Automated Grading of SQL Queries

Bikash Chandra* S. Sudarshan
IIT Bombay
{bikash,sudarsha}@cse.iitb.ac.in

Abstract

In a traditional classroom setting, instructors and teaching assistants either grade SQL queries either manually or using fixed datasets. Manual grading is tedious, error-prone and does not scale well for courses with a large number of students or for online courses. Using fixed datasets may miss even common errors and mark incorrect student queries as correct. In this article, we discuss our system XData that, given a set of correct queries, can automatically evaluate the correctness of SQL queries using datasets designed to catch errors in the given correct queries. If a student query is found to be incorrect, XData can even assign grades to the query based on how close they are to a correct query. In our experience, these techniques can be used to grade queries for settings with a large number of students with little human effort and involvement.

1 Introduction

Complex SQL queries may be written in several different ways and are difficult for beginners to get right. In courses that teach SQL queries, student SQL queries are often graded manually. The manual grading involves reading the student query manually comparing it to a correct query and/or executing the query on fixed datasets. Manually reading the query and comparing queries may be difficult and is also prone to errors while using grading using fixed datasets may miss errors in student queries. Let us consider an example where the correct query Q is

```
SELECT course.id, department.dept_name
FROM course LEFT OUTER JOIN
  (SELECT * from department WHERE department.budget > 70000) d
USING (dept_name);
```

A common mistake made by students is to write the following query Q_s instead

```
SELECT course.id, department.dept_name
FROM course LEFT OUTER JOIN department
USING (dept_name) WHERE department.budget > 70000;
```

The student query looks sufficiently similar for a grader to miss the difference. The queries, however, are not equivalent since they give different results on departments with a budget less than 70000. The query Q would output such courses with a NULL department name while Q_s would not output such courses. Even using a fixed

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Currently at Meta Platforms Inc.

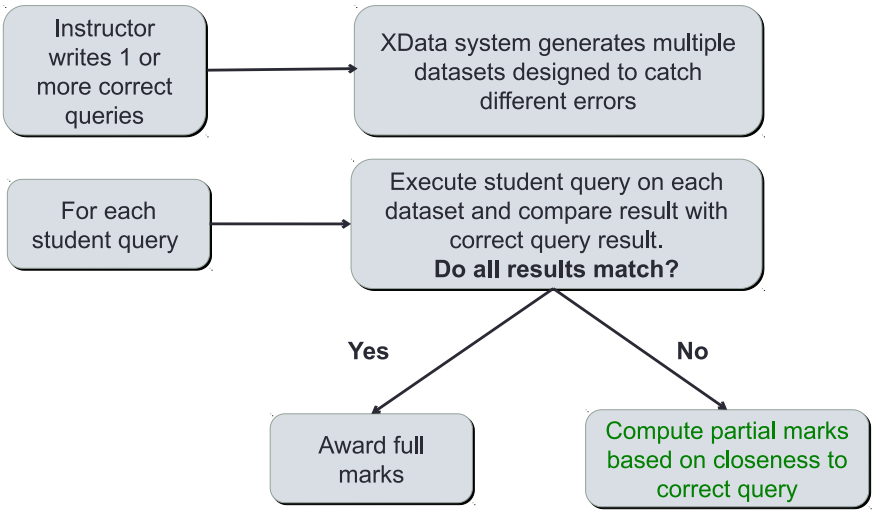

Figure 1: Automated Grading Workflow

dataset may not be able to find the difference unless the dataset has a tuple where the department budget is less than 70000. Thus a fixed dataset may also miss such errors in grading.

Even when the student query is incorrect, the grader is often expected to provide partial marks to the student query based on how close the student query is to being correct. A naive approach of counting the number of datasets for which a query gives the correct answer may not be fair for partial marking since a small error may cause many or all test datasets to fail. For example, if a student used a selection condition $a > 10$ instead of $a < 10$, most test datasets would fail. Conversely, a query with basic conceptual errors may still give the correct result on some datasets, especially ones where an empty answer is expected. Awarding partial marks manually is tedious and error-prone. Let us consider the following correct query provided by the instructor

```
SELECT * FROM r INNER JOIN s ON (r.A=s.A) WHERE r.A>10
```

and a student submitted the query

```
SELECT * FROM r INNER JOIN s ON (r.A=s.B) WHERE s.A>10
```

A grader manually evaluating the student query above may deduct marks for two errors - one for the join condition and another for the selection condition. However, if the join condition in the student query is fixed, the student query is equivalent to the given correct query since now r.A and s.A are equivalent in the student query. Hence only marks for one error should have been deducted.

In a database course, it would also be helpful for the student to receive specific feedback as to where they went wrong and how their mistakes could have been corrected. This is very time-consuming for TAs and is rarely done well even for moderately sized classes. With the growing popularity of online courses, where students expect instant feedback on the answers they submit there is an increasing need for automated and instantaneous feedback, Manual grading does not scale for large online courses, and just showing datasets where the query gave a wrong result may not provide clear feedback to students.

In our work on XData [1–3], we developed techniques to automatically grade student queries, given a set of correct queries. XData has two steps for grading student SQL queries as shown in Figure 1. The instructor first provides the question text and some correct queries. Based on the correct queries, XData generates multiple datasets that are tailored to catch different types of errors on the given queries. Since SQL queries may be written in several different ways, allowing instructors to specify multiple correct queries allows us to ensure more coverage of test cases. It also helps us get more query correct structures which is useful for our partial marking technique. The test data generation technique can also be used to test database queries and applications as described in [1, 4].

When evaluating a student query, the student query is run against the datasets generated by XData and the results are compared with the correct query. For correct student queries, the results generated by the student query and the instructor query would be the same across all generated datasets. Such queries would be awarded full marks. We note that techniques for checking query equivalence could potentially be used to check for equivalence of a student query to a correct query, but the state of the art for equivalence checking does not handle many SQL features such as null values, and has limitations in reasoning about equivalence with the given database constraints. While there is a risk of labeling an incorrect query as correct using our approach, we have not found it to be an issue in practice.

For incorrect queries, XData compares the student query with the correct query using an edit-based technique and provides a score as well as the changes that need to be made in the student query to make it a correct query. Our approach scales to large class sizes and can grade student queries and provide feedback instantly.

In this article, we first discuss, in Section 1, techniques for automatically generating test data and how the test data generated can be used to check the correctness of SQL queries submitted by students. In Section 3, we show how edit-based grading can be used to both award partial marks and provide individualized feedback for incorrect student queries. We share our experience of using the XData grading system in Section 4 and discuss related work in Section 5. We conclude the article in Section 6 and discuss some open challenges.

2 Using Datasets to Check Correctness

Test data generation in XData is done based on the correct queries provided by the instructor. Unlike fixed datasets that are query agnostic, these datasets are designed to catch errors based on the correct queries provided by the instructor and are hence much better at catching errors.

2.1 Common Errors in SQL Queries

Students make several types of errors when writing SQL queries. Some students may use an inner join when a left outer join was required, others may use a count aggregation when a count distinct was needed. Mutation testing is a well-known approach to check the adequacy of test cases for a program [5]. We use a similar approach for mutation testing of SQL queries. We consider mutations as (syntactically correct) changes to an SQL query. Errors made by student queries may be seen as mutations of the correct query and the incorrect query is called a non-equivalent mutant of the correct query. A dataset that produces different results on the correct and incorrect queries is said to kill the mutation.

XData produces multiple datasets. The first dataset is aimed to produce a non-empty result for the query. This dataset itself kills several mutations. For the remaining datasets, XData considers single mutations at a time on the correct query provided by the instructor and generates datasets that are targeted to kill the mutations. Each dataset is marked with a tag to indicate which type of mutations a dataset is designed to catch. Note that a dataset designed to catch one type of mutation may catch other types of mutations as well.

XData considers a large number of mutations in SQL queries including but not limited to the following.

- **Join type mutation**: A join type mutation involves replacing one of `{INNER, LEFT OUTER, RIGHT OUTER} JOIN` with another. Since the same join query may be written using different join orders, XData considers mutations across different join orders. Mutations involving missing or additional join orders are also considered.

- **Selection predicate mutation**: For selection conditions, XData considers mutations of the relational operator where any occurrence of one of $\{=, <>, <, >, \leq, \geq\}$ is replaced by another or if the selection condition is missing. XData also considers mutations of selection predicates between `IS NULL` and `NOT IS NULL` and for missing `IS NULL`. These predicates may be on integer, floating, text attributes or even aggregates. Mutations involving changing the constant in the selection condition are also considered.

- **Aggregation mutation**: Aggregations may be either unconstrained (at the root of the query tree) or constrained (having a condition with the aggregate). In both cases, the aggregation function can be mutated among `MAX, MIN, SUM, AVG, COUNT` and their `DISTINCT` versions. Mutations involving `COUNT(attr)` to `COUNT(*)`, in case `attr` is nullable, are also considered.

- **Group by attribute mutation**: For queries involving the `GROUP BY` clause, XData considers mutations involving additional or missing group by attributes both in the presence and absence of the `HAVING` clause.

- **Like operator mutation**: Like operators are used in SQL to match patterns in text attributes. SQL like operators include `LIKE, NOT LIKE, ILIKE and NOT ILIKE`. XData considers mutation of any one SQL like operator to other like operators. XData also considers mutations in the patterns used with the LIKE operator such as replacing as '%' with '_' and vice versa or missing '%' or '_' in the pattern.

- **Nested subquery mutations**: XData considers mutations between `IN` $vs.$ `NOT IN`, `EXISTS` $vs.$ `NOT EXISTS` and `ALL` $vs.$ `ANY/SOME`. Mutations on the queries in the nested subquery are also considered for test data generation so that errors inside subqueries are also caught.

- **Set operator mutations**: Set operators are used in compound queries to combine the results of two underlying results. Set operator mutations include changing one of the following operators to another: `UNION, UNION ALL, INTERSECT, INTERSECT ALL, EXCEPT, EXCEPT ALL`. Similar to nested subqueries, mutations of the subqueries whose results are input to these set operators are also considered.

- **Distinct mutation**: Duplicates in the results may be filtered using the `DISTINCT` clause. XData considers mutations of a missing or extraneous `DISTINCT` clause.

2.2 Test Data Generation to Detect Errors

For each type of mutation, we design specific conditions that the datasets must satisfy in order to kill such mutations. Let us take the following example query to demonstrate the mutations considered for queries with join and selections and how we generate datasets to kill the mutations.

```
    SELECT course.course_id
FROM course INNER JOIN takes USING(course_id)
WHERE course.credits >= 6
```

Some of the mutations that XData would consider and the techniques to kill those are the following.

1. **Join type mutation**: Consider the mutation from department INNER JOIN course to department LEFT OUTER JOIN course. In order to kill this mutation, we need to ensure that there is a tuple in department relation that does not satisfy the join condition with any tuple in course relation. The INNER JOIN query would not output that tuple in the department relation while the LEFT OUTER JOIN would.

   In general, a join query can be specified in a join order independent fashion, with many equivalent join orders for a given query. Hence, the number of join type mutations across all these orders is exponential. From the join conditions specified in the query, XData forms equivalence classes of <relation, attribute> pairs such that elements in the same equivalence class need to be assigned the same value to meet (one or more) join conditions. Using these equivalence classes, XData generates a linear number of datasets to kill join type mutations across all join orderings. If a pair of relations involve multiple join conditions

2. **Selection Predicate mutation**: For killing mutations for the selection condition A1 relop A2, XData generates 3 datasets where tuples satisfy the conditions (1) A1 $>$ A2, (2) A1 $<$ A2, and (3) A1 $=$ A2. These three datasets kill all non- equivalent mutations from one relop to another relop. These datasets also kill mutations because of missing selection conditions.

For the given query example the constraints would be (1) `course.credits>6`, (2) `course.credits<6` and (3) `course.credits=6`. If the student query uses a different operator instead of $>=$ or misses the selection condition one of the three datasets will catch the error.

Details on test data generation for killing other types of mutations are presented in [1]. We omit the details here for brevity.

It is not sufficient for only the conditions for killing mutations to be satisfied when generating the test database. The difference at one level, say the join condition must change the result of the query for the difference to be observed in the query result. For the given example query, consider the join mutation. If all tuples in course have grade less than 6, both the `INNER JOIN` and the `LEFT OUTER JOIN` would give empty results. Hence, when generating a dataset, XData ensures that the tuple has `grade >= 6`.

In order to generate a dataset, we generate constraints using an SMT solver [6]. In XData, we support CVC3 [7], CVC4 [8] as well as Z3 [9] as the constraint solvers. We encode text attributes as enumerates types and enumerates types are modeled as subtypes of integers or rationals. A tuple type is created for each relation to represent one row and an array of tuples represents the relation table. We also add other database constraints such as primary key and foreign key constraints, unique attribute constraints as well as domain constraints. The domain constraints ensure that we generate the correct types of values for each column of the database, the values generated are within the range specified by the schema and only nullable columns can have NULL values.

We also note that some mutations may be semantically equivalent to the correct queries and it may not be possible to kill such mutations. For such cases, the constraints for killing the mutations would not be satisfiable and the SMT solver would fail to generate the dataset to kill the mutation.

For the given query, a simplified version of the constraints, to generate the dataset that produces a non-empty result would be as follows (assuming none of the columns are nullable).

```
%Data definition
DATATYPE course_id = CS-101 | BIO-301 | CS-312 | PHY-101 END;
credits:TYPE = SUBTYPE (LAMBDA (x: INT): x > 1 AND x < 11);
course_tuple_type:TYPE = [course_id,credits];
course: ARRAY INT OF course_tuple_type;

%Primary key constraints
ASSERT FORALL(i:course_index, j:course_index):
       course[i].0 = course[j].[0] => course[i].1 = course[j].1

%Foreign key constraints
ASSERT FORALL(i: takes_index):
       EXISTS (j: course_index): takes[i].1 = course[j].0;

%Query conditions
ASSERT course[1].0 = takes[1].1;
ASSERT course[1].1 >= 6;
```

To support nullable columns, we add additional values (outside the domain of the column) for the datatype that correspond to NULL values and explicitly mark those as NULL values. Also, in practice, we found that unfolding the constraints (i.e., specifying the constraints for each tuple instead of FORALL, NOT EXISTS) gives us much better performance [10]. We decide the number of tuples upfront and assert the constraint on each tuple.

2.3 Evaluating Correctness of Student Queries

The dataset generation for each correct query is done once across all students. Based on the datasets generated, XData compares the results of each student query and each correct query provided by the instructor. If all results of the student query are found to match that of a correct query, the student query passes that correct query.

When an instructor specifies multiple correct queries, XData allows the instructor to specify one of the two options

- The additional queries were added to provide more coverage and better testing and all correct queries are equivalent. The student query will need to pass all correct queries for it to be marked as correct.
- The question test provided by the instructor was ambiguous and there could be multiple interpretations of the correct result. In this case, the student query will be marked correct if it passes any one of the correct queries.

The XData system ensures that student queries are safely executed on a different database using temporary tables to ensure that their queries do not interfere with the main database or that the queries of one evaluation do not interfere with another.

3 Edit Based Suggestions For Learning

Generating test data using correct queries provided by the instructor works great for finding student queries with errors. However, once the query is found to be incorrect, the student should be awarded partial marks based on the extent of correctness. It is also useful to make suggestions to the students so that can understand the errors in their queries.

One way to award marks for correctness could have been to use the fraction of datasets (generated by XData) that the student query could pass. This approach turns out to be unfair and could penalize small errors heavily while providing a better score to queries that have more errors. Such examples are shown in [3]. Another way to grade student queries would be to just check for the differences between the correct query and the student query. However, this approach may deduct more marks than required as explained below.

3.1 Approach

The approach we use instead in XData is to compare the student query to each correct query and make changes or edits to the student query to attempt to make it equivalent to the correct query. After each edit, the student query is compared to the correct query and the changes are stopped once the student query is equivalent to the correct query. Checking for equivalence after each edit could have been done using test data generation but that would be very expensive and grading each student query could take minutes. Other approaches for checking equivalence such as Cosette [11] as well as techniques based on tableau [12] work on a limited subset of query constructs and were hence not considered.

Marks are deducted based on the required edits. The edits required are also used to suggest what changes the students should have made to their query to make it correct, thereby providing individualized feedback to each student without any additional human involvement. For each type of query construct being edited, the instructor of the course can specify the weight for that edit. For example, for a query where the instructor is evaluating the student's ability to write aggregations correctly, the instructor may want to deduct more marks for errors in aggregation than for other errors. By default, XData assigns equal weight to each edit being considered.

In general, more than one edit may be needed to make the student query equivalent to the correct query. It is important to note that the order in which the edits are made is also important and that is not sufficient to just compare the query trees of the correct query and the student query to find the changes. One edit to a student query

may allow us to rewrite other parts of the query in a different way. As an example consider the following pair of queries from Section 1.

- Correct query: `SELECT * FROM r INNER JOIN s ON (r.A=s.A) WHERE r.A>10`
- Student query: `SELECT * FROM r INNER JOIN s ON (r.A=s.B) WHERE s.A>10`

There are two differences between the student query and the correct query - the selection condition and the join condition. If the student query is graded just based on these differences, marks corresponding to two edits would be deducted. Even if we grade based on the edits required, if the selection condition is edited first, followed by the join condition 2 edits would be required. On the other hand, if the join condition is edited first and the join condition in the student query is changed to `r.A=s.A`, the selection condition in the student query becomes equivalent to that of the correct query.

3.2 Query Canonicalization

The student and correct query may be written in different ways. For example, the correct query may use a selection condition `A>5` while the student query may write the condition as `NOT(A<=5)`. In order to compare the query structure of the student query to the correct query, we need to make them comparable. The student query and the correct query are made comparable by using canonicalizations.

XData considers two types of canonicalizations:

- **Syntactic Canonicalization**: This is the pre-processing step to reduce irrelevant syntactic differences. These include attribute disambiguation, replacing `NOT, BETWEEN, and WITH` constructs and removing `ORDER BY` from subqueries.
- **Semantic Canonicalization**: In this step, based on the query conditions and the database constraints, the queries are canonicalized semantically. Such canonicalizations include but are not limited to removing distinct clauses based on primary key information and converting outer joins to inner joins based on non-nullable foreign key information.

A detailed list of the canonicalizations is provided in [3]. Such canonicalization rules are often used in query optimizers. However, the goal of the canonicalizations in XData is to get more standard forms of the query.

XData also flattens the query tree where possible (`INNER JOIN, UNION(ALL), INTERSECT(ALL)` as well as predicates involving AND or OR). For the parsed query tree shown in Figure 2, XData would flatten the tree to the one shown in Figure 3. The flattened tree children are compared in an ordered way for non-commutative operators such as `LEFT OUTER JOIN, EXCEPT(ALL)` and `ORDER BY` attribute lists while for commutative operators the order of operands is ignored while matching.

3.3 Edit Sequence Based Grading

XData considers the following form of edits to the flattened tree generated from the student query.

- inserting a node/subtree into the flattened tree
- removing a node/subtree from the flattened tree
- replacing an existing node/subtree from a flattened tree with another node/subtree in the flattened tree
- moving a node/subtree from one position of the flattened tree to another

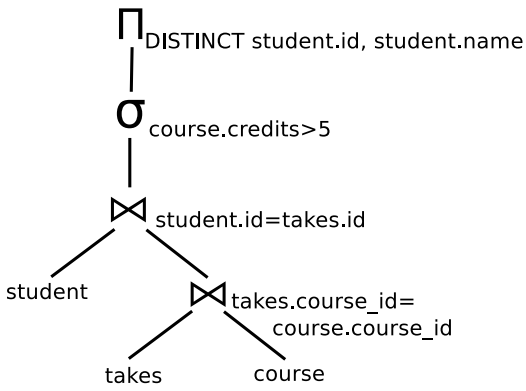

Figure 2: Parsed Tree From Query

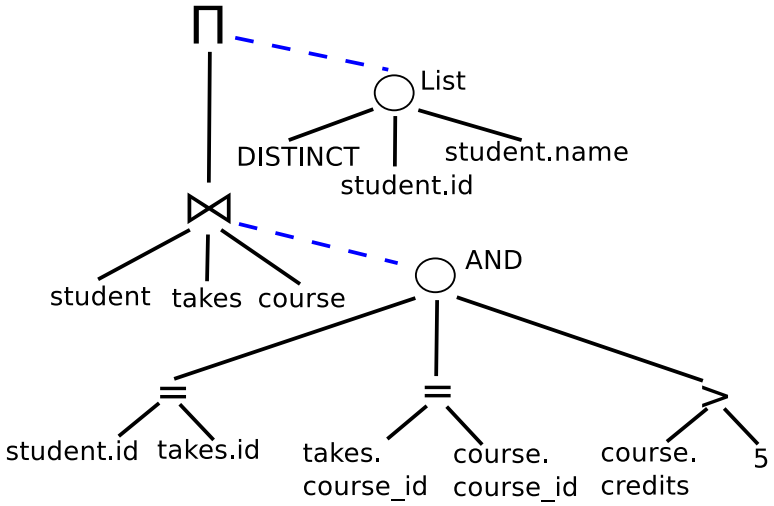

Figure 3: Flattened Tree

When editing a flattened tree generated from a student query, an infinite number of possible edits could be made. However only edits that make the query more similar to the correct query would be useful. In order to add edits that make the student query more similar to the correct query, XData uses the correct query to guide the edits that are generated. The guided edits are based on the differences the student flattened student query tree has with the flattened tree of the correct query. For example, query attributes/conditions/constructs not used in the correct query but present in the student query will be removed when generating edits. For each query edit that XData generates, marks corresponding to the edit as configured by the instructor are deducted. The marks deducted for the edit can be considered the edit cost.

XData can generate multiple guided edits on the student query at each step. From each of these edited queries, more edits are possible. Consider a graph whose nodes are all queries for the given schema. For a student query SQ, edits of the query are also nodes in the graph. Let these edited queries be connected to query SQ with an edge whose weight is the edit cost of the edited query. Canonically equivalent queries, i.e. their canonical forms are the same, are connected by 0 cost edges. The sequence of edits that has the least cumulative cost can now be determined based on the shortest path in this graph from the student query node in the graph to a correct query node. Partial marks can now be awarded based on this shortest path. Since the weight of each edge, which represents the cost of edit is non-negative, the shortest possible path may be found using Dijkstra's shortest path algorithm. Hence, given a set of edits and using a given set of canonicalizations, the shortest path in the graph, as defined above, gives the edit sequence with the least cost. We note that the graph discussed above is for the ease of understanding only and XData does not proactively try to generate the entire graph. In practice, XData generates the nodes of the graph on the fly as needed.

In case the instructor specifies multiple correct queries, the edit sequence based algorithm run based on all correct queries and the best partial marks obtained is awarded.

3.4 Heuristic solution

Even with using only guided edits, the search space is still very large for larger queries if we consider all guided edits to get the shortest path from the student query to the correct query. Hence in practice, we use a greedy heuristic. The heuristic uses a cost benefit model. For each edited query we can get an estimate of how incorrect the query is by finding the differences between the canonicalized versions of the edited query and the correct query. A weighted sum (based on the weight assigned by the instructor for each edit) can be used to find an edit distance which we call the *canonicalized edit distance*. Each guided edit reduces the canonicalized edit distance

to the correct query. The reduction in the canonicalized edit distance from the edit is the benefit of the edit.

For the heuristic algorithm, at each edit step, we find the $benefit - cost$ for each edit. We then pick the edit that has the highest value of the $benefit - cost$ and use it to generate further edits. The remaining edits are discarded at each step. Using the heuristic allows XData to search a much smaller search space. For the incorrect student queries that we had in our course, we found that the heuristic solution works as well and takes orders of magnitude less time as compared to the exhaustive solution [3].

4 Automated Grading Experience

We have successfully used the XData automated grading system across several offerings of undergraduate database courses at IIT Bombay. Before using XData in a course, we empirically confirmed, using results from a previous database course, that XData was able to catch as many as or more errors than when the grading was done manually or when fixed datasets are used for grading. This result was consistent across all questions that XData was able to grade. We found, in several cases, that manual grading had missed subtle errors such as a missing distinct clause.

For the initial course offerings, we used only generated dataset based grading to check for correctness and had to award partial marks manually. The dataset-based grading significantly reduced the human effort involved and allowed us to catch more errors than would have been possible with manual grading. In several cases, however, students were not satisfied since they could not intuitively understand how marks had been deducted or what the error in their specific query was. The tagged datasets on which the student queries failed were shown but often there would be too many of them to understand the specific error. A dataset designed to catch one type of mutation may catch other types of mutations as well and hence it was not always clear what the actual error was. Several students contested the scores that they had been awarded when their query was found to be incorrect. Awarding partial marks manually by the graders was still tedious since students often wrote queries in very different and complex ways. Manually transforming such student queries to a simpler form was difficult. For instance, in one case a correct query involved using a NOT IN clause and some students used a combination of multiple EXCEPT and INTERSECT clauses.

When using partial marking in combination with dataset based evaluation, we reduced the human effort as well as provided much better feedback. We experienced fewer students contest grades when it was assigned automatically compared to when a human would manually assign grades. The edit-based guidance was also useful for students to understand where they went wrong. For the first course setting where we used automated partial marking for evaluation, we found that across 1800 student query submissions that were graded by XData, only 2 queries were contested by students. In both cases, we traced back the errors in grade to bugs in our code. One of the main challenges when using automated grading was to provide sufficient types of correct queries that covered the student queries.

Since the edit sequence based guided edits provide a way to change an incorrect student query to a correct query, the guided edits can be used by students to learn the mistakes that they made and how the mistakes could have been corrected. Such feedback was very helpful especially for beginners to understand how to write correct SQL queries.

5 Related Work

There has been a significant interest in testing query equivalence or correctness and in automated grading. Related work include the following.

Checking Query Equivalence

XData uses test data generation based on mutation testing to generate test datasets to check the equivalence of the correct query and student query. Tuya et al. [13] describe a number of possible mutations for SQL queries. However, they do not handle test data generation for killing these mutations. Other approaches on testing query equivalence using datasets include Qex [14] and SQL full predicate coverage by Riva et al. [15]. Test data generation in these systems is aimed at testing SQL queries in database applications and they consider only a limited subset of SQL query constructs.

Techniques based on tableau [16] and its extensions [12, 17] can be used to check for query equivalence for a restricted class of conjunctive queries. Cosette [11] and U-semiring [18] can also be used to check for SQL semantic equivalence using a restricted set of axioms. SPES [19] uses a symbolic approach for checking query equivalence on SQL queries under bag semantics for select-project-join(SPJ) queries as well as aggregate and union queries.

Grading SQL Queries

The Gradiance system [20] provides multiple choice type questions where the instructor has to provide some correct as well as incorrect answers and explanations of incorrectness. In such assignment settings, since the students are only able to select from limited options, subtle mistakes that students could make may not always be covered by the incorrect answers. Gradescope [21], uses a fixed dataset to evaluate the correctness of student SQL queries. As discussed earlier, using fixed datasets may miss errors and would not be able to provide any meaningful feedback to incorrect student queries. RATest [22] provides feedback for incorrect queries by deriving small datasets that produce different results in a student query as compared to a correct query. I-Rex [23] allows users to trace the SQL query evaluation for each constituent block in the query execution.

Automated Grading for Programming Assignments

Grading programming assignments has some similar challenges as grading database queries. CPSGrader [24] can grade programming assignments for cyber-physical systems using constraints synthesis and uses reference solutions to provide feedback. AutoGrader [25] can grade introductory level python programs and provide student feedback using program synthesis and high-level error modeling specifications. However, AutoGrader can only model specific predictable errors.

SARFGEN [26] provides feedback to student queries by aligning student programs to similar correct reference programs and finds the minimum number of edits to the student program to match the chosen reference program. This approach is similar to our approach of edit-based grading. However, SQL queries have database constraints that are not part of the query but need to be accounted for during edits and equivalence checking. Hence we have a more complex semantic canonicalization step that can take into account constraints such as primary keys and foreign keys.

6 Conclusion and Open Challenges

The XData system is very useful for grading assignments on SQL queries and for providing feedback to students about mistakes they made. For large classes or online courses, such automated grading systems are essential and the automated individualized feedback would be very useful to new learners. Our experience in using the grading system has been very positive from both the teaching assistants as well as the students. The XData grading system as well as the source code are available for download from `http://www.cse.iitb.ac.in/infolab/xdata`.

One of the key challenges in the XData grading scheme is the number of ways a correct SQL query can be written. We have found that students sometimes write queries using a very different approach than we had

anticipated. Catching errors with such approaches as well as awarding partial marks in these cases may be challenging. One way to address these would be to automatically group student query submissions and generate datasets for one student query in each group. We could then use the datasets to identify correct queries from the group and automatically add these as additional correct queries for grading. Another key challenge is in dealing with student queries that have additional query constructs that do not affect the query results. One of the cases that we found in our course was when a student had used the query `Q UNION Q'` where Q' was always empty. The query had one error in Q but marks were also deducted based on edit for Q'.

Acknowledgments: We thank all students and researchers who worked on the XData project as well as those who used the system and provided their valuable feedback. This work was partially supported by research funding and a PhD fellowship from Tata Consultancy Services.

References

[1] B. Chandra, B. Chawda, B. Kar, K. V. M. Reddy, S. Shah, and S. Sudarshan, "Data generation for testing and grading SQL queries," *VLDB Journal*, vol. 24, no. 6, pp. 731–755, 2015.

[2] A. Bhangadiya, B. Chandra, B. Kar, B. Radhakrishnan, K. V. M. Reddy, S. Shah, and S. Sudarshan, "The XDa-TA system for automated grading of SQL query assignments," in *International Conference on Data Engineering (ICDE)*, 2015.

[3] B. Chandra, A. Banerjee, U. Hazra, M. Joseph, and S. Sudarshan, "Edit based grading of SQL queries," in *CODS-COMAD 2021: 8th ACM IKDD CODS and 26th COMAD*, 2021, pp. 56–64.

[4] P. Agrawal, B. Chandra, K. V. Emani, N. Garg, and S. Sudarshan, "Test data generation for database applications," in *International Conference on Data Engineering (ICDE)*, 2018, pp. 1621–1624.

[5] A. J. Offutt, "A practical system for mutation testing: Help for the common programmer," in *International Conference on Test (ICT)*, 1994, pp. 824–830.

[6] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. IOS Press, 2009, vol. 4, ch. 8.

[7] C. Barrett and C. Tinelli, "CVC3," in *Computer Aided Verification (CAV)*, 2007, pp. 298–302.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 171–177.

[9] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[10] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. P. Gupta, and D. Vira, "Generating test data for killing SQL mutants: A constraint-based approach," in *International Conference on Data Engineering (ICDE)*, 2011.

[11] S. Chu, C. Wang, K. Weitz, and A. Cheung, "Cosette: An Automated Prover for SQL," in *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[12] Y. E. Ioannidis and R. Ramakrishnan, "Containment of conjunctive queries: Beyond relations as sets," in *ACM Transactions on Database Systems (TODS)*, vol. 20, no. 3, 1995, pp. 288–324.

[13] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva, "Mutating database queries," in *Information and Software Technology*, 2007, pp. 398–417.

[14] M. Veanes, N. Tillmann, and J. de Halleux, "Qex: Symbolic SQL Query Explorer," in *Logic Programming and Automated Reasoning (LPAR)*, 2010, pp. 425–446.

[15] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya, "Constraint-based test database generation for SQL queries," in *Workshop on Automation of Software Test*, ser. AST '10, 2010, pp. 67–74.

[16] A. V. Aho, Y. Sagiv, and J. D. Ullman, "Equivalences among relational expressions," *SIAM Journal on Computing (SICOMP)*, vol. 8, no. 2, pp. 218–246, 1979.

[17] Y. Sagiv and M. Yannakakis, "Equivalence among relational expressions with the union and difference operation," in *International Conference on Very Large Data Bases (VLDB)*, 1978, pp. 535–548.

[18] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu, "Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries," in *Proceedings of the VLDB Endowment (PVLDB)*, vol. 11, no. 11, 2018, pp. 1482–1495.

[19] Q. Zhou, J. Arulraj, S. Navathe, W. Harris, and J. Wu, "SPES: A symbolic approach to proving query equivalence under bag semantics," in *International Conference on Data Engineering (ICDE)*, 2022.

[20] "Gradiance: The Gradiance service for database systems," http://www.gradiance.com/db.html (Retrieved on Aug. 1, 2022).

[21] "Gradescope," https://www.gradescope.com (Retrieved on Aug. 1, 2022).

[22] Z. Miao, S. Roy, and J. Yang, "Explaining wrong queries using small examples," in *ACM SIGMOD International Conference on Management of Data (SIGMOD)"*, 2019, pp. 503–520.

[23] Y. Hu, Z. Miao, Z. Leong, H. Lim, Z. Zheng, S. Roy, K. Stephens-Martinez, and J. Yang, "I-rex: An interactive relational query debugger for SQL," in *SIGCSE 2022: The 53rd ACM Technical Symposium on Computer Science Education*, 2022, p. 1180.

[24] G. Juniwal, A. Donzé, J. C. Jensen, and S. A. Seshia, "CPSGrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory," in *International Conference on Embedded Software (EMSOFT)*, 2014.

[25] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013, pp. 15–26.

[26] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 481–495.