Demonstration of the Cosette Automated SQL Prover

Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung and Dan Suciu
University of Washington
{chushumo, dli132, clwang, akcheung, suciu}@cs.washington.edu
http://cosette.cs.washington.edu

ABSTRACT

In this demonstration, we showcase Cosette, the first automated prover for determining the equivalences of SQL queries. Despite theoretical limitations, Cosette leverages recent advances in both automated constraint solving and interactive theorem proving to decide the equivalences of a wide range of real world queries, including complex rewrite rules from the database literature. Cosette can also validate the inequality of queries by finding counter examples, i.e., database instances which, when executed on the two queries, will return different results. Cosette can find counter examples of many real world inequivalent queries including a number of real-world optimizer bugs. We showcase three representative applications of Cosette: proving a query rewrite rule from magic set rewrite, finding counter examples from the infamous optimizer bug, and an interactive visualization of automated grading results powered by Cosette, where Cosette is used to check the equivalence of students' answers to the standard solution. For the demo, the audience can experience through the three applications, and explore the Cosette by interacting with the tool using an easy-to-use web interface.

1. INTRODUCTION

We present Cosette, the first automated SQL prover that can decide whether two SQL queries are equivalent or not. Built on top of prior research on the theoretical limitations of reasoning about semantic query equivalences, we have used Cosette to the equivalences of a wide range of real world SQL queries efficiently [7].

The Cosette Prover. To determine the equivalences of SQL queries, Cosette leverages recent advances in both automated constraint solving and interactive theorem proving. If two queries are inequivalent, Cosette uses rosette [16], an automatic constraints solver, to find counter examples, i.e., database instances in which running the two queries on will return different results. If two queries are equivalent, Cosette finds a formal proof for the equivalence and validates the proof using a proof assistant, Coq [3], a widely used in verifying machine checkable proofs.

Cosette encodes SQL queries to logic formulas to find counter examples of inequivalent SQL queries by constraint solvers. Cosette encodes SQL queries to K-Relations [10], where each relation R is represented as a mathematical function that takes a tuple as input and returns its multiplicity in R. This allows the proof assistant to easily search for machine checkable proofs of equivalence. In addition, Cosette includes a number of optimizations to make solving and proof search efficient. For example, our data model allows constraints to be easily generated and solved, and we have also developed a number of proof tactics to speed up proof search. Besides the fully automated mode, Cosette also allows developers to interact with the tool by writing their own proof scripts using a library of lemmas provided by Cosette. As a result, Cosette can solve a wide range of real world SQL queries, including equivalent SQL rewrites and inequivalent SQL queries such as real world optimizer bugs efficiently [7].

Applications of Cosette. As a computer-aided tool for reasoning about SQL query equivalences, Cosette can be used in a variety of real-world database applications:

Correctness of RDBMS Rewrite rules are the basis of query optimizers. Creating new rewrite rules is error prone, and discovering bugs in them is difficult. One example is the infamous `COUNT` bug [11], which took 5 years identify. Using Cosette, on the other hand, it only takes fewer than 10 seconds to generate a counter example to prove that the rewrite is not sound.

Semantic Caching. Prior work on query caching is based on the exact matching of query strings, which loses the opportunity to cache semantically equivalent queries. Using Cosette, many database applications can cache query results and use them to answer a wider range of queries, which will improve application efficiency.

Automatic Grading. The need for computer science education keeps growing recently. Unfortunately, grading students' homework is a tedious and time consuming task. Cosette provide a scalable solution for grading data management course homework, and can mitigate the workload of course instructors. In addition, the counter examples returned by Cosette can help students to learn SQL interactively online, where usually timely reaction to students is lacking.

Using Cosette. Both machine checkable theorem proving and constraints generation require high experty in formal methods. To use Cosette, users needs to understand none

SIGMOD'17, May 14 - 19, 2017, Chicago, IL, USA

ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3058728

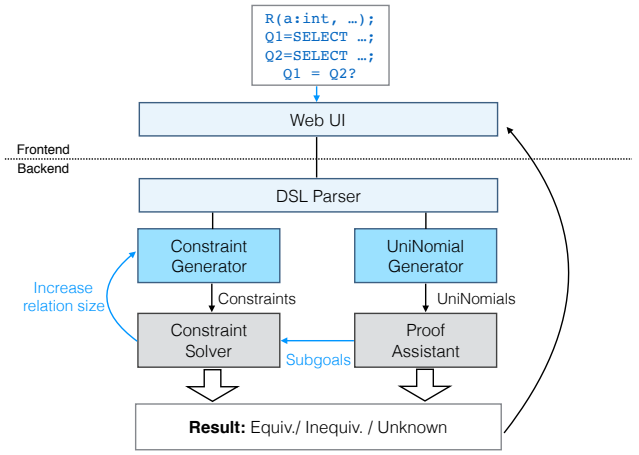

Figure 1: Cosette architecture, where texts and arrows in blue indicate user interactions.

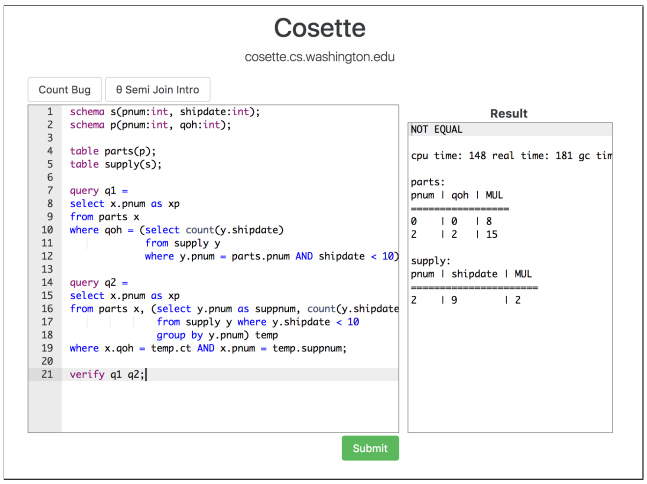

Figure 2: Cosette Web User Interface.

of them. Cosette provide a DSL (Domain Specific Language) to let users specify queries to be verified. Given the input, Cosette will compile users' queries to encodings that will be sent to the proof assistant and constraint solvers. The Cosette DSL supports declarations of schemas, relations and predicates, and queries to be checked using SQL syntax. In addition, Cosette DSL supports symbolic relations and predicates as well as extensible schemas, so that users can use Cosette to check the equivalences of two templated queries in addition to concrete queries. Cosette provides a web interface for users to write Cosette DSL in the editor pane and shows the result as well as error messages. If the result is not equivalent, the web interface will show the counter examples.

Detailed examples of using Cosette will be provided in Sec. 3.

2. Cosette OVERVIEW

In this section, we present an overview of Cosette as the context for understanding the novel applications that we will demonstrate.

Figure 1 shows the overall architecture of Cosette. Cosette takes users' query rewritten in the Cosette DSL using the web interface (shown in Figure 2). In the Cosette DSL, users can write declarations of schemas, relations, predicate and aggregation functions. Users then specify the SQL queries to be checked on these declarations as well as literals. Cosette web user interface passes user inputs to the DSL parser, which parses them to Cosette ASTs (Abstract Syntax Tree) and passes the ASTs to the two compilation toolchains shown in the bottom of Figure 1.

On one hand, the *constraint generator* translates the ASTs into constraints. During the translation, the constraints generator bounds the size of each symbolic relation. The generator 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 constraint solver then solves the generated constraints. If a counter example is found, the input queries are inequivalent, and the example is returned to the user via the web interface. If a counter example cannot be found, the generator increases the size of the symbolic relations, generates constraints on the increased sized symbolic relations and calls the constraint solver again. This process will be repeated until timeout or a counter example is found.

On the other hand, if the constraint solver cannot find a counter example until timeout, Cosette will forward the ASTs 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 generator sends the UniNomials to the proof assistant. The proof assistant uses our proof search heuristics (tactics) to find a valid proof for the equivalences of the queries. If the proof assistant cannot find proof, it will return *unknown* to the user through the web interface. The user has the following options:

1. Increase the time budget so that a larger counter example can be searched.

2. Manually complete the unfinished proof and check the validity of the proof using the proof assistant. The user can still use the lemmas from the Cosette library as well as use automatic tactics for proving subgoals and constraint solver for disproving subgoals.

Cosette includes many optimization to make solving efficient. For example, we use standard syntactic rewrites [5] to support grouping on top of K-relations. We model multiplicity as symbolic variables so that many aggregation queries requires less symbolic variables to encode. More details can be found in [7, 8].

3. DEMONSTRATIONS

We demonstrates three use cases of Cosette, 1) proving equivalent rewrite rules, 2) finding optimizer bugs, and 3) automated homework grading and analysis. Cosette's web interface and the Cosette DSL will be demonstrated when we walk through these use cases.

3.1 Proving Query Equivalences

Users can use Cosette to prove query equivalences by specifying queries to be checked in Cosette DSL. Figure 3 shows an example of specifying one of magic set rewrite rules, introduction of θ-semijoin, in Cosette. Magic set rewrites are well-known rewrite rules that are used for rewriting complex decision support queries in commercial systems

```
-- schema declarations
schema s1(??);
schema s2(??);
-- table declarations
table r1(s1);
table r2(s2);
-- predicate declarations
predicate θ(s2,s1);
-- queries
query q1 =
select * from r1 x, r2 y where θ(x,y);
query q2 =
select * from (r2 SEMIJOIN r1 ON θ) x, r1 y
    where θ(x,y);
-- verify statement
verify q1 q2;
```

Figure 3: Example of using Cosette DSL to specify one of the magic set rewrite rules: Introduction of θ-SemiJoin

Result

```
Proof.
    semi_ring.
    apply path_universe_uncurried.
    apply hprop_prod_l'.
    intros [h0 h1].
    apply tr.
    destruct t as [t1 t2].
    refine (t2; _).
    cbn in *.
    refine (h1, h0).
==========================

EQUAL
```

Figure 4: Result of running the Cosette program in Figure 3.

such as IBM's DB2 database [13, 15]. The queries we show here is from one of the three basic rewrite rules that can compose magic set rewrites [15].

As shown in Figure 3, schemas and tables need to be first declared using the Cosette DSL. Since this rewrite rules does not require any specific attributes on the schemas of the both tables, we put `??` in the schema declaration, which means schema `s1` and `s2` can contain any attribute of any datatype. Then we declare the predicate that will be used in the queries, θ, and the schema of the tuples that this predicate will be evaluated on. In this example, θ(`s2, s1`) means θ will be evaluated on tuples with the schema that is the union of `s2` and `s2`. The predicate we defined here is called *symbolic predicate*. It will be used in the queries to be checked later. If we prove the equivalence of queries with *symbolic predicates*, the equivalencies of queries where such symbolic predicates are instantiated with *any* concrete predicate is also proved. For example, if we proved the equivalences of the queries in Figure 3, the equivalences of the two concrete queries by replacing θ with `x.a = y.a` (or any other predicate) is also validated.

After declaring schemas and predicates, users write the queries that are to be proved using standard SQL syntax. These queries can have literals, such as numbers and strings, as well as the symbolic tables and symbolic predicates that are declared earlier. When using symbolic predicates such as θ in queries, users need to replace its schemas with the

```
-- schema declarations
schema s(pnum:int, shipdate:int);
schema p(pnum:int, qoh:int);
-- table declarations
table parts(p);
table supply(s);
-- query 1
query q1 =
select x.pnum as xp
from parts x
where qoh = (select count(y.shipdate)
             from supply y
             where y.pnum = parts.pnum and
                   shipdate < 10);
-- query 2
query q2 =
select x.pnum as xp
from parts x,
     (select y.pnum as suppnum,
             count(y.shipdate) as ct
      from supply y
      where y.shipdate < 10
      group by y.pnum) temp
where x.qoh = temp.ct and x.pnum = temp.suppnum
    ;
-- verify statement
verify q1 q2;
```

Figure 5: `COUNT` bug specified in Cosette DSL

actual tables that this predicate is on, in the example, θ(`x, y`) means that θ is evaluated on `x` and `y`.

The editor provided by the Cosette web interface shown in Figure 2 is where users write the queries to be checked using the Cosette DSL. The web editor supports syntax highlighting and visual marking on parsing errors for users' convenience. In this example, after submitting the Cosette DSL program using the web UI, since the two queries are indeed equal, Cosette shows the proof script containing the steps it takes to prove the equivalence (Figure 4).

3.2 Finding Counter Examples for Inequivalent Queries

Cosette can also be used to find counter examples of inequivalent queries. The steps that users take to find counter examples is similar to the steps that are used to prove equivalent queries. We provided a example Cosette program expressing the infamous `COUNT` optimizer bug. The bug was a proposed rewrite rule for rewriting queries with aggregation and grouping by Kim [11]. This rewrite rule is discovered to be incorrect in a paper published 5 years later [9]. Figure 5 shows the `COUNT` bug expressed in Cosette DSL.

When users runs this program in Cosette, Cosette will return the counter example it finds for these two queries (in our experiment this was done within 1 second). The counterexample contains two concrete tables as shown in Figure 2. While solving, Cosette will show the process of increasing the bound of size of the symbolic tables to the users. If Cosette cannot find a counterexample given the default time limit, the users could choose to increase the time budget or allocating more computational power to the backend.

3.3 Automated Grading

Cosette can be used to power automated grading of data management course homework, especially in MOOC set-

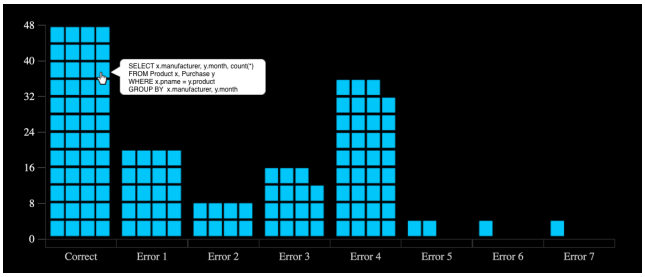

Figure 6: SandDance visualization of students answer on a homework problem in undergraduate database course at the University of Washington. Each square represents one student's answer. Each bar represents an equivalence class.

tings. These courses often need to teach students to write complex SQL queries. Previous automated grading solution runs students answers using some test input, and check whether the results are the same as the result by running the standard solution. This has two drawbacks. 1) the test input provided may not capture some of the errors in student answer. 2) designing good test input is hard and requires extra work. Using COSETTE can solve these two problems with the ability to reason about the equivalences automatically. In addition, being able to return counter examples when the answers from students are wrong can provide similar feedback compared with a human tutor. The students are able to reason about the their errors by using the counter examples provided.

We demonstrate an automated grading result of a homework problem from the undergraduate data management course (CSE344 [1]) using COSETTE. Figure 6 shows an interactive visualization for the course tutors based on the result of running students' answers using COSETTE. This visualization is inspired by the SandDance visualization [2] from Microsoft. In this visualization, we clustered all the answers into equivalence classes based on their semantic equivalences. Each square represents an student's answer. Naively, this requires $O(N^2)$ times calling COSETTE by checking equivalences of the answers pairwise. However, some checkings can be avoided since we know the transitivity of the equivalences, e.g., $Q1 = Q2 \wedge Q2 \neq Q3 \Rightarrow Q1 \neq Q3$. In Figure 6, the size of each bar is proportional to the size of the equivalence class that it represents. The course tutor can use such visualization to identify the most common error made by the students very easily. When the cursor is hovered on top of a square, the student answer that it represents will show up as well. The demo audience will be able to interact with the visualizations based on pre-computed results of student submissions.

4. RELATED WORK

The theoretical limitations of query equivalences have been studied extensively. Except for conjunctive queries, which have a decision procedure to check equivalences [4], most interesting classes of SQL queries are undecidable [17, 14]. Previous work on applying formal methods to database queries include [12, 18], which focus on constructing a provably-correct database implementation [12] or test generation [18]. The data models used in COSETTE are inspired by prior work, including work on provenance [10], test generation [18] and application query compilation [6]. To our knowledge, COSETTE is the first tool that supports deciding *both* equivalence and inequivalence of SQL queries.

5. CONCLUSION

In this proposal, we demonstrate the key features of COSETTE and COSETTE's novel applications through three interactive use cases. COSETTE is the first automated prover for SQL equivalences. COSETTE leverages advances in formal methods and can efficiently prove the equivalences or find counter examples on two SQL queries. COSETTE can be used in many scenarios, such as improving the correctness and efficiency of database applications, and auto-grading of SQL homework assignments.

Acknowledgement. This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1614738, CCF-1535565 and CNS-1563788; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; and gifts from Adobe, Amazon, and Google.

6. REFERENCES

[1] Cse344: Introduction to data management. courses.cs.washington.edu/courses/cse344/, 2017.
[2] Sanddance project. https://www.microsoft.com/en-us/research/project/sanddance/, 2017.
[3] The Coq Proof Assistant. https://coq.inria.fr/, 2017.
[4] S. Abiteboul et al. *Foundations of Databases*. Addison-Wesley, 1995.
[5] P. Buneman et al. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
[6] A. Cheung et al. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14. ACM, 2013.
[7] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for SQL. In *CIDR*. www.cidrdb.org, 2017.
[8] S. Chu, K. Weitz, A. Cheung, and D. Suciu. Hottsql: Proving query rewrites with univalent sql semantics. In *PLDI*. ACM, 2017.
[9] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *SIGMOD Conference*, pages 23–33. ACM Press, 1987.
[10] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.
[11] W. Kim. On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.
[12] J. G. Malecha et al. Toward a verified relational database management system. In *POPL*, pages 237–248, 2010.
[13] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *SIGMOD Conference*, pages 247–258, 1990.
[14] Y. Sagiv et al. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.
[15] P. Seshadri et al. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD Conference*, pages 435–446, 1996.
[16] E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, page 54. ACM, 2014.
[17] B. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *D. Akad. Nauk USSR*, 70(1):569–572, 1950.
[18] M. Veanes et al. Qex: Symbolic SQL query explorer. In *LPAR (Dakar)*, volume 6355, pages 425–446. Springer, 2010.