Cosette Autograder with Hints for Query Correctness

Angela Xiao
University of California, Berkeley
angela.xiao@berkeley.edu

Brian DeLeonardis
University of California, Berkeley
bdeleonardis@berkeley.edu

John Yang
University of California, Berkeley
john.yang20@berkeley.edu

***Abstract*—The ability to determine whether two queries are semantically equivalent has countless applications, in terms of both practical usage as well as theoretical insight. In this paper, we contribute to the practical research by developing an autograder built on top of the Cosette prover [1]. Unlike current solutions for database class autograders, an autograder built on Cosette is guaranteed to always identify incorrect queries as well as potentially perform better. In addition to the grading features useful to instructors, our autograder additionally provides students with hints about why their queries might be incorrect.**

***Index Terms*—Databases, Query Processing, Query Equivalence, Computer Science Education**

I. Introduction

Cosette is an automated prover that determines whether two SQL queries are equivalent for a given schema. If the queries are semantically equivalent, it returns a proof of their equivalence. If the queries are not equivalent, it returns a counterexample dataset for which the two queries return different results. Up until now, Cosette, while interesting in its theoretical implications, has not been applied to practical usages. In this paper, we introduce an autograder that is built on top of Cosette. Our autograder not only determines whether student submitted queries are incorrect, but additionally leverages Cosette to provide the students with a hint as to where their query went wrong.

Typical grading applications for a databases class (like Berkeley's CS W186) will run both the correct query and the student query on a dataset and check both results for equality in order to determine whether or not the queries are "equivalent" as well. This approach has two major drawbacks. The first is that it might claim that a student query is correct when it is not actually logically the same as the instructor provided query, especially if the test dataset is limited in breadth and variety. The two queries might return the same result for the test dataset, but different results for other excluded datasets. The second issue with the standard autograder implementation is that it can have a significant runtime if the dataset it is being tested on is large (since it requires actually running the student query for every check).

The grading component of our autograder is relatively straightforward. The instructor provides the autograder with text files containing the correct queries for each question and the student submits a solution file with their queries for each question. For each question, the autograder sends both the correct query and the student query to Cosette to determine whether they are correct or not. This implementation solves the first issue (false positives for query correctness) because Cosette is guaranteed to be correct in its equivalence determinations. This implementation could also potentially solve the second issue if Cosette could quickly find an answer, but unfortunately, in our testing, this was not the case. None of our implementation is groundbreaking for the grading portion - the major contribution of this is that it demonstrates that it is possible to develop an easy-to-use real world application on top of Cosette.

The more interesting part of our work is the hint generation for incorrect queries. The goal of this portion of the autograder is to help students get to the correct query. An example of one of our generated hints is as follows.

```
Correct Query:
SELECT x.a
FROM r x
WHERE x.a > 2;

Student Query:
SELECT x.a
FROM r x;

Hint:
Your query returned too many rows for
at least some dataset.
Conditions that may be missing: [r.a > 2]
```

To see some more examples of hints our hint generator can create, see the appendix. These hints are generated through a combination of using Cosette's counterexample to learn about the type of error that is being created and our own analysis of the different fields of the query. We found that, through some relatively simple analysis, our hint generator correctly identifies the problem in the student query and can generate a useful hint for queries that take a similar approach to the instructor query. Our hint generator does not work well for queries that take a drastically different approach (i.e if the instructor used a UNION ALL and the student used a subquery). To summarize, the main contributions of this paper are the demonstration that a practical application can be built on the Cosette system (to possibly improve upon existing

solutions), and a novel hint generation system that can improve SQL learning for students.

This paper will give an overview on how to use our Cosette-driven autograder as well as an overview on the architecture behind the hint generation logic. Afterwards, we will discuss our performance findings and plans, as well as go into the limitations of our autograder and how future work aims to address these concerns.

II. Related Work

Since its publication in 2017, the Cosette paper has been cited 31 times, with many of the following works being either applications of query equivalence to a particular domain or further work in the direction of the theory behind how Cosette works. The related works we're interested in are more oriented towards practical use cases. The general idea of proving equivalence, whether it is between two database-driven applications [7] or even spark programs [8], is gaining traction as a a general solution to prove that a refactored program or database maintains the correctness or certain desired properties of a previous version. This related work gives us insight into the principles of proving query equivalence that either vary or stay constant across different domains. The problem of developing a representation of the program or code that makes proving equivalence possible (i.e. relational algebra, abstract syntax trees) is an interesting topic with many case studies that introduce ideas regarding how to implement the parsing and framing of a language's features. For instance, the Spark language's semantics translate well to Relational Algebra notation, which is then fed into an SMT based technique for proving Spark programs' equivalence.

In the last year, there has been a proliferation of work in the domain of debugging database queries using query equivalence, and a general overview survey of existing techniques suggests an existing, wide sentiment that typical users' debugging strategies are not reflected or aided well by existing DBMSs [11]. Several notable efforts towards the development of DBMS tools for pedagogical purposes have been well received. The XData system uses Cosette and query equivalence to find the minimal sequence of edits of an incorrect student query, such that it can be transformed to a query that is equivalent to a correct query [9]. The nature of their work closely resembles our autograder, but while their paper uses the system to assign partial marks to incorrect queries, our system is more geared towards providing hints that would guide the student towards the correct query.

The latest paper with work most similar to ours is the RATesT system [4]. Their work also focuses on checking query equivalence between two queries. However, unlike our autograder which presents a series of human readable hints to the user to indicate where the error might be, the RATesT system finds a smaller, more concise counterexample to demonstrate query in-equivalence. RATesT's evaluation also includes tests based on student usage to validate its pedagogical efficacy. The RATesT publication indicates that the idea of applying query equivalence to a pedagogical setting has value. Building user friendly tools to help programmers learn and debug database queries is an interesting and immediately practical research direction. Admittedly, building a tool that encompasses the full range of SQL functionality is a challenging problem. The main distinction between our autograder and RATesT is that we aim to provide more actionable feedback.

III. Grading Overview

To grade student submissions the instructor must first create a solution file called *solution.py*. This file contains a solution dictionary which has fields for the tables, their schemas, and the solution queries. The instructor can also specify point values for the questions. The schema syntax matches what Cosette uses and is easy to read and write. An example of the content of the file is below:

```
solution = {
    "schemas": [
        "schema s1(a:int, b:int);",
        "schema s2(s:str);"
    ],
    "tables": [
        "table r(s1);",
        "table tbl(s2);"
    ],
    "queries": {
        "q1": '''query q1
                 `select distinct x.a as a
                  from r x`;''',
        "q2": '''query q2
                 `select distinct x.s as s
                  from tbl x`;'''
    }
}
```

Every student will submit one submission file that is a valid SQL file. For every question, the student must create a view that has the name as the question name and their answer is the query that defines the view. An example submission file is:

```
create view q1 as
select x.a as a
from r x;

create view q2 as
select distinct x.s as s
from tbl x;
```

The instructor can then put all of these submission files into a folder and run the autograder. The autograder will produce a JSON file that looks like the following:

```
{
    "john_yang": {
     "q1": "CORRECT",
```

```
    "q2": "CORRECT",
    "earned_points": 2,
    "possible_points": 2
  },
  "brian_deleonardis": {
    "q1": "INCORRECT - HINT: Your query
    returned too many rows for at least
    some dataset. Conditions that may
    be missing: [r.a > 2]\n",
    "q2": "CORRECT",
    "earned_points": 1,
    "possible_points": 2
  }
}
```

There are various command-line options available to the instructor (i.e. include specific hints in the output, name of submissions folder, name of submissions file, etc.). When the autograder is run, it loops through the files in the submission folder. For every file it extracts the queries from the file, it then constructs a valid Cosette program (which includes the queries and schema info) for every question using the schemas, tables, and queries from the solution.py file and the student query from their submission file. It then sends the program to the Cosette web API and uses the result to create the output JSON object.

It is worth noting that Cosette doesn't guarantee that it will find an answer for every pair of queries that you provide it because the problem of query equivalence is NP-Complete [12]. Occasionally Cosette will return an error saying that it can't solve the problem. We were not able to find two queries where this actually happened so it is likely that this either rarely happens or occurs only for some particular query structure cases. Still, in order to use the autograder in production, a feature would need to be added where a more traditional autograder is used as a backup if Cosette cannot figure out the answer. That being said, the query comparisons necessary for most database courses are expected to be rather limited in structure variety and thus this edge case is unlikely to be a significant issue.

IV. Hint Generation Overview

Unlike the Cosette solver, our autograder doesn't compile submitted queries into logic formulas to run constraint solvers against [1]. Due to the specificity of the intended usage - that is, to generate hints for student-submitted SQL queries against staff solutions - we prioritized runtime efficiency and implementation simplicity over in-depth comparison computations. This limits the overall accuracy of the hint generator, but for our purposes, we would rather be able to quickly generate hints that are occasionally wrong rather than have hint generation take a long time to execute. Instead, our approach simply analyzes the different clauses in the queries, and flags differences as potential hints.

Currently, the autograder supports checks for SQL SELECT, FROM, WHERE, DISTINCT, GROUP BY, ORDER BY, and LIMIT clauses. These checks are largely based off of the ones provided by the Cosette solver, with some additional ones that are not yet supported by Cosette such as ORDER BY and LIMIT.

In order to encourage student learning, our autograder only generates one hint per submission. For example, if the student's incorrect query has multiple issues (e.g. wrong SELECT columns, wrong WHERE conditions, etc), the autograder will only return a hint pertaining to a single incorrect SQL clause. This is to discourage students from using the autograder as a crutch, pushing students to work towards the solution themselves and think about why the change(s) need to be made. In order to meet this expected goal, the number of submissions per student should also be limited to a certain predetermined number of attempts.

Note that our autograder is currently set up to give hints in the logical SQL query processing order (i.e. FROM table hints before SELECT column hints). However, in order to make the autograder more flexible, we also implemented a feature that allows the user to adjust the hint order. This comes in the form of an additional command-line argument passed to the autograder program. For instance, given that a student query has incorrect WHERE and GROUP BY conditions, our autograder returns a hint for the WHERE clause by default. However, if the order option is enabled, the student can request a hint for the GROUP BY clause instead. This allows students to have more control over the guidance that our autograder provides, given they already have some idea of what might be wrong with their query. In the case that there is nothing incorrect about the specified clause, another available hint will be provided. Again, in order to prevent abuse of the query hints, limiting retry attempts is recommended. In the performance evaluation section, we discuss different kinds of tests that would help us validate whether the above configuration is the right approach.

V. Hint Generation Architecture Overview

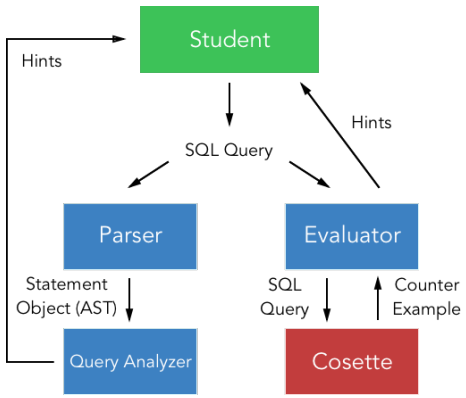

Fig. 1. Autograder Architecture

The hint generation system is made of three components - the evaluator, the parser, and the query analyzer. The evaluator uses the counterexample that Cosette generates to give some

initial feedback on the query to the student. The parser takes the query in as a string and constructs an abstract syntax tree (AST) out of it. The query analyzer compares the ASTs of the student and correct queries to attempt to determine what is causing the problem that the evaluator discovered.

VI. Implementation: Evaluator

In the evaluator, we use the counterexample that Cosette returns to us to see what the problem with the query actually is. We use a SQLite database to run both queries and then compare the output. The first step is to create the tables in the database using CREATE TABLE statements. We construct these statements from the schemas that the instructor provided. Next, we add the counterexample to the SQLite database using INSERT statements. Finally, we run each query and examine the results. From this we can tell if the number of rows is wrong, if the number of columns is wrong, or if the content of the rows is wrong. For example, if a query is missing a WHERE condition the evaluator would see that the student query returns too many rows and will therefore add the phrase "Your query returned too many rows for at least some dataset" to the hint returned to the student.

The evaluator works excellently for some types of queries, but has some flaws for other types of queries. For example, if the student is missing a WHERE condition, the only possible error is that there are too many rows. In this case, the evaluator will always properly identify the problem. However, if a student includes a WHERE condition but gets the actual condition wrong, then the type of error could be very different. Depending on the dataset, the student query may return too many rows, too few rows, or the correct number but incorrect rows. The evaluator will only ever return one of these three possibilities because Cosette only returns one counterexample. This could be potentially remedied if Cosette returned multiple counterexamples instead of just one. It would be ideal if Cosette was smart enough to identify the different types of failure cases and return one error for each type, but even if it generated several random counterexamples, there is a decent chance that our evaluator would be able to find the different types of errors.

VII. Implementation: Parser

To make comparing the different clauses of the query easier, we first send each query through the parser which converts the text query into an AST. The AST is rooted in the Statement class which then has attributes for the different clauses of the tree. Each clause of the tree holds a list of the expressions that make up the clause (for example, the FROM clause has a list of table names that are being selected from). The different expressions also implement an equals() method that makes comparing expressions from both queries easy.

The process for doing this is as follows. First we use a third party library module called "pglast" to convert the textual query to an AST. The AST this library generates, however, is a series of nested dictionaries that is very difficult to work with. For this reason we constructed our own AST (described above) and converted these nested dictionaries to our AST. We extract the relevant clauses out of the AST returned by pglast and feed them into the Statement class, which is forwarded to the query analyzer to perform hint generation. We go into more details on the parser in the appendix, but since there isn't too much innovation in these sections, the reader can feel free to continue on to section VIII about the Query Analyzer.

VIII. Implementation: Query Analyzer

The query analyzer uses the output of the parser to compare the student and the correct query. For each clause (i.e. SELECT, FROM, WHERE) the analyzer examines that clause for each query. If there is any difference, a corresponding hint is generated. We keep track of hints for every clause, but only return one of them to the student so that the student must still work to figure out the correct query. The hint provided is either determined from our default ordering (the logical execution ordering of the clauses of a SQL query) or chosen from the user-provided ordering of the clauses.

A. Query Analyzer - FROM Clause

The query analyzer examines the FROM clause by checking to ensure that both the student and the correct query are getting data from the same tables. If the student includes an unnecessary table or does not include a required table, the analyzer adds that information to the hint. For example, consider the following two queries:

```
Correct:
SELECT a
FROM r , t

Student:
SELECT a
FROM r
```

The analyzer will add the following phrase to the hint: "Tables that possibly should be joined: [t]." Note that it is possible for the analyzer to give bad hints with this technique. For example, if a student joins a table that does not need to be joined, but this join does not affect the output (maybe because of a join condition that the student uses), the analyzer will still flag it for the hint. It is very rare for a student to write a query with this exact problem though, so we tolerate the occasional mistake. Additionally, the hints may in fact aid the student in improving the efficiency of their query even if their submitted version is technically "correct".

B. Query Analyzer - SELECT Clause

We evaluate the SELECT clause very similarly to how we implemented the FROM clause. We compare each column being selected in the student query to the columns selected in the correct query. If any columns are not in the correct

query, or if any columns from the correct query are not in the student query, we add that to the hint. Two column selections are equivalent if the expression generating the column is the same and the name of the column (either from the alias or implicitly is the same). For example, consider the following two queries:

```
Correct:
SELECT x.a, x.b
FROM r x;

Student:
SELECT x.a
FROM r x;
```

The analyzer will add: "Select clauses that may be missing: [r.b]" to the hint. We use the table name in the hint as opposed to the alias name because it is more descriptive.

Evaluating if expressions are the same is tricky, but we only supported the expressions that Cosette supports (which are column references, SUM and COUNT aggregates, and boolean expressions involving =, <, >).

For two column references to be equivalent, both the column name and the table they are selected from must be the same. Determining if the table they are selected from is the same is tricky because the tables might be aliased differently in different queries. To fix this, during the parsing stage we keep track of an alias map which maps the alias name to the table it comes from. Instead of storing the alias name in the column reference AST node, we instead store the table name itself. If the query does not preface the column with a table/alias name, then the column cannot be ambiguous (we assume we are provided queries that will not error) so the column reference must be the same.

For aggregate expressions, the function name must be the same and the expression passed in as a parameter must also be the same.

For $<$ or $>$, either the left expression, the operator, and the right expression must all be the same or the expressions are flipped and the opposite operator is used. For example: a $>$ b is equivalent to b $<$ a. For =, the operator (=) must be the same and the left and right expressions must both be present (but the order they appear in does not matter).

C. Query Analyzer - WHERE Clause

The WHERE clause is also evaluated very similarly to the SELECT and FROM clauses. We iterate through the conditions of both queries and flag missing/extra conditions. The WHERE clause, however, has additional complications because conditions are not simply comma separated. The way the conditions are conjoined (i.e. AND, OR, and using parentheses for order of operations) is significant.

For some types of errors, it is not easy for even humans to generate a precise hint. For example, consider the conditions:

```
a and b or c
vs
c and a or b
```

How would you even describe the problem? The authors struggled with this question so we decided on a simple scheme. We compare the individual conditions just like we do in evaluating the SELECT clause flagging any missing or extra conditions. Then we count the number of ANDs and ORs in each query. If the number of occurrences are different, then it is likely that the student made a mistake in conjoining the conditions. As an example, consider the following two queries:

```
Correct:
SELECT x.a
FROM r x
WHERE x.a > 5 AND x.a < 10

Student:
SELECT x.a
FROM r x
WHERE x.a > 5 OR x.a < 10
```

The only error in the student query is joining conditions with an OR rather than an AND. The query analyzer recognizes this and adds to the hint: "Some of your ORs possibly should be ANDs."

Finally, if the conditions are correct, the AND/OR counts are correct, the order of operations in the WHERE clause is different than the instructor query, and we find no other problems in the query, we add that the the student should "Check the order of operations in your WHERE clause." Here is an example of when this would happen:

```
Correct:
SELECT x.a
FROM r x
WHERE x.a > 5 AND x.a < 10 OR x.b > 20

Student:
SELECT x.a
FROM r x
WHERE x.a > 5 OR x.a < 10 AND x.b > 20
```

While our scheme for giving hints for errors with conjoining the conditions is not complicated, we have found that it works very well in practice for simple queries. For the other clauses, the only errors in the types of hints we throw are ones in which the clause is correct, but the query analyzer thinks there is a mistake. The query analyzer never misses a mistake for the other clauses - it only occasionally throws false positive errors. Because of this, the only time that the "Check the order of operations in your WHERE clause" can be thrown is when it is the only possible source of error in the student's query.

It is worth noting that when the student takes a different approach to the WHERE clause then the instructor's query does, this approach breaks down. As a simple example, let's say the instructor's WHERE clause is $x < 5$ and the student's condition is $x < 5$ and $x < 7$, the $x < 7$ condition would be flagged in the hint even though it does not affect the correctness of the query. For another example, suppose we have an integer column a. If the instructor's WHERE clause is $a <= 20$ and the student clause is $a < 21$, they are effectively the same thing but the query analyzer will not realize this. This is a theme for the entire hint generator - when the student takes a similar approach to the instructor the hint generator does a great job, but when the student takes a different approach the generator breaks down. Thus, the performance of our autograder is greatly aided by course instructors providing multiple examples of correct queries so that our autograder can compare student queries to the most similar-in-structure correct queries.

D. Query Analyzer - DISTINCT, GROUP BY, HAVING, ORDER BY, and LIMIT Clauses

All of these clause implementations are either relatively straightforward or similar to the previously discussed clauses. GROUP BY and ORDER BY are implemented similarly to FROM. HAVING is implemented like WHERE. For LIMIT, we simply check if both queries have the same value for it if present. For DISTINCT, we check that both queries either have or do not have it.

IX. Performance Evaluation

Unfortunately, we were not able to do benchmarking on a real autograder (our old friends on the CS W186 course staff were not the most accommodating). However, we know that the bottleneck in SQL query autograders is running the student query on the test data set. That means we can compare the time it takes to run queries to the time it takes Cosette to decide if queries are equivalent in order to get a good idea for how our autograder's performance will compare to a traditional autograder.

To experiment, we ran 3 different SQL queries on Postgres and compared that to running Cosette programs. For each query, we created a Cosette program comparing the query to itself and a program comparing the query to a slightly modified version to see if there was any difference in the performance for equivalent queries and non-equivalent queries.

To time the Postgres queries, we used \timing. To time the Cosette programs, we used python's time.time to take the system time immediately before and after making the API request and then subtracted the two values. All experiments were made on a 2018 Lenovo Thinkpad E540 with an i5 processor and 8GB of RAM. The queries were selected from the first CS W186 class project and were run on the 2017 Lahman baseball database. We do not know what kind of server Cosette is running on. Each query and Cosette round was run 10 times and the average is displayed in the table in figure 1. To see the exact queries we ran, see the appendix.

Obviously, this is not the result that we were hoping for. Cosette performed about 500x worse than it takes Postgres to run a student query so it seems impossible to build an autograder that is competitive performance-wise. There are, however, still a variety of bottlenecks that may be limiting the execution speed of Cosette in the tests. First, our existing implementation makes calls to the Cosette web API, instead of a locally compiled version. Packaging and sending the API request and then unwrapping the return value adds a good amount of latency to the request. We estimated that this only takes about 500ms, however, by sending the Cosette API a bad request that returns an error immediately. Having a local executable that runs Cosette would resolve this issue.

Even if we had an executable running locally, it is hard to believe that Cosette in its current form would be competitive with a traditional autograder given how far off the execution times are. One way Cosette could potentially fix this problem for us is by providing a batching feature that is more efficient. Cosette only needs to analyze the instructor provided query once, so there is no reason to repeat that process every time. The authors wonder if there is some way in which Cosette can compare several queries to the instructor query at once, which would dramatically improve the autograder's runtime.

Because Cosette in its current form is not really practical for an autograder, we did not take the necessary steps to actually deploy it to production. We think that some additional infrastructure will be necessary to make the autograder useful in a classroom setting. One performance study that we could do to support this claim is see how long it takes for the autograder to process different types of SQL queries. We anticipate that queries involving more expensive operations, such as UNION ALL and subqueries, might take longer to prove equivalence. Knowing the absolute numbers would help us determine how to best deploy the autograder for usage at the scale of multiple requests per second. It's very likely that a single running instance of the autograder would still be too slow to handle requests at more than one per second, in which case looking into Docker and writing a Docker compose script with scaling would be the best immediate solution.

A. Hint Generation Evaluation

Another area of evaluation we would've liked to explore would be concerning the usage of the hint generation. This type of evaluation would deviate from the traditional database performance metrics, and would be more concerned with social factors that attempt to answer how helpful the autograder actually is. The performance benchmarks for these kinds of evaluations would tap into the related work and research into existing systems that assist in writing SQL queries. To answer this question, A/B testing and student feedback on the autograder would be the primary tools and metrics of evaluation. Our original plan involved running these kinds of evaluations during mid April, where we would ask randomly-selected CS W186 students to test the hint generation. These plans unfortunately fell through due to the COVID-19 situation.

figure 1

Query	Description	Postgres Runtime(ms)	Cosette Equivalent Runtime(ms)	Cosette Error Runtime(ms)
1	Basic select, from, where	16.742	8243	8101
2	Uses group by, having	26.452	8320	9415
3	Has a join	15.124	8050	8212

We're primarily interested in testing how useful the autograder hints are to students. A basic set of tests would involve taking two simple random sample groups of students. Both groups would be presented with the same database and asked to write queries against Postgres. One group would have the autograder hints to work with while the other group would not. We'd then time how long it takes for each individual in the group to complete the queries.

This basic test could be augmented and designed more thoroughly to yield different kinds of metrics. First, the types of questions we ask students to write can test how effective each hint is for each SQL feature. In practice, we would write questions that exercise a single SQL feature. The distribution of time spent until the correct query is achieved, in addition to student feedback, could provide insight into the correctness and specificity of the autograder hints for different kinds of mistakes. To ensure the hints are teaching the student in the process, and not just helping them get the right answer, we would then need to test both groups of students without the autograder to ensure the group with hints actually learned from the process of writing the query at least as well as the group that did not receive a hint.

We also want to evaluate whether or not our decision to only provide one hint (as opposed to a hint for every mistake we found) is a good one. It is our theory that providing all the hints will help a student write the correct query faster, but they will learn less from doing it. We can validate this theory by having students write queries against two versions of the autograder: one that produces a hint per run, and another that outputs an exhaustive list of hints for all errors. It's probable that students armed with the exhaustive list of hints would complete the tasks faster. However, to test the pedagogical efficacy of the two approaches, we would again ask students to write queries without the assistance of the hints to determine how much they actually learned.

An interesting extension concerning how hints should be presented, that we believe is worth looking into, is, if a query has multiple errors, would the order of hints affect the effectiveness or usability of the autograder? For instance, if a student's query has errors in their HAVING, WHERE, and GROUP BY clauses, would the order in which the hints for each error are presented affect how quickly a student arrives at the correct query? If it does, what factors affect how hints are prioritized? The following example illustrate a potential case study:

```
Correct:
SELECT a, b, c FROM r
WHERE a < 3
GROUP BY a
HAVING COUNT(a) > 25;

Student:
SELECT a, b, c FROM r
WHERE a > 3
GROUP BY b
HAVING COUNT(b) < 25;
```

From the example above, it arguably makes more sense to present the hint for the GROUP BY error before the hint for the HAVING error, because determining the groups, then the conditions on each group, might be the way we'd want a student to think about the relationship between these clauses. If different orders of hints do yield discernible differences, then there's the problem of figuring out why, and whether that influences how hints should be presented to maximize its pedagogical productivity. Our guess is that the "why" could be a mix of technical and social factors, ranging from the complexity of the SQL query to the students' responses to hints. From our current standpoint, this seems like an interesting and worthwhile, yet very open ended question that we are not quite sure how to frame and measure.

B. Hint Generation Discussion

Even without the formal experimentation, the authors have a pretty good idea on how well the hint generation performs. For student queries that take a similar approach to the instructor provided query, the hint generator does very well. It is really good at finding small construct errors (such as an incorrect WHERE condition, a missing SELECT, etc). Because of SQL's declarative nature, we believe that most student queries (at least for an intro databases class) will follow the same approach as the instructor query. The techniques we used in this paper would certainly not work for a standard imperative language, but because SQL is declarative there is often one obvious way to approach simple queries.

However, the hint generator will be very bad when the student takes an alternative approach to the query. This hopefully only happens with complex queries and not so much with the more straight-forward queries typical of most database classes. For example, if the instructor query uses a subquery but the student uses UNION ALL, the hint generator will not have any idea about what is wrong and what is right about the student query.

One potential solution to the alternative approach problem is for the instructor to provide multiple correct queries (one for every common valid approach). Then, the autograder can use some form of code similarity to determine which correct query is syntactically closest to the student's query. The hints corresponding to the closest correct query would then be offered to the student. For SQL query similarity, we could examine methods for clustering similar SQL queries [2]. Testing this would be quite straightforward, and our hope is that an autograder referencing multiple correct queries would lead to an even greater reduction in the amount of time a student takes to write a correct query for complex SQL questions.

X. Cosette Limitations

We've discussed some limitations specific to our current autograder implementation in the previous sections, but have yet to address the limitations inherited from building our autograder on top of Cosette. For instance, in order to support SQL constructs in its K-relation interpretations, Cosette uses a complex semiring in the form of Homotopy Type Theory (HoTT) [13]. The limitations of this approach are that HoTT is still a developing area of research itself and that foreign keys (currently unsupported by Cosette) are difficult to model using the HoTT library in Coq [14] Thus, in order to overcome these inherited limitations, our autograder will need to include workarounds, possibly by incorporating other methods of query equivalence determination for query structures that aren't supported by Cosette. There are several alternative methods currently under research, such as unbounded semiring (U-semiring) and satisfiability modulo theories [14], [15], with no conclusive "best" approach given different advantages and trade-offs. Thus, if the goal is to improve the determinability of query equivalence in our autograder, or improve runtime efficiency, it may be worthwhile to experiment with a hybrid of approaches.

Furthermore, as discussed previously in the paper, some interactions between our autograder and Cosette could be improved as with the case that Cosette provides only a single counterexample per query comparison call. Again, overcoming these limitations could involve coming up with alternative solutions. Ultimately, the usage of Cossette in our autograder is positive in that it demonstrates the practical potential of Cosette's evaluation approach, but also potentially an issue in that the reliance on Cosette could possibly limit future development on our autograder.

XI. Future Work

Some SQL clauses that are currently a work-in-progress in our autograder implementation include UNION ALL and subqueries. Unfortunately, we were unable to fully include these clauses in our autograder functionality by the time of paper submission. These, as well as additional features not yet supported by the Cosette solver such as different join types, are all left as future improvements to the autograder.

Currently, our autograder is not compatible with Windows systems due to the usage of certain modules, such as pglast, which are currently unsupported by Windows. Some future work on the autograder could include adding Windows support - possibly by modifying the project to remove reliance on outside modules. This step might be a bit complicated in that we would have to find a suitable replacement for constructing the abstract syntax tree. While our code currently uses a python module, the Scala combinator classes could be worth looking into for a ground up parser for creating an AST representation of the code, although this sounds like we'd be reinventing the wheel.

Lastly, again, it is important to test our autograder on real query sets. If we could run our autograder for actual classes (e.g. DS 100, CS W186), we would get a lot of useful feedback on how our autograder performs, as well as on how useful the generated hints are to students. Ultimately, our autograder is intended to be a more performance-efficient, practical application of Cosette and it's thus important to benchmark it on real-world usage, as discussed in the evaluation section. We think there is a large variety of tests we could use to quantify the technical and pedagogical effectiveness of the autograder.

XII. Conclusion

Database query equivalence has a wide range of applications. In this project, we have laid the groundwork for one of its more practical usages for pedagogical purposes, building an application that provides hints about SQL query assignments on top of the prover. As of the submission of this paper, we'd like to think that our application is ready to be tested and is capable of presenting hints for a basic but popular set of SQL features.

As we continue to develop each of the three components that constitutes the hint generation architecture, our goal is to add more features and more fine-grained hints for each feature. There's much more performance evaluation work to be done, and we believe our tests will help us quantify how useful this application and how much additional engineering would be needed to make this application deliverable so that it can respond to multiple requests per second. Behavior-oriented A/B testing will also help us make key decisions regarding how the autograder should present hints and how it might evolve to better service increasingly complex SQL questions with potentially multiple correct answers.

Acknowledgment

Thanks to Professor Alvin Cheung for helping us develop the initial project idea. Thanks to Professor Joe Hellerstein for providing valuable feedback throughout the course of our project.

References

[1] S. Chu, D. Li, C. Wang, A. Cheung, and D. Suciu. Demonstration of the cosette automated sql prover. In Proceedings of the 2017 International Conference on Management of Data, pages 1591–1594. ACM, 2017.

[2] G. Kul, D. T. A. Luong, T. Xie, V. Chandola, O. Kennedy and S. Upadhyaya, "Similarity Metrics for SQL Query Clustering," in IEEE Transactions on Knowledge and Data Engineering, vol. 30, no. 12, pp. 2408-2420, 1 Dec. 2018, doi: 10.1109/TKDE.2018.2831214.
[3] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, Dong Xu. Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. PVLDB, 12(11): 1276 - 1288, 2019. DOI: https://doi.org/10.14778/3342263.3342267
[4] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. In Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19). Association for Computing Machinery, New York, NY, USA, 503–520. DOI:https://doi.org/10.1145/3299869.3319866
[5] Chu, Shumo. Automated Reasoning of Database Queries. Diss. 2019.
[6] Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S. Lam. 2019. Genie: a generator of natural language semantic parsers for virtual assistant commands. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019). Association for Computing Machinery, New York, NY, USA, 394–410. DOI:https://doi.org/10.1145/3314221.3314594
[7] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2017. Verifying equivalence of database-driven applications. Proc. ACM Program. Lang. 2, POPL, Article 56 (January 2018), 29 pages. DOI:https://doi.org/10.1145/3158144
[8] Grossman, Shelly, et al. "Verifying equivalence of spark programs." International Conference on Computer Aided Verification. Springer, Cham, 2017.
[9] Chandra, Bikash, et al. "Automated Grading of SQL Queries." 2019 IEEE 35th International Conference on Data Engineering (ICDE). IEEE, 2019.
[10] Bodik, Chenglong Wang Alvin Cheung Rastislav. "Synthesizing Highly Expressive SQL Queries from Input-Output Examples."
[11] Gathani, Sneha, Peter Lim, and Leilani Battle. "Debugging Database Queries: A Survey of Tools, Techniques, and Users." Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. 2020.
[12] Alto, A. V., Sagiv, Y., and Ullman, J. D., Equivalence Among Relational Expressions, SLAM J. Computing, 8(2):218-246 (May 1979).
[13] The Univalent Foundations Program. Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.
[14] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. PVLDB 11, 11 (2018), 1482–1495.
[15] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, Dong Xu. Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. PVLDB, 12(11): 1276 - 1288, 2019.

XIII. Appendix

A. Example Hints

```
Correct:
select SUM(x.a)
from r x
```

```
Student:
select COUNT(x.a)
from r x
```

Hint:
Select clauses that may be wrong: [count(r.a)]
Select clauses that may be missing: [sum(r.a)]

```
Correct:
select y.a
from r y
where y.a > 1
```

```
Student:
select x.a
from r x
```

Hint:
Your query returned too many rows for at least some dataset.
Conditions that may be missing: [y.a > 1]

```
Correct:
select COUNT(*)
from r x
group by x.a
```

```
Student:
select COUNT(*)
from r x
group by x.b
```

Hint:
The following columns may not need to be in the GROUP BY: [r.b]
The following columns are potentially missing from the GROUP BY: [r.a]

```
Correct:
select x.a as a, x.b as b
from r x
```

```
Student:
select x.a as a
from r x
```

Hint:
Your query is missing 1 columns.
Select clauses that may be missing: [b: r.b]

```
Correct:
select x.a as a
from r x
where x.a = 2
```

```
Student:
select x.a as a
from r x
where x.a = 3
```

Hint:
Your query returned too many rows for at least some dataset.
Conditions that may not be necessary: [r.a = 3]
Conditions that may be missing: [r.a = 2]

To see a full list of examples, see our test file: https://github.com/bdeleonardis1/286proj/blob/master/test/test_hints.py

B. Parser Details

1) Parser - Implemented Features: We used the pglast (PostgreSQL Language AST) library to generate the AST. Passing in a SQL query to the `parse_sql` method returns a parse tree in a nested python dictionary format, wrapped in a Node class defined by the pglast library that makes traversal easier. The Node class also performs its own analysis of

the AST and exposes several fields that allows us to extract features without having to traverse the AST. In general, we can perform search traversals on the tree to determine the contents of the clauses that form the query. Each SQL feature and clause that we support has a corresponding extraction method that we've written as part of the parser. Right now, our parser has implemented basic support for the SELECT, FROM, WHERE, DISTINCT, GROUP BY, HAVING, ORDER BY, and LIMIT clauses.

2) Parser - WHERE and HAVING Clauses: While the implementations for extracting the SELECT, FROM, DISTINCT, ORDER BY, and LIMIT clauses were quite straightforward, in the sense that the columns and tables used in the query were stored as lists in the pglast Node class representation of the AST, the WHERE and HAVING clauses required a more recursive implementation due to the nature of multiple conditions being joined with ANDs and ORs. Our implementation is as follows:

```
def get_where_clauses(whereClause):
    where_info = WhereClause([], 0, 0)
    if whereClause == Missing:
        return where_info
    if whereClause.node_tag == 'BoolExpr':
        for expr in whereClause.args:
            # Recursive call
            temp_where_clause =
                get_where_clauses(expr)
            <code adding each condition
                to where_info>
    elif whereClause.node_tag == 'A_Expr':
        # Base Case: Reached condition
        # No more and/or's
        where_info.conditions
            .append(whereClause)
    return where_info
```

While this implementation works for the time being, to implement more complex features such as subqueries as discussed in the following section, we would have to include clauses to identify and handle different possible node tags.

3) Parser - Future Features: As we begin adding hint generation support for new features such as UNION ALL and subqueries, we would have to write new corresponding extraction methods within the parser. For more complicated features that involve multiple sets of tables, particularly subqueries, we expect to implement a Statement class with a more recursive structure. Implementing hints for subqueries has been particularly tricky for several reasons. First and foremost, subqueries can occur in SELECT, FROM, or WHERE clauses. Implementing subqueries would require rewriting these features' to be able to point at a `Statement` class object. The second difficulty is the ambiguity that might arise if there are similar errors in multiple, different subqueries within the same query. Consider the following example query reading from a CUSTOMERS database with a SALARY column:

```
Correct:
SELECT * FROM CUSTOMERS
    WHERE SALARY > 3000
    ID IN (SELECT ID FROM CUSTOMERS
    WHERE SALARY > 4500)

Student:
SELECT * FROM CUSTOMERS
    WHERE SALARY < 3000
    ID IN (SELECT ID FROM CUSTOMERS
    WHERE SALARY < 4500)
```

In the above query, the features used in the inner query (SELECT, FROM, WHERE) are not only present, but identical conditions-wise to the features in the outer query. The errors made by the student for both where conditions are the same. In this case, it'd be important to make sure that the hints regarding the WHERE clause errors in the inner and outer queries are differentiated when presented to the user. Developing aliases to indicate which hint is for which sub-query is a feature we have not yet figured out.

XIV. Performance Analysis Queries

Query 1:

```
SELECT p.namefirst, p.namelast, p.birthyear
FROM people p
WHERE p.weight > 300;
```

Modified Query 1:

```
SELECT p.namefirst, p.namelast, p.birthyear
FROM people p
WHERE p.weight > 200;
```

Query 2:

```
SELECT p.birthyear,  COUNT(*)
FROM people p
GROUP BY p.birthyear
HAVING COUNT(*) > 100;
```

Modified Query 2:

```
SELECT p.birthyear,  COUNT(*)
FROM people p
GROUP BY p.birthyear
HAVING COUNT(*) > 200
```

Query 3:

```
SELECT m.namefirst, m.namelast, h.playerid,
    h.yearid
FROM halloffame h, people m
WHERE h.playerid = m.playerid
AND h.inducted = 'Y'
```

Modified Query 3:

```
SELECT m.namefirst, m.namelast, h.playerid,
    h.yearid
FROM halloffame h, people m
WHERE h.playerid = m.playerid
AND h.inducted = 'N'
```