Partial Marking for Automated Grading of SQL Queries *

Bikash Chandra, Mathew Joseph, Bharath Radhakrishnan †
Shreevidhya Acharya, S. Sudarshan

IIT Bombay

{bikash,mathewj,bharathrk13,shreevidya,sudarsha}@cse.iitb.ac.in

ABSTRACT

The XData system, currently being developed at IIT Bombay, provides an automated and interactive platform for grading student SQL queries, as well as for learning SQL. Prior work on the XData system focused on generating query specific test cases to catch common errors in queries. These test cases are used to check whether the student queries are correct or not. For grading student assignments, it is usually not sufficient to just check if a query is correct: if the query is incorrect, partial marks may need to be given, depending on how close the query is to being correct. In this paper, we extend the XData system by adding features that enable awarding of partial marks to incorrect student queries. Our system is able to go beyond numerous syntactic features when comparing a student query with a correct query. These features of our grading system allow the grading of SQL queries to be fully automated, and scalable to even large class sizes such as those of MOOCs.

1. INTRODUCTION

Grading of SQL queries is traditionally done by instructors and teaching assistants (TAs), by (a) reading and manually comparing a student query with the correct query, and/or (b) by comparing the query results of a correct query and the student query on one or more ad hoc datasets. Manually reading and grading SQL queries is very tedious when the number of students is large, while also being error prone, especially for complex queries with many possible ways of writing the query. Grading SQL queries by using fixed query-independent or manually constructed datasets is used by systems such as Gradience [3]. However, these datasets might miss errors, and could lead to incorrect queries being marked as correct. In particular, subtle errors might be missed.

*Work partially supported by a research grant from Tata Consultancy Services

†Currently working at Amazon, India

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.

Proceedings of the VLDB Endowment, Vol. 9, No. 13

Copyright 2016 VLDB Endowment 2150-8097/16/09.

The XData system [6, 2] generates a number of datasets specific to a query Q (provided by the instructor), such that common errors in Q can be caught. In [1], we show how datasets generated to catch errors on an instructor query can be used to test if a student query is correct or not. For grading, detection of incorrect queries is necessary, but is not sufficient, since it is usually important to give partial marks to queries that are partially correct.

An initial approach that we tried was to assign partial marks based on the fraction of datasets on which the results of the student query and an instructor query match. This approach, unfortunately, gives very poor results. For example, if a student writes an incorrect selection condition `r.A<5` instead of `r.A>5`, it would fail almost all datasets and the student would lose almost all marks. On the other hand, a student query which intentionally returns the empty set every time (e.g. by using a selection condition `1=2`) may be able to get some marks, since its output would match on all datasets on which the instructor query is expected to give an empty result (and there are usually several such datasets).

In this paper, we summarize our techniques for awarding partial marks to student queries, based on how close the student query is to an instructor query. The same SQL query can be written using different syntactic variations, which must be taken into account when comparing queries. For example, if an attribute `r.A` is an integer, and an instructor query uses a selection condition `r.A>5`, the student may write the same condition as `r.A>=6`. SQL queries may also contain extra relations or conditions which may be removed without affecting the result of the SQL query. Hence, we first canonicalize the instructor query and the student query to make the queries comparable.

The canonicalized queries are then broken into components. The components of the canonicalized student query are then matched with the components of the canonicalized instructor query. Matches are rewarded, while mismatches penalized, by assigning appropriate scores. Finally, a weighted aggregated score is computed, which gives a measure of how close the student query is to the given instructor query.

The same SQL query may be written in very different ways and canonicalization of the queries using our techniques may not be able to make the queries comparable. For partial marking to be more effective, the instructor can upload multiple correct queries, each written in a different way. Our partial marking system assigns marks to the student query by comparing it to each of the correct queries to get a partial mark with respect to each correct query, and then using the maximum of the marks thus obtained.

We ran our grading tool on SQL queries written by students and found our techniques to be very effective in assigning meaningful partial marks. Since our tool gives an explanation for the marks obtained and is more uniform in grading, we believe it will be perceived as more fair in awarding partial marks, as compared to a human grader.

2. BACKGROUND

Most incorrect queries are small syntactic deviations (or mutations) of the correct queries. A mutation is defined as a single syntactically correct change to the correct query and the changed query is said to be a mutant of the original query. A dataset that is able to produce different results on the correct query and its mutant (thereby showing that the mutant is not equivalent to a correct query) is said to kill the mutation.

The XData [6, 2] system takes a query as input and generates one or more datasets such that the common errors/mutations are caught. The XData system handles a large variety of SQL constructs including selections, joins, aggregates, subqueries and set operators. Currently, the mutations targeted by XData includes join mutations, comparison operator mutations, aggregate operator mutations, group by attribute mutations, like mutations, subquery mutations, set operator mutations and distinct mutations. For each correct query, XData generates multiple datasets, each targeted to kill one or more mutations.

Using the XData grading system, instructors can create SQL query assignments. Instructors need to provide details about the schema on which the queries are based, along with the database connection details on which their queries will be run. Instructors can set two types of assignments: (a) graded assignments - where the student queries are evaluated after the deadline is over, and (b) learning assignments - where the student queries are graded immediately and feedback is provided. For each assignment, the instructor can create multiple questions and for each question, the instructor can provide one or more correct answers.

For each question that has more than one correct query, the instructor can mark if test cases from all queries need to pass for the student query (more erroneous queries can be caught) to be marked correct or only one of the queries need to match the correct query (the question was ambiguous). Datasets are generated based on the queries provided by the instructor.

Evaluating correctness of student SQL queries is done by comparing the result of the student query with that of a correct query on the generated datasets.

3. PARTIAL MARKING

If the student query is marked as correct, using the generated datasets, the query is awarded full marks. However, in case the student query is incorrect, we use techniques that we describe in this section to assign partial marks.

The basic idea is to compare the student query with an instructor query and award partial marks based on how close the student query is to an instructor query. To perform the comparison, we first canonicalize both the student query and the instructor query to remove any irrelevant syntactic variations, and then perform a component wise comparison to award partial marks.

It might appear that if we can canonicalize an instructor query and student query, and then compare them, we can detect if the student query is correct without running the query on any datasets. However, canonicalization cannot in general guarantee that the instructor query and student query will be canonicalized to the same form even if they are actually equivalent. Thus, a correct student query may get less than full marks. We, therefore, first check for correctness by using datasets generated by the XData system. A query judged as correct gets full marks; if a query is marked as incorrect, then the techniques described in this section are used to award partial marks.

3.1 Initial Preprocessing

We first perform some preprocessing on the SQL query so that the query conditions are made comparable.

Attribute disambiguation: An attribute `A` without a relation is changed to `r.A` where `A` in inferred to be from `R`.

WITH Clause Elimination: Non-recursive WITH clauses are replaced in the query by expanding the WITH clauses inline.

BETWEEN Predicate Elimination: BETWEEN predicates are replaced with the equivalent conditions using the relational operators. For example `r.A BETWEEN 5 and 10` is replaced with `r.A>5 AND r.A<10`.

Normalization of Relational Predicates: Selection conditions involving NOT are converted to remove the NOT operator by adjusting the relational operator appropriately. For example, `NOT(A>B)` is converted to `A<=B`. Selection conditions involving > (resp. >=) are converted to < (resp. <=), by exchanging the operands; for example $A > B$ is converted to $B < A$. Selection conditions involving `A<B` are converted to `A<=B+1`, provided both operands are of integer type.

Normalization of Nested Queries: A nested subquery with an IN/ANY connective can be converted to use an EXISTS connective, by using the attributes involved in the IN/ANY connective to create a correlation condition. For example,

```
r.A >ANY (SELECT s.A FROM s WHERE s.B>10)
```

can be converted to

```
EXISTS (SELECT s.A FROM s WHERE s.B>10 AND r.A>s.A)
```

Join Processing: Any NATURAL INNER JOIN is replaced with an INNER JOIN with equivalent join conditions added using the ON clause. Occurrences of USING clause in JOIN conditions is replaced with ON clause with the equivalent join conditions.

3.2 Equivalence Classes of Attributes

Consider the following query

```
SELECT employee.deptId FROM  employee INNER JOIN
department ON employee.deptId=department.id
```

In this query, `SELECT department.id` can be used in place of `SELECT employee.deptId`, since the two attributes are guaranteed to have the same value thanks to the join condition, `employee.deptId = department.id`.

In general, when $A = B$, $B = C$, $C = D$ .., are conjuncts in the join conditions of a query, attributes A, B, C, D, ... are said to belong to the same *equivalence class*; any occurrence of an attribute in an equivalence class can be replaced with any other attribute from the equivalence class, at any place in the query tree above the occurrence of the join conditions, without changing the result of the query.

A canonicalization step is therefore performed by replacing all occurrences of an attribute above join condition, by the lexicographically least variable from its equivalence class. In the above query, since `department.id` lexicographically precedes `employee.deptid`, `employee.deptId` is replaced by `department.id` in the SELECT clause.

Mapping variables to equivalence classes is used by query optimizers for join reordering and correct selection estimation whereas we use it for comparing queries.

3.3 Join Minimization

Removal of redundant joins, and conversion of outer joins to inner joins, are well known steps in query optimization. We use them as part of our canonicalization, before comparing queries. Consider the following query:

```
Q1: SELECT employee.Id, department.Id
    FROM employee  INNER JOIN department
    ON employee.deptId=department.Id
```

and suppose that `employee.deptId` is non nullable, and is a foreign key referring to `department.Id`. The non-nullable foreign key dependency ensures that for each employee tuple t_1 there exists a matching department tuple t_2 (i.e., one s.t. $t_1[\text{deptId}] = t_2[\text{Id}]$). Since the projection attribute `department.Id` can be replaced by `employee.deptId` from the same equivalence class, the query can be rewritten to the equivalent query:

```
SELECT employee.Id, employee.deptId FROM employee
```

Join minimization as described above is used to remove redundant relations from both the student query and the instructor query.

Consider the query:

```
SELECT * FROM department LEFT OUTER JOIN
employee ON department.Id = employee.deptId
WHERE employee.deptId > 5
```

The selection condition, `employee.deptId > 5`, fails when `employee.deptId` has a null value. Thus the query is equivalent to the one where inner join is used instead of a left outer join. In general, if at a point in the query above a left-outer join, there is a null-rejecting condition on an attribute from the right input of the left outer join, we replace the left outer join by an inner join. The case of right outer join is symmetric.

Conversion of outer joins to inner joins is done before computing variable equivalence classes.

3.4 Functional Dependencies

Functional dependencies can be used to infer that textually different ORDER BY or GROUP BY clauses are actually equivalent [4], which is used for query optimization. We now describe how we use functional dependencies for comparison of ORDER BY and GROUP BY clauses of student and instructor queries.

Canonicalizing ORDER BY attributes

Consider an SQL query Q with the clause ORDER BY a, b. Let us suppose that Q satisfies the functional dependency $a \rightarrow b$, then Q is equivalent to a query Q' obtained by replacing the ordering clause with ORDER BY a. Due to the functional dependency, two tuples with the same value for a would have the same value for b, making the ordering by b irrelevant. ORDER BY clauses are canonicalized by removing all attributes that are functionally determined by other attributes appearing earlier in the ORDER BY clause.

Comparing GROUP BY attributes

Consider the following query

```
SELECT COUNT(*) FROM employee GROUP BY id, name
```

Suppose `id` functionally determines `name` (for example, because `id` is declared as a primary key). Then, the GROUP BY clause can be equivalently written as `GROUP BY id`.

However, unlike with ORDER BY clauses, there may be completely different sets of attributes that give the same grouping, and getting a unique canonicalization is not possible [4]. Instead, we check whether each attribute in the GROUP BY clause in the instructor query is present in the cover of the GROUP BY clause of the student query and *vice versa*; attributes missing in the student query, or extraneous in the student query, indicate errors.

Canonicalizing Duplicate Removal

Duplicate removal using SELECT DISTINCT can be redundant if there are no duplicates in the list of attributes; if we infer absence of duplicates, the DISTINCT clause can be removed. Similarly, for INTERSECT ALL absence of duplicates in at least of the inputs, and for EXCEPT ALL, in the left input, means we can drop the ALL clause. Primary key constraints on input relations, coupled with equality predicates in select and join predicates can be used to infer absence of duplicates in the result of joins, as described in [5].

DISTINCT clauses can be deleted from EXISTS/IN/ALL/ANY subqueries, as well as their NOT variants, regardless of the presence of duplicates.

3.5 Computing Partial Marks

The resulting queries after preprocessing, minimization, and canonicalization steps are compared with each other by a syntactical component matching and weighted marking technique. An SQL query is divided into components, such as SELECT list, list of relations in the FROM clause, WHERE clause predicates, set operators, etc., as well as subqueries which (recursively) have their own components.

Given a student query and instructor query, function *calculateScore* matches components of the canonicalized student query with the components of the canonicalized instructor query. Note that each component in general has subparts; for e.g., the subparts of a join or selection predicate would the conjuncts, the subparts of a GROUP BY, ORDER BY, or SELECT clause would be the attributes in the list.

For each component of the instructor query, the subparts from the instructor query are matched with the corresponding subparts from the student query. Note that wherever order is irrelevant (for example among conjuncts of a predicate, or the attributes of a GROUP BY clause), ordering is ignored when finding matches.

Missing subparts are penalized by giving marks for that component in proportion to the number of instructor query subparts that are actually present. Extraneous subparts in the student query are penalized, by assigning appropriate negative scores. Marks are computed in this manner for each subpart and added to get a mark for each component. The minimum score for each component is set to 0 so that the student query is not excessively penalized.

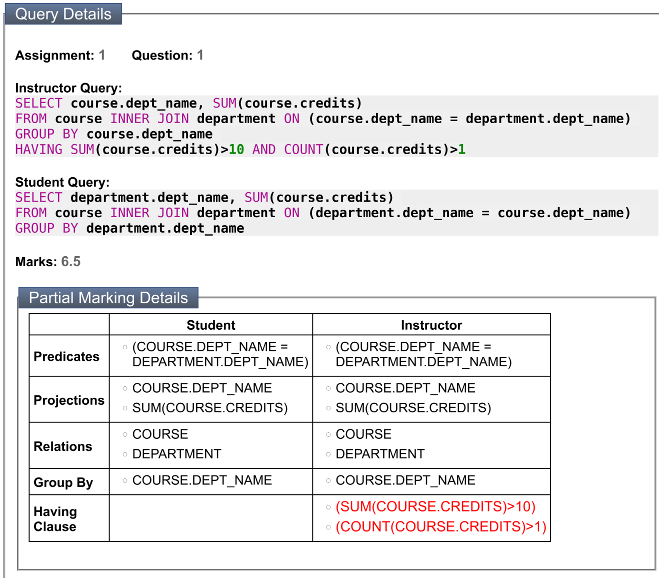

Figure 1: Component-wise partial marking

Note that marks for subqueries are computed by recursive calls to function *calculateScore*. In case there are multiple subqueries, the best matching pair are considered for computing the score. The inputs to outer joins are matched separately by recursive calls.

Each component of the instructor query is assigned a weight W_c; weights can be adjusted by the instructor using a GUI slide bar in our system. The marks assigned to a student query is computed as $\Sigma_{c \in \text{components}} W_c * M_c$, where W_c and M_c are the weight and marks assigned to a component. Student queries may have extraneous components that are not in the instructor query. Such extraneous components are penalized by negative marking for each such component.

Since the instructor also has the option to specify multiple correct queries, the final marks in such cases is the maximum of the set of all marks computed by applying *calculateScore* on the student query against each of the instructor queries. Once the evaluation is done students can see component wise comparison and marks obtained as shown in Figure 1.

Performance

The time taken for partial marking per student query is of the order of fractions of a second. The time taken for data generation is about 1-2 mins, but it is done only once for every instructor query, irrespective of the number of student queries. In a preliminary performance evaluation based on student queries collected from a course, we found that the partial marks awarded by our system to erroneous queries were well correlated with the marks assigned by TAs.

4. DEMONSTRATION

We demonstrate the grading tool using the University schema from [7] and the PostgreSQL database. The LTI interface provided by our system allows users logged in to a learning management system (Moodle, in our demo) to navigate to our system. The audience will able to interact with the system in the following modes:

- **Instructor mode**: Using this mode, users can create questions or modify existing ones. They may create new assignments and link the assignments to Moodle. For a given question, instructors will also be able to adjust partial marking parameters. Once some student queries are submitted the queries can be evaluated and the instructor can view the marks awarded for the student queries. They will also be able to observe how marks are assigned based on the comparison of canonicalized versions of the student query and the instructor query.
- **Student mode**: Using this mode, users will be able to browse the existing assignments and attempt questions. After evaluation, students can check whether their query is correct or not. In case their query is marked as incorrect, the system provides details regarding the datasets on which the query failed along with details of how partial marks were awarded.

The XData system can be downloaded from http://www.cse.iitb.ac.in/infolab/xdata/XData.

5. CONCLUSION

The grading interface has been used successfully at IIT Bombay for two offerings of the under-graduate database course. The extensions to partial marking are new but have been tested on existing student queries. We are confident that instructors and TAs of database courses will benefit from our system, and especially so for MOOCs.

Canonicalization of subqueries by decorrelation is an important area of future work. Canonicalization of DISTINCT placement in FROM clause subqueries versus outer queries is another area of future work.

6. REFERENCES

[1] A. Bhangdiya, 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 *ICDE*, pages 1468–1471, 2015.

[2] B. Chandra, B. Chawda, B. Kar, K. V. M. Reddy, S. Shah, and S. Sudarshan. Data generation for testing and grading SQL queries. *VLDB J.*, 24(6):731–755, 2015.

[3] Gradience: The gradiance service for database systems. http://www.gradiance.com/db.html.

[4] T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *VLDB*, pages 960–971, 2004.

[5] G. N. Paulley and P.-A. Larson. Exploiting uniqueness in query optimization. In *CASCON*, pages 804–822, 1993.

[6] 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 *ICDE*, 2011.

[7] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 6th edition, 2010.