Sia: Optimizing Queries using Learned Predicates

Qi Zhou
qzhou80@gatech.edu
Georgia Institute of Technology
Atlanta, USA

Joy Arulraj
arulraj@gatech.edu
Georgia Institute of Technology
Atlanta, USA

Shamkant Navathe
navathe@yahoo.com
Georgia Institute of Technology
Atlanta, USA

William Harris
wrharris@galois.com
Galois Inc
Portland, USA

Jinpeng Wu
jinpeng.wjp@alibaba-inc.com
Alibaba Group
HangZhou, CHINA

Abstract

Predicate-centric rules for rewriting queries is a key technique in optimizing queries. These include pushing down the predicate below the join and aggregation operators, or optimizing the order of evaluating predicates. However, many of these rules are only applicable when the predicate uses a certain set of columns. For example, to move the predicate below the join operator, the predicate must only use columns from one of the joined tables. By generating a predicate that satisfies these column constraints and preserves the semantics of the original query, the optimizer may leverage additional predicate-centric rules that were not applicable before.

Researchers have proposed syntax-driven rewrite rules and machine learning algorithms for inferring such predicates. However, these techniques suffer from two limitations. First, they do not let the optimizer constrain the set of columns that may be used in the learned predicate. Second, machine learning algorithms do not guarantee that the learned predicate preserves semantics.

In this paper, we present SIA, a system for learning predicates while being guided by counter-examples and a verification technique, that addresses these limitations. The key idea is to leverage satisfiability modulo theories to generate counter-examples and use them to iteratively learn a valid, optimal predicate. We formalize this problem by proving the key properties of synthesized predicates. We implement our approach in SIA and evaluate its efficacy and efficiency. We demonstrate that it synthesizes a larger set of valid predicates compared to prior approaches. On a collection of 200 queries derived from the TPC-H benchmark, SIA successfully rewrites 114 queries with learned predicates. 66 of these rewritten queries exhibit more than 2× speed up.

ACM Reference Format:
Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. 2021. Sia: Optimizing Queries using Learned Predicates. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457262

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457262

1 Introduction

Query optimization using predicate-centric rules is a widely-studied topic in database management systems (DBMSs) [19, 28, 37, 44, 45, 48]. Researchers have proposed several rules for *moving* predicates across query blocks to improve performance (*e.g.*, moving predicate below join operator [45], moving predicate below aggregation operator [28]). However, 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.

Opportunity: To facilitate the application of many predicate-centric optimization rules, we seek to synthesize predicates that only use a given set of columns. However, we must ensure that the semantics of the query is preserved while rewriting the query. We may transform $Q1$ to $Q2$:

```
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 synthesized fourth 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). Since this predicate does *not* alter the semantics of the query, it is a *valid* predicate. 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 how rewriting queries by introducing a valid predicate using columns from only one table allows the optimizer to push the predicate below the join operator that it could not previously do. Thus, synthesizing valid predicates over a given set of columns enables the optimizer to apply many predicate-centric rules that it could not previously leverage. We illustrate the importance of this problem in practice using a case study based on the Alibaba MaxCompute platform in §6.2.

It is important to ensure that the newly synthesized predicate is as *strong* as possible (*i.e.*, less selective) to improve performance. For example, the newly synthesized predicate could be $A.val > 10$ (instead of $A.val > 20$), but the resulting query execution plan is not optimal. We refer to a valid predicate as *optimal* if there exists no tuple that another valid predicate rejects but this predicate accepts.

Prior Work: In prior work, researchers have proposed syntax-driven rules for tackling this problem (*e.g.*, constant propagation [13] and transitive closure [24]). However, due to the complexity of predicates in real-world queries (*e.g.*, arithmetic operations, inequality relation, and logic combination), these syntax-driven 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 depend on the syntax of the original predicate.

Another promising line of recent research focuses on using a machine learning algorithm to train a binary classifier to accelerate inference [29]. In this case, a predicate is treated as a binary classifier that separates the desired tuples (`TRUE` samples) from the rest of the dataset (`FALSE` samples). However, this approach suffers from two 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 choose the set of columns that the synthesized predicate uses. 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.

Our Approach: We address these limitations in SIA, a system for synthesizing valid predicates. To address the first limitation, SIA leverages satisfiability modulo theories (SMT) [15, 16, 18, 32] 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, we prove the properties of tuples that should be selected or rejected by an optimal, valid predicate over the given set of columns. We encode these properties as an SMT formula and leverage the SMT solver to generate `TRUE` and `FALSE` samples for training the binary classifier. We prove that each `TRUE` sample must be accepted by a valid predicate, and that each `FALSE` sample must be rejected by an optimal, valid predicate. To improve the efficacy of the learning algorithm, we propose a novel learning process guided by counter-examples. In each iteration of the learning loop, if the learned predicate is not valid, then we 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 we generate `FALSE` samples that the optimal predicate should reject, but the current learned predicate selects.

We implemented our counter-example guided learning technique that is augmented with a verification scheme in SIA. We evaluate SIA on 200 queries derived from the TPC-H benchmark [43]. We 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× 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, we make the following contributions:

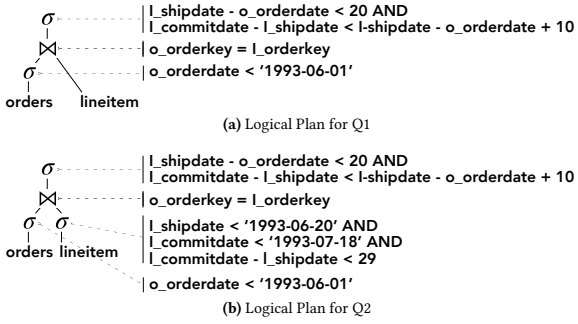


Figure 1: Logical Query Execution Plans – Queries Q1 and Q2 are semantically-equivalent. However, the optimizer computes a better query execution plan for Q2.

- We motivate the need for synthesizing valid predicates on a given set of columns using an example in §2.
- We formalize the problem and prove the key properties of synthesized predicates in §4.
- We present a novel technique for learning strictly-valid predicates while being guided by counter-examples in §3 and §5.
- We implement our approach in SIA and evaluate its efficacy and efficiency. We demonstrate that it synthesizes a larger set of valid predicates compared to syntax-driven rules and a non-iterative learning algorithm in §6.
- We demonstrate that SIA speeds up 66 queries derived from TPC-H benchmark exhibit by more than 2× in §6.

2 Motivation

We now motivate the need for automatically synthesizing predicates using an example. Consider the following query derived from the TPC-H benchmark [43].

```
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 [33, 40]. 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*).
- 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.

We run this query in the Postgres DBMS (v12) [4]. The query optimizer constructs the logical query execution plan P_1 shown in Fig. 1a. 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.

We 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 we run $Q2$ on Postgres, we obtain a 2× more performant plan P_2 shown in Fig. 1b. $Q1$ 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.

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× faster than $Q1$ (94 s). We defer a detailed description of our empirical setup to §6.

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 [13] and transitive closure transformation [24] cannot be applied in this case due to their dependence on syntax. For instance, constant propagation is only applicable for equality relation:

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

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.

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

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, we present a novel technique for learning predicates using a set of counter-examples while preserving the semantics of the query.

3 Overview

We first present an overview of our counter-example guided learning technique in §3.1. We then illustrate how this technique handles the motivating example given in §2. In particular, we discuss how SIA synthesizes *weaker predicates* that accept all the tuples that original predicates accept.

3.1 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 [16] .

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 Fig. 2a, 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 Fig. 2b, 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. We defer formal definitions to §4.2.

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

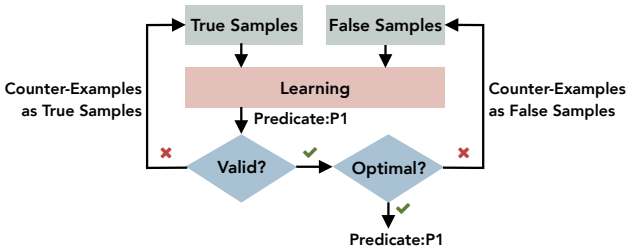


Figure 3: Counter-Example Guided Learning – The iterative learning process used in SIA.

to learn a valid and optimal p_1. We formalize these properties of unsatisfaction tuple in §4.1.

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. We discuss how SIA encodes p in §5.2. For FALSE samples, SIA takes the similar approach but feeds a complementary SMT formula to the solver. We defer a detailed discussion on how SIA generates training samples to §5.3.

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. Fig. 3 illustrates this process. In the first step, SIA takes the two sets of training samples generated in the previous stage and learns 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 given predicate p implies synthesized predicate p_1. 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. We describe the learning step in §5.4.

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 implies p_1. If p does not imply p_1, 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. We discuss how SIA generates such counter-examples in §5.5. SIA then loops back to the learning step with these additional true samples. 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. We formalize the notion of an optimal synthesized predicate in §4.1. 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, we configure the maximum number of iterations that SIA may take over the learning loop.

We 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.

3.2 Motivating Example

We next revisit the example in §2 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, we 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
```

We 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. We defer a discussion on how SIA encodes conditions and why we choose this encoding schema to §5.2.

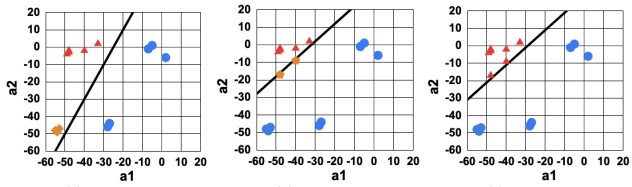
To generate the initial TRUE samples, SIA repeatedly feeds the symbolic formula to the solver. In each iteration, it generates a model with concrete values for $a1$, $a2$ and $b1$ that satisfy the original predicate p. It then adds additional constraints so that the model obtained in the next iteration 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)
```

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



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3

Figure 4: Learning Process – Three iterations of the learning loop in SIA guided by counter-examples.

predicate from these samples:

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

Fig. 4 illustrates the learning process. As shown in Fig. 4a, 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 we set $b1$ to -5, then the tuple satisfies p. But this pair of values is rejected by the current p_1. We represent these counter-examples using yellow diamonds on the bottom left in Fig. 4a. 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 Fig. 4b, 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 Fig. 4b. 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 Fig. 4b, 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 $a2$.

4 Problem Formulation

We now formalize the problem of learning a valid, optimal predicate. We first define the syntax of predicates that SIA supports and our problem formulation in §4.1. We then present the key conceptual insights in §4.2.

4.1 Problem Definition

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

P ::= E CP E | P L P | **Not** P; E ::= Column | Const | E OP E

CP ::= $>$ | $<$ | $=$ | $\leq$ | $\geq$; OP ::= $+$| $-$ | $\times$ |$\div$; L ::= AND | 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 τ_{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. We 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 $\text{Preds}_{\text{Cols}}$. Note that each predicate $p \in \text{Preds}_{\text{Cols}}$ is a predicate over all sets of column Cols′ such that Cols′ $\subseteq$ Cols. A *tuple over columns* Cols is a map from each column col $\in$ Cols to a value of corresponding column type τ_{col}. The set of tuples over Cols is denoted $\text{Tuples}_{\text{Cols}}$. Each predicate $p \in \text{Preds}_{\text{Cols}}$ can be evaluated on each tuple $t \in \text{Tuples}_{\text{Cols}}$ to produce a boolean output, denoted $p(t)$. If we substitute t(col) for each column col in p and it evaluates to True (*i.e.*, $p(t)$ is True), then we say that t *satisfies* p (alternately, that p *accepts* t). If $p(t)$ 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′.

Def 1. *Predicate* $p \in \text{Preds}_{\text{Cols}}$ *over columns* Cols implies *predicate* $p' \in \text{Preds}_{\text{Cols}}$ *over* Cols *if for each tuple* $t \in \text{Tuples}_{\text{Cols}}$ *that satisfies* p *(i.e.,* $p(t) = \text{True}$*),* t *also satisfies* p' *(i.e.,* $p'(t) = \text{True}$*).*

The fact that p implies p′ is denoted $p \implies p'$.

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.

Def 2. $p' \in \text{Preds}_{\text{Cols}'}$ *is a* valid dimensionality reduction *of predicate* $p \in \text{Preds}_{\text{Cols}}$ *with* Cols′ $\subseteq$ Cols *if* $p \implies p'$.

Valid dimensionality reduction enables the application of optimization rules related to predicates [28, 45]. 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 Cols′ ⊆ col. The requirement that a dimensionality reduction over Cols′ is in $\text{Preds}_{\text{col}}$ 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. We will be primarily concerned with synthesizing predicates that are as less selective as possible.

Def 3. *$p_1 \in \text{Preds}_{\text{Cols}'}$, a valid reduction of $p \in \text{Preds}_{\text{Cols}}$ (Def 2) is* optimal *if for each $p_2 \in \text{Preds}_{\text{Cols}'}$ that is a valid dimensional reduction of p to* Cols′*, it holds that $p_1 \implies p_2$.*

We prove that every predicate has an optimal dimensionality reduction to each subset of its columns in §4.2. One of our key contribution in SIA is an automatic procedure for synthesizing a valid dimensionality reduction of p to Cols′, given a predicate $\text{p} \in \text{Preds}_{\text{Cols}}$ and a set of columns Cols′ ⊆ Cols.

4.2 Key Conceptual Insights

Given the problem definition in §4.1, we now discuss the key insights for solving it. First, we 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, we 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, we first define the *restriction* and *extension* properties of tuples that determine the set of columns that they may refer to. For a tuple $t \in \text{Tuples}_{\text{Cols}}$ and a set of columns Cols′ ⊆ Cols, restriction of t to columns in Cols′ is denoted by $t|_{\text{Cols}'}$. In this case, t extends $t|_{\text{Cols}'}$ 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.

Def 4. *For a set of columns* Cols′ ⊆ Cols *and predicate $p \in \text{Preds}_{\text{Cols}}$, tuple $t \in \text{Tuples}_{\text{Cols}'}$ is a* feasible restriction *for p if some extension of t to* Cols *satisfies p.*

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

Properties of Dimensionality Reduction: In order to prove the key properties of dimensionality reduction, we will use the following lemma which establishes that predicates over a restricted set of columns treat tuples and their restrictions equivalently.

Lemma 1. *For columns* Cols′ ⊆ Cols *and predicate $p \in \text{Preds}_{\text{Cols}'}$, $p(t) = p(t|_{\text{Cols}'})$ for each tuple $t \in \text{Tuples}_{\text{Cols}}$.*

Dimensionality reduction is closed under conjunction.

Lemma 2. *If $p_0, p_1 \in \text{Preds}_{\text{Cols}'}$ are valid dimensionality reductions of predicate $p \in \text{Preds}_{\text{Cols}}$ to* Cols′*, then $p_0 \wedge p_1$ is a valid dimensionality reduction of p to* Cols′.

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

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

Based on Lemma 4, 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 $\text{p}_1(t)$ is `TRUE`, then p_1 is an optimal predicate. Thus, we 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:

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

This formula contains an alternating quantifier that supports linear arithmetic over integer, real number, and bit vectors. So it is a decidable problem [14, 17]. Thus, the problem of deciding if a given valid synthesized predicate is optimal is also decidable.

5 Synthesizing Predicates

In this section, we first present the overall algorithm that SIA uses to synthesize a valid, optimal predicate in §5.1. We then cover the key sub-procedures in the following sub-sections. In §5.2, we discuss how SIA encodes a predicate as an SMT formula. In §5.3, we describe how SIA generates the initial learning samples. In §5.4, we explain why SIA uses a linear SVM and discuss how it uses this machine learning model to learn a predicate. Finally, in §5.5, we present how SIA verifies if the learned predicate is valid, and generates counter-examples accordingly.

5.1 Predicate Synthesis

Alg. 1 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 (§5.4) to learn a new predicate p_2 based on the given training samples. The Learn procedure returns a predicate that is guaranteed to classify

Algorithm 1: 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 p1
1  Procedure Synthesize(p, Cols′)
2      Procedure SynthesizeAux(p, Cols′, p1, Ts, Fs, i)
3          if i > max then return p1;
4          p2 ← Learn(Ts, Fs)
5          isValid ← Verify(p2, p)
6          if isValid then
7              p3 ← p1 ∧ p2
8              Fs1 ← CounterF(p3, p, Fs)
9              if Fs1 = ∅ then return p3;
10             else return
                 SynthesizeAux(p, Cols′, p3, Ts, Fs ∪ Fs1, i + 1);
11         else
12             Ts1 ← CounterT(p1, p, Ts)
13             return SynthesizeAux(p, Cols′, p1, Ts ∪ Ts1, Fs, i + 1)
14     (Ts, Fs) ← GenerateSamples(p, Cols)
15     return SynthesizeAux(p, Cols, True, Ts, Fs, 0)
```

all Ts samples as `TRUE`. The SynthesizeAux procedure then uses the Verify procedure (§5.5) 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 (§5.4) 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 (§5.4) 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 (§5.3) 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.

Counter-Example Guided Synthesis (CEGIS): Modern SMT solvers use CEGIS to solve the constrained horn clauses problem (CHC) (*e.g.*, duality in Z3 [8]). So, the problem of synthesizing a valid predicate may be encoded as a CHC problem and solved using duality. However, SIA not only tries to synthesize a valid predicate, but also an *highly selective* predicate. If we directly encode the property that original predicate p implies a valid synthesized predicate p_1 as a CHC problem, the solver may always return the trivial predicate `TRUE`. Even if we encoded the property that p_1 implies the negative of `FALSE` samples generated by SIA, the solver may always return a predicate that is simply the conjunction of not equivalent to `FALSE` samples. In both cases, the solver returned by the predicate is not optimal and therefore does not reduce query execution time. This is why SIA leverages an SVM-based CEGIS algorithm to generate optimal predicates.

5.2 Predicate Encoding

Since SIA leverages the solver in several procedures (*e.g.*, CounterT), we 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 §4.1, 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.

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 [49]. 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. 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.

Non-Linear Arithmetic: The satisfaction problem of a SMT formula with integer, non-linear arithmetic is undecidable [30]. So, SIA cannot directly convert a predicate with multiplication or division of two integer-valued columns. 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).

5.3 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 we proved in Lemmas 3 and 4, 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:

$$p \wedge \text{NotOld}$$

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 Def 4, 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 formula into the solver to generate FALSE samples:

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

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 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: We use additional heuristics for forcing the solver to generate useful training samples depending on the machine learning model. For example, SIA constrain that the values must not be equal to zero.

5.4 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 [10, 34, 42] 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 [17].

Algorithm 2: Procedure for learning a valid predicate

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

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. 2, 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.

Predicate Construction: Each linear SVM model is a vector of weights $\vec{w}$. Each dimension in the space of samples corresponds to a column. SIA constructs the predicate by setting the sum of the products of the weight and the corresponding column to be greater than zero [11]. Since SIA constructs a arithmetic binary predicate for each SVM model, the disjunction of models maps to to a disjunction of predicates.

5.5 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:

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

Both formulae use the encoding scheme that supports three-valued logic (§5.2). 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 Def 2, 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:

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

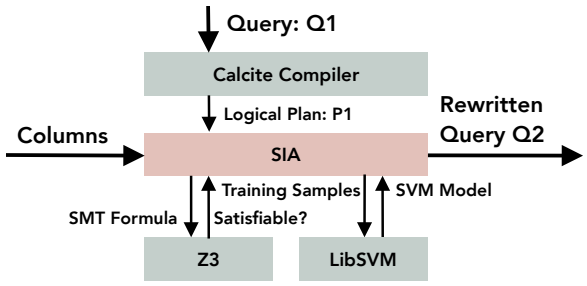


Figure 5: Architecture of SIA– SIA leverages three components: (1) Calcite query optimization framework, (2) Z3 SMT solver, and (3) SVM library.

Here, p represents the original predicate and $\neg p_1$ 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:

$$\exists col_1 \in Cols' \; s.t. p_1 \wedge NotOld \wedge (\forall col_2 \notin Cols' \; s.t. \neg p)$$

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 4, p_1 is optimal.

6 Evaluation

We now describe our implementation and evaluation of SIA. We begin with a description of our implementation in §6.1. We then present a case study based on the Alibaba MaxCompute system in §6.2. We next discuss how we construct a collection of queries derived from the TPC-H benchmark [43] to evaluate SIA in §6.3. We then report the results of our comparative analysis of SIA in §6.4 and §6.5. We next cover the impact of SIA on runtime performance in §6.6. We discuss the limitations of SIA in §6.7.

6.1 Implementation

The architecture of SIA is illustrated in Fig. 5. 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 [2]. 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 [5]. ❸ The third component is an SVM library [3]. We have provided the source code of SIA along with this submission and plan to open-source it if the paper is published.

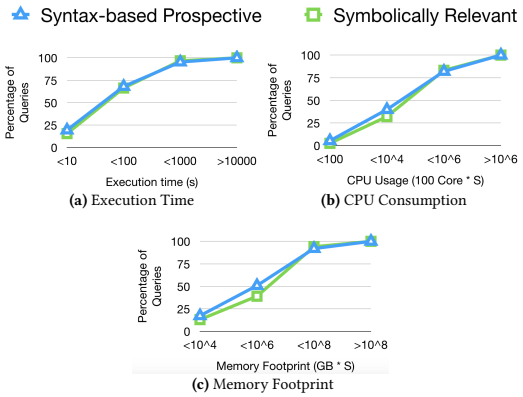


Figure 6: Case Study Metrics: Execution time, CPU consumption, and memory footprint of: (1) syntax-based prospective, and (2) symbolically relevant queries.

6.2 Case Study

To motivate the importance of synthesizing predicates on a given set of columns, we present a case study based on the Alibaba MaxCompute system. MaxCompute is a general purpose, fully managed data processing platform for large-scale OLAP workloads [1]. We focus on the predicate push-down through join optimization rule in this experiment. We examined one day's worth of production queries on the MaxCompute platform. We first examine the syntax of the queries to find those that contain *at least* one predicate that refers to columns across multiple tables, such that at least one of those tables (say $\mathcal{T}$) does not have another predicate only referring to its column(s). In this case, the optimizer may not be able to push-down the predicate and the impact of not being able to do so is significant since $\mathcal{T}$ must be fully scanned. We refer to the queries satisfying this property as **syntax-based prospective** queries. Since there is a predicate that refers to columns from the table, it is *possible* that there exists a weaker predicate that only refers to columns from that $\mathcal{T}$. We note that these queries may have multiple predicates satisfying this property such that many tables must be fully scanned for processing the query. For the target databases in MaxCompute whose sizes range from TBs to PBs, these queries have a significant overhead.

Among the syntax-based prospective queries, we next examine if SIA is capable of finding a predicate that only refers to columns from $\mathcal{T}$. SIA encodes the original predicate as a SMT formula, and then tries to generate unsatisfaction tuples for the given table. If

SIA successfully generates an unsatisfication tuple, then it synthesizes a predicate that only refers to columns in $\mathcal{T}$, thereby enabling the application of the predicate push-down rule. We refer to these queries as **symbolically relevant** queries. Among the syntax-based prospective queries, some queries may not be symbolically relevant only because SIA does not support certain complex predicates. It does not imply that a weaker predicate does not exist.

We report the query execution time, CPU consumption, and memory footprint of both categories of queries: 204, 287 syntax-based prospective queries and 26, 104 symbolically relevant queries in Fig. 6. The most notable observation in Fig. 6 is that majority of these queries (74.63%) take longer than 10 seconds to execute and will benefit from SIA. This is sufficient to justify the optimization time for those queries. More importantly, most of these queries are stored procedures that are optimized only once and their query execution plans are stored in a plan cache. For these queries, the optimizer may use SIA with an explicit timeout (*e.g.*, 5 seconds).

6.3 Benchmark

To evaluate the efficacy of SIA in generating valid predicates, we construct a collection of queries based on the TPC-H benchmark [43]. The reasons for constructing this benchmark are twofold. First, we seek to make the queries publicly available. Second, we generate queries to simulate the characteristics of predicates in production query workloads. In particular, we use a sub-query of TPC-H Q4 with more complex predicates. 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 between: (1) a column, (2) a binary arithmetic expression, (3) a date constant, or (4) an interval constant (*i.e.*, number of days). We constrain *predicate* to use three columns from *lineitem* table (*l_shipdate*, *l_commitdate*, and *l_receiptdate*), and one column from *orders* table (*o_orderdate*). We ensure that each generated binary predicate refers to the column in *orders*. Thus, the optimizer cannot push down the original predicate below the join operator to the *lineitem* table. Each *predicate* contains from three through eight terms. We re-generate the query if the *predicate* cannot be satisfied by any tuples. In this manner, we construct a collection of 200 queries.

Baselines: We 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× 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 1.

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). We implement a syntax-driven transitive closure transformation for our comparative analysis.

	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 1: Baselines – We compare SIA against two non-iterative baselines.

6.4 Efficacy of SIA

In this experiment, we examine whether SIA is able to effectively synthesize predicates over the given set of columns. We run SIA on each query with all possible subsets of three columns *l_shipdate*, *l_commitdate*, and *l_receiptdate* from *lineitem* table. In SIA, we set the number of initial `TRUE` and `FALSE` samples to 10, respectively. In each iteration of the learning loop, we configure the number of newly added training samples to 5 (either `TRUE` or `FALSE` depending on the requirements of the learning process). We 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, we 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_$v1$, we 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_$v2$, we set the number of initial `TRUE` and `FALSE` samples to 220, respectively (2× the number of samples given to SIA_$v1$). We conduct this experiment on a commodity server (Intel Core i7-860 processor with 16 GB RAM).

Table 2 shows the results of this experiment. For each query, we configure SIA to generate synthesized predicates with varying complexity (ranging from one through three columns from *lineitem* table). We 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). We 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 we classify it as two possible predicates in the first category.

The most notable observation in Table 2 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. We 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_$v2$). We 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_$v1$

		SIA		Transitive Closure	SIA _v1		SIA _v2	
# of Used Columns	# of Possible Predicates	# of Valid	# of Optimal	# of Valid	# of Valid	# of Optimal	# of Valid	# of Optimal
one	233	182	158	18	158	75	166	98
two	160	102	20	4	11	3	17	4
three	30	20	0	0	2	0	1	0

Table 2: Efficacy of SIA– Comparative analysis of SIA against the baselines with respect to their ability to synthesize valid (possibly optimal) predicates.

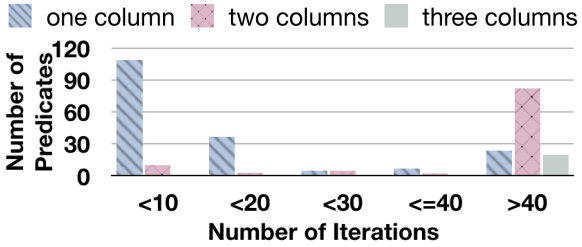
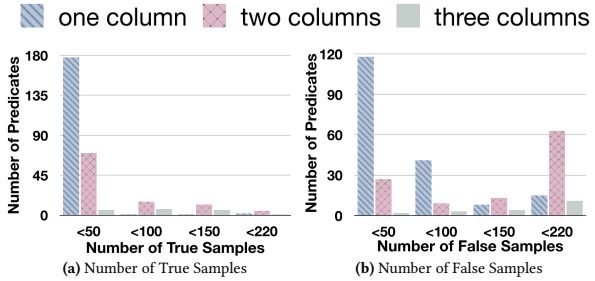
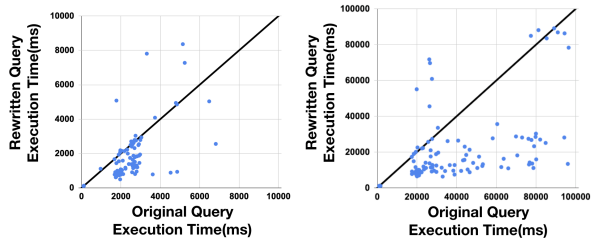


Figure 7: Efficiency of Learning Loop – Average number of iterations that SIA takes to converge to an optimal predicate.



(a) Number of True Samples **(b)** Number of False Samples

Figure 8: Sample Distribution – Distribution of the number of training samples generated by SIA before the final iteration.



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

Figure 9: Impact on Runtime Performance – Comparison of the time taken to execute the original and rewritten queries.

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.

6.5 Efficiency of SIA

In this experiment, we study the efficiency of SIA. We first measure the time taken by SIA and its baselines to synthesize the predicates. We 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 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 3 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, we must reduce the number of generated training samples.

Learning Loop: We next examine the efficiency of the learning loop. We measure the number of iterations SIA takes to synthesize the optimal predicate. Fig. 7 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 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. We discuss this in §6.7.

We 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. Fig. 8a 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. Fig. 8b 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. We discuss this limitation in §6.7.

6.6 Impact on Runtime Performance

We next conduct an experiment to study the impact of SIA on runtime performance. In particular, we 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. We measure the runtime performance of these 114 queries (without and with the synthesized predicates). It is important to note that the rewritten queries are semantically equivalent to their original counterparts. We execute these queries on the TPC-H database on PostgreSQL (v12). We consider two scale-factors: one and ten.

The results are shown in Fig. 9. The x-axis and y-axis in these plots represent the time taken to execute the original and rewritten queries, respectively. We 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

# of Used Columns	SIA			SIA _v1			SIA _v2		
	Generation Time (ms)	Learning Time (ms)	Validation Time (ms)	Generation Time (ms)	Learning Time (ms)	Validation Time (ms)	Generation Time (ms)	Learning Time (ms)	Validation Time (ms)
one	893.2	1.8	98.5	2625	0.5	1	9304	1.9	11.3
two	2933	14.6	281.4	2739	1.0	7.3	10159	3.2	11.64
three	4154	38.9	328.2	3801	1.0	8.5	11859	5.0	12.0

Table 3: Efficiency of SIA– Comparative analysis of SIA against the baselines with respect to their time taken to synthesize predicates.

Scale Factor	# of Faster	Avg. Selectivity	# of 2× Faster	Avg. Selectivity	# of Slower	Avg. Selectivity	# of 2× Slower	Avg. Selectivity
one	85	0.76	36	0.69	29	0.97	2	0.98
ten	95	0.78	66	0.74	19	0.96	4	0.94

Table 4: Selectivity – Average selectivity of synthesized predicates with respect to $lineitem$ table. We classify them based on their performance impact.

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 (Fig. 9b). 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, we measure the selectivity of the predicates with respect to the *lineitem* table. We classify these 114 synthesized predicates into four categories based on their impact on runtime performance on the TPC-H database. As summarized in Table 4, 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. The newly added predicate allows the optimizer to push down the predicate in the re-written query. So, the execution plan of the re-written query contains an additional filter operator before the join operator compared to that of the original query. If the selectivity of the synthesized predicate is low, then the execution time of the additional filter higher is higher than the time saved in the join operator. In this case, the re-written query takes more time to execute compared to the original query.

6.7 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 during the learning step. 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, during the learning step, 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. We note that SIA always discards these incorrect predicates during the verification step.

7 Related Work

Predicate Synthesis: Researchers have focused on learning approximate predicates to accelerate query execution [25, 29, 39]. This line of research includes: training probabilistic predicates to accelerate inference in machine learning pipelines [29], inferring simpler approximate predicates from expensive UDFs [25]. SIA differs from these efforts in two ways. First, the predicate it learns is strictly weaker than the original predicate. Second, it is guaranteed to preserve the semantics of the original predicate.

Another seminal work in this area focuses on inferring strictly weaker predicates for expensive mining models [41]. SIA differs from this work in that the predicates it generates are guaranteed to use the given set of columns Cols′. In contrast, the predicates inferred in the previous work use all the input columns for the model (Cols). Generating predicates that only use a subset of columns enables the optimizer to apply predicate-centric optimization rules.

Another line of research focuses on inferring predicates using column's statistics [26] and data correlations [23, 27]. SIA seeks to synthesize predicates with a given set of columns by only examining the original predicate (and does not rely on real data).

Symbolic Reasoning: Researchers have proposed using an SMT solver to generate tuples for: (1) database testing [6, 46, 47], (2) to prove or disprove query equivalence [12, 49]. The problem of synthesizing a valid reduced predicate maps to a constrained horn clause (CHC) problem. Many CHC solvers have been proposed [7, 21, 22, 31, 35, 50], including techniques for synthesizing invariant based on learning [9, 20, 36, 38]. However, these techniques only require the predicate to be valid. Thus, unlike SIA, they may always return a trivial valid predicate (*e.g.*, `TRUE`).

8 Conclusion

In this paper, we presented the design and implementation of SIA, a system for synthesizing a valid predicate based on the given predicate and a subset of columns in the given predicate. SIA uses an SMT solver to generate samples for learning the predicate and to verify the learned predicate. We formalize the problem of computing a valid, optimal predicate and present a proof for its optimality of the synthesized predicate. Our evaluation shows that SIA effectively synthesizes valid predicates. On a collection of 114 queries derived from the TPC-H benchmark that are rewritten by SIA, 66 queries exhibit more than 2× speed up.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation (IIS-1850342, IIS-1908984), ONR Award(AWD-101549-S1), Alibaba Innovative Research (AIR) Program, and Intel. We thank Zhenyu Hou and Xiaowei Jiang for their constructive feedback and helping us access the query log. We are grateful to the reviewers for their insightful feedback.

References

[1] [n.d.]. Alibaba MaxCompute. https://www.alibabacloud.com/product/maxcompute.
[2] [n.d.]. Apache Calcite Project. http://calcite.apache.org/.
[3] [n.d.]. LibSVM. https://github.com/cjlin1/libsvm.
[4] [n.d.]. PostgreSQL. https://www.postgresql.org/.
[5] [n.d.]. Z3Prover: Z3 Theorem Prover. https://github.com/Z3Prover/z3.
[6] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Laleye, and Sarfraz Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *ASE*.
[7] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. 2012. Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In *VMCAI*.
[8] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses.
[9] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018. *ICE-Based Refinement Type Discovery for Higher-Order Functional Programs*.
[10] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* 2, 3 (2011).
[11] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2012. Using Program Synthesis for Social Recommendations. In *CIKM*. 1732–1736.
[12] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated SQL Prover.. In *CIDR*.
[13] Mariano P. Consens, Alberto O. Mendelzon, Dimitra Vista, and Peter T. Wood. 1995. Constant Propagation Versus Join Reordering in Datalog. In *RIDS*.
[14] Dennis W. Cooper. 1972. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*.
[15] Martin Davis, George Logemann, and Donald W. Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* (1962).
[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.
[17] Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Alex Aiken. 2012. Minimum Satisfying Assignments for SMT. In *CAV*.
[18] Bruno Dutertre. 2014. Yices 2.2. In *CAV*.
[19] Mostafa Elhemali, César A. Galindo-legaria, Torsten Grabs, and Milind M. Joshi. 2007. Execution Strategies for SQL Subqueries. In *SIGMOD*.
[20] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *POPL*.
[21] Sergey Grebenshchikov, Nuno Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. (2012).
[22] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2010. Nested Interpolants. In *POPL*.
[23] Ihab F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. 2004. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*.
[24] Yannis Ioannidis and Raghu Ramakrishnan. 1988. Efficient Transitive Closure Algorithms.. In *VLDB*.
[25] Manas Joglekar, Hector Garcia-Molina, Aditya Parameswaran, and Christopher Re. 2015. Exploiting Correlations for Expensive Predicate Evaluation. In *SIGMOD*.
[26] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates through Joins in Big-Data Clusters. (2019).
[27] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stan Zdonik. 2009. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. (2009).
[28] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. 1994. Query Optimization by Predicate Move-Around. In *VLDB*.
[29] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *SIGMOD*.
[30] Yuri V. Matiyasevich. 2005. Hilbert's Tenth Problem and Paradigms of Computation. In *CiE*.
[31] Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *CAV*.
[32] Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* (1979).
[33] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. https://www.gartner.com/doc/2657815/.
[34] John C Platt. 1999. Advances in Kernel Methods. (1999).
[35] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2013. Disjunctive Interpolants for Horn-Clause Verification. In *CAV*.
[36] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivancic, and Aarti Gupta. 2008. Dynamic inference of likely data preconditions over predicates by tree learning. In *ISSTA*.
[37] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. 1996. Cost-Based Optimization for Magic: Algebra and Implementation. In *SIGMOD*.
[38] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya Nori. 2013. Verification as Learning Geometric Concepts. In *SAS*.
[39] Narayanan Shivakumar, Hector Garcia-Molina, and Chandra Chekuri. 1998. Filtering with Approximate Predicates. In *VLDB*.
[40] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *SIGMOD*. 731–742.
[41] Chaudhuri Surajit, Narasayya Vivek, and Sarawagi Sunita. 2002. Efficient evaluation of queries with mining predicates. In *ICDE*.
[42] J. A. K. Suykens and J. Vandewalle. 1999. Least Squares Support Vector Machine Classifiers. *Neural Process. Lett.* (1999).
[43] The Transaction Processing Council. 2013. TPC-H Benchmark (Revision 2.16.0). http://www.tpc.org/tpch/.
[44] Nga Tran, Andrew Lamb, Lakshmikant Shrinivas, Sreenath Bodagala, and Jaimin Dave. 2014. The Vertica Query Optimizer: The case for specialized query optimizers. In *ICDE*.
[45] Jeffrey ullman. 1989. Principle of database and knowledge-bas systems.
[46] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. 2009. Symbolic Query Exploration. In *FormaliSE*.
[47] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL Query Explorer. In *LPAR*.
[48] Brett Walenz, Sudeepa Roy, and Jun Yang. 2017. Optimizing Iceberg Queries with Complex Joins. In *SIGMOD*.
[49] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Dong Xu. 2019. Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. *PVLDB* 12, 11 (2019), 1276–1288.
[50] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver.