Automated Grading and Tutoring of SQL Statements to Improve Student Learning

Carsten Kleiner
University of Applied Sciences&Arts
Ricklinger Stadtweg 120
30459 Hannover, Germany
carsten.kleiner@hs-hannover.de

Christopher Tebbe
University of Applied Sciences&Arts
Ricklinger Stadtweg 120
30459 Hannover, Germany
christopher.tebbe@hs-hannover.de

Felix Heine
University of Applied Sciences&Arts
Ricklinger Stadtweg 120
30459 Hannover, Germany
felix.heine@hs-hannover.de

ABSTRACT

In this paper we present a concept and prototypical implementation of a software system (aSQLg) to automatically assess SQL statements. The software can be used in any introductory database class that teaches students the use of SQL. On one hand it increases the efficiency of grading students submissions of SQL statements for a given problem statement by automatically determining a score for the statement based on different aspects. On the other hand it may also be used to improve student learning of SQL statements by enabling them to continuously (re-)submit their solutions and determine improvements in quality by comparing the automatically determined scores. In order to keep the administrative overhead for using it minimal we have implemented the software in a way that it may be plugged into any course/learning management system with minimal overhead. We have used it in conjunction with WebCAT as well as our own proprietary course management system. Student feedback collected after its first usage in a database class shows promising results for future usage of the system.

Categories and Subject Descriptors

K.3 [**Computers and Education**]: Computer and Information Science Education, Computer Uses in Education; H.2.3 [**Database Management**]: Languages

General Terms

Computer science education, information systems education, distance learning, SQL

Keywords

Automated grading, automated assessment, tutoring, learning management system, SQL

Koli Calling '13, November 14-17 2013, Koli, Finland

978-1-4503-2482-3/13/11 ...$15.00
http://dx.doi.org/10.1145/2526968.2526986.

1. INTRODUCTION AND MOTIVATION

Apart from database schema modeling, learning to use the SQL query language in a correct and efficient manner is among the most important goals of an introductory database system course. Whereas quality of modeling (at least on the conceptual level) seems almost impossible to assess in an automated or at least computer-assisted manner, the correctness and efficiency of SQL statements seems more appropriate to automated or computer-supported grading.

Automated or computer-supported grading is particularly important in introductory database system courses, since on one hand the number of students enrolled is rather large and on the other hand funds tend to be low and proficient students to perform grading and support lab sessions are difficult to find. Thus in order to improve efficiency of the educational institution as well as making individual assessment feasible, computer assistance is very helpful in this context.

Another aspect mandating a web-based system to be employed in computer science education in general is the possibility to provide students with space- and time-independent access to the outcome (result and contextual feedback) of exercises they submitted. At first this improves the attractiveness of the particular institution for students; also it may improve student learning outcomes since students can decide on their own when and how to work on exercises. It can also be useful to improve performance of students in distance learning classes by including electronic tutors as described in [12] for database courses. Thus using a web-based automated assignment grader is highly recommended.

For all these advantages to play a major role, it is not sufficient to develop and use a tool for a database system course alone. It is rather important to employ a single system that may be used in several different courses throughout the curriculum. By doing so, many classes can benefit from the aforementioned advantages. Also usage of the system by the students becomes more efficient and less error-prone since they are already used to the general functionality and look-and-feel of the software in place. Consequently, we never aimed at building a new course management system (CMS). We rather implemented it in a way that it may plugged into existing CMS with minimal effort. We have used it as a plug-in to WebCAT ([5]) as well as our university's proprietary CMS.

We have developed a tool called aSQLg which can perform an automated assessment of student submitted SQL queries. It may on one hand be used as a tutoring tool to improve the quality of the student results. This is achieved by them trying to improve their scores with repeated submission of hopefully better solutions based on the feedback. On the other hand it can also be used for improving efficiency in assignment (or even exam) assessment; in

this case the number of possible submissions by a student can be restricted, thus delivering a fair assessment of the quality of each solution without the (in this case undesired) tutoring capabilities. The tool had been used in first experiments based on real database course assignments with a few students at first. After achieving some maturity it has been used in our regular introductory database system classes in 2012 and 2013 in two different departments of our university. Even though the number of students enrolled is too small (lower three digit number in total) to derive statistically significant results important positive and negative feedback has been collected and will be discussed here.

This paper is organized as follows: after reviewing related work in the areas of automated grading as well as support for computer-supported learning of SQL we will present the concept which forms the foundation for our implementation in section 3. The concept will then be illustrated by giving some example statements and results which have been used in the real course in section 4. Thereafter we briefly explain the implementation which further illustrates the general functionality in section 5. We present some interesting aspects from the student evaluation in section 6 before we finally conclude with a summary and ideas for future work.

2. RELATED WORK

Publications related to our work may be roughly categorized into two areas, namely improving learning of SQL on one hand and automated grading and assessment on the other hand. Almost no work to the same level of detail as ours is known in the intersection of the two which is one of the major contributions of this paper. We briefly review relevant literature from each of the two areas below.

The first group of related publications focuses on the learning process of developing SQL queries by students. In [15] an intelligent SQL tutoring system is described. The system has been extended to use a web-based front-end and is focused on improving student learning by giving them real-time feedback on the submitted SQL queries. It supports them in working on the queries until they are correct. Similarly [12] describes a tutoring system for database system courses which is also focused on improving student learning by immediate web-based feedback. In contrast to [15] it seems to also support other aspects of database systems basics apart from SQL queries. Since the concept we propose in this paper is also targeted towards automated grading within a CMS rather than only focusing on assisting student learning, the tools described in the previously mentioned references could complement our concept. Ideas from these tutoring systems could be used to improve our feedback component in the future.

There are also a number of papers focusing on the grading aspect of SQL queries: the work described in [17, 18] introduces an assessment strategy for SQL queries that may be used in web-based tutoring and assessment systems. It is designed in order to improve student learning since the way students learn is greatly influenced by the way the assessment is performed. The system focuses on interactive query refinement rather than a mere grading of a given solution as in our case. Our grading approach is more complex and detailed though. In [8] the authors describe a tool (SQLify) for partially automated grading of SQL queries using an elaborate algorithm for assessment. This tool focuses heavily on peer-reviews and interaction in order to improve student learning. Thus it still requires a lot of manual work as opposed to our software. In [4] the authors sketch a system architecture which seems to include both a (newly designed) tutoring as well as a scoring component. No details on the grading component are explained though since the short article focuses on the overall system architecture.

In [7], an application based on the GNU SQL Tutor (see [1]) is described. It contains both a tutorial part and an exam part. The grading part is not explained in detail in the paper. The grading score seems to be binary for a single question.

SQL-KnoT [6] features question templates, which are used to generate actual questions increasing the variability. We believe this feature to be highly valueable in our setting and plan to integrate similar functionality.

To test whether a given student's SQL statement is a valid answer to a given problem, most tools (including our aSQLg), compare the outcome of the statement with a reference solution. A different approach for this problem is followed by SQLator [20]. This tool uses various heuristics to detect equivalence between the solution and the reference statement. The drawback is that possbily correct solutions are not detected. An intermediate aproach is followed by [9], who uses result comparison first to check correctness, and semantic comparison by heuristic reformulation rules to give hints to the student.

On the other hand at institutions with a large student to instructor (or TA if present) ratio one would probably prefer to focus on the automated grading portion rather than on the interactive approach to student learning since this increases efficiency. Therefore we advocate the inclusion of the SQL grading component into a general purpose web-based grading tool such as WebCAT ([5]) or general purpose CMS such as Moodle. There are also other learning management systems described in the literature: [16] introduces a generic automated grading system which may be used for different types of courses. Good experiences are reported for different programming classes; the system is not web-based though, but rather requires a full client application. Also it is not obvious on how it could be extended to cater for the specifics of SQL grading without additional information. Other similar systems for programming or other computer science courses are described in [10, 11, 14]. In [19] there is an interesting overview of general purpose as well as course specific learning management systems (LMS) that had been introduced in the literature. The overview does not only include course management but also lists a lot of systems focusing on improving student learning along with an impressive list of references. Any of the systems listed in the online submission and automated assessment section of that paper could in principle be used as foundation for implementation of our concept.

3. AUTOMATED SQL GRADING CONCEPT

We will now present the concept of how our SQL grading is performed in detail. The concept is visualized in figure 1 as a flow diagram starting with loading all statements, which have been provided. It finishes with setting the points for the graded assignments and delivering them to the student. The complete flow of all actions during the grading process is as follows:

1. Load all SQL-statements
2. Check one statement for forbidden elements (optional)
3. Check if the statement equals reference solution (optional)
4. Check for syntactical correctness (points)
5. Check statement cost (points)
6. Check correctness of result of the statement (points)
7. Check statement style (points)
8. After all statements processed: add points & generate report
9. Additional manual grading of statements (optional)

10. Update total grade

At the beginning of the grading process all statements of an assignment are loaded into the plug-in. Optionally it is possible to load a reference solution, too. After all statements have been loaded, the statements are analyzed individually, one statement at a time. The first thing checked is, whether a statement contains forbidden elements or not. This step is optional and can be selected and configured by an instructor. The filter uses a set of allowed or forbidden elements, so whitelists or blacklists may be used for filtering. To get all elements of a statement a SQL-parser called JSqlParser [2] is used. The parser reads a statement and creates a tree-structure of all elements. Every element is represented by a corresponding Java class. All values in a statement are saved in variables of types like Integer, String and so on. Such classes are used to check whether an element of the statement is allowed or not. JSqlParser is used in some later steps, too. If a statement is filtered out by the statement filter, it is ignored in subsequent steps and the next statement gets checked by the filter.

If a statement is allowed, the statement is compared to the reference solution. If they match the statement is correct, gains the maximum points and the grading is complete for this statement. Because using a reference solution is optional, this step is performed only when it is available.

If a statement does not match, the next step is performed. The test of syntactical correctness of a statement is performed using the database. We do not use the JSqlParser here as it did not accept all possible queries according to the specific database system dialect used (Oracle) in our class turning out to be a major source of frustration on the student side. Syntactical correctness is determined by using the query cost of a statement. If the result does not contain an error, the syntax of the statement is correct and the syntactical correctness points for the statement are awarded. If the syntax contains errors, the statement is discarded. If the syntax is correct the query cost which has already been retrieved is evaluated. The instructor can choose a limit of the maximum allowed cost, to prevent the execution of a long running statement which could block the rest of the statement grading like a denial of service attack. If the query cost of the statement is not acceptable, the check for correctness is skipped.

Evaluating the cost of a statement makes sense only, when it is at least partly correct. That's why the syntactical correctness is checked first. The number of points awarded for efficiency depends on how close the query cost of the students statement is to the reference solution or to a given value from the configuration file. When the cost is not to high the correctness of the statement can be checked using the statement in listing 1 which should return no results.

```
( (<student-statement>)
    UNION
  (<reference-statement>))
MINUS
( (<student-statement>)
    INTERSECT
  (<reference-statement>));
```

Listing 1: Statement used to compare the student statement with the reference solution

In the practical experiments with the students it turned out that such a strict correctness check led to a lot of frustration. This happened because students had problems finding the exact correct solution without any support. Thus we introduced two additional steps in correctness checking which helped the students correct their previous errors. At first before executing the statement in listing 1 we checked if the datatypes of the student and the reference solution match and are in the correct order. If that was not the case the student is informed by a corresponding message. After the execution in order to be able to verify correct usage of sorting there was an additional check if the sorting of the student had been the same as in the reference solution. This check can be activated for each individual problem as it is not required for all statements. These changes are subsumed in figure 1 in the box check correctness.

Checking the correctness of a student statement using the statement 1 is only possible if a reference solution is available. If not, the student's statement is executed in the database and the result set is saved for manual grading. If the statement is used, the result sets of the student statement and the reference solution are compared to each other. If they match, the statement gets the maximum correctness points possible. Currently partial credit for correctness cannot be assigned by the automated grading step. If that is desired, an additional manual grading step by an instructor may be added.

The last automated grading step is the style check of a statement. In general it is very difficult to assess the style of a SQL statement as there are no generally accepted rules on SQL style in contrast to e.g. Java. Thus we decided to use a set of rules which describe what a SQL statement should look like these have been taken from different sources. aSQLg is delivered with a small list of rules describing a general style. This list does not cover a complete style description, hence a complete list of rules has to be generated by the instructor. The number of points awarded during this step depends on how many style rules are followed by the student statement.

Now all checks for the statement have been executed. If there are statements left, the next statement is chosen to be analyzed by our grader. If the last statement has been checked the total points of the automated grading part are calculated. Additionally reports about the single grading steps are generated, which contain errors that may have occurred and other information including the number of points received by each statement in every step. The reports are shown to the student in the regular GUI of the CMS used. The number of available points for a statement in each step may be set in a configuration file. The four grading steps for syntactical correctness, cost of the statement, result correctness and style are evaluated. The weight of the different grading steps may as well be set in the configuration file. If maximum points for one of the grading steps is set to 0, the step is skipped during grading.

After the report generation and point calculation the statements can be optionally graded manually by an instructor. This step may be included directly into the CMS, e. g. by a special web form. The instructor has total access to the results of the automatic grading steps performed. The main task of the instructor normally is to check the correctness of the students statement, if the correctness could not be verified during the automated part. The instructor may award additional points to the student.

One point to consider is the danger of SQL-injection attacks on the database used to analyze and execute student statements. Malicious students could try to attack the database. In our concept this is not really an issue, because of two things: all students have read access to the database schema used for the queries anyway and no student query is ever directly executed on the system. They are just executed in the context as explained in listing 1. Thus in combination with correct setting of access rights to system tables by the database administrator students cannot perform an insider attack.

Some students may try to get full credit without learning and working on the statements. For example, if a task of an exercise demands a complex statement, which just returns one number as

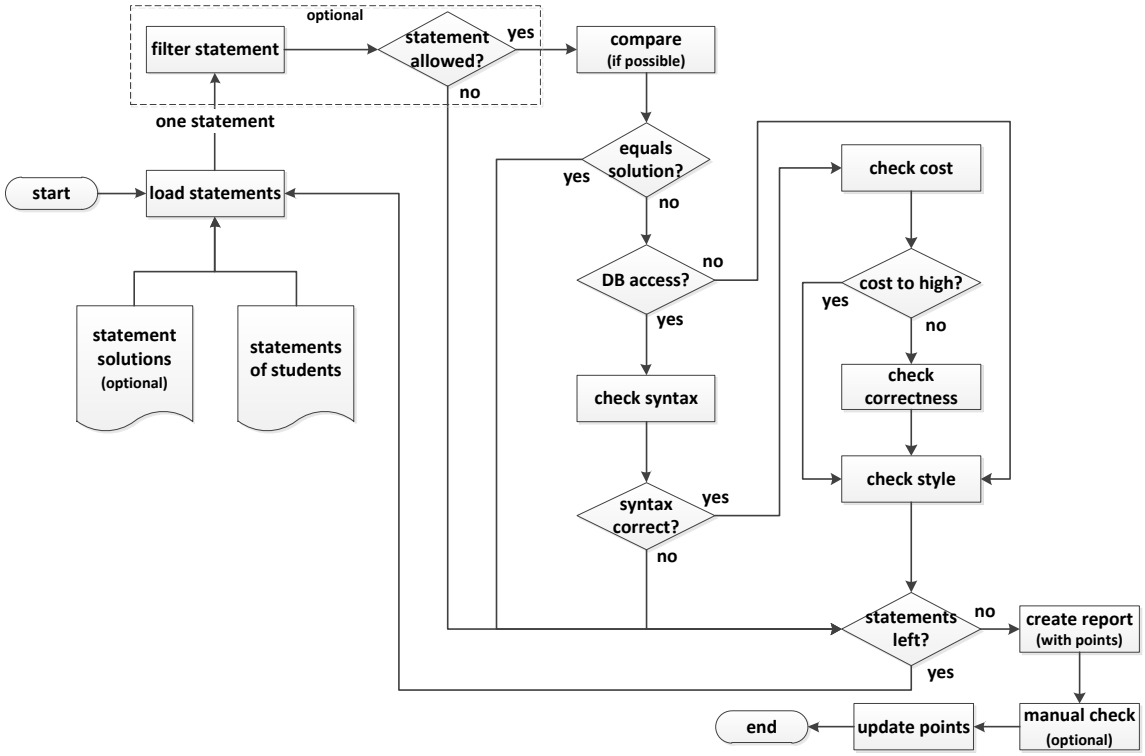

Figure 1: Flow diagram of the different steps in our SQL-statement Grader (based on [13])

result, a student could decide to cheat. When the statement result is 5 the student could use the statement "SELECT 5 FROM DUAL", which would return the correct result, without really working on the task. Countermeasures are needed to prevent this. That is why the filter is used at the beginning of the analyses process. Using the filter, usage of the table "DUAL" can be prevented. Alternatively execution of a statement on different datasets is possible which can also prevent such simple cheating attempts.

There is no semantic analysis of the provided solutions in comparison to the reference solution (i. e. comparison of query trees) so far. This would facilitate assigning partial credit for almost correct solutions. This feature is on the list of open issues for a future version of the system and would be an addition to the correctness check.

The described concept covers the whole grading process of SQL-statements. Nearly all checks are automated and statements containing forbidden elements can be identified and discarded. Only checking the results of partly correct statements has to be done manually. The syntax check and the filter need manually created lists to work properly, but the execution is done automatically. The only thing not mentioned so far is the identification and prevention of plagiarism, which figured out to be to complex to implement during the project.

4. SOME EXAMPLES

In this section, we present a real-life example showing how the grader was used during a first year database course. Part of this course was an introduction to SQL, which was accompanied by tutorials and about 40 SQL exercises of increasing difficulty.

The chosen example refers to an exercise from the beginning of the SQL tutorial. It uses the well-known Oracle example table `employees`. The task was to select every employee (first and last name as a single string) and the year of the hire date. The result had to be sorted by hire year and last name. A valid solution was:

```
SELECT first_name || ' ' || last_name name,
TO_NUMBER(TO_CHAR(hire_date, 'YYYY')) hired
FROM hr.employees
ORDER BY hired, last_name;
```

We now describe a simplified log of a student trying to solve the task. The first upload to aSQLg contained the following SQL statement:

```
Select First_Name||' '||Last_Name,
to_char(Hire_Date, YEAR)
from hr.employees
order by Hire_Date, Last_Name;
```

The syntax check of the grader failed, delivering a message to the student including the original database error: `ORA-00904: "YEAR": invalid identifier`. Due to the incorrect syntax, all further checks were skipped. In the following attempt, the student fixed the syntax error:

```
... to_char(Hire_Date, 'YYYY') ...
```

Now the syntax check succeeded, so the grader could proceeded with the cost check, which turned out to be uncritical. In the next step, the column count was compared with the correct solution, which also succeeded. However, the data type check failed, because column two returned a character value while a numeric value was expected. The following message was shown:
`Datatype of column 2 is wrong.`

```
Expected: NUMBER, your solution: VARCHAR2
```
The student went on with this solution:

```
... to_number(to_char(Hire_Date, 'YYYY')) ...
```

This change corrected the problem, so the grader now confirmed that the column count and data types were correct. The row count and row contents were also correct. The final error message was stating that the result rows were not sorted as expected, because the `order by` columns were still wrong (sorted by the whole `hire_date` instead of only the year). In the final submission, the student responded to this hint:

```
Select First_Name||' '||Last_Name,
to_number(to_char(hire_date,'YYYY')) AS datum
from hr.employees
order by datum,Last_Name;
```

Finally, the student was informed that the solution had been fully correct and that the full score had been recorded. Only warnings remained about the naming of the columns of the result, because of the differences to the proposed solution.

Summarizing, the example shows how the detailed messages generated by aSQLg can guide a student step-by-step towards the correct solution of a given exercise.

5. IMPLEMENTATION OVERVIEW

Our system consists of six modular components. Together all of this components implement the grading environment, facilitate statement grading, perform the grading steps and the generation of reports.

The goals of the implementation architecture are:

1. Modular structure
2. Extensibility of SQL dialects supported
3. Possibility of reuse for modules in other projects
4. Adaptable for different e-learning platforms with minimal programming effort

Figure 2 shows the components and their relationships. This structure makes it possible to easily adapt the statement grader to other environments than WebCAT by changing only a single component.

In the sequel we will briefly describe each of the components.

5.1 aSQLg Core

The aSQLg Core embeds the plug-in into the hosting environment system, e. g. WebCAT. This component configures the other components and implements the overall flow control for SQL grading.

The configuration of all other components is done in the order Reporter, StatementLoader, StatementViewBuilder, StatementFilter and StatementTester. While the first step sets up the reporting engine, the next two steps do the necessary groundwork to transform statements into an internal structure in order to be able to grade them. The StatementTester finally performs the actual grading steps with help of the CheckStyle component.

5.2 Reporter

The Reporter component collects information, that is generated by the other components, and creates report files. This component is used by most of the other components.

Information is generated for two receivers. The main receiver is the student who receives general information on the grading process, error notes and warnings for his statement as well as the awarded points. The instructor and assistants receive all this information on demand as well. In addition they have the option to add comments and points in the manual grading step. Finally instructor or system administrator receive detailed information about configuration errors and database failures.

There are two types of messages used to submit all of the above pieces of information. Plain messages have a simple text or XML content. Referenced messages also have a text or XML content but in addition contain information about the code position, to which the message refers. The latter makes it possible to show a code fragment within the results and mark the position where an error occurred - a helpful feature for grading assistants.

After the completion of the grading process the reporter generates an XML formatted report. This report can be converted by an XSLT stylesheet. By default the report is converted to XHTML, but the WebCAT plug-in package comes with a specialized transformation file to embed the reports into the WebCAT user interface.

5.3 StatementViewBuilder

For grading of SQL statements, in addition to the plain syntax, one needs a structured representation of the statement. For instance one of the tokens may belong to the blacklist of forbidden elements when used as a table name whereas it may be allowed as column name. Thus the StatementViewBuilder component builds a structured object view of a statement in form of a tree.

The information about the semantics of a token can be gained by parsing the statement. Our implementation uses a combination of JSqlParser [2] with our own simple extension. By implementing a different StatementViewBuilder interface this base is easily changeable. For each token in the statement a node in the StatementView tree is generated. This node contains the string representation of the token, as given in the input statement, a syntax type and a semantic type. The main syntax types occurring are keywords and parameters. The semantic types used vary from table, over column to numbers and texts.

5.4 StatementFilter

Giving students the possibility to submit any statement for grading by a central server, may incur a security risk as well. Even if not intended a statement might be harmful for the system. E.g. in the case where the instructor requests the statement `DROP Student-Table` a student could send the statement `DROP Instructor-Table`. Our StatementFilter component analyzes statements submitted in order to prevent such insider attacks. Denial of service attacks by inefficient statements are handled by the StatementTester later on.

The base for the StatementFilter is the previously generated StatementView. The filter can be configured by either using a whitelist of allowed or a blacklist of forbidden elements. Such elements will be detected by the StatementView. If one of it exists, the statement will be marked illegal, is never executed and a message to the student is generated.

5.5 StatementTester

The StatementTester is the central component of the grading process. It has two main functions, statement loading and execution of the statement.

The loading step includes reading SQL statements (as strings), using the StatementViewBuilder to form a StatementView and finally point allocation. All statements are loaded from files sub-

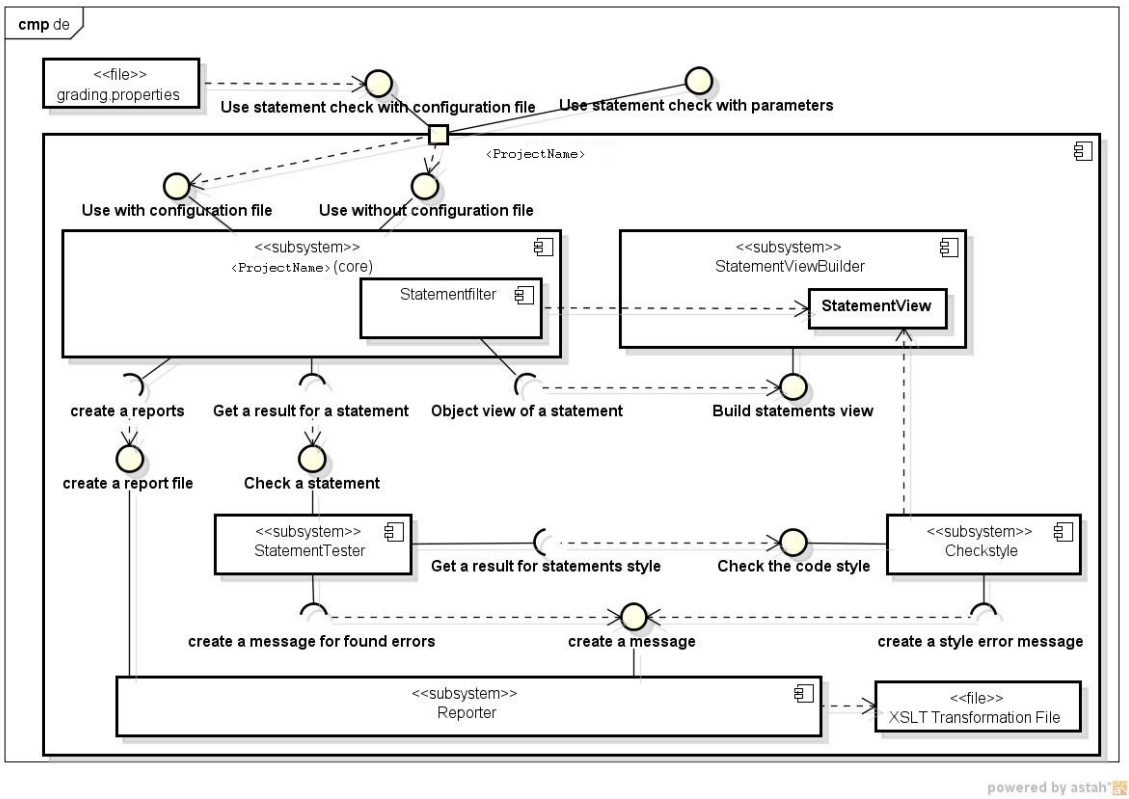

Figure 2: Overview of the components for implementation of aSQLg

mitted by students. Statements can have a reference statement, a solution given by the instructor. Student statements are compared against the instructors statement.

Allocation of points may consist of the following components: syntax check, query cost check, result check, style check and manual grading. For each of these aspects a maximum number of points may be defined by the instructor. Apart from the correctness components (syntax and result) where only full or no credit is possible for an individual statement, partial credit is possible.

After all the groundwork the actual grading is performed for legal statements. As the grading depends on the particular database system we use a "default" StatementTester class which may be extended to cater for specific systems. We use an OracleStatementTester extension in our classes. The default implementation may also be used as a fall back. It provides checks that can be used if there is no specific SQL dialect to be used.

The default StatementTester implements the general flow of the check as explained in section 3. The result check for SELECT statements is performed by comparing the results of the students statement and the instructors statement. If they are identical, the step is passed and full credit awarded. The syntax check in the generic tester is done by parsing the statement with the zql parser [3]. If the statement can be parsed, the syntax is correct, otherwise the parser throws an exception with the information of the line and column where a syntax error exists. This information is added to the reporter data to be provided to the student later on. The last step is the style check. It uses the CheckStyle component.

The OracleStatementTester uses the possibilities of the Oracle database and checks the statement for the Oracle specific SQL dialect. Here the syntax check is performed by using functions for calculating query costs. These functions return an error, if the statement contains a syntax error. The query cost check uses the result from the previous syntax check, because it is computed there anyway. The result and style check are performed using the implementations of the default StatementTester.

5.6 CheckStyle

Base for the CheckStyle component is the StatementView with its syntactic and semantic information. The instructor may define a style rule file. Each rule defines to which syntax and/or semantic type it is applicable. Furthermore rules can be defined for special keywords. As there is no commonly accepted *good* style for SQL queries so far, this part is completely configurable by the instructor. A foundation observing some of the most commonly rules for *good* style is provided as a reference.

6. EVALUATION

As mentioned earlier aSQLg has been used in our introductory database system classes in two different departments in 2012 and 2013. SQL statements have not only been assigned in aSQLg but classical paper assignments have also been used. On one hand that reduced the risk associated with employing a completely new and unproven tool. On the other hand that put the students in the position to better compare the two approaches and make informed judgments about the usefulness of automated SQL grading. In addition to the regular student course evaluation an online survey specifically targeted at the evaluation of the usage of aSQLg has been administered. In total 114 students (about 80 % of students en-

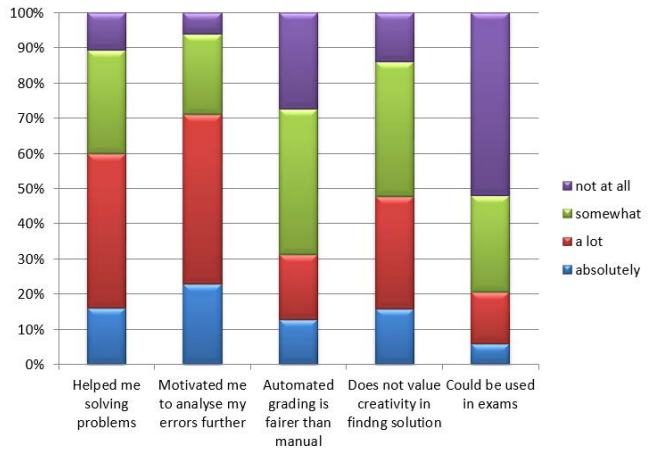

Figure 3: Exemplary results from student survey

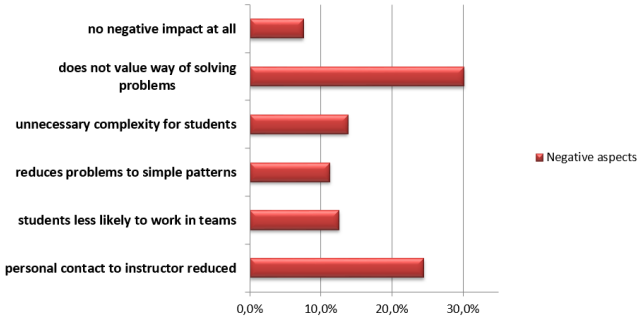

Figure 4: Most significant negative aspect of automated grading (percentage of students agreeing)

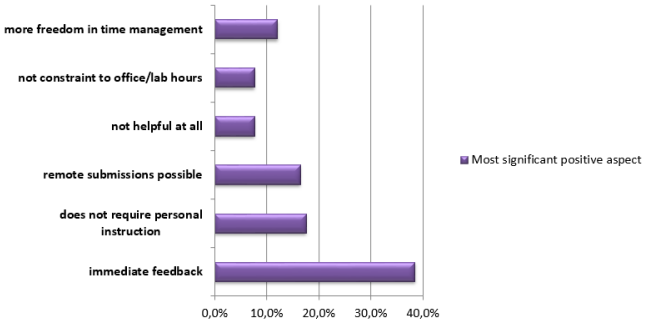

Figure 5: Most significant positive aspect of automated grading (percentage of students agreeing)

rolled in the class) submitted answers to this specific survey which consisted of 16 questions.

Some interesting results from the survey are displayed in figure 3. They show that about 60% of the students felt supported a lot or more by aSQLg in finding a solution (and only just over 10% did not feel supported at all). This supports the claim that there is potential for being used as a tutoring system as students believe in the tool helping them find solutions. Consequently almost 40% reported that the immediate feedback was the most helpful feature of the tool.

Motivation by students to dig deeper into problems when they are encountered is increasingly difficult to achieve; this in our experience is particularly true for SQL and database classes in general. In the case of aSQLg 70% of the students were motivated a lot or more by the tool to further analyze their errors in more detail. This is an extremely good and promising result. Maybe that has also led to the better student judgment of the whole course when compared to previous years.

Still most students (about 70%) expected manual grading to be fairer than automated grading, probably because they (just over 50%) did not like that just the result of the submission counts as opposed to the way of determining it which might be taken into account during manual grading. In a future version of aSQLg we would like to (at least partially) replace the result-oriented assessment with a semantic assessment. That kind of assessment would be able to compare the query tree of the ideal solution structurally against the submitted solution and thus value the way of solving the problem better even if there is a minor error which may render the result completely wrong. This would also remedy the most significant negative aspect of the system (30%, cf. Fig. 4).

Just over 10% of the students feared that students would tend to work alone instead of in teams as a consequence of using aSQLg and just over 15% stated that an advantage of the tool would be reduced need for personal instruction (cf. figure 5). Also about 25% said that reduced personal contact to the instructor might be the most significant drawback. These results show that as far as tutoring is concerned automated grading can be a good addition but will and should in general not replace personal lab sessions. Again students feared most that grading would be reduced to a comparison of results instead of valuing the way that has been used to find a solution, see above.

On the positive side from a student's perspective the immediate feedback to their solutions has been the most important significant aspect (almost 40%). Also aspects relating to distance education and electronic self-guided learning such as no need for personal instruction (18%), remote submission feature (17%) and more freedom in time management (12%) have been named often by the students. This supports the claim about the positive impact of aSQLg for electronic tutoring and distance education.

Finally, even in its first two instances the tool worked pretty good already with very few false negatives (would be very frustrating for students) and almost no false positives. Nevertheless students would not like to see it being used in exams (just over 50% said *not at all suitable*). That is probably because as long as there is at least a minimal chance of a false negative assessment students fear to be penalized in their grades for an erroneous software. Almost 60% of the students would like to see aSQLg be used in advanced database classes again and about 40% would like to use automated grading in other classes, too.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented a concept for automated assessment of SQL statements. The proposed algorithm uses the following aspects of the statement submitted in order to determine the score: syntactical correctness, efficiency of statement, correctness of results and style. Apart from these automatically assessed dimensions where points for each part are configurable it is also possible to assign points in a manual grading step as well. Finally in order to be able to practically use the software and prevent harm from the system by students submitting malicious solutions we have added the following features. There is the possibility to use black- or white-listing for allowed parts of the statement. Moreover statements beyond certain configurable efficiency bounds are

not executed in order to prevent denial of service attacks.

We have already used the system with individual students in a lab environment as well as in introductory database classes in two different departments at our school in 2012 and 2013. In efforts to get personal feedback and be able to make the system as student-friendly as possible, valuable input by the students has already been used in improvement steps of the system. We also received several of the ideas on how to attack the system in these lab sessions.

The system may both be used as an electronic tutor as well as for efficient course management. In the first case it guides the students to better results by them trying to increase the automatically assigned score. It also simplifies to take part in lab sessions for students who cannot or do not want to attend sessions on campus. For the second case it should be used in conjunction with a CMS. To achieve fair scores for the students in this setting the number of possible submissions should be restricted. Still manual grading (at least partially or to double-check the automated results) is possible within this framework if desired by the instructor.

The gain in efficiency by using the system as compared to pure manual assessment is obvious. This is because a completely manual grading process is still possible and just using the correctness check for instance is already an improvement over manual checking. The improvement in student learning by being able to submit result candidates multiple times and improve on the solution by comparing automatically computed scores which are shown as immediate responses has been the second goal of the system. This goal seems achieved partly already when looking at the feedback in figure 3. Also the student responses improved from 2012 to 2013 showing that the improvements in the system have been valued. Nevertheless the system is not yet perfect as some false negatives show, we will further have to work on that. Similarly assignment of points to different grading steps for different problems and at different points in a course needs to be further elaborated.

To further improve on the tutoring capabilities it would be very interesting to integrate our assessment system with one of the tutoring systems described in section 2. Thus one would obtain an even more valuable SQL learning tool for student self-study. The tutoring capabilities in addition to the score related feedback would be a perfect combination for self learning. Thus distance learning would be even more simplified as students can submit the solutions from any place and use the responses to improve their results an their own.

In addition in the near future we will plug the software into the new university-wide CMS lonCAPA. As explained above architecture and design simplify moving to such different hosting systems. Also we plan to publish the code once the system has reached production maturity for use at other institutions and also to get more feedback from a wider audience.

Finally the correctness check could be significantly improved by performing a semantic comparison of student and reference solution. An extended statement parser is needed for this check; this will be part of a future version of aSQLg .

We would like to thank Andreas Stöcker and Florian Fehring for supporting implementation and student evaluation for aSQLg .

8. REFERENCES

[1] GNU SQL tutor - website. http://www.gnu.org/software/sqltutor/. (visited on 16.09.2013).

[2] JSqlParser - website. http://jsqlparser.sourceforge.net/. (visited on 23.07.2011).

[3] ZQLParser - website. http://sourceforge.net/projects/zql/. (visited on 18.08.2011).

[4] A. Abelló, E. Rodríguez, T. Urpí, X. B. Illa, M. J. Casany, C. Martín, and C. Quer. LEARN-SQL: Automatic Assessment of SQL Based on IMS QTI Specification. In *ICALT*, pages 592–593. IEEE, 2008.

[5] R. Agarwal, S. H. Edwards, and M. A. Pérez-Quiñones. Designing an adaptive learning module to teach software testing. *SIGCSE Bull.*, 38:259–263, March 2006.

[6] P. Brusilovsky, S. Sosnovsky, M. V. Yudelson, D. H. Lee, V. Zadorozhny, and X. Zhou. Learning SQL Programming with Interactive Tools: From Integration to Personalization. *Trans. Comput. Educ.*, 9(4):19:1–19:15, Jan. 2010.

[7] A. Cepek and J. Pytel. SQLtutor. In *Professional Education 2009 – FIG International Workshop Vienna*. FIG Fédération Internationale de Géomètres, 2009.

[8] S. Dekeyser, M. de Raadt, and T. Y. Lee. Computer Assisted Assessment of SQL Query Skills. In J. Bailey and A. Fekete, editors, *ADC*, volume 63 of *CRPIT*, pages 53–62. Australian Computer Society, 2007.

[9] R. Dollinger. SQL Lightweight Tutoring Module – Semantic Analysis of SQL Queries based on XML Representation and LINQ. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2010*, pages 3323–3328, Toronto, Canada, June 2010. AACE.

[10] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *ACM Journal of Educational Resources in Computing*, 5(3), 2005.

[11] X. Fu, B. Peltsverger, K. Qian, L. Tao, and J. Liu. Apogee: automated project grading and instant feedback system for web based computing. In J. D. Dougherty, S. H. Rodger, S. Fitzgerald, and M. Guzdial, editors, *SIGCSE*, pages 77–81. ACM, 2008.

[12] C. Kenny and C. Pahl. Automated tutoring for a database skills training environment. *SIGCSE Bull.*, 37:58–62, February 2005.

[13] C. Kleiner. A concept for automated grading of exercises in introductory database system courses. In *Proceedings of the 7th International Workshop on Teaching, Learning and Assessment of Databases (TLAD)*, 2009.

[14] L. Malmi, V. Karavirta, A. Korhonen, and J. Nikander. Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. *ACM Journal of Educational Resources in Computing*, 5(3), 2005.

[15] A. Mitrovic. An intelligent sql tutor on the web. *I. J. Artificial Intelligence in Education*, 13(2-4):173–197, 2003.

[16] R. E. Noonan. The back end of a grading system. In D. Baldwin, P. T. Tymann, S. M. Haller, and I. Russell, editors, *SIGCSE*, pages 56–60. ACM, 2006.

[17] J. C. Prior. Online assessment of sql query formulation skills. In T. Greening and R. Lister, editors, *ACE*, volume 20 of *CRPIT*, pages 247–256. Australian Computer Society, 2003.

[18] J. C. Prior and R. Lister. The backwash effect on sql skills grading. In R. D. Boyle, M. Clark, and A. N. Kumar, editors, *ITiCSE*, pages 32–36. ACM, 2004.

[19] G. Rößling, M. Joy, and et al. Enhancing learning management systems to better support computer science education. *SIGCSE Bulletin*, 40(4):142–166, 2008.

[20] S. Sadiq, M. Orlowska, W. Sadiq, and J. Lin. SQLator: an online SQL learning workbench. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '04, pages 223–227, New York, NY, USA, 2004. ACM.