Automated Verification of Query Equivalence Using Satisfiability Modulo Theories

Qi Zhou   Joy Arulraj   Shamkant Navathe
equitas@cc.gatech.edu
Georgia Institute of Technology

William Harris
wrharris@galois.com
Galois.Inc

Dong Xu
shaojie@alibaba-inc.com
Alibaba Group

ABSTRACT

Database-as-a-service offerings enable users to quickly create and deploy complex data processing pipelines. In practice, these pipelines often exhibit significant overlap of computation due to redundant execution of certain sub-queries. It is challenging for developers and database administrators to manually detect overlap across queries since they may be distributed across teams, organization roles, and geographic locations. Thus, we require automated cloud-scale tools for identifying equivalent queries to minimize computation overlap.

State-of-the-art algebraic approaches to automated verification of query equivalence suffer from two limitations. First, they are unable to model the semantics of widely-used SQL features, such as complex query predicates and three-valued logic. Second, they have a computationally intensive verification procedure. These limitations restrict their efficacy and efficiency in cloud-scale database-as-a-service offerings.

This paper makes the case for an alternate approach to determining query equivalence based on symbolic representation. The key idea is to effectively transform a wide range of SQL queries into first order logic formulae and then use satisfiability modulo theories to efficiently verify their equivalence. We have implemented this symbolic representation-based approach in EQUITAS. Our evaluation shows that EQUITAS proves the semantic equivalence of a larger set of query pairs compared to algebraic approaches and reduces the verification time by $27\times$. We also demonstrate that on a set of 17,461 real-world SQL queries, it automatically identifies redundant execution across 11% of the queries. Our symbolic-representation based technique is currently deployed on Alibaba's MaxCompute database-as-a-service platform.

1. INTRODUCTION

The proliferation of cloud computing has resulted in the availability of a growing number of database-as-a-service (DBaaS) offerings (e.g., Microsoft's Azure Data Lake [9], Google's BigQuery [11], and Alibaba's MaxCompute [1]). These DBaaS solutions enable users

to quickly create and deploy complex data processing pipelines. In practice, these pipelines may have significant overlap of computation (i.e., redundant execution of certain sub-queries). For example, around 45% of the queries executed on Microsoft's SCOPE service have computation overlap with other queries [40]. This results in increased consumption of computational resources, higher data processing costs, and longer query execution times.

Developers and database administrators (DBAs) may resolve these problems by increasing the modularity of their data processing pipelines to reuse the results of frequently executed sub-queries. However, in practice, it is challenging for developers and DBAs to *manually* detect overlap across queries since they may be distributed across teams, organization roles, and geographic locations. Thus, we require automated cloud-scale tools for identifying semantically equivalent queries to minimize computation overlap.

The fundamental problem of determining if two SQL queries are semantically equivalent is undecidable [14, 18]. Given this constraint, prior efforts have focused on identifying a subset of relational algebra where it is feasible to determine equivalence of queries under set and bag semantics [1] [22, 52, 29, 39]. This line of research examined the theoretical underpinnings of this problem by targeting only conjunctive queries thereby limiting their ability to identify overlap in other types of SQL queries.

More recently, Chu et al. have proposed a pragmatic approach to determining the semantic equivalence of queries [51, 26]. Their COSETTE and UDP tools transform SQL queries to algebraic expressions and then use a *decision procedure* to compare the resultant algebraic expressions. The decision procedure includes a set of re-write rules for verifying the equivalence of algebraic expressions. COSETTE and UDP tools vary with respect to the algebraic representation to which they convert the given queries. While the former tool uses $\mathcal{K}$-relations, the latter leverages $\mathcal{U}$-semirings. Their experiments demonstrate that these algebraic structures are sufficient to model the semantics of complex real-world SQL queries.

The algebraic approaches are geared towards verifying the correctness of rewrite rules in query optimizers. They can, therefore, validate complex structural transformations of SQL queries. However, they suffer from two limitations. First, they are unable to model the semantics of widely-used SQL features, such as complex query predicates, arithmetic operations, and three-valued logic for supporting NULL [47]. This limits their ability to support a wide range of real-world SQL queries. Second, they have a computationally intensive decision procedure. This is because they apply a series of re-write rules on the given algebraic expressions to determine

---

[1] A bag (a.k.a. multi-set) is an unordered collection of elements with duplicates [15]. In contrast, a set is an unordered collection of elements without duplicates [44].

their equivalence. These two limitations restrict their efficacy and efficiency in cloud-scale DBaaS offerings.

In this paper, we propose to address these limitations by leveraging an alternate approach for determining query equivalence (QE). We derive the *symbolic representation* (SR)[2] of SQL queries and use *satisfiability modulo theories* (SMT) to determine their equivalence [32]. This approach can model the semantics of widely-used SQL features, such as complex query predicates, arithmetic operations, and three-valued logic. Reducing the problem of determining the equivalence of queries to that of deciding the satisfiability of formulae in *first-order logic (FOL)* enables the usage of computationally efficient SMT solvers. Deciding the satisfiability of FOL formulae is an NP-complete problem. However, in practice, modern solvers employ heuristics from satisfiability theory [30] to efficiently solve FOL formulae [32, 34, 45].

The problem of deciding the equivalence of a pair of `SELECT-PROJECT-JOIN` queries can be reduced to the constraint satisfiability problem. This is because if these queries were equivalent, for all possible inputs, every pair of equivalent tuples in their output tables must be constructed from the same finite set of tuples in their input tables. So, we can derive constraints between SR of these queries and then determine the satisfiability of those constraints. However, this approach is not sufficient for handling more complex SQL queries containing aggregate functions and different types of `OUTER JOIN`. Unlike `SELECT-PROJECT-JOIN` queries, in these complex SQL queries, there is no *fixed* set of input tuples that contribute to an arbitrary output tuple across all possible inputs. We address this challenge by introducing a set of rules for decomposing the equivalence proof into proving relational constraints between the constituent sub-queries.

We implemented our SR-based approach in EQUITAS, a tool for automatically verifying the equivalence of SQL queries under set semantics [44]. We evaluate EQUITAS using a collection of pairs of equivalent SQL queries available in the Apache CALCITE framework [3]. Each pair is constructed by applying various query optimization rules on complex SQL queries with a wide range of features, including arithmetic operations, three-valued logic for supporting `NULL`, sub-queries, grouping, and aggregate functions. Our evaluation shows that EQUITAS can prove the semantic equivalence of a larger set of query pairs (67 out of 232) compared to UDP (34 out of 232). Furthermore, EQUITAS is $27\times$ faster than UDP on this benchmark. In addition to the Apache Calcite benchmark, we evaluate the efficacy of EQUITAS on a cloud-scale workload comprising of $17,461$ real-world SQL queries from Ant Financial Services Group [2]. 11% of the queries in this workload are redundantly being executed. These queries contain heavyweight relational operators, such as aggregate functions. We demonstrate the impact of redundant query materialization on runtime performance of this production workload in a DBaaS platform.

In summary, we make the following contributions:

- We illustrate the limitations of algebraic approaches and motivate the need for an alternate approach to determining the equivalence of SQL queries (Section 2).

- We introduce a symbolic representation-based approach that enables the usage of computationally efficient SMT solvers (Section 3).

- We propose a rule-based extension to this approach for handling more complex SQL queries containing aggregate functions and different types of `OUTER JOIN` (Section 3).

---

[2]The symbolic representation of a query Q is a set of formulae in first-order logic that denote the relational operators, predicates, and other components of Q.

- We implemented our SR-based approach in EQUITAS and evaluated its efficacy and efficiency on two benchmarks. We demonstrate that this approach proves the semantic equivalence of a larger set of query pairs in the CALCITE benchmark compared to UDP, the state-of-the-art QE verifier (Section 6).

- We illustrate the utility of EQUITAS in identifying computation overlap in $17,461$ production SQL queries from a financial company and highlight the impact of redundant query materialization on runtime performance (Section 6).

2. BACKGROUND

We begin by highlighting the limitations of algebraic approaches for determining query equivalence (QE) in Section 2.1. We then illustrate how an alternate approach based on symbolic representations can reduce the problem of QE to that of verifying the satisfiability of FOL formulae in Section 2.2. We conclude with a brief overview of SMT solvers in Section 2.3.

2.1 Challenges for Algebraic Approaches

State-of-the-art automated tools for determining QE adopt algebraic approaches [51, 26]. While COSETTE uses $\mathcal{K}$-relations for representing SQL queries, UDP leverages $\mathcal{U}$-semirings. The latter tool covers a broader set of SQL features compared to COSETTE. At a high level, the UDP algorithm performs query rewrites using $\mathcal{U}$-expressions reminiscent of the chase/back-chase procedure [46, 52]. After translating queries to algebraic expressions, it applies a set of rules for canonizing and minimizing the expressions. Lastly, it performs a sequence of tests to check for isomorphisms and homomorphisms between the rewritten algebraic expressions to determine the equivalence of the original queries.

UDP can prove the equivalence of complex SQL queries by using algebraic reasoning. However, it is unable to support certain widely-used SQL features. We motivate the need for an alternate approach to proving QE using three illustrative examples derived from the CALCITE framework [3]. We explain why algebraic approaches are unable to decide the equivalence of these semantically equivalent query pairs. We will later present how an alternate approach based on symbolic representation can address these limitations in Section 3. These query pairs operate on two tables:

- Employee table (`EMP`): $\langle$ `EMP_ID, EMP_NAME, DEPT_ID` $\rangle$

- Department table (`DEPT`): $\langle$ `DEPT_ID, DEPT_NAME` $\rangle$.

EXAMPLE 1. COMPLEX ARITHMETIC EXPRESSIONS:

```
Q1: SELECT * FROM
    (SELECT * FROM EMP WHERE DEPT_ID = 10) AS T
    WHERE T.DEPT_ID + 5 > T.EMP_ID;

Q2: SELECT * FROM
    (SELECT * FROM EMP WHERE DEPT_ID = 10) AS T
    WHERE 15 > T.EMP_ID;
```

Q1 is a nested query where the inner query selects employees whose `DEPT_ID` is 10. The outer query then applies another filter on the results of the inner query by retrieving tuples where `DEPT_ID + 5` is larger than `EMP_ID`. Q2 is another nested query where the inner query retrieves tuples from `EMP` whose `DEPT_ID` is 10. The outer query then selects a subset of those tuples whose `EMP_ID` is less than 15. Since the inner query in Q2 only selects tuples whose `DEPT_ID` is 10, the outer predicates of both queries Q1 and Q2 are equivalent. The algebraic representation of these queries are as follows:

$$\text{Q1} : [\texttt{t.DEPT\_ID} = 10] \times [\texttt{t.DEPT\_ID} + 5 > \texttt{t.EMP\_ID}] \times \texttt{EMP(t)}$$

$$\text{Q2} : [\texttt{t.DEPT\_ID} = 10] \times [15 > \texttt{t.EMP\_ID}] \times \texttt{EMP(t)}$$

Each algebraic expression is a function that returns the number of times a given tuple `t` is present in the output table. $\times$ represents the arithmetic multiplication operation. For example, Q1 returns the number of times a tuple `t` in `EMP` is returned. Each predicate is a function that emits 1 when it holds, and returns 0 otherwise. For example, $[\texttt{t.DEPT\_ID} = 10]$ returns one when $\texttt{t.DEPT\_ID} = 10$. `EMP(t)` is function that returns the number of times t is present in `EMP`.

Algebraic approaches are unable to prove the equivalence of these expressions since they do not model the semantics of arithmetic expressions. The automated proof assistant must infer that the two predicates $[\texttt{t.DEPT\_ID} + 5 > \texttt{t.EMP\_ID}]$ and $[15 > \texttt{t.EMP\_ID}]$ are equivalent when the predicate $[\texttt{t.DEPT\_ID} = 10]$ holds. It is challenging for a *proof assistant* to infer this fact due to the inherent complexity of arithmetic expressions. For instance, the predicate $[\texttt{t.DEPT\_ID} + 5 > \texttt{t.EMP\_ID}]$ can be rewritten as $[\texttt{t.DEPT\_ID} > \texttt{t.EMP\_ID} - 5]$.

EXAMPLE 2. THREE-VALUED LOGIC:

```
Q1: SELECT EMP_ID FROM EMP
    WHERE EMP_ID = 10 AND EMP_ID IS NOT NULL;

Q2: SELECT EMP_ID FROM EMP
    WHERE EMP_ID = 10;
```

Q1 selects all tuples from `EMP` whose `EMP_ID` is 10 and is not `NULL`. Q2 retrieves employees whose `EMP_ID` is 10. With three-valued logic, `EMP_ID` is not `NULL` when it is equal to 10. So these two queries are equivalent. The algebraic representation of these queries are as follows:

$$\text{Q1} : [\texttt{t.EMP\_ID} = 10] \times [\texttt{NOT-NULL(t.EMP\_ID)}] \times \texttt{EMP(t)}$$

$$\text{Q2} : [\texttt{t.EMP\_ID} = 10] \times \texttt{EMP(t)}$$

Algebraic approaches are unable to prove the equivalence of these queries since they do not model the semantics of three-valued logic [47]. The proof assistant must infer that `t.EMP_ID` cannot be `NULL` if the predicate $[\texttt{t.EMP\_ID} = 10]$ holds. It is challenging for the proof assistant to support this kind of inference due to the inherent complexity of three-valued logic. For instance, `t.EMP_ID` *can* be `NULL` even if the predicate $[\texttt{TRUE OR t.EMP\_ID}]$ holds.

EXAMPLE 3. LEFT OUTER JOIN:

```
Q1: SELECT EMP.EMP_ID, DEPT.DEPT_NAME FROM
     EMP JOIN DEPT ON EMP.DEPT_ID = DEPT.DEPT_ID;

Q2: SELECT * FROM
    (SELECT EMP.EMP_ID, DEPT.DEPT_NAME FROM
      EMP LEFT OUTER JOIN DEPT
        ON EMP.DEPT_ID = DEPT.DEPT_ID
    ) WHERE DEPT.DEPT_ID IS NOT NULL;
```

Q1 retrieves the name and department of employees by performing an `INNER JOIN` of `EMP` and `DEPT` tables on `DEPT_ID`. Q2 is a nested query where the inner query selects the name and department of employees by performing a `LEFT OUTER JOIN` of `EMP` and `DEPT` tables on `DEPT_ID`. The outer query then applies a filter on the results of the inner query by retrieving those tuples whose `DEPT_ID` is not null. Application of the `NOT-NULL` predicate to the output tuples of `LEFT OUTER JOIN` eliminates all tuples from `DEPT_ID` is `NULL`. These queries are, therefore, equivalent. Algebraic approaches are, however, unable to prove the equivalence of these queries since they do not model the semantics of complex relational operators such as `LEFT OUTER JOIN`.

These examples highlight the limitations of algebraic approaches. First, they do not model the semantics of complex predicates (e.g., those containing arithmetic expressions). Next, they are unable to support three-valued logic. Lastly, they cannot model the semantics of certain SQL operators (e.g., aggregate functions and different types of `OUTER JOIN`). These limitations restrict the set of queries that can be supported by UDP's proof assistant [26].

2.2 Symbolic Representation-Based Approach

We propose to address these limitations of algebraic approaches using an alternate approach based on SR. With this approach, we represent tuples in input tables using symbolic tuples. We construct these symbolic tuples using a collection of *symbolic variables* that represent an arbitrary tuple. The SR-based approach models the semantics of SQL queries using FOL formulae. It enables the usage of SMT solvers to determine QE by verifying the relationship between the SR of the given queries under *set* semantics [3].

We can reduce the problem of verifying QE to that of verifying the *containment relationship* between those queries. Q1 *contains* Q2 if and only if for all valid input tuples, the tuples returned after executing Q2 on the input tuples are a subset of those returned after executing Q1 on the same set of input tuples. If Q1 contains Q2 and Q2 contains Q1, then they are equivalent. We will formalize these definitions in Section 3.

EXAMPLE 1. COMPLEX ARITHMETIC EXPRESSIONS: Consider the example with complex arithmetic expressions shown in Section 2.1. For each tuple returned by these queries, there exists a corresponding input tuple in `EMP` that satisfies the predicate. More generally, for `SELECT-PROJECT-JOIN` queries, each output tuple is derived from a finite set of tuples chosen from the input tables, and the size of this set can be determined for all valid inputs. Thus, we can symbolically represent an *arbitrary* output tuple with a finite number of symbolic tuples that represent arbitrary tuples from the associated input tables. For instance, the SR of queries Q1 and Q2 are as follows:

```
Q1: <COND1, COLS1, ASSIGN1>
COND1: (v3 = 10 and !n3) and
        ((v3 + 5 > v1) and (!n3 and !n1))
COLS1: {(v1,n1),(v2,n2),(v3,n3)}
ASSIGN1: ---

Q2: <COND2, COLS2, ASSIGN2>
COND2: (v3 = 10 and !n3) and ((15 > v1) and !n1)
COLS2: {(v1,n1),(v2,n2),(v3,n3)}
ASSIGN2: ---
```

Here, $\{(\texttt{v1}, \texttt{n1}), (\texttt{v2}, \texttt{n2}), (\texttt{v3}, \texttt{n3})\}$ represents an arbitrary input tuple in `EMP`. Each pair of symbolic variables represents a column of the tuple in `EMP`. For example, $(\texttt{v1}, \texttt{n1})$ denotes `EMP_ID` in this symbolic tuple. While `v1` represents the *value* of `EMP_ID`, the boolean symbolic variable `n1` indicates if `EMP_ID` is `NULL`. This symbolic tuple represents an arbitrary input tuple in `EMP`. For each tuple returned by Q1 and Q2, there exists one input tuple in `EMP`.

COND_1 and COND_2 are FOL formulae that represent the constraints that the `EMP` tuple must satisfy for it to be returned by Q1 and Q2, respectively. For instance, the formula $(\texttt{v3} = 10) \,\&\&\, (!\texttt{n3})$, which is a part of COND_1, encodes the semantics of the predicate `DEPT_ID = 10` in Q1. It is satisfied only when the value of `DEPT_ID` in the tuple equals 10 and it is not `NULL`. COLS_1 and COLS_2 are the symbolic tuples returned by Q1 and Q2 when the conditions COND_1

---

and $COND_2$ are satisfied, respectively. Since Q1 and Q2 only filter out tuples in EMP and do not modify them, $COLS_1$ and $COLS_2$ are set to be the input symbolic tuple. Lastly, we use $ASSIGN_1$ and $ASSIGN_2$ to specify relational constraints between symbolic variables while handling complex SQL operators, such as aggregate functions. We do not set these constraints in this example. We defer a discussion on how to derive the SR of a query to Section 3.

For determining QE, we must prove that Q1 and Q2 contain each other. To show that Q1 contains Q2, we must prove two properties: (1) Every tuple in EMP returned by Q2 is also returned by Q1. In other words, if $COND_2$ is satisfied, then $COND_1$ also holds. (2) If a given tuple in EMP is returned by both queries, then they must emit the same output tuple. In other words, the symbolic tuple $COLS_2$ is equivalent to $COLS_1$ when the conditions $COND_1$ and $COND_2$ are met. In this example, the latter condition trivially holds since neither query modifies the input symbolic tuple. More generally, we use SMT solvers to verify these two properties between the SR of queries. We adopt the same technique to determine if Q2 contains Q1, and thereby conclude if they are equivalent. In this manner, the SR-based approach determines equivalence of queries with complex arithmetic expressions.

EXAMPLE 4. AGGREGATE FUNCTIONS:
```
Q1: SELECT COUNT(*) FROM
        (SELECT * FROM
          (SELECT * FROM EMP WHERE DEPT_ID = 10) AS T
        WHERE T.DEPT_ID + 5 > T.EMP_ID);

Q2: SELECT COUNT(*) FROM
        (SELECT * FROM
          (SELECT * FROM EMP WHERE DEPT_ID = 10) AS T
        WHERE 15 > T.EMP_ID);
```

Q1 and Q2 calculate the number of employees that satisfy certain predicates. As shown in Section 2.1, these predicates are equivalent. Algebraic approaches are, however, unable to prove the equivalence of these queries with aggregate functions since they do not model the semantics of arithmetic expressions. The SR of queries Q1 and Q2 are as follows:

```
Q1: <COND1, COLS1, ASSIGN1>
COND1: (v3 = 10 and !n3) and
       (v3 + 5 > v1) and (!n3 and !n1) )
COLS1: {(v4,n4)}
ASSIGN1: ---

Q2: <COND2, COLS2, ASSIGN2>
COND2: (v3 = 10 and !n3) and ((15 > v1) and !n1 )
COLS2: {(v5,n5)}
ASSIGN2: ---
```

Unlike SELECT-PROJECT-JOIN queries, each output tuple of these queries is *not* derived from a finite set of tuples in EMP. Furthermore, the size of this set of input tuples cannot be statically determined across all valid inputs. Thus, we cannot build the SR of Q1 and Q2 using the same input symbolic tuple in EMP.

We address this problem by introducing a two pairs of *independent symbolic variables* $(v4, n4)$ and $(v5, n5)$ to represent the values of the aggregate function COUNT(EMP_ID) returned by Q1 and Q2. We set $COLS_1$ and $COLS_2$ to these pairs, respectively. These symbolic variables do not depend on each other and can take up arbitrary values.

To prove the equivalence of Q1 and Q2, we need to prove two properties between their SR as we discussed in Section 2.2. For the first property, we use the same approach as in the previous example. However, we cannot do so for the second property (i.e., $COLS_1$ is equivalent to $COLS_2$ when $COND_1$ and $COND_2$ hold). This is because $(v4, n4)$ and $(v5, n5)$ are independent variables that have

no relationship with each other. For proving the second property, we formulate a set of rules to derive *relational constraints* for the given pair of queries. We defer a discussion on how we derive relational constraints to Section 4.

In this example, $(v4, n4)$ and $(v5, n5)$, the two pairs of independent symbolic variables representing aggregate values, are equivalent if and only if: (1) their sub-queries are equivalent, (2) the set of expressions in their GROUP BY clauses are equivalent, and (3) they use the same aggregate function. We leverage an SR-based approach to verify these constraints. If all these conditions are met, we verify the equivalence of $COLS_1$ and $COLS_2$ under these additional relational constraints that encode the equivalence of aggregate values:

$$(v4 = v5) \,\&\&\, (n4 = n5)$$

along with the $COND_1$ and $COND_2$ constraints. In this manner, the SR-based approach models the semantics of SQL operators when coupled with complex predicates.

2.3 SMT Solvers

We now present a brief overview of SMT solvers [32]. An SMT solver is a tool that decides if a given FOL formula has a solution (i.e., a collection of values that satisfy the formula). If the formula is satisfiable, then the solver returns a *model* of variables that meet the constraints in the formula. For example, when we feed in the formula $[x > 0] \,\&\&\, [x < 5]$ to the SMT solver, it determines that this formula is satisfiable (e.g., $x = 1$ is a solution). However, the solver decides that the formula $[x > 10] \,\&\&\, [x < 5]$ is not satisfiable since there is no value of x that satisfies these constraints.

To verify the properties of SR of queries, we first encode these properties as FOL formulae, and then use the SMT solver to determine the satisfiability of these constraints.

EXAMPLE 5. SMT SOLVER: Consider the queries in Example 1 (Section 2.2). We leverage the SMT solver to verify that $COND_1$ implies $COND_2$, and that $COLS_2$ is equivalent $COLS_2$ under $COND_1$ and $COND_2$ conditions.

1: To verify that $COND_1$ implies $COND_2$, We feed in these constraints to the SMT solver:

$$COND_1 \wedge \neg COND_2$$

The solver determines that this formula is not satisfiable, which implies that there is no counterexample to the fact that $COND_1$ implies $COND_2$. Thus, $COND_1 \implies COND_2$.

2: To verify that $COLS_1$ is equivalent $COLS_2$ under the $COND_1$ and $COND_2$ conditions, we feed in these constraints to the solver:

$$(COND_1 \wedge COND_2) \wedge \neg(COLS_1 = COLS_2)$$

The SMT solver decides that this formula is not satisfiable, thus proving the property the tuples returned by Q1 and Q2 are equivalent. Thus, Q1 contains Q2.

3. SPJ QUERIES

In this section, we discuss how EQUITAS determines the equivalence and containment relationships between queries. We begin with a formal definition of query equivalence in Section 3.1. We then describe how EQUITAS verifies the relationship between SRs of tables returned by a pair of queries in Section 3.2. We then discuss how to construct an SR for SELECT-PROJECT-JOIN queries in Section 3.3. We present techniques for handling three-valued logic, user-defined functions, complex predicates, complicated expressions, and arithmetic operations in Section 3.4.

3.1 Query Equivalence

We define the QE relationship in terms of the query containment relationship. We now formally define the latter relationship.

DEFINITION 1. CONTAINMENT: *Given a pair of SQL queries Q1 and Q2, Q1 contains Q2 if and only if, for all valid inputs T, T_1 and T_2 are the output tables of executing Q1 and Q2 on T respectively, for each tuple in T_2 is present in T_1. We denote this containment relationship by Q1 $\subseteq$ Q2.*

This definition is under set semantics. In other words, if tuple x appears three times in T_2 and only once in T_1, Q1 still contains Q2 based on our definition. We next define the QE relationship.

DEFINITION 2. EQUIVALENCE: *Two queries are semantically equivalent if and only if they contain each other. Q1 is* equivalent *to Q2, if and only if Q1 $\subseteq$ Q2 and Q2 $\subseteq$ Q1. We denote this equivalence relationship by Q1 $\equiv$ Q2.*

Since we define the containment relationship under set semantics, this definition is also under set semantics (rather than bag semantics). Having formalized the problem of determining the equivalence relationship between a pair of SQL queries, we next describe how to automatically deduce that a given pair of SQL queries are equivalent under set semantics.

3.2 Verifying the Equivalence of Queries

We begin by defining the SR of a table constructed by executing an SQL query. We will discuss how to determine the relationship between queries using the representations of tables that they return in Section 3.2.1. The *SR* of a query Q is a tuple:

$$\langle \text{COND}, \vec{\text{COLS}}, \text{ASSIGN} \rangle$$

COND is an FOL formula that represents the constraint(s) that must be satisfied for the symbolic tuple $\vec{\text{COLS}}$ to be valid (i.e., a condition that an arbitrary tuple needs to satisfied in the output table).
COLS is a vector of pairs of FOL formulae that represent an arbitrary tuple that can be returned by Q. Each element (VAL, IS-NULL) $\in$ $\vec{\text{COLS}}$ represents a column, where VAL constrains the value of the column and IS-NULL constrains whether the column is null.
ASSIGN is another formula that models the relationship between the symbolic variables used in COND and $\vec{\text{COLS}}$. We use this formula to handle complex SQL features. For example, the SR of a pair of queries with arithmetic expressions is shown in Example 1 (Section 2.2).

We observe that for SELECT-PROJECT-JOIN queries, an arbitrary tuple $\vec{\text{COLS}}$ in the output table is derived from a *finite* number of tuples present in the input tables referred to in Q. In Section 4, we discuss how EQUITAS handles queries that contain aggregate functions and different types of OUTER JOIN.

3.2.1 Verifying Equivalence

Given the definition of QE in Definition 2, to verify the equivalence of two queries Q1 and Q2, EQUITAS needs to assert that they have a containment relationship. We next describe how to prove that Q1 contains Q2. To prove that Q1 contains Q2, EQUITAS must verify that all tuples that are in the output table of Q2 are also present in that of Q1. This is equivalent to proving that for an arbitrary tuple T in Q2's output table, there exists a corresponding tuple in Q1's output table. EQUITAS attempts to prove that there exists a tuple in Q1's output table, which is derived from the same set of tuples in the input tables, that is equivalent to t. This is sufficient to show that Q1 contains Q2.

EQUITAS validates that Q1 contains Q2 in two steps. It first constructs the SR of the output tables obtained by running the queries. It then verifies two formal properties between these representations using a decision procedure. We next describe these two steps.

EQUITAS first attempts to show that for an arbitrary tuple T in the output table of Q2, the tuple derived by executing Q1 on the same set of input tuples is equivalent to T. For this proof, EQUITAS uses the Construct procedure to build the symbolic representation of their output tables: (COND_1, $\vec{\text{COLS}}_1$, ASSIGN_1) and (COND_2, $\vec{\text{COLS}}_2$, ASSIGN_2) respectively. We defer a discussion of the Construct procedure to Section 3.3. Since EQUITAS only needs to consider tuples that are derived from the same set of input tuples and the size of this set is bounded[4], the SR of output tables of Q1 and Q2 share the same set of variables.

To show that Q1 contains Q2, EQUITAS must prove two properties between the SR of the output tables of these queries.
1: When a tuple $\vec{\text{COLS}}_2$ exists in the output table of Q2, a corresponding tuple constructed from the same set of input also exists in Q1's output table. EQUITAS proves this property by showing that whenever COND_2 is satisfied, COND_1 is also met. This property is formalized as the constraint: $\text{COND}_1 \implies \text{COND}_2$.
2: When both tuples $\vec{\text{COLS}}_2$ and $\vec{\text{COLS}}_1$ are present in their respective output tables, they are equivalent. This property is formalized as the constraint: $(\text{COND}_1 \wedge \text{COND}_2) \implies (\vec{\text{COLS}}_1 = \vec{\text{COLS}}_2)$.

EQUITAS checks these two properties using an SMT solver.
1: For the first property, $\text{COND}_2 \implies \text{COND}_1$, EQUITAS feeds this formula to the solver: $(\text{ASSIGN}_1 \wedge \text{ASSIGN}_2) \wedge (\text{COND}_2 \wedge \neg \text{COND}_1)$. If the solver determines that the formula cannot be satisfied, that shows that there exists no input tuple T that satisfies COND_2 while not meeting COND_1. In other words, $\text{COND}_2 \implies \text{COND}_1$ within the context of ASSIGN_1 and ASSIGN_2 for a given input tuple.
2: For the second property, EQUITAS feeds this formula to the solver: $(\text{ASSIGN}_1 \wedge \text{ASSIGN}_2) \wedge (\text{COND}_2 \wedge \text{COND}_1) \wedge \neg (\vec{\text{COLS}}_1 = \vec{\text{COLS}}_2)$. If the solver determines that the formula cannot be satisfied, that demonstrates that there exists no input tuple T for which the queries Q2 and Q1 return different output tuples when both conditions are satisfied. This implies that given an arbitrary input tuple T, Q1 and Q2 return the same tuple in their output tables.

To summarize, EQUITAS determines whether Q1 contains Q2 by validating the properties between the SR of their output tables using the SMT solver. It uses the same approach to determine if Q2 contains Q1. It finally combines the results of these containment relationship checks to prove the equivalence of Q1 and Q2.

3.3 Symbolic Representation Construction

We now describe a recursive algorithm for constructing the SR of the output table of a query. We begin by presenting the Construct algorithm that supports SELECT-PROJECT-JOIN queries. We will extend this algorithm to handle more advanced SQL features in Section 4. Table 1 presents an overview of the SR of different types of SQL queries and highlights the fields modified.

As shown in Algorithm 1, the inputs for the Construct procedure include the query Q and the schemata of its input tables $\mathcal{S}$. The Construct procedure synthesizes different structures depending on the query type.

SCAN: If the given query Q is a SELECT operator on table T, then Construct creates a set of symbolic variables to represent a tuple in T based on the table's schema (T-SCHEMA). This sub-procedure is denoted by Init. It sets the COND and ASSIGN constraints to TRUE. The reasons for this are twofold. First, the SCAN operator returns all

---

[4] The size of this set can be arbitrarily large for queries with aggregate functions and different types of OUTER JOIN.

Algorithm 1: Procedure for constructing the SR of a given Q and the schemata of its input tables $\mathcal{S}$.

Input : Query Q, Schemata of its input tables schemas $\mathcal{S}$
Output: SR of the output table returned by Q

```
1  Procedure Construct(Q, S)
2  │  switch Q do
3  │  │  case Scan(n)
4  │  │  │  return (TRUE, Init(T-SCHEMA(S[n])), TRUE)
5  │  │  end
6  │  │  case Filter(pₛ, Qₛ)
7  │  │  │  (CONDₛ, CŌLSₛ, ASSIGNₛ) ← Construct(Qₛ, S)
8  │  │  │  COND ← CONDₛ ∧ ConstructPred(pₛ, CŌLSₛ)
9  │  │  │  return (COND, CŌLSₛ, ASSIGNₛ)
10 │  │  end
11 │  │  case Proj(ē, Qₛ)
12 │  │  │  (CONDₛ, CŌLSₛ, ASSIGNₛ) ← Construct(Qₛ, S)
13 │  │  │  CŌLS ← ConstExpr′(ē, CŌLSₛ)
14 │  │  │  return (CONDₛ, CŌLS, ASSIGNₛ)
15 │  │  end
16 │  │  case Join(Inner, k⃗₁ = k⃗₂, Q1, Q2)
17 │  │  │  (COND₁, CŌLS₁, ASSIGN₁) ← Construct(Q1, S)
18 │  │  │  (COND₂, CŌLS₂, ASSIGN₂) ← Construct(Q2, S)
19 │  │  │  CŌLS ← CŌLS₁ : CŌLS₂
   │  │  │  Key ← ConstructPred(k⃗₁ = k⃗₂, CŌLS)
20 │  │  │  COND ← COND₁ ∧ COND₂ ∧ Key
21 │  │  │  ASSIGN ← ASSIGN₁ ∧ ASSIGN₂
22 │  │  │  return (COND, CŌLS, ASSIGN)
23 │  │  end
24 │  endsw
```

Table 1: SR of SQL queries - ✓ indicates that the particular field is reconstructed instead of being inherited from those in the SR of constituent sub-queries.

SQL Query	COND	CŌLS	ASSIGN
SELECT			
Filter	✓		
PROJECT		✓	
INNER JOIN	✓	✓	
OUTER JOIN	✓	✓	✓
Aggregate		✓	

tuples in T. Second, since SCAN is a trivial constructor, there are no additional assignment constraints for constructing the output tuples.

FILTER: If the given query Q is a SELECT operator with a filter, then Construct represents the SELECT operator as a sub-query Q_s and applies the filter on the results of Q_s. It first recurses onto the sub-query and creates an SR of Q_s (COND$_s$, CŌLS$_s$, ASSIGN$_s$). We denote the filter by Filter(p_s, Q_s). This indicates that this operation consists of applying the predicate p_s on the results of Q_s. Construct creates an SR of the filter by invoking the ConstructPred procedure on p_s and the symbolic tuple CŌLS$_s$. We defer a discussion of the ConstructPred procedure to Section 3.4.2. It then derives COND by combining the SR of the filter with COND$_s$ using a conjunction operator. Lastly, it returns (COND, CŌLS$_s$, ASSIGN$_s$) as the representation of Q. As shown in Table 1, only the condition formula differs between Q and Q_s. This is because p_s filters out a subset of tuples in Q_s and otherwise does not alter the semantics of Q_s.

PROJECTION: Similar to the filter operator, if the given query Q is a PROJECT operator, then Construct represents the SELECT operator as a sub-query Q_s and applies the projection on the results of Q_s. It first recurses onto the sub-query Q_s and creates its symbolic representation. We denote the projection operator by Proj($\vec{e}$, Q_s). The Construct procedure materializes an SR of the projected tuple CŌLS by invoking the ConstExpr′ procedure on the columns in CŌLS$_s$. This ConstExpr′ procedure applies a set of transformations using a vector of expressions $\vec{e}$. Internally, it calls the ConstExpr procedure on each expression in $\vec{e}$ on the symbolic tuple CŌLS$_s$ and then collects the returned variables to materialize CŌLS.

Given a symbolic tuple and an expression e, the ConstExpr procedure applies the transformation associated with e on the tuple. We defer a description of the ConstExpr procedure to Section 3.4.1. Lastly, Construct returns (COND$_s$, CŌLS, ASSIGN$_s$) as the representation of Q. Since the PROJECT operator only applies transformations on the columns of the input tuples, the COND$_s$ and ASSIGN$_s$ remain unchanged, as shown in Table 1.

INNER JOIN: If the given query Q is a JOIN, then Construct recurses into two sub-queries Q1 and Q2 that represent the tables that are being joined. We denote the JOIN operator by Inner Join($\vec{k_1} = \vec{k_2}$, Q1, Q2). After deriving the SR of the sub-queries (COND$_1$, CŌLS$_1$, ASSIGN$_1$) and (COND$_1$, CŌLS$_2$, ASSIGN$_2$), it constructs the output symbolic tuple CŌLS by concatenating CŌLS$_1$ and CŌLS$_2$. It combines the COND$_1$ and COND$_2$ constraints along with the SR of the join predicate ($\vec{k_1} = \vec{k_2}$) to derive COND. Similarly, it coalesces the ASSIGN$_1$ and ASSIGN$_2$ constraints using the conjunction operator to materialize ASSIGN. In this manner, the JOIN operator is realized by combining the output tuples of the sub-queries Q1 and Q2 using the join predicate. We note that Construct relies on ConstructPred procedure to encode filter and join predicates.

3.4 Encoding Expressions and Predicates

We next describe how EQUITAS encodes expressions, predicates, and the CASE statement. We begin with a description of how EQUITAS represents expressions, including arithmetic operations and user-defined functions (UDFs), in Section 3.4.1. We then discuss how EQUITAS encodes predicates in Section 3.4.2. In particular, we detail how it uses three-valued logic for supporting NULL. Lastly, we describe how we combine these techniques to handle the CASE statement in Section 3.4.3.

3.4.1 Expression

We define the syntax of an expression as follows:

e ::= Column i|Const v|NULL|Bin e op e|Fun N ($\vec{e}$)
op ::= +| − | × | ÷ |mod

An expression can be: (1) a reference to a column, (2) a constant value, (3) a NULL value, (4) a binary arithmetic operator combining the values of two expressions, or (5) a UDF operating on a vector of expressions.

Algorithm 2 presents the ConstExpr procedure for deriving the SR of an expression e based on the input symbolic tuple CŌLS. EQUITAS represents an expression as a pair of FOL formulae (VAL, IS-NULL). Here, the first formula denotes the value and the second one IS-NULL indicates if the value is NULL. The input symbolic tuple CŌLS, that is referred to by e, is a vector of pairs of FOL formulae, as detailed in Section 3.2. The ConstExpr procedure synthesizes different structures depending on the expression type.

COLUMN REFERENCE: If e is a reference to the ith column in the symbolic tuple, then ConstExpr returns the corresponding element in CŌLS.

CONSTANT: If e is a constant value Const v, then ConstExpr returns (v, FALSE) since the v is not NULL. In contrast, if e is NULL, then it emits (0, TRUE). EQUITAS sets the type of 0 to be that of the associated column.

BINARY ARITHMETIC OPERATOR: If e contains a binary arithmetic operator combining two expressions Bin e_1 op e_2, then

Algorithm 2: Procedure for deriving the SR of an expression e based on the input symbolic tuple $\vec{\text{COLS}}$.

Input : Expression e, Input symbolic tuple $\vec{\text{COLS}}$
Output: SR of e
1 **Procedure** *ConstExpr*(e, $\vec{\text{COLS}}$)
2   **switch** e **do**
3   **case** *Column i* **return** $\vec{\text{COLS}}[i]$;
4   **case** *Const v* **return** (v, FALSE) ;
5   **case** NULL **return** $(0, \text{TRUE})$;
6   **case** *Bin e_1 op e_2*
7        $(\text{VAL}_1, \text{IS-NULL}_1) \leftarrow \text{ConstExpr}(e_1, \vec{\text{COLS}})$
8        $(\text{VAL}_2, \text{IS-NULL}_2) \leftarrow \text{ConstExpr}(e_2, \vec{\text{COLS}})$
9        $\text{VAL} \leftarrow \text{ConstBin}(op, \text{VAL}_1, \text{VAL}_2)$
10   **return** $(\text{VAL}, \text{IS-NULL}_1 \vee \text{IS-NULL}_2)$
11   **end**
12   **case** *Fun n $(\vec{e_1})$*
13        $\text{sym-}\vec{e}_1 \leftarrow \text{ConstExpr}'(\vec{e_1}, \vec{\text{COLS}})$
14        $(\text{F-VAL}, \text{F-NULL}) \leftarrow \text{GetFun}(n)$
15   **return** $(\text{F-VAL}(\text{sym-}\vec{e}_1), \text{F-NULL}(\text{sym-}\vec{e}_1))$
16   **end**
17   **endsw**

ConstExpr recursively derives the representations of e_1 and e_2. It then invokes the ConstBin procedure to construct an FOL formula that combines VAL_1 and VAL_2 using the binary operator *op*. ConstBin handles addition and subtraction operations by appending the SR of e_1 and e_2 with the corresponding binary operator. ConstBin supports multiplication, division, and modulo operations in two ways depending on whether both VAL_1 and VAL_2 are variables or not. In the former case, it represents the operation as an uninterpreted function since the problem of deciding the satisfiability of a quantifier-free non-linear integer arithmetic formula is undecidable[43]. EQUITAS can decide the equivalence of formula containing uninterpreted functions only when the operands of these functions are equal. For instance, for a non-linear operator $\times$, EQUITAS determines that $(a \times b) = (c \times d)$ only when $a = c$ and $b = d$. When either VAL_1 or VAL_2 is not a variable, then ConstBin derives a formula with the corresponding operator.

USER-DEFINED FUNCTION: If e is a UDF Fun $F(\vec{e_1})$ that operates on a vector of expressions $\vec{e_1}$, then ConstExpr first invokes the ConstExpr' procedure on $\vec{e_1}$ to derive the SR of all the expressions in the vector (sym-$\vec{e}_1$). It then obtains the representation of function F using the GetFun procedure which returns a pair of uninterpreted functions F-VAL and F-NULL. While the former function models the value computed by F, the latter function represents if F returns NULL values.

GetFun disambiguates functions based on names. Given a function named F, it always returns the same pair of uninterpreted functions. EQUITAS can decide the equivalence of these uninterpreted functions if and only if their arguments take the same values. This encoding captures the semantics of deterministic UDFs that can contain arbitrary logic. EQUITAS does not support non-deterministic UDFs. However, it can be extended to allow users to define properties of UDFs. ConstExpr applies the pair of uninterpreted functions F-VAL and F-NULL on the UDF's inputs (sym-$\vec{e}_1$) to derive the SR of e.

3.4.2 Predicate

We define the syntax of a predicate as follows:

p ::= BinE e cp e|BinL p logic p|**Not** p|**IsNull** e
cp ::= $>$ | $<$ | $=$ | $\leq$ | $\geq$
logic ::= AND| OR

A predicate can be: **(1)** a comparison of two expressions, **(2)** a combination of two predicates using Boolean logic, **(3)** negation of

Algorithm 3: Procedure for deriving the SR of a predicate p that represents its satisfiability when evaluated on an input tuple $\vec{\text{COLS}}$.

Input : Predicate p, Input symbolic tuple $\vec{\text{COLS}}$
Output: SR of p
1 **Procedure** *ConstPred*(p, $\vec{\text{COLS}}$)
2   **Procedure** *ConstPredAux*(p, $\vec{\text{COLS}}$)
3   **switch** p **do**
4   **case** *BinE e_1 cp e_2*
5        $(\text{VAL}_1, \text{IS-NULL}_1) \leftarrow \text{ConstExpr}(e_1, \vec{\text{COLS}})$
6        $(\text{VAL}_2, \text{IS-NULL}_2) \leftarrow \text{ConstExpr}(e_2, \vec{\text{COLS}})$
7        $\text{VAL} \leftarrow \text{ConstComp}(\text{VAL}_1, \text{VAL}_2, \text{cp})$
8   **return** $(\text{VAL}, \text{IS-NULL}_1 \vee \text{IS-NULL}_2)$
9   **end**
10   **case** *BinL p_1 l_1 p_2*
11        $(\text{VAL}_1, \text{IS-NULL}_1) \leftarrow \text{ConstPredAux}(p_1, \vec{\text{COLS}})$
12        $(\text{VAL}_2, \text{IS-NULL}_2) \leftarrow \text{ConstPredAux}(p_2, \vec{\text{COLS}})$
13        $(\text{VAL}, \text{IS-NULL}) \leftarrow$
        $\text{ConstLogic}(l_1, \text{VAL}_1, \text{VAL}_2, \text{IS-NULL}_1, \text{IS-NULL}_2)$
14   **return** $(\text{VAL}, \text{IS-NULL})$
15   **end**
16   **case** *Not p_1*
17        $(\text{VAL}_1, \text{IS-NULL}_1) \leftarrow \text{ConstPredAux}(p_1, \vec{\text{COLS}})$
18   **return** $(\neg\text{VAL}_1, \text{IS-NULL}_1)$
19   **end**
20   **case** *IsNull e*
21        $(\text{VAL}_1, \text{IS-NULL}_1) \leftarrow \text{ConstExpr}(e, \vec{\text{COLS}})$
22   **return** $(\text{IS-NULL}_1, \text{FALSE})$
23   **end**
24   **endsw**
25    $(\text{VAL}, \text{IS-NULL}) \leftarrow \text{ConstPredAux}(e, \vec{\text{COLS}})$
26   **return** $(\text{VAL} \wedge \neg\text{IS-NULL})$

another predicate, or **(4)** a Boolean representing if an expression is NULL or not.

Algorithm 3 presents the ConstPred procedure that derives a an FOL formula to represent the satisfiability of a given predicate p when evaluated on an input symbolic tuple $\vec{\text{COLS}}$. ConstPred internally invokes an auxiliary ConstPredAux procedure that constructs a pair of FOL formulae. This pair represents the result of evaluating p on $\vec{\text{COLS}}$. While the first formula denotes the boolean value of the predicate, the second one indicates if the predicate is NULL (i.e., UNKNOWN). EQUITAS leverages the latter information to support three-valued logic [47]. ConstPredAux synthesizes different pairs of FOL formulae depending on the type of the predicate.

EXPRESSIONS: If p compares two expressions, then procedure ConstPredAux first obtains the representations of e_1 and e_2 using ConstExpr. It then invokes the ConstComp procedure on the comparison operator cp and the SR of VAL_1 and VAL_2. ConstComp derives a Boolean formula VAL to represent the comparison of VAL_1 and VAL_2 using cp. Lastly, it returns $(\text{VAL}, \text{IS-NULL}_1 \vee \text{IS-NULL}_2)$ as the SR of p. It uses the disjunction operator to combine IS-NULL_1 and IS-NULL_2 because if either of these expressions is NULL, then the value of p is unknown.

BINARY LOGICAL OPERATOR: If p is a combination of two predicates using a binary logic, then ConstPredAux first recursively derives the SR of predicates p_1 and p_2. The base cases of this recursive procedure are the non-recursive rules for comparing expressions and determining whether an expression is NULL or not. ConstPredAux then uses the auxiliary ConstLogic procedure to derive the SR of p by using the associated logical operator (AND, OR) to combine $(\text{VAL}_1, \text{IS-NULL}_1)$ and $(\text{VAL}_2, \text{IS-NULL}_2)$. ConstLogic employs three-valued logic to derive the SR of the combination of p_1 and p_2.

NEGATION: If p is the negation of another predicate p_1, then ConstPredAux first derives the SR of p_1. It returns the logical negation of VAL_1 and sets IS-NULL based on IS-NULL_1.

NULL: If p is a boolean predicate representing if an expression e_1 is NULL or not, then ConstPredAux invokes ConstExpr to obtain the SR of e_1. The value of p is given by the boolean Is-Null_1 that indicates if e_1 is NULL. Since it is impossible for the p to be NULL, ConstPredAux sets Is-Null to be false.

Lastly, we describe how ConstPred transforms the results obtained from its auxiliary ConstPredAux procedure. Given a predicate p, procedure ConstPredAux returns a pair of FOL formulae (Val, Is-Null). While the first formula represents the Boolean value of p, the second one indicates whether the predicate is NULL. By three-valued logic, ConstPred holds if and only if it is true and it is not unknown. Thus, ConstPred returns the conjunction of Val and the negation of Is-Null to represent the satisfiability of p.

3.4.3 Case Constructor

EQUITAS handles more complex features of SQL by leveraging the ConstExpr and ConstPred procedures presented in Sections 3.4.1 and 3.4.2. We next detail how it supports the CASE expression in this manner.

We define the syntax of the CASE expression as follows:

$$\text{CASE} := \text{WHEN } p_1 \ e_1; \ldots \text{WHEN } p_n \ e_n; \ \text{ELSE } e_d;$$

A CASE expression consists of a list of predicates $(p_1, \ldots, p_n)$. It returns one of multiple possible result sub-expressions $(e_1, \ldots, e_n)$ depending on the first predicate in the list that holds. If none of the predicates hold, it returns the final sub-expression (e_d). All of these expressions must have the same type.

Similar to other structures, EQUITAS creates a pair of symbolic variables (Val, Is-Null) to represent the CASE expression. Since the CASE expression may return any sub-expression, EQUITAS captures the relationship between the predicates and sub-expressions using an FOL formula (ASSIGN). EQUITAS combines this ASSIGN formula with that already present in the symbolic representation of query containing the CASE expression using a conjunction operator.

Given a CASE expression e_c, EQUITAS first uses the ConstExpr and ConstPred procedures to obtain the SR of the predicates and sub-expressions. The SR of e_c is then given by:

$$(p_1, (\text{Val}_1, \text{Is-Null}_1)); \ldots (p_n, (\text{Val}_n, \text{Is-Null}_n));$$

$$(\text{TRUE}, (\text{Val}_d, \text{Is-Null}_d)))$$

This representation captures the semantics of the CASE expression. If p_1 holds, then (Val, Is-Null) is given by $(\text{Val}_1, \text{Is-Null}_1)$. If all predicates prior to p_n do not hold and p_n holds, then (Val, Is-Null) is given by $(\text{Val}_n, \text{Is-Null}_n)$. EQUITAS models the relationship between e_c and (Val, Is-Null) as follows:

$$\bigvee_{i \leqslant n} [p_i \wedge \bigwedge_{s < i} \neg p_s \implies (\text{Val} = \text{Val}_i \wedge \text{Is-Null} = \text{Is-Null}_i)]$$

EXAMPLE 7. CASE: Consider the following query and its SR:

```sql
SELECT CASE
  WHEN EMPNO < 10 THEN DEPTNO + 1 ELSE DEPTNO END
FROM EMP;

COND: ---
COLS: {(v4,n4)}
ASSIGN: (v1 < 10 => (v4 = v3 +1 ) and (n4 = n3))
        or ((v1 >= 10) => (v4 = v3) and (n4 = n3))
```

In this example, (v1,n1),(v2,n2), and (v3,n3) represents a symbolic tuple from EMP table. Given the CASE expression, we represent the output column using new variables (v4,n4). ASSIGN encodes the relationship between (v4,n4) and (v3,n3) based on the conditions in the CASE expression.

Algorithm 4: Extended version of the Construct procedure that supports queries containing OUTER JOIN and aggregate functions.

Input : Query Q, Schemata of its input tables schemas $\mathcal{S}$
Output : SR of the output table returned by Q

1 **Procedure** *Construct*(Q, S)
2   **switch** Q **do**
3   **case** *Join*($Left$, $\vec{k_1} = \vec{k_2}$, Q1, Q2)
4        $(\text{COND}_1, \vec{\text{COLS}}_1, \text{ASSIGN}_1) \leftarrow$ Construct(Q1, S)
5        $(\text{COND}_2, \vec{\text{COLS}}_2, \text{ASSIGN}_2) \leftarrow$ Construct(Q2, S)
6        $Key \leftarrow$ ConstructPred($\vec{\text{COLS}}_1, \vec{\text{COLS}}_2, \vec{k_1} = \vec{k_2}$)
7        $(B, \text{COND}, \vec{\text{COLS}}) \leftarrow$ Fresh()
8        $cstr_1 \leftarrow$ Asg($\text{COND}_1, \text{COND}_2, Key, B, \text{COND}$)
9        $cstr_2 \leftarrow$ Asg($\vec{\text{COLS}}_1, \vec{\text{COLS}}_2, \vec{\text{COLS}}, B$)
10        $\text{ASSIGN} \leftarrow \text{ASSIGN}_1 \wedge \text{ASSIGN}_2 \wedge cstr_1 \wedge cstr_2$
11   **return** ($\text{COND}, \vec{\text{COLS}}, \text{ASSIGN}$)
12   **end**
13   **case** *Join*($Full$, $\vec{k_1} = \vec{k_2}$, Q1, Q2)
14   ......
15   **end**
16   **case** *Aggregate*($a\vec{g}g$, $\vec{g}$, Q_s)
17        $(\text{COND}_s, \vec{\text{COLS}}_s, \text{ASSIGN}_s) \leftarrow$ Construct(Q$_s$, S)
18        $\vec{\text{COLS}} \leftarrow$ Fresh($a\vec{g}g$)
19   **return** ($\text{COND}_s, \vec{\text{COLS}}, \text{ASSIGN}_s$)
20   **end**
21   ......
22   **endsw**

We next discuss how EQUITAS supports SQL queries with advanced features, such as aggregate functions and different types of OUTER JOIN.

4. BEYOND SPJ QUERIES

A distinctive feature of queries containing OUTER JOIN and aggregate functions is that, across all possible input tables, a tuple in the final output table is *not* derived from a fixed number of tuples from the input tables. This differentiates them from SELECT-PROJECT-JOIN queries that we covered in Section 3.3.

Consider the queries in Example 4 (Section 2.2). Q1 and Q2 calculate the number of employees in the EMP table that satisfy certain constraints. The output tuples depend on all the input tuples in EMP that meet these constraints.

Thus, there is no bounded number k such that for all possible input tables, tuples returned by Q1 and Q2 are guaranteed to be derived from k tuples in the input tables. Hence, EQUITAS cannot use the variables present in the symbolic tuples of the input tables to derive the SR of queries containing OUTER JOIN and aggregate functions. We next discuss how EQUITAS overcomes this challenge using independent variables in SRs (Section 4.1) and relational constraints for proving QE (Section 4.2).

4.1 Independent Variables

Algorithm 4 illustrates the extended version of the Construct procedure that supports queries containing OUTER JOIN and aggregate functions. The procedure for handling SELECT-PROJECT-JOIN queries, that we covered in Section 3.3, remains unchanged. We next discuss how EQUITAS supports other types of queries.

LEFT OUTER JOIN: If Q is $\langle$Join($Left$, $\vec{k_1} = \vec{k_2}$, Q1, Q2)$\rangle$, then Construct first recursively operates on the sub-queries Q1 and Q2 to derive their SR (COND_1, $\vec{\text{COLS}}_1$) and (COND_2, $\vec{\text{COLS}}_2$), respectively. Given the semantics of the left outer join, a tuple in the output table can be constructed either: (1) by concatenating a pair of tuples from left and right tables if they satisfy the join predicate ($\vec{k_1} = \vec{k_2}$), or (2) by concatenating a tuple from the left table with a vector of NULL

values in the shape of the right table when the left tuple does not match with any tuple in the right table.

We now explain why it is challenging to derive an SR that handles the latter case. Q1 and Q2 symbolically represent *one* arbitrary tuple in the left and right tables, respectively. In the former case, we only need to construct a one-to-one mapping between Q1 and Q2 using the join predicate. However, in the latter case, we need to derive an SR of *all* tuples in the right table that do not match Q1. It is not possible to encode this constraint using Q1 and Q2.

We address this challenge using *independent symbolic variables*. Construct creates an independent Boolean variable B that indicates if a given tuple in the left table has no match in the right table. Unlike SELECT-PROJECT-JOIN queries, Construct returns two different expressions for representing the output tuple depending on whether there is a match or not. Fresh() creates a vector of variables $\vec{COLS}$ to represent the output symbolic tuple and an associated Boolean condition variable COND.

Since the output tuple can be one of two expressions, Construct constructs $cstr_1$ to model the relationship between the new condition COND and the old conditions as follows:

$$(B \wedge (COND = COND_1)) \vee (\neg B \wedge (COND = (COND_1 \wedge COND_2 \wedge Key)))$$

$cstr_1$ indicates that if the Boolean variable B holds (i.e., there is no match for the left tuple in the right table), then COND only needs to satisfy the left condition $COND_1$. Otherwise, then COND is the same as the INNER JOIN condition.

Construct constructs $cstr_2$ to model the relationship between the new symbolic tuple $\vec{COLS}$ and the old symbolic tuples as follows:

$$(B \wedge (\vec{COLS} = \vec{COLS_1} : \vec{NULL})) \vee$$
$$(\neg B \wedge (\vec{COLS} = \vec{COLS_1} : \vec{COLS_2}))$$

$cstr_2$ indicates that if B holds, then $\vec{COLS}$ is given by the concatenation of $\vec{COLS_1}$ and a vector of NULL values in the shape of the right table. Otherwise, if B not holds, we construct the new symbolic tuple by appending the old tuples $\vec{COLS_1}$ and $\vec{COLS_2}$.

It derives ASSIGN by combining $ASSIGN_1$ and $ASSIGN_2$ with $cstr_1$ and $cstr_2$. (COND, $\vec{COLS}$, ASSIGN) represents the output of the LEFT OUTER JOIN query. Without loss of generality, a similar procedure is used for handling a RIGHT OUTER JOIN query.

FULL OUTER JOIN: If Q is $\langle Join(Full, \ \vec{k_1} = \vec{k_2}, \ Q1, \ Q2) \rangle$, the procedure used by Construct to derive the query's SR is similar to that used for a query containing a LEFT OUTER JOIN. The key difference is that EQUITAS must handle an additional case due to the semantics of FULL OUTER JOIN. The third scenario arises when the right tuple does not match with any tuple in the left table.

Construct supports these three scenarios by introducing two Boolean independent variables B_1 and B_2. While B_1 indicates whether there are no matches in Q2 for a given tuple in Q1, B_2 denotes whether there are no matches in Q1 for a given tuple in Q2. Besides this difference, the procedure is similar to that used for a query containing a LEFT OUTER JOIN.

AGGREGATE FUNCTIONS: If Q contains an aggregate function $\langle Aggregate(a\vec{g}g, \ \vec{g}, \ Q_s) \rangle$, then Construct first derives the SR of the sub-query Q1. The aggregation function performs a calculation on a set of values in the input tuples, and returns a single aggregate value (e.g., SUM). Aggregate functions may be used with the GROUP BY clause. In this case, the aggregation function returns a value for every group of tuples that have the same set of values for the columns listed in the GROUP BY clause.

Since the SR of Q1 can only represent *one* arbitrary output tuple, Construct creates a vector of variables $\vec{COLS}$ that correspond to the expressions containing aggregate functions in the select list denoted by $a\vec{g}g$. These variables indicate that these expressions

can take up arbitrary values. For every input tuple in Q1, there is a corresponding aggregate output tuple. Hence, as shown in Table 1, the condition formula for the aggregation function is the same as that of Q1 ($COND_s$). Thus, Construct returns (COND$_s$, $\vec{COLS}$, ASSIGN$_s$) as the SR of Q.

In this manner, EQUITAS introduces independent variables in the symbolic representations of queries containing OUTER JOIN and aggregate functions. For instance, Section 2.2 presents the SR of a pair of queries with aggregate functions. The two pairs of variables, (v4, n4) and (v5, n5), do not depend on the input tuples in EMP.

To determine the equivalence of queries containing independent variables, EQUITAS must deduce that these variables are equivalent. It derives *relational constraints* to model the relationship between independent symbolic variables. We next describe how EQUITAS uses inference rules to construct these relational constraints and thereby deduce the equivalence of independent variables.

4.2 Relational Constraints

EQUITAS contains a set of inference rules for deriving relational constraints. While verifying the relationship between the SR of two queries using the SMT solver, EQUITAS appends the relational constraints to determine the equivalence of independent variables.

LEFT OUTER JOIN: While comparing two queries:

$$Q1 : \langle Join(Left, \ \vec{k_1} = \vec{k_2}, \ Q3, \ Q4) \rangle$$
$$Q2 : \langle Join(Left, \ \vec{k_3} = \vec{k_4}, \ Q5, \ Q6) \rangle$$

EQUITAS uses Boolean independent variables B_1 and B_2 in the SR of Q1 and Q2, respectively. These variables indicate if there are no matches for a left tuple in the right table in the respective queries.

EQUITAS derives relational constraints between B_1 and B_2 using the following inference rule. If sub-queries Q5 contains Q3 and Q4 contains Q6, then B_1 implies B_2. This is because if Q5 contains Q3, then for an arbitrary tuple in Q3, there is a corresponding tuple in Q5. Since Q4 contains Q6, if there is no match for a Q5 tuple in Q3, then there will be no match for corresponding Q6 tuple in Q4. Thus, B_2 holds whenever B_1 holds (i.e., $B_1 \implies B_2$). EQUITAS uses the algorithm described in Algorithm 1 to determine the containment relationship between two queries. It follows a similar inference rule for handling FULL OUTER JOIN queries.

AGGREGATE FUNCTIONS: While comparing two queries:

$$Q1 : \langle Aggregate(a\vec{g}g_1, \ \vec{g_1}, \ Q3) \rangle$$
$$Q2 : \langle Aggregate(a\vec{g}g_2, \ \vec{g_2}, \ Q4) \rangle$$

EQUITAS uses two vectors of independent variables $\vec{COLS_1}$ and $\vec{COLS_2}$ in the SR of Q1 and Q2, respectively. These variables denote the expressions containing aggregate functions in the select lists of these queries.

EQUITAS derives relational constraints between $\vec{COLS_1}$ and $\vec{COLS_2}$ using the following inference rule. If the aggregate function is dependent on the cardinality of input tuples (e.g., COUNT), then the two symbolic tuples are equivalent if the sub-query Q3 is equivalent to Q4 under bag semantics. In this case, EQUITAS can verify the equivalence only if both sub-queries are SELECT-PROJECT-JOIN queries. In contrast, if the aggregate function is *not* dependent on the cardinality of input tuples (e.g., MIN and MAX), then the two symbolic tuples are equivalent if Q3 is equivalent to Q4 under set semantics. EQUITAS can verify this relationship for all types of sub-queries. Example 4 in Section 2.2 illustrates how we use relational constraints to prove QE.

5. SOUNDNESS AND COMPLETENESS

We now show that the procedure used in EQUITAS for checking the equivalence of two queries is *sound* under the set definition. We then prove that the decision procedure is *complete* for SELECT-PROJECT-JOIN queries that do not: (1) repeatedly scan the same table, or (2) have predicates whose satisfiability cannot be determined by the SMT solver.

THEOREM 1. SOUNDNESS: *Given two queries* Q1 *and* Q2, *if the SMT solver decides that the following formulae are unsatisfiable based on their SR:*

$$(1)(\texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2) \wedge (\texttt{COND}_2 \wedge \neg\texttt{COND}_1)$$
$$(2)(\texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2) \wedge (\texttt{COND}_2 \wedge \texttt{COND}_1)$$
$$\wedge \neg(\vec{\texttt{COLS}}_1 = \vec{\texttt{COLS}}_2)$$

then Q1 *contains* Q2.

PROOF. We prove this theorem using the method of contraposition. Suppose that Q1 does not contain Q2. By the definition of containment relationship in Section 3, there exists a set of valid input tables T such that there is an output tuple t obtained by executing Q1 on T that is *not* present in the output table derived by executing Q2 on T. Given the SR derived in EQUITAS, this implies that there exists a model (i.e., a set of concrete values for all symbolic variables) that satisfies the SR of Q1 but does not satisfy that of Q2. Thus, there exists a model that either satisfies the former formula: $(\texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2) \wedge (\texttt{COND}_2 \wedge \neg \texttt{COND}_1)$, or the latter formula: $(\texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2) \wedge (\texttt{COND}_2 \wedge \texttt{COND}_1) \wedge \neg (\vec{\texttt{COLS}}_1 = \vec{\texttt{COLS}}_2)$. In this case, the solver will not decide that both formulae are unsatisfiable. By contraposition, this proves that Q1 contains Q2. □

THEOREM 2. COMPLETENESS: *Given two* SELECT-PROJECT-JOIN *queries* Q1 *and* Q2 *that do not: (1) repeatedly scan the same table, or (2) have predicates whose satisfiability cannot be determined by the SMT solver, if* Q1 *contains* Q2, *then* EQUITAS *can prove that* Q1 *contains* Q2.

PROOF. We prove this theorem using the method of contraposition. Suppose that EQUITAS cannot prove that Q1 contains Q2. Since FOL formulae are decidable by the SMT solver [53], there exists a model M that satisfies either the former formula: $(\texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2) \wedge (\texttt{COND}_2 \wedge \neg \texttt{COND}_1)$, or the latter formula $(\texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2) \wedge (\texttt{COND}_2 \wedge \texttt{COND}_1) \wedge \neg (\vec{\texttt{COLS}}_1 = \vec{\texttt{COLS}}_2)$. Thus, we can construct a set of valid input tables T such that each input table only contains one tuple that matches the values in M. We require that the queries do not repeatedly scan the same input table and that the satisfiability of all the predicates can be determined by the SMT solver. Given these constraints: **(1):** if M satisfies the former formula, then executing Q1 and Q2 on T will return an empty and a non-empty output table, respectively. In this case, Q1 does not contain Q2. **(2):** If M satisfies the latter formula, then executing Q1 and Q2 on T will return different tuples, and the corresponding output tables will only contain those tuples. Again, in this case, Q1 does not contain Q2. Given these two scenarios, by contraposition, this proves that EQUITAS can prove that Q1 contains Q2. □

6. EVALUATION

In this section we describe our implementation and evaluation of EQUITAS. We begin with a description of our implementation in Section 6.1. Then, in Section 6.2, we report the results of a comparative analysis of EQUITAS against UDP [26], the state-of-the-art automated SQL QE verifier. We then examine the efficacy of EQUITAS in identifying overlap across production SQL queries in Section 6.3. We conclude with the limitations of the current implementation of EQUITAS in Section 6.4.
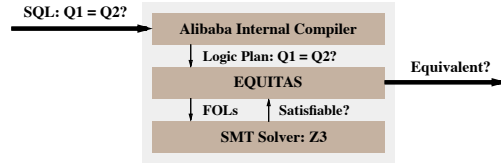

Figure 1: Query Equivalence Verification Pipeline - The pipeline for determining the equivalence of SQL queries. EQUITAS internally uses the Z3 SMT solver for determining the satisfiability of FOL formulae.

6.1 Implementation

We implemented EQUITAS as an SQL equivalence verification tool in Java. Figure 1 illustrates the pipeline for determining QE. Our implementation takes as input a pair of SQL queries to be checked (Q1 and Q2) and returns a decision (TRUE or FALSE) that indicates whether the given pair of queries are equivalent. The pipeline consists of three stages.

1. The first stage is a compiler that takes the given pair of SQL queries and converts them into logical query execution plans. Our implementation is tailored for an Alibaba internal SQL compiler.

2. The second stage consists of EQUITAS which determines the equivalence of the logical query execution plans emitted by the compiler. EQUITAS is written in 3660 lines of Java.

3. The third stage is an SMT solver that is leveraged by EQUITAS for determining the satisfiability of FOL formula. EQUITAS leverages the open-source Z3 SMT solver [12].

6.2 Comparison against UDP

We now compare the efficacy of EQUITAS against UDP [26]. To the best of our knowledge, UDP is the state-of-the-art automated SQL equivalence verifier. For this comparative analysis, we used these tools to prove the equivalence of real-world SQL queries.

We use queries contained in the test suite of Apache Calcite [3], an open-source query optimization framework. The reasons for using this benchmark are twofold. First, the CALCITE optimizer powers many widely-used data processing engines, including Apache Drill [4], Apache Flink [5], and others [6, 7, 8]. It contains 232 test cases, each of which contains a pair of SQL queries, a set of input tables, and the expected results. Every pair consists of a query and its optimized variant that is generated by CALCITE. The test suite validates the optimization rules in CALCITE and covers a wide range of SQL features [5]. Second, since UDP is evaluated on the queries contained in the CALCITE test suite [26], we can quantitatively and qualitatively compare the efficacy of these tools.

We send every pair of queries and the schemata of their input tables to EQUITAS and ask it to prove QE. We conducted this experiment on a commodity server (Intel Core i7-860 processor, 8 MB L3 Cache, and 16 GB RAM).

The results of this experiment are shown in Table 2. For comparative analysis against UDP, we present the results reported in the corresponding paper [26] [6]. The most notable observation from this experiment is that EQUITAS is able to effectively prove the equivalence of a larger set of query pairs (67 out of 232) compared to UDP (34 out of 232).

Among the 232 pairs of SQL queries, 91 pairs use SQL features that EQUITAS currently supports. The remaining pairs either: (1) contain SQL features that are not yet supported by EQUITAS (e.g., EXIST and CAST), or (2) cannot be compiled by the SQL compiler at

---

Table 2: Comparative analysis of EQUITAS and UDP - The results include the number of SQL query pairs in the CALCITE benchmark that these tools support, the number of pairs whose equivalence can be proved by these tools, and the average time taken by these tools to determine QE.

Tool	Number of Pairs Supported	Number of Pairs Proved	Average Time(s)	Number of SPJ Pairs	Average Time(s)	Number of Aggregate Pairs	Average Time(s)	Number of Outer Join Pairs	Average Time(s)
EQUITAS	91	67	0.15	28	0.10	32	0.19	9	0.19
UDP	39	34	4.16	21	2.7	11	6.9	n/a	n/a

Table 3: Efficacy of EQUITAS on Production Queries - The second column refers to number of query pairs that operate on the same set of input tables. The fifth column reports the number of queries that exhibit at least one equivalence or containment relationship with another query in the same set. The sixth column indicates the highest frequency of a query in equivalent and containment query pairs. Lastly, the seventh column reports the number of query pairs with equivalence or containment relationships that contain advanced SQL features, such as aggregate functions and different types of join.

Query Set	Number of Queries	Compared Query Pairs	Query Pairs with Equivalence Relationship	Query Pairs with Containment Relationship	Duplicate Queries	Highest Query Frequency	Query Pairs with Aggregate Functions and Joins
Set 1	3285	122900	413	403	456	28	279
Set 2	3633	55311	432	259	442	22	366
Set 3	4182	61748	368	120	448	14	203
Set 4	3793	31774	249	100	427	13	165
Set 5	2568	15442	170	56	228	14	97
Total	17461	287175	1632	938	2001 (11%)	NA	1110 (43%)

Alibaba due to syntactical issues. Among the 91 test cases supported by EQUITAS, it can prove that 67 pairs (73%) are equivalent. In contrast, UDP is able to prove the equivalence of 34 pairs. We categorize the 67 proved pairs into three categories:

- **SPJ Pairs:** Queries that are SELECT-PROJECT-JOIN.
- **Aggregate Pairs:** Queries that have at least one aggregate.
- **Outer Join Pairs:** Queries that have at least one outer join.

We also report the number of pairs proved by UDP that have similar characteristics of each categories. Specifically, UDP reporsts 21 proved equivalent pairs of queries that are conjunctive union of SPJ queries, and 11 proved equivalent pairs of queries that have aggregate. UDP did not report the number of proved pairs that contain outer-join. UDP also reports that two proved cases require integrity constraints, and one case contains the key word **DISTINCT**, which requires reasoning about the query's interpretation in a bag semantics. We defer the discussion of supporting these two categories to Section 6.4

We measured the average time taken by EQUITAS to prove the equivalence of a pair of queries. This is an important metric since EQUITAS will need to efficiently determine QE for it to be deployed in cloud-scale DBaaS platforms. The average time is computed from only pairs that were successfully proved by EQUITAS and UDP. The average time taken by EQUITAS to prove the equivalence of a pair of queries is 0.15s. In contrast, the average execution time of UDP is 4.16s [26]. Thus, EQUITAS is 27× faster than UDP on these benchmarks. For SPJ and Aggregate queries, EQUITAS is consistently faster than UDP.

6.3 Efficacy on Production SQL Queries

We next examine the efficacy of EQUITAS in identifying overlap across production SQL queries. For this analysis, we curated five sets of SQL queries from the risk control department in Ant Financial Services Group [2]. These queries are used for detecting fraud and assigning credit scores, and are representative of complex production queries in business analytics. We investigate how EQUITAS improves the computational efficiency of data processing engines by identifying overlap across recurring resource-intensive analytical queries.

Within each set, we pass every pair of queries that operate on same set of input tables $\mathcal{T}$ to EQUITAS. In this experiment, EQUITAS determines the equivalence and containment relationships between the given pair of queries and their constituent sub-queries.

If EQUITAS determines that two queries Q1 and Q2 are not equivalent but have a common sub-query Q3, then we materialize the results of Q3 and execute Q1 and Q2 on top of the materialized results. We discard queries that only differ in the parameters passed on to their predicates and those that only comprise of scans over tables. EQUITAS trivially identifies equivalence across such closely related queries. We conducted this experiment on a development server in Alibaba.

The results of this experiment, as shown in Table 3, demonstrate that EQUITAS effectively identifies overlap across these diverse real-world analytical queries. Among the 17461 queries, we found that 11% of the queries exhibit at least one equivalence or containment relationship with another query in the same set. EQUITAS reports that these queries or their constituent sub-queries are present in at least one equivalent or containment query pair.

Certain SQL queries are repeatedly executed across the workload. We measured the highest frequency of a query in equivalent and containment query pairs. In the first set of queries, the result of the most frequently executed query is used in 28 other queries within the same set. In practice, the performance of production workloads is often limited by the time spent on executing queries that contain advanced SQL features, such as aggregate functions and different types of join. 43% of the query pairs with equivalence and containment relationships, that were identified by EQUITAS, contain these heavyweight SQL operators. These metrics highlight the utility of materializing the results of frequently executed queries, especially those containing heavyweight SQL operators.

6.3.1 Impact on Runtime Performance

We next examine the performance impact of materializing the results of queries identified by EQUITAS. For this analysis, we chose ten representative query pairs from the first set of queries. These pairs contain equivalent sub-queries with either aggregate functions or different types of join.

We materialized the results of these common sub-queries and manually rewrote the queries to operate on the materialized results. If these results are not materialized, these sub-queries would have to be executed twice. In contrast, they are executed only once if they are identified by EQUITAS and their results are materialized.

We measure the execution time and memory footprint of these query pairs in the two scenarios: (1) without materialization, and (2) with materialization. The queries are executed on an internal DBaaS
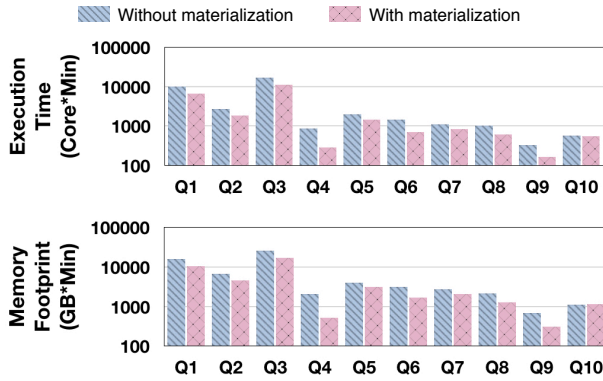
Figure 2: Impact on Runtime Performance - We examine the performance impact of materializing the results of queries identified by EQUITAS. We compare the execution time and memory footprint of these query pairs without and with materialization, respectively.

platform at Alibaba. We report these metrics in terms of the compute (virtual CPU-minutes) and memory resources (GB-minutes) consumed. The results shown in Figure 2 illustrate that materialization reduces the compute and memory resources consumed by 36% and 35%, respectively, among the examined query pairs.

6.4 Limitations

UNSUPPORTED SQL FEATURES: EQUITAS currently does not support SQL features, such as `EXIST` and `CAST`. However, many of these features can be supported using the existing framework. For instance, we can model `EXIST` using the `COUNT` aggregate function by ensuring that the resultant aggregate is greater than zero.

INTEGRITY CONSTRAINTS: EQUITAS currently does not support integrity constraints, such as primary key, foreign key and not-null column. However, integrity constraints can be supported by EQUITAS with additional engineering effort. For instance, when EQUITAS converts a left outer join of two tables with foreign key constraints into SR, EQUITAS can model it with additional constraints that each tuple in the left table has at least one match in the right table.

QUERY PAIRS WITH STRUCTURAL DIFFERENCES: Since EQUITAS relies on inference rules to handle aggregate functions and different types of `OUTER JOIN`, it is unable to detect containment and equivalence relationships in some query pairs with high-level structural differences that contains aggregate functions and different types of `OUTER JOIN`. For instance, EQUITAS cannot handle the following query pair in Calcite [3]:

```
Q1: SELECT EMP.DEPTNO, SUM(EMP.SAL) FROM
       EMP GROUP BY EMP.DEPTNO
```

```
Q2: SELECT t.DEPTNO, SUM(t.sum) FROM
    (SELECT EMP.DEPTNO, SUM(EMP.SAL) FROM
     EMP GROUP BY EMP.DEPTNO) AS t GROUP BY t.DEPTNO
```

To prove the equivalence relationship between Q1 and Q2, EQUITAS must infer that the two aggregate functions in Q2 can be reduced to a single aggregate function as in Q1. We must codify structural transformation rules for EQUITAS to support such query pairs with high-level structural differences. We plan to address this limitation in future work.

SET SEMANTICS: EQUITAS proves QE under set semantics [44]. For `SELECT-PROJECT-JOIN` queries, it proves QE under bag semantics. However, it cannot prove QE under bag semantics for complex queries containing aggregate functions and different types of `OUTER JOIN`. This is because we will need to model a table as an *unbounded*

data structure to handle these queries. Verifying properties of such complex data structures is an active area of research [41, 19, 20].

7. RELATED WORK

CONTAINMENT AND EQUIVALENT OF QUERIES: Researchers have recently proposed a pragmatic approach to determining the semantic equivalence of queries based on an algebraic representation [51, 25, 27]. These include the COSETTE and UDP tools that use $\mathcal{K}$-relations and $\mathcal{U}$-semirings. We highlighted the differences between our SR-based technique and these techniques in Section 6.

In general, proving containment and equivalence relationships between queries is undecidable [14, 18]. Prior efforts have focused on proving these properties for a subset of SQL queries: (1) conjunctive queries [24], (2) conjunctive queries with additional constraints [21, 36, 31], and (3) conjunctive queries under bag semantics [37]. The theoretical connection between containment of conjunctive queries and constraint satisfaction has been pointed in [42]. Another line of research focuses on constructing decision procedures for proving equivalence of a subset of SQL queries under set [22, 52, 49] and bag semantics [29, 39, 23]. Although these efforts have studied the theoretical aspects of proving QE, they have rarely been prototyped and applied on real-world SQL queries.

Prior work includes efficient procedures for deciding the equivalence of conjunctive queries[33, 46, 52, 28]. These efforts are geared towards query optimization transformations, and therefore cannot prove equivalence of queries with complex semantically-equivalent predicates. Another line of research focuses on proving equivalence of database schema [16, 17] and entity SQL queries [48]. Another line of research focuses on efficiently choosing effective equivalent sub-queries to materialize, in order to accelerate the evaluation of overlapping queries [40].

SMT SOLVER IN DATABASE SYSTEMS: Researchers have proposed several applications of SMT solvers in database systems, wherein a domain specific problem is reduced to logical constraints and then solved using the SMT solver. These include: (1) tools for automatically generating test cases for database applications [54, 55, 13], (2) tools that verify the correctness of database applications [56, 38, 35], (3) a tool for disproving the equivalence of SQL queries [51], and (4) a tool for synthesizing big data queries [50].

8. CONCLUSION

This paper presented an alternate approach to determining the equivalence of queries based on symbolic representation. We formulated algorithms for deriving the symbolic representation of widely-used SQL features, such as complex query predicates, three-valued logic, and aggregate functions. Our evaluation of this approach using EQUITAS showed that it proves the semantic equivalence of a larger set of query pairs compared to state-of-the-art algebraic approaches and reduces the verification time by $27\times$.

9. ACKNOWLEDGEMENT

10. REFERENCES

[1] Alibaba MaxCompute. https://www.alibabacloud.com/product/maxcompute.

[2] Ant Financial Services Group. https://www.antfin.com/.

[3] Apache Calcite project. http://calcite.apache.org/.

[4] Apache Drill project. http://drill.apache.org/.

[5] Apache Flink project. http://flink.apache.org/.

[6] Apache Hive project. http://hive.apache.org/.

[7] Apache Kylin project. http://kylin.apache.org/.

[8] Apache Phoenix project. http://phoenix.apache.org/.

[9] Azure Data Lake. https://azure.microsoft.com/en-us/solutions/data-lake/.

[10] Cosette: An automated SQL solver. https://github.com/uwdb/Cosette.

[11] Google BigQuery. https://cloud.google.com/bigquery/.

[12] Z3prover: Z3 theorem prover. https://github.com/Z3Prover/z3.

[13] S. Abdul Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *ASE*, 09 2008.

[14] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] J. Albert. Algebraic properties of bag data types. In *VLDB*, 1991.

[16] J. Albert, Y. Ioannidis, and R. Ramakrishnan. Conjunctive query equivalence of keyed relational schemas. In *PODS*, 1997.

[17] J. Albert, Y. Ioannidis, and R. Ramakrishnan. Equivalence of keyed relational schemas by conjunctive queries. In *Journal of Computer and System Sciences*, volume 58, 1999.

[18] B.A.Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. In *Journal of Symbolic Logic*, 1950.

[19] A. Bouajjani, C. Drăgoi, C. Enea, A. Rezine, and M. Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *CAV*, 2010".

[20] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, 2011.

[21] D. Calvanese, G. D. Giacomo, and M. Lenzerini. Conjunctive query containment and answering under description logic constraints. In *TOCL*, 2008.

[22] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[23] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *PODS*, 1993.

[24] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *ICDT*, 1997.

[25] S. Chu, D. Li, C. Wang, A. Cheung, and D. Suciu. Demonstration of the Cosette automated sql prover. In *SIGMOD*, 2017.

[26] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of sql queries. *PVLDB*, 11(11):1482–1495, 2018.

[27] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL: proving query rewrites with univalent sql semantics. In *PLDI*, 2017.

[28] S. Cohen, W. Nutt, and Y. Sagiv. Containment of aggregate queries. In *Lecture Notes in Computer Science*, 2002.

[29] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *PODS*, 1999.

[30] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 1962.

[31] G. De Giacomo, D. Calvanese, and M. Lenzerini. On the decidability of query containment under constraints. In *PODS*, 12 1999.

[32] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[33] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, 03 2002.

[34] B. Dutertre. Yices 2.2. In *CAV*, 2014.

[35] S. Grossman, S. Cohen, S. Itzhaky, N. Rinetzky, and M. Sagiv. Verifying equivalence of spark programs. In *CAV*, 2017.

[36] I. Horrocks, U. Sattler, S. Tessaris, and S. Tobies. How to decide query containment under constraints using a description logic. In *LPAR*, 2000.

[37] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. In *TODS*, 1995.

[38] S. Itzhaky, T. Kotek, N. Rinetzky, M. Sagiv4, O. Tamir5, H. Veith, and F. Zuleger. On the automated verification of web applications with embedded sql. In *ICDT*, 2017.

[39] T. S. Jayram, P. G. Kolaitis, and E. Vee. The containment problem for real conjunctive queries with inequalities. In *PODS*, 2006.

[40] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.

[41] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, 2009.

[42] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. In *PODS*, 1998.

[43] Y. V. Matiyasevich. Hilbert's tenth problem and paradigms of computation. In *CiE*, 2005.

[44] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of sql queries. In *ACM Trans. Database Syst.*, 1991.

[45] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1979.

[46] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far? In *SIGMOD*, 2002.

[47] C. Rubinson. Nulls, three-valued logic, and ambiguity in sql: Critiquing date's critique. *SIGMOD*, 2007.

[48] G. Rull, P. Bernstein, I. Santos, Y. Katsis, S. Melnik, and E. Teniente. Query containment in entity sql. In *SIGMOD*, 2013.

[49] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. In *J. ACM*, 1980.

[50] M. Schlaipfer, K. Rajan, A. Lal, and M. Samak. Optimizing big-data queries using program synthesis. In *SOSP*, 2017.

[51] C. Shumo, W. Chenglong, W. Konstantin, and C. Alvin. Cosette: An automated SQL prover. In *CIDR*, 2017.

[52] V. Tannen and L. Popa. An equational chase for path-conjunctive queries, constraints, and views. In *ICDT*, 1999.

[53] A. Tarski. A decision method for elementary algebra and geometry. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, 1951.

[54] M. Veanes, P. Grigorenko, P. de Halleux, and N. Tillmann. Symbolic query exploration. In *FormaliSE*, 2009.

[55] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic sql query explorer. In *LPAR*, 2010.

[56] Y. Wang, I. Dillig, S. K. Lahiri, and W. Cook. Verifying equivalence of database-driven applications. In *PACMPL*, 2017.