SYMBOLIC REASONING FOR QUERY VERIFICATION AND OPTIMIZATION

A Dissertation
Presented to
The Academic Faculty

by

Qi Zhou

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2020

SYMBOLIC REASONING FOR QUERY VERIFICATION AND OPTIMIZATION

Thesis committee:

Dr. William Harris

Galois, Inc

Dr. Shamkant B.Navathe
School of Computer Science
Georgia Institute of Technology

Dr. Joy Arulraj
School of Computer Science
Georgia Institute of Technology

Dr. John Regehr
School of Computing
University of Utah

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Date approved: November 24, 2020

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Structured Query Language (SQL) is the most widely used language for interacting with many database management systems (DBMS). Thus, the problems of optimizing and verifying SQL queries are two of the most studied problems in the DBMS community. Traditional techniques for optimizing and verifying SQL queries are based on syntax-driven approaches, which suffer many limitations in terms of effectiveness and efficiency.

In this dissertation, I investigate two important problems in query verification and optimization to demonstrate the limitations of syntax-driven techniques: (1) proving query equivalence under set and bag semantics; (2) optimizing queries with learned predicates. I propose to use symbolic reasoning to address the limitations of syntax-driven approaches in these two problems. I first present two techniques for proving query equivalence under set and bag semantics based on symbolic representation. Both approaches are significantly more efficient and effective than the previous state-of-the-art syntax-driven techniques. I then present a novel algorithm that combines symbolic reasoning with machine learning to synthesize new predicates for optimizing queries. This algorithm enables the query optimizer to leverage more optimization rules that it cannot previously apply. This technique significantly speeds up the execution of queries with complex predicates. In conclusion, this thesis proved that using symbolic reasoning can significantly improve the efficiency and effectiveness of techniques for query equivalence verification and query optimization.

CHAPTER I

INTRODUCTION

Structured Query Language (SQL) has been the most widely used language for interacting with many database management(DBMS) system. Thus, SQL query verification and optimization have become one of the most widely studied topic in DBMS community. In this dissertation, I present the investigation of leveraging automatic symbolic reasoning in SQL query verification and optimization. This dissertation presents three works: **(1)** proving query equivalence under set semantics, **(2)** proving query equivalence under bag semantics, **(3)** and optimizing queries with learned predicates.

In this chapter, I first explain the limitation of previous syntax-driven approach in the problem of verifying query equivalence and optimizing query with predicates. I then give a short introduction of symbolic reasoning, and how it can be used in query verification and optimization. I then list the main contributions of this dissertation. I finally conclude this chapter by giving the outline of the dissertation.

1.1 Limitation of Syntax-Driven Approach

In this section, I explain the limitations of using syntax-driven approach in verifying query equivalence and optimizing queries with predicates, respectively.

VERIFYING QUERY EQUIVALENCE: 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]). They offer multiple benefits over traditional on-premises DBMSs, including lower software licensing and infrastructure costs, rapid provisioning, reduced infrastructure management overhead, ability to elastically scale resources to meet demand, and higher availability. However, in practice, these pipelines may have a 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 [51]. This results in increased consumption of computational resources, higher data

processing costs, and longer query execution times. Addressing this problem requires automated cloud-scale tools for identifying semantically equivalent queries to minimize computation overlap.

Recently, Chu et al. have proposed a pragmatic approach to determining the semantic equivalence of queries [73, 28]. Their COSETTE and UDP tools transform SQL queries to algebraic expressions, normalize each algebraic expressions by a set of pre-defined rules, and decide if the queries are equivalent by finding an isomorphism between the resulting normalized algebraic expressions. COSETTE and UDP tools vary with respect to the algebraic representation to which they convert the given queries. Their experiments demonstrate they can prove equivalence over queries with significant structural difference.

However, they suffer from three 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` [68]. This limits their ability to support a wide range of real-world SQL queries. Second, they cannot prove queries with different predicates. This is because the decision procedure depends on a set of syntax-driven normalization rules to normalize each algebraic expressions. These syntax-driven normalization rules fails to normalize queries with different predicates into isomorphic algebraic expressions. Third, its decision procedure is computationally expensive. This is because they apply a series of rewrite rules on the given algebraic expressions to determine their equivalence with a large number of possible rules that can apply for each step. These three limitations restrict their efficacy and efficiency in cloud-scale DBaaS offerings.

PREDICATE-CENTRIC QUERY OPTIMIZATION: Researchers have proposed several predicate-centric rules for *moving* predicates across query blocks to improve performance (e.g., moving predicate below join operator [81], moving predicate below aggregation operator [58]). However, all these optimization rules are syntax-driven rules. As the result, all these rules may only be applied if the predicate depends on a given set of columns. Consider the following query:

```
Q1:  SELECT * FROM A, B
      WHERE A.id = B.id
        AND A.val + 10 > B.val + 20 AND B.val + 10 > 20
```

The optimizer may only move the third predicate ($B.val + 10 > 20$) below the join operator. It cannot push down the second predicate ($A.val + 10 > B.val + 20$) below the join operator since

it depends on columns from *both* tables A and B. The optimizer may apply this rule only if the predicate uses columns from *only one* table. Other predicate-centric optimization rules have similar restrictions related to the set of columns that the predicate depends on. For example, the optimizer may push the predicate below the aggregation operator only if the predicate uses columns from the `GROUP BY` set.

However, we may transform Q_1 to Q_2:

```
Q2:  SELECT * FROM A, B
      WHERE A.id = B.id
        AND A.val + 10 > B.val + 20
        AND B.val + 10 > 20
        AND A.val > 20
```

The newly added predicate ($A.val > 20$) can be inferred from the original predicates and is *weaker* than the original predicates (i.e., it accepts all the tuples that the original predicates accept). This predicate does *not* alter the semantics of the query. Furthermore, as it only uses columns from table A, the optimizer may push it below the join operator to filter tuples in A. The rewritten query $Q2$ is, thus, faster than the original query $Q1$.

This example illustrates the limitations of the current syntax-driven predicate-centric optimization rules. Two queries are equivalent queries if they are semantic equivalent (i.e., returns the same output table for all valid inputs). Two equivalent queries can have different performed execution plan because they are syntactically different. Researchers have proposed other syntax-driven rules for tackling this problem by rewriting the predicates (e.g., constant propagation [33] and transitive closure transformation over inequality relations [47]). However, due to the complexity of predicates in real-world queries (e.g., arithmetic operations, inequality relation, and logic combination), these syntax-driven rewrite rules have limited efficacy. These techniques also do not allow the optimizer to control the subset of columns in the original predicate that the synthesized predicate may use. Instead, the columns in the synthesized predicate still depend on the syntax of the original predicate.

1.2 Symbolic Reasoning

In this dissertation, I investigated using symbolic reasoning to address the limitations of syntax-driven approaches in verifying query equivalence and optimizing queries with predicates.

Symbolic reasoning is a technique invented in program verification area [55]. It converts the program into a set of satisfiable modulo theory (SMT) formula, and reduces the program verification problem into the problem of satisfiability of an SMT formula. An SMT solver determines if a given SMT formula is satisfiable. For example, the solver decides that the following formula can be satisfied: $x + 5 > 10 \wedge x > 3$ when x is six. Similarly, it determines that the following formula cannot be satisfied: $x + 5 > 10 \wedge x < 4$ since there is no integral value of x for which this formula holds. A detailed description of solvers is available in Section 3.2.

For verifying query equivalence under set semantics, I derive the *symbolic representation* [1] of SQL queries and use *satisfiability modulo theories* (SMT) to determine their equivalence [37]. I reduce the problem of determining the equivalence of queries under set semantics to the problem of determining the containment relationship between two symbolic representations of queries. This problem can be further reduced to the problem of deciding the satisfiability of an SMT formula.

For verifying query equivalence under bag semantics, I reduce the problem of determining the equivalence of queries under bag semantics to the problem of determining the existence of an identical, bijective map between output tables of two queries for all valid input. I decompose the proof into two steps. In the first step, I prove the existence of bijective map between output tables of two queries for all valid input. In the second step, I prove this bijective map is always the identical function. In both step, I derive the *query pair symbolic representation* for two given queries. Thus, in both step, I reduce the problem to the problem of deciding the satisfiability of an SMT formula.

By using symbolic approach in proving query equivalence under set and bag semantics, symbolic approach can model the semantics of widely-used SQL features, such as complex query predicates, arithmetic operations, and three-valued logic. It can prove queries with syntactically different predicates are equivalent. It also leverage the development of modern SMT solvers to efficiently solve the query equivalence problem [37, 40, 62].

For optimizing queries with predicates, I proposed using machine learning algorithm to learn a predicate that only uses given set of columns to enable the usage of syntax-driven, predicate-centric optimization rules. In this case, a predicate can be viewed as a binary classifier that separates the

[1] 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`.

desired tuples (TRUE samples) from the rest of the dataset (FALSE samples). Using a machine learning algorithm to train a binary classifier has been proposed in previous work to accelerate inference [59]. However, previous approach suffers from three limitations. First, there is no guarantee that the trained classifier is weaker than the original predicate. In other words, the rewritten query with newly learned predicate may not be *semantically equivalent* to the original query. While this is acceptable in a machine learning pipeline, it is not sufficient for canonical queries with strict accuracy constraints. Second, this approach is not capable of allowing the optimizer to choose the set of columns that the synthesized predicate uses. This is because it uses tuples labeled by the original predicate to train a binary classifier (that takes all the columns in the original predicate as inputs). Constraining the set of columns in the synthesized predicate could result in mis-labeling training samples with respect to the labels emitted by the original predicate. Third, this approach trains the classifier on real data during the query optimization stage. I seek to synthesize a valid predicate that does not depend on the current state of the database (i.e., only depends on the original predicate and works for all possible database states).

To address the first limitation, the algorithm uses an SMT solver to verify that the learned predicate is weaker than the original predicate. Thus, the rewritten query is guaranteed to be semantically equivalent to the original query. To address the second limitation, I formally prove the properties of tuples that should be selected or rejected by an optimal, valid predicate over the given set of columns. I encode these properties as an SMT formula and leverage the SMT solver to generate TRUE and FALSE samples for training the binary classifier. To improve the efficacy of the learning algorithm, I propose a novel learning process guided by counter-examples. In each iteration of the learning loop, if the learned predicate is not valid, then I use the SMT solver to generate TRUE samples that a valid predicate should select, but the current learned predicate rejects. If the learned predicate is valid but not optimal, then I use the SMT solver to generate FALSE samples that the optimal predicate should reject, but the current learned predicate selects. Thus, I addresses the third limitation by relying exclusively on the original predicate and the solver (and not on the contents of the database).

1.3 Result and Contribution

I implemented symbolic representation based approach under set and bag semantics as EQUITAS and SPES, respectively. I evaluated EQUITAS and SPES 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. The evaluation shows that both EQUITAS and SPES can prove the semantic equivalence of a larger set of query pairs (67 out of 232) under set semantics, and (90 out of 232) under bag semantics compared to UDP (34 out of 232). In addition to the Apache Calcite benchmark, I evaluated the efficacy of EQUITAS on a cloud-scale workload comprising of real-world SQL queries from Ant Financial Services Group [2]. Both EQUITAS and SPES are able to decides queries in this workload are overlapping with other queries.

I also implemented counter-example guided learning with verification in SIA. I evaluate SIA on 200 queries derived from the TPC-H benchmark [79]. It demonstrate that SIA effectively and efficiently synthesizes valid predicates, compared to syntax-driven rules and a non-iterative learning algorithm. Among the 114 queries that SIA rewrites, 66 queries exhibit more than $2\times$ speed up on average. These results show that SIA accelerates query execution by allowing the optimizer to apply more predicate-related optimization rules that it could not apply in the original query.

In summary, I make the following contributions in this dissertation:

- I motivate the need for using symbolic representation approach for proving query equivalence and synthesizing valid predicates.
- I propose two symbolic representation approaches for proving query equivalence under set semantics, and under bag semantics respectively.
- I present counter-example guided learning with verification approach for synthesizing strictly-valid predicates to optimize queries.
- I show that using symbolic representation approach can significantly improve effectiveness and efficiency of query verification and optimization.

1.4 Outline

The dissertation is organized as follows:

Chapter 2 introduces the related works for verifying query equivalence, predicate-centric query optimization, and symbolic reasoning for DBMS .

Chapter 3 presents the background works for query equivalence, and SMT solver.

Chapter 4 demonstrates a symbolic representation approach for proving query equivalence under set semantics .

Chapter 5 illustrates another symbolic representation approach with new problem formulation for proving query equivalence under bag semantics.

Chapter 6 describes the new counter-example guided learning with verfication for synthesizing predicates to optimzie queries Chapter 6.

Chapter 7 concludes this disseration.

CHAPTER II

RELATED WORK

In this chapter, I introduce previous related work. Section 2.1 introduces the previous works about proving containment and equivalent relationships between queries. Section 2.2 describes the previous works in predicate-centric query optimization. Section 2.3 shows previous works about symbolic reasoning in database related questions

2.1 Containment and Equivalence of Queries:

Previous efforts on proving containment and equivalence between queries focus on two parts: theoretical foundation, and the design and implementation of practical tools.

THEORETICAL FOUNDATION: In general, proving containment and equivalence relationships between queries is undecidable [16, 20]. Prior efforts have focused on proving these properties for a subset of SQL queries: (1) conjunctive queries [26], (2) conjunctive queries with additional constraints [21, 45, 36], and (3) conjunctive queries under bag semantics [48]. The theoretical connection between containment of conjunctive queries and constraint satisfaction has been pointed by Kolaitis [57]. Another line of research focuses on constructing decision procedures for proving equivalence of a subset of SQL queries under set [22, 77, 69] and bag semantics [32, 50, 25]. Although these efforts have studied the theoretical aspects of proving query equivlaence, they have rarely been prototyped and applied on real-world SQL queries. Prior work describes efficient procedures for deciding the equivalence of conjunctive queries [38, 66, 77, 31]. These efforts are geared towards query optimization transformations, and therefore cannot prove equivalence of queries with complex semantically-equivalent predicates.

PRACTICAL APPROACHES: Researchers have recently proposed a pragmatic approach to determining the semantic equivalence of queries based on an algebraic representation [73, 27, 29]. These include the COSETTE and UDP tools that use $\mathcal{K}$-relations and $\mathcal{U}$-semirings. Researchers have also proposed to use string matching to check containment relationship between select queries [42].

Researchers have also implemented efficient algorithms to decide containment between conjunctive queries [67].

2.2 Predicate-Centric Query Optimization:

PREDICATE SYNTHESIS: Researchers have focused on learning approximate predicates to accelerate query execution [72, 52, 59]. This line of research includes: training probabilistic predicates to accelerate inference in machine learning pipelines [59], inferring simpler approximate predicates from expensive UDFs [52]. Another seminal work in this area focuses on inferring strictly weaker predicates for expensive mining models [75]. Another line of research focuses on inferring predicates using column's statistics [53] and data correlations [56, 46].

PREDICATE MOVE AROUND Prior efforts have proposed a variety of techniques for migrating predicates [41, 58, 71, 80, 84, 81] to optimize queries. Another line of research focuses on normalizing predicates and choosing their order of execution [44, 54, 63].

2.3 Symbolic Reasoning in DMBS:

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 a solver. These include: (1) tools for automatically generating test cases for database applications [82, 83, 15], (2) tools that verify the correctness of database applications [85, 49, 43], (3) a tool for disproving the equivalence of SQL queries [73], and (4) a tool for synthesizing queries over big data [70]. These works are not specifically design to proving query equivalence under set or bag semantics.

CHAPTER III

BACKGROUND

In this chapter, I present the background knowledge for this dissertation. This chapter has two sections. Section 3.1 presents the theoretical foundation of verifying equivalence. Section 3.2 contains a short introduction of satisfiability modulo theories(SMT) and SMT solvers.

3.1 Theoretical Foundations

In this section, I will introduce the most important previous work on providing a theoretical foundation for proving query equivalence and query containment. This previous theoretical work shows the difficulty and the complexity of this series of problems. Before I introduce previous work, I will first formally define the containment and equivalent relationships of queries under different semantics. There are two conventional definitions for a table: a **set** of tuples or a **bag**, which is a multiset. For a given pairs of queries, if two queries are equivalent under set or bag semantics, it means that for any given database, the two tables that are returned by evaluating two queries on the database are equal sets or bags. For a given pairs of queries `Q1` and `Q2`, if `Q1` contains `Q2` under set or bag semantics, it means that for any given database, the table that is returned by `Q1` always contains the table that is returned by `Q2`, under set or bag semantics.

In general, deciding the containment and equivalence of queries under set or bag semantics is an undecidable problem [16]. Thus, previous work focuses on finding a subset of queries for which deciding containment and equivalence becomes decidable. Chandra and Merlin prove the problem of deciding containment of conjunctive queries under set semantics is decidable, and its complexity is NP-complete [23]. Sagiv and Yannakakis prove that the problem of deciding containment of union of conjunctive queries under set semantics is decidable, and its complexity is also NP-complete [69]. However, determining the complexity of deciding containment of conjunctive queries under bag semantics is still an open problem today [17].

3.2 SMT Solvers

I now present a brief overview of SMT solvers [37]. The satisfiability modulo theories (SMT) problem is, given a Boolean formula over predicates and functions in the vocabularlies of a set of *background theories*, to determine if the formula has a model in the combination of background theories. SMT supports a wide array of background theories, including integer linear arithmetic with integer, rational linear arithmetic, and uninterpreted functions.

An SMT *solver* is a tool that decides if a given 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, a solver, given the formula $(\mathtt{x} > 0) \wedge (\mathtt{x} < 5)$, determines that this formula is satisfiable (e.g., $\mathtt{x} = 1$ is a solution). However, the solver decides that the formula $(\mathtt{x} > 10) \wedge (\mathtt{x} < 5)$ is not satisfiable since there is no value of $\mathtt{x}$ that satisfies these constraints. In general, deciding satisfiability of SMT formulas is NP-hard. However, in practice, modern solvers employ heuristics from satisfiability theory [35] to efficiently solve SMT formulae [37, 40, 62]. Reducing the problem of determining the equivalence of queries to that of deciding the satisfiability of SMT formulae enables the usage of computationally efficient SMT solvers.

CHAPTER IV

QUERY EQUIVALENCE UNDER SET SEMANTICS

4.1 Overview

In this section, I first give a query example to show the limitation of previous algebraic expression based approach in Section 4.1.1. I then use the example to describe how to construct symbolic representation for queries and its intuition Section 4.1.2. I conclude by showing how to use the SMT solver to prove the properties of two symbolic representation to prove the containment relationship between two queries in Section 4.1.3.

4.1.1 An Query Example:

I give a pair of queries as an example to hightlight the limitaion of previous algebraic expression based approach. This query pairs operate on two tables:

- Employee table (EMP): ⟨ EMP_ID, EMP_NAME, DEPT_ID ⟩
- Department table (: ⟨ DEPT_ID, DEPT_NAME ⟩.

EXAMPLE 1. COMPLEX ARITHMETIC EXPRESSIONS: This example shows that two queries with different syntax predicate can be equivalent.

```
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.

Conventional algebraic approaches convert queries to algebraic expressions. For example, the state-of-the-art automated tool COSETTE [73] uses $\mathcal{K}$-relations for representing SQL queries, while UDP [28] leverages $\mathcal{U}$-semirings. The latter tool covers a broader set of SQL features compared to COSETTE. At a high level, the UDP algorithm rewrites queries using $\mathcal{U}$-expressions reminiscent of the chase/back-chase procedure [66, 77]. 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 isomorphism 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 prove queries with different predicates.

The algebraic representation of queries in the example **complex arithmetic expressions** are as follows:

$$\texttt{Q1} : [\texttt{t.DEPT\_ID} = 10] \times [\texttt{t.DEPT\_ID} + 5 > \texttt{t.EMP\_ID}] \times \texttt{EMP}(\texttt{t})$$
$$\texttt{Q2} : [\texttt{t.DEPT\_ID} = 10] \times [15 > \texttt{t.EMP\_ID}] \times \texttt{EMP}(\texttt{t})$$

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 $\texttt{t.DEPT\_ID} = 10$. $\texttt{EMP}(\texttt{t})$ is a function that returns the number of times t is present in EMP.

Algebraic approaches are unable to prove the equivalence of these expressions because 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 $[\texttt{t.DEPT\_ID} > \texttt{t.EMP\_ID} - 5]$.

4.1.2 Symoblic-Representation Based Approach:

Under set semantics, two queries are semantically equivalent if and only if for all valid input tables, the output tuples obtained after executing the queries on the input tables and *eliminating duplicates* are equivalent [61]. Under set semantics, `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 under set semantics. Thus, I reduce the problem of verifying query equivalence under set semantics to that of verifying the *containment relationship* between those queries. I formalize these definitions in Section 4.2.1.

I prove query equivalence under set semantics based on symbolic representation. With this approach, I represent tuples in output tables using symbolic tuples. I construct these symbolic tuples using a collection of *symbolic variables* that represent an arbitrary tuple. The symbolic representation models the semantics of SQL queries using SMT formulae. It enables the usage of SMT solvers to determine query equivalence by verifying the relationship between the symbolic representation of the two given queries.

Consider the example with complex arithmetic expressions shown in Section 4.1.1. 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, I 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 symbolic representation 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.

$\texttt{COND}_1$ and $\texttt{COND}_2$ 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 $\texttt{COND}_1$, encodes the semantics of the predicate $\texttt{DEPT\_ID} = 10$ in Q1. It is satisfied only when the value of DEPT_ID in the tuple equals 10 and it is not NULL. $\texttt{COLS}_1$ and $\texttt{COLS}_2$ are the symbolic tuples returned by Q1 and Q2 when the conditions $\texttt{COND}_1$ and $\texttt{COND}_2$ are satisfied, respectively. Since Q1 and Q2 only filter out tuples in EMP and do not modify them, $\texttt{COLS}_1$ and $\texttt{COLS}_2$ are set to be the input symbolic tuple. Lastly, I use $\texttt{ASSIGN}_1$ and $\texttt{ASSIGN}_2$ to specify relational constraints between symbolic variables while handling complex SQL operators, such as aggregate functions. I do not set these constraints in this example. I describe how to construct the symbolic representation of a query to Section 4.3.1.

For determining query equivalence under set semantics, I must prove that Q1 and Q2 contain each other. To show that Q1 contains Q2, I must prove two properties: (1) Every tuple in EMP returned by Q2 is also returned by Q1. In other words, if $\texttt{COND}_2$ is satisfied, then $\texttt{COND}_1$ 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 $\texttt{COLS}_2$ is equivalent to $\texttt{COLS}_1$ when the conditions $\texttt{COND}_1$ and $\texttt{COND}_2$ are satisfied. In this example, the latter condition trivially holds since neither query modifies the input symbolic tuple. More generally, I use SMT solvers to verify these two properties between the SR of queries. I use the same technique to determine if Q2 contains Q1, and thereby conclude if they are equivalent under set semantics. In this manner, the symbolic representation approach determines equivalence of queries with complex arithmetic expressions under set semantics.

4.1.3 Using SMT Solver

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

Consider the symbolic representation of queries in Example 1 (Section 4.1.2). I leverage the SMT solver to verify that $\texttt{COND}_1$ implies $\texttt{COND}_2$, and that $\texttt{COLS}_2$ is equivalent $\texttt{COLS}_2$ under $\texttt{COND}_1$ and $\texttt{COND}_2$ conditions.

1: To verify that $\texttt{COND}_1$ implies $\texttt{COND}_2$, I feed in these constraints to the SMT solver:

$$\texttt{COND}_1 \wedge \neg\texttt{COND}_2$$

The solver determines that this formula is not satisfiable, which implies that there is no counterexample to the fact that $\texttt{COND}_1$ implies $\texttt{COND}_2$. Thus, $\texttt{COND}_1 \implies \texttt{COND}_2$.

2: To verify that $\texttt{COLS}_1$ is equivalent $\texttt{COLS}_2$ under the $\texttt{COND}_1$ and $\texttt{COND}_2$ conditions, I feed in these constraints to the solver:

$$(\texttt{COND}_1 \wedge \texttt{COND}_2) \wedge \neg(\texttt{COLS}_1 = \texttt{COLS}_2)$$

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`.

4.2 Verifying Query Equivalence

In this section, I first give the formal definition of query equivalence under set semantics in Section 4.2.1. I then give the formal definition of symbolic representation of an query in Section 4.2.2. I finally describe how EQUITAS verifies the relationship between two symbolic representation of queries to verify the containment relationship in Section 4.2.3.

4.2.1 Problem Definition

I define the query equivalence under set semantics in terms of the query containment relationship. I 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*. I 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. I next define the query equivalence 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* $Q2 \subseteq Q1$*. I denote this equivalence relationship by* $Q1 \equiv Q2$*.*

Since I 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, I next describe how to automatically deduce that a given pair of SQL queries are equivalent under set semantics.

4.2.2 Symbolic Representation

I begin by defining the symbolic representation of a table constructed by executing an SQL query. I will discuss how to determine the relationship between queries using the representations of tables that they return in Section 4.2.3. The *SR* of a query Q is a tuple:

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

COND is an FOL formula that represents the constraint(s) that must be satisfied for the symbolic tuple $\vec{\texttt{COLS}}$ to be valid (i.e., a condition that an arbitrary tuple needs to satisfied in the output table).

$\vec{\texttt{COLS}}$ is a vector of pairs of FOL formulae that represent an arbitrary tuple that can be returned by Q. Each element $(\texttt{Val}, \texttt{Is-Null}) \in \vec{\texttt{COLS}}$ 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 $\vec{\texttt{COLS}}$. I use this formula to handle complex SQL features.

I observe that for SELECT-PROJECT-JOIN queries, an arbitrary tuple $\vec{\texttt{COLS}}$ in the output table is derived from a *finite* number of tuples present in the input tables referred to in Q. In Section 4.4, I discuss how EQUITAS handles queries that contain aggregate functions and different types of OUTER JOIN.

4.2.3 Verifying Equivalence

Given the definition of query equivalence in Definition 2, to verify the equivalence of two queries Q1 and Q2, EQUITAS needs to assert that they have a containment relationship. I 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. I 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: ($\texttt{COND}_1$, $\vec{\texttt{COLS}}_1$, $\texttt{ASSIGN}_1$) and ($\texttt{COND}_2$, $\vec{\texttt{COLS}}_2$, $\texttt{ASSIGN}_2$) respectively. I defer a discussion of the Construct procedure to Section 4.3.1. 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[1], 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 $\vec{\texttt{COLS}}_2$ 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

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

whenever $\texttt{COND}_2$ is satisfied, $\texttt{COND}_1$ is also met. This property is formalized as the constraint: $\texttt{COND}_1 \implies \texttt{COND}_2$.

2: When both tuples $\texttt{COL}\vec{\texttt{S}}_2$ and $\texttt{COL}\vec{\texttt{S}}_1$ are present in their respective output tables, they are equivalent. This property is formalized as the constraint: $(\texttt{COND}_1 \wedge \texttt{COND}_2) \implies (\texttt{COL}\vec{\texttt{S}}_1 = \texttt{COL}\vec{\texttt{S}}_2)$.

EQUITAS checks these two properties using an SMT solver.

1: For the first property, $\texttt{COND}_2 \implies \texttt{COND}_1$, EQUITAS feeds this formula to the solver: $(\texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2) \wedge (\texttt{COND}_2 \wedge \neg\, \texttt{COND}_1)$. If the solver determines that the formula cannot be satisfied, that shows that there exists no input tuple T that satisfies $\texttt{COND}_2$ while not meeting $\texttt{COND}_1$. In other words, $\texttt{COND}_2 \implies \texttt{COND}_1$ within the context of $\texttt{ASSIGN}_1$ and $\texttt{ASSIGN}_2$ for a given input tuple.

2: For the second property, EQUITAS feeds this formula to the solver: $(\texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2) \wedge (\texttt{COND}_2 \wedge \texttt{COND}_1) \wedge \neg\, (\texttt{COL}\vec{\texttt{S}}_1 = \texttt{COL}\vec{\texttt{S}}_2)$. 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.

4.3 SPJ Queries

In this section, I first discuss how to construct an symbolic representation for SELECT-PROJECT-JOIN queries in Section 4.3.1. I then describe how to construct symbolic representation for complicated projection expressions, complex predicates, and CASE operator in Sections 4.3.2 to 4.3.4, respectively.

4.3.1 Symbolic Representation Construction

I now describe a recursive algorithm for constructing the SR of the output table of a query. I begin by presenting the Construct algorithm that supports SELECT-PROJECT-JOIN queries. I will extend this algorithm to handle more advanced SQL features in Section 4.4. Table 1 presents an overview of the SR of different types of SQL queries and highlights the fields modified.

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

Procedure *Construct*(Q, S)
  switch Q **do**
    case *Scan*(n) **do**
      return $(\texttt{TRUE}, \textit{Init}(\texttt{T-Schema}(S[n])), \texttt{TRUE})$
    end
    case *Filter*$(p_s,\ Q_s)$ **do**
      $(\texttt{COND}_s, \vec{\texttt{COLS}}_s, \texttt{ASSIGN}_s) \leftarrow \mathsf{Construct}(\texttt{Q}_\texttt{s}, S)$
      $\texttt{COND} \leftarrow \texttt{COND}_s \wedge \mathsf{ConstructPred}(\texttt{p}_\texttt{s}, \vec{\texttt{COLS}}_s)$
      return $(\texttt{COND}, \vec{\texttt{COLS}}_s, \texttt{ASSIGN}_s)$
    end
    case *Proj*$(\vec{e},\ Q_s)$ **do**
      $(\texttt{COND}_s, \vec{\texttt{COLS}}_s, \texttt{ASSIGN}_s) \leftarrow \mathsf{Construct}(\texttt{Q}_\texttt{s}, S)$
      $\vec{\texttt{COLS}} \leftarrow \mathsf{ConstExpr}'(\vec{\texttt{e}}, \vec{\texttt{COLS}}_s)$
      return $(\texttt{COND}_s, \vec{\texttt{COLS}}, \texttt{ASSIGN}_s)$
    end
    case *Join*$(Inner,\ \vec{k_1} = \vec{k_2},\ Q1,\ Q2)$ **do**
      $(\texttt{COND}_1, \vec{\texttt{COLS}}_1, \texttt{ASSIGN}_1) \leftarrow \mathsf{Construct}(\texttt{Q1}, S)$
      $(\texttt{COND}_2, \vec{\texttt{COLS}}_2, \texttt{ASSIGN}_2) \leftarrow \mathsf{Construct}(\texttt{Q2}, S)$
      $\vec{\texttt{COLS}} \leftarrow \vec{\texttt{COLS}}_1 : \vec{\texttt{COLS}}_2\ Key \leftarrow \mathsf{ConstructPred}(\vec{\texttt{k}_1} = \vec{\texttt{k}_2}, \vec{\texttt{COLS}})$
      $\texttt{COND} \leftarrow \texttt{COND}_1 \wedge \texttt{COND}_2 \wedge Key$
      $\texttt{ASSIGN} \leftarrow \texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2$
      return $(\texttt{COND}, \vec{\texttt{COLS}}, \texttt{ASSIGN})$
    end
  end

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

SQL Query	COND	$\vec{\texttt{COLS}}$	ASSIGN
SELECT			
Filter	✓		
PROJECT		✓	
INNER JOIN	✓	✓	
OUTER JOIN	✓	✓	✓
Aggregate		✓	

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 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 $\mathtt{Q_s}$ and applies the filter on the results of $\mathtt{Q_s}$. It first recurses onto the sub-query and creates an SR of $\mathtt{Q_s}$ ($\mathtt{COND}_s$, $\vec{\mathtt{COLS}}_s$,$\mathtt{ASSIGN}_s$). I denote the filter by $\mathsf{Filter}(\mathtt{p_s},\ \mathtt{Q_s})$. This indicates that this operation consists of applying the predicate $\mathtt{p_s}$ on the results of $\mathtt{Q_s}$. Construct creates an SR of the filter by invoking the ConstructPred procedure on $\mathtt{p_s}$ and the symbolic tuple $\vec{\mathtt{COLS}}_s$. I defer a discussion of the ConstructPred procedure to Section 4.3.3. It then derives COND by combining the SR of the filter with $\mathtt{COND}_s$ using a conjunction operator. Lastly, it returns (COND, $\vec{\mathtt{COLS}}_s$, $\mathtt{ASSIGN}_s$) as the representation of Q. As shown in Table 1, only the condition formula differs between Q and $\mathtt{Q_s}$. This is because $\mathtt{p_s}$ filters out a subset of tuples in $\mathtt{Q_s}$ and otherwise does not alter the semantics of $\mathtt{Q_s}$.

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 $\mathtt{Q_s}$ and applies the projection on the results of $\mathtt{Q_s}$. It first recurses onto the sub-query $\mathtt{Q_s}$ and creates its symbolic representation. I denote the projection operator by $\mathsf{Proj}(\vec{\mathsf{e}},\ \mathtt{Q_s})$. The Construct procedure materializes an SR of the projected tuple $\vec{\mathtt{COLS}}$ by invoking the ConstExpr′ procedure on the columns in $\vec{\mathtt{COLS}}_s$. This ConstExpr′ procedure applies a set of transformations using a vector of expressions $\vec{\mathsf{e}}$. Internally, it calls the ConstExpr procedure on each expression in $\vec{\mathsf{e}}$ on the symbolic tuple $\vec{\mathtt{COLS}}_s$ and then collects the returned variables to materialize $\vec{\mathtt{COLS}}$.

Given a symbolic tuple and an expression e, the ConstExpr procedure applies the transformation associated with e on the tuple. I defer a description of the ConstExpr procedure to Section 4.3.2. Lastly, Construct returns ($\mathtt{COND}_s$, $\vec{\mathtt{COLS}}$, $\mathtt{ASSIGN}_s$) as the representation of Q. Since the PROJECT operator only applies transformations on the columns of the input tuples, the $\mathtt{COND}_s$ and $\mathtt{ASSIGN}_s$ 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. I denote the JOIN operator by $\mathsf{Inner\ Join}(\vec{\mathtt{k_1}} = \vec{\mathtt{k_2}},\ \mathtt{Q1},\ \mathtt{Q2})$. After deriving the SR of the sub-queries ($\mathtt{COND}_1$, $\vec{\mathtt{COLS}}_1$, $\mathtt{ASSIGN}_1$) and ($\mathtt{COND}_1$, $\vec{\mathtt{COLS}}_2$, $\mathtt{ASSIGN}_2$), it constructs the output symbolic tuple $\vec{\mathtt{COLS}}$ by concatenating $\vec{\mathtt{COLS}}_1$ and $\vec{\mathtt{COLS}}_2$. It

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

```
Input  : Expression e, Input symbolic tuple COLS⃗
Output : SR of e
Procedure ConstExpr(e, COLS⃗)
    switch e do
        case Column i do return COLS⃗[i] ;
        case Const v do return (v, FALSE) ;
        case NULL do return (0, TRUE) ;
        case Bin e1 op e2 do
            (VAL1, Is-NULL1) ← ConstExpr(e1, COLS⃗)
            (VAL2, Is-NULL2) ← ConstExpr(e2, COLS⃗)
            VAL ← ConstBin(op, VAL1, VAL2)
            return (VAL, Is-NULL1 ∨ Is-NULL2)
        end
        case Fun n (e⃗1) do
            sym⃗-e1 ← ConstExpr′(e⃗1, COLS⃗)
            (F-VAL, F-NULL) ← GetFun (n)
            return (F-VAL(sym⃗-e1), F-NULL(sym⃗-e1))
        end
    end
```

combines the $\texttt{COND}_1$ and $\texttt{COND}_2$ constraints along with the SR of the join predicate ($\vec{\texttt{k}_1} = \vec{\texttt{k}_2}$) to derive COND. Similarly, it coalesces the $\texttt{ASSIGN}_1$ and $\texttt{ASSIGN}_2$ 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. I note that Construct relies on ConstructPred procedure to encode filter and join predicates.

4.3.2 Encoding Expressions

I next describe how EQUITAS represents expressions, including arithmetic operations and user-defined functions (UDFs). I define the syntax of an expression as follows:

$$\text{e} ::= \text{Column } i | \text{Const } v | \texttt{NULL} | \text{Bin e op e} | \text{Fun } N\ (\vec{\text{e}})$$
$$\text{op} ::= + | - | \times | \div | mod$$

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 $\vec{\texttt{COLS}}$. 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 $\vec{\text{COLS}}$, that is referred to by e, is a vector of pairs of FOL formulae. 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 $\vec{\text{COLS}}$.

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 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[60]. 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 ($\vec{\text{e}_1}$) that operates on a vector of expressions $\vec{\text{e}_1}$, then ConstExpr first invokes the ConstExpr′ procedure on $\vec{\text{e}_1}$ to derive the SR of all the expressions in the vector ($\vec{\text{sym}}$-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 disambiguate 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 ($\vec{\text{sym}}$-e_1) to derive the SR of e.

4.3.3 Encoding Predicates

I then discuss how EQUITAS encodes predicates. In particular, I detail how it uses three-valued logic for supporting NULL. I define the syntax of a predicate as follows:

$$\text{p} ::= \text{BinE e cp e} | \text{BinL p logic p} | \textbf{Not p} | \textbf{IsNull e}$$
$$\text{cp} ::= \ > | < | = | \leq | \geq$$
$$\text{logic} ::= \texttt{AND} | \texttt{OR}$$

A predicate can be: **(1)** a comparison of two expressions, **(2)** a combination of two predicates using Boolean logic, **(3)** negation of 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 $\vec{\texttt{COLS}}$. ConstPred internally invokes an auxiliary ConstPredAux procedure that constructs a pair of FOL formulae. This pair represents the result of evaluating p on $\vec{\texttt{COLS}}$. 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 [68]. 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 $\texttt{VAL}_1$ and $\texttt{VAL}_2$. ConstComp derives a Boolean formula VAL to represent the comparison of $\texttt{VAL}_1$ and $\texttt{VAL}_2$ using cp. Lastly, it returns $(\texttt{VAL}, \texttt{Is-NULL}_1 \vee \texttt{Is-NULL}_2)$ as the SR of p. It uses the disjunction operator to combine $\texttt{Is-NULL}_1$ and $\texttt{Is-NULL}_2$ because if either of these expressions is NULL, then the value of p is unknown.

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

```
   Input  : Predicate p, Input symbolic tuple COLS
   Output : SR of p
1  Procedure ConstPred(p, COLS)
2      Procedure ConstPredAux(p, COLS)
3          switch p do
4              case BinE e1 cp e2 do
5                  (VAL1, Is-NULL1) ← ConstExpr(e1, COLS)
6                  (VAL2, Is-NULL2) ← ConstExpr(e2, COLS)
7                  VAL ← ConstComp(VAL1, VAL2, cp)
8                  return (VAL, Is-NULL1 ∨ Is-NULL2)
9              end
10             case BinL p1 l1 p2 do
11                 (VAL1, Is-NULL1) ← ConstPredAux(p1, COLS)
12                 (VAL2, Is-NULL2) ← ConstPredAux(p2, COLS)
13                 (VAL, Is-NULL) ← ConstLogic(l1, VAL1, VAL2, Is-NULL1, Is-NULL2)
14                 return (VAL, Is-NULL)
15             end
16             case Not p1 do
17                 (VAL1, Is-NULL1) ← ConstPredAux(p1, COLS)
18                 return (¬VAL1, Is-NULL1)
19             end
20             case IsNull e do
21                 (VAL1, Is-NULL1) ← ConstExpr(e, COLS)
22                 return (Is-NULL1,FALSE)
23             end
24         end
25     (VAL, Is-NULL) ← ConstPredAux(e, COLS)
26     return (VAL ∧ ¬Is-NULL)
```

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 $(\texttt{VAL}_1, \texttt{Is-NULL}_1)$ and $(\texttt{VAL}_2, \texttt{Is-NULL}_2)$. 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 $\texttt{VAL}_1$ and sets Is-NULL based on $\texttt{Is-NULL}_1$.

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 $\texttt{Is-NULL}_1$ 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, I 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.

4.3.4 Encoding Case Constructor

Lastly, I describe how I combine these techniques to handle the CASE statement. EQUITAS handles more complex features of SQL by leveraging the ConstExpr and ConstPred procedures presented in Sections 4.3.2 and 4.3.3. I next detail how it supports the CASE expression in this manner.

I define the syntax of the CASE expression as follows:

$$\texttt{CASE} := \texttt{WHEN}\ \mathsf{p}_1\ \mathsf{e}_1; \ldots \texttt{WHEN}\ \mathsf{p}_n\ \mathsf{e}_n;\ \texttt{ELSE}\ \mathsf{e}_d;$$

A CASE expression consists of a list of predicates $(\mathsf{p}_1, \ldots, \mathsf{p}_n)$. It returns one of multiple possible result sub-expressions $(\mathsf{e}_1, \ldots, \mathsf{e}_n)$ 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:

$$(\mathsf{p}_1, (\textsc{Val}_1, \textsc{Is-Null}_1); \ldots (\mathsf{p}_n, (\textsc{Val}_n, \textsc{Is-Null}_n));$$

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

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

$$\bigvee_{i \leqslant n} [\mathsf{p}_i \wedge \bigwedge_{s<i} \neg \mathsf{p}_s \implies (\textsc{Val} = \textsc{Val}_i \wedge \texttt{Is-Null} = \texttt{Is-Null}_i)]$$

EXAMPLE 2. CASE: Consider the following query and its SR:

```
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, I 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.

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

4.4 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 I covered in Section 4.3.1. 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. I next discuss how EQUITAS

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`

Procedure *Construct*(Q, S)
  switch `Q` **do**
    case *Join*$(Left,\ \vec{k_1} = \vec{k_2},\ Q1,\ Q2)$ **do**
      $(\texttt{COND}_1, \vec{\texttt{COLS}}_1, \texttt{ASSIGN}_1) \leftarrow \textsf{Construct}(\texttt{Q1}, S)$
      $(\texttt{COND}_2, \vec{\texttt{COLS}}_2, \texttt{ASSIGN}_2) \leftarrow \textsf{Construct}(\texttt{Q2}, S)$
      $Key \leftarrow \textsf{ConstructPred}(\vec{\texttt{COLS}}_1, \vec{\texttt{COLS}}_2, \vec{\texttt{k}_1} = \vec{\texttt{k}_2})$
      $(B, \texttt{COND}, \vec{\texttt{COLS}}) \leftarrow \textsf{Fresh}()$
      $cstr_1 \leftarrow \textsf{Asg}(\texttt{COND}_1, \texttt{COND}_2, Key, B, \texttt{COND})$
      $cstr_2 \leftarrow \textsf{Asg}(\vec{\texttt{COLS}}_1, \vec{\texttt{COLS}}_2, \vec{\texttt{COLS}}, B)$
      $\texttt{ASSIGN} \leftarrow \texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2 \wedge cstr_1 \wedge cstr_2$
      return $(\texttt{COND}, \vec{\texttt{COLS}}, \texttt{ASSIGN})$
    end
    case *Join*$(Full,\ \vec{k_1} = \vec{k_2},\ Q1,\ Q2)$ **do**
      .....
    end
    case *Aggregate*$(a\vec{g}g,\ \vec{g},\ Q_s)$ **do**
      $(\texttt{COND}_s, \vec{\texttt{COLS}}_s, \texttt{ASSIGN}_s) \leftarrow \textsf{Construct}(\texttt{Q}_s, S)$
      $\vec{\texttt{COLS}} \leftarrow \textsf{Fresh}(a\vec{g}g)$
      return $(\texttt{COND}_s, \vec{\texttt{COLS}}, \texttt{ASSIGN}_s)$
    end
    .....
  end

overcomes this challenge using independent variables in SRs (Section 4.4.1) and relational constraints for proving query equivalence (Section 4.4.2).

4.4.1 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 I covered in Section 4.3.1, remains unchanged. I next discuss how EQUITAS supports other types of queries.

LEFT OUTER JOIN: If `Q` is $\langle \textsf{Join}(Left,\ \vec{\texttt{k}_1} = \vec{\texttt{k}_2},\ \texttt{Q1},\ \texttt{Q2}) \rangle$, then Construct first recursively operates on the sub-queries `Q1` and `Q2` to derive their SR $(\texttt{COND}_1, \vec{\texttt{COLS}}_1)$ and $(\texttt{COND}_2, \vec{\texttt{COLS}}_2)$, 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 $(\vec{\texttt{k}_1} = \vec{\texttt{k}_2})$, 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.

I 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, I only need to construct a one-to-one mapping between Q1 and Q2 using the join predicate. However, in the latter case, I 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.

I 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 $\vec{\texttt{COLS}}$ 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:

$$(B \wedge (\texttt{COND} = \texttt{COND}_1)) \vee (\neg B \wedge (\texttt{COND} = (\texttt{COND}_1 \wedge \texttt{COND}_2 \wedge Key)))$$

$cstr_1$ 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 $\texttt{COND}_1$. Otherwise, then COND is the same as the INNER JOIN condition.

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

$$(B \wedge (\vec{\texttt{COLS}} = \vec{\texttt{COLS}}_1 : \vec{\texttt{NULL}}))\vee$$
$$(\neg B \wedge (\vec{\texttt{COLS}} = \vec{\texttt{COLS}}_1 : \vec{\texttt{COLS}}_2))$$

$cstr_2$ indicates that if B holds, then $\vec{\texttt{COLS}}$ is given by the concatenation of $\vec{\texttt{COLS}}_1$ and a vector of NULL values in the shape of the right table. Otherwise, if B not holds, I construct the new symbolic tuple by appending the old tuples $\vec{\texttt{COLS}}_1$ and $\vec{\texttt{COLS}}_2$.

It derives ASSIGN by combining $\texttt{ASSIGN}_1$ and $\texttt{ASSIGN}_2$ with $cstr_1$ and $cstr_2$. $(\texttt{COND}, \vec{\texttt{COLS}}, \texttt{ASSIGN})$ 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 $\langle \mathsf{Join}(Full,\ \vec{\texttt{k}_1} = \vec{\texttt{k}_2},\ \texttt{Q1},\ \texttt{Q2}) \rangle$, 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 $\langle \mathsf{Aggregate}(a\vec{g}g,\ \vec{g},\ \texttt{Q}_\texttt{s}) \rangle$, 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 $\vec{\texttt{COLS}}$ 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 ($\texttt{COND}_s$). Thus, Construct returns $(\texttt{COND}_s, \vec{\texttt{COLS}}, \texttt{ASSIGN}_s)$ as the SR of Q. In this manner, EQUITAS introduces independent variables in the symbolic representations of queries containing OUTER JOIN and aggregate functions.

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. I next describe how EQUITAS uses inference rules to construct these relational constraints and thereby deduce the equivalence of independent variables.

4.4.2 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:

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

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:

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

EQUITAS uses two vectors of independent variables $\texttt{COL}\vec{\texttt{S}}_1$ and $\texttt{COL}\vec{\texttt{S}}_2$ 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 $\texttt{COL}\vec{\texttt{S}}_1$ and $\texttt{COL}\vec{\texttt{S}}_2$ 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.

4.5 Soundness and Completeness

I now show that the procedure used in EQUITAS for checking the equivalence of two queries is *sound* under the set semantics. I 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. I prove this theorem using the method of contraposition. Suppose that Q1 does not contain Q2. By the definition of containment relationship in Section 4.3, 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. □

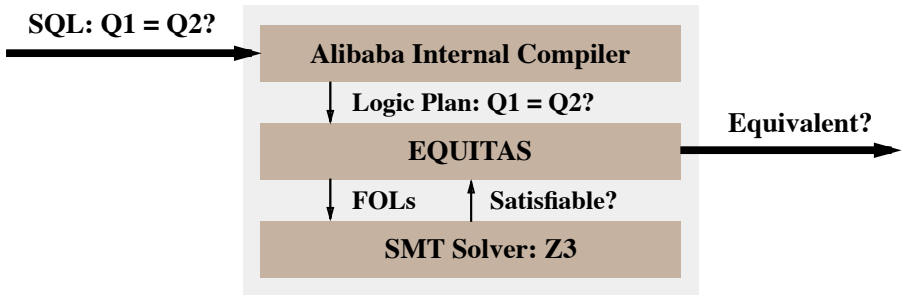


Figure 1: Query Equivalence Verification Pipeline - The pipeline for determining the equivalence of SQL queries. EQUITAS internally uses the Z3 SMT solver for determining the satisfiability of FOL formulae.

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. I 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 [78], 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 $(\texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2) \wedge (\texttt{COND}_2 \wedge \texttt{COND}_1) \wedge \neg\, (\vec{\texttt{COL}}\texttt{S}_1 = \vec{\texttt{COL}}\texttt{S}_2)$. Thus, I can construct a set of valid input tables T such that each input table only contains one tuple that matches the values in M. I 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`. □

4.6 Evaluation

In this section I describe our implementation and evaluation of EQUITAS. I begin with a description of our implementation in Section 4.6.1. Then, in Section 4.6.2, I report the results of a comparative analysis of EQUITAS against UDP [28], the state-of-the-art automated SQL query equivalence verifier. I examine the efficacy of EQUITAS in identifying overlap across production SQL queries in Section 4.6.3.

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 query equivalence.

Tool	Number of Pairs Supported	Number of Pairs Proved	Average Time(s)
EQUITAS	91	67	0.15
UDP	39	34	4.16

Table 3: Comparative analysis of EQUITAS and UDP - The result shows, in different category, the number of query pairs are determined query equivalence, and the average time taken by these tools.

Tool	Number of SPJ Pairs	Average Time(s)	Number of Aggregate Pairs	Average Time(s)	Number of Outer Join Pairs	Average Time(s)
EQUITAS	28	0.10	32	0.19	9	0.19
UDP	21	2.7	11	6.9	n/a	n/a

Table 4: Efficacy of EQUITAS on Production Queries - The second column refers to number of query pairs that operate on the same set of input tables.

Query Set	Number of Queries	Compared Query Pairs	Query Pairs with Equivalence Relationship	Query Pairs with Containment Relationship
Set 1	3285	122900	413	403
Set 2	3633	55311	432	259
Set 3	4182	61748	368	120
Set 4	3793	31774	249	100
Set 5	2568	15442	170	56
Total	17461	287175	1632	938

Table 5: Summary of EQUITAS on Production Queries - The second column reports the number of queries that exhibit at least one equivalence or containment relationship with another query in the same set. The thrid column indicates the highest frequency of a query in equivalent and containment query pairs. Lastly, the fourth 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	Duplicate Queries	Highest Query Frequency	Query Pairs with Aggregate Functions and Joins
Set 1	456	28	279
Set 2	442	22	366
Set 3	448	14	203
Set 4	427	13	165
Set 5	228	14	97
Total	2001 (11%)	NA	1110 (43%)

4.6.1 Implementation

I implemented EQUITAS as an SQL equivalence verification tool in Java. Figure 1 illustrates the pipeline for determining query equivalence. 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 [14].

4.6.2 Comparison against UDP

I now compare the efficacy of EQUITAS against UDP [28]. To the best of our knowledge, UDP is the state-of-the-art automated SQL equivalence verifier. For this comparative analysis, I used these tools to prove the equivalence of real-world SQL queries.

I 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 [2]. Second, since UDP is evaluated on the queries contained in the CALCITE test suite [28], I can quantitatively and qualitatively compare the efficacy of these tools.

I send every pair of queries and the schemata of their input tables to EQUITAS and ask it to prove query equivalence. I 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 Tables 2 and 3 . For comparative analysis against UDP, I present the results reported in the corresponding paper [28] [3]. The most notable observation

[2] The test cases used in this experiment were obtained from the open-sourced COSETTE repository [10].

[3] I were unable to conduct a comparative performance analysis under the same environment since UDP is currently not open-sourced.

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 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. I 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.

I also report the number of pairs proved by UDP that have similar characteristics of each categories. Specifically, UDP reports 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.

I 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 query equivalence 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 [28]. Thus, EQUITAS is 27$\times$ faster than UDP on these benchmarks. For SPJ and Aggregate queries, EQUITAS is consistently faster than UDP.

4.6.3 Efficacy on Production SQL Queries

I next examine the efficacy of EQUITAS in identifying overlap across production SQL queries. For this analysis, I 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 analytic. I investigate how EQUITAS improves the computational efficiency of data processing engines by identifying overlap across recurring resource-intensive analytical queries.

Within each set, I 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 I materialize the results of `Q3` and execute `Q1` and `Q2` on top of the materialized results. I 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. I conducted this experiment on a development server in Alibaba.

The results of this experiment, as shown in Table 4 and Table 5, demonstrate that EQUITAS effectively identifies overlap across these diverse real-world analytical queries. Among the 17461 queries, I 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. I 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.

4.6.4 Impact on Runtime Performance

I next examine the performance impact of materializing the results of queries identified by EQUITAS. For this analysis, I 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.

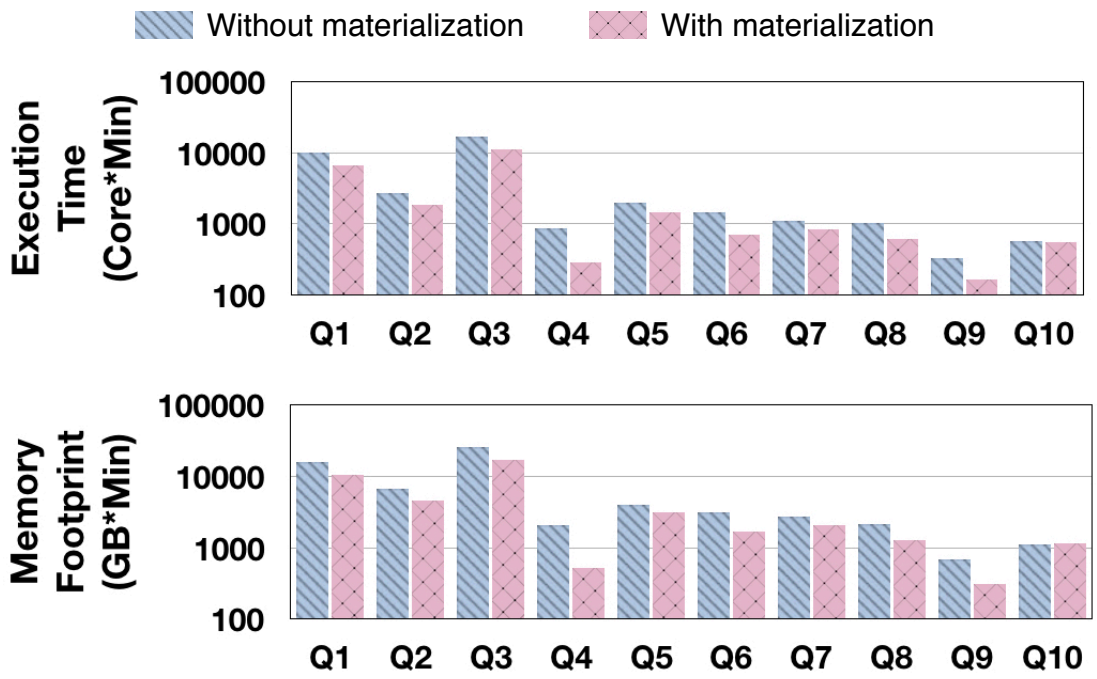


Figure 2: Impact on Runtime Performance - I examine the performance impact of materializing the results of queries identified by EQUITAS. I compare the execution time and memory footprint of these query pairs without and with materialization, respectively.

I 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.

I 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 platform at Alibaba. I 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.

CHAPTER V

QUERY EQUIVALENCE UNDER BAG SEMANTICS

In this first chapter, I present an symbolic representation approach to prove query equivalence under set semantics. However, In practice, all modern DBMS rely on *bag* semantics (i.e., output tables may contain duplicate tuples [19]). In this chapter, I present using symbolic representation approach with new problem formulation to prove query equivalence under bag semantics.

5.1 overview

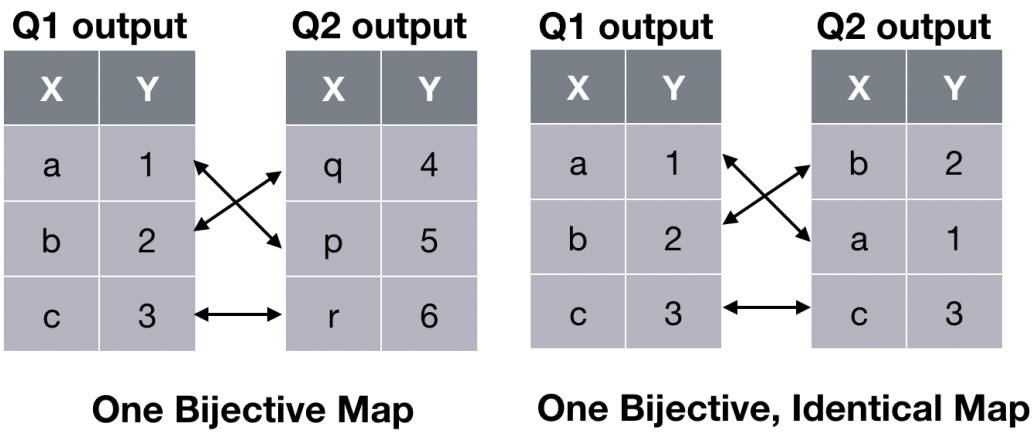
In this section, I first give an overview of proving query equivlence under bag semantics with new problem formulation in Section 5.1.1. I then use an example to demonstrate how this approach work in Section 5.1.2.

5.1.1 Query Pair Symbolic Representation Appraoch

I decompose proving query equivalence under bag semantics in two steps.

CARDINAL EQUIVALENCE: In the first step, SPES first verifies if the given pair of ARs are *cardinally equivalent* under bag semantics. Two queries are cardinally equivalent if and only if for all valid inputs, their output tables contain the same number of tuples. I defer a formal definition of cardinal equivalence to Definition 3. If two queries are cardinally equivalent, then there exists a *bijective map* between the tuples returned by these two queries for all valid inputs, as shown in Figure 3a. In this map, each tuple in the first table is mapped to a unique tuple in the second table, and all tuples in second table are covered by the map. I note that the contents of the output tables of two cardinally equivalent queries may differ.

SPES constructs a *Query Pair Symbolic Representation (QPSR)* for two cardinally equivalent queries to symbolically represent the bijective map between the returned tuples. It proves the cardinal equivalence of two queries by recursively constructing the QPSR of their sub-queries and using the SMT solver to verify specific properties of construed sub-QPSR based on the semantics of different



(a) Cardinal Equivalence **(b)** Full Equivalence

Figure 3: Types of Query Equivalence – Bijective maps implicitly constructed by SPES to determine: (a) cardinal equivalence and (b) full equivalence of queries under bag semantics.

types of ARs. I defer a discussion of how SPES proves cardinal equivalence and constructs QPSR to Sections 5.3.1 to 5.3.5.

FULL EQUIVALENCE: In the second step, SPES uses the constructed QPSR to verify that the given pair of queries are *fully equivalent* under bag semantics. Two queries are fully equivalent if and only if for all valid input tables, their output tables contain the same tuples (ignoring the order of the tuples). I defer a formal definition of full equivalence to Definition 4. If two queries are fully equivalent, then there exists a *bijective, identity map* between the tuples returned by these two queries for all valid inputs, as shown in Figure 3b. In this map, each tuple in the first table is mapped to a unique, identical tuple in the second table. All tuples in the second table are covered by this map. Since the QPSR of two given ARs symbolically represents the bijective map between the returned tuples, SPES proves the full equivalence of two ARs by using the SMT solver to show that the bijective map is an identity map.

SPES VS EQUITAS: The key difference between SPES and EQUITAS lies in how they prove query equivlaence. SPES transforms the query equivalence verification problem to prove the existence of a bijective, identity map between tuples in the output tables of the given queries for all valid inputs. EQUITAS reduces the query equivalence verification problem to determine the query containment relationships. These different problem formulations allow SPES and EQUITAS to prove query equivalence under bag and set semantics, respectively. As shown in Table 6, SPES supports a larger set of q features in comparison to UDP, and EQUITAS.

Table 6: Support for q Features – Comparison of the q features supported by UDP, EQUITAS and SPES. ✓ denotes that the tool supports this feature. Complex predicates include those using: (1) arithmetic operations, (2) NULL, and (3) CASE.

	EQUITAS	UDP	SPES
SPJ	✓	✓	✓
Aggregate	✓	✓	✓
Union		✓	✓
Outer-Join	✓		✓
Complex Predicate	✓		✓
Table Semantics	set	bag	bag

5.1.2 Illustrative Example

I use the following pairs of queries to show how SPES proves the equivalence of queries under bag semantics.

```
Q1: SELECT EMP.DEPT_ID, SUM(EMP.SALARY) FROM EMP, DEPT
    WHERE EMP.DEPT_ID = DEPT.DEPT_ID AND EMP.SALARY > 1000
    GROUP BY EMP.DEPT_ID ;
Q2: SELECT T.DEPT_ID, SUM(T.s) FROM
    (SELECT EMP.DEPT_ID, EMP.LOCATION,
      SUM(EMP.SALARY) as s FROM DEPT, EMP
      WHERE EMP.DEPT_ID = DEPT.DEPT_ID AND
      EMP.SALARY + 1000 > 2000
    GROUP BY EMP.DEPT_ID, EMP.LOCATION) as T GROUP BY T.DEPT_ID;
```

Q1 is an aggregation query that calculates the sum of salaries of employees whose salary is greater than 1000 grouped by their department id. Q2 is a nested query. The inner query calculates the sum of salaries of all employees whose salary plus 1000 is greater than 2000, grouped by their department id and location. The outer query then calculates the sum of salaries of those employees grouped by their department id. Q1 and Q2 are equivalent because the *group set* of the outer query in Q2 is a subset of the group set of the inner query.

NORMALIZATION STAGE: Before SPES try to prove the equivalence of Q1 and Q2 under bag semantics, it normalize Q1 and Q2 by using algebraic expressions (ARs). Figure 4 shows the ARs of two queries Q1 and Q2. The AR of Q1 is an aggregate AR that takes a SELECT-PROJECT-JOIN (SPJ) AR as input, the department id as the group set, and the sum of salaries as the aggregate operation. The SPJ AR takes two table ARs (EMP and DEPT) as input (EMP with the filter predicates. The AR

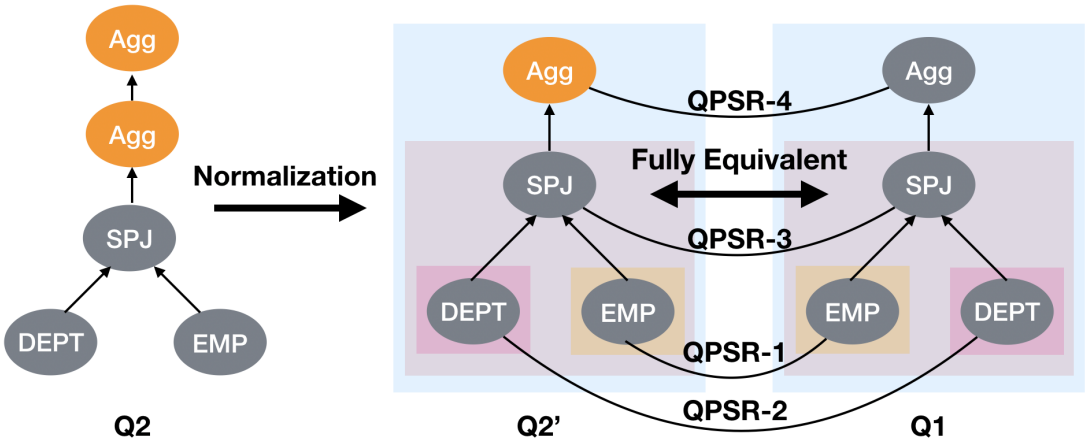


Figure 4: Illustrative Example – The two-stage approach that SPES uses to prove query equivalence under bag semantics.

of Q2 is the same aggregate AR as Q1 except it takes another aggregate AR as input. The input aggregate AR takes an SPJ AR as input, department id and location as group set, and sum of salaries as aggregate operation. The SPJ AR is the same as the SPJ AR in Q1 except that its predicate is different (i.e., EMP.SALARY + 1000 > 2000), and the order of input table ARs is reversed.

SPES applies a set of rewrite rules to normalize these two ARs. Specifically, it merges the two aggregate ARs within Q2 into a single one. This normalized AR of Q2 is denoted by Q2' in Figure 4. The AR of Q1 remains unchanged after normalization.

PROVING STAGE: SPES tries to prve the equivalence of two normalized ARs Q1 and Q2'. In this first step, SPES first verifies the cardinal equivalence of two aggregate ARs. In order to verify the cardinal equivalence of two aggregate ARs, SPES recursively constructs the QPSR of two SPJ ARs that the aggregate ARs take as inputs. To verify the cardinal equivalence of two SPJ ARs, it finds a bijective map between their input ARs and checks if each pair of input ARs cardinally equivalent. If that is the case, then it constructs a QPSR for each pair of table ARs. SPES maps the EMP table AR in Q1 with the EMP table AR in Q2', and the DEPT table AR in Q2 with the DEPT table AR in Q2'.

QPSR-1: The QPSR for the pair of EMP table ARs:

```
COND: True
COLS1: {(v1,n1),(v2,n2),(v3,n3),(v4,n4)}
COLS2: {(v1,n1),(v2,n2),(v3,n3),(v4,n4)}
```

Here, $\texttt{COLS}_1$ and $\texttt{COLS}_2$ symbolically represent two corresponding tuples returned by the two cardinally equivalent table ARs, respectively. Each symbolic tuple is a vector of pairs of FOL terms.

I present the formal definitions of $\texttt{COLS}_1$ and $\texttt{COLS}_2$ in §5.2.3. This pair of symbolic tuples $\texttt{COLS}_1$ and $\texttt{COLS}_2$ defines a bijective map between the tuples returned by the table ARs. Since both table ARs refer to `EMP`, the bijective map is an identity map.

$\{(\texttt{v1}, \texttt{n1}), (\texttt{v2}, \texttt{n2}), (\texttt{v3}, \texttt{n3}), (\texttt{v4}, \texttt{n4})\}$ symbolically represents a tuple returned by the `EMP` table AR. Each pair of symbolic variables represents a column. For instance, $(\texttt{v1}, \texttt{n1})$ denotes `EMP_ID` in this symbolic tuple. `v1` represents the value of EMP_ID, `n1` indicates if the value is NULL. The encoding scheme is the same as the one used for proving query equivalence under set semantics. `COND` is an FOL formula that represents the filter conditions. It must be satisfied for the tuples to be present in the output table of this AR. `COND` is `TRUE` because table AR returns all tuples.

QPSR-2: The QPSR for the pair of `DEPT` table ARs:

```
COND: True; COLS1: {(v5,n5),(v6,n6)}; COLS2: {(v5,n5),(v6,n6)}
```

$\{(v5, n5), (v6, n6)\}$ symbolically represents a tuple is returned by the `DEPT` table AR.

QPSR-3: SPES uses these two QPSRs and leverages the SMT solver to verify that predicates always return the same boolean results for the corresponding tuples in the join table to verify that the two SPJ ARs are cardinally equivalent. SPES then constructs a QPSR for these two SPJ ARs:

```
COND: (v2 + 1000 > 2000 and !n2) and (v2 > 1000 and !n2)
COLS1: {(v1,n1),(v2,n2),(v3,n3),(v4,n4),(v5,n5),(v6,n6)}
COLS2: {(v1,n1),(v2,n2),(v3,n3),(v4,n4),(v5,n5),(v6,n6)}
```

$\texttt{COLS}_1$ and $\texttt{COLS}_2$ symbolically represent a bijective map between tuples in the output tables of two SPJ ARs. This bijective map *preserves* the two bijective maps in the two sub-QPSRs between their input table ARs.

In other words, if a tuple t_1 is mapped to another tuple t_2 in QPSR-1, and a tuple t_3 is mapped to another tuple t_4 in QPSR-2, then the join tuple of t_1 and t_2 maps to that of t_3 and t_4 in QPSR-3. In this manner, the mapping in the lower-level QPSRs is preserved in the higher-level QPSR. `COND` is the conjunction of the filter predicates.

QPSR-4: SPES uses QPSR-3 and the SMT solver to verify that the two aggregate ARs are cardinally equivalent. If so, it constructs a QPSR for the aggregate ARs (i.e., `Q1` and `Q2`):

```
COND: (v2 + 1000 > 2000 and !n2) and (v2 > 1000 and !n2)
COLS1: {(v1,n1),(v7,n7)}; COLS2: {(v1,n1),(v7,n7)}
```

Here, $\texttt{COLS}_1$ and $\texttt{COLS}_2$ symbolically represent a bijective map between tuples returned by `Q1` and `Q2`, respectively. $(v7, n7)$ represents the sum of salaries column.

FULL EQUIVALENCE: After determining cardinal equivalence, SPES uses QPSR-4 to proves the full equivalence of `Q1` and `Q2` by showing the bijective map is an identity map. It uses an SMT solver to verify the following property of QPSR-4: $\texttt{COND} \implies \texttt{COLS}_1 = \texttt{COLS}_2$. It feeds the negation of the property to the solver. The solver determines that it cannot be satisfied, thereby showing that the paired symbolic tuples are always equivalent when `COND` holds. Thus, the bijective map between the tuples returned by the ARs is an identity map. So, `Q1` and `Q2` are fully equivalent under bag semantics.

SUMMARY: SPES first constructs QPSR-1 for `EMP` table ARs and QPSR-2 for able ARs. It then uses these QPSRs to determine the cardinal equivalence of SPJ ARs. Next, it constructs QPSR-3 for the SPJ ARs. SPES then uses QPSR-3 to determine the cardinal equivalence of aggregate ARs and constructs QPSR-4 for the overall queries. Lastly, it uses QPSR-4 to decide the full equivalence of `Q1` and `Q2`. Thus, SPES only establishes cardinal equivalence before constructing the QPSRs. It only checks full equivalence for the top-level QPSR (i.e., QPSR-4).

5.2 Verifying Query Equivalence

In this section, I first define the syntax and the semantics of the algebraic representation(AR) that capture the semantics of SQL queries in Section 5.2.1. I then give the formal definition of query equivalence under bag semantics in Section 5.2.2. Fianlly, I present how SPES proves the full equivalence under bag semantics of a pair of cardinally equivalent ARs using their query pair symbolic representation in Section 5.2.3.

5.2.1 Syntax and Semantics

I first define the syntax of the AR. I then describe the semantics of the AR based on the relationships between the input and output tables. An AR e is defined as:

$$\mathsf{e} ::= \textsc{Table}(n) | \mathsf{SPJ}(\vec{\mathsf{e}}, \textsc{p}, \vec{o}) | \mathsf{Aggregate}(\mathsf{e},\ \vec{g},\ a\vec{g}g) | \mathsf{Union}(\vec{\mathsf{e}})$$

In SPES, an AR can be: (1) a table AR, (2) an SPJ AR, (3) an aggregate AR, or (4) a union AR. I consider a table to be a bag (i.e., multi-valued set) of tuples as it best represents real-world databases. I now formally define the semantics AR queries, using the following formal notation. $\Downarrow$ is the evaluation symbol. The left side of this symbol is an algebraic expression that is evaluated on valid input tables `Ts`. The right side of this symbol is the evaluation result, which is the output table. All output tables are bags (i.e., can contain duplicate tuples). A horizontal line separates the pre- and the post-conditions. The pre-conditions on the top of the line include a set of evaluation relations. The post-condition on the bottom side of the line is an evaluation relation. If all the relations in the pre-conditions hold, then the relation in the post-condition holds.

- Given a set of valid input tables `Ts`, the table AR returns all the tuples in table n.
- Given a set of valid input tables `Ts`, the SPJ AR first evaluates the vector of input ARs on `Ts` to obtain a vector of input tables. For each tuple t in the cartesian product of the vector of input tables, if t satisfies the given predicate p, it then applies the vector of expressions $\overrightarrow{o}$ on the selected tuple t and emits the transformed tuple.
- Given a set of valid input tables `Ts`, this aggregate AR first evaluates the input AR on `Ts` to get an input table T_0. Then, it uses `part` to partition the input table T_0 into a set of bags of tuples as defined by a set of group set $\vec{g}$ (tuples in each bag take the same values for the grouping attributes). Lastly, for each bag of tuples, it applies the vector of aggregate functions and returns one tuple.
- Given a set of valid input tables `Ts`, this union AR first evaluates the vector of input ARs on `Ts` to get a vector of input tables. It then returns all the tuples present in the input tables, which does not eliminate duplicate tuples.

5.2.2 Problem Definition

To define the full equivalence of queries under bag semantics, I first define the cardinal equivalence relationship.

Definition 3. CARDINAL EQUIVALENCE: *Given a pair of queries `Q1` and `Q2`, `Q1` and `Q2` are cardinally equivalent if and only if (iff), for all valid input tables, the output tables T_1 and T_2 of `Q1` and `Q2` contain the same number of tuples.*

If Q1 and Q2 are cardinally equivalent, for all valid inputs, each tuple in T_1 can be mapped to a unique tuple in T_2, and all tuples in T_2 are in the map. Thus, it is a bijective (one-to-one) map between tuples in T_1 and T_2. However, the two mapped tuples may differ in their values, as shown in Figure 3a.

Definition 4. FULL EQUIVALENCE: *Given a pair of queries Q1 and Q2, Q1 and Q2 are fully equivalent iff, for all valid input tables Ts, the output tables T_1 and T_2 of Q1 and Q2 are identical.*

If Q1 and Q2 are fully equivalent, for all valid inputs, there exists a bijective map between tuples in T_1 and T_2, and this bijective map is an identity map. In other words, each tuple in T_1 can always be mapped to a unique, identical tuple in T_2, and all tuples in T_2 are in the map, as shown in Figure 3b.

MOTIVATION I first try to prove cardinal equivalence before checking for full equivalence. This is because if Q1 and Q2 are fully equivalent, then they must be cardinally equivalent. To prove full equivalence, I prove that the bijective map between tuples in the output tables is an identity map. In the rest of the paper, *equivalent* queries without any qualifier refer to fully-equivalent queries.

SPES can prove that ARs are fully equivalent even if their sub-ARs are *only* cardinally equivalent.

5.2.3 Proving Full Equivalence

I now define the symbolic representation of normalized ARs that SPES uses for proving equivalence. QPSR is an extension of the SR defined in EQUITAS. QPSR is used to prove query equivlaence under bag semantics. In QPSR, I augment the SR to use a pair of symbolic tuples to track a bijective map between the tuples that are returned by two cardinally equivalent ARs. QPSR of a pair of cardinally equivalent ARs Q1 and Q2 is a tuple of the form:

$$\langle \vec{\texttt{COLS}}_1, \vec{\texttt{COLS}}_2, \texttt{COND}, \texttt{ASSIGN} \rangle$$

$\vec{\texttt{COLS}}_1$ is a vector of pairs of FOL terms that represent an arbitrary tuple returned by Q1. Each element of this vector represents a column and is of the form: $(\texttt{VAL}, \texttt{Is-NULL})$, where VAL represents the value of the column and Is-NULL denotes the nullability of the column. $\vec{\texttt{COLS}}_2$ is another vector of pairs of FOL terms that represents a tuple returned by Q2. Since Q1 and Q2 must be cardinally equivalent before SPES constructs their QPSR, the two symbolic tuples $\vec{\texttt{COLS}}_1$ and $\vec{\texttt{COLS}}_2$

define a bijective map between the returned tuples. **COND** is an FOL formula that represents the constraints that must be satisfied for the symbolic tuples $\texttt{COL}\vec{\texttt{S}}_1$ and $\texttt{COL}\vec{\texttt{S}}_2$ to be returned by Q1 and Q2, respectively. They encode the semantics of the predicates in the queries. **ASSIGN** is another FOL formula that specifies the relational constraints between symbolic variables used in $\texttt{COL}\vec{\texttt{S}}_1$, $\texttt{COL}\vec{\texttt{S}}_2$ and COND. This formula is used for supporting complex SQL operators, such as CASE.

VERIFYING FULL EQUIVALENCE: To prove that two cardinally equivalent ARs Q1 and Q2 are fully equivalent, SPES needs to prove that the bijective map between returned tuples is an identity map. In other words, SPES needs to prove that, for an arbitrary tuple t returned by Q1, the bijective map associates t to an identical tuple returned by Q2 with the same values. SPES verifies this property using the QPSR of Q1 and Q2. When both symbolic tuples satisfy the predicate (i.e., COND), it must verify that $\texttt{COL}\vec{\texttt{S}}_1$ is equivalent to $\texttt{COL}\vec{\texttt{S}}_2$. This property is formalized as:

$$\texttt{COND} \wedge \texttt{ASSIGN} \implies \texttt{COL}\vec{\texttt{S}}_1 = \texttt{COL}\vec{\texttt{S}}_2$$

SPES verifies this property using an SMT solver [37]. If the property does not hold, then the negation of this property is satisfiable. SPES feeds the negation of this property into the SMT solver. If the solver determines that this formula is unsatisfiable, then I prove that $\texttt{COL}\vec{\texttt{S}}_1$ and $\texttt{COL}\vec{\texttt{S}}_2$ are always identical. In this manner, I leverage the QPSR to prove full equivalence.

5.3 Proving Cardinal Equivalence

In this section, I discuss how it decides if a pair of ARs are cardinally equivalent, and how it constructs QPSR when they are cardinally equivalent in Sections 5.3.1 to 5.3.5.

5.3.1 Construction of QPSR

Alg. 5 presents a recursive procedure VeriCard for verifying the cardinal equivalence of two ARs. The VeriCard procedure takes a pair of ARs as inputs (i.e., Q1's AR and Q2's AR). It first checks the types of the given ARs. If they are of the same type, then it invokes the appropriate sub-procedure for that particular type. I describe these four sub-procedures in Sections 5.3.2 to 5.3.5. If Q1 and Q2 are cardinally equivalent, then VeriCard returns their QPSR. If these ARs are of different types, it

Algorithm 5: Procedure for verifying cardinal equivalence of ARs. It constructs the QPSR only if they are cardinally equivalent.

```
   Input  : A pair of ARs (i.e., Q1 and Q2)
   Output : QPSR of Q1 and Q2 or NULL
1  Procedure VeriCard(Q1, Q2)
2      switch TypeOf(Q1, Q2) do
3          case Table do return VeriTable(Q1, Q2) ;
4          case SPJ do return VeriSPJ(Q1, Q2) ;
5          case Union do return VeriUnion(Q1, Q2) ;
6          case Agg do return VeriAgg(Q1, Q2) ;
7          case Type Mismatch do return NULL ;
8      end
```

returns NULL to indicate that it cannot determine their cardinal equivalence. This is because each type of AR has different semantics (§5.2.1).

Some sub-procedures recursively invoke VeriCard to verify the cardinal equivalence between their sub-queries. It applies the normalization rules to transform the given two ARs so that they are of the same type (and the sub-queries are also of the same types recursively). This normalization process is *incomplete* (i.e., SPES may conclude that two ARs are not cardinally equivalent since they cannot be normalized to the same type, even if they are actually cardinally equivalent). I discuss this limitation in §5.4.4.

Each sub-procedure takes a pair of ARs of the same type as inputs. It first attempts to determine if they are cardinally equivalent. If they are cardinally equivalent, then it constructs the QPSR of Q1 and Q2. Otherwise, it returns NULL to indicate that it cannot determine their cardinal equivalence.

In each of the following sub-sections, I first describe the conditions that are sufficient for proving cardinal equivalence based on the semantics of the AR. I then describe how each sub-procedure verifies these conditions to prove cardinal equivalence. I then discuss how SPES constructs the QPSR if they are cardinally equivalent. Lastly, I describe their soundness and completeness properties [1].

5.3.2 Table AR

Alg. 6 illustrates the VeriTable procedure for table ARs.

CARDINAL EQUIVALENCE:

[1] A sub-procedure P is sound if whenever it returns a QPSR, the given ARs are cardinally equivalent and the two symbolic tuples define a bijective map. A sub-procedure P is complete if whenever it returns NULL, the given ARs are not cardinally equivalent.

Algorithm 6: Comparison function for Table ARs

```
Input  : A pair of table ARs
Output : QPSR of the table ARs or NULL
Procedure VeriTable(TABLE(n1), TABLE(n2))
    if n1 = n2 then
        COLS1 ← Init(T-SCHEMA(n1))
        COLS2 ← COLS1
        return (COLS1, COLS2, TRUE, TRUE)
    else return NULL;
```

Lemma 1. *A pair of table ARs* $\text{TABLE}(n_1)$ *and* $\text{TABLE}(n_2)$ *are cardinally equivalent iff their input tables are the same. (i.e.,* $n_1 = n_2$*).*

Since the table AR returns all tuples from the input table, thus if two table ARs' input tables are the same, then they will always have the same number of tuples. So VeriTable compares the names of the two input tables (i.e., n_1 and n_2). SPES cannot show that tables with different names are cardinally equivalent in the presence of integrity constraints.

QPSR: I define the QPSR of the two cardinally equivalent table ARs using an *identity map* between the returned tuples (e.g., QPSR-1 in Section 5.1.2). VeriTable first constructs the symbolic tuple $\vec{\text{COLS}}_1$ using a vector of new pairs of variables based on the table schema, and then sets the symbolic tuple $\vec{\text{COLS}}_2$ to be the same as $\vec{\text{COLS}}_1$. These two equivalent tuples $\vec{\text{COLS}}_1$ and $\vec{\text{COLS}}_2$ define a bijective map between returned tuples. VeriTable sets the `COND` and `ASSIGN` fields as `TRUE` since there are no additional constraints that the tuples in the table must satisfy.

PROPERTIES: VeriTable is sound and complete. These two properties directly follow from Lemma 1.

5.3.3 SPJ AR

Alg. 7 illustrates the VeriSPJ procedure for SPJ ARs. VeriSPJ leverages two procedures from proving query equivalence under set semantics: ConstExpr and ConstructPred .

ConstExpr takes a vector of projection expressions and a symbolic tuple as inputs, and returns a new symbolic tuple with additional constraints `ASSIGN` that models the relation between variables. This new symbolic tuple represents the modified tuple based on the vector of projection expressions. ConstructPred takes a predicate and a symbolic tuple as the input and returns a boolean formula `COND` with additional constraints `ASSIGN`. `COND` symbolically represents the result of evaluating the predicate on the symbolic tuples. ConstructPred supports higher-order predicates, such as `EXISTS`, by encoding them as an uninterpreted function.

Algorithm 7: Comparison function for SPJ ARs

Input : A pair of SPJ ARs
Output : QPSR of given SPJ ARs or NULL

1 **Procedure** *VeriSPJ*($\texttt{SPJ}(\vec{e_1}, p_1, \vec{o_1}), \texttt{SPJ}(\vec{e_2}, p_2, \vec{o_2})$)
2 $\{\vec{QPSR}\} \leftarrow \textsf{VeriVec}(\vec{e_1}, \vec{e_2})$
3 **foreach** $\vec{QPSR} \in \{\vec{QPSR}\}$ **do**
4 $(\vec{\texttt{COLS}}_1, \vec{\texttt{COLS}}_2, \texttt{COND}, \texttt{ASSIGN}) \leftarrow \textsf{Compose}(\vec{QPSR})$
5 $(\texttt{COND}_1, \texttt{ASSIGN}_1) \leftarrow \textsf{ConstructPred}(p_1, \vec{\texttt{COLS}}_1)$
6 $(\texttt{COND}_2, \texttt{ASSIGN}_2) \leftarrow \textsf{ConstructPred}(p_2, \vec{\texttt{COLS}}_2)$
7 **if** $\texttt{COND}_1 \leftrightarrow \texttt{COND}_2$ **then**
8 $(\vec{\texttt{COLS}}'_1, \texttt{ASSIGN}_3) \leftarrow \textsf{ConstExpr}(\vec{\texttt{COLS}}_1, \vec{o_1})$
9 $(\vec{\texttt{COLS}}'_2, \texttt{ASSIGN}_4) \leftarrow \textsf{ConstExpr}(\vec{\texttt{COLS}}_2, \vec{o_2})$
10 $\texttt{COND} \leftarrow \texttt{COND}_1 \wedge \texttt{COND}_2 \wedge \texttt{COND}$
11 $\texttt{ASSIGN} \leftarrow \texttt{ASSIGN} \wedge \texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2 \wedge \texttt{ASSIGN}_3 \wedge \texttt{ASSIGN}_4$
12 **return** $(\vec{\texttt{COLS}}'_1, \vec{\texttt{COLS}}'_2, \texttt{COND}, \texttt{ASSIGN})$
13 **end**
14 **end**
15 **return** NULL

CARDINAL EQUIVALENCE: As covered in §5.2.1, an SPJ AR first computes the cartesian product of all input ARs as the intermediate table (JOIN). It then selects all tuples in the intermediate table that satisfy the predicate (SELECT), and applies the projection on each selected tuple (PROJECT).

Lemma 2. *A pair of SPJ ARs* $\texttt{SPJ}(\vec{e_1}, p_1, \vec{o_1})$ *and* $\texttt{SPJ}(\vec{e_2}, p_2, \vec{o_2})$ *are cardinally equivalent if there is a bijective map* m *between tuples in intermediate join tables, such that the predicates* p_1 *and* p_2 *always return the same result for the corresponding tuples in* m*.*

To prove that there is a bijective map between the tuples in the two intermediate join tables, VeriSPJ first uses the VeriVec procedure to find a bijective map between sub-ARs such that each pair of sub-ARs are cardinally equivalent. VeriVec exhaustively examines all possible maps and recursively uses VeriCard to verify the cardinal equivalence between two sub-ARs. VeriVec returns all possible candidate maps wherein each pair of sub-ARs are cardinally equivalent ($\{\vec{QPSR}\}$). Each candidate map is represented by a vector of QPSR ($\vec{QPSR}$), wherein each QPSR defines a bijective map between tuples returned by a pair of cardinally equivalent sub-ARs.

VeriSPJ then uses the Compose procedure to construct two symbolic tuples $\vec{\texttt{COLS}}_1$ and $\vec{\texttt{COLS}}_2$ (line 4) that represent a bijective map between the tuples in the two intermediate join tables. These two symbolic tuples are constructed by concatenating symbolic tuples from the QPSRs of sub-ARs based on the order of sub-ARs in the input vectors. Compose also constructs COND and ASSIGN by taking the conjunction of COND and ASSIGN from the QPSRs of sub-ARs, respectively.

VeriSPJ then tries to prove that the two predicates always return the same result for the two symbolic tuples. VeriSPJ first leverages the ConstructPred procedure to encode predicates p_1 and p_2 on $\vec{\texttt{COLS}}_1$ and $\vec{\texttt{COLS}}_2$, respectively (line 6). VeriSPJ uses an SMT solver to prove this property under sub-conditions COND and all relational constraints: ASSIGN, $\texttt{ASSIGN}_1$, $\texttt{ASSIGN}_2$ (line 7). If the property holds, then negation of this property is unsatisfiable:

$$\texttt{COND} \wedge \texttt{ASSIGN} \wedge \texttt{ASSIGN}_1 \wedge \texttt{ASSIGN}_2 \wedge \neg(\texttt{COND}_1 = \texttt{COND}_2)$$

VeriSPJ feeds this formula to an SMT solver. If the solver determines that this formula is unsatisfiable, then we prove $\texttt{COND}_1$ and $\texttt{COND}_2$ are always equivalent when the relational constraints $\texttt{ASSIGN}_0$, $\texttt{ASSIGN}_1$, and $\texttt{ASSIGN}_2$ and sub-conditions COND hold.

Consider the cardinally equivalent SPJ ARs shown in Figure 6. In this case, VeriSPJ first verifies that sub-AR $E11$ is cardinally equivalent to sub-AR $E22$, and sub-AR $E12$ is cardinally equivalent to sub-AR $E21$. Thus, the two intermediate join tables (i.e., cartesian product of sub-tables) are cardinally equivalent. VeriSPJ constructs two symbolic tuples to represent the bijective map between these intermediate join tables by leveraging the two bijective maps between the underlying tables. VeriSPJ then verifies that two corresponding tuples in the map either both satisfy the predicate or not satisfy the predicate. Thus, the bijective map between the tuples in the intermediate join tables is the bijective map between the tuples in the output tables before projection.

QPSR: Since VeriSPJ verifies that the given pair of SPJ ARs are cardinally equivalent, the two symbolic tuples $\vec{\texttt{COLS}}_1$ and $\vec{\texttt{COLS}}_2$ define a bijective map between tuples in the output tables before projection. Projection does not change the bijective map between tuples as it is applied separately on each tuple. Thus, VeriSPJ leverages ConstExpr to construct new symbolic tuples $\vec{\texttt{COLS}}'_1$ and $\vec{\texttt{COLS}}'_2$ based on the vector of projection expressions and the given symbolic tuples. The QPSR consists of the derived symbolic tuples $\vec{\texttt{COLS}}'_1$, $\vec{\texttt{COLS}}'_2$, the conjunction of $\texttt{COND}_1$, $\texttt{COND}_2$ and COND, and the conjunction of all the relational constraints.

PROPERTIES: VeriSPJ is sound. Based on Lemma 2, if VeriSPJ returns the QPSR, then the given SPJ ARs are cardinally equivalent.

Algorithm 8: Comparison function for aggregate ARs

Input : A pair of aggregate ARs
Output : QPSR of given aggregate ARs or `NULL`

Procedure *VeriAgg*(*Aggregate*$(e_1,\ \vec{g_1},\ a\vec{g}\vec{g}_1)$, *Aggregate*$(e_2,\ \vec{g_2},\ a\vec{g}\vec{g}_2)$)
  $QPSR \leftarrow$ VeriCard(e_1, e_2)
  if *QPSR !=* `NULL` **then**
    $(\vec{\texttt{COLS}}_1, \vec{\texttt{COLS}}_2, \texttt{COND}, \texttt{ASSIGN}) \leftarrow QPSR$
    if $\vec{g_1} \leftrightarrow \vec{g_2}$ **then**
      $\vec{\texttt{COLS}}_1 \leftarrow$ InitAgg$(a\vec{g}\vec{g}_1) :: \vec{g_1}$
      $\vec{\texttt{COLS}}_2 \leftarrow$ CtrAgg$(a\vec{g}\vec{g}_1, \vec{\texttt{COLS}}_1, a\vec{g}\vec{g}_2) :: \vec{g_2}$
      return $(\vec{\texttt{COLS}}_1, \vec{\texttt{COLS}}_2, \texttt{COND}, \texttt{ASSIGN})$
    end
  end
  else return `NULL`;

In general, VeriSPJ is *not complete*. The reasons are threefold. First, the SMT solver is only complete for linear operators. If the predicates have non-linear operators (e.g., multiplication between columns), then the solver may return `UNKNOWN` when it should return `UNSAT`. Second, SPES encodes all user-defined functions, string operations, and higher-order predicates as uninterpreted functions. These encodings do not preserve the semantics of these operations. Third, VeriCard is not complete (§5.3.1).

VeriSPJ procedure is complete if all input ARs for the given two SPJ ARs are table ARs, and the SMT solver can determine the satisfiability of the predicates. This is because the problem of deciding equivalence of two conjunctive (i.e., SPJ) queries is decidable [30].

5.3.4 Aggregate AR

Alg. 8 illustrates the VeriAgg procedure for aggregate ARs.

CARDINAL EQUIVALENCE: An aggregate AR groups the tuples in the input table based on the `GROUP BY` column set, then returns a tuple by applying the aggregate function on each group.

Lemma 3. *Two aggregate ARs* *Aggregate*$(e_1,\ \vec{g_1},\ a\vec{g}\vec{g}_1)$ *and* *Aggregate*$(e_2,\ \vec{g_2},\ a\vec{g}\vec{g}_2)$ *are cardinally equivalent if two conditions are satisfied:* ***(1)*** *the two input sub-ARs* e_1 *and* e_2 *are cardinally equivalent;* ***(2)*** *for any two pairs of corresponding tuples in a bijective map of the QPSR of* e_1 *and* e_2*, two tuples in* e_1 *belong to the same group as defined by* g_1 *iff their associated tuples in* e_2 *belong to the same group as defined by* g_2*.*

VeriAgg first recursively invokes the VeriCard procedure to determine the cardinal equivalence of the two input sub-ARs e_1 and e_2 (line 2). If VeriCard returns the QPSR of e_1 and e_2, then VeriAgg has proved the first condition in Lemma 3.

To prove the second condition, VeriAgg collects the symbolic tuples $\texttt{COL}\vec{\texttt{S}}_1$ and $\texttt{COL}\vec{\texttt{S}}_2$ from the QPSR. Since these two symbolic tuples define a bijective map between tuples returned by e_1 and e_2, VeriAgg replaces all variables in $\texttt{COL}\vec{\texttt{S}}_1$ and $\texttt{COL}\vec{\texttt{S}}_2$ by a set of fresh variables to generate a second pair of symbolic tuples $\texttt{COL}\vec{\texttt{S}}'_1$ and $\texttt{COL}\vec{\texttt{S}}'_2$ that represents the same bijective map with different tuples.

I decompose the proof for the second condition into two stages (line 5). In the first stage, I want to prove that if $\texttt{COL}\vec{\texttt{S}}_1$ and $\texttt{COL}\vec{\texttt{S}}'_1$ belong to the same group, then $\texttt{COL}\vec{\texttt{S}}_2$ and $\texttt{COL}\vec{\texttt{S}}'_2$ also belong to the same group. To prove this, VeriAgg extracts the `GROUP BY` column sets $\vec{g_1}$, $\vec{g'_1}$, $\vec{g_2}$ and $\vec{g'_2}$ from $\texttt{COL}\vec{\texttt{S}}_1$, $\texttt{COL}\vec{\texttt{S}}'_1$, $\texttt{COL}\vec{\texttt{S}}_2$ and $\texttt{COL}\vec{\texttt{S}}'_2$, respectively. It then attempts to prove the property:

$$(\texttt{COND} \wedge \texttt{ASSIGN} \wedge \vec{g_1} = \vec{g'_1}) \implies \vec{g_2} = \vec{g'_2}$$

VeriAgg sends the negation of this property to the solver. If the solver decides that this formula is unsatisfiable, then it is impossible to find two tuples returned by e_1 that are assigned to the same group by $\vec{g_1}$, such that their corresponding tuples returned by e_2 are assigned to different groups by $\vec{g_2}$. In the second stage, I use the same technique in the reverse direction of the implication.

Consider the cardinally equivalent aggregate ARs shown in Figure 7. VeriAgg first verifies that the two input ARs $E1$ and $E2$ are cardinally equivalent, and then constructs the QPSR to represent the bijective map between their returned tuples. VeriAgg then verifies that if two arbitrary tuples in $E1$ belong to same group (e.g., first two tuples), then the two corresponding tuples in $E2$ also belong to the same group. It also verifies that if two arbitrary tuples in $E1$ belong to different groups (e.g., first and third tuples), then the two corresponding tuples in $E2$ also belong to different groups. VeriAgg verifies two aggregate ARs are cardinally equivalent by verifying that they emit the same number of groups.

QPSR: VeriAgg constructs the QPSR of two given aggregate ARs after proving they are cardinally equivalent. $\texttt{COL}\vec{\texttt{S}}_1$ and $\texttt{COL}\vec{\texttt{S}}_2$ define a bijective map between tuples returned by input ARs, and can also be used to define a bijective map between groups in two aggregate ARs. If two aggregate

Algorithm 9: Comparison function for Union ARs

```
Input  : A pair of union ARs
Output : QPSR of given two Union ARs or NULL
Procedure VeriUnion(Union(e1⃗), Union(e2⃗))
    {QPSR⃗} ← VeriVec(e1⃗, e2⃗)
    if {QPSR⃗} != ∅ then
        COLS1⃗ ← Init(); COLS2⃗ ← Init()
        QPSR⃗ ← {QPSR⃗}
        (COND, ASSIGN) ← ConstAssign(QPSR⃗, COLS1⃗, COLS2⃗)
        return (COLS1⃗, COLS2⃗, COND, ASSIGN)
    end
    else return NULL;
```

functions in $a\vec{g}\vec{g}_1$ and $a\vec{g}\vec{g}_2$ are the same and operate on same values (i.e., input columns of the symbolic tuples are the same), then the aggregate values in the output tuples are the same, since each group contains the same number of tuples.

VeriAgg invokes the InitAgg procedure on $a\vec{g}\vec{g}_1$ to construct a vector of pairs of new symbolic variables as the symbolic tuples for aggregate functions. In each pair of symbolic variables, the first variable represents the aggregate value. The second variable indicates if the aggregate value is `NULL`. VeriAgg concatenates the `GROUP BY` column set $\vec{g_1}$ with the symbolic tuple $\vec{\texttt{COLS}}_1$. VeriAgg then invokes the CtrAgg procedure to construct the symbolic columns for $a\vec{g}\vec{g}_2$, and then concatenates with the `GROUP BY` column set $\vec{g_2}$. CtrAgg uses the same pairs of symbolic variables for all aggregation operations in $a\vec{g}\vec{g}_2$, where the aggregation function type and operand columns are the same in $a\vec{g}\vec{g}_1$. VeriAgg propagates `COND` and `ASSIGN` into the sub-QPSRs.

PROPERTIES: VeriAgg is sound. Based on Lemma 3, if VeriAgg returns the QPSR, then the two given aggregate ARs are cardinally equivalent. This is because the two symbolic tuples $\vec{\texttt{COLS}}_1$ and $\vec{\texttt{COLS}}_2$ are constructed from corresponding groups. Thus, $\vec{\texttt{COLS}}_1$ and $\vec{\texttt{COLS}}_2$ define a bijective map between tuples returned by the two aggregate ARs.

VeriAgg is not complete. The sources of incompleteness are threefold: (1) incompleteness of VeriCard, (2) limitations of the SMT solver, and (3) when VeriCard returns the QPSR of two input sub-ARs, the symbolic tuples in the QPSR define only one possible bijective map between tuples in the input tables. If VeriAgg fails to prove the second condition in Lemma 3, it is still possible that there exists another bijective map that satisfies the second condition.

5.3.5 Union AR

Alg. 9 illustrates the VeriUnion procedure for union ARs.

CARDINAL EQUIVALENCE:

Lemma 4. *Two union ARs* Union($\vec{e_1}$) *and* Union($\vec{e_2}$) *are cardinally equivalent if there exists a bijective map between the two input sub-ARs* $\vec{e_1}$ *and* $\vec{e_2}$*, such that each pair of ARs are cardinally equivalent.*

The lemma follows from the semantics of the union AR. VeriUnion procedure invokes VeriVec to find a bijective map between $\vec{e_1}$ and $\vec{e_2}$ (line 2), such that each pair of ARs are cardinally equivalent.

QPSR: VeriVec finds all candidate bijective maps ($\{\vec{QPSR}\}$) between two input sub-ARs $\vec{e_1}$ and $\vec{e_2}$, such that each pair of sub-ARs are cardinally equivalent. In each candidate bijective map ($\vec{QPSR}$), a vector of QPSRs is constructed such that each QPSR defines a bijective map between tuples returned by a pair of sub-ARs. VeriUnion gets an arbitrary $\vec{QPSR}$ (i.e., one candidate bijective map between the sub-ARs). It seeks to construct a bijective map between tuples returned by two union ARs that preserves all of the bijective maps between tuples returned by sub-ARs in that $\vec{QPSR}$. It first constructs two fresh symbolic tuples $\vec{\texttt{COLS}}_1$ and $\vec{\texttt{COLS}}_2$. It then invokes the ConstAssign procedure to set `ASSIGN` such that both $\vec{\texttt{COLS}}_1$ and $\vec{\texttt{COLS}}_2$ are always equivalent to the symbolic tuples in one sub-QPSR returned by VeriVec, and `COND` such that `COND` in one sub-QPSR holds when symbolic tuples equal to the tuples in that sub-QPSR. ConstAssign creates a vector of boolean variables to set these constraints. VeriUnion returns these two symbolic tuples, `COND`, and `ASSIGN` as the QPSR of the given union ARs.

PROPERTIES: VeriUnion is sound. Based on Lemma 4, if VeriUnion returns the QPSR, then the two union ARs are cardinally equivalent. The symbolic tuples $\vec{\texttt{COLS}}_1$ and $\vec{\texttt{COLS}}_2$ define a bijective map between tuples returned by two union ARs that preserves all of the bijective maps between tuples in their cardinally equivalent sub-ARs.

VeriUnion is incomplete. The sources of incompleteness are threefold: (1) incompleteness of VeriCard, (2) limitations of the SMT solver, and (3) two union ARs may be cardinally equivalent even if there is no bijective map between their sub-ARs such that each pair of sub-ARs is cardinally equivalent.

Table 7: Comparative analysis between SPES, EQUITAS, and UDP - The results include the number of query pairs in the CALCITE benchmark that these tools support, the number of pairs whose equivalence they can prove, and the average time they take to determine query equivalence.

QE Tool	Supported Semantics	Supported Pairs	Proved Pairs	Average Time (s)
SPES	Bag	120	90	0.05
EQUITAS	Set	91	67	0.15
UDP	Bag	39	34	4.16

QE Tool	Supported Semantics	USPJ Pairs	Average Time (s)	Aggregate Pairs	Average Time (s)	Outer-Join Pairs	Average Time (s)
SPES	Bag	39	0.3	42	0.6	20	0.9
EQUITAS	Set	28	0.10	32	0.19	9	0.19
UDP	Bag	21	2.7	11	6.9	–	–

5.4 Evaluation

In this section, I describe my implementation and evaluation of SPES. I begin with a description of our implementation in §5.4.1. I next report the results of a comparative analysis of SPES against UDP [28] and EQUITAS, state-of-the-art automated query equivalence verifiers based on AR and SR, respectively in §5.4.2. I then quantify the efficacy of SPES in identifying overlapping queries across production SQL queries in §5.4.3. I conclude with the limitations of the current implementation of SPES in §5.4.4.

5.4.1 Implementation

The architecture of SPES is illustrated in Figure 8. SPES takes a pair of SQL queries as inputs and returns a boolean decision that indicates whether they are fully equivalent. The query equivalence verification pipeline consists of three components: ❶ The compiler converts the given queries to logical query execution plans. I use the open-source CALCITE framework [3]. ❷ SPES operates on these logical plans in two stages. First, it converts them to their ARs and normalizes these ARs. Next, it uses the third component to verify the cardinal equivalence of ARs and then constructs their QPSR. It also uses the third component for verifying the properties of QPSR to determine full equivalence. This component is implemented in Java (2,065 lines of code). ❸ The third component is an SMT solver Z3 that SPES leverages for determining the satisfiability of FOL formulae [14].

5.4.2 Comparative Analysis

BENCHMARK: I use queries in the test suite of Apache CALCITE [3] as our benchmark. This test suite contains 232 semantically equivalent query pairs. The reasons for using this benchmark are twofold. First, the CALCITE optimizer is widely used in data processing engines [4, 5, 6, 7, 8]. So, it covers a wide range of SQL features[2]. Second, since UDP and EQUITAS are both evaluated on this query pair benchmark, I can quantitatively and qualitatively compare the efficacy of these tools. I send every query pair with the schemata of their input tables to SPES and ask it to check their query equivalence. I conduct this experiment on a commodity server (Intel Core i7-860 processor and 16 GB RAM).

AUTOMATED SQL QUERY EQUIVALENCE VERIFIERS: The results of this experiment are shown in Table 7. I compare SPES against EQUITAS in the same environment. I present the results reported in the UDP paper [28][3].

SPES proves the equivalence of a larger set of query pairs (90/232) compared to UDP (34/232) and EQUITAS (67/232). SPES currently supports 120 out of 232 pairs. The un-supported queries either: (1) contain `q` features that are not yet supported (e.g., `CAST`), or (2) cannot be compiled by CALCITE due to syntax errors. Among the 120 pairs supported by SPES, it proves that 90 pairs (75%) are equivalent under bag semantics. In contrast, UDP proves the equivalence of 34 pairs under bag semantics. EQUITAS proves the equivalence of 67 pairs, but only under set semantics. I group the proved query pairs into three categories:

- **USPJ:** Queries that are union of `SELECT-PROJECT-JOIN`.
- **Aggregate:** Queries containing at least one aggregate.
- **Outer-Join:** Queries containing at least one outer `JOIN`.

Table 7 reports the number of pairs proved by UDP and EQUITAS in each category. The number of proved pairs containing outer `JOIN` is not known in case of UDP. SPES outperforms the other tools on queries containing aggregate and outer `JOIN` operators.

[2]The test cases were obtained from the open-sourced COSETTE repository [10].

[3]I were unable to conduct a comparative performance analysis under the same environment since UDP is currently not open-sourced.

Table 8: Efficacy of SPES on Production Queries - "Highest Query Frequency" indicates the highest frequency of a query in equivalent query pairs. “Compared Query Pairs" refers to number of query pairs that operate on the same set of input tables.

Query Set	Number of Queries	Queries with Overlapping Computation	Highest Query Frequency
Set 1	3285	943	52
Set 2	3633	984	97
Set 3	2568	664	30
Total	9486	2591 (27%)	–

Query Set	Compared Query Pairs	Equivalent Query Pairs	Query Pairs with Aggregate and Joins
Set 1	122900	3344	653
Set 2	55311	7225	4822
Set 3	15442	1521	356
Total	193633	12090	5831 (48%)

I next compare the average time taken by SPES, UDP and SPES to prove the equivalence of a pair of queries in each category. This is an important metric for a cloud-scale tool that must be deployed in a DBaaS platform. I only compute this metric for the pairs that these tools can prove. SPES, UDP, and EQUITAS take 0.05 s, 4.16 s, and 0.15 s on average to prove query equivalence. So, SPES is 83× faster than UDP and 3× faster than EQUITAS on this benchmark.

5.4.3 Efficacy on Production Queries

In this experiment, I quantify the efficacy of SPES in detecting overlap in production q queries. I leverage three sets of real production queries from Ant Financial [2], a financial technology company. These queries are used to detect fraud in business transactions. In each set, I run SPES on each pair of queries that operate on the same set of input tables. If SPES decides that a given pair of queries are not equivalent, then I check any constituent sub-queries that operate on the same input tables. I skip checking queries containing only table scans and those that only differ in the parameters passed on to their predicates. This is because SPES trivially proves their equivalence and the computational resources needed for evaluating such queries are negligible.

Table 8 presents the results of this experiment. SPES effectively identifies overlap between complex analytical queries. Among 9486 queries, SPES finds overlapping computation between 2591 (27%) queries, while EQUITAS only finds overlapping computation between 1126 (12%) queries. I also report the highest frequency of queries present in these pairs that are repeatedly executed in the workload. In practice, most of the computational resources are expended on executing queries

containing aggregate functions or different types of join. Among 12090 equivalent pairs, 5831 (48%) contain join and aggregate operations. This illustrates that SPES works well on queries containing these operators.

QUERY COMPLEXITY: Figure 9 illustrates the complexity of queries in this workload. I compute the distribution of the number of algebraic expressions (i.e., sub-ARs) in a given query (complex queries will have a larger set of expressions). I found that the average number of algebraic expressions in the Ant Financial workload (45.38) is 8× larger than that in the CALCITE benchmark (5.37).

5.4.4 Limitations

In general, the problem of deciding query equivalence is undecidable [18]. Among the 120 query pairs supported by SPES, it cannot prove the query equivalence of 30 pairs. I classify them into three categories: (1) lack of normalization rules [22], (2) support for integrity constraints [7], and (3) support for type casting [1].

NORMALIZATION RULES: SPES can verify the cardinal equivalence of two ARs only if it can normalize them into the same type of AR using a set of pre-defined semantically-equivalent rewrite rules (§5.3.1). I will need to introduce additional normalization rules for ARs with: (1) union and aggregate [15], (2) join and aggregate [7], and (3) multiple aggregates with a complex relationship [2]. Adding these rewrite rules in the normalization stage will enable SPES to prove the query equivalence of these 22 pairs. However, that will also increase the average query equivalence verification time. Furthermore, these rules are not required for supporting production queries discussed in §5.4.3.

INTEGRITY CONSTRAINTS: SPES currently does not support integrity constraints (e.g., distinct values, foreign keys, and primary keys). I will need to encode these integrity constraints in our normalization rules. For example, I may normalize an `OUTER JOIN` operation based on a foreign key to an `INNER JOIN` operation.

$$\text{E-TABLE}\ \frac{}{\langle \mathtt{Ts} \models \textsc{Table}(n) \rangle \Downarrow [t | \forall t \in n]}$$

$$\text{E-SPJ}\ \frac{\begin{array}{c}\vec{\mathsf{e}} = e_0, e_1, \ldots, e_n \\ \langle \mathtt{Ts} \models e_0 \rangle \Downarrow T_0 \ldots \langle \mathtt{Ts} \models e_n \rangle \Downarrow T_n\end{array}}{\langle \mathtt{Ts} \models \textsc{SPJ}(\vec{\mathsf{e}}, \textsc{p}, \vec{o}) \rangle \Downarrow [(\vec{o}(t) | \forall t \in (T_0 \times \cdots \times T_n), p(t)]}$$

$$\text{E-AGG}\ \frac{\langle \mathtt{Ts} \models \mathsf{e} \rangle \Downarrow T_0}{\langle \mathtt{Ts} \models \mathsf{Aggregate}(\mathsf{e},\ \vec{g},\ a\vec{g}g) \rangle \Downarrow [a\vec{g}g(t) | \forall t \in part(T_0, \vec{g})]}$$

$$\text{E-UNION}\ \frac{\begin{array}{c}\vec{\mathsf{e}} = e_0, e_1, \ldots, e_n \\ \langle \mathtt{Ts} \models e_0 \rangle \Downarrow T_0 \ldots \langle \mathtt{Ts} \models e_n \rangle \Downarrow T_n\end{array}}{\langle \mathtt{Ts} \models \textsc{Union}\ \vec{\mathsf{e}} \rangle \Downarrow [t | \forall t \in T_0 + \cdots + T_n]}$$

Figure 5: Semantics – Semantics of AR used in SPES

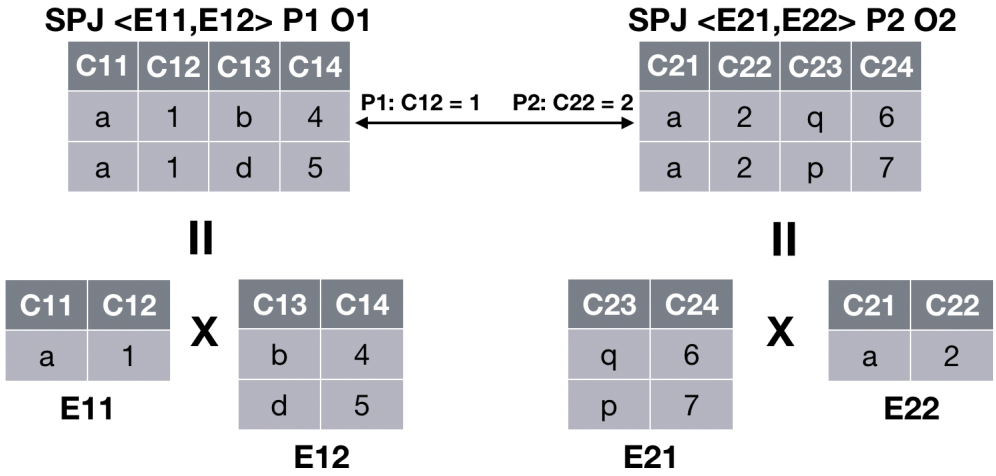


Figure 6: SPJ ARs – Cardinally equivalent SPJ ARs.

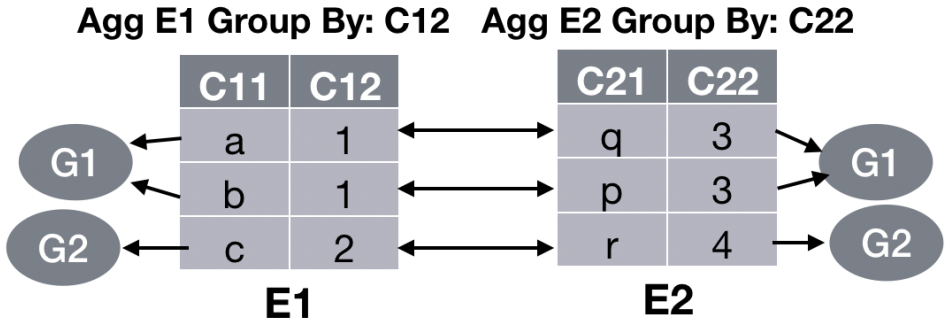


Figure 7: Aggregate ARs – Cardinally equivalent aggregate ARs.

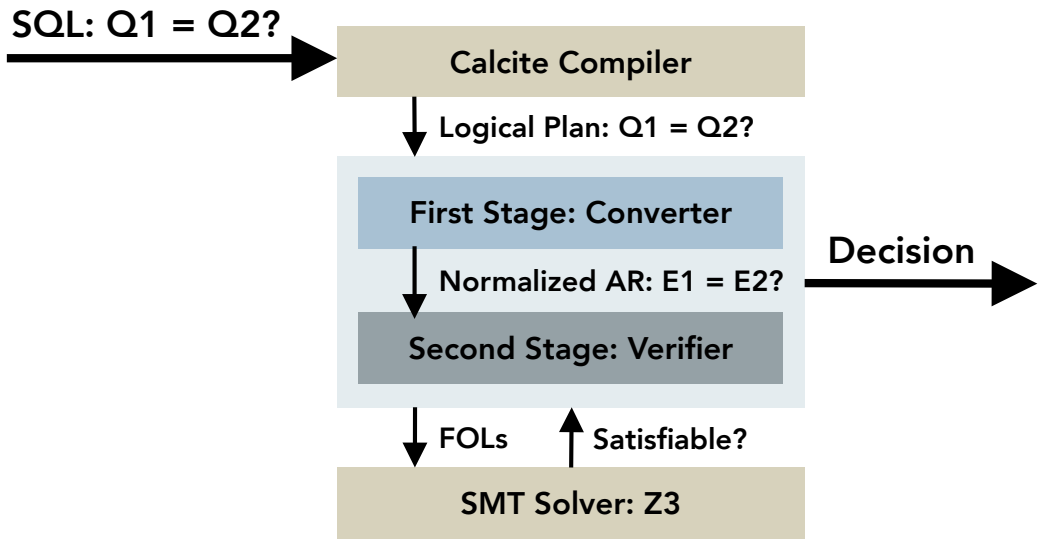


Figure 8: Query Equivalence Verification Pipeline - The pipeline for determining the equivalence of SQL queries.

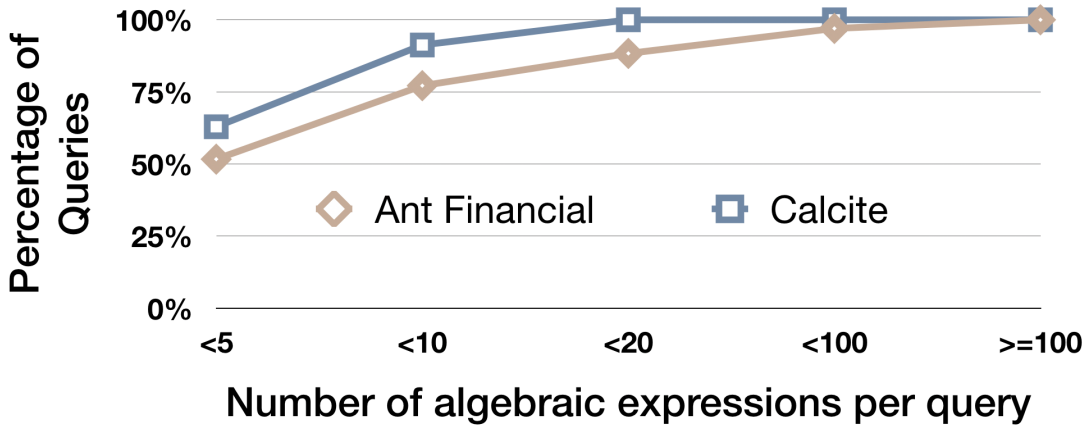


Figure 9: Complexity of Production Queries - I quantify the complexity of production queries in the Ant Financial workload by measuring the number of algebraic expressions (sub-ARs) in each query.

CHAPTER VI

OPTIMIZING QUERIES WITH LEARNED PREDICATE

In previous two chapter, I present two symbolic representation based appraoches that proving query equivalence under set and bag semantics. These two symbolic representation based appraoches leverage the STM solver to proving query equivlaence based on the semantics rather than synatx. These two approahces enable more aggressive transformation rules in query optimization stage. In this chapter, I present a machine learning based query optimization technqiue with verification that guarantee the correctness.

6.1 Overview

In this section, I first present an example that motivates the need for this new query optimization rule in Section 6.1.1. I then describe the overview of this counter-example guided learning with verification optimization rule in Section 6.1.2. I finally use the gvien example to illustrate how this approach rewrites the original query and speed up the exectuion in Section 6.1.3.

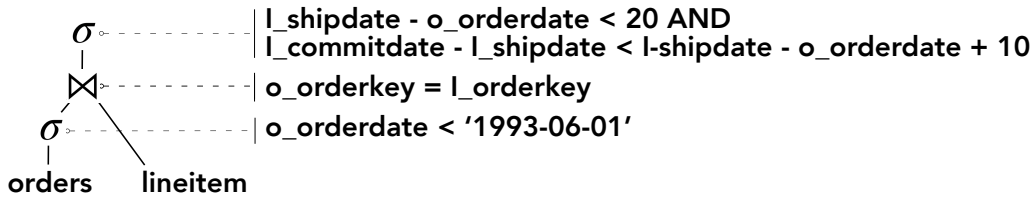
6.1.1 Query Example

I now motivate the need for automatically synthesizing predicates using an example. Consider the following query derived from the TPC-H benchmark [79].

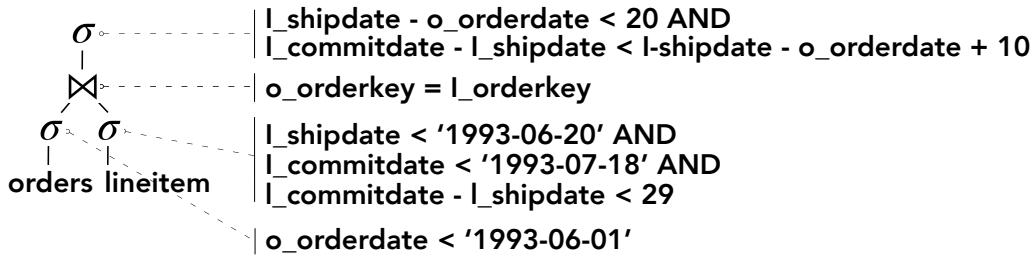
```
Q1: SELECT * FROM lineitem, orders  WHERE o_orderkey = l_orderkey
      AND l_shipdate - o_orderdate < 20 AND o_orderdate < '1993-06-01'
      AND l_commitdate - l_shipdate < l_shipdate - o_orderdate + 10;
```

This query is joining the $lineitem$ and $orders$ tables and applying a set of predicates. It is representative of analytical queries in on-line analytical processing (OLAP) and hybrid transaction-analytical processing (HTAP) applications [74, 64]. The tables are joined based on the order key. The other predicates in the query apply the following conditions:

- The ship date ($l\_shipdate$) is no later than 20 days from the order date ($o\_orderdate$).



(a) Logical Plan for Q1



(b) Logical Plan for Q2

Figure 10: Logical Query Execution Plans – Queries Q1 and Q2 are semantically-equivalent. However, the optimizer computes a better query execution plan for Q2.

- The gap between the commit ($l\_commitdate$) and ship dates is 10 days shorter than that between the ship and order date.
- The order date is earlier than 1993-06-01.

I run this query in the Postgres DBMS (v12) [13]. The query optimizer constructs the logical query execution plan P_1 shown in Figure 10a. With this plan, the query execution engine first filters the tuples in $orders$ using this predicate: $o\_orderdate < 1993-06-01$. It then applies an inner join of the filtered table and the $lineitem$ table using the join predicate ($o\_orderkey = l\_orderkey$). Lastly, it applies another filter on the joined table with this complex predicate: $l\_shipdate - o\_orderdate < 20$ && $l\_commitdate - l\_shipdate < l\_shipdate - o\_orderdate + 10$ to obtain the final output table.

I may rewrite $Q1$ into the following query $Q2$:

```
Q2:  SELECT * FROM lineitem, orders WHERE o_orderkey = l_orderkey
       AND l_shipdate - o_orderdate < 20 AND o_orderdate < '1993-06-01'
       AND l_commitdate - l_shipdate < l_shipdate - o_orderdate + 10
       AND l_shipdate < '1993-06-20' AND l_commitdate < '1993-07-18'
       AND l_commitdate - l_shipdate < 29;
```

When I run $Q2$ on Postgres, I obtain a $2\times$ more performant plan P_2 shown in Figure 10b. $Q1$ and $Q2$ are *semantically-equivalent* queries. $Q2$ differs from $Q1$ in that it has three additional predicates:

(1) $l\_shipdate < 1993 - 06 - 20$; **(2)** $l\_commitdate < 1993 - 07 - 18$ and; **(3)** the difference between $l\_commitdate$ and $l\_shipdate$ is less than 29 days. All of these additional conditions may be inferred from the original conditions in $Q1$.

For instance, $Q1$ requires $o\_orderdate$ to be less than $1993 - 06 - 01$ and the difference between $l\_shipdate$ and $o\_orderdate$ to be less than 20 days. Thus, the $l\_shipdate$ must be less than $1993 - 06 - 20$. More importantly, all of these additional *inferred predicates* only depend on columns present in the $lineitem$ table (i.e., they do not depend on columns in both tables and are thus more efficient to compute).

Plan P_2 differs from P_1 in that it applies a filter on the $lineitem$ table before applying the inner join, thereby reducing the number of tuples being joined. The cost of the join operation depends on the number of tuples in each of the tables being joined. Although P_2 contains an additional filter operation on $lineitem$, it is faster to execute than P_1 (while returning the same output table). On the TPC-H dataset (scale factor = 10), $Q2$ (50 s) is $2\times$ faster than $Q1$ (94 s). I defer a detailed description of our empirical setup to §6.4.

DISCUSSION: Postgres generates a more performant logical plan for $Q2$ since it has three additional predicates that only depend on columns in the $lineitem$ table. This allows the optimizer to push down the predicates below the join operator. In contrast, all the conditions in $Q1$ refer to columns in the $orders$ table. So, there is no predicate that may be applied on the $lineitem$ table before the join operator. This example illustrates the benefits of automatically synthesizing predicate that: (1) only depend on a given set of columns (e.g., predicates that only depend on columns in the $lineitem$ table), and (2) preserve the semantics of the original query. Such synthesized predicates will allow the optimizer to generate a faster query execution plan. In particular, the optimizer may leverage additional query rewrite rules that may not be feasible with the original query (e.g., predicate push down for the $lineitem$ table).

PRIOR WORK: Syntax-driven rules such as constant propagation [33] and transitive closure transformation [47] cannot be applied in this case due to their dependence on syntax. For instance, constant propagation is only applicable for equality relation:

$$x = 5 \;\&\&\; x + y = 20 \longrightarrow x = 5 \;\&\&\; 5 + y = 20$$

Original Predicate: a1 - a2 < b1 and b1 + 5 < 10

a1	a2	b1	satisfy?
17	4	any	×
14	2	any	×

(a) FALSE Samples

a1	a2	b1	satisfy?
5	4	2	✓
7	5	3	✓

(b) TRUE Samples

Figure 11: Types of Training Samples – (1) unsatisfaction tuples (i.e., FALSE samples), and (2) satisfaction (i.e., TRUE samples).

Similarly, transitive closure is only applicable for inequality relation when the direction of the inequality is aligned and the expressions syntactically match:

$$y1 > x \;\&\&\; x > y2 \longrightarrow y1 > y2$$

In our motivating example, these heuristics are not capable of inferring the three additional conditions in $Q2$. This is because it requires reasoning about inequality relation with arithmetic operators.

In general, syntax-driven rules cannot handle the complexity of inequality relation, arithmetic operators and combination of predicates using boolean logic. Furthermore, they do not allow the optimizer to constrain the set of columns used in the synthesized predicate. This limits the ability of the optimizer to apply predicate-centric optimization rules. To tackle these challenges, I present a novel technique for learning predicates using a set of counter-examples while preserving the semantics of the query.

6.1.2 Counter-Example Guided Learning

SIA decomposes the problem of synthesizing weaker predicates that only use the given set of columns into two stages: **(1)** generation of training data, and **(2)** learning predicates.

❶ **GENERATION OF TRAINING SAMPLES:** In the first stage, for a given predicate p and a set of columns Cols, SIA leverages an SMT solver to generate the training samples for the second stage [37].

SIA uses the solver to obtain two types of tuples: **(1)** *unsatisfaction* and **(2)** *satisfaction* tuples. While the former set of tuples must not be accepted by the valid optimal synthesized predicate (i.e., FALSE samples), the latter set must be accepted (i.e., TRUE samples). Given a predicate p and a set of columns Cols, an unsatisfaction tuple is a tuple that takes concrete values for all of the columns in

Cols such that it cannot satisfy p, for *all* possible values for other columns *not* in Cols. As shown in Figure 11a, for the FALSE tuples with concrete values for $a1$ and $a2$, there is no possible value for $b1$ such that the entire tuple satisfies the original predicate p. In contrast, a satisfaction tuple is a tuple that takes concrete values for all of the columns in Cols such that it satisfies p, for *at least one* set of appropriate values for other columns *not* in Cols. As shown in Figure 11b, for the TRUE tuples with concrete values for $a1$ and $a2$, there is at least one value for $b1$ such that the entire tuple satisfies p. I defer formal definitions to §6.2.2.

SIA seeks to synthesize a predicate that preserves the semantics of the original query. To accomplish this, the synthesized predicate p_1 must imply the original predicate p. So, it must be a weaker predicate than p (i.e., if a tuple is accepted by p, then it must also be accepted by p_1). Thus, a satisfaction tuple for Cols and p must be accepted by p_1. In contrast, if p_1 is the optimal predicate, an unsatisfaction tuple for a set of columns Cols and p must be rejected by p_1. This is why SIA tries to construct unsatisfaction and satisfaction tuples for Cols and p so that these training samples may be used to learn a valid and optimal p_1. I formalize these properties of unsatisfaction tuple in §6.2.1.

SIA leverages the SMT solver to generate the training samples. For TRUE samples, it encodes that the predicate p over the columns Cols is TRUE in a symbolic formula, and repeatedly feeds it to the solver to obtain a *model* (i.e., a set of concrete values for the symbolic variables that satisfies the constraints in the formula). In each iteration, it adds additional constraints to ensure that the solver generates a new model. In each model generated by the solver, SIA extracts the concrete values for Cols and constructs a TRUE sample. I discuss how SIA encodes p in §6.3.2. For FALSE samples, SIA takes the similar approach but feeds a complementary SMT formula to the solver. I defer a detailed discussion on how SIA generates training samples to §6.3.3.

❷ **LEARNING PREDICATES:** In the second stage, SIA iteratively applies two steps to synthesize a valid optimal predicate: **(1)** learning step, and **(2)** verification and counter-example generation step. Figure 12 illustrates the iterative learning process. In the first step, SIA takes the two sets of training samples generated in the previous stage and seeks to learn a *binary classifier* that separates these two sets. SIA uses linear support vector machines (SVM) for learning the classifier. The reasons for this are twofold. First, SIA must map the binary classifier back to an SQL predicate. By using a linear SVM, SIA quickly maps the classifier to a predicate. Second, SIA must verify the synthesized

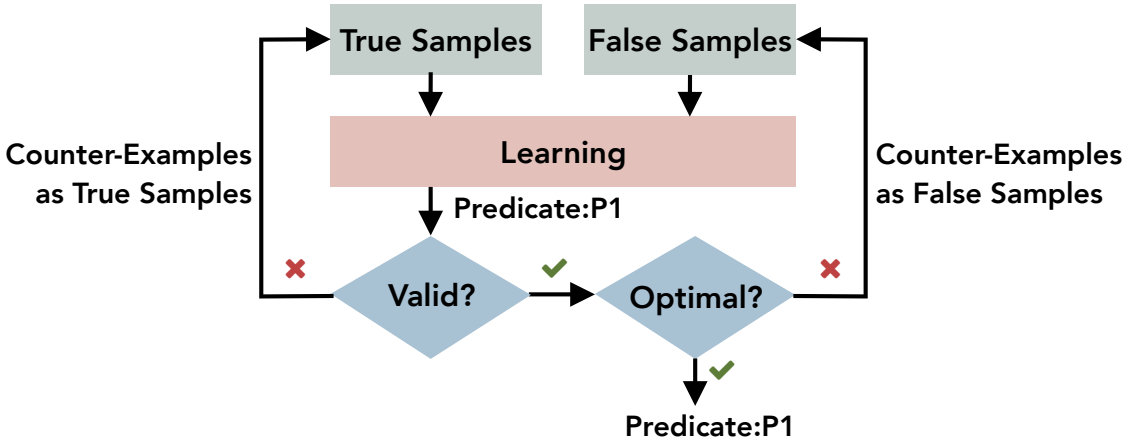


Figure 12: Counter-Example Guided Learning – The iterative learning process used in SIA.

predicate p_1 implies the given predicate p. With linear SVM, the synthesized predicate is guaranteed to be linear (e.g.no multiplication of columns), thus ensuring that the subsequent verification problem is decidable. I describe the learning step in §6.3.4.

The second step consists of verification and generation of counter-examples. Given a predicate p and a learned predicate p_1, SIA uses the SMT solver to verify that p_1 implies p. There are two possibilities. First, if p_1 does not imply p, then p_1 is not a valid predicate (since it does not preserve the semantics of the original query). In this case, SIA uses the solver to generate additional `TRUE` samples. These samples satisfy p but do not satisfy p_1. So, these additional samples are *counter-examples* wherein p_1 fails. I discuss how SIA generates such counter-examples in §6.3.5. SIA then loops back to the learning step with these additional true samples. Next, if p_1 does imply p, then p_1 is valid. However, p_1 may still not be the *optimal* synthesized predicate. This is because there may be a valid synthesized predicate that rejects tuples that are accepted by p_1. I formalize the notion of an optimal synthesized predicate in §6.2.1. In this case, SIA leverages the solver to generate additional `FALSE` training samples (i.e., unsatisfaction tuples that are accepted by p_1). These additional samples are the ones that render p_1 to be sub-optimal. If the solver cannot generate additional `FALSE` samples, then p_1 is optimal. In this case, SIA exits the learning loop and returns p_1. Otherwise, it loops back to the first step with these additional false samples. To bound the query rewriting time, I configure the maximum number of iterations that SIA may take over the learning loop.

I refer to this technique as learning guided by counter-examples. This is because in each iteration of the learning loop, SIA either generates counter-examples that p_1 is supposed to accept but rejects, or that it is supposed to reject but accepts.

6.1.3 Demonstration Example

I next revisit the example in §6.1.1 to illustrate the learning technique. SIA first converts all the columns of `DATE` type to columns of `INTEGER` type by treating a specific date as the origin (i.e., zero), and by encoding other dates with the number of days between them and the *origin date*. For example, in $Q1$, it treats $1993-06-01$ as the origin date. To simplify our presentation, I refer to $l\_commitdate$ by $a1$, $l\_shipdate$ by $a2$, and $o\_orderdate$ by $b1$. With this representation, the conditions in $Q1$ reduce to:

```
a2 - b1 < 20 AND a1 - a2  < a2 - b1 + 10 AND b1 < 0
```

I now seek to synthesize a weaker predicate that only refers to columns $a1$ and $a2$.

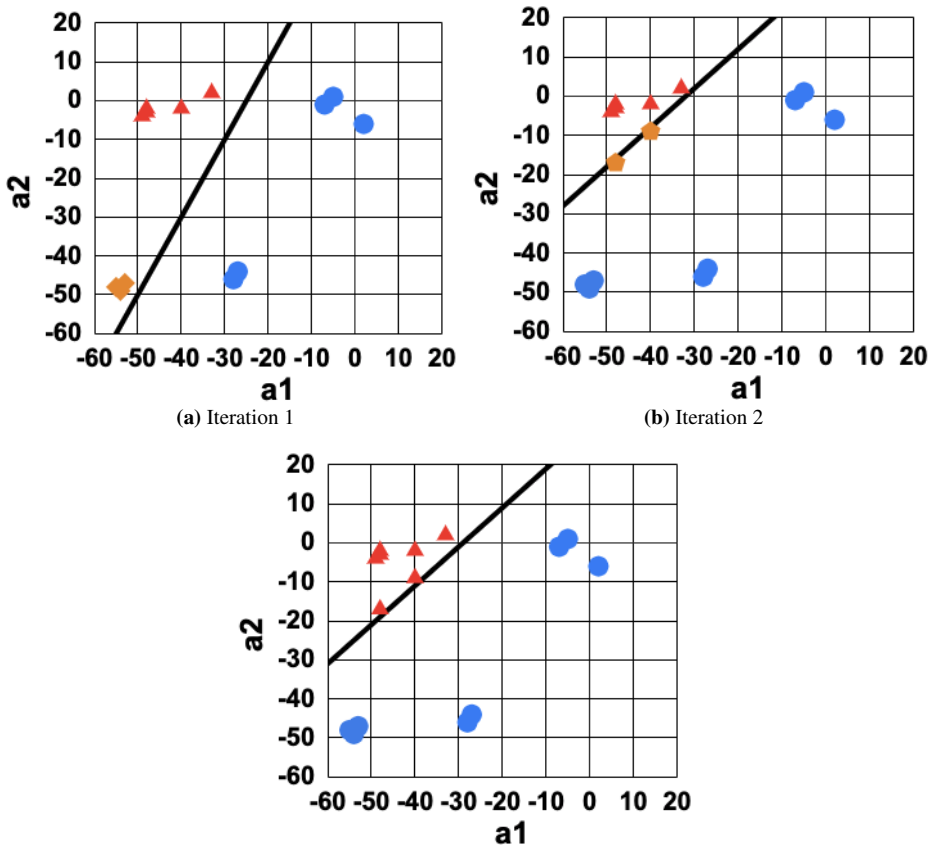
GENERATION OF TRAINING SAMPLES: To generate the initial training samples, SIA first encodes the conditions symbolically as a set of formulae in first-order logic:

$$a2 - b1 < 20 \wedge a1 - a2 < a2 - b1 + 10 \wedge b1 < 0$$

$a1$, $a2$, and $b1$ are symbolic variables in this formula that represent an arbitrary tuple before the filtering operation. I defer a discussion on how SIA encodes conditions and why I choose this encoding schema to §6.3.2.

To generate the initial `TRUE` samples, SIA repeatedly feeds the symbolic formula to the solver. In each iteration, it generates a model with concrete values for $a1$, $a2$ and $b1$ that satisfy the original predicate p. It then adds additional constraints so that the model obtained in the next iteration is not the same as the one obtained in prior iterations. Since SIA seeks to synthesize a weaker predicate that only uses columns $a1$ and $a2$, it only retains the concrete values for $a1$ and $a2$ from the models returned by the solver. For $Q1$, it generates the following pairs of values as the initial `TRUE` samples.

```
True: (-5,1); (2,-6); (-27,-44); (-28,-46); (-7,-1)
```



(a) Iteration 1 **(b)** Iteration 2

(c) Iteration 3

Figure 13: Learning Process – Three iterations of the learning loop in SIA guided by counter-examples.

To generate the initial FALSE samples, SIA repeatedly feeds the negation of the symbolic formula to the solver with additional constraints to force the solver to generate new values for $a1$ and $a2$. This formula represents that values of columns not in the given set do not satisfy the predicate. In each iteration, it generates a model with concrete values for $a1$ and $a2$ such that there is no possible values for $b1$ that satisfy the original predicate p. For $Q1$, it generates the following pairs of values as the initial FALSE samples.

```
False: (-40,-2); (-56,-2); (-53,-2); (-48,-2)
```

LEARNING GUIDED BY COUNTER-EXAMPLES: SIA iteratively applies two steps to synthesize a weaker predicate p_1. In the first iteration, it begins with the learning step using the initial TRUE and FALSE samples. SIA uses a linear SVM to learn a disjunction of linear predicates on columns $a1$ and $a2$. It learns the following linear predicate from these samples:

$$2 * a1 + a2 + 50 > 0$$

Figure 13 illustrates the learning process. As shown in Figure 13a, the predicate is represented by the black line that separates all TRUE samples (blue circles) from FALSE samples (red triangles). SIA then uses the solver to verify that the newly learned predicate is weaker than p. However, its verification algorithm determines that this predicate is not weaker than p. So the learned predicate is not valid.

SIA then generates counter-examples, which are tuples that satisfy p, but do not satisfy the learned predicate. The following pairs of values are generated as counter-examples:

```
False: (-53,-47); (-54,-49); (-55,-48);
```

For example, with $(-53, -47)$, if I set $b1$ to -5, then the tuple satisfies p. But this pair of values is rejected by the current p_1. I represent these counter-examples using yellow diamonds on the bottom left in Figure 13a. These counter-examples are TRUE samples, but they are wrongly classified by the learned predicate as FALSE.

In the next iteration, SIA adds these counter-examples to TRUE samples, and applies the same learning algorithm. It learns the following linear predicate with the new samples:

$$a1 - a2 + 32 > 0$$

As shown in Figure 13b, the newly learned predicate (shifted black line) correctly classifies the counter-examples generated in the previous iteration as TRUE samples (now represented using blue circles).

SIA again uses the solver to verify that current p_1 is weaker than p. Although the current p_1 is valid, it determines that a learned predicate *stronger* than p_1 (and still weaker than p) exists. SIA then generates counter-examples that are rejected by p, but accepted by the current p_1. The following pairs of values are new generated counter-examples:

```
False: (-40,-9); (-48,-17);
```

For example, with $(-40, -9)$, there is no possible value for $b1$ such that the tuple satisfies p. This pair of values should be rejected by the optimal predicate, but it satisfies the current p_1. These counter-examples are marked using yellow pentagons in Figure 13b. These counter-examples are FALSE samples, but they are wrongly classified by the learned predicate as TRUE.

In the next iteration, SIA adds these counter-examples to FALSE samples, and applies the same learning algorithm. It learns the following linear predicate with the new samples:

$$a1 - a2 + 29 > 0$$

As shown in Figure 13b, this learned predicate separates all the TRUE samples from the FALSE samples including newly added counter-examples (now marked using red triangles). Lastly, SIA verifies if the learned predicate p_1 is valid. If it cannot generate additional counter-examples, then p_1 is also optimal. In this manner, it synthesizes a valid optimal predicate referring to only columns $a1$ and $a2$.

6.2 Problem Formulation

In this section, I now formalize the problem of learning a valid, optimal predicate. I first define the syntax of predicates that SIA supports and our problem formulation in §6.2.1. I then present the key conceptual insights in §6.2.2.

6.2.1 Problem Definition

The syntax of the set of predicates supported by SIA is given by:

$$\mathsf{P} ::= \mathsf{Bin\ E\ cp\ E} \mid \mathsf{Bin\ P\ logic\ P} \mid \textbf{Not}\ \mathsf{P}; \quad \mathsf{E} ::= \mathsf{Column} \mid \mathsf{Const} \mid \mathsf{Bin\ E\ op\ E}$$

$$\mathsf{cp} ::= > \mid < \mid = \mid \leq \mid \geq; \quad \mathsf{op} ::= + \mid - \mid \times \mid \div; \quad \mathsf{logic} ::= \texttt{AND} \mid \texttt{OR}$$

A predicate P is either: (1) a comparison of two arithmetic expressions, (2) a conjunction or disjunction of two predicates, or (3) a negation of a predicate. An arithmetic expression E is either a constant, a reference to a column, or a binary expression with four basic arithmetic operators. Each column COL is associated with a data type that is denoted as $\tau_{\texttt{COL}}$.

SIA currently supports the following data types: INTEGER, DOUBLE, DATE, and TIMESTAMP. It transforms the latter two data types to an integral type while preserving the arithmetic and inequality relations of the predicate. It currently does not support the TEXT type. I next present formal definitions of predicates and tuples.

DEFINITIONS: Predicate p is a *predicate over columns* Cols if each column that occurs in the predicate p is in Cols. The set of predicates over Cols is denoted $\mathsf{Preds}_{\mathsf{Cols}}$. Note that each predicate $\mathsf{p} \in \mathsf{Preds}_{\mathsf{Cols}}$ is a predicate over all sets of column Cols' such that $\mathsf{Cols}' \subseteq \mathsf{Cols}$. A *tuple over columns* Cols is a map from each column $\textsc{Col} \in \mathsf{Cols}$ to a value of corresponding column type $\tau_{\textsc{Col}}$. The set of tuples over Cols is denoted $\mathsf{Tuples}_{\mathsf{Cols}}$. Each predicate $\mathsf{p} \in \mathsf{Preds}_{\mathsf{Cols}}$ can be evaluated on each tuple $t \in \mathsf{Tuples}_{\mathsf{Cols}}$ to produce a boolean output, denoted $\mathsf{p}(t)$. If I substitute $t(\textsc{Col})$ for each column $\textsc{Col}$ in p and it evaluates to True (i.e., $\mathsf{p}(t)$ is True), then I say that t *satisfies* p (alternately, that p *accepts* t). If $\mathsf{p}(t)$ is False, then t does not satisfy p (alternately, p *rejects* t).

PREDICATE IMPLICATION: Predicate p *implies* predicate p' if each tuple that satisfies p also satisfies p'.

Definition 5. *Predicate* $p \in \mathsf{Preds}_{\mathsf{Cols}}$ *over columns* Cols implies *predicate* $p' \in \mathsf{Preds}_{\mathsf{Cols}}$ *over* Cols *if for each tuple* $t \in \mathsf{Tuples}_{\mathsf{Cols}}$ *that satisfies* p *(i.e.,* $p(t) = \mathsf{True}$*),* t *also satisfies* p' *(i.e.,* $p'(t) = \mathsf{True}$*).*

The fact that p implies p' is denoted $\mathsf{p} \implies \mathsf{p}'$.

VALID PREDICATES: A valid *dimensionality reduction* of a predicate p is a predicate over a *subset* of the columns of p that is implied by p.

Definition 6. $p' \in \mathsf{Preds}_{\mathsf{Cols}'}$ *is a* valid dimensionality reduction *of predicate* $p \in \mathsf{Preds}_{\mathsf{Cols}}$ *with* $\mathsf{Cols}' \subseteq \mathsf{Cols}$ *if* $p \implies p'$.

Valid dimensionality reduction enables the application of optimization rules related to predicates [81, 58]. For example, it may be used to lower a predicate p on the result of a join operation over columns Cols of *multiple* tables down to a predicate p' over columns Cols' of *one* input table, where $\mathsf{Cols}' \subseteq \textsc{Col}$. The requirement that a dimensionality reduction over Cols' is in $\mathsf{Preds}_{\textsc{Col}}$ ensures that the reduction is defined over the component table. The requirement that a *reduced predicate* p' over Cols' is implied by p ensures that it does not remove tuples that may need to be provided to the join (i.e., ensures soundness). Thus, dimensionality reduction enables the potential application of a predicate push-down below join operator rule that was not previously feasible.

However, not all *valid* dimensionality reductions are useful in practice. For instance, any trivial predicate that is satisfied by all tuples is technically valid. I will be primarily concerned with synthesizing predicates that are as less selective as possible.

Definition 7. $p_1 \in \mathsf{Preds}_{\mathsf{Cols}'}$*, a valid reduction of* $p \in \mathsf{Preds}_{\mathsf{Cols}}$ *(Definition 6) is* optimal *if for each* $p_2 \in \mathsf{Preds}_{\mathsf{Cols}'}$ *that is a valid dimensional reduction of* p *to* Cols'*, it holds that* $p_1 \implies p_2$*.*

I prove that every predicate has an optimal dimensionality reduction to each subset of its columns in §6.2.2. One of our key contribution in SIA is an automatic procedure for synthesizing a valid dimensionality reduction of p to Cols', given a predicate $\mathsf{p} \in \mathsf{Preds}_{\mathsf{Cols}}$ and a set of columns $\mathsf{Cols}' \subseteq \mathsf{Cols}$.

6.2.2 Key Conceptual Insights

Given the problem definition in §6.2.1, I now discuss the key insights for solving it. First, I show that an entire class of tuples (i.e., concrete values of the columns in the predicate) rejected by a given predicate map to an individual tuple rejected by its valid reduced predicate. Second, I show that the property of being an *optimal* valid reduced predicate may be represented as an SMT formula.

DEFINITIONS: To elaborate on the first observation, I first define the *restriction* and *extension* properties of tuples that determine the set of columns that they may refer to. For a tuple $t \in \mathsf{Tuples}_{\mathsf{Cols}}$ and a set of columns $\mathsf{Cols}' \subseteq \mathsf{Cols}$, restriction of t to columns in Cols' is denoted by $t|_{\mathsf{Cols}'}$. In this case, t extends $t|_{\mathsf{Cols}'}$ to Cols. An *unsatisfaction tuple* of a predicate p is a tuple over Cols' that may *only* be extended to form tuples that do not satisfy p.

Definition 8. *For a set of columns* $\mathsf{Cols}' \subseteq \mathsf{Cols}$ *and predicate* $p \in \mathsf{Preds}_{\mathsf{Cols}}$*, tuple* $t \in \mathsf{Tuples}_{\mathsf{Cols}'}$ *is a* feasible restriction *for* p *if some extension of* t *to* Cols *satisfies* p*.*

If $t \in \mathsf{Preds}_{\mathsf{Cols}'}$ is not a feasible restriction for $p \in \mathsf{Preds}_{\mathsf{Cols}}$, then I say that t is an *unsatisfaction tuple* of p.

PROPERTIES OF DIMENSIONALITY REDUCTION: In order to prove the key properties of dimensionality reduction, I will use the following lemma which establishes that predicates over a restricted set of columns treat tuples and their restrictions equivalently.

Lemma 5. *For columns* $\mathsf{Cols}' \subseteq \mathsf{Cols}$ *and predicate* $p \in \mathsf{Preds}_{\mathsf{Cols}'}$, $p(t) = p(t|_{\mathsf{Cols}'})$ *for each tuple* $t \in \mathsf{Tuples}_{\mathsf{Cols}}$.

Lemma 5 follows directly from the semantics of predicate satisfaction.

Valid dimensionality reduction is closed under conjunction.

Lemma 6. *If* $p_0, p_1 \in \mathsf{Preds}_{\mathsf{Cols}'}$ *are valid dimensionality reductions of predicate* $p \in \mathsf{Preds}_{\mathsf{Cols}}$ *to* Cols', *then* $p_0 \wedge p_1$ *is a valid dimensionality reduction of* p *to* Cols'.

Lemma 6 follows directly from Definition 7. So, I omit a formal proof.

Valid dimensionality reductions always *accepts* feasible restrictions. The operational consequence of this lemma is that our synthesizer will label all feasible restrictions as `TRUE` as it iteratively learns dimensionality reductions.

Lemma 7. *For a set of columns* $\mathsf{Cols}' \subseteq \mathsf{Cols}$ *and predicates* $p \in \mathsf{Preds}_{\mathsf{Cols}}$ *and* $p' \in \mathsf{Preds}_{\mathsf{Cols}'}$, p' *is a valid dimensionality reduction of* p *to* Cols' *if and only if* p' *accepts every feasible restriction for* p.

Proof. For the forward direction of this bi-directional implication, let $t \in \mathsf{Tuples}_{\mathsf{Cols}}$ satisfy p. Thus $t|_{\mathsf{Cols}'}$ is a feasible restriction for p, by Definition 8; Since p' is a valid dimensionality reduction of p to Cols' by assumption, p' accepts t by Definition 6. thus p' accepts $t|_{\mathsf{Cols}'}$ by Lemma 5; thus p' accepts every feasible restriction for p.

For the reverse direction of this bi-directional implication, let $t \in \mathsf{Tuples}_{\mathsf{Cols}}$ satisfy p; thus $t|_{\mathsf{Cols}'}$ is a feasible restriction for p, by Definition 8; thus $t|_{\mathsf{Cols}'}$ satisfies p', by assumption; thus t satisfies p', by Lemma 5. p' accepts each tuple that satisfies p; thus, p' is a valid dimensionality reduction of p to Cols' by Definition 6. □

Optimal reduced predicate always *rejects* unsatisfaction tuples. The operational consequence of this lemma is that our synthesizer will label all unsatisfaction tuples as `FALSE` as it iteratively learns dimensionality reductions.

Lemma 8. *For a set of columns* $\mathsf{Cols}' \subseteq \mathsf{Cols}$ *and predicates* $p \in \mathsf{Preds}_{\mathsf{Cols}}$ *and* $p' \in \mathsf{Preds}_{\mathsf{Cols}'}$, p' *is optimal if and only if it rejects each tuple* $t \in \mathsf{Tuples}_{\mathsf{Cols}'}$ *that is an unsatisfaction tuple of* p.

Proof. For the forward direction of this bi-directional implication, assume that p' is an optimal dimensionality reduction and let t be an unsatisfaction tuple of p. Let φ_t be a predicate that exactly

accepts t. $p' \wedge \neg\varphi_t$ is a valid dimensionality reduction of p, by the definition of unsatisfaction tuples. Thus $p' \wedge \neg\varphi_t$ is a valid dimensionality reduction by Lemma 6. Assume that t satisfies p'; then p' does not imply $p' \wedge \varphi_t$ by Definition 5, and thus p' is not optimal by Definition 7, which is a contradiction. The only assumption not in the premise is that t satisfies p'; therefore, t does not satisfy p', by contradiction.

For the reverse direction of this bi-directional implication, let $p'' \in \mathsf{Preds}_{\mathsf{Cols}'}$ be a valid dimensionality reduction of p to Cols' and let $t \in \mathsf{Tuples}_{\mathsf{Cols}'}$ satisfy p'. Thus, t is a feasible restriction of p by the assumption that p' rejects all unsatisfiable tuples. So t satisfies p'', by Lemma 7. Each tuple accepted by p' is accepted by p''; therefore, p' implies p'', by Definition 5; p' implies each valid dimensionality reduction of p; therefore p' is optimal, by Definition 7. □

Based on Lemma 8, given a valid synthesized predicate p_1 for the original predicate p and a set of columns Cols', if there is no unsatisfaction tuple t such that $\mathsf{p}_1(t)$ is `TRUE`, then p_1 is an optimal predicate. Thus, I can reduce the problem of deciding if a given valid predicate p_1 is optimal to the problem of deciding if following formula is satisfiable:

$$\exists col_1 \in \mathsf{Cols}' \; s.t. \; \mathsf{p}_1 \wedge (\forall col_2 \notin \mathsf{Cols}' \; s.t. \; \neg\mathsf{p})$$

This formula contains an alternating quantifier that supports linear arithmetic over integer, real number, and bit vectors. So it is a decidable problem [34, 39]. Thus, the problem of deciding if a given valid synthesized predicate is optimal is also decidable.

6.3 Synthesizing Predicates

In this section, I first present the overall algorithm that SIA uses to synthesize a valid, optimal predicate in §6.3.1. I then cover the key sub-procedures in the following sub-sections. In §6.3.2, I discuss how SIA encodes a predicate as an SMT formula. In §6.3.3, I describe how SIA generates the initial learning samples. In §6.3.4, I explain why SIA uses a linear SVM and discuss how it uses this machine learning model to learn a predicate. Finally, in §6.3.5, I present how SIA verifies if the learned predicate is valid, and generates counter-examples accordingly.

Algorithm 10: Procedure for synthesizing a weaker predicate

```
Input  : A predicate p, and a set of columns Cols', where Cols' is a subset of p's dependency
         columns Cols
Output : A valid synthesized predicate p_1
1  Procedure Synthesize(p, Cols')
2      Procedure SynthesizeAux(p, Cols', p_1, Ts, Fs, i)
3          if i > max then return p_1;
4          p_2 ← Learn(Ts, Fs)
5          isValid ← Verify(p_2, p)
6          if isValid then
7              p_3 ← p_1 ∧ p_2
8              Fs_1 ← CounterF(p_3, p, Fs)
9              if Fs_1 = ∅ then return p_3;
10             else return SynthesizeAux(p, Cols', p_3, Ts, Fs ∪ Fs_1, i + 1);
11         else
12             Ts_1 ← CounterT(p_1, p, Ts)
13             return SynthesizeAux(p, Cols', p_1, Ts ∪ Ts_1, Fs, i + 1)
14         end
15     end
16     (Ts, Fs) ← GenerateSamples(p, Cols)
17     return SynthesizeAux(p, Cols, True, Ts, Fs, 0)
18 end
```

6.3.1 Predicate Synthesis

Alg. 10 presents the procedure for synthesizing valid predicates. The Synthesize procedure takes two inputs: (1) an original predicate p, and (2) a set of columns Cols′, which is a subset of p's dependency columns Cols. It returns a valid synthesized predicate p_1. The Synthesize recursively uses the SynthesizeAux sub-procedure. SynthesizeAux takes six inputs: (1) the original predicate p, (2) the set of columns Cols, (3) a valid synthesized predicate p_1, (4) true training samples Ts, (5) false training samples Fs, and (6) the current iteration number i. It returns a valid synthesized predicate that *at least as strong* as the given valid synthesized predicate p_1.

Within the SynthesizeAux procedure, SIA first compares the current iteration number i against the maximum number of iterations max that is pre-defined. If i is greater than max, then it simply returns p_1. If not, SynthesizeAux uses the Learn procedure (§6.3.4) to learn a new predicate p_2 based on the given training samples. The Learn procedure returns a predicate that is guaranteed to classify all Ts samples as TRUE. The SynthesizeAux procedure then uses the Verify procedure (§6.3.5) to verify if p_2 is valid.

If p_2 is valid, then the SynthesizeAux procedure computes the conjunction of new learned predicate p_2 with the input valid synthesized predicate p_1 to obtain a new predicate p_3. The

SynthesizeAux procedure then uses the CounterF procedure (§6.3.4) to generate new FALSE training samples. These samples are unsatisfaction tuples for original predicate p and Cols, but are classified as TRUE by predicate p_3. These FALSE samples must be different from previous FALSE samples. If CounterF cannot generate new FALSE samples, then SIA returns p_3 (because it is optimal). Otherwise, SynthesizeAux recursively calls itself with the same inputs, except for the new valid synthesized predicate p_3, a larger set of FALSE samples, and an updated iteration number.

If p_2 is not valid, then the SynthesizeAux procedure uses the CounterT procedure (§6.3.4) to generate additional TRUE samples. These TRUE samples are classified as False by p_2, and must be different from previous TRUE samples. The SynthesizeAux procedure recursively calls itself with the same inputs, except for a larger set of TRUE samples, and an updated iteration number.

Synthesize uses the GenerateSamples procedure (§6.3.3) to obtain the initial training samples: Ts and Fs. It invokes the SynthesizeAux procedure with these inputs: predicate p and Cols, initial valid synthesized predicate TRUE, initial training samples, and initial iteration count 0. TRUE is a trivial valid synthesized predicate because conjunction of p with TRUE implies p.

6.3.2 Predicate Encoding

Since SIA leverages the solver in several procedures (e.g., CounterT, CounterF, Verify, and GenerateSamples), I first discuss how it converts a predicate expressed in SQL to a logical formula supported by the SMT solver. Since the solver supports all the arithmetic operators, arithmetic comparators, and the logical operators presented in §6.2.1, this is a straightforward procedure except for these three problems.

TYPE CONVERSION: The solver only supports four primitive data types: integer, real, boolean, and bit vector. SIA converts all the supported data types (e.g., DATE) to these primitive data types while preserving all arithmetic relations. For example, SIA converts the DATE type to integer by choosing an origin date, and representing a given date based on the number of days from the origin date (integer value) as I showed in §6.1.3.

THREE-VALUED LOGIC: SIA supports three-valued logic in SQL. A tuple may take a NULL value for a given column. A predicate may evaluate to three possible values: True, False, or NULL. To support the three-valued logic, SIA uses the encoding scheme in previous two chapter. It represents

a column with a pair of symbolic variables. The first variable represents the value of the column. The second boolean variable indicates if the value is `NULL`. SIA only uses this encoding scheme in the Verify procedure. This scheme ensures that Verify correctly validates the newly learned predicate using three-valued logic. It is crucial to preserve the semantics of the original query.

In other procedures associated with generating training samples, it uses an alternate encoding scheme with only the first variable. This is because these procedures generate non-`NULL` values to synthesize a predicate with arithmetic comparator. To handle the special `NULL` value, SIA separately infers an additional predicate with the `IS_NULL` function based on the original predicate.

NON-LINEAR ARITHMETIC: The satisfaction problem of a SMT formula with integer, non-linear arithmetic is undecidable [60]. So, SIA cannot directly convert a predicate with multiplication or division of two integer-valued columns. This is because the resulting formula is a non-linear arithmetic formula, that renders the Verify procedure to be undecidable. To partially circumvent this problem, SIA treats multiplication and division of columns as a single column while converting the predicate to a formula (if these columns are not used in other parts of the predicate).

In the following sub-sections, I refer to the SMT formula that is obtained from the SQL predicate as a predicate.

6.3.3 Generation of Initial Samples

The Synthesize procedure uses the GenerateSamples procedure to generate the initial training samples. This procedure takes the original predicate p and a set of columns Cols' (subset of dependency columns of p) as inputs. It returns two sets of training samples: Ts and Fs. Each training sample is a list of values for each column in Cols'. Based on the properties I proved in Lemmas 7 and 8, the training samples in Ts and Fs are satisfaction and unsatisfaction tuples, respectively. GenerateSamples leverages the SMT solver to generate these samples.

GENERATING TRUE SAMPLES: Given the original predicate p, and a set of columns Cols', GenerateSamples iteratively feeds the following formula into the solver to generate the `TRUE` samples:

$$\mathsf{p} \wedge \mathsf{NotOld}$$

Here, p is a formula that represents the original predicate. NotOld is another formula that SIA uses to force the solver to generate a new model for Cols′. NotOld is a conjunction formula where each term is a constraint that sets the variables representing columns in Cols′ not to be equal to any of the values in already existing TRUE samples. In each iteration, GenerateSamples updates this NotOld formula by adding an additional term that constrains the columns in Cols′ to not be equal to the sample generated in the last iteration.

If the solver decides that the given formula is satisfiable, then GenerateSamples generates a new sample by extracting the values in the satisfaction model for all columns in Cols′. The satisfaction model gives concrete values for columns *not* in Cols′ along with concrete values for columns in Cols′ that satisfy p. Given the definition of unsatisfaction tuple in Definition 8, this sample is clearly not an unsatisfaction tuple. So, it is a TRUE sample.

If the solver decides the given formula is unsatisfiable, then there is no new satisfaction tuple for predicate p and the set of columns Cols′. In this case, there are a finite number of tuples over columns in Cols′ that satisfy the predicate, and all these tuples have been found. SIA constructs the strongest valid synthesized predicate by taking the disjunction of a set of constraints wherein each constraint sets the columns in Cols to be equal to TRUE samples.

GENERATING FALSE SAMPLES: GenerateSamples iteratively feeds the following formula into the solver to generate the FALSE samples:

$$\exists \mathtt{Col}_1 \in \mathsf{Cols}' \; s.t. \mathsf{NotOld} \wedge \; (\forall \mathtt{Col}_2 \notin \mathsf{Cols}' \; s.t. \; \neg \mathsf{p})$$

Here, ¬p is the negation of the formula that represents the original predicate. NotOld is the SMT formula that SIA uses to force the solver to generate a new model for Cols. SIA updates NotOld in each iteration in the same manner as when it generates TRUE samples.

If the solver decides that the given formula is satisfiable, then GenerateSamples generates a new FALSE sample by extracting the values in the satisfaction model. If the solver decides that the formula is unsatisfiable, then there is no additional unsatisfaction tuple for predicate p over Cols′. In this case, there are finite number of tuples over Cols′ that do not satisfy the valid synthesized predicate, and all these tuples have been found. SIA constructs the strongest valid synthesized

Algorithm 11: Procedure for learning a valid predicate

```
Input  : Two sets of training samples
Output : A learned predicate that correctly classifies all Ts samples
Procedure Learn(Ts, Fs)
    Models ← {}
    while Ts ≠ ∅ do
        model ← linearSVM(Ts, Fs)
        Models ← Models ∪ model
        Ts ← misclassified(Ts, model)
    end
    return ∨Models
end
```

predicate by taking the negation of disjunction of a set of constraints wherein each constraint sets the columns in Cols to be equal to `FALSE` samples.

ADDITIONAL HEURISTICS: I use additional heuristics for forcing the solver to generate useful training samples depending on the machine learning model. For example, I may constrain that the values must not be equal to zero. I may constrain the values to fall within a certain range. I may specify additional linear arithmetic relations between columns. These empirical heuristics depend on the predicates, the data distribution, and the choice of the machine learning model. SIA only employs the first heuristic (since it does not rely on data). The latter heuristics may be used to improve the quality of training samples if the optimizer has access to the statistical distribution of the data in the target columns.

6.3.4 Predicate Learning

Given two sets of training samples, the Learn procedure returns a predicate that correctly classifies all the `TRUE` samples. Because SIA needs to verify the learned predicate is valid, there are two criteria that the underlying machine learning model must satisfy. First, the trained model must be interpretable. This allows SIA to convert the model to an SMT formula for verification. Second, the satisfaction problem for the generated SMT formula must be decidable. This is because the verification procedure must be decidable. Given these two criteria, Learn uses a standard linear SVM [76, 65, 24] as the underlying machine learning model. Since the trained SVM model is a linear function over the input columns, it may be converted to an SMT formula with numerical linear arithmetic. Furthermore, the satisfaction problem for numerical linear arithmetic is decidable [39].

Learn must return a predicate that should correctly classify all TRUE samples. If the two sets of input samples are *not* linearly separable, then the linear SVM may return a model that classifies certain TRUE samples as FALSE. To address this problem, Learn iteratively trains multiple linear SVM models. As shown in Alg. 11, it first trains a linear SVM model over all training samples. If this model classifies certain TRUE samples as FALSE, then it trains another model with the mis-classified TRUE samples along with the FALSE samples. It keeps training models in this manner until all the TRUE samples are correctly classified. Lastly, Learn returns the disjunction of all models as the learned predicate.

6.3.5 Validation & Counter-Example Generation

LEARNED PREDICATE VALIDATION: SynthesizeAux procedure uses the Verify procedure to verify if the learned predicate is valid. The latter procedure uses the solver for validation. Given the original predicate p, and the learned predicate p_1, the Verify procedure feeds the following formula into the solver:

$$\mathsf{p} \wedge \neg \mathsf{p}_1$$

Both formulae use the encoding scheme that supports three-valued logic (§6.3.2). If the solver decides that this formula is unsatisfiable, then there is no tuple that satisfies p but not p_1. In other words, for any given tuple, if p accepts this tuple, then p_1 also accepts this tuple. Based on Definition 6, p_1 is thus a valid synthesized predicate. In this case, SynthesizeAux uses CounterF to generate additional FALSE samples to strengthen the predicate.

If the solver decides that this formula is satisfiable, then there is at least one tuple that satisfies p but does not satisfy p_1. In this case, p_1 is invalid. SynthesizeAux uses CounterT to generate additional TRUE samples to be used in next iteration of the learning process.

GENERATION OF TRUE COUNTER-EXAMPLES: SynthesizeAux uses the CounterT procedure to generate TRUE counter-examples. Given the original predicate p and an invalid learned predicate p_1, this procedure generates additional TRUE samples such that each sample satisfies p but does not satisfy p_1. CounterT leverages the solver to generate these samples. It feeds the following formula to the solver:

$$\mathsf{p} \wedge \neg \mathsf{p}_1 \wedge \mathsf{NotOld}$$

Here, p represents the original predicate and $\neg\mathsf{p}_1$ represents the negation of the learned predicate. NotOld constrains the model to not pick prior TRUE samples. This SMT formula is satisfiable. Since p_1 is invalid, it is guaranteed that there exists a TRUE sample that is incorrectly classified by p_1 as FALSE. CounterT extracts the values of columns in the model returned by the solver to construct a counter-example. This new TRUE samples is distinct from prior TRUE samples, and does not satisfy p_1. CounterT repeatedly feeds the formula to the solver to get multiple samples.

GENERATION OF FALSE COUNTER-EXAMPLES: SynthesizeAux procedure uses the CounterF procedure to generate FALSE counter-examples. Given the original predicate p and a valid learned predicate p_1, this procedure generates additional FALSE samples such that each sample does not satisfy p but does satisfy p_1. CounterF procedure feeds the following formula to the SMT solver:

$$\exists \mathtt{COL}_1 \in \mathsf{Cols}' \; s.t. \mathsf{p}_1 \wedge \mathsf{NotOld} \wedge (\forall \mathtt{COL}_2 \notin \mathsf{Cols}' \; s.t. \neg \mathsf{p})$$

Here, p_1 represents the valid synthesized predicate. NotOld constrains the model to not pick prior FALSE samples. The last part of the formula ensures that it is an unsatisfaction tuple. If the solver decides that this formula is satisfiable, then CounterT extracts the values from the model to generate a new FALSE sample. This new FALSE samples is distinct from prior FALSE samples, and does satisfy p_1. If the solver decides that this formula is unsatisfiable, then CounterT cannot generate additional FALSE samples. In this case, based on Lemma 8, p_1 is optimal.

6.4 Evaluation

I now describe our implementation and evaluation of SIA. I begin with a description of our implementation in §6.4.1. I next discuss how I construct a collection of queries derived from the TPC-H benchmark [79] to evaluate SIA in §6.4.2. I then report the results of our comparative analysis of SIA in §6.4.3 and §6.4.4. I next cover the impact of SIA on runtime performance in §6.4.5. I conclude with a discussion on the broader impacts of learned predicates in §6.4.6. I discuss the limitations of SIA in §6.4.7.

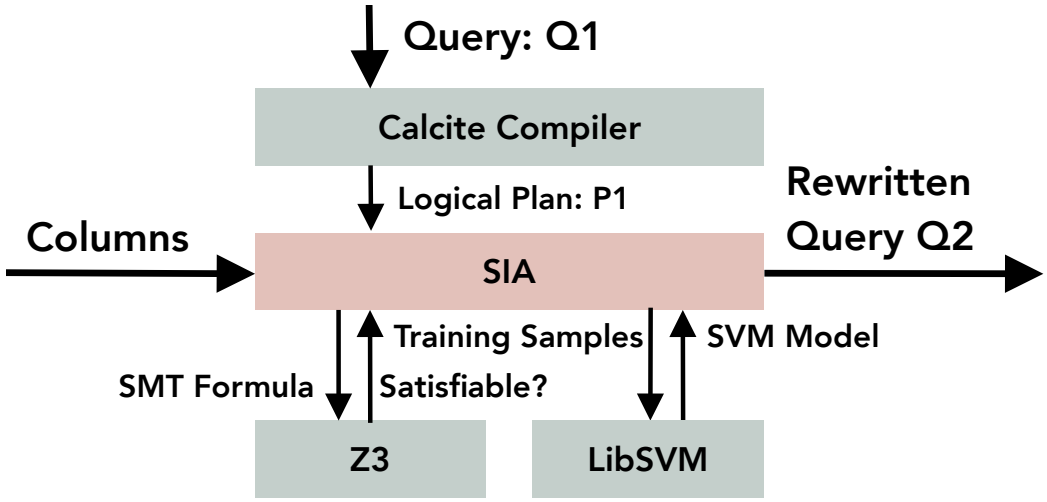


Figure 14: Architecture of SIA– SIA leverages three components: (1) CALCITE query optimization framework, (2) Z3 SMT solver, and (3) SVM library.

6.4.1 Implementation

The architecture of SIA is illustrated in Figure 14. SIA takes a predicate p and a subset of columns Cols' from the Cols used in p as inputs. It returns a valid synthesized predicate p' that only uses the columns in Cols'. To facilitate integration with DBMSs, SIA directly operates on SQL queries.

SIA leverages three components: ❶ A query compiler converts the given SQL query to a relational algebraic representation. SIA uses the open-source CALCITE query optimization framework for this purpose [3]. It then converts the predicate into an SMT formula, and implements the counter-example guided learning technique. SIA uses the second component to generate training samples and to validate the learned predicate. It uses the third component to train a linear SVM model that is used for learning the predicate. SIA is implemented in Java (2,925 lines of code). ❷ The second component is the Z3 SMT solver that SIA leverages for determining the satisfiability of an SMT formula and for generating models if the given formulae is satisfiable [14]. ❸ The third component is an SVM library [12].

6.4.2 Benchmark

To evaluate the efficacy of SIA in generating valid predicates, I construct a collection of queries based on the TPC-H benchmark [79]. The reasons for constructing this benchmark are twofold. First, I seek to make the queries publicly available. Second, I generate queries to simulate the characteristics of predicates in production query workloads. In particular, I use a sub-query of TPC-H Q4 with more

complex predicates. Third, I gain more control over the complexity of the predicate (e.g., number of dependent columns). All of these queries follow this template:

```
Q:  SELECT * FROM lineitem, orders
       WHERE o_orderkey = l_orderkey
       AND predicate
---
predicate = Term-1 AND Term-2 AND ..... Term-K
Term = Expr Compare Expr
Expr = Column | Arithmetic Expr | Date | Interval
```

Here, $predicate$ is a randomly-generated predicate in conjunctive normal form consists of a set of terms. Each $Term$ is a binary, arithmetic predicate, wherein each expression in the binary predicate may be: (1) a column, (2) a binary arithmetic expression, (3) a date constant, or (4) an interval constant (i.e., number of days). I constrain $predicate$ to use three columns from $lineitem$ table ($l\_shipdate$, $l\_commitdate$, and $l\_receiptdate$), and one column from $orders$ table ($o\_orderdate$). I ensure that each generated binary predicate in the overall $predicate$ refers to the column in $orders$. Thus, no binary predicate may only depend on columns from $lineitem$. This constraint ensures that the optimizer cannot push down the original predicate below the join operator to the $lineitem$ table. I configure each $predicate$ to contain from three through eight terms. I re-generate the query if the $predicate$ cannot be satisfied by any tuples. In this manner, I construct a collection of 200 queries. I have provided the query benchmark along with this submission.

BASELINES: I compare four techniques: (1) syntax-driven rules, (2) SIA_$v1$ (only one iteration; 110 `TRUE` and `FALSE` samples, respectively), (3) SIA_$v2$ (only one iteration; 2$\times$ more samples compared to SIA_$v1$), (4) SIA (at most 41 iterations; 10 initial `TRUE` and `FALSE` samples, respectively; at most the same number of samples as SIA_$v1$). All the variants of SIA are listed in Table 9.

To the best of our knowledge, SIA is the first system to synthesize valid, reduced predicates by leveraging machine learning and verification algorithms. Previous state-of-the-art approaches are based on syntax-driven rules (e.g., transitive closure). I implement a syntax-driven transitive closure transformation for our comparative analysis.

Table 9: Baselines – I compare SIA against two non-iterative baselines.

	Max Iteration #	# Initial True Samples	# Initial False Samples	# Samples per Iteration
SIA _v1	1	110	110	N/A
SIA _v2	1	220	220	N/A
SIA	41	10	10	5

Table 10: Efficacy of SIA– Comparative analysis of SIA against the baselines with respect to their ability to synthesize valid (possibly optimal) predicates.

		SIA		Transitive Closure
# of Used Columns	**# of Possible Predicates**	**# of Valid**	**# of Optimal**	**# of Valid**
one	233	182	158	18
two	160	102	20	4
three	30	20	0	0

		SIA _v1		SIA _v2	
# of Used Columns	**# of Possible Predicates**	**# of Valid**	**# of Optimal**	**# of Valid**	**# of Optimal**
one	233	158	75	166	98
two	160	11	3	17	4
three	30	2	0	1	0

6.4.3 Efficacy of SIA

In this experiment, I examine whether SIA is able to effectively synthesize predicates over the given set of columns. I run SIA on each query with all possible subsets of three columns $l\_shipdate$, $l\_commitdate$, and $l\_receiptdate$ from $lineitem$ table. In SIA, I set the number of initial `TRUE` and `FALSE` samples to 10, respectively. In each iteration of the learning loop, I configure the number of newly added training samples to 5 (either `TRUE` or `FALSE` depending on the requirements of the learning process). I set the maximum number of allowed iterations to 41. After 41 iterations, SIA either returns the current synthesized predicate, or returns `NULL` if SIA cannot synthesize any valid predicate other than the trivial predicate (`TRUE`).

To evaluate the efficacy of the iterative learning process guided by counter-examples used in SIA, I use two non-iterative baselines (i.e., number of iterations = 1). These baselines (SIA_$v1$ and SIA_$v2$) seek to directly learn a predicate from initial training samples. In SIA_$v1$, I set the number of initial `TRUE` and `FALSE` samples to 110, respectively. This is equivalent to the total number of samples generated by SIA after it hits the final iteration. In SIA_$v2$, I set the number of initial `TRUE` and `FALSE` samples to 220, respectively ($2\times$ the number of samples given to SIA_$v1$). I conduct this experiment on a commodity server (Intel Core i7-860 processor with 16 GB RAM).

Table 10 shows the results of this experiment. For each query, I configure SIA to generate synthesized predicates with varying complexity (ranging from one through three columns from $lineitem$ table). I classify the synthesized predicates into three categories based on the number of columns they use. SIA seeks to construct a predicate that uses all columns (i.e., coefficients must be non-zero). I refer to the number of valid predicates referring to the given set of columns as the *number of possible predicates*. For example, if a query has two valid predicates, one using $l\_shipdate$ and another one using $l\_commitdate$, then I classify it as two possible predicates in the first category.

The most notable observation in Table 10 is that SIA effectively synthesizes valid predicates over the given columns. For predicates that must only use one column, SIA successfully generates 182 out of 233 predicates, while SIA_$v1$ only generates 158 predicates and SIA_$v2$ only generates 166 predicates. I note that even though SIA runs for 41 iterations, it may only generate 220 total training samples (comparable to the samples used by SIA$_v1$ and half of that used by SIA_$v2$). I found that the transitive closure transformation is not effective at this task.

The benefits of counter-example guided learning in SIA is more prominent for more complex predicates that use two and three columns. Specifically, for predicates with two columns, SIA successfully generates 102 out of 160 predicates, while SIA_$v1$ and SIA_$v2$ only generate 4 and 17 predicates, respectively. For predicates with three columns, SIA generates 20 out of 30 predicates, while SIA_$v1$ generates two and SIA_$v2$ only generates one predicate, respectively. Besides synthesizing more predicates across all categories, SIA also generates significantly more number of optimal predicates in the first two categories compared to SIA_$v1$ and SIA_$v2$. This is because the initial training samples used by SIA_$v1$ and SIA_$v2$ are completely random and may cluster together. In contrast, SIA's iterative counter-example guided learning forces the generated samples to be of higher quality, thereby allowing it to learn more, stronger valid predicates.

6.4.4 Efficiency of SIA

In this experiment, I study the efficiency of SIA. I first measure the time taken by SIA and its baselines to synthesize the predicates. I classify the total time taken into three categories: (1) generation time, (2) validation time, and (3) learning time. Generation time refers to the time taken

Table 11: Efficiency of SIA– Comparative analysis of SIA against the baselines with respect to their time taken to synthesize predicates.

	EQUITAS		
# of Used Columns	**Generation Time (ms)**	**Learning Time (ms)**	**Validation Time (ms)**
one	893.2	1.8	98.5
two	2933	14.6	281.4
three	4154	38.9	328.2

	EQUITAS _v1			EQUITAS _v2		
# of Used Columns	**Generation Time (ms)**	**Learning Time (ms)**	**Validation Time (ms)**	**Generation Time (ms)**	**Learning Time (ms)**	**Validation Time (ms)**
one	2625	0.5	1	9304	1.9	11.3
two	2739	1.0	7.3	10159	3.2	11.64
three	3801	1.0	8.5	11859	5.0	12.0

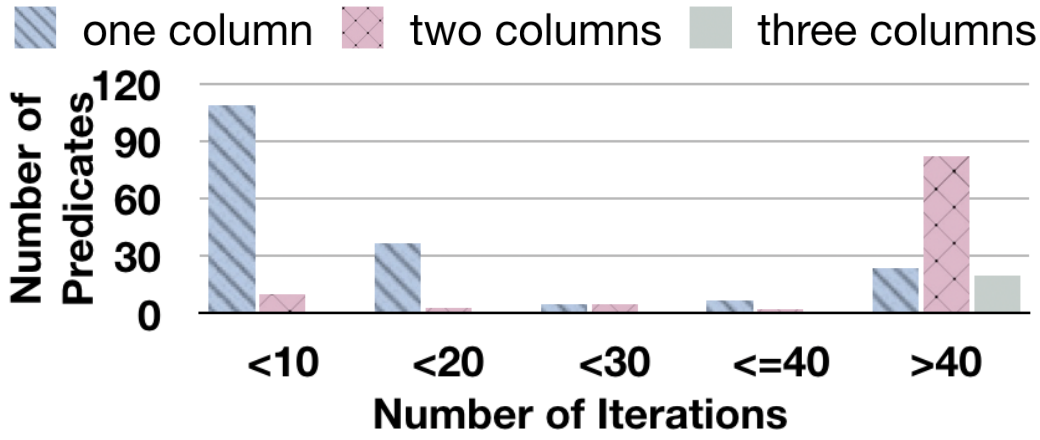
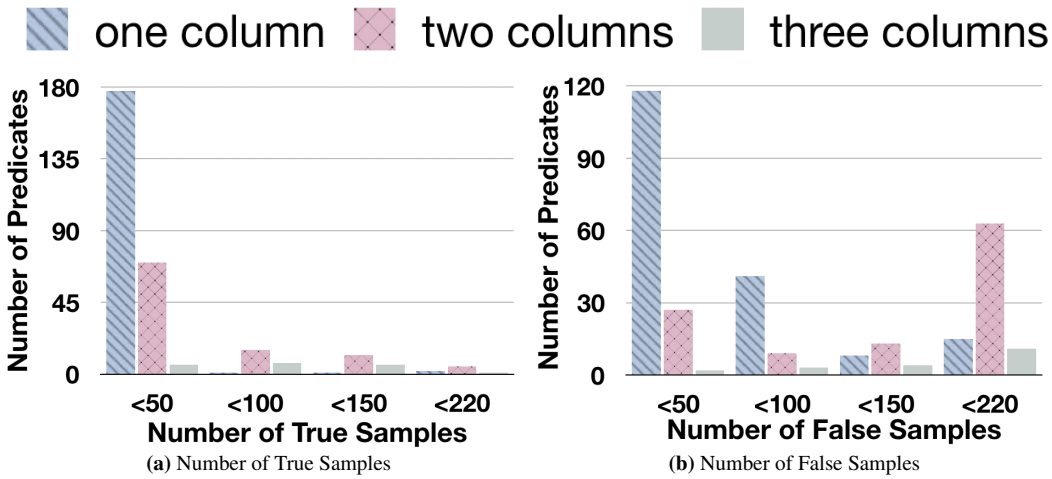


Figure 15: Efficiency of Learning Loop – Average number of iterations that SIA takes to converge to an optimal predicate.

to obtain the initial training samples and the counter-example samples from the solver. Learning time refers to the time taken to train the SVM model using the generated samples. Validation time refers to the time taken to check if the synthesized predicate is valid or if a valid synthesized predicate is optimal using the solver. Table 11 shows the results of this analysis. SIA executes nearly as fast as SIA_$v1$. SIA_$v2$ is slower than these two other techniques since the data generation time dominates the overall synthesis pipeline. Thus, to accelerate the synthesis process, I must reduce the number of generated training samples.

LEARNING LOOP: I next examine the efficiency of the learning loop. I measure the number of iterations SIA takes to synthesize the optimal predicate. Figure 15 shows that SIA synthesizes 109 optimal predicates (out of 182 generated predicates) in the first category within 10 iterations. For more complex predicates that use two or three columns, SIA often fails to find the optimal predicate



(a) Number of True Samples

(b) Number of False Samples

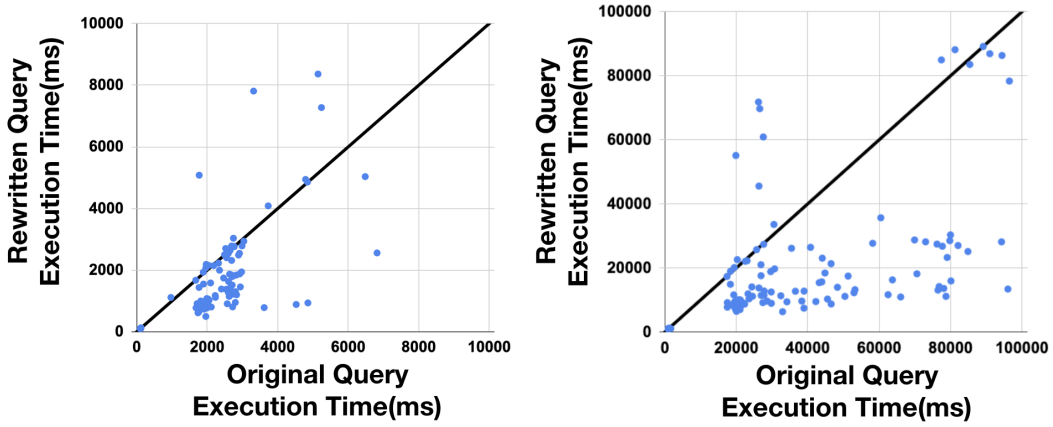
Figure 16: Sample Distribution – Distribution of the number of training samples generated by SIA before the final iteration.

within the maximum number of iterations. Even if it does find the optimal predicate, it requires more iterations compared to that needed for predicates in the first category. I discuss this limitation in §6.4.7.

I next measure the number of `TRUE` and `FALSE` samples that SIA generates. This is important because the data generation time dominates the overall time taken to synthesize predicates. Figure 16a shows the distribution of the number of `TRUE` samples in the final iteration of the learning loop. Most of the successfully generated one-column predicates (178 out of 182) require less than 50 `TRUE` samples. More complicated predicates require more `TRUE` samples to learn a valid predicate. Figure 16b shows the distribution of number of `FALSE` samples in the final iteration of the learning loop. Most of the optimal one-column predicates (118 out of 158) require less than 100 `FALSE` samples. More complicated predicates do not converge even with more `FALSE` samples. I discuss this limitation in §6.4.7.

6.4.5 Impact on Runtime Performance

I next conduct an experiment to study the impact of SIA on runtime performance. In particular, I examine if the predicates synthesized by SIA enable the optimizer to apply predicate-centric optimization rules to speed up query execution. Across 200 queries, SIA successfully generates valid predicates for 114 queries that only depend on columns from $lineitem$ table. I measure the runtime performance of these 114 queries (without and with the synthesized predicates). It is important to



(a) Scale Factor = 1 **(b)** Scale Factor = 10

Figure 17: Impact on Runtime Performance – Comparison of the time taken to execute the original and rewritten queries.

Table 12: Selectivity – Average selectivity of synthesized predicates with respect to $lineitem$ table. I classify them based on their performance impact.

Scale Factor	# of Faster	Avg. Selectivity	# of 2× Faster	Avg. Selectivity	# of Slower	Avg. Selectivity
one	85	0.76	36	0.69	29	0.97
ten	95	0.78	66	0.74	19	0.96

note that the rewritten queries are semantically equivalent to their original counterparts. I execute these queries on the TPC-H database on PostgreSQL (v12). I consider two scale-factors: one and ten.

The results are shown in Figure 17. The x-axis and y-axis in these plots represent the time taken to execute the original and rewritten queries, respectively. I highlight the break-even point for each query using a black slanted line. With a scale-factor of one, 85 out of 114 rewritten queries are below the the break-even line (i.e., faster than their original counterparts). This highlights the impact of SIA on runtime performance. Only 29 rewritten queries fall above the break-even line. Furthermore, 36 rewritten queries exhibit more than 2× speedup. Only 2 of them slow down by more than 2×. The benefits of SIA on more significant when the scale factor is set to ten (Figure 17b). Here, 95 rewritten queries fall below the break-even line, and 66 of them are at least 2× faster. Only 19 rewritten queries fall above the break-even line, and 4 of them are more than 2× slower.

SELECTIVITY OF PREDICATES: To examine the efficacy of the synthesized predicates in accelerating queries, I measure the selectivity of the predicates with respect to the $lineitem$ table. I classify these 114 synthesized predicates into four categories based on their impact on runtime performance on the TPC-H database. As summarized in Table 12, the selectively of the synthesized predicate determines its impact. When the scale factor is set to one, the average selectivity of synthesized

predicates in faster and slower rewritten queries is 0.76 and 0.97, respectively. Thus, the optimizer should add a synthesized predicate only when its selectivity is low.

6.4.6 Discussion

I focus on pushing predicates below the join operator in our evaluation (due to the template of the original queries). However, I note that synthesizing valid predicates over a given set of columns enables more predicate-centric optimization rules. These include moving the predicate below the aggregation operator [58] and constructing predicates that only use already-indexed columns [44].

However, the cost of evaluating synthesized predicates is not negligible in certain settings. As shown in §6.4.5, the benefits of adding a synthesized predicate is prominent only if its selectivity is low. Thus, the problems of choosing the set of columns over which I seek to synthesize a predicate, and deciding if the synthesized predicate is beneficial are non-trivial problems.

6.4.7 Limitations

The key limitation of SIA manifests when the generated `TRUE` and `FALSE` samples are *not* linearly separable. In this case, it fails to synthesize optimal or even valid predicates. Consider the following predicate: $a > b \;\&\&\; a < b + 50 \;\&\&\; b > 0 \;\&\&\; b < 150$. In this case, the `FALSE` samples are on both sides of `TRUE` samples. So, SIA either returns a disjunction of predicates that is not optimal, or returns an invalid predicate because the underlying linear SVM model only seeks to minimize the penalty term. I could tackle this limitation using another interpretable machine learning algorithm that copes with a set of samples that are not linearly separable by learning a boolean combination of linear predicates.

CHAPTER VII

CONCLUSION

In this dissertation, I presented three symbolic based approaches for three problems. For the problem of proving query equivalence under set semantics, I reduce the problem to the problem of proving the containment relationship between queries. Then I leverage the SMT solver to verify the relational properties of symbolic representations of two queries to prove the containment relationship. For the problem of proving query equivalence under bag semantics, I reduce the problem to the problem of proving the existence of an identical, bijective map between tuples that are returned by two queries for all valid inputs. I also leverage the SMT solver to verify the conditions of the query pair symbolic representation of two queries to prove the existence of an identical, bijective map. For the problem of optimizing queries with learned predicates, I leverage the SMT solver to generate training samples, and verify the correctness of learned predicate. By investigating these three problems with symbolic based approaches, I proved that using symbolic based approaches can significantly improve the effectiveness and efficiency of verifying query equivalence and optimizing queries with predicates.

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] "LibSVM." https://github.com/cjlin1/libsvm.

[13] "PostgreSQL." https://www.postgresql.org/.

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

[15] ABDUL KHALEK, S., ELKARABLIEH, B., O. LALEYE, Y., and KHURSHID, S., "Query-aware test generation using a relational constraint solver," in *ASE*, 09 2008.

[16] ABITEBOUL, S., HULL, R., and VIANU, V., *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[17] AFRATI, F., DAMIGOS, M., and GERGATSOULIS, M., "Query containment under bag and bag-set semantics," *Inf. Process. Lett.*, vol. 110, pp. 360–369, 04 2010.

[18] AHO, A. V., SAGIV, Y., and ULLMAN, J., "Equivalence among relational expressions," *SIAM Journal of Computing*, vol. 8, no. 9, pp. 218–246, 1979.

[19] ALBERT, J., "Algebraic properties of bag data types," in *VLDB*, 1991.

[20] B.A.TRAKHTENBROT, "Impossibility of an algorithm for the decision problem in finite classes," in *Journal of Symbolic Logic*, 1950.

[21] CALVANESE, D., GIACOMO, G. D., and LENZERINI, M., "Conjunctive query containment and answering under description logic constraints," in *TOCL*, 2008.

[22] CHANDRA, A. K. and MERLIN, P. M., "Optimal implementation of conjunctive queries in relational data bases," in *STOC*, 1977.

[23] CHANDRA, A. K. and MERLIN, P. M., "Optimal implementation of conjunctive queries in relational data bases," in *STOC*, pp. 77–90, 1977.

[24] CHANG, C.-C. and LIN, C.-J., "Libsvm: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, 2011.

[25] CHAUDHURI, S. and VARDI, M. Y., "Optimization of real conjunctive queries," in *PODS*, 1993.

[26] CHEKURI, C. and RAJARAMAN, A., "Conjunctive query containment revisited," in *ICDT*, 1997.

[27] CHU, S., LI, D., WANG, C., CHEUNG, A., and SUCIU, D., "Demonstration of the Cosette automated sql prover," in *SIGMOD*, 2017.

[28] CHU, S., MURPHY, B., ROESCH, J., CHEUNG, A., and SUCIU, D., "Axiomatic foundations and algorithms for deciding semantic equivalences of sql queries," *PVLDB*, vol. 11, no. 11, pp. 1482–1495, 2018.

[29] CHU, S., WEITZ, K., CHEUNG, A., and SUCIU, D., "HoTTSQL: proving query rewrites with univalent sql semantics," in *PLDI*, 2017.

[30] COHEN, S., NUTT, W., and SAGIV, Y., "Deciding equivalences among conjunctive aggregate queries," 1998.

[31] COHEN, S., NUTT, W., and SAGIV, Y., "Containment of aggregate queries," in *Lecture Notes in Computer Science*, 2002.

[32] COHEN, S., NUTT, W., and SEREBRENIK, A., "Rewriting aggregate queries using views," in *PODS*, 1999.

[33] CONSENS, M. P., MENDELZON, A. O., VISTA, D., and WOOD, P. T., "Constant propagation versus join reordering in datalog," in *RIDS*, 1995.

[34] COOPER, D. W., "Theorem proving in arithmetic without multiplication," in *Machine Intelligence*, 1972.

[35] DAVIS, M., LOGEMANN, G., and LOVELAND, D. W., "A machine program for theorem-proving," *Commun. ACM*, 1962.

[36] DE GIACOMO, G., CALVANESE, D., and LENZERINI, M., "On the decidability of query containment under constraints," in *PODS*, 12 1999.

[37] DE MOURA, L. M. and BJØRNER, N., "Z3: an efficient SMT solver," in *TACAS*, 2008.

[38] DEUTSCH, A., POPA, L., and TANNEN, V., "Physical data independence, constraints, and optimization with universal plans," in *VLDB*, 03 2002.

[39] DILLIG, I., DILLIG, T., MCMILLAN, K. L., and AIKEN, A., "Minimum satisfying assignments for smt," in *CAV*, 2012.

[40] DUTERTRE, B., "Yices 2.2," in *CAV*, 2014.

[41] ELHEMALI, M., GALINDO-LEGARIA, C. A., GRABS, T., and JOSHI, M. M., "Execution strategies for sql subqueries," in *SIGMOD*, 2007.

[42] GOLDSTEIN, J. and LARSON, P.-A., "Optimizing queries using materialized views: A practical, scalable solution.," vol. 30, pp. 331–342, 06 2001.

[43] GROSSMAN, S., COHEN, S., ITZHAKY, S., RINETZKY, N., and SAGIV, M., "Verifying equivalence of spark programs," in *CAV*, 2017.

[44] HELLERSTEIN, J. M. and STONEBRAKER, M., "Predicate migration: Optimizing queries with expensive predicates," in *SIGMOD*, 1993.

[45] HORROCKS, I., SATTLER, U., TESSARIS, S., and TOBIES, S., "How to decide query containment under constraints using a description logic.," in *LPAR*, 2000.

[46] ILYAS, I. F., MARKL, V., HAAS, P., BROWN, P., and ABOULNAGA, A., "Cords: automatic discovery of correlations and soft functional dependencies," in *SIGMOD*, 2004.

[47] IOANNIDIS, Y. and RAMAKRISHNAN, R., "Efficient transitive closure algorithms.," in *VLDB*, 1988.

[48] IOANNIDIS, Y. E. and RAMAKRISHNAN, R., "Containment of conjunctive queries: Beyond relations as sets," in *TODS*, 1995.

[49] ITZHAKY, S., KOTEK, T., RINETZKY, N., SAGIV4, M., TAMIR5, O., VEITH, H., and ZULEGER, F., "On the automated verification of web applications with embedded sql," in *ICDT*, 2017.

[50] JAYRAM, T. S., KOLAITIS, P. G., and VEE, E., "The containment problem for real conjunctive queries with inequalities," in *PODS*, 2006.

[51] JINDAL, A., KARANASOS, K., RAO, S., and PATEL, H., "Selecting subexpressions to materialize at datacenter scale," *PVLDB*, vol. 11, no. 7, pp. 800–812, 2018.

[52] JOGLEKAR, M., GARCIA-MOLINA, H., PARAMESWARAN, A., and RE, C., "Exploiting correlations for expensive predicate evaluation," in *SIGMOD*, 2015.

[53] KANDULA, S., ORR, L., and CHAUDHURI, S., "Pushing data-induced predicates through joins in big-data clusters," 2019.

[54] KEMPER, A., MOERKOTTE, G., PEITHNER, K., and STEINBRUNN, M., "Optimizing disjunctive queries with expensive predicates," in *SIGMOD*, 1994.

[55] KHURSHID, S., PĂSĂREANU, C. S., and VISSER, W., "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for the Construction and Analysis of Systems* (GARAVEL, H. and HATCLIFF, J., eds.), (Berlin, Heidelberg), pp. 553–568, Springer Berlin Heidelberg, 2003.

[56] KIMURA, H., HUO, G., RASIN, A., MADDEN, S., and ZDONIK, S., "Correlation maps: A compressed access method for exploiting soft functional dependencies.," 2009.

[57] KOLAITIS, P. G. and VARDI, M. Y., "Conjunctive-query containment and constraint satisfaction," in *PODS*, 1998.

[58] LEVY, A. Y., MUMICK, I. S., and SAGIV, Y., "Query optimization by predicate move-around," in *VLDB*, 1994.

[59] LU, Y., CHOWDHERY, A., KANDULA, S., and CHAUDHURI, S., "Accelerating machine learning inference with probabilistic predicates," in *SIGMOD*, 2018.

[60] MATIYASEVICH, Y. V., "Hilbert's tenth problem and paradigms of computation," in *CiE*, 2005.

[61] NEGRI, M., PELAGATTI, G., and SBATTELLA, L., "Formal semantics of sql queries," in *ACM Trans. Database Syst.*, 1991.

[62] NELSON, G. and OPPEN, D. C., "Simplification by cooperating decision procedures," *ACM Trans. Program. Lang. Syst.*, 1979.

[63] NEUMANN, T., HELMER, S., MOERKOTTE, G., BELL, D., and HONG, J., "On the optimal ordering of maps, selections, and joins under factorization," in *ICDE*, 2006.

[64] PEZZINI, M., FEINBERG, D., RAYNER, N., and EDJLALI, R., "Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation." `https://www.gartner.com/doc/2657815/`, 2014.

[65] PLATT, J. C., "Advances in kernel methods," 1999.

[66] POPA, L., DEUTSCH, A., SAHUGUET, A., and TANNEN, V., "A chase too far?," in *SIGMOD*, 2002.

[67] POTTINGER, R. and HALEVY, A., "Minicon: A scalable algorithm for answering queries using views," *VLDB*, vol. 10, 12 2001.

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

[69] SAGIV, Y. and YANNAKAKIS, M., "Equivalences among relational expressions with the union and difference operators," in *J. ACM*, 1980.

[70] SCHLAIPFER, M., RAJAN, K., LAL, A., and SAMAK, M., "Optimizing big-data queries using program synthesis," in *SOSP*, 2017.

[71] SESHADRI, P., HELLERSTEIN, J. M., PIRAHESH, H., LEUNG, T. Y. C., RAMAKRISHNAN, R., SRIVASTAVA, D., STUCKEY, P. J., and SUDARSHAN, S., "Cost-based optimization for magic: Algebra and implementation," in *SIGMOD*, 1996.

[72] SHIVAKUMAR, N., GARCIA-MOLINA, H., and CHEKURI, C., "Filtering with approximate predicates," in *VLDB*, 1998.

[73] SHUMO, C., CHENGLONG, W., KONSTANTIN, W., and ALVIN, C., "Cosette: An automated SQL prover.," in *CIDR*, 2017.

[74] SIKKA, V., FÄRBER, F., LEHNER, W., CHA, S. K., PEH, T., and BORNHÖVD, C., "Efficient transaction processing in SAP HANA database: The end of a column store myth," in *SIGMOD*, pp. 731–742, 2012.

[75] SURAJIT, C., VIVEK, N., and SUNITA, S., "Efficient evaluation of queries with mining predicates," in *ICDE*, 2002.

[76] SUYKENS, J. A. K. and VANDEWALLE, J., "Least squares support vector machine classifiers," *Neural Process. Lett.*, 1999.

[77] TANNEN, V. and POPA, L., "An equational chase for path-conjunctive queries, constraints, and views," in *ICDT*, 1999.

[78] TARSKI, A., "A decision method for elementary algebra and geometry," in *Quantifier Elimination and Cylindrical Algebraic Decomposition*, 1951.

[79] THE TRANSACTION PROCESSING COUNCIL, "TPC-H Benchmark (Revision 2.16.0)." `http://www.tpc.org/tpch/`, June 2013.

[80] TRAN, N., LAMB, A., SHRINIVAS, L., BODAGALA, S., and DAVE, J., "The vertica query optimizer: The case for specialized query optimizers," in *ICDE*, 2014.

[81] ULLMAN, J., "Principle of database and knowledge-bas systems," 1989.

[82] VEANES, M., GRIGORENKO, P., DE HALLEUX, P., and TILLMANN, N., "Symbolic query exploration," in *FormaliSE*, 2009.

[83] VEANES, M., TILLMANN, N., and DE HALLEUX, J., "Qex: Symbolic sql query explorer," in *LPAR*, 2010.

[84] WALENZ, B., ROY, S., and YANG, J., "Optimizing iceberg queries with complex joins," in *SIGMOD*, 2017.

[85] WANG, Y., DILLIG, I., K. LAHIRI, S., and COOK, W., "Verifying equivalence of database-driven applications," in *PACMPL*, 2017.