Cosette: An Automated Prover for SQL

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung
University of Washington

ABSTRACT

Deciding query equivalence is an important problem in data management with many practical applications. Solving the problem, however, is not an easy task. While there has been a lot of work done in the database research community in reasoning about the semantic equivalence of SQL queries, prior work mainly focuses on theoretical limitations. In this paper, we present COSETTE, a fully automated prover that can determine the equivalence of SQL queries. COSETTE leverages recent advances in both automated constraint solving and interactive theorem proving, and returns a counterexample (in terms of input relations) if two queries are not equivalent, or a proof of equivalence otherwise. Although the problem of determining equivalence for arbitrary SQL queries is undecidable, our experiments show that COSETTE can determine the equivalences of a wide range of queries that arise in practice, including conjunctive queries, correlated queries, queries with outer joins, and queries with aggregates. Using COSETTE, we have also proved the validity of magic set rewrites, and confirmed various real-world query rewrite errors, including the famous `COUNT` bug. We are unaware of any prior tool that can automatically determine the equivalences of a broad range of queries as COSETTE, and believe that our tool represents a major step towards building provably-correct query optimizers for real-world database systems.

1. INTRODUCTION

Given two queries Q_1 and Q_2, the query equivalence problem asks whether Q_1 and Q_2 are *semantically equivalent*, i.e., they always return the same results when executed on any input database instance. This problem has many real-world applications in data management. For instance, all query optimizers contain a plan generator that enumerates plans during query optimization [13], and the enumerated plans should be semantically equivalent to the input query. This applies to compilers for integrated query and application languages as well [25, 14]. Determining query equivalences is also important in generating test cases for database implementations [29], building teaching tools for developers [21], and auto-grading student assignments [11].

While the problem has attracted much attention from the database theory research community, prior work has focused mostly on the theoretical limitations in solving the problem [31, 27, 20]. We are unaware of any practical solvers that can determine query equivalences, and this has unfortunately contributed to buggy database implementations [17, 1, 3], and many widely deployed query rewrites techniques like magic set rewrites [28] are left unverified.

In this paper, we present COSETTE, a solver that can determine whether two SQL queries are semantically equivalent. COSETTE builds on top of the recent advances in the formal methods research community. In particular, COSETTE leverages the strengths of two active branches of research from that community: symbolic execution and constraint solving, and interactive proving. Given a logic formula containing symbolic (i.e., unknown) variables, constraint solvers are developed with various specialized heuristics [16, 26] that can efficiently find *models*, i.e., values for the symbolic variables, that make the formula true, or return unsatisfiable otherwise. For instance, given the Boolean logic formula `v1 && (v2 || v3) == (v1 && v2) || (v1 && v3)`, a constraint solver might return the model `{v1:True, v2:True, v3:False}`. Users typically use this model-finding property of constraint solvers to show the *falsity* of logic formulas: if the solver can find a model (also called a counterexample in this case) for the negated formula, then the original formula must be false. This strategy has been applied in numerous real-world scenarios (e.g., [34, 12]). In fact, solvers are now available for many different domains [7, 6], and there are annual competitions for the most efficient solvers as well.

Interactive proof assistants [4, 5], on the other hand, do not come with heuristics for finding models. They instead allow developers to provide a *proof script* to derive the validity of logic formulas in a step-by-step manner. For example, to prove the above Boolean formula to be true for all possible values of `v1`, `v2`, and `v3`, one can apply the distributivity property of logical conjunction to the left side of the formula, and then check that the two sides are subsequently equal syntactically. This two step strategy can be encoded in a proof script to be executed by the proof assistants, and can furthermore be programmed into *proof tactics* that the proof assistant can apply when similar formulas are subsequently encountered. Unlike constraint solvers, proof assistants are efficient in searching for proofs, making them useful tools to demonstrate the *validity* of formulas. As such, they have been used recently to prove the correctness of many software systems [22, 19].

COSETTE combines the strengths of constraint solvers and proof assistants to determine the equivalence of SQL queries. It consists of two components: a compiler that translates the input SQL queries into logic formulas, and subsequently uses a constraint solver to find counterexamples to show that the input queries are not equivalent; and a separate compiler translating queries into K-relations [18], and then uses a proof assistant to validate the equivalence of the two queries. The encoding of SQL to logic formulas allows us find counter examples of inequivalent SQL queries by constraint solvers. The encoding of SQL to K-Relations, which represents a relation as mathematical function that takes as input a tuple and returns its multiplicity in the relation, enables convenient machine checkable proofs of equivalent queries in a proof assistant. As we will see, this unique combination of proving techniques enables COSETTE to efficiently determine the equivalence SQL queries.

The query equivalence problem for arbitrary SQL queries is undecidable in general. Trakthenbrot's theorem [31, 23] states that

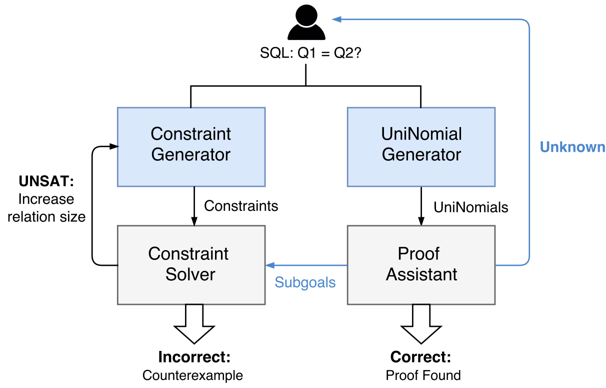

Figure 1: COSETTE architecture, where texts and arrows in blue indicate user interactions.

the problem *given an FO sentence φ, check if φ has a finite model* is undecidable. We can reduce this problem to query equivalence by defining Q_1 to be a query that checks checks φ and returns the empty set if φ is false, or returns some non-empty set if φ is true, and defining Q_2 to be the query that always returns the empty set (as above), then checking $Q_1 \equiv Q_2$). So an automated proof system for SQL will never be complete. Our experiments show that COSETTE can already determine equivalence for a wide variety of real-world queries.

In summary, this paper makes the following contributions:

1. We develop COSETTE, a fully automated solver for SQL. COSETTE leverages proof assistants to identify semantically equivalent queries, and constraint solving to determine queries that are inequivalent. To our knowledge COSETTE is the first such solver for SQL queries.
2. COSETTE comes with a number of optimizations to make solving efficient. In particular, our data model enables generation of constraints that are easy to solve, and we also developed a number of proof tactics to speed up proof search. As we will explain, the design of COSETTE enables the two components in COSETTE to complement each other to improve solving efficiency. In addition, besides the fully automated mode, COSETTE also allows developers to interact with the tool as well.
3. We have implemented a prototype of COSETTE, and our experiments show that COSETTE can formally verify many well-known SQL rewrite rules, including those based on relational algebra, conjunctive queries, and magic set rewrites. In addition, it can disprove various query equivalences, including the well-known `COUNT` bug, along with other real-world optimizer bugs in Postgres and Oracle.

In the rest of this paper, we first describe the overall architecture of COSETTE in Section 2. Section 3 then describes how COSETTE translates the input queries into constraints, and Section 4 discusses the use of proof assistant in showing the validity of two queries, followed by its interactive aspects in Section 5. We report or experiment results using both textbook and real-world queries in Section 6, and discuss related work in Section 7.

2. OVERVIEW

Figure 1 shows the architecture of COSETTE. COSETTE takes in two SQL queries and returns either "equivalent," "inequivalent," or "unknown." Besides the input queries, users can provide the actual contents of the relations that the queries are executed on. If the contents of all involved relations are provided, then determining the equivalence of the two queries reduces to executing them and checking if their result sets are the same. Otherwise, for the *symbolic relations*, i.e., relations that are queried but whose contents are not provided, COSETTE assumes that they can range over any valid schemas and values. The goal of COSETTE then is to check whether the input queries are equivalent when executed on all possible relations. To infer the schemas for the symbolic relations, COSETTE scans the input queries for the attributes that are referenced. For example, if `Emp` is a symbolic relation, and one of the queries contains the predicate `Emp.age > 21`, then COSETTE will infer that `Emp` contains at least the integer attribute `age`. COSETTE assumes that symbolic relations with different names are distinct. Furthermore, predicates can be symbolic as well, meaning that they represent any Boolean functions that take tuples as input. As we will see in Section 4, this is useful for checking the equivalences of query rewrite rules that are part of query optimizers, where such rules are often expressed over arbitrary predicates.

COSETTE passes the input queries to the two compilation toolchain as discussed. On the one hand, the *constraints generator* translates the input queries into constraints. This involves bounding the size of each symbolic relation and determining its schema. The compiler then uses fresh symbolic variables to represent each of the tuples in the symbolic relations, and translates the semantics of the input queries into constraints over the symbolic variables. The generated constraints are sent to a constraints solver. If the solver returns a counterexample, then the input queries are proven to be inequivalent, and the counterexample is returned to the user.

On the other hand, if the constraint solver cannot find a counterexample, COSETTE will then forward the queries to the *uninomial generator*. The generator compiles the symbolic relations to K-relations, which are mathematical functions that return the multiplicity of a given tuple, and translates the queries into algebraic expressions over K-relations called UniNomials. The UniNomials are sent to the proof assistant to look for an equivalence proof. If the proof assistant fails to show their equivalence, it then interacts with the constraints solver and the user to solicit a proof, as we will explain in Section 5.

We next describe the constraints and uninomial generators in detail, followed by evaluations using COSETTE.

3. FINDING COUNTEREXAMPLES WITH CONSTRAINT SOLVER

In this section we describe how COSETTE translates input queries into constraints using symbolic execution, with the generated constraints sent to a constraints solver in search of counterexamples. If found, the input queries are proven to be inequivalent, and the counterexamples are returned to the user as evidence.

To illustrate this process, we use the queries shown in Figure 3 as a running example. The example illustrates the famous `COUNT` bug that involves rewriting of correlated subqueries. It involves two symbolic relations, `Parts` and `Supply`, with a derived view `Temp`. Using the architecture of the constraint generator shown in Figure 2, we explain the solving process in detail below.

3.1 Data Model

COSETTE models relations using bag semantics. Following prior modeling of relations [33, 24], in the constraints generator a tuple is encoded as a list of integers (strings are modeled as integers, and floating point numbers are currently not supported), and a relation is defined as an unordered set of `Pairs`:

```
Tuple     := List<Integer>
```

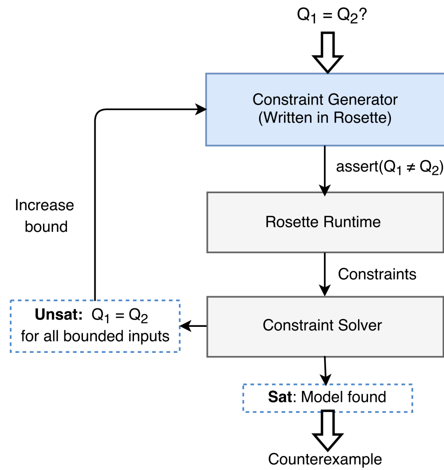

Figure 2: Architecture of the constraints generator its interaction with the underlying constraint solver.

```
SELECT pnum FROM Parts                                -- Q1
WHERE qoh = (SELECT COUNT(shipdate)
             FROM Supply
             WHERE Supply.pnum = Parts.pnum
             AND shipdate < 10);

WITH Temp AS                                          -- Q2
  SELECT pnum, COUNT(shipdate) AS ct
  FROM Supply
  WHERE shipdate < 10
  GROUP BY pnum
SELECT pnum FROM Parts, Temp
WHERE Parts.qoh = Temp.ct
AND Parts.pnum = Temp.pnum;
```

Figure 3: Constraints generation example using the `COUNT` bug.

```
Relation := List<Pair<Tuple, Integer>>
```

Here the first element in the pair is a tuple, and the second element represents the multiplicity of the tuple. For example, the list `[([1,2],`**`2`**`),([2,5],`**`3`**`)]` represents a relation with 2 tuples of `[1,2]` and 3 tuples of `[2,5]`.[1]

The constraints generator compiles each symbolic relation into a set of fixed size consisting of symbolic values. For example, the `Parts` symbolic relation in the running example is compiled to:

```
 Parts = [([sv0,sv1],sv2), ([sv3,sv4],sv5)]
Supply = [([sv6,sv7],sv8)]
```

where each $\texttt{sv}_i$ represents a symbolic value whose value will be assigned by the solver. Note that the multiplicity of each tuple is a symbolic value as well. In Section 3.4 we describe how COSETTE uses incremental solving to dynamically increase the size of each symbolic relation.

3.2 Compiling Queries to Constraint Programs

Given the data model, the symbolic execution engine in the constraints generator compiles a SQL query to a function written in Rosette that computes over symbolic relations. Rosette [30] is a language for constraints programming. When executed, the query function will generate constraints that can be sent to the constraints solver to solve. As an example, `Q2` in Figure 3 is compiled to the function below[2]:

```
def Q2():
  r = []
  for t in xprod(Parts, Temp):
    if p1(t) and p2(t):
      r.append([t[0]])
  return r
```

In the code fragment, `p1` represents the predicate `Parts.qoh = Temp`, `p2` is the predicate `Parts.pnum = Temp.pnum`, and `xprod` is the Cartesian product operator on two relations that we have implemented in Rosette. Just like its SQL counterpart, the constraint program iterates over each tuple from the Cartesian product of `Parts` and `Temp`, and appends the projection of the iterated tuple (`t[0]`, where 0 is the index of the projected `pnum` attribute) to the output relation if it satisfies the two predicates. Unlike SQL queries, however, the contents of `r` is not a set of concrete tuples, but rather a number of constraints that encodes the semantics of `Q2` over the input symbolic variables, as we will describe in Section 3.3.

Correlated Subqueries. Correlated subqueries (as in that in `Q1`) are compiled in a similar manner, except that the generated query functions take in a tuple from the enclosing query as parameter. For example, the subquery in the `WHERE` clause in `Q1` is compiled to the procedure `SubQ1` below (left), where `out_t` represents the tuple that is passed in from the enclosing query. And `Q1` is compiled to procedure `Q1` (right), with a call to `SubQ1` on line 4.

```
def SubQ1(out_t):
  r = []
  for t in Supply:
    if (t[0] == out_t[1]
          and t[1] < 10):
      r.append([t[1]])
  return [r.size()]
```

```
def Q1():
  r = []
  for t in Parts:
    if t[1] == SubQ1(t):
      r.append(t[0])
  return r
```

Aggregation and Grouping. The constraint generator also supports aggregation functions such as `COUNT`, `SUM`, `AVERAGE` on individual attributes. Grouping operations are rewritten to correlated subqueries and aggregates on single-columned relations following standard practice [10].

Other Features and Limitations. The COSETTE constraints generator currently supports most standard SQL features: besides features mentioned above, keywords including `EXISTS`, `IN`, `LEFT OUTER JOIN` are also supported. Currently COSETTE does not support string operations (e.g., `LIKE`) since COSETTE currently does not model strings as character arrays.

3.3 Generating Constraints

After compiling SQL queries to Rosette programs, they are executed by the Rosette runtime to generate constraints expressed in SMT-LIB format [9]. For example, `Q1` in Section 3.2 generates the following set of constraints:

```
(assert r[0] =
  (if (sv1 = subQ1(([sv0,sv1],sv2))
   then ([sv0],sv2)
   else (if (sv4 = subQ1(([sv3,sv4],sv5))
         then ([sv3],sv5)
         else Nil))
... ...
```

Note that the contents of `r` is a set of constraints. The constraint for the first row of the table (i.e., `r[0]`), for instance, says that `r[0]` equals to a single element list containing the second element (`sv0`)

[1] We use bold font for tuple multiplicities for ease of read purpose.

[2] The part that computes `Temp` is not shown due to space.

from the table `Parts` with multiplicity **sv2** if `sv1` equals to the result of evaluating `SubQ1` on the first tuple of the `Parts` table. Otherwise, it equals to either `([sv3],sv5)` or an empty list. Similar constraints are generated for `r[1]` as well.

All these constraints restrict the set of choices that the constraint solver can assign to each of the symbolic variables, a topic that we discuss next.

3.4 Finding Counterexamples

After compiling queries to constraints over symbolic variables, COSETTE send the constraints to the solver by asking it to find a model to the formula `Q1()≠Q2()`. As discussed in Section 1, two queries are inequivalent if a model (i.e., a counterexample) is found. While many solvers are available, in the current prototype COSETTE uses solving is done using the solver that comes with the Rosette runtime. On the other hand, if the solver is unable to find a counterexample, that means either a counterexample does not exist for the given size of the symbolic relations, or that the two queries are equivalent. For the former, COSETTE will increase the size of the symbolic relations and regenerate the constraints (as shown in Figure 2) until a counter example found or predetermined timeout. Once timed out, COSETTE will forward the queries to the Uninomial generator and proof assistant, as we will describe next.

4. VALIDATING EQUIVALENCES WITH PROOF ASSISTANT

While constraint solvers are efficient in disproving query equivalences, they cannot be used directly to validate equivalences as they only check on symbolic relations of bounded sizes (while true equivalences have to hold for relations of any size). As discussed in Section 1, in COSETTE, this is done by compiling the input queries to UniNomials, and then generating a proof script that is sent to a proof assistant to validate query equivalence. In this section we discuss this process.

4.1 Data Model

Rather than modeling relations as lists as described in Section 3.1, to prove query equivalences COSETTE instead models relation as a mathematical function that takes in a tuple and returns its multiplicity as a cardinal number. In other words, a relation R has type `Tuple` $\rightarrow \mathbb{N}$, and $R(t)$ is the multiplicity of a tuple t in R, with 0 meaning that t is not in R. This approach is inspired by K-relations [18], and doing so allows COSETTE to compile SQL queries to algebraic expressions consisting of operations such as addition, multiplication, summation, and truncation ($\|\cdot\|$) over cardinal numbers. We call such expressions UniNomials.[3]

By lifting SQL queries to UniNomials, proving the equivalence of two SQL queries becomes proving the equality of two UniNomials by syntactic comparison and the proof assistant's built-in axioms. This greatly simplifies the proofs and enables automation.

4.2 Compiling to UniNomials

We demonstrate how COSETTE compiles SQL queries to UniNomials using the following query Q as an example:

```
SELECT x FROM R
WHERE EXISTS (SELECT * FROM S WHERE b) -- Q
```

Q contains two symbolic relations, `R` and `S`, and a symbolic predicate `b`. First, the subquery on `S` is compiled to:

$$SubQ(t_o, t) : S(t) \times b(t_o, t)$$

[3] We implemented UniNomials in Coq using Univalent types [15], hence the name.

```
-- Q1
SELECT * FROM (R UNION ALL S)
WHERE b
-- Q2
(SELECT * FROM S WHERE b)
UNION ALL
(SELECT * FROM R WHERE b)
```

$Q_1(t) : b(t) \times (R(t) + S(t))$

$Q_2(t) : b(t) \times S(t) + b(t) \times R(t)$

```
Lemma 1:
∀ t, Q1(t) = Q2(t).
Proof:
apply comm_plus.
apply dist_prod.
reflexivity.
Qed.
```

Figure 4: Sample queries, their compilation to UniNomials, and script for their equivalence proof.

Given a tuple t, $SubQ$ first computes its multiplicity in S, and checks if it passes the symbolic predicate b. Here b is a function that takes in t and the tuple from the enclosing query, t_0, and returns 0 or 1. The multiplicity of t then equals to the product.

The existential predicate is compiled to another function $ExistsQ$:

$$ExistsQ(t_o) : \left\|\sum_u SubQ(t_o, u)\right\|$$

Given a tuple from the enclosing query t_o, $ExistsQ$ checks whether t_o is part of the results returned by the subquery. Conceptually, checking is done by iterating over all possible tuples u and summing up the multiplicities returned by $SubQ(t_o, u)$. Recall that if u is not in `S`, then $SubQ(t_o, u) = 0$. Since $ExistsQ$ is used as a predicate, we use the truncation function to return 1 if the sum is larger than 0, or 0 otherwise.

Finally, `Q` is compiled to:

$$Q(t) : \sum_u R(u) \times eq(u.x, t) \times SubQ(t)$$

Given a tuple t, Q checks whether it satisfies the selection predicate by passing it to $SubQ$. Then Q checks if t equals to the projection of the x attribute of some tuple u in R by calling eq. Like before, Q iterates over all possible tuples u and sum up the multiplicities after multiplying the three terms together.

4.3 Proving Equivalences

After compiling SQL queries to UniNomials, COSETTE generates a proof script for the proof assistant to check for query equivalence. The proof script contains instructions to break the proof goal into multiple subgoals so that proving all subgoals is sufficient to prove the original goal.

Figure 4 shows an example of two queries and their compilation to UniNomials, along with the proof script to show their equivalence. When executed, the proof assistant first applies the commutativity of $+$ and the distributivity of $\times$ over $+$ to rewrite $Q_1(t)$ to be syntactically identical as $Q_2(t)$, and then invoke the reflexivity axiom to finish the equivalence proof.

4.4 Proof Automation

Proof assistants often come with tactics to automatically search for proofs. In addition to the standard ones implemented in Coq, we have implemented a number of new tactics for UniNomials in COSETTE as described below.

HoTTRing Many proofs, such as the one in Figure 4, consist of associative-commutative rewrites. `HoTTRing` attempts to re-arrange UniNomial expressions into a standard form so that equality can be easily decided using syntactic comparison.

Congruence `Congruence` applies transitivity to eliminate variables in the proof goal. Furthermore, it applies transitivity to higher-order pure functions as well, i.e., $\forall a, b.(a = b) \Rightarrow (f(a) = f(b))$.

CQSolve Conjunctive Query is a well-known subclass of SQL with a complete decidable procedure for equivalence [15]. CQSolve is used to decide conjunctive query equivalence in COSETTE.

DeductSolve If the input queries return sets (e.g., starts with SELECT DISTINCT), DeductSolve turns the proof into bi-implication, since the multiplicity of each tuple in the result is either 0 or 1. Furthermore, the tactic splits the proof into two sub-proofs ($Q_1 \to Q_2$ and $Q_2 \to Q_1$), and applies other tactics to finish the proof.

Since the equivalence of two arbitrary SQL queries is undecidable [31], it is impossible to make the proof search completely automatic. However, as we will discuss in Section 6, the tactics described above enable COSETTE to automatically determine the equivalences for many queries with practical applications.

5. INTERACTIVE PROVING

Deciding the equivalence of arbitrary SQL queries is undecidable [31]. Hence, in COSETTE, there are cases that the constraint solver fails to find counterexamples, and the proof assistant cannot find a valid equivalence proof. COSETTE is designed to be interactive for such cases, both between the two proof engines and also with the user as we describe below.

5.1 Checking for Subgoal Validity

The proof assistant decomposes the equivalence proof goal into a number of subgoals during the proving process. However, it might not be able to prove some of the subgoals due to tactic limitations. When that happens, COSETTE will translate these subgoals to constraints, and invoke the constraint solver to try to falsify these subgoals. If the constraint solver falsifies *any* subgoal, then this means either the two queries are inequivalent (but the constraint solver failed to find a counterexample due to size of the symbolic relations), or that the proof assistant decomposed the subgoals incorrectly (e.g., the subgoals are logically stronger than needed) In both cases COSETTE will inform the user, may adjust the decomposition or disprove the equality between the input queries.

5.2 Interactions with the User

In addition to the automated mode, COSETTE is designed to interact with the user. First, if the constraint solver timed out while finding counterexamples and the proof assistant cannot find a valid proof, the proof goal expressed using UniNomials will be returned to the user. The user can then rewrite her proof scripts with the help of a library of lemmas provided by COSETTE, and resubmit the proof to COSETTE. COSETTE will incorporate them as part of the proof search procedure.

Second, a user can choose different tactics to break the proof goal into multiple subgoals in the proof assistant. COSETTE will apply automated tactics to try to solve these subgoals and only return the unsolved ones to the user for manual proofs.

In general, interactive proving brings the user into the loop to solve more advanced SQL queries. COSETTE is designed to be both fully automated, and also leverage the user's help via interaction after reducing the amount of proof burden. However, in practice our experiments show that COSETTE can readily prove many non-trivial query equivalences without user interactions as we will describe next.

6. EVALUATION

We have implemented a prototype of the COSETTE solver using Rosette (version 2.2) and Coq (version 8.5pl1). Our prototype includes 3k lines of Rosette code and 2k lines of Coq code. In this section we evaluate COSETTE's ability to determine query equivalence on four real-world datasets:

Dataset	Equiv.?	Total Number	Automatically Decided		Interactively Decided
			No.	Avg Time	
Bugs	No	3	3	8.8 s	-
Exams	No	5	5	1.3 s	-
XData	No	9	9	< 1 s	-
Rules	Yes	23	17	< 1 s	6
Exams	Yes	4	3	< 1 s	1

Figure 5: Evaluation Summary.

- **Bugs** contains 3 real-world bug reports including the COUNT bug [17], and two other optimizer errors in real-world systems [1, 3].
- **Exams** is a set of questions from the undergraduate data management class [2] where students are asked to identify whether two queries are equivalent or not.
- **XData** contains query and mutant pairs collected from XData [29]. Each mutant query is generated by mutating the original query, and COSETTE is asked to identify if the mutant preserves the original query's semantics.
- **Rules** contains various classical SQL optimizer rewrite rules ranging from relational algebra rewrites to conjunctive query equivalences and others. The full list is described in [15].

Figure 5 shows the summary of the evaluation. COSETTE found counterexamples for **Bugs**, all inequivalent queries in **Exams** and all mutant queries listed in **XData**. We select a few inequivalent queries and describe how COSETTE generates small counterexamples for them below. COSETTE also automatically proved 17 out of 23 equivalent queries in **Rules**, with the remaining proved with interaction. Among the automatically proven queries, 7 are conjunctive queries that are proved using COSETTE's CQSolve tactic. The remaining are solved using other tactics discussed in Section 4.4.

6.1 Finding Counterexamples

The COUNT Bug. The COUNT bug is an incorrect optimizer rewrite rule [17] expressed using Q1 and Q2 as shown in Figure 3. COSETTE returns a counterexample containing the following two concrete tables when asked whether the two queries are equivalent:

pnum	qoh	multiplicity
0	0	**8**
2	2	**15**

pnum	shipdate	multiplicity
2	9	**2**

When executed, Q1 returns [([0],**8**), ([2],**15**)] while Q2 returns [([2],**15**)]. COSETTE took 2 iterations to find the counterexamples since the Parts table requires at least two unique tuples to demonstrate the inequivalence.

Oracle 12c Optimizer Bug. We next asked COSETTE to determine query equivalence of a real-world bug report on Oracle 12c [3]. In the report, the user mentioned that the result of the query below is incorrect due to an optimizer bug where it converted the first left outer join (Line 7) to a hash join. This resulted in a wrong execution plan, since tuples from thing that have no match in tr are removed by the hash join but retained by the original outer join.

```
1 --Table schemas:
2 --   thing(tid, tname), tr(rid, tid, type),
3 --   ta_status(rid, status), tb_status(rid, status)
4 SELECT t.tid, t.name, tas.status, tbs.status
5 FROM thing t LEFT JOIN tr
6   ON t.tid = tr.tid
7 LEFT JOIN ta_status tas
8   ON (tr.rid IS NOT NULL
9       AND tr.type = 1 AND tr.rid = tas.rid)
10 LEFT JOIN tb_status tbs
11   ON (tr.rid IS NOT NULL
12       AND tr.type = 2 AND tr.rid = tbs.rid)
```

Although the bug is difficult for users to identify, COSETTE identified the incorrectness of the optimization efficiently[4]: when fed with the original and optimized queries into COSETTE, the counterexample below is returned:

```
thing = [([0,0],15)]        tr = []
ta_status = [([0,0],4)]     tb_status = [([0,0],3)]
```

Given the generated counter example, the original query evaluates to `[([0,0,null,null],15)]` while the incorrect optimized query evaluates to an empty table, indicating that the rewrite is incorrect.

Two Inequivalent Queries from Exams. One question from the exams ask students whether the two queries below are equivalent:

```
SELECT x.uid, x.uname,                            -- Q1
       (SELECT count(*) FROM Picture y
        WHERE x.uid = y.uid AND y.num > 1000000)
FROM Usr x
WHERE x.city = 'Denver';

SELECT x.uid, x.uname, COUNT(*)                   -- Q2
FROM Usr x, Picture y
WHERE x.uid = y.uid AND y.num > 1000000
      AND x.city = 'Denver'
GROUP BY x.uid, x.uname;
```

`Q1` and `Q2` are inequivalent as `Q2` filtered out all the cities that are not `'Denver'` first, so the count only considers `'Denver'` tuples, whereas `Q1` counts all tuples prior to filtering. COSETTE took 3 iterations to find a counterexample.

6.2 Proving Equivalence of SQL Queries

Two Queries from Exams. The following is one of the questions from the Exams dataset:

```
SELECT DISTINCT x.uid, x.uname                    -- Q1
FROM Usr x, Picture u, Picture v, Picture w
WHERE x.uid = u.uid AND x.uid = v.uid
      AND x.uid = w.uid AND u.size > 1000000
      AND v.size < 3000000 AND w.size = u.size;

SELECT DISTINCT x.uid, x.uname                    -- Q2
FROM Usr x, Picture u, Picture v, Picture w
WHERE x.uid = u.uid AND x.uid = v.uid
      AND x.uid = w.uid AND u.size > 1000000
      AND v.size < 3000000 AND w.size = v.size;
```

While `Q1` and `Q2` are not conjunctive queries, COSETTE proves `Q1 = Q2` using `DeductSolve` as they both return sets.

Magic Set. Magic set rewrites are widely used to improve the performance of complex decision support queries [28]. Magic set rewrites can be decompose to a set of rewrites using semi-joins, with one of them expressed using the following two SQL queries:

```
SELECT * FROM R, S WHERE b;                       -- Q1
SELECT * FROM (R SEMIJOIN S ON b), S WHERE b;     -- Q2
```

Here `b` is a symbolic predicate. COSETTE first rewrites the semijoin into join and then compile the queries into UniNomials as follows:

$$Q_1(t) : b(t) \times R(t.R.*) \times S(t.S.*)$$
$$Q_2(t) : b(t) \times \|\Sigma_u b(t.R.*, u) \times S(u)\| \times R(t.R.*) \times S(t.S.*)$$

[4] COSETTE currently uses integers to model strings.

Here $t.R.*$ projects all attributes of `R` from t, and similarly for $t.S.*$. Proving the equivalence of `Q1` and `Q2` cannot be done automatically. We proved the equivalence interactively by first calling `HoTTRing` to rewrite the UniNomials to:

$$Q_1(t) : b(t) \times R(t.R.*) \times S(t.S.*)$$
$$Q_2(t) : b(t) \times R(t.R.*) \times S(t.S.*) \times \|\Sigma_u b(t.R.*, u)) \times S(u)\|$$

Then we manually applied the lemma: $\forall T, P : \mathbb{N}$, *if* P *is either* 0 *or* 1, *then* $(T \to P) \Rightarrow (T = T \times P)$. After applying this lemma with $T = Q_1$, and $P = \|\Sigma_u b(t.R.*, u)) \times S(u)\|$, we only need to prove the implication using the tactics described in Section 4.4.

7. RELATED WORK

Query equivalence is a topic that has been studied extensively. As discussed in Section 1, prior work focuses on the decidability of the problem, with most classes of queries proven to be undecidable [31, 27]. One notable exception is conjunctive queries where a complete decision procedure is available [8].

While there has been work on applying formal methods to query execution, they focus on building a provably-correct database implementation [24] or test generation [33]. The data models used in COSETTE are inspired by prior work on modeling relations, including work on test generation [33] and program compilation [14]. To our knowledge COSETTE is the first tool that supports deciding *both* equivalence and inequivalence of SQL queries.

Finally, there are also various SQL test generation tools available. Such tools use techniques like mutation [29] and classification [32] to generate test data. COSETTE instead relies on formal methods to formally prove the equivalence of queries.

8. CONCLUSION

We presented COSETTE, the first solver for SQL queries that leverages recent advances in formal methods. While COSETTE cannot solve equivalences of all SQL queries due to theoretical limitations, our experiments show that it can efficiently determine the equivalences of a wide variety of real-world queries.

9. ACKNOWLEDGEMENTS

We thank Dan Suciu for his contribution to this paper. We thank Kaiyuan Zhang for his help with the early prototype of the system and Emina Torlak for the discussion. We would like also to thank the reviewers for their advices to improve the paper.

10. REFERENCES

[1] Bug 5673: Optimizer creates strange execution plan leading to wrong results. http://tinyurl.com/hwwn53r.

[2] CSE 344 Introduction to Data Management, Fall 2013, University of Washington. https://courses.cs.washington.edu/courses/cse344/13au/.

[3] Query featuring outer joins behaves differently in oracle 12c. http://stackoverflow.com/questions/19686262.

[4] The Coq Proof Assistant. https://coq.inria.fr/.

[5] The Lean Theorem Prover. https://leanprover.github.io/.

[6] The Yices SMT Solver. http://yices.csl.sri.com/.

[7] The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[8] S. Abiteboul et al. *Foundations of Databases*. Addison-Wesley, 1995.

[9] C. Barrett et al. SMT-LIB standard v2.5. http://smtlib.cs.uiowa.edu.

[10] P. Buneman et al. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[11] B. Chandra et al. Data generation for testing and grading SQL queries. *VLDB J.*, 24(6):731–755, 2015.

[12] S. Chandra et al. Angelic debugging. In *ICSE*. ACM, 2011.

[13] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[14] A. Cheung et al. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14. ACM, 2013.

[15] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics. *ArXiv e-prints*.

[16] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[17] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *SIGMOD Conference*, pages 23–33. ACM Press, 1987.

[18] T. J. Green et al. Provenance semirings. In *PODS*, pages 31–40, 2007.

[19] C. Hawblitzel et al. Ironfleet: proving practical distributed systems correct. In *SOSP*, pages 1–17. ACM, 2015.

[20] T. S. Jayram et al. The containment problem for REAL conjunctive queries with inequalities. In *PODS*, pages 80–89. ACM, 2006.

[21] R. Kearns et al. A teaching system for SQL. In *ACSE*, volume 2, pages 224–231. ACM, 1997.

[22] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[23] L. Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

[24] J. G. Malecha et al. Toward a verified relational database management system. In *POPL*, pages 237–248, 2010.

[25] E. Meijer et al. LINQ: reconciling object, relations and XML in the .net framework. In *SIGMOD Conference*, page 706. ACM, 2006.

[26] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[27] Y. Sagiv et al. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[28] P. Seshadri et al. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD Conference*, pages 435–446, 1996.

[29] S. Shah et al. Generating test data for killing SQL mutants: A constraint-based approach. In *ICDE*, pages 1175–1186, 2011.

[30] E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, page 54. ACM, 2014.

[31] B. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *D. Akad. Nauk USSR*, 70(1):569–572, 1950.

[32] Q. T. Tran et al. Query by output. In *SIGMOD Conference*, pages 535–548. ACM, 2009.

[33] M. Veanes et al. Qex: Symbolic SQL query explorer. In *LPAR (Dakar)*, volume 6355, pages 425–446. Springer, 2010.

[34] W. Visser et al. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.