Enhancing Feedback Generation for Autograded SQL Statements to Improve Student Learning

Carsten Kleiner
carsten.kleiner@hs-hannover.de
University of Applied Sciences & Arts Hannover
Faculty IV - Department of Computer Science
Hannover, Germany

Felix Heine
felix.heine@hs-hannover.de
University of Applied Sciences & Arts Hannover
Faculty IV - Department of Computer Science
Hannover, Germany

ABSTRACT

Several tools to support autograding of student provided SQL statements have already been introduced. The full potential of such tools can only be leveraged, if they extend beyond grading efficiency by also providing tutoring capabilities to the students. With that, tools become really useful by offering self-paced and individually timed learning experiences. In this paper we present an extension for an SQL autograder which improves the hints generated for students in cases where their solution is not entirely correct. Our approach is to compare the student's solution with the model solution structurally to identify differences between the syntax trees describing the statements. This complements comparing the student's query with a model solution based on query results. In addition to improving the quality of hints generated for the students, this concept can also be used easily for data manipulation language (DML) or data definition language (DDL) statements, thus extending the applicability of the autograder. Along with details about the concept we present some example hints generated to illustrate the usefulness of the approach. We also report anecdotally on experiences with the system in two different level database courses. Results from different instances of one of them show improvements of student learning as well as student involvement by using the newly generated hints.

CCS CONCEPTS

• **Applied computing → Interactive learning environments**; **Computer-managed instruction**.

KEYWORDS

autograding, SQL statements, database class, tutoring, hint generation, self-contained learning

ACM Reference Format:
Carsten Kleiner and Felix Heine. 2024. Enhancing Feedback Generation for Autograded SQL Statements to Improve Student Learning. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653579

ITiCSE 2024, July 8–10, 2024, Milan, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0600-4/24/07
https://doi.org/10.1145/3649217.3653579

1 INTRODUCTION

The concept of automated grading of SQL statements submitted by students as exercise solutions has been introduced in [13] for the first time. Several implementations have been completed and described in the literature since then, see Sec. 2 for a more detailed discussion. Almost all of those fully automated systems, including the one used at the authors' institution ([12]), focus on comparing the results of the student submitted query with results of an instructor-provided sample solution. In addition, most of these systems (not the one in [12] though) use fixed and rather small database schemata and content, so that the reliability of a result-based grading is further reduced. Does the student query really solve the problem or is the result correct just by accident? Also, this cannot reasonably be applied to queries with empty or simple results (consider e. g. an aggregation with a single number as aggregation result). For such problems and also for problems involving joins, listing sample missing or gratuitous result rows is not a feasible approach. Moreover the concept of result based grading is directly only applicable to querying data (for DML or DDL statements defining and comparing results requires additional concepts) and, most importantly, the possibilities of supporting students in improving their solutions are limited.

The grading system at the authors' institution has thus been extended by an additional grading step which performs a structural comparison of the query tree provided by the student with the model solution tree supplied by the instructor. The idea is that differences in the tree are very likely to correspond to differences in the result and are thus potential errors in the student's solution. Naturally, such a comparison has to go beyond a mere syntactical comparison as a certain level of sensitivity to unharmful differences is needed. Consider for instance a different naming of aliases or using a different order of relations contributing to a join query. In such cases differences between student and instructor solution are not indicators of errors. On the other hand different or forgotten selection predicates are likely to be errors in the student query. Thus, an intelligent comparison algorithm is needed to generate helpful hints how students can improve their solution.

The algorithm used in our grading system computes a similarity score between student and instructor solution as additional result of the comparison. This score can be used to assign partial credit for the solution in case it is not entirely correct. Note, that partial credit is very difficult to assign in a fair manner when using result based grading only, as the number of rows differing from the correct solution is not a good quality measure, e. g. in cases where only a minor error might produce a completely different result.

The remainder of the paper is organized as follows: we give an overview of previously published tools in Sec. 2. After a brief overview of our own autograder in Sec. 3 we explain more details on the newly added structural comparison and hint generation in Sec. 4. Following are practical examples in Sec. 5 and a report on experiences and some quantitative evaluation in Sec. 6, before we conclude with some ideas for future improvements in Sec. 7.

2 RELATED WORK

Automated grading of SQL statements has been in the focus of instructors for over two decades now. Thus, several systems for automated grading of SQL statements have been reported in the literature. Among the older ones are e. g. [16] and [13]. More recently other systems have been proposed such as [9], [8] or [19] . Another one being cited rather often (probably, since it provides code on GitHub and thus can be easily used at one's own institution) is [11]. An overview of those and even more systems is part of [15]. This publication also reviewed and compared the systems in more detail. One of the major conclusions is that only SQLify ([5]) provides useful feedback to students in terms of improving the solution by tutoring. However, that feedback is primarily provided by peer reviews and not in an automated way. Interestingly, even the systems reported in the literature with a focus on the tutoring aspect (rather than the grading efficiency) did not solve the hint generation issue. Particularly the older ones had this focus and [16] introduced interesting first steps towards valuable hints, but those have not been realized yet. A practical evaluation in [14] has shown that high level feedback is more valuable for students than detail feedback. This aspect is addressed by our proposal by the option to generate more intelligent feedback from higher levels of the syntax tree as opposed to low level comments (which might be helpful for rather simple problems anyway). Consequently, improving the quality of the hints for students in order for them to autonomously improve their solutions is a crucial advancement for any SQL autograder. A more general systematic overview of SQL teaching approaches beyond autograders can be found in [18].

Ideas complementing our approach are presented in [1] where the focus is on disposing of syntax errors. Since we assume these are addressed by error messages from the database system itself, such concepts are not found in our approach. Similarly, [6] presents an interesting concept to support students in improving their individual approach to problem solving. Also, ideas from hint generation in general programming tutoring could be used, see [4].

Apart from improving hint generation fairer partial grading is also facilitated by our approach. Most other systems focus on correctness testing and grading such as [17]. In addition, [11] uses the grades as guideline for the students trying to support learning mostly by improving correctness grading results. Grading-wise the most similar system to ours is [19]. They also use different grading aspects, which in addition can be flexibly combined as in our approach. But they provide only a limited structural analysis of the student solution. Another approach with good partial credit assignment, but very limited hint generation can be found in [9]. Also, each SQL construct has to be added separately with its own concept there. [2] also presents the idea of similarity based grading of solutions, however they do not provide hint generation.

Most similar to our approach are [10], which is based on a conceptually sound concept with different grading aspects, however more basic in terms of hints, and [7], following a similar idea, however, queries are transformed into XML to detect differences. We see advantages in directly using the parsing tree of the SQL expression as in our approach instead of transforming it into XML first. Also no evaluation results are presented there.

3 BRIEF OVERVIEW OF THE AUTOGRADER

The autograder used at our institution, before adding the features introduced in this paper. is explained in [12]. Basically, it uses a multiple step approach for grading where each grading step can be individually activated or deactivated per problem. Traditional grading steps (along with typical level in Bloom's taxonomy) are:

- Syntactical correctness (remember)
- Efficiency of execution (apply)
- Result correctness (apply/analyze)
- Style check (remember)

Execution of grading steps per submission can be made dependent on successful completion of previous grading steps (e. g. only syntactically correct statements can be executed for result correctness). In addition, each grading step can be assigned a certain weight for the final grade which may also be 0 in cases where no credit shall be awarded for certain steps (e. g. syntactical correctness in advanced classes) despite them being checked.

The result correctness step at first compares the metadata of the student and instructor query such as names and datatypes of the result columns as well as number of rows in the results. On one hand this is useful, as a comparison of query results only makes sense, if both results consist of the same columns using the same datatypes. On the other hand this facilitates at least some helpful hints for the students in cases where the result is not correct. In case metadata matches result correctness is checked by computing the union of the difference sets between student and sample solution inside the database. If this set is empty, the results are identical. This approach is very efficient since the outcome can be computed inside the database system. However, only binary grading (correct/incorrect) can be used for correctness. Other approaches such as intersection over union may often lead to inappropriate or unfair grades as small mistakes may lead to large differences in result sets whereas large mistakes may only result in small differences.

Finally, our autograder is integrated into the university's LMS (Moodle) which simplifies many LMS-typical tasks such as user management, submission handling and managing grades produced by the autograder in the context of other grades in the course.

4 CONCEPT FOR IMPROVED HINT GENERATION

Since the hints generated based on the result correctness step did not solve all problems students had in providing correct solutions, we looked for additional hints to be generated in order to address the analyze level of Bloom's taxonomy better. One problem of hints based on result correctness is that they only address metadata of the query. They do not consider internal aspects of the student's query. Thus, a wide range of issues cannot be addressed this way. These range from rather simple errors (such as malformed selection criteria) over medium difficulty (such as wrong combinations in join formulation) up to more complex issues such as unnecessary or malformed subqueries. Such issues can only be detected and addressed in hints presented to the student, if the structure of the student's statement is analyzed and compared to the structure of the sample solution.

As explained above, the basic idea of the improvement is to generate parse trees of both the student as well as the instructor query and then to look for structural differences in the two trees. These can be on lower levels of the trees for the more simple errors or on higher levels for the more complex issues. Technically, we decided to add the new functionality as another grading step in the grading process to retain maximum flexibility.

Algorithm 1 Tree Mapping and Hint Generation

```
Require: modelSolution, studentSolution syntactically correct
  modeltree ← generateParseTree(modelSolution);
  studenttree ← generateParseTree(studentSolution);
  if !(modeltree is identical to studenttree) then
      // Mapping
      loop Perform parallel preorder traversal of trees
          if similarity(modnode, studnode) ≥ $mapThreshold$ then
              map nodes
          end if
      end loop
      for all unmapped nodes in studenttree do
          $tsim \leftarrow 0$
          for all unmapped nodes in modeltree do
              $tsim$ ← max(similarity(modnode, studnode), $tsim$)
              if $tsim \geq mapThreshold$ then
                  map nodes
              end if
          end for
      end for
      $treeSimilarity \leftarrow avg(mappingFactor)$ over all nodes
      // Hint Generation
      for all mapped nodes in studenttree do
          mark node as done
      end for
      for all mapped nodes in modeltree do
          mark node as done
      end for
      // Analyze undone nodes (forming blocks for subtrees etc)
      loop all undone nodes in studenttree, modeltree
          form blocks for semantically coherent subtrees
      end loop
      Generate hints for undone node blocks in three categories:
      // Differing in studenttree from modeltree
      // Only in studenttree
      // Only in modeltree
  else
      $treeSimilarity \leftarrow 1$
  end if
```

An abstract pseudo-code description in Alg. 1 illustrates the procedure in more detail. Obviously, since we compare the parse trees of the statements, these must be syntactically correct in order to be able to compute a parse tree at all. This is typically ensured by previous execution of the syntactical correctness grading step.

In the first part of the algorithm a parallel preorder traversal of both trees is performed where a mapping of visited nodes based on a node similarity function is tried (if the similarity is above a pre-defined threshold the nodes are considered matching). Since typically the general structure of the parse trees tends to be similar at least on higher tree levels, this already identifies most of the similarities. Only nodes not assigned a mapping partner (and similarity score) in the student tree are then examined to find a suitable mapping to any of the remaining unmapped nodes in the model solution tree. The node with the highest similarity score which is above the threshold is used as mapping partner (in most cases there is at most a single node satisfying this condition anyway). After the mapping process is complete, based on the similarity scores of all nodes an overall similarity score of the trees is computed.

In the second part of the algorithm the hints for the students which will be further illustrated in Sec. 5 are generated. For this process only unmapped nodes from the previous phase are considered since the mapped nodes are interpreted as equivalent between the two queries. Note that it is not a viable option to simply output all unmapped nodes of the parse tree to the student. On one hand, by listing all nodes of the sample solution that have not been mapped to the student solution, the student, after supplying a trivial and incorrect answer, would essentially be given the whole sample solution. He could then assemble this into his own solution without gaining any actual knowledge, why this solution solves the given problem. On the other hand, for typical medium to advanced levels of difficulty in problems the number of unmapped nodes will be very large. This will only lead to more confusion on the part of the students as they need to navigate through a large list of nodes. For instance, for the problem in example 5.2 the student parse tree consists of 90 nodes, the instructor parse tree of 82 nodes and 22 of these nodes could not be mapped, even though the statements are not too different. Listing all the 22 unmapped nodes would definitely confuse a student rather than help improve his solution.

In addition, not all nodes in the parse tree correspond to syntactically recognizable parts of the query, so that it is very likely that students who are struggling to write correct SQL queries will be overwhelmed with the much higher level of abstraction required to interpret elements of a parse tree. Thus, there is a need to perform the abstraction within the grading system and only provide the students with a limited set of meaningful differences to the sample solution. A similar observation has also been reported in [14].

To address this, several optimizations are performed in an analysis phase such as forming semantically relevant subtrees of unmapped nodes in order to reduce the number of hints. In addition, nodes with syntactical relevance are identified, since hints always have to correspond to a specific part of the student statement, i. e. some syntactical element. Finally, hints are generated in three categories, namely components only present in either the student or the model solution and components present in both, but not equivalent. These will lead to hints of the categories `Unnecessary`, `Missing` and `MixedUp` (cf. Sec. 5), respectively.

An important part of the algorithm is the node similarity function computing a similarity score between two nodes. This function is a weighted sum of several parameters of the comparison of the nodes consisting of the following aspects:

- logical position of nodes within the SQL statement
- string representation of nodes as boolean value
- syntactical type of nodes
- semantical type of nodes
- similarity of path to the root of the tree
- depth in the tree
- column position of the nodes within the query string
- nodes' sizes

The weights for these similarity factors have been determined heuristically, resulting in decreasing weights in the order of the above description and weights ranging from 0.35 down to 0.02. It should be noted though that these have not yet been systematically optimized in order to get the best mapping results. However, in most cases the mapping has proven to be appropriate in practice.

Currently, in cases where students use the same tables at the same logical positions inside an SQL statement (e. g. within a join) but assign different aliases the similarity comparison will detect that both solutions are actually semantically equivalent by properly mapping the table nodes against each other inside of the also properly mapped join nodes. This is done by managing a list of aliases and their corresponding fully qualified efficient table names and then mapping based on these names, so that different aliases do not prevent the semantically correct mapping.

5 EXAMPLES

We will now present two different examples of questions and corresponding incorrect student solutions including a rather simple as well as a more complex query. Both are from actual instances of our database systems 1 or 2 classes and the solutions are actual solutions students submitted. We will use these to illustrate which hints can be generated by the tree comparison and how they are more useful for the student than the hints generated by the previously available result correctness grading step. For all examples we will use the well-known Oracle HR example database schema, but since the grader is schema agnostic it would work accordingly for other schemata as well.

Example 5.1. Missing selection predicate

The task is to select names and salaries of all employees that have no manager or are not assigned to any department. The model solution is as follows:

```sql
SELECT first_name , last_name , salary
FROM hr.employees
WHERE department_id IS NULL OR manager_id IS NULL;
```

A student missing the cases where there is no department assigned submitted the following solution:

```sql
SELECT first_name,last_name, salary
FROM hr.employees
WHERE manager_ID IS NULL;
```

When using the result correctness grader it will determine that the number of rows returned by the student solution is wrong (and thus obviously also the result content), while all other metadata such as column names and datatypes are correct. Still it will assign 0 points for correctness with the message:

```
ERROR Your result has an incorrect number of rows.
Correct solution: 2 rows. Your solution: 1 rows.
```

This does not really help the student in determining the error, he just learns that he needs a different number of result rows, but not why some results are excluded in his solution. The output of the new hint generation, however, is more helpful:

```
WARNING Hint: Missing: [department_id]
WARNING Hint: Missing: Symbol[OR]
WARNING Hint: Missing: [IS NULL]
```

These hints inform the student about components missing in his solution without actually providing the solution[1]. He learns about missing components in his solution, yet still has to arrange them in the correct order. In addition, a similarity score of the trees is computed at 0.902 which might be used to assign partial credit. Probably, there is the need for some transformation of the similarity score into a partial credit grade as 90% of the correctness points might be too much for this simple query.

Example 5.2. Incorrect join

Here the task is to select names and countries (only ID) where employees are working. Note that countries are retrieved from a relation `location` which is referenced from `employees` only via the department relation in which an employee works. Since employees w/o any department also need to be selected, we require an outer join operation. The sample solution is:

```sql
SELECT e.first_name || '␣' || e.last_name AS name,
    l.country_id AS country
FROM hr.employees e
    LEFT JOIN hr.departments d
        ON (e.department_id = d.department_id)
    LEFT JOIN hr.locations l
        ON (d.location_id = l.location_id)
```

A student solution submitted did not leverage the functionality of the outer join and instead tried to fix this by a complex function in the projection part as follows:

```sql
SELECT e.first_name || '␣' || e.last_name AS name,
    CASE WHEN l.country_id = '' THEN 'NULL'
    ELSE l.country_id END AS country
FROM hr.departments d
    JOIN hr.locations l
        ON l.location_id = d.location_id
    JOIN hr.employees e
        ON e.department_id = d.department_id;
```

The traditional result correctness check delivers a similar message as the one in Ex. 5.1 (in addition correct metadata of the result is reported):

```
ERROR Your result has an incorrect number of rows.
Correct solution: 107 rows. Your solution: 106 rows.
```

The correctness score is 0. Note that in this case, even though 106 of the 107 result rows have been retrieved correctly, there are still serious errors and misconceptions in the student solution, so that a correctness score based on the proportion of correctly selected rows, such as $\frac{106}{107} = 0.9907$ would be much too generous here.

[1]Even though, in this rather simple example, there is only a small step from the hints to the correct solution; yet, actually there is not much more complexity in such a simple exercise at all anyway.

The first issue again is that the student did not understand how an outer join can be used in this case. The second issue is that, apart from the stylistically problematic try to fix this by a CASE statement in the projection, the student assigned the string NULL instead of the semantically different value of NULL. All these aspects are reflected in the hints generated by the new grading step as follows:

```
WARNING Hint: Redundant: Column[l.country_id]
WARNING Hint: Redundant: [CASE]
WARNING Hint: Redundant: [WHEN]
WARNING Hint: Redundant: ['NULL',NULL,'']
WARNING Hint: Redundant: [THEN]
WARNING Hint: Redundant: [ELSE]
WARNING Hint: Redundant: [END]
WARNING Hint: Missing: Join[LEFT]
```

While the issue with the unnecessary CASE part leads to many hints which should be grouped in a future version of the grader, there are also hints pointing to the NULL issue as well as the missing outer join. Note that the different order of join components used by the student did not confuse the grader and did not yield any unnecessary messages. This has been prevented by the node mapper being agnostic of the order of subnodes in join situations. Also, the hints do not directly solve the problem for the student, but should point him to checking what an outer join is (last hint message) which he missed. This should ultimately make him remedy the problems himself. In this case the tree comparison computed a similarity score of 0.849 which might also be pretty high to assign partial credit in this case. Yet, it is significantly smaller than the score in Ex. 5.1 which seems appropriate and also much better than the potential 0.9907 from the correctness grading.

6 EXPERIENCES AND EVALUATION

6.1 Experiences

We have used the autograder with the additional hint generation step in our introductory database class in spring 2021 and 2022 as well as in the advanced database system class in winter 2021. The enrollment is about 100 in the introductory class and about 50 in the advanced class. Students were able to achieve a bonus for the exam by succesfully completing the autograded exercises (and some others), so about 70% completed the autograded exercises.

In general, the autograding was well received by the students as documented by frequent usage and they especially liked the option to complete the exercises at any time and still have support in case they had difficulties. In spring 2021 the whole semester was an online semester, so the autograding, esp. the improved hints, significantly simplified and almost only facilitated providing practical support as in the online sessions individual support for students by the instructor was very hard to provide. Without the additional hints generated by the tree comparison (as in earlier years) the students would not have been able to complete most of the exercises as they typically do not advance much based on the correctness grading and thus relied on personal support by the instructor. The improved hints helped some students complete the exercises on their own. Still, there were several students which did not look at the hints at all or did not try to leverage the information in order to improve their solution. This phenomenon is also known in programming classes where students tend to ignore error messages generated by the compiler. Some students when instructed to read the hints were actually able to improve their solution accordingly.

A well appreciated improvement over previous years was the possibility to assign partial credit for partially correct solutions. Some students were happy with almost correct solutions and others, which at least tried to solve the exercises, were better motivated by the partial credit. Ultimately, the partial credit is still very generous if based directly on the similarity score (cf. Sec. 4) which might have contributed to this phenomenon.

An important aspect is the flexibility of the tree comparison which does not generate hints for any difference between the trees, just for those that might be relevant to improve the solution (cf. the join order example 5.2). Similarly, the grouping of differences in the analysis phase has reduced the number of hints and also lifted the hints to a higher level than a mere syntactical comparison. This part is still not perfect as not all semantically equivalent constructs are yet identified (consider, e. g. $a >= 10$ comparsion versus $a > 9$ comparison on integers). This is also an area for future work.

As mentioned before the translation of similarity score to partial credit needs to be fine tuned in the future. This is particularly important for simple problems where a mere SELECT ... FROM ... WHERE with some working content will already lead to a pretty high similarity score. This is not a technical issue as this can be achieved by our Moodle integration. However, it might be more challenging to define this score properly from a semantic point of view for all cases during grading.

An inherent drawback of the tree comparison approach is that the model solution plays a central role. If there are different ways to solve a problem and the student's try is almost correct but using a different solution concept, the hints will always try to lure the student to the concept used in the model solution. A classical example for this issue are non-existence queries between tables, e. g. find all department names where no employee works in this department. This can be done by an outer join followed by a NULL check or alternatively by a subquery inside an EXISTS. Both solutions are equally good, yet the system will always try to move the student towards the concept used in the model solution instead of trying to improve his specific solution concept.

6.2 Quantitative Evaluation

In this section we will present some sample quantitative evaluations of the autograder and its new hint feature. For this we use data from three instances of the introduction to database systems course, namely spring semesters of 2020, 2021, and 2022. Threats to validity of the results are the small and thus potentially non-representative number of courses the evaluation is based on as well as that in each instance the autograder has been used in a different configuration:

- 2020: hint feature not used at all
- 2021: hint feature used, but only displayed inconspicuously and not used in grading
- 2022: hint feature used for grading and peculiarly displayed, individual problem variant per student introduced

In Fig. 1 we give an overview of the average maximum result scores[2] per student achieved for each of the autograded problems

[2]Note that even in the presence of tree similarity scores from 2021 onwards, for comparison purposes we always use the mere result score in this section.

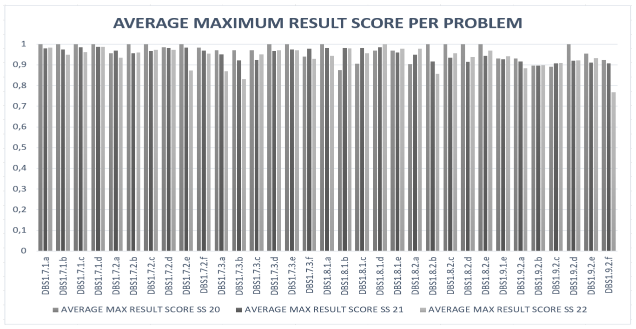

Figure 1: Average maximum result scores per student for different problems and instances

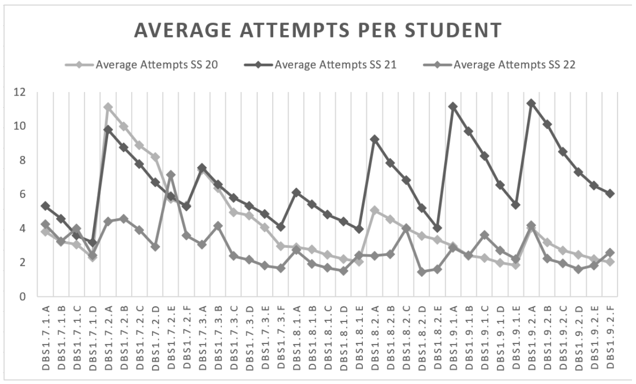

Figure 2: Average number of attempts per student for different problems and instances

in different semesters; this is also the final score for each student as the maximum has been counted ultimately. The problem keys are shown on the x-axis. The overall high scores show the effectiveness of the autograder in general, since most students achieved pretty high result scores ultimately. There is no large difference between different years which is a remarkable result with regard to 2022 when students received individual problem variants, so that mere result copying from peers would result in a result score of 0.

While it is difficult to illustrate the individual improvements per student based on the provided hints in a summative fashion, Fig. 2 tries to do this to a certain degree by displaying the average number of tries that the a student needed to achieve the maximum result score. The benefit of the new hint generation shows in a reduced number of attempts for each problem per student in 2022 compared to previous years. This effect is even more impressive, since the number of submitters is still high and the novel variant feature being in place preventing result copying. The helpfulness of the hints is also underlined by the fact that the average result score per attempt is lowest in 2022 (graph omitted due to space constraints). This means the highest improvement in result score has been achieved with the lowest number of attempts in 2022.

Regarding the preliminary tree similarity scores assigned in 2022 in addition to the 0/1-result scores, we observed an average similarity score per attempt of 72.1% compared to an average result score of 50.5%. This seems to be a motivating aspect for students. The average maximum similarity score of 79.2%, however, illustrates a need for a better assessment in the future as it is well below the 93.1% average maximum result score. This has not been perceived a problem by students so far, though, as correct results always received full credit for the entire problem.

Thus, we can conclude that students obviously individually improved their solutions significantly faster with fewer attempts. This combination of effects shows the usefulness of the hint generation feature and the partial score assignment based on similarity.

7 CONCLUSION AND FUTURE WORK

As explained above the extension of our autograder by an additional grading step comparing the parse trees of the student solution and the model solution made it possible to generate much more precise and helpful hints for the students in cases where their solution is not entirely correct. It is important to note again that this is considered just one additional step in the grading process (cf. Sec. 3) on top of e. g. result correctness based grading or style checking. This additional option facilitates advanced self improvement options of the student solution by following the hints. This is helpful in distance learning situations and/or in situations where students work on problems outside of regular class hours. Similarly, this approach lowers the tutoring effort on the instructor side, especially in cases with rather simple to fix issues which can be detected by the comparison algorithm. Another benefit is the possibility to assign fairer partial credit for partially correct solutions as possible by a mere result based comparison. Recall that the number of correctly returned rows might not be a good measure for solution quality.

While the system has already proven useful, there is still a lot of improvement possible and planned based on a more thorough quantitative evaluation. The first item is improving the translation of tree similarity into partial credit, so that it better reflects a grade that would be given by a human grader; currently the system is rather generous to the students. To remedy the issue that a student is always hinted towards the single model solution even if he tries to use a different, but correct solution concept, we plan to add the option of providing multiple model solutions and only generate hints towards the one that is closest to the student's try. This is similar to an idea in [3].

Another planned improvement is the node analysis for hint generation. We want to be able to identify further situations in which subtrees are equivalent and thus do not need to be hinted at all. Also, we would like to assemble unmapped nodes in even larger subtrees in order to make the hints even more helpful, e. g. generate a `missing subquery` hint instead of a `missing select`. The idea of grouping student answers in clusters as described in [20] could be used in the future.

Finally, we could also improve the node mapping algorithm by finding better weighting parameters in the computation of node similarity. It might even be an option to use machine learning algorithms on the rather large existing example set of trees assembled in the last semesters to learn these parameters better than in the currently heuristic definition. However, we are pretty confident that the set of influential parameters should already be comprehensive and that just the weights have to be optimized.

REFERENCES

[1] Alireza Ahadi, Vahid Behbood, Arto Vihavainen, Julia Prior, and Raymond Lister. 2016. Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and Its Application to Predicting Students' Success. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 401–406. https://doi.org/10.1145/2839509.2844640

[2] Peter Brusilovsky, Sergey Sosnovsky, Michael V. Yudelson, Danielle H. Lee, Vladimir Zadorozhny, and Xin Zhou. 2010. Learning SQL Programming with Interactive Tools: From Integration to Personalization. *ACM Trans. Comput. Educ.* 9, 4, Article 19 (jan 2010), 15 pages. https://doi.org/10.1145/1656255.1656257

[3] Bikash Chandra, Ananyo Banerjee, Udbhas Hazra, Mathew Joseph, and S. Sudarshan. 2021. Edit Based Grading of SQL Queries. In *Proceedings of the 3rd ACM India Joint International Conference on Data Science & Management of Data* (Bangalore, India) *(CODS-COMAD '21)*. Association for Computing Machinery, New York, NY, USA, 56–64. https://doi.org/10.1145/3430984.3431012

[4] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent Tutoring Systems for Programming Education: A Systematic Review. In *Proceedings of the 20th Australasian Computing Education Conference* (Brisbane, Queensland, Australia) *(ACE '18)*. Association for Computing Machinery, New York, NY, USA, 53–62. https://doi.org/10.1145/3160489.3160492

[5] Michael de Raadt, Stijn Dekeyser, and Tien Yu Lee. 2006. Do Students SQLify? Improving Learning Outcomes with Peer Review and Enhanced Computer Assisted Assessment of Querying Skills. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006* (Uppsala, Sweden) *(Baltic Sea '06)*. Association for Computing Machinery, New York, NY, USA, 101–108. https://doi.org/10.1145/1315803.1315821

[6] Tadej Matek Dejan Lavbič and Aljaž Zrnec. 2017. Recommender system for learning SQL using hints. *Interactive Learning Environments* 25, 8 (2017), 1048–1064. https://doi.org/10.1080/10494820.2016.1244084

[7] Robert Dollinger and Nathaniel A. Melville. 2011. Semantic evaluation of SQL queries. In *2011 IEEE 7th International Conference on Intelligent Computer Communication and Processing*. 57–64. https://doi.org/10.1109/ICCP.2011.6047844

[8] Mario Fabijanic and Igor Mekterović. 2023. Partial SQL Query Assessment. *2023 46th MIPRO ICT and Electronics Convention (MIPRO)* (2023), 1317–1322. https://api.semanticscholar.org/CorpusID:259299956

[9] Mario Fabijanić, Goran Đambić, and Jan Sasunić. 2022. Automatic, configurable, and partial assessment of student SQL queries with subqueries. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. 542–547. https://doi.org/10.23919/MIPRO55190.2022.9803559

[10] Mohammad Karimzadeh and Hasan M Jamil. 2022. ViSQL: An Intelligent Online SQL Tutoring System. In *2022 International Conference on Advanced Learning Technologies (ICALT)*. 212–213. https://doi.org/10.1109/ICALT55010.2022.00069

[11] Anthony Kleerekoper and Andrew Schofield. 2018. SQL Tester: An Online SQL Assessment Tool and Its Impact. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018)*. Association for Computing Machinery, New York, NY, USA, 87–92. https://doi.org/10.1145/3197091.3197124

[12] Carsten Kleiner, Christopher Tebbe, and Felix Heine. 2013. Automated Grading and Tutoring of SQL Statements to Improve Student Learning. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '13)*. Association for Computing Machinery, New York, NY, USA, 161–168. https://doi.org/10.1145/2526968.2526986

[13] Antonija Mitrovic. 1998. Learning SQL with a Computerized Tutor. *SIGCSE Bull.* 30, 1 (mar 1998), 307–311. https://doi.org/10.1145/274790.274318

[14] A. Mitrovic and B. Martin. 2000. Evaluating the effectiveness of feedback in SQL-Tutor. In *Proceedings International Workshop on Advanced Learning Technologies. IWALT 2000. Advanced Learning Technology: Design and Development Issues*. 143–144. https://doi.org/10.1109/IWALT.2000.890591

[15] Sidhidatri Nayak, Reshu Agarwal, and Sunil Kumar Khatri. 2022. Review of Automated Assessment Tools for grading student SQL queries. In *2022 International Conference on Computer Communication and Informatics (ICCCI)*. 1–4. https://doi.org/10.1109/ICCCI54379.2022.9740799

[16] Julia Coleman Prior. 2003. Online Assessment of SQL Query Formulation Skills. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20* (Adelaide, Australia) *(ACE '03)*. Australian Computer Society, Inc., AUS, 247–256.

[17] Shazia Sadiq, Maria Orlowska, Wasim Sadiq, and Joe Lin. 2004. SQLator: An Online SQL Learning Workbench. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE '04)*. Association for Computing Machinery, New York, NY, USA, 223–227. https://doi.org/10.1145/1007996.1008055

[18] Toni Taipalus and Ville Seppänen. 2020. SQL Education: A Systematic Mapping Study and Future Research Agenda. *ACM Trans. Comput. Educ.* 20, 3, Article 20 (aug 2020), 33 pages. https://doi.org/10.1145/3398377

[19] Paul J. Wagner. 2020. The SQL File Evaluation (SQLFE) Tool: A Flexible and Extendible System for Evaluation of SQL Queries. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 1334. https://doi.org/10.1145/3328778.3372599

[20] Matthew Weston, Haorong Sun, Geoffrey L Herman, Hisham Benotman, and Abdussalam Alawini. 2021. Echelon: An AI Tool for Clustering Student-Written SQL Queries. In *2021 IEEE Frontiers in Education Conference (FIE)*. 1–8. https://doi.org/10.1109/FIE49875.2021.9637203