Polygon: Symbolic Reasoning for SQL using Conflict-Driven Under-Approximation Search

PINHAN ZHAO, University of Michigan, USA
YUEPENG WANG, Simon Fraser University, Canada
XINYU WANG, University of Michigan, USA

We present a novel symbolic reasoning engine for SQL which can efficiently generate an input I for n queries $P_1, \cdots, P_n$, such that their outputs on I satisfy a given property (expressed in SMT). This is useful in different contexts, such as disproving equivalence of two SQL queries and disambiguating a set of queries. Our first idea is to reason about an *under-approximation* of each P_i—that is, a subset of P_i's input-output behaviors. While it makes our approach both semantics-aware and lightweight, this idea alone is incomplete (as a fixed under-approximation might miss some behaviors of interest). Therefore, our second idea is to perform *search* over an *expressive family* of under-approximations (which collectively cover all program behaviors of interest), thereby making our approach complete. We have implemented these ideas in a tool, Polygon, and evaluated it on over 30,000 benchmarks across two tasks (namely, SQL equivalence refutation and query disambiguation). Our evaluation results show that Polygon significantly outperforms all prior techniques.

CCS Concepts: • **Theory of computation → Automated reasoning**; **Program reasoning**; • **Software and its engineering → Formal methods**.

Additional Key Words and Phrases: Automated Reasoning, Testing, Databases.

ACM Reference Format:
Pinhan Zhao, Yuepeng Wang, and Xinyu Wang. 2025. Polygon: Symbolic Reasoning for SQL using Conflict-Driven Under-Approximation Search. *Proc. ACM Program. Lang.* 9, PLDI, Article 200 (June 2025), 26 pages. https://doi.org/10.1145/3729303

1 Introduction

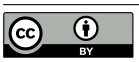
The general problem we study in this paper is the following.

> Given n programs $P_1, \cdots, P_n$, how to generate an input I such that their outputs on I satisfy a given property C (which is expressed as an SMT formula over variables $y_1, \cdots, y_n$). That is, $C\left[y_1 \mapsto P_1(I), \cdots, y_n \mapsto P_n(I)\right]$ is true.

This problem can be viewed as a form of "test input generation" task, which can be instantiated to different applications with different *application conditions* C. One example application is to disprove equivalence of two programs, with C being simply $y_1 \neq y_2$. Another is program disambiguation—e.g., find I that can divide n programs into disjoint groups, such that: (i) programs within the same group return the same output given I, while (ii) outputs of those from different groups are distinct. Finding such distinguishing inputs is crucial for example-based program synthesis [32, 33, 46]

Instantiation to SQL. We focus on one instantiation of this general problem, where P_i's are written in SQL (which is a widely used domain-specific language). The aforementioned applications still

Authors' Contact Information: Pinhan Zhao, University of Michigan, Ann Arbor, USA, pinhan@umich.edu; Yuepeng Wang, Simon Fraser University, Burnaby, Canada, yuepeng@sfu.ca; Xinyu Wang, University of Michigan, Ann Arbor, USA, xwangsd@umich.edu.



ACM 2475-1421/2025/6-ART200
https://doi.org/10.1145/3729303

hold—generating counterexamples to refute SQL equivalence is useful in many ways [9, 10, 15, 28], and query disambiguation is critical for example-based SQL query synthesis [6, 42, 67, 68].

State-of-the-art. To the best of our knowledge, no existing work can *efficiently* generate such inputs for an *expressive* subset of SQL. While conventional testing-based approaches—e.g., those based on fuzzing [18] and evolutionary search [7]—can generate many inputs quickly, they do not consider the (highly complex) semantics of SQL. As a result, they fall short of capturing subtle differences across queries, which is crucial for generating distinguishing inputs. While some works (e.g., XDATA [9, 10]) consider basic semantic information (such as join and selection conditions), they support a very limited subset of SQL and are specialized in equivalence checking. On the other end of the spectrum, techniques based on formal methods [15, 28, 66, 69, 71] perform symbolic reasoning—typically by encoding *complete* query semantics in SMT. While boiling the problem down to SMT solving, these methods often create large SMT formulas that are computationally expensive to solve, especially for problems that involve many large queries with complex semantics. For instance, for operators like group-by and aggregation, fully encoding all grouping possibilities would cause an exponential blow-up in the resulting formula's size.

Key challenge. While clearly critical to consider *some* semantic information, it would significantly slow down the reasoning process if we consider the *full* semantics. The core challenge hence is how to take into account SQL semantics in a way that is *lightweight without hindering completeness.* Prior works fall into two extremes of the spectrum: either (1) fully encoding semantics for all inputs thus heavyweight and not scalable, or (2) fast by considering no or very little semantic information but at the cost of missing inputs of interest frequently (i.e., incomplete).

Our key insight. Our key insight is to reason about an *under-approximation (or UA)* of the program. This, while much faster than analyzing the full program semantics, is incomplete. Therefore, we also perform search over an *expressive family of UAs* (which collectively cover all program behaviors of interest), thereby making our entire approach complete.

Specifically, given $P_1, \cdots, P_n$ and application condition C, we begin with a UA for each P_i which encodes a subset $\mathbb{O}_i$ of *reachable* outputs for P_i (together with their corresponding inputs). Here, an output is reachable, if it can indeed be returned by the program on an input [49]. We then check if there exist $O_1 \in \mathbb{O}_1, \cdots, O_n \in \mathbb{O}_n$ such that $C[y_1 \mapsto O_1, \cdots, y_n \mapsto O_n]$ is true. If so, we can easily solve our problem by deriving the corresponding input given the under-approximate semantics (or UA semantics) of $P_1, \cdots, P_n$. Otherwise, we try again but use a different choice of UA. This process terminates, when a desired UA is found or no such UA can be found for the given family of UAs.

Below we briefly summarize the key challenges and our solutions. Section 2 will further illustrate our approach using a concrete example.

Under-approximating SQL query semantics. The first challenge is how to under-approximate a query P. Our idea is to encode P's UA semantics in an SMT formula Ψ whose models correspond to *genuine* input-output pairs of P. In other words, Ψ always encodes reachable outputs. On the other hand, not all reachable outputs are necessarily encoded by Ψ; therefore Ψ is an under-approximation. Our first novelty lies in a *compositional* method to build Ψ for P, by conjoining Ψ_i for each AST node v_i of P. Here, each Ψ_i under-approximates the semantics of the query operator F at v_i.

Defining a family of under-approximations. This compositional encoding method lends itself well to addressing our second challenge—how to define a family of UAs for a query P? Our idea is to first define a family $\mathcal{U}_F$ of UAs for each query operator F in the query language, which collectively covers all inputs of interest to F. Then, we define a *UA map* M for P, which maps each AST node v_i in P to some UA in $\mathcal{U}_{op(v_i)}$. Each UA map M can be encoded into an SMT formula $\Psi := \mathit{Encode}(M)$,

by taking the conjunction of the encodings of M's entries. We can define a family of such UA maps, where each Ψ under-approximates P in a different way. This approach can be further generalized to under-approximate n queries, by extending the UA map M to include *all* AST nodes *across* n *queries*. Conceptually, if $\Phi := \Psi \wedge C$ is satisfiable, we can derive a satisfying input I from a satisfying assignment of Φ that solves our reasoning task for n queries.

Conflict-driven under-approximation search. Our third research question is: how to efficiently find a *satisfying UA map* M (i.e., $Encode(M) \wedge C$ is satisfiable)? We propose a novel search algorithm that explores a sequence of M_i's, until reaching a satisfying one. Each iteration is fast, and the number of iterations is typically small. Our novelty lies in how we generate M_{i+1} from an unsatisfiable M_i. In particular, we first obtain a subset of M_i's entries for a subset V of AST nodes in M_i—i.e., $M_i \downarrow V$—such that $Encode(M_i \downarrow V) \wedge C$ is unsat. In other words, $M_i \downarrow V$ is a *conflict*. Then, we resolve this conflict by mapping some nodes in V to new UAs, such that $M_{i+1} \downarrow V$ is satisfiable. During this process, we might also need to adjust mappings for nodes outside V, but we do not analyze their semantics. Our novelty lies in the development of a lattice structure of UAs and an algorithm that exploits this structure for efficient conflict resolution.

Evaluation. We have implemented these ideas in a tool called Polygon[1], and evaluated it on two applications. The first one is SQL query equivalence refutation. Our evaluation result on 24,455 benchmarks reveals that Polygon can disprove a large number of query pairs using a median of 0.1 seconds—significantly outperforming all prior techniques. Our evaluation result on a total of 6,720 query disambiguation benchmarks also shows Polygon significantly beats all existing approaches both in terms of the number of benchmarks solved and the solving time.

Contributions. This paper makes the following contributions.

- Develop a new symbolic reasoning engine for SQL based on under-approximate reasoning.
- Formulate the reasoning task as an under-approximation search problem.
- Propose a compositional method to define the search space of under-approximations.
- Design an efficient conflict-driven algorithm for searching under-approximations.
- Evaluate an implementation, Polygon, of these ideas on more than 30,000 benchmarks.

2 Overview

Background on SQL equivalence checking. In this section, we will present a simple example to illustrate how our approach works for checking the equivalence of two SQL queries. This is an important problem with applications in various downstream tasks. One such task is automated grading of SQL queries (consider LeetCode[2]): a user-submitted query needs to be checked against a predefined "ground-truth" query; in case of non-equivalence, provide a counterexample to users. Another task is to validate query rewriting, where a slow query P_1 is transformed to a faster query P_2 using rewrite rules. We can perform translation validation by checking P_1 is equivalent to P_2. A counterexample in this case can help developers fix the incorrect rewrite rule. We also refer readers to recent work on SQL equivalence checking [15, 28] for more details.

Equivalence refutation example. Consider a database with the following three relations.

$$\begin{aligned} \textit{Customers} &: [\text{ customer\_id, customer\_name, email }] \\ \textit{Contacts} &: [\text{ user\_id, contact\_name, contact\_email }] \\ \textit{Invoice} &: [\text{ invoice\_id, price, user\_id }] \end{aligned}$$

[1]When you under-approximate circle (rhymes with SQL), you get Polygon.
[2]https://leetcode.com/ is an online platform that provides coding problems in different languages (including SQL).

```
v1: SELECT invoice_id, T.customer_name, cnt
    FROM Invoices I
v2: LEFT JOIN (
L2:     SELECT A.customer_id, customer_name,
               COUNT(contact_name) AS cnt
        FROM Customers A
v4:           LEFT JOIN Contacts B
φ2:           ON A.customer_id = B.user_id
v3:     GROUP BY A.customer_id, customer_name) T
φ1: ON I.user_id = T.customer_id
```

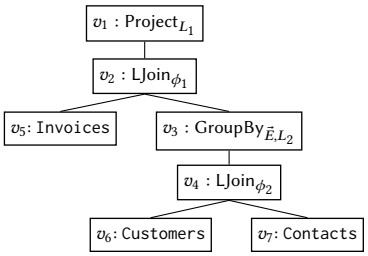


Fig. 1. SQL query P (left) and its corresponding AST (right). While P is written in standard SQL syntax, we express its AST following our grammar in Figure 7. In particular, an AST node is labeled with a query operator, with non-query parameters (such as column lists and predicates) being part of the label. Project in the AST means **SELECT**, LJoin is **LEFT JOIN**, L_1 corresponds to "`invoice_id, T.customer_name, cnt`" on the left, etc. Each AST node is annotated with a unique id v_i. We also annotate some parts of P to illustrate this mapping.

```
L′: SELECT invoice_id, c.customer_name,
           COUNT(t.contact_name) cnt
    FROM invoices i
v′4: LEFT JOIN customers c
φ′3: ON i.user_id = c.customer_id
v′3: LEFT JOIN contacts t
φ′2: ON c.customer_id = t.user_id
v′2: LEFT JOIN customers c1
φ′1: ON t.contact_email = c1.email
v′1: GROUP BY invoice_id, c.customer_name
```

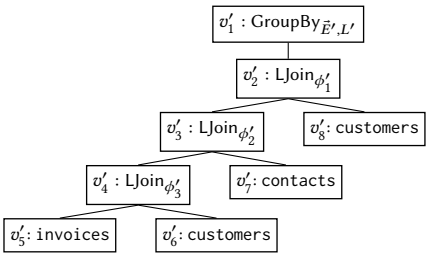


Fig. 2. SQL query P' (left) and its corresponding AST (right), following the same protocol as in Figure 1.

Here, *Customers* table contains customer information, *Contacts* stores contact information for each customer, and *Invoice* tracks price and customer information for invoices.

Let us consider the following simplified task from a LeetCode problem[3]: for each invoice, find its corresponding customer's name and the number of contacts for this customer. Figures 1 and 2 show two SQL queries (which are simplified from real-life queries submitted by LeetCode users), where P is a correct solution but P' is not. P first counts the number of contacts for each customer using a subquery (rooted at AST node v_3; see Figure 1), then joins it with the *Invoices* table, and finally selects the desired columns. On the other hand, P' first joins *Invoices*, *Customers*, and *Contacts* to find the customer name and the count of contacts for each invoice (see v'_3 in Figure 2), and then joins the *Customers* table again. P and P' yield different outputs when multiple rows in *Customers* share the same email address (which is possible): in this case, a customer's contact for an invoice would be counted multiple times in the output of P', leading to incorrect aggregation results.

Figure 3 shows a counterexample input, witnessing the non-equivalence of P and P'. Note that Alice and Bob share the same email address in *Customers*. The goal of equivalence refutation is to find an input I (like the one in Figure 3) such that $P(I) \neq P'(I)$.

Prior work. Despite the abundance of work on SQL equivalence checking, existing techniques—such as EvoSQL [7], DataFiller [18], XData [9, 10], Cosette [19, 69], and Qex [66, 69]—all fail to refute the equivalence of P and P'. VeriEQL [28] is the only tool that succeeds, though taking 132 seconds to solve our (original, unsimplified) example. The reason it takes this long is because VeriEQL *precisely* encodes the semantics of P and P' for *all inputs*. In particular, it first considers

[3]https://leetcode.com/problems/number-of-trusted-contacts-of-a-customer/description/

customer_id	customer_name	email
1	Alice	a@g.com
2	Bob	a@g.com

Customers

user_id	contact_name	contact_email
2	A	a@g.com

Contacts

invoice_id	user_id	price
3	2	10

Invoices

Fig. 3. A counterexample input database on which P and P' return different outputs.

$$M = \left\{ \begin{array}{llll} v_1 \mapsto [\mathsf{T}, \mathsf{F}] & v_2 \mapsto \big[[\mathsf{T}, \mathsf{F}], [\mathsf{F}, \mathsf{F}]\big] & v_3 \mapsto [\mathsf{T}_{\text{True}}, \mathsf{T}_{\text{True}}] & v_4 \mapsto \big[[\mathsf{F}, \mathsf{T}], [\mathsf{F}, \mathsf{F}]\big] \\ v'_1 \mapsto [\mathsf{T}_{\text{True}}, \mathsf{F}] & v'_2 \mapsto \big[[\mathsf{T}, \mathsf{T}], [\mathsf{F}, \mathsf{F}]\big] & v'_3 \mapsto \big[[\mathsf{F}, \mathsf{T}], [\mathsf{F}, \mathsf{F}]\big] & v'_4 \mapsto \big[[\mathsf{F}, \mathsf{T}], [\mathsf{F}, \mathsf{F}]\big] \end{array} \right\}$$

Fig. 4. An example UA map M, which can be used to generate the counterexample input in Figure 3 to refute the equivalence of P (in Figure 1) and P' (in Figure 2).

all input tables with at most 1 tuple: under this bound, P and P' are equivalent. Then, it bumps up the bound to 2 tuples per input table. This ends up creating a complex formula (due to various complex features, including nested joins and group-by, among others not shown in our simplified example). It takes a state-of-the-art SMT solver—in particular, z3 [21]—121 seconds to solve.

Insight 1: under-approximating queries. Our key idea is to reason about an under-approximation (UA), which is represented as a UA map M. In particular, given n queries represented as ASTs, M maps each AST node v_i to a UA that under-approximates the query operator at v_i. For instance, consider the M in Figure 4, which under-approximates P and P' from Figures 1 and 2.

Let us explain some of the entries in M. At a high level, M maps each AST node to a so-called "UA choice" (or, simply UA) that is represented by an array (potentially more than one-dimensional). For example, v_4's UA u_4—denoted by a 2×2 matrix $\big[[\mathsf{F}, \mathsf{T}], [\mathsf{F}, \mathsf{F}]\big]$—considers *Customers* tables with up to 2 tuples and *Contacts* tables with up to 2 tuples (recall from Figure 1 that v_4 is a LJoin operator). Importantly, the T value at position (1,2) constrains that only the first tuple in *Customers* and the second tuple in *Contacts* meet the join predicate ϕ_2: this additional constraint allows us to focus on a small set of input-output behaviors. The UAs for v_2, v_4, v'_2, v'_3 are defined in the same way (see Example 3.3 with a more detailed explanation), and the UAs for v_1 and v'_1 follow a similar rationale. M can be encoded into $\Psi := \Psi_1 \wedge \cdots \wedge \Psi_4 \wedge \Psi'_1 \wedge \cdots \wedge \Psi'_4$, where each Ψ_i (resp. Ψ'_i) encodes the input-output behaviors for v_i (resp. v'_i). As a whole, Ψ encodes a subset of behaviors of P and P'. While we do not show any actual SMT formulas in this example, Section 3.4 will describe how to build such formulas in detail.

Given this Ψ and application condition C (which encodes "the outputs of P and P' are distinct"), we obtain $\Phi := \Psi \wedge C$, which in this example is satisfiable. A satisfying assignment of Φ corresponds to a counterexample input that witnesses the non-equivalence of P and P' (e.g., I from Figure 3). Notably, checking the satisfiability of such Φ is typically cheap. For instance, z3 gives a model in 0.02 seconds, for the corresponding Φ in our original (unsimplified) example.

Insight 2: under-approximation search for completeness. As mentioned earlier, reasoning over a fixed UA may miss some behaviors of interest (i.e., Φ may be unsat) and therefore is incomplete. Our second insight is to construct a family of UA maps and search over this space for one that is satisfiable. This search space is defined compositionally by defining a family of UAs for each query operator. Let us take v_4 from the above Figure 4 as an example: its family contains $2^4 = 16$ UAs (i.e., each of the four elements can be either T or F), which collectively cover all inputs of interest (i.e., input tables with up to two tuples). Each of these 16 UAs can be encoded in an SMT formula whose models correspond to genuine input-output behaviors for the LJoin operator at v_4. We define UA families for the other AST nodes similarly. These operator-level UAs induce the search space of UA maps for queries. The search problem then is how to find a *satisfying UA map* M (mapping all AST nodes in P and P' to UAs), such that $\Phi := \mathit{Encode}(M) \wedge C$ is satisfiable.

$$M_1 = \left\{ \begin{array}{l} v_1 \mapsto [\star, \star] \\ v_1' \mapsto [\star, \star] \end{array} \right\} \quad M_2 = \left\{ \begin{array}{ll} v_1 \mapsto [\mathsf{T}, \mathsf{F}] & v_2 \mapsto \big[[\star, \star], [\star, \star]\big] \\ v_1' \mapsto [\mathsf{F}, \mathsf{F}] & v_2' \mapsto \big[[\star, \star], [\star, \star]\big] \end{array} \right\}$$

$$M_3 = \left\{ \begin{array}{llll} v_1 \mapsto [\mathsf{T}, \mathsf{F}] & v_2 \mapsto \big[[\mathsf{F}, \mathsf{F}], [\mathsf{F}, \mathsf{T}]\big] & v_3 \mapsto [\star, \star] & v_4 \mapsto \big[[\star, \star], [\star, \star]\big] \\ v_1' \mapsto [\mathsf{F}, \mathsf{F}] & v_2' \mapsto \big[[\mathsf{F}, \mathsf{F}], [\mathsf{F}, \mathsf{F}]\big] & v_3' \mapsto \big[[\star, \star], [\star, \star]\big] & v_4' \mapsto \big[[\star, \star], [\star, \star]\big] \end{array} \right\}$$

Fig. 5. An example sequence of UA maps $M_1 \to M_2 \to M_3 \to M$ that our algorithm may explore, to refute the equivalence of P and P'. Here, M is the satisfying UA map in Figure 4.

Our search algorithm explores a sequence of UA maps M_i's until reaching a satisfying one. In general, M_i is partial (i.e., containing a subset of AST nodes). Given M_i at each step, we produce the next M_{i+1} by either adding more nodes or by adjusting UAs of existing nodes. Figure 5 shows one such sequence $M_1 \to M_2 \to M_3 \to M$, where M is the satisfying UA map from Figure 4.

While each M_i is structurally similar to M, they include a special "top" value $\star$, which intuitively means "I don't know." For instance, v_4's UA in M_3 considers all *Customers* and *Contacts* tables both with up to 2 tuples, with no additional constraints. Intuitively, this UA "subsumes" the UA for v_4 in M, because the latter considers a specific way of joining the two input tables. Allowing $\star$ essentially creates a lattice of UAs, which we exploit to perform efficient under-approximation search. Let us explain how this works in detail, still using Figure 5 as an example.

The algorithm begins with M_1, obtains its encoding Ψ_1, and confirms $\Phi_1 := \Psi_1 \wedge C$ is satisfiable. In this case, we map v_1 and v_1' to UAs derived from a model of Φ_1, and add 2 new nodes v_2, v_2' (both mapped to $\star$)—this yields M_2. Φ_2 for M_2 is also sat; therefore, we update M_2 in the same way (except adding 4 nodes this time) and obtain M_3.

Φ_3 for M_3, however, is unsat. In this case, we first obtain a subset V (namely, $\{v_1', v_2', v_3', v_4', v_1, v_2\}$) of nodes, whose UAs in M_3 are in *conflict*; that is, $\mathit{Encode}(M_3 \downarrow V) \wedge C$ is unsat ($M_3 \downarrow V$ gives the "sub-map" of M_3 for nodes in V). Then, we aim to build a new UA map M_4 which (i) maps nodes in V to new UAs and (ii) maps nodes outside V to UAs, such that (a) $\mathit{Encode}(M_4 \downarrow V) \wedge C$ is sat and (b) M_4 does not contain any previously discovered conflicts. In this example, we have $M_4 = M$, which is a satisfying UA map. Note that step (i) requires searching, among all combinations of UAs for nodes in V, for one that meets (a). To do this efficiently, we take advantage of the aforementioned lattice structure of UAs to partition this large space into subspaces shown below in Figure 6, which enables efficient search over this combinatorial space.

(1) $v_1 \mapsto [\star, \star], v_1' \mapsto [\star, \star], v_2 \mapsto \big[[\star, \star], [\star, \star]\big], v_2' \mapsto \big[[\star, \star], [\star, \star]\big], v_3' \mapsto \big[[\mathsf{T}, \mathsf{T}], [\star, \star]\big], v_4' \mapsto \big[[\mathsf{T}, \mathsf{T}], [\star, \star]\big]$
(2) $v_1 \mapsto [\star, \star], v_1' \mapsto [\star, \star], v_2 \mapsto \big[[\star, \star], [\star, \star]\big], v_2' \mapsto \big[[\star, \star], [\star, \star]\big], v_3' \mapsto \big[[\mathsf{F}, \mathsf{T}], [\star, \star]\big], v_4' \mapsto \big[[\mathsf{T}, \mathsf{T}], [\star, \star]\big]$
(3) $v_1 \mapsto [\star, \star], v_1' \mapsto [\star, \star], v_2 \mapsto \big[[\star, \star], [\star, \star]\big], v_2' \mapsto \big[[\star, \star], [\star, \star]\big], v_3' \mapsto \big[[\mathsf{T}, \mathsf{T}], [\star, \star]\big], v_4' \mapsto \big[[\mathsf{F}, \mathsf{T}], [\star, \star]\big]$
(4) $v_1 \mapsto [\star, \star], v_1' \mapsto [\star, \star], v_2 \mapsto \big[[\star, \star], [\star, \star]\big], v_2' \mapsto \big[[\star, \star], [\star, \star]\big], v_3' \mapsto \big[[\mathsf{F}, \mathsf{T}], [\star, \star]\big], v_4' \mapsto \big[[\mathsf{F}, \mathsf{T}], [\star, \star]\big]$
...
(16) $v_1 \mapsto [\star, \star], v_1' \mapsto [\star, \star], v_2 \mapsto \big[[\star, \star], [\star, \star]\big], v_2' \mapsto \big[[\star, \star], [\star, \star]\big], v_3' \mapsto \big[[\mathsf{F}, \mathsf{F}], [\star, \star]\big], v_4' \mapsto \big[[\mathsf{F}, \mathsf{F}], [\star, \star]\big]$

Fig. 6. An example partition of the UA space for AST nodes $v_1', v_2', v_3', v_4', v_1, v_2$ from P and P'.

Note that each subspace is a UA map over nodes in V. Our algorithm processes them one by one. It first encodes the UA map $M_V^{(1)}$ in (1) from Figure 6—that is, $\mathit{Encode}\big(M_V^{(1)}\big) \wedge C$—and finds it to be unsat. In other words, this $M_V^{(1)}$ is still a conflict. Similarly, $M_V^{(2)}$ and $M_V^{(3)}$ are also unsat. On the other hand, $M_V^{(4)}$ is sat. We therefore obtain a model for $M_V^{(4)}$, from which we can generate M_4, which is identical to M. This concludes the search process. In total, it takes 0.3 seconds for Polygon to solve the original (unsimplified) example.

$$
\begin{array}{rrcl}
\textit{Query} & P & ::= & R \mid \text{Project}_L(P) \mid \text{Filter}_\phi(P) \mid \text{Rename}_R(P) \mid \text{UnionAll}(P_1, P_2) \mid \otimes\,(P_1, P_2) \\
 & & \mid & \text{Distinct}(P) \mid \text{GroupBy}_{\vec{E},L,\phi}(P) \mid \text{OrderBy}_E(P) \mid \text{With}(\vec{P}, \vec{R}, P) \\
\textit{Join Op} & \otimes & ::= & \text{Product} \mid \text{IJoin}_\phi \mid \text{LJoin}_\phi \mid \text{RJoin}_\phi \mid \text{FJoin}_\phi \\
\textit{Attr List} & L & ::= & [E, \cdots, E] \\
\textit{Pred} & \phi & ::= & b \mid \text{Null} \mid E \odot E \mid \vec{E} \in \vec{v} \mid \vec{E} \in Q \mid \text{IsNull}(E) \mid \text{Exists}(P) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\
\textit{Expr} & E & ::= & a \mid v \mid E \diamond E \mid \mathcal{G}(b, E) \mid \text{ITE}(\phi, E, E) \mid \text{Case}(\vec{\phi}, \vec{E}, E) \mid \text{Rename}_a(E) \\
\textit{Arith Op} & \diamond & ::= & + \mid - \mid \times \mid / \mid \% \\
\textit{Logic Op} & \odot & ::= & \leq \mid < \mid = \mid \neq \mid > \mid \geq
\end{array}
$$

$$R \in \textbf{Relation Names} \quad a \in \textbf{Attribute Names} \quad v \in \textbf{Values} \quad b \in \textbf{Bools}$$

$$\mathcal{G} \in \{\text{Count}, \text{Min}, \text{Max}, \text{Sum}, \text{Avg}\}$$

Fig. 7. Syntax of our query language.

3 Conflict-Driven Under-Approximation Search for SQL

We begin with the syntax of our query language (Section 3.1). Then, we define under-approximations for this language and formalize its under-approximate semantics (Sections 3.2-3.4). Finally, we give our under-approximation search algorithm (Sections 3.5-3.8).

3.1 Query Language

Syntax. Figure 7 shows the syntax of our language, which is a highly expressive subset of SQL. A query P can be a relation R, a projection $\text{Project}_L(P)$ that selects columns L from P's result, a filter $\text{Filter}_\phi(P)$ that retains from P the rows satisfying predicate ϕ, a renaming operator $\text{Rename}_R(P)$ that renames the output of P to relation R. It also supports bag union UnionAll and different types of joins (including Cartesian product Product, inner join IJoin_ϕ, left join LJoin_ϕ, right join RJoin_ϕ, and full outer join FJoin_ϕ). $\text{Distinct}(P)$ removes duplicate rows. $\text{GroupBy}_{\vec{E},L,\phi}(P)$ first groups all rows from P based on $\vec{E}$, computes expressions L for each group, and returns the result for groups satisfying ϕ. $\text{OrderBy}_E(P)$ sorts (in ascending order) P's output by expression list E. $\text{With}(\vec{P}, \vec{R}, P)$ creates intermediate relations $\vec{R}$ with the result of queries $\vec{P}$, and returns the result of P (which potentially uses $\vec{R}$). It is worth noting that we support if-then-else $\text{ITE}(\phi, E, E)$, case-when $\text{Case}(\vec{\phi}, \vec{E}, E)$, and predicates like $\vec{E} \in P$ and $\text{Exists}(P)$.

Semantics. Our semantics is based on prior work [28], which will be formalized in Section 3.3.

3.2 Representing Under-Approximations

This section presents a method to define a family of under-approximations for each *Query* operator F from Figure 7. In a nutshell, our under-approximation (UA) describes some subset of F's *reachable* outputs (i.e., outputs that can indeed be produced by F for some input). This notion is consistent with the under-approximation idea from O'Hearn's seminal paper on incorrectness logic [49]. Our work, however, realizes UAs using a two-step approach: (i) first describe a subset of inputs for F, (ii) then define their corresponding input-output behaviors (per F's semantics). Here, (i) is simply a "pointer" that refers to a subset of inputs, whereas (ii) is the actual encoding of the UA. This section focuses on step (i), while Sections 3.3-3.4 will explain step (ii).

In this work, we call a subset of inputs in step (i) "a UA choice"—or, with a slight abuse of notation, simply "a UA"—which is denoted by u. More specifically, u for each F is always represented by an array (potentially more than one-dimensional) of values. The interpretation of u is specific to F, which we will explain below. As mentioned earlier, u is just used to refer to a subset of inputs.

Filter. The UA u for Filter_ϕ is always of the form $\{\mathsf{T}, \mathsf{F}, \star\}^n$. That is, u is always a vector of n values, where each value is T, F or $\star$. Here, n is the maximum input table size; that is, we consider input tables R with at most n tuples. If the ith value u_i is T, it means the ith tuple t_i in R satisfies predicate ϕ. On the other hand, $u_i = \mathsf{F}$ means either t_i is not present[4] in R, or t_i does not satisfy ϕ. Finally, $\star$ is the "top" value (i.e., "I don't know"), meaning t_i can be either T or F. As we can see, any table (of size up to n) can be covered by some u, no matter how its tuples satisfy the filtering condition ϕ.

Example 3.1. Let us consider $\text{Filter}_{\texttt{id>3}}$, and a UA choice $u = [\mathsf{T}, \mathsf{F}]$ for it. Here, u refers to those (input) relations R whose *first* tuple has `id` greater than 3—this is what the first value $u_1 = \mathsf{T}$ in u means. The second value $u_2 = \mathsf{F}$ means that the second tuple in R is either not present (meaning R has exactly one row), or its `id` is *not* greater than 3 (in this case, R has exactly two rows). Note that R has at most 2 tuples, since u has length 2. As another example, let us consider $u' = [\mathsf{T}, \star]$ for the same filter operator. Different from u_2, u'_2 puts no restrictions on the second tuple. So u' refers to all tables with up to two tuples, whose first tuple must satisfy predicate $\texttt{id} > 3$. In other words, u' "subsumes" u, or u "refines" u'.

Projection and UnionAll. The UAs for projection are defined in the same way as filter, but their interpretation is slightly different. In particular, u_i being T or F indicates whether or not t_i from the input table is present. On the other hand, $\star$ is still the top value that means either T or F. The UA for UnionAll is essentially the same as for projection, except that it is a vector of $n_1 + n_2$ values, where the first n_1 (resp. the last n_2) values correspond to tuples from the first (resp. second) table.

Joins. Now, let us consider the join operators. Take the inner join IJoin_ϕ as an example. Its UA u is an $n_1 \times n_2$ matrix, where $u_{i,j}$ is T, F or $\star$. Here, n_1 (resp. n_2) is the maximum size of the first (resp. second) table. For each $u_{i,j}$, T means the ith tuple t_i from the first table and the jth tuple t_j from the second are both present and satisfy the join condition ϕ; whereas F means at least one of t_i, t_j is deleted, or they are both present but cannot be joined. The UA definitions for the other joins are quite similar; we refer readers to the extended version [72] of this paper for more details.

GroupBy and Distinct. The UA definition for $\text{GroupBy}_{\vec{E},L,\phi}$ is different from all operators above. Recall that GroupBy first groups all input tuples based on $\vec{E}$, then evaluates L for each group, and finally returns an output table with groups satisfying predicate ϕ. A UA choice u for GroupBy is always of the form $\{\mathsf{T}, \mathsf{T}_\phi, \mathsf{F}, \star\}^n$. Similar to before, u_i corresponds to the ith tuple t_i in the input. The interpretation, however, is different. If $u_i = \mathsf{T}$ or T_ϕ, it means t_i is present and distinct from all tuples $t_j (j < i)$ before it, with respect to columns in $\vec{E}$. In other words, t_i will form a new group. The difference between T and T_ϕ is that T_ϕ means this new group further satisfies ϕ, whereas T means it does not. On the other hand, F means t_i will not lead to a new group (either it is deleted, or it belongs to an existing group). Finally, $\star$ is still our top value, which means $\mathsf{T}, \mathsf{T}_\phi$, or F. The UA u for Distinct is a vector of n values chosen from $\{\mathsf{T}, \mathsf{F}, \star\}$, where $u_i = \mathsf{T}$ means t_i is distinct from all prior tuples $t_j (j < i)$, and F means there exists some $j < i$ such that $t_i = t_j$ (i.e., t_i is a duplicate).

Example 3.2. Consider $\text{GroupBy}_{\vec{E},L_2}$ at node v_3 from Figure 1(b), where L_2 is shown in Figure 1(a) and $\vec{E} = [\texttt{A.customer\_id}, \texttt{customer\_name}]$. Note that the having predicate ϕ here is True (so our notation omitted it). Consider UA $u = [\mathsf{T}_\phi, \mathsf{F}]$, which refers to tables with at most two tuples t_1, t_2. It further states: (i) t_1 exists and forms a new group, and this group satisfies ϕ (which is always true as we have $\phi =$ True in this example); (ii) t_2 either does not exist, or $\vec{E}$ evaluates on t_2 to the same result as a previous tuple (in this example, t_1).

[4]The formal encoding of a table will be described in Section 3.3. In brief, while a table always has exactly n tuples, we allow some of the tuples to be deleted (i.e., not present). This allows us to encode all tables with *at most* (not just exactly) n tuples.

Lattice of UAs. As we can see, UAs for each operator form a (semi-)lattice, due to the top value $\star$. In particular, given two UAs u and u' for the same operator, we define $u \sqsupseteq u'$ if we have $u_i \sqsupseteq u'_i$ for all i. That is, u subsumes u' (or, u' refines u) if the ith value in u subsumes the ith value in u'. The value subsumption relation is straightforward: the top value $\star$ subsumes any non-top value, and non-top values do not subsume each other. A UA whose values are all $\star$ is called a "top UA", while we call a UA with only non-top values a "minimal UA". We use $\mathcal{U}_F$ to denote the set of UAs for F.

Other query operators. We refer readers to the extended version [72] for OrderBy's UA definition. For operators R, Rename, and With, we have one UA that always considers *all* inputs up to a given size (i.e., effectively no under-approximation).

3.3 Encoding Full Semantics

This section formalizes the *full* semantics of query operators from Figure 7, which Section 3.4 will use to build *under-approximate* semantics. Our formalization is based on prior work [28]: a table is encoded symbolically using n symbolic tuples, a tuple can be deleted (this allows us to consider all tables *up to* size n), and operator semantics is encoded in SMT following SQL standard. The *key distinction* from prior work is that we embed the UA choices into the semantics encoding—we will further elaborate on this as we go over the encoding rules below. We note that, our "full semantics" is still bounded, in that it considers tables up to a finite size bound. We use "full" just to differentiate it from under-approximate semantics (which will be presented in Section 3.4).

Figure 8 gives the semantics encoding rules for a representative subset of query operators. Please find the full set of rules in our extended version [72]. Here, *EncFullSemantics* generates an SMT formula Ψ that encodes a given operator F's semantics for all input tables up to a certain bound.[5] Ψ is always over free variables x (denoting an input table) and y (denoting F's output table). Any satisfying assignment of Ψ corresponds to a genuine input-output behavior of F; that is, all outputs encoded by Ψ are reachable. Ψ has another free variable z, called "UA variable", which denotes the UA choice—as we will see below, this UA variable z plays a central role in our encoding.

Filter. Rule (1) encodes Filter_ϕ's reachable outputs for all input tables with up to n tuples. Each t_i is a symbolic tuple from the input table x, and t'_i is the corresponding symbolic tuple in the output table y. We use an uninterpreted predicate Del to denote if a tuple is deleted. Some t_i's may be deleted, so x may have fewer than n tuples. For a non-deleted t_i that does not meet the filtering condition ϕ, we use $\text{Del}(t'_i)$ to mean the corresponding t'_i is deleted in y.

Our key novelty is to embed UA variable z into the semantics encoding. For Filter_ϕ, z is a vector of variables, where each z_i denotes the ith value u_i in the UA. Recall from Section 3.2 that $u_i = \mathsf{T}$ means t_i is present and satisfies ϕ. For such t_i, $\Psi_{i,\mathsf{T}}$ encodes the corresponding t'_i. $\Psi_{i,\mathsf{F}}$ handles the F case. The final formula Ψ is built *compositionally*, by conjoining $\Psi_{i,\mathsf{T}} \wedge \Psi_{i,\mathsf{F}}$ across all i's from 1 to n. As we will show shortly, all of our semantics encoding rules are compositional. To simplify our presentation, Rule (1) uses some auxiliary functions. $\text{Exist}(t_i, t'_i, \phi) := \left(\neg\text{Del}(t_i) \wedge [\![\phi]\!]_{t_i}\right) \wedge \neg\text{Del}(t'_i)$ encodes the case where input tuple t_i is present and satisfies ϕ, and (therefore) the output tuple t'_i exists. In this case, we also have $\text{Copy}(t_i, t'_i, L) := \bigwedge_{a \in L} [\![t'_i.a]\!] = [\![t_i.a]\!]$, which creates the content of t'_i by copying values of attributes in L[6] from t_i to t'_i. $\Psi_{i,\mathsf{F}}$ encodes the situation where t'_i is deleted using $\text{NotExist}(t_i, t'_i, \phi) := \left(\text{Del}(t_i) \vee \left(\neg\text{Del}(t_i) \wedge \neg[\![\phi]\!]_{t_i}\right)\right) \wedge \text{Del}(t'_i)$. Here, the corresponding t_i is either not present or does not meet ϕ. Finally, we note that z_i only considers non-top values—i.e., T and F for filter—because considering $\star$ is not necessary when encoding the full semantics.

[5] Note that the actual values of F's non-table arguments (such as predicates and expressions) are all given.

[6] We assume attributes are given, to simplify our presentation. In general, they can be easily inferred from the input schema.

$$(1)\ \frac{\begin{array}{l} x = [t_1, \cdots, t_n] \quad y = [t'_1, \cdots, t'_n] \quad z = [z_1, \cdots, z_n] \quad L = \mathrm{Attrs}(x) \\ \Psi_{i,\mathsf{T}} = (z_i = \mathsf{T}) \rightarrow \left(\mathrm{Exist}(t_i, t'_i, [\![\phi]\!]_{t'_i}) \wedge \mathrm{Copy}(t_i, t'_i, L)\right) \quad \Psi_{i,\mathsf{F}} = (z_i = \mathsf{F}) \rightarrow \mathrm{NotExist}(t_i, t'_i, [\![\phi]\!]_{t'_i}) \end{array}}{\mathit{EncFullSemantics}\left(\mathrm{Filter}_\phi\right) \twoheadrightarrow \bigwedge_{i=1,\cdots,n} \Psi_{i,\mathsf{T}} \wedge \Psi_{i,\mathsf{F}}}$$

$$(2)\ \frac{\begin{array}{l} x = [t_1, \cdots, t_n] \quad y = [t'_1, \cdots, t'_n] \quad z = [z_1, \cdots, z_n] \\ \Psi_{i,\mathsf{T}} = (z_i = \mathsf{T}) \rightarrow \left(\mathrm{Exist}(t_i, t'_i, \mathrm{True}) \wedge \mathrm{Copy}(t_i, t'_i, L)\right) \quad \Psi_{i,\mathsf{F}} = (z_i = \mathsf{F}) \rightarrow \mathrm{NotExist}(t_i, t'_i, \mathrm{True}) \end{array}}{\mathit{EncFullSemantics}\left(\mathrm{Project}_L\right) \twoheadrightarrow \bigwedge_{i=1,\cdots,n} \Psi_{i,\mathsf{T}} \wedge \Psi_{i,\mathsf{F}}}$$

$$(3)\ \frac{\begin{array}{l} x_1 = [t_1, \cdots, t_{n_1}]\ \ x_2 = [t'_1, \cdots, t'_{n_2}]\ \ y = [t''_{1,1}, \cdots, t''_{n_1,n_2}]\ \ z = \left[[z_{1,1}, \cdots z_{1,n_2}], \cdots, [z_{n_1,1}, \cdots, z_{n_1,n_2}]\right]\ \ L_i = \mathrm{Attrs}(x_i) \\ \Psi_{i,j,\mathsf{T}} = (z_{i,j} = \mathsf{T}) \rightarrow \mathrm{Exist}\Big((t_i, t'_j), t''_{i,j}, [\![\phi]\!]_{t_i,t'_j}\Big) \wedge \mathrm{Copy}(t_i, t''_{i,j}, L_1) \wedge \mathrm{Copy}(t'_j, t''_{i,j}, L_2) \\ \Psi_{i,j,\mathsf{F}} = (z_{i,j} = \mathsf{F}) \rightarrow \mathrm{NotExist}\Big((t_i, t'_j), t''_{i,j}, [\![\phi]\!]_{t_i,t'_j}\Big) \end{array}}{\mathit{EncFullSemantics}\left(\mathrm{IJoin}_\phi\right) \twoheadrightarrow \bigwedge_{i=1,\cdots,n_1} \bigwedge_{j=1,\cdots,n_2} \Psi_{i,j,\mathsf{T}} \wedge \Psi_{i,j,\mathsf{F}}}$$

$$(4)\ \frac{\begin{array}{l} x = [t_1, \cdots, t_n] \quad y = [t'_1, \cdots, t'_n] \quad z = [z_1, \cdots, z_n] \\ \Psi_{i,\mathsf{F}} = (z_i = \mathsf{F}) \rightarrow \Big(\left(\mathrm{Del}(t_i) \vee \left(\neg\mathrm{Del}(t_i) \wedge \bigvee_{j=1,\cdots,i-1} \left(\neg\mathrm{Del}(t_j) \wedge \bigwedge_{a \in \vec{E}} [\![t_i.a]\!] = [\![t_j.a]\!] \wedge \mathrm{g}(t_i) = j\right)\right)\right) \wedge \mathrm{Del}(t'_i)\Big) \\ \Psi_{i,\neg\mathsf{F}} = \neg\mathrm{Del}(t_i) \wedge \neg \bigvee_{j=1,\cdots,i-1} \Big(\neg\mathrm{Del}(t_j) \wedge \bigwedge_{a \in \vec{E}} [\![t_i.a]\!] = [\![t_j.a]\!]\Big) \wedge \mathrm{g}(t_i) = i \\ \Psi_{i,\mathsf{T}} = (z_i = \mathsf{T}) \rightarrow \Big(\Psi_{i,\neg\mathsf{F}} \wedge \neg[\![\phi]\!]_{\mathrm{g}^{-1}(i)} \wedge \mathrm{Del}(t'_i)\Big) \\ \Psi_{i,\mathsf{T}_\phi} = (z_i = \mathsf{T}_\phi) \rightarrow \Big(\Psi_{i,\neg\mathsf{F}} \wedge [\![\phi]\!]_{\mathrm{g}^{-1}(i)} \wedge \neg\mathrm{Del}(t'_i) \wedge \mathrm{Copy}(\mathrm{g}^{-1}(i), t'_i, L)\Big) \end{array}}{\mathit{EncFullSemantics}\left(\mathrm{GroupBy}_{\vec{E},L,\phi}\right) \twoheadrightarrow \bigwedge_{i=1,\cdots,n} \Psi_{i,\mathsf{T}} \wedge \Psi_{i,\mathsf{T}_\phi} \wedge \Psi_{i,\mathsf{F}}}$$

Fig. 8. Full semantics for a representative subset of query operators from Figure 7.

Projection. Rule (2) is almost the same as Rule (1), except that the attribute list L is given and the "filtering condition" is always True. Note that aggregate functions are always precisely encoded (i.e., no UAs) as also shown in the rule, although under-approximation happens when encoding the semantics of projection. This is also the case for other operators that use aggregate functions.

Inner join. Rule (3) follows the same principle, but is slightly more involved. First, each tuple $t''_{i,j}$ in y corresponds to the join result of t_i from x_1 and t'_j from x_2. Variable $z_{i,j}$ denotes UA value $u_{i,j}$. $\Psi_{i,j,\mathsf{T}}$ encodes the input-output behavior when $z_{i,j} = \mathsf{T}$. Here, we first assert $t''_{i,j}$ exists in the output, and then use Copy to construct the content in $t''_{i,j}$. Note that Exist is extended here to accept a pair of input tuples: $\mathrm{Exist}\big((t_i, t'_j), t''_{i,j}, \phi\big) := \big(\neg\mathrm{Del}(t_i) \wedge \neg\mathrm{Del}(t'_j) \wedge [\![\phi]\!]_{t_i,t_j}\big) \wedge \neg\mathrm{Del}(t''_{i,j})$. $\Psi_{i,j,\mathsf{F}}$ encodes the F case, with an extended NotExist function.

$$\mathrm{NotExist}\big((t_i, t'_j), t''_{i,j}, \phi\big) := \Big(\mathrm{Del}(t_i) \vee \mathrm{Del}(t'_j) \vee \big(\neg\mathrm{Del}(t_i) \wedge \neg\mathrm{Del}(t'_j) \wedge \neg[\![\phi]\!]_{t''_{i,j}}\big)\Big) \wedge \mathrm{Del}(t''_{i,j})$$

GroupBy. Rule (4) also constructs the final encoding compositionally, by conjoining the formulas for $\mathsf{T}, \mathsf{T}_\phi$ and F across all i's from 1 to n. Recall from Section 3.2 that $u_i = \mathsf{F}$ means t_i does not form a new group. Therefore, $\Psi_{i,\mathsf{F}}$ encodes the constraints for t_i and asserts t'_i is not present. There are two possibilities: t_i is deleted, or there exists some $t_j (j < i)$ in the same group as t_i. Here, g is an uninterpreted function to record a tuple's belonging group, which will be useful later. $\Psi_{i,\mathsf{T}}$ and Ψ_{i,T_ϕ} both consider the case where t_i forms a new group; so they share $\Psi_{i,\neg\mathsf{F}}$ as a common piece, which simply says t_i is present and there is no prior tuple t_j that belongs to the same group. $\Psi_{i,\mathsf{T}}$ further states that those tuples in t_i's group do not satisfy ϕ and therefore we have t'_i deleted. On the other hand, Ψ_{i,T_ϕ} encodes the case where ϕ is met and output tuple t'_i is present. Here, Copy creates the content in t'_i, based on a set $\vec{t}$ of input tuples belonging to a group (given by g^{-1}).

$$\mathrm{Copy}(\vec{t}, t'_i, L) := \bigwedge_{a \in L} [\![t'_i.a]\!] = [\![a]\!]_{\vec{t}}$$

3.4 Encoding Under-Approximate Semantics

To under-approximate a query, we first under-approximate its operators.

Encoding UA semantics for query operators. Given a UA choice $u \in \mathcal{U}_F$ for a query operator F, we encode the UA semantics of F against u as follows.

$$\mathit{EncUASemantics}(F, u) := \mathit{EncUAChoice}(u) \wedge \mathit{EncFullSemantics}(F)$$

EncFullSemantics is described in Figure 8, and *EncUAChoice* is defined below.

$$\begin{aligned} \mathit{EncUAChoice}(u) &:= \textstyle\bigwedge_{u_i \in u} \mathit{EncUAChoiceValue}(u_i) \\ \mathit{EncUAChoiceValue}(u_i) &:= \begin{cases} z_i = u_i & \text{if } u_i \neq \star \\ \bigvee_{c \in \mathrm{dom}(u_i) \setminus \{\star\}} z_i = c & \text{if } u_i = \star \end{cases} \end{aligned}$$

Here, $\mathrm{dom}(u_i)$ gives all possible values for u_i—including top $\star$—but *EncUAChoiceValue* removes $\star$; hence the final encoding considers only minimal UAs. While *EncFullSemantics* constructs a fixed formula for F that encodes reachable outputs for *all* inputs (up to a bound), *EncUASemantics* only encodes reachable outputs for the *subset* of inputs specified by u. This (further) under-approximates F, and enables efficient symbolic reasoning.

Encoding UA semantics for queries. We can further encode the UA semantics for query P, given a *UA map* M that maps each of P's AST node v to a UA choice $u \in \mathcal{U}_v$, as follows.

$$\mathit{EncUASemantics}(M) := \bigwedge_{\substack{(v \mapsto u) \in M \\ \mathit{children}(v) = [v_1, \cdots, v_l]}} \Big(\mathit{EncUASemantics}(v, u) \wedge (x_1^v = y^{v_1} \wedge \cdots \wedge x_l^v = y^{v_l}) \Big)_{lv}$$

Here, v is an AST node in P with l children, and $\mathcal{U}_v = \mathcal{U}_{op(v)}$. $\mathit{EncUASemantics}(v, u)$ is defined as:

$$\mathit{EncUASemantics}(v, u) := \begin{cases} \Big(\mathit{EncFullSemantics}\big(op(v)\big)\Big)\big[x_1 \mapsto x_1^R, \cdots, x_l \mapsto x_l^R, y \mapsto y^v\big] & \text{if } op(v) = R \\ \Big(\mathit{EncFullSemantics}\big(op(v)\big)\Big)\big[x_1 \mapsto x_1^v, \cdots, x_l \mapsto x_l^v, y \mapsto y^v\big] & \text{if } op(v) = \text{With or Rename} \\ \Big(\mathit{EncUAChoice}(u) \wedge \mathit{EncFullSemantics}\big(op(v)\big)\Big)\big[x_1 \mapsto x_1^v, \cdots, x_l \mapsto x_l^v, y \mapsto y^v, z \mapsto z^v\big] & \text{otherwise} \end{cases}$$

which encodes the UA semantics for query operator $op(v)$ at v against u. We highlight the "otherwise" case, where we rename each variable x_i to x_i^v (that denotes the ith input to v), and similarly y to y_i^v (which denotes v's output), as well as z to z^v (which denotes the UA choice at v). Operators in the other cases always have one UA, so we do not need to encode it. For operator R, we use x^R (instead of x^v), since multiple AST nodes may be labeled with the same relation. $\mathit{EncUASemantics}(M)$ conjoins encodings across all $v \mapsto u$ entries in M. Each clause is labeled with lv, which (as we will see in later sections) is used for conflict extraction. As a final minor note, *EncUASemantics* assumes access to P's schema, which allows *EncFullSemantics* to obtain attributes of P's intermediate tables.

$\mathit{EncUASemantics}(M)$ can be generalized to encode UA semantics for multiple queries $P_1, \cdots, P_n$, by extending M to include all $v \mapsto u$ entries across all P_i's.

Example 3.3. Consider the UA map M from Figure 4. Let us briefly explain some of its entries. Consider $v_2 \mapsto \big[[\mathsf{T}, \mathsf{F}], [\mathsf{F}, \mathsf{F}]\big]$, where v_2 is a left join operator with two input tables $x_1^{v_2}$ (i.e., *Invoices*) and $x_2^{v_2}$ (i.e., output of v_3). $M(v_2)$ states: $x_1^{v_2}$ and $x_2^{v_2}$ have up to 2 tuples; the first tuple in $x_1^{v_2}$ and the first tuple in $x_2^{v_2}$ meets the join condition; the other 3 tuple pairs do not join. Consider $v_1 \mapsto [\mathsf{T}, \mathsf{F}]$, where v_1 is a projection. $M(v_1)$ states that only the first tuple in x^{v_1} exists. In other words, although y^{v_2} has up to 2 tuples (according to $M(v_2)$), $M(v_1)$ considers only those tables of size one[7]. Consider

[7] Polygon sets a node's UA size heuristically, ranging from 2 to 16 and 2x2 to 16x16 for unary and binary operators.

Algorithm 1 Top-level algorithm.

procedure GenInput($P_1, \cdots, P_n, C$)
input: Each P_i is a query. C is an application condition (expressed as an SMT formula).
output: A database I that satisfies C, or *null* indicating no such I is found.
1: M := ConflictDrivenUASearch($P_1, \cdots, P_n, C$);
2: **if** M = *null* **then return** *null*;
3: **return** *ExtractInputDB*(M, C);

$v_3 \mapsto [\mathsf{T}_{\text{True}}, \mathsf{T}_{\text{True}}]$, where v_3 is GroupBy. $M(v_3)$ describes tables x^{v_3} with 2 tuples, where each tuple leads to a new group. $v_1' \mapsto [\mathsf{T}_{\text{True}}, \mathsf{F}]$ also concerns GroupBy but uses a different UA. It considers $x^{v_1'}$ with up to 2 tuples, where the first tuple forms a new group but the second does not.

3.5 Top-Level Algorithm and Problem Statement

Let us switch gears and present our under-approximation search algorithm. Algorithm 1 shows our top-level algorithm. Given $P_1, \cdots, P_n$ and an SMT formula C (encoding the application condition) over variables $y_1, \cdots, y_n$ (each y_i denoting P_i's output), GenInput returns a *satisfying input* I such that $C[y_1 \mapsto P_1(I), \cdots, y_n \mapsto P_n(I)]$ is true, or returns *null* if no such I is found.

Our key novelty lies in ConflictDrivenUASearch (line 1), which searches for a *satisfying* UA map M. Given such an M, we invoke *ExtractInputDB* (line 3) to derive a satisfying input. Below, we first formulate the under-approximation search problem, and then explain *ExtractInputDB*.

Definition 3.4. **(Under-Approximation Search Problem).** Given n queries $P_1, \cdots, P_n$ from the language in Figure 7, given $\mathcal{U}_F$ that defines a family of UAs for each query operator F, and given an SMT formula C over variables $y_1, \cdots, y_n$, find a UA map M which maps *each* AST node v in P_i (for all $i \in [1, n]$) to a *minimal* UA $u \in \mathcal{U}_{op(v)}$ such that

$$\mathit{EncUASemantics}(M) \wedge \left(C[y_1 \mapsto y^{v_1}, \cdots, y_n \mapsto y^{v_n}]\right)_{lc} \tag{1}$$

is satisfiable. Here, v_i is P_i's root AST node (and recall that we use variable y^v to denote v's output), and lc is simply a label for the application condition (which will later be used for conflict extraction). We call such M a *satisfying under-approximation map*, or *satisfying UA map* for short.

In fact, given any M that maps AST nodes to UAs (which can be non-minimal), we can check the satisfiability of the above formula (1): if it is satisfiable, we can obtain a satisfying UA map from M, by refining all non-minimal UAs in M to minimal ones with the help of a satisfying assignment of (1). The next section will explain how this works in detail.

Given a satisfying UA map M, *ExtractInputDB* first obtains a model σ for the formula in (1). Then, $\sigma(x^R)$ gives the content for each input relation R.

Example 3.5. Figure 4 shows a satisfying UA map M for the equivalence refutation example from Section 2. Given this M, *ExtractInputDB* can generate the database in Figure 3.

3.6 Conflict-Driven Under-Approximation Search

Now, let us unpack the ConflictDrivenUASearch procedure—see Algorithm 2. At a high level, it iteratively generates a sequence of *candidate* UA maps: M is initially empty (line 1), and is iteratively updated in two ways depending on if it satisfies Φ at line 3. This Φ is essentially the same condition as in (1), but we omit the variable renaming part and the label for C, to simplify the presentation. Note that before termination, M is *partial*; that is, M does not contain all AST nodes from all P_i's.

Algorithm 2 Algorithm for ConflictDrivenUASearch.

procedure ConflictDrivenUASearch($P_1, \cdots, P_n, C$)
input: Each P_i is a query. C is an application condition.
output: A satisfying UA map M, or *null* indicating no such M is found.
1: $M := \emptyset$; $\Omega := \emptyset$; $W := \bigcup_{i\in[1,n]} ASTNodes(P_i)$;
2: **while** true **do**
3: $\Phi := EncUASemantics(M) \wedge C$;
4: **if** Φ is satisfiable **then**
5: $\sigma := GetModel(\Phi)$; $M := M\big[v \mapsto \sigma(z^v) \mid v \in \mathrm{dom}(M)\big]$;
6: **if** $W = \emptyset$ **then return** M;
7: $(v_1, \cdots, v_k) := W.remove()$; $M := M[v_1 \mapsto GetTopUA(\mathcal{U}_{v_1}), \cdots, v_k \mapsto GetTopUA(\mathcal{U}_{v_k})]$;
8: **else**
9: $V := ExtractConflict(\Phi)$; $(M, \Omega) :=$ ResolveConflict(M, V, C, Ω);
10: **if** $M = null$ **then return** *null*;

To update M, we either (i) add more entries with new AST nodes (line 7), or (ii) modify existing entries by remapping some of the existing nodes to new UAs (done by ResolveConflict at line 9). In what follows, we explain in more detail how (i) and (ii) work, beginning with (i).

Lines 4-7. If Φ is satisfiable (line 4), we first update M using a satisfying assignment σ of Φ (line 5). In particular, σ maps z^v to a *minimal* UA, for every AST node in the domain of M. This updated M is guaranteed to satisfy Φ and contains only minimal UAs. But it may still be partial. Therefore, line 6 checks if W (which initially includes all AST nodes) contains any additional nodes. If not (meaning M contains all nodes), M must be a satisfying UA map and hence the algorithm terminates (line 6). Otherwise (i.e., M is partial), line 7 adds k entries to M, each mapping a new node v_i (removed from W) to a Top UA (from $\mathcal{U}_{v_i}$). An implication of line 7 is that M at line 3 may map some nodes to Top UAs—this is exactly why we need line 5. This way, we also maintain an invariant that M at line 3 has at most k entries with non-minimal UAs (k is a hyperparameter that can be tuned heuristically). Maintaining such a *lightweight* M helps make the satisfiability check (at line 4) fast. We will next explain how the else branch (lines 8-10) maintains this invariant.

Example 3.6. Consider P and P' from Section 2. Suppose M_1 in Figure 5 is the (partial) UA map at line 3. M_1's corresponding Φ_1 is satisfiable, so line 5 will obtain a satisfying assignment σ. Suppose $\sigma(z^{v_1}) = [\mathsf{T}, \mathsf{F}]$ and $\sigma(z^{v_1'}) = [\mathsf{F}, \mathsf{F}]$. This leads to $M_1 = \{v_1 \mapsto [\mathsf{T}, \mathsf{F}], v_1' \mapsto [\mathsf{F}, \mathsf{F}]\}$ after line 5. W is not empty at line 6, so line 7 loads in more nodes—say v_2, v_2'—from W, and maps them to Top UAs. At this point, we obtain the UA map M_2 shown in Figure 5.

Lines 8-10. Let us now examine the else branch where Φ is unsatisfiable. Intuitively, this means some entries M must be adjusted, in order for M to be compatible with C. Recall that Φ is a conjunction of $|M| + 1$ clauses, one for C and each of the entries in M. So if Φ is unsat, we know a subset of clauses must be in conflict (i.e., their conjunction is unsat). In this case, we first use *ExtractConflict* to obtain the set V of nodes corresponding to these conflicting clauses (line 9). More formally:

$$V = \{v_i \mid lw_i \in \textsc{UnsatCore}(\Phi)\}$$

Here, UnsatCore extracts an unsatisfiability core of Φ. Then, V includes an AST node v_i only if this unsat core has a clause labeled lw_i (recall that a different label lc is used for C). Given V, we define the following "projection" operation which gives us the subset of entries from M at V.

$$M \downarrow V = \{v \mapsto M(v) \mid v \in V\}$$

We call this set of entries a *conflict*. M must be updated to map some nodes in V to different UAs, because otherwise $M \downarrow V$ will always be a conflict, no matter how the other entries are modified or what new entries are added. This update is done by ResolveConflict (line 9), which Section 3.7 will describe in detail. At a high level, ResolveConflict returns a new M with no conflicts at V, and also maintains the aforementioned invariant that M is lightweight. It also takes as input Ω—which is a set of currently discovered conflicts—and returns a new one. For instance, the previous conflict $M \downarrow V$ will be added to Ω. ResolveConflict guarantees the returned M does not manifest any of these known conflicts; this is critical for termination. If M is *null* (line 10), it means no UA map exists—given the family of UAs—that can avoid the current conflict at V. We note that, while the update of M is driven by a conflict of Φ, the updated M may not satisfy Φ; however, future iterations will keep fixing new conflicts until satisfaction.

Example 3.7. Consider our example in Section 2, and suppose M_3 in Figure 5 is the UA map at line 3. Its corresponding Φ_3 is unsat. *ExtractConflict* at line 9 returns $V = \{v_1', v_2', v_3', v_4', v_1, v_2\}$. That is, $M_3 \downarrow V$ is a conflict—to understand why, focus on *Invoices* which is shared by P and P'. Let us assume Φ_3 is sat. Given $M_3(v_1'), M_3(v_2'), M_3(v_3'), M_3(v_4')$, we know *Invoices* must be empty, because $M_3(v_1')$ constrains its input to be empty and v_2', v_3', v_4' are all left join operators. But, $M_3(v_1), M_3(v_2)$ suggest otherwise (i.e., *Invoices* is non-empty). Contradiction. Note that v_3 and v_4 are not part of the conflict (although they were added by the previous iteration). ResolveConflict takes this conflict as input, and yields the M in Figure 4. In particular, UAs at V for M are not in conflict anymore.

3.7 Conflict Resolution and Accumulation

Now let us proceed to the ResolveConflict procedure, which is presented in Algorithm 3. It takes four inputs. C is the application condition, M is a UA map, $V \subseteq \text{dom}(M)$ is a subset of AST nodes from M where $M \downarrow V$ is a conflict, and Ω is a set of conflicts. It returns a pair (M', Ω'). In particular, M' is guaranteed to satisfy the following three properties, if it is not *null*.

(1) M' is conflict-free at V. That is, $\mathit{EncUASemantics}(M' \downarrow V) \wedge C$ is satisfiable.
(2) M' doesn't exhibit any of the conflicts in $\Omega' \supseteq \Omega$. That is, no subset of entries from M' is in Ω'.
(3) M' is lightweight. In fact, all entries in M use minimal UAs.

If no such M' exists, for the given family of UAs, *null* will be returned.

Let us now dive into the internals of ResolveConflict. Line 1 first creates Ω' that additionally includes conflict $M \downarrow V$. Then, lines 2-8 aim to generate M' that satisfies the above three properties. The basic idea is simple. We first try to fix the conflict at V, by considering "each"[8] UA u_i from $\mathcal{U}_{v_i}$ for each node v_i in V (line 2). Given $(u_1, \cdots, u_m)$, we encode the UA semantics of M_V (line 3). If the conflict persists (line 4), we add a new conflict M_V to Ω' and continue. On the other hand, if Φ_V is satisfiable—meaning property (1) holds at this point—we further make sure property (2) also holds. This is done by line 5 that further encodes Ω' and all UAs for M's nodes outside V. Specifically,

$$\mathit{EncConflicts}(\Omega') := \bigvee_{\{v_1 \mapsto u_1, \cdots, v_r \mapsto u_r\} \in \Omega'} \bigwedge_{i=1,\cdots,r} \mathit{EncUAChoice}(u_i)[z \mapsto z^{v_i}]$$

If this Φ' is satisfiable (line 6), we can easily construct an M' from a satisfying assignment σ of Φ' (line 7) that is guaranteed to satisfy both properties (1) and (2). Here, σ maps every z^v to a minimal UA; therefore, M' also meets property (3). We finally return at line 8. If Φ' is unsat for all iterations (meaning all combinations of UAs at V are exhausted), we return *null* (line 9).

Remarks. A few things are worth noting. First, line 2 uses *CoverUAs* which does not return all UAs in $\mathcal{U}_{v_i}$, but returns a *covering* set $S \subseteq \mathcal{U}_{v_i}$ of UAs. That is, for every minimal UA $u \in \mathcal{U}_{v_i}$, there exists

[8]We actually only consider those u_i's from a covering set. We will expand on this later.

Algorithm 3 Algorithm for ResolveConflict.

procedure ResolveConflict(M, V, C, Ω)
input: V is a subset of AST nodes from M. C is the application condition. Ω is a set of known conflicts. In particular, the entries from M at V (i.e., $M \downarrow V$) are known to cause a conflict.
output: A new UA map M' with no conflict at V. $\Omega' \supseteq \Omega$ is a new set of conflicts.

1: $\Omega' := \Omega \cup \{M \downarrow V\}$;
2: **for** $u_1 \in \mathit{CoverUAs}(\mathcal{U}_{v_1}), \cdots, u_m \in \mathit{CoverUAs}(\mathcal{U}_{v_m})$ where $V = \{v_1, \cdots, v_m\}$ **do**
3: $\quad M_V := \{v_1 \mapsto u_1, \cdots, v_m \mapsto u_m\}$; $\quad \Phi_V := \mathit{EncUASemantics}(M_V) \wedge C$;
4: $\quad$ **if** Φ_V is not satisfiable **then** $\Omega' := \Omega' \cup \{M_V\}$; **continue**;
5: $\quad \Phi' := \Phi_V \wedge \mathit{EncUAChoice}\Big(\big\{v \mapsto \mathit{TopUA}(\mathcal{U}_v) \mid v \in \mathrm{dom}(M) \setminus V\big\}\Big) \wedge \neg \mathit{EncConflicts}(\Omega')$;
6: $\quad$ **if** Φ' is satisfiable **then**
7: $\quad\quad \sigma := \mathit{GetModel}(\Phi')$; $\quad M' := M\big[v \mapsto \sigma(z^v) \mid v \in \mathrm{dom}(M)\big]$;
8: $\quad\quad$ **return** (M', Ω');
9: **return** (null, Ω');

a UA $u' \in S$ such that $u' \sqsupseteq u$. This allows us to search the entire space of UAs *symbolically*, without explicitly enumerating all (minimal) UAs, thereby speeding up the search. The implementation of *CoverUAs* can be tuned heuristically. Polygon chooses to set a fixed number of values in the UA for each $v_i \in V$ to top, and then enumerate all permutations of non-top values for the rest. Second, line 4 accumulates additional conflicts, with the same goal of accelerating the search. Finally, Φ' (at line 5) encodes all UAs choices for AST nodes outside V. This is necessary for completeness. In other words, our algorithm would become incomplete, if it were to only modify UAs for V.

Example 3.8. Consider M_3 from Figure 5, which has a conflict at $V = \{v_1', v_2', v_3', v_4', v_1, v_2\}$. Suppose Figure 6 corresponds to line 2. That is,

$$\mathit{CoverUAs}(v_1) = \{[\star, \star]\} \qquad \mathit{CoverUAs}(v_1') = \{[\star, \star]\}$$

$$\mathit{CoverUAs}(v_2) = \{\big[[\star, \star], [\star, \star]\big]\} \qquad \mathit{CoverUAs}(v_2') = \{\big[[\star, \star], [\star, \star]\big]\}$$

$$\mathit{CoverUAs}(v_3') = \left\{ \begin{array}{l} \big[[\mathsf{T}, \mathsf{T}], [\star, \star]\big], \big[[\mathsf{T}, \mathsf{F}], [\star, \star]\big] \\ \big[[\mathsf{F}, \mathsf{T}], [\star, \star]\big], \big[[\mathsf{F}, \mathsf{F}], [\star, \star]\big] \end{array} \right\} \qquad \mathit{CoverUAs}(v_4') = \left\{ \begin{array}{l} \big[[\mathsf{T}, \mathsf{T}], [\star, \star]\big], \big[[\mathsf{T}, \mathsf{F}], [\star, \star]\big] \\ \big[[\mathsf{F}, \mathsf{T}], [\star, \star]\big], \big[[\mathsf{F}, \mathsf{F}], [\star, \star]\big] \end{array} \right\}$$

In particular, during the first iteration of the loop (lines 2-8), M_V at line 3 is the first UA map $M_v^{(1)}$ from Figure 6. Φ_V is unsat; therefore, M_V is added to Ω' (line 4). The next two iterations add $M_V^{(2)}$ and $M_V^{(3)}$ to Ω' (line 4), since they are both unsat. At this point, Ω' contains a total of four conflicts, including the initial $M_3 \downarrow V$. Now, consider the fourth iteration. $M_V^{(4)}$ from Figure 6 corresponds to a satisfiable Φ_V at line 3. In this case, line 5 encodes the UA choices for nodes outside V—namely, v_3 and v_4—and also encodes the current conflicts Ω', which are conjoined with Φ_V to form Φ'. Note that Φ' only considers UA semantics for those AST nodes in V. In other words, if V is small, the satisfiability check at line 6 should be pretty fast. It turns out Φ' is indeed sat. Therefore, we obtain a model σ at line 7, from which we construct M'—this is the M from Figure 4. ResolveConflict terminates at this point, returning M and Ω (with four conflicts).

3.8 Theorems

This section presents key theorems, whose proofs can be found in the extended version [72].

Theorem 3.9 (Correctness of UA semantics). *Suppose $\mathit{EncUASemantics}(F, u)$ yields an SMT formula φ, for query operator F and UA $u \in \mathcal{U}_F$. For any model σ of φ, the corresponding inputs $\sigma(\vec{x})$*

and output $\sigma(y)$ are consistent with the precise semantics of F; that is, $[\![F]\!]_{\sigma(\vec{x})} = \sigma(y)$. Intuitively, this theorem states that any under-approximation u of F is always encoded into an SMT formula whose satisfying assignments correspond to genuine input-output behaviors of F. Therefore, our approach is consistent with incorrectness logic [49] in that both consider reachable outputs (no false positives).

Theorem 3.10 (Refinement of UA Semantics). *Given query operator F, and two UAs $u \in \mathcal{U}_F$ and $u' \in \mathcal{U}_F$ where u' refines u (i.e., $u' \sqsubseteq u$), we have $\mathit{EncUASemantics}(F, u) \Rightarrow \mathit{EncUASemantics}(F, u')$. Intuitively, this means the set of F's reachable outputs for u' should be a subset of that for u; therefore, the UA semantics encoding for u' is entailed by that for u.*

Theorem 3.11 (Soundness). *Given queries $P_1, \cdots, P_n$ and application condition C, if GenInput returns an input database I, then we have $C[y_1 \mapsto P_1(I), \cdots, y_n \mapsto P_n(I)]$ is true. Intuitively, this is because our UAs always encode genuine input-output behaviors of P_i.*

Theorem 3.12 (Completeness). *Given queries $P_1, \cdots, P_n$ and application condition C, if GenInput returns null, then there does not exist an input I (with respect to the semantics presented in Figure 8) for which $C[y_1 \mapsto P_1(I), \cdots, y_n \mapsto P_n(I)]$ is true. Intuitively, this is because our approach essentially performs exhaustive search while using the lattice structure of UAs to soundly prune the search space.*

4 Evaluation

This section describes a series of experiments designed to answer the following questions:

- **RQ1**: Can Polygon effectively solve real-world benchmarks?
- **RQ2**: How does Polygon compare against state-of-the-art techniques?
- **RQ3**: How useful are various ideas in Polygon?

Two applications. The first one is the long-standing problem of SQL equivalence checking [13, 15, 28, 69]. Given P_1 and P_2, the goal is to generate I such that $y_1 \neq y_2$, where y_i is P_i's output $P_i(I)$. The other is query disambiguation [6, 69]. Given $P_1, \cdots, P_n$, find an input I as well as an even split of them into two disjoint sets, such that for all P_i and P_j: if they belong to the same set, $O_i = O_j$; otherwise $O_i \neq O_j$.[9] In what follows, we describe how we collect benchmarks for both applications.

Equivalence refutation benchmarks. We reuse the 24,455 benchmarks from a recent equivalence-checking work VeriEQL [28], where each benchmark is a pair of SQL queries. These query pairs are obtained from a wide range of downstream tasks, including auto-grading, query rewriting and mutation testing. For instance, more than 20,000 query pairs correspond to auto-grading, where one query in the pair is the ground-truth and the other is a user submission accepted by LeetCode (i.e., passing LeetCode's test cases). In other words, non-equivalence in this case indicates inadequate testing, and counterexamples can help LeetCode developers further strengthen their test suites. We believe this is a comprehensive set of benchmarks for evaluating Polygon. Finally, we note that, while VeriEQL [28] was able to refute more than 3,000 of these benchmarks, the solvability for the rest is unknown (and manually checking these benchmarks is a non-starter).

Disambiguation benchmarks. We curate an extensive set of query disambiguation benchmarks, based on query synthesis tasks from the Cubes work [6] (which is a state-of-the-art SQL synthesizer). In particular, given input-output examples from each Cubes task, we generate one disambiguation *task*, containing all satisfying queries that can be synthesized within 2 minutes. This yields 2,861 disambiguation tasks. The number of queries per task ranges from 2 to 3,434, with an average of 272 and a median of 191. While Polygon is directly applicable, a challenge for conducting our evaluation here is how to interpret the results: if a task was not solved, is that because it is not solvable at all,

[9] The encoding of this application condition can be found in the extended version [72].

or is it due to Polygon's inability? Manually solving these tasks (even a subset) is nearly impossible. To address this, we develop a procedure to curate disambiguation *benchmarks* (based on our 2,816 tasks) which are by construction solvable. In particular, given a set S of synthesized queries from the task, our first step is to partition S into a set of equivalence classes G as follows.

```
procedure Partition(S)
  G := ∅;
  while S is not empty do
    P := S.remove();    C = [P];
    foreach P_i ∈ S do
      if P_i is equivalent to P up to bound b then C.add(P_i);  S.remove(P_i);
      else if P_i is not equivalent to P then continue;
      else S.remove(P_i);    ▷ bounded verifier timed out
    G := G ∪ {C};
  return G;
```

The key idea is to leverage a (bounded) equivalence verifier (in particular, VeriEQL [28]) to partition S: queries in the same class C are equivalent to each other (up to a certain bound b), whereas queries from different classes are not. In particular, the first query P in each class C is a representative. We have a counterexample for any two representatives from two classes.

Then, given G, we create a disambiguation benchmark with $2n$ queries, by selecting the first n queries from each of any two classes. We consider $n = 25, 50$ in our evaluation, yielding 4,245 and 2,475 disambiguation benchmarks, respectively. We name them D-50 and D-100.

4.1 RQ1: Can Polygon Solve Real-World Benchmarks?

> Polygon solved 5,497 equivalence refutation benchmarks (out of 24,455 in total), with a median running time of 0.1 seconds per benchmark. Among 4,245 disambiguation benchmarks (each with 50 queries), Polygon solved 94% of them using a median of 3.7 seconds per benchmark.

Setup. Given a benchmark (either equivalence refutation or disambiguation), we run Polygon and record: (1) if the benchmark is solved before timeout (1 minute), and (2) if so, the running time. We also log detailed statistics, which we will summarize and report below.

Results. Table 1 summarizes our key results. Polygon can solve 22.5% of the equivalence refutation benchmarks. This is a surprisingly high ratio, given that 98.1% of our ER benchmarks are queries accepted by LeetCode. For disambiguation, Polygon consistently solves over 90% of the benchmarks in all settings. The solving time in general is pretty fast. Polygon slows down when disambiguating more queries, which is expected; but still, within 40 seconds, it can solve over 83% of the benchmarks.

The next columns report some key internal statistics. In general, it takes Polygon a small number of iterations to solve a benchmark. This is because—while the number of AST nodes in a benchmark is large—line 7 in Algorithm 2 loads in multiple nodes in one iteration; therefore, we observe far fewer iterations. Some iterations are spent on fixing conflicts, by invoking ResolveConflict; we report the number of such iterations in "#ResolveConflict". While the maximum can be over 100, the number of such calls is quite small in general. But the impact of each call is much larger—look at the "#Nodes adjusted" column. On average, over a quarter of a benchmark's nodes are remapped to different UAs. In other words, there are quite some backtrackings happening, but they are packed into a small number of ResolveConflict calls. We should note that the number of nodes in conflict (i.e., $|V|$ at line 9 of Algorithm 2) is typically small. In particular, the median and average for ER are 4 and 5.1. For D-50 and D-100, median/average are 2/49.2 and 2/64.3, respectively. In other words,

Table 1. Polygon results across all benchmarks for equivalence refutation (ER) and disambiguation (D-50 and D-100). We first report the (average, median, maximum) number of AST nodes per benchmark—the number of AST nodes for a benchmark is the sum of AST sizes across all queries in the benchmark. Then we show the number of benchmarks solved (with the total number of benchmarks below it) in "#Solved (#Total)" column, followed by the (average, median) running times across all solved benchmarks (recall: timeout is 1 minute). "#Iterations" shows the (average, median, maximum) number of iterations (lines 2-10 in Algorithm 2) across all *solved* benchmarks. The next column reports the number of calls to ResolveConflict. Finally, we present the (average, median, maximum) number of AST nodes adjusted before and after calling ResolveConflict—that is, the number of AST nodes in M (see Algorithm 3) that are mapped to different UAs in M' (i.e., $|M' \setminus M|$).

	#Nodes per benchmark			#Solved	Time (sec)		#Iterations			#ResolveConflict			#Nodes adjusted		
	avg.	*med.*	*max.*	(#Total)	*avg.*	*med.*	*avg.*	*med.*	*max.*	*avg.*	*med.*	*max.*	*avg.*	*med.*	*max.*
ER	9	8	33	5,497 (24,455)	0.6	0.1	4.5	3	139	2.5	1	137	6	5	26
D-50	211	213	343	4,004 (4,245)	6.4	3.7	4.8	4	39	1.8	1	36	82	51	280
D-100	435	442	616	2,392 (2,475)	21.7	18	4.9	4	23	1.9	1	20	126	101	380

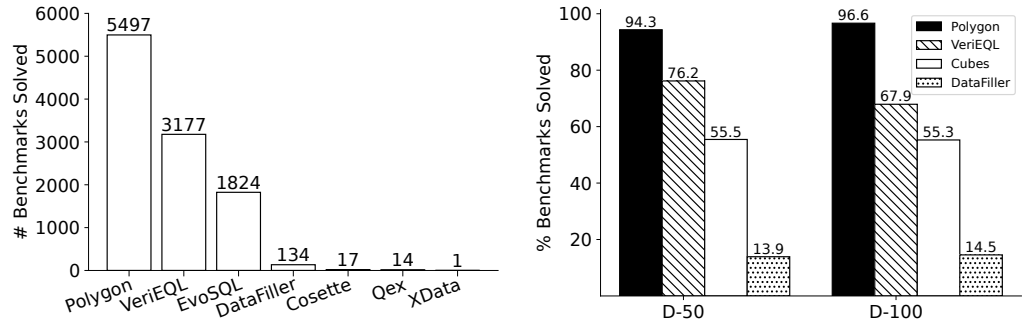
line 5 of Algorithm 3 encodes UA semantics for very few nodes (i.e., those in V), whereas it encodes UA choices for far more nodes (i.e., those outside V). The latter is typically much cheaper to solve.

Discussion. In our experience, the number of AST nodes added to M (i.e., k at line 7 in Algorithm 2) in each iteration has a significant impact on the overall performance. One extreme is to set $k = 1$; in this case, the number of iterations would be at least the number of AST nodes across P_i's (see "#Nodes per benchmark" column in Table 1). This would slow down the algorithm significantly. On the other hand, adding all nodes in one iteration is also suboptimal (as we will show in Section 4.3). Polygon's heuristic is to add all nodes for k queries (e.g., $k = 50$ for D-100), which seems to achieve a good balance between (1) the number of iterations and (2) SMT solving overhead in each iteration. The implementation of *CoverUAs* also matters noticeably in practice. The two extremes (namely, returning Top UA for each v_i at line 2 of Algorithm 3, and returning all minimal UAs for v_i) would be slow (which we will show in Section 4.3). Polygon's heuristic is to use top for a fixed number of values in v_i's UA (in particular, we set 8 UA choices to $\star$ in our experiments), while spelling out all permutations for the rest of the UA values. In our experience, how many UA values are set to top seems to impact the performance more than which values.

4.2 RQ2: Polygon vs. State-of-the-Art

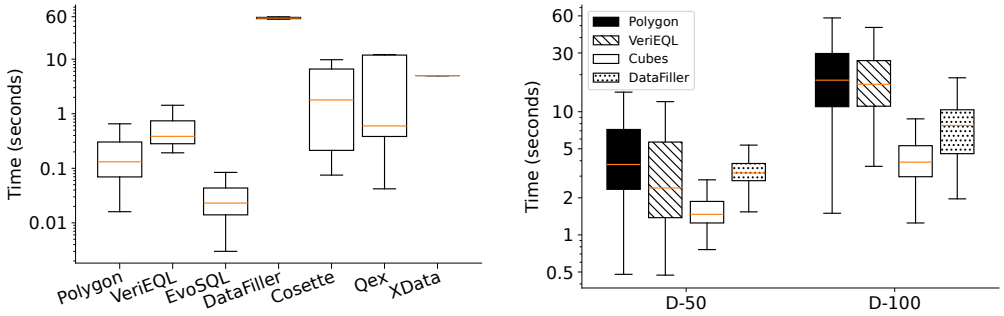
Polygon outperforms all state-of-the-art techniques by a significant margin—in particular, 1.7x more benchmarks solved for equivalence refutation and 1.4x for disambiguation.

Baselines. For each application, we consider all relevant existing work—to our best knowledge—as our baselines. For equivalence refutation, we include 6 baselines: (1) VeriEQL [28, 71] (SMT-based), (2) Cosette [15] (based on Rosette [63], with provenance-based pruning [69]), (3) Qex [66] (SMT-based, with provenance-based pruning [69]), (4) DataFiller [18] (fuzzing-based), (5) XData [9, 10] (mutation-based tester), and (6) EvoSQL [7] (search-based tester using random search and genetic algorithms). For disambiguation, we include 3 baselines: (i) the disambiguation component (fuzzing-based) from Cubes [6], (ii) a modified version of VeriEQL that encodes full semantics for multiple (not just two) queries and the disambiguation condition, and (iii) DataFiller.



(a) Equivalence refutation. (b) Disambiguation.

Fig. 9. Polygon vs. baselines, in terms of benchmarks solved. Note that for disambiguation, we present the *percentage* of benchmarks solved. (Recall: all of our disambiguation benchmarks are solvable by construction).



(a) Equivalence refutation. (b) Disambiguation.

Fig. 10. Polygon vs. baselines, in terms of solving time. For each tool (including Polygon and all baselines), we present the quartile statistics of its solving times across all solved benchmarks.

Setup. We use the same setup as RQ1 for all baselines (with 1-minute timeout). For each application, we record the benchmarks that are solved by each tool and the corresponding solving times.

Results. Figure 9 and Figure 10 present our results. Polygon is a clear winner for both applications. It disproves significantly more query pairs than the best baseline VeriEQL (5,497 vs. 3,177). Across all refuted query pairs, Polygon's median running time is 0.1 seconds, which is even faster than VeriEQL (with a median of 0.4 seconds across its solved benchmarks). For disambiguation, Polygon also solves significantly more benchmarks than all baselines. It is interesting that Polygon solves more D-100 benchmarks than D-50, which is also observed for DataFiller. In terms of time—see Figure 10(b)—Polygon is comparable with VeriEQL. DataFiller and Cubes are understandably faster (due to their fuzzing-based methods), but they solve far fewer benchmarks (see Figure 9).

4.3 RQ3: Ablation Studies

All of our design choices (including the top-level conflict-driven search algorithm architecture, the lattice structure of UAs, conflict extraction from unsatisfiability core, conflict accumulation) play an important role for Polygon's performance.

Two sets of ablations. The first set considers ablations that perform brute-force enumeration over UAs, with the goal of understanding the impact of the top-level architecture of our algorithm. Variants in the second set alter lower-level designs, and reuse the same architecture as Polygon.

First set of ablations. To be complete, these ablations must search a covering set of UAs for $\mathcal{U}_v$, for each AST node v. We create the following variants that use different covering sets.

- Enum-MinUAs, which considers all *minimal* UAs for each v. It enumerates all combinations of minimal UAs (one per node), and stops when a satisfying UA map is found.
- Enum-TopUAs, which considers only *Top UA* for each v; i.e., it encodes the full semantics for v.
- Enum-50%Top, which considers UAs with *half* of the values set to top. In particular, for each v, we first set 50% of the values (randomly picked) in v's UA to top. Then, we create all permutations of non-top values for the remaining half, each of which corresponds to a UA for v.
- Enum-25%Top and Enum-75%Top, which are constructed in the same fashion as Enum-50%Top but set 25% and 75% (respectively, randomly selected) of the values to top.

While still based on UAs, these ablations differ from our algorithm architecturally: they enumerate UA choices for every AST node in a brute-force manner, and return the first working combination. The reason we consider five ablations here is to obtain a full range of their performance data.

Second set of ablations. We create the following variants.

- TopUACover, where *CoverUAs* is changed to return Top UA for each v_i at line 2 of Algorithm 3.
- MinUAsCover, where *CoverUAs* returns all minimal UAs for each v_i at line 2 of Algorithm 3.
- NoNewConflicts, which removes line 4 from Algorithm 3 and hence does not add new conflicts other than the one at line 1 (which is necessary for termination).
- NoUnsatCore, where V includes all nodes from M at line 9 of Algorithm 2.
- AddMinUAs, which maps each v_i at line 7 of Algorithm 2 to a minimal UA (randomly selected from $\mathcal{U}_{v_i}$) and also removes line 5 (which is no longer necessary).

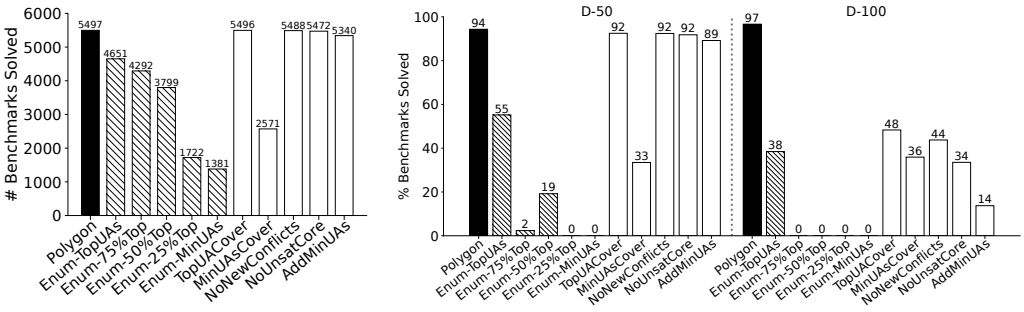
Setup. We use the same setup as in RQ2 to run all ablations and collect experimental data.

Results. Figure 11 and Figure 12 present our results. Let us begin with the first set of ablations. Our take-away is that the Enum-X ablations are significantly worse than Polygon, both in terms of benchmarks solved and the solving time—for both equivalence refutation and disambiguation. This underscores the advantage of Polygon's algorithm architecture. On the other hand, ablations from the second set are on par with Polygon (except for MinUAsCover which performs poorly) on ER and D-50: they solve slightly fewer benchmarks than Polygon, using slightly more time. However, they become significantly worse on D-100 (with more queries to disambiguate). This highlights the importance of our various design choices—e.g., the lattice structure of UAs which allows *CoverUAs* to partition the search space—especially for harder problems (e.g., those involving more queries).

5 Related Work

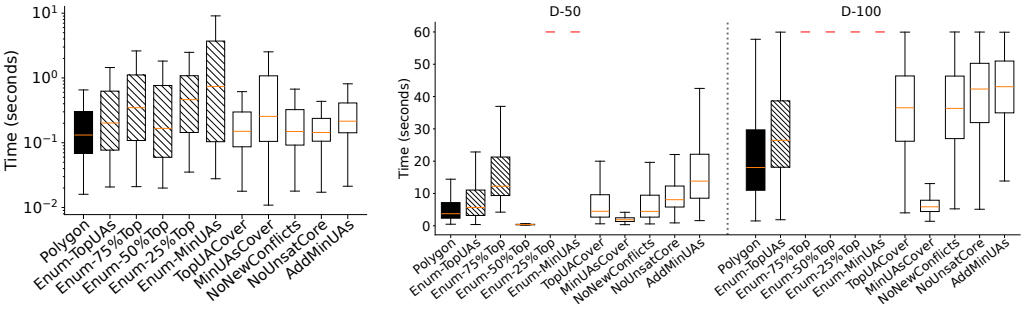
This section briefly discusses some closely related work.

Under-approximate reasoning. Our work is inspired by O'Hearn's seminal work on incorrectness logic [49] and a long line of works that leverage under-approximate reasoning for various tasks, such as proving non-termination [55], detecting memory errors [40], reasoning about concurrency [5, 54], scaling static analysis [22], dynamic symbolic execution [24], among others [4, 20, 26, 34, 48, 51, 53]. Polygon can be viewed as a successful use of under-approximate reasoning to generate test inputs for SQL—in particular, for equivalence refutation and disambiguation. Building upon the literature, we contribute a compositional approach (based on SMT) to define a family of under-approximations per SQL operator, and a fast algorithm to search within this space for desired under-approximations.



(a) Equivalence refutation. (b) Disambiguation.

Fig. 11. Polygon vs. ablations, in terms of benchmarks solved.



(a) Equivalence refutation. (b) Disambiguation.

Fig. 12. Polygon vs. ablations, in terms of solving time. A red bar at the top means "timeout on all benchmarks".

Symbolic execution. Our work can be viewed as a form of (backward) symbolic execution [3, 8, 11]: it begins with the application condition C (i.e., an assertion) and under-approximates the semantics (akin to picking execution paths) of AST nodes top-down (i.e., backwards). We perform backtracking when the current UA map does not meet C—but in a different fashion from prior work—by analyzing a subset of UAs that are in conflict. In particular, we utilize our lattice structure of UAs to search many UAs at the same time, which allows us to efficiently find a fix for the conflict. During this process, we discover and block additional conflicts, to speed up future search. This is related to, but different from, prior pruning techniques, such as those based on interpolation [31, 47] and detecting inconsistent code [57]. Our work is also related to summary-based symbolic execution [1, 59, 60]—especially compositional dynamic symbolic execution [2, 24, 58]—in the sense that we also utilize (under-approximate) function summaries. Our contribution is a novel compositional method to define a lattice of summaries (per SQL operator), which allows us to perform combinatorial search efficiently. Non-minimal UAs essentially correspond to state merging [24, 35, 41, 52], and are used locally (when analyzing conflicts or adding new AST nodes) to not overly stress the SMT solver.

Symbolic reasoning for SQL. Polygon is especially related to prior symbolic reasoning techniques that are tailored towards SQL [12, 14–16, 28, 66, 69] (all of which we compare with in our evaluation). Unlike these works, Polygon under-approximates SQL semantics and performs reasoning while searching under-approximations. There is a long line of work on SQL equivalence verification [14, 16, 27, 74, 75]. Polygon focuses on generating inputs to satisfy properties of query outputs, including but not limited to checking (non-)equivalence. Alloy [29, 30, 64] represents another line of related

work, which natively supports relational operators and hence in principle can be used to reason about SQL. The key distinction of our work lies in the granularity of UAs and the capability of performing search over them. In contrast, it is unclear if Alloy's "scope" mechanism is as flexible as our UAs. More importantly, Alloy's scope is always fixed beforehand and cannot be dynamically changed during analysis.

Test data generation for SQL. Polygon is also closely related to works on testing SQL queries. For instance, XData [9, 10] is a mutation-based tester to detect common SQL mistakes. DataFiller [18] is a fuzzer that generates random test inputs, given the database schema. EvoSQL [7] generates test data via an evolutionary search algorithm, guided by predicate coverage [65]. Different from these approaches, we incorporate SQL semantics to generate satisfying inputs for a given property. Polygon is also related to property-based testing [17, 25, 36–39, 50, 76], in the sense that we aim to generate input databases for a given property over SQL queries. Our generation process, however, uses SMT-based under-approximate reasoning over both the property and the queries, rather than some form of random input generation (as in many prior works) or enumeration [45, 56].

Conflict-driven search. The idea of conflict-driven search has received success in multiple areas. For example, modern constraint solvers use conflict-driven clause learning to derive new clauses for faster boolean satisfiability solving [43]. In program synthesis, a candidate program that fails to meet the specification can be viewed as a conflict. Various algorithms [23, 44, 61, 70] have been proposed to generalize such a conflict to unseen programs that would fail due to the same reason. Our work is distinct in a few ways: we perform conflict-driven search over under-approximations, our conflict is generalized to new ones in a way that takes advantage of a predefined lattice structure of UAs, and we aim to generate inputs for SQL queries to meet a given property.

6 Conclusion and Discussion

This paper presented a new method based on under-approximation search to perform symbolic reasoning for SQL. Our evaluation demonstrated significant performance boost over all state-of-the-art techniques, for two reasoning tasks (namely SQL equivalence refutation and disambiguation).

While this work is largely focused on SQL, we believe the underlying principles have the potential to generalize to other languages and domains. A fundamental assumption is: analyzing an under-approximation (UA) of a program is cheap—which we believe holds true in general. Then, we need to curate a family of UAs for the language. In the case of SQL, we were able to do this compositionally. We believe this is also possible, in general, for programs representable using a loop-free composition of blocks (like an AST). Finally, our search technique (Algorithms 2 and 3) does not assume SQL. It, however, assumes a lattice of UAs, which we believe is definable for other languages.

To explore the full generality of this idea, one interesting future direction is to build the idea on top of the Rosette solver-aided programming language [62]. For instance, one approach is to build a Rosette-based symbolic interpreter that is parameterized with a UA. Then, given n programs, it can perform symbolic execution to check against the given application condition with respect to the given UA. On top of this symbolic interpreter, we can implement the UA search algorithm.

Acknowledgments

We would like to thank the PLDI anonymous reviewers for their insightful feedback. We thank Zheng Guo, Chenglong Wang, and Wenxi Wang for their feedback on earlier drafts of this work. We would also like to thank Danny Ding for helping with a baseline in the evaluation, Xiaomeng Xu and Yuxuan Zhu for their contributions to earlier versions of this work, and Brian Zhang for helping process some of the benchmarks. This research is supported by the National Science Foundation

under Grant Numbers CCF-2210832, CCF-2318937, CCF-2236233, and CCF-2123654, as well as an NSERC Discovery Grant.

Artifact Availability Statement

The artifact that implements the techniques and supports the evaluation results reported in this paper is available on Zenodo [73].

References

[1] Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even Mendoza, Grigory Fedyukovich, Antti EJ Hyvärinen, and Natasha Sharygina. 2017. HiFrog: SMT-based function summarization for software verification. In *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II 23*. Springer, 207–213.

[2] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 367–381.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[4] Thomas Ball, Orna Kupferman, and Greta Yorsh. 2005. Abstraction for falsification. In *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*. Springer, 67–81.

[5] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28.

[6] Ricardo Brancas, Miguel Terra-Neves, Miguel Ventura, Vasco Manquinho, and Ruben Martins. 2022. CUBES: a parallel synthesizer for SQL using examples. *arXiv preprint arXiv:2203.04995* (2022).

[7] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2018. Search-based test data generation for SQL queries. In *Proceedings of the 40th international conference on software engineering*. 1220–1230. doi:10.1145/3180155.3180202

[8] Marek Chalupa and Jan Strejček. 2021. Backward symbolic execution with loop folding. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings 28*. Springer, 49–76.

[9] Bikash Chandra, Ananyo Banerjee, Udbhas Hazra, Mathew Joseph, and S Sudarshan. 2019. Automated grading of sql queries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1630–1633. doi:10.1109/ICDE.2019.00159

[10] Bikash Chandra, Bhupesh Chawda, Biplab Kar, KV Maheshwara Reddy, Shetal Shah, and S Sudarshan. 2015. Data generation for testing and grading SQL queries. *The VLDB Journal* 24, 6 (2015), 731–755.

[11] Satish Chandra, Stephen J Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 363–374.

[12] Alvin Cheung, Maaz Bin Safeer Ahmad, Brandon Haynes, Chanwut Kittivorawong, Shadaj Laddad, Xiaoxuan Liu, Chenglong Wang, and Cong Yan. 2023. Towards Auto-Generated Data Systems. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4116–4129.

[13] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. 2017. Demonstration of the cosette automated sql prover. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1591–1594.

[14] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *arXiv preprint arXiv:1802.02229* (2018).

[15] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *CIDR*.

[16] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices* 52, 6 (2017), 510–524.

[17] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

[18] Fabien Coelho. 2013. DataFiller – generate random data from database schema. https://github.com/memsql/datafiller.

[19] Cosette. 2018. Cosette website. https://cosette.cs.washington.edu/.

[20] Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic:(Dis-) Proving Program Hyperproperties. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1485–1509.

[21] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in*

Computer Science, Vol. 4963). 337–340. doi:10.1007/978-3-540-78800-3_24

[22] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.

[23] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 420–435. doi:10.1145/3192366.3192382

[24] Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 47–54.

[25] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[26] Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. 2019. A true positives theorem for a static race detector. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[27] Todd J Green. 2009. Containment of conjunctive queries on annotated relations. In *Proceedings of the 12th international conference on database theory*. 296–309.

[28] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024. VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1071–1099.

[29] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on software engineering and methodology (TOSEM)* 11, 2 (2002), 256–290.

[30] Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis*. MIT press.

[31] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A Navas. 2013. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 48–58.

[32] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 215–224.

[33] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1143–1158.

[34] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. 2015. Under-approximating loops in C programs for fast counterexample detection. *Formal methods in system design* 47 (2015), 75–92.

[35] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *Acm Sigplan Notices* 47, 6 (2012), 193–204.

[36] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. 2017. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 114–129.

[37] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[38] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2017. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[39] Leonidas Lampropoulos and Benjamin C Pierce. 2018. QuickChick: Property-Based Testing in Coq. *Software Foundations series* 4 (2018).

[40] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27.

[41] Sirui Lu and Rastislav Bodík. 2023. Grisette: Symbolic Compilation as a Functional Programming Library. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 455–487.

[42] Vasco Manquinho and Ruben Martins. 2024. Towards Reliable SQL Synthesis: Fuzzing-Based Evaluation and Disambiguation. In *FASE 2024*, Vol. 14573. Springer Nature, 232.

[43] João Marques-Silva, Inês Lynce, and Sharad Malik. 2021. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability - Second Edition*. Vol. 336. IOS Press, 133–182. doi:10.3233/FAIA200987

[44] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An Extensible Synthesis Framework for Data Science. *Proc. VLDB Endow.* 12, 12 (2019), 1914–1917. doi:10.14778/3352063.3352098

[45] Rudy Matela Braquehais. 2017. *Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing*. Ph. D. Dissertation. University of York.

[46] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 291–301.

[47] Kenneth L McMillan. 2010. Lazy annotation for program testing and verification. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 104–118.

[48] Toby Murray. 2020. An under-approximate relational logic: heralding logics of insecurity, incorrect implementation & more. *arXiv preprint arXiv:2003.04791* (2020).
[49] Peter W O'Hearn. 2019. Incorrectness logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.
[50] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401.
[51] Corina S Păsăreanu, Radek Pelánek, and Willem Visser. 2005. Concrete model checking with abstract matching and refinement. In *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*. Springer, 52–66.
[52] Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. 2022. A formal foundation for symbolic evaluation with merging. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–28.
[53] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local reasoning about the presence of bugs: Incorrectness separation logic. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*. Springer, 225–252.
[54] Azalea Raad, Julien Vanegue, Josh Berdine, and Peter O'Hearn. 2023. A General Approach to Under-Approximate Reasoning About Concurrent Programs. In *34th International Conference on Concurrency Theory (CONCUR 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.
[55] Azalea Raad, Julien Vanegue, and Peter O'Hearn. 2024. Non-termination Proving at Scale. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 246–274.
[56] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *Acm sigplan notices* 44, 2 (2008), 37–48.
[57] Daniel Schwartz-Narbonne, Martin Schäf, Dejan Jovanović, Philipp Rümmer, and Thomas Wies. 2015. Conflict-directed graph coverage. In *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings 7*. Springer, 327–342.
[58] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 842–853.
[59] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2011. Interpolation-based function summaries in bounded model checking. In *Haifa verification conference*. Springer, 160–175.
[60] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2012. Incremental upgrade checking by means of interpolation-based function summaries. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 114–121.
[61] Aalok Thakkar, Nathaniel Sands, George Petrou, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2023. Mobius: Synthesizing Relational Queries with Recursive and Invented Predicates. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1394–1417. doi:10.1145/3622847
[62] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 135–152.
[63] Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 530–541. doi:10.1145/2594291.2594340
[64] Emina Torlak and Daniel Jackson. 2007. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 632–647.
[65] Javier Tuya, María José Suárez-Cabal, and Claudio De La Riva. 2010. Full predicate coverage for testing SQL database queries. *Software Testing, Verification and Reliability* 20, 3 (2010), 237–288.
[66] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL query explorer. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 425–446.
[67] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive query synthesis from input-output examples. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1631–1634.
[68] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 452–466.
[69] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2018. Speeding up symbolic reasoning for relational queries. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.
[70] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration using Datalog Program Synthesis. *Proc. VLDB Endow.* 13, 7 (2020), 1006–1019. doi:10.14778/3384345.3384350
[71] Pinhan Zhao, Yang He, Xinyu Wang, and Yuepeng Wang. 2024. Demonstration of the VeriEQL Equivalence Checker for Complex SQL Queries. *Proceedings of the VLDB Endowment (PVLDB)* 17, 12 (2024), 4437–4440.
[72] Pinhan Zhao, Yuepeng Wang, and Xinyu Wang. 2025. Polygon: Symbolic Reasoning for SQL using Conflict-Driven Under-Approximation Search. arXiv:2504.06542

[73] Pinhan Zhao, Yuepeng Wang, and Xinyu Wang. 2025. Polygon: Symbolic Reasoning for SQL using Conflict-Driven Under-Approximation Search. doi:10.5281/zenodo.15059866
[74] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1276–1288.
[75] Qi Zhou, Joy Arulraj, Shamkant B Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2735–2748.
[76] Zhe Zhou, Ashish Mishra, Benjamin Delaware, and Suresh Jagannathan. 2023. Covering all the bases: Type-based verification of test input generators. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1244–1267.

Received 2024-11-15; accepted 2025-03-06