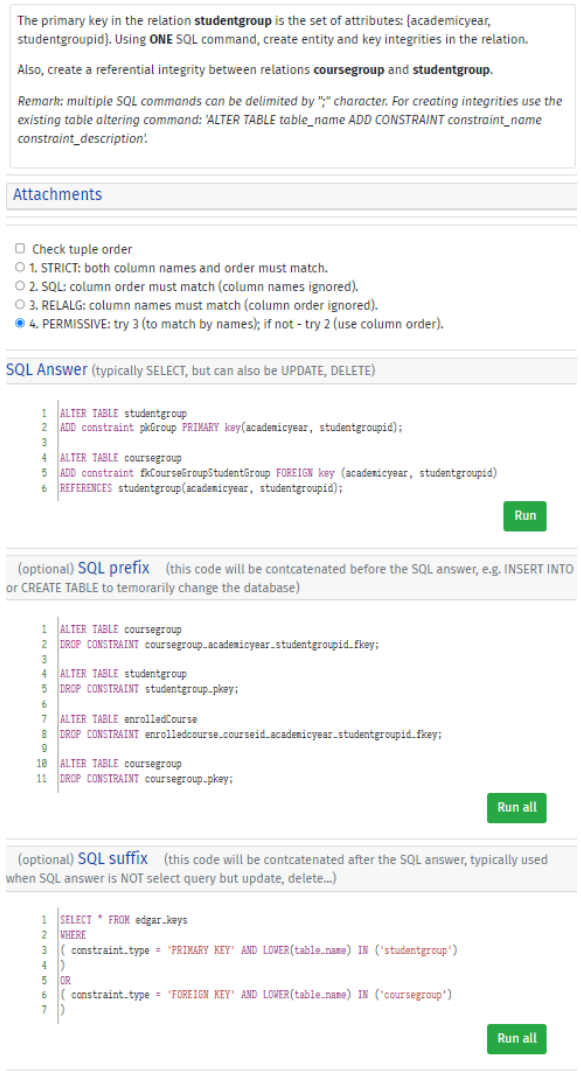


Figure 4. Definition of a question whose answer is to create a constraint in the database

## I. CONCLUSION

In this paper, we discussed the use of the Edgar APAS for teaching and assessing SQL queries within database courses. Using several instances of the exam database with multiple schemas is one of the strategies we identified to help us handle system component administration issues and increase APAS performance while reviewing SQL questions on mass examinations.

Additionally, we described how to define and evaluate SQL queries in Edgar, emphasizing the usage of programmable templates. We showed how to run SQL queries in a safe sandbox setting, which made it possible to assess a variety of SQL statements.

Overall, this paper provides insight into the challenges and complications of putting into practice an APAS designed especially for teaching and evaluating SQL queries in database courses.

TABLE IV. EVALUATING CORRECTNESS FOR THE EXAMPLE ON FIGURE IV

| | Assembled code to execute: |
|---|---|
| Teacher | `BEGIN WORK;`<br>`SET SEARCH_PATH TO studAdmin, public;` |
| | `SQL prefix` |
| | `ALTER TABLE studentgroup`<br>`ADD constraint pkGroup`<br>`PRIMARY KEY(academicyear,`<br>`studentgroupid);`<br><br>`ALTER TABLE coursegroup`<br>`ADD constraint`<br>`fkCourseGroupStudentGroup FOREIGN key`<br>`(academicyear, studentgroupid)`<br>`REFERENCES studentgroup(academicyear,`<br>`studentgroupid);` |
| | `SQL suffix:` |
| | `ROLLBACK WORK;` |
| Student | `BEGIN WORK;`<br>`SET SEARCH_PATH TO studAdmin, public;` |
| | `SQL prefix` |
| | `Student's solution` |
| | `SQL suffix:` |
| | `ROLLBACK WORK;` |

## REFERENCES


[1] R. Kearns, S. Shead, and A. Fekete, "A teaching system for SQL," *ACM Int. Conf. Proceeding Ser.*, vol. Part F1293, pp. 224–231, 1997, doi: 10.1145/299359.299391.

[2] A. Mitrovic, "Learning SQL with a computerized tutor," *Poceedings Conf. Integr. Technol. into Comput. Sci. Educ. ITiCSE*, no. March 1998, pp. 307–311, 1998, doi: 10.1145/273133.274318.

[3] S. Sadiq, M. Orlowska, W. Sadiq, and J. Lin, "SQLator - An online SQL learning workbench," *SIGCSE Bull. (Association Comput. Mach. Spec. Interes. Gr. Comput. Sci. Educ.*, vol. 36, no. 3, pp. 223–227, 2004, doi: 10.1145/1026487.1008055.

[4] S. Dekeyser, M. de Raadt, and T. Y. Lee, "Computer assisted assessment of sql query skills," *Conf. Res. Pract. Inf. Technol. Ser.*, vol. 63, pp. 53–62, 2007.

[5] C. Kleiner, C. Tebbe, and F. Heine, "Automated grading and tutoring of SQL statements to improve student learning," *ACM Int. Conf. Proceeding Ser.*, pp. 161–168, 2013, doi: 10.1145/2526968.2526986.

[6] J. Heil and D. Ifenthaler, "Online Assessment in Higher Education: A Systematic Review," *Online Learn. J.*, vol. 27, no. 1, pp. 187–218, 2023, doi: 10.24059/olj.v27i1.3398.

[7] "Web-CAT - Web-CAT." https://web-cat.org/projects/Web-CAT/ (accessed Feb. 05, 2024).

[8] L. Brkić, I. Mekterović, M. Fertalj, and D. Mekterović, "Peer assessment methodology of open-ended assignments: Insights from a two-year case study within a university course using novel open source system," *Comput. Educ.*, vol. 213, p. 105001, May 2024, doi: 10.1016/J.COMPEDU.2024.105001.

[9] I. Mekterovic, L. Brkic, and M. Horvat, "Scaling Automated Programming Assessment Systems," *Electron. 2023, Vol. 12, Page 942*, vol. 12, no. 4, p. 942, Feb. 2023, doi: 10.3390/ELECTRONICS12040942.

[10] I. Mekterovic, L. Brkic, B. Milasinovic, and M. Baranovic, "Building a comprehensive automated programming assessment system," *IEEE Access*, vol. 8, pp. 81154–81172, 2020, doi: 10.1109/ACCESS.2020.2990980.

[11] "PostgreSQL: The world's most advanced open source database." https://www.postgresql.org/ (accessed Feb. 07, 2024).