Speeding up Symbolic Reasoning for Relational Queries

CHENGLONG WANG, University of Washington, USA
ALVIN CHEUNG, University of Washington, USA
RASTISLAV BODIK, University of Washington, USA

The ability to reason about relational queries plays an important role across many types of database applications, such as test data generation, query equivalence checking, and computer-assisted query authoring. Unfortunately, symbolic reasoning about relational queries can be challenging because relational tables are multisets (bags) of tuples, and the underlying languages, such as SQL, can introduce complex computation among tuples.

We propose a space refinement algorithm that soundly reduces the space of tables such applications need to consider. The refinement procedure, independent of the specific dataset application, uses the abstract semantics of the query language to exploit the provenance of tuples in the query output to prune the search space. We implemented the refinement algorithm and evaluated it on SQL using three reasoning tasks: bounded query equivalence checking, test generation for applications that manipulate relational data, and concolic testing of database applications. Using real world benchmarks, we show that our refinement algorithm significantly speeds up (up to 100×) the SQL solver when reasoning about a large class of challenging SQL queries, such as those with aggregations.

CCS Concepts: • **Theory of computation → Logic and verification**; **Database query languages (principles)**;

Additional Key Words and Phrases: Symbolic Reasoning, Verification, Relational Query

ACM Reference Format:
Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2018. Speeding up Symbolic Reasoning for Relational Queries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 157 (November 2018), 25 pages. https://doi.org/10.1145/3276527

1 INTRODUCTION

The relational model [Codd 1970] is one of the most popular ways to represent data. Under the relational model, data is organized into tables. Each table consists of a bag of tuples that contains multiple attributes and their corresponding values, with all tuples in the same table sharing the same number of attributes. The simplicity of the relational model has led to its widespread adoption among database systems, with numerous commercial and open-source implementations available.

The popularity of the relational model has also led to the development of various development tools and applications that utilize relational databases. Many such applications require reasoning about tables. For instance, one way to determine whether two relational queries, q_1 and q_2, are semantically equivalent is to check if there exists a table T such that the two queries return different results when evaluated on T. Furthermore, database testing tools require the generation of test

Authors' addresses: Chenglong Wang, University of Washington, USA, clwang@cs.washington.edu; Alvin Cheung, University of Washington, USA, akcheung@cs.washington.edu; Rastislav Bodik, University of Washington, USA, bodik@cs.washington.edu.

2475-1421/2018/11-ART157
https://doi.org/10.1145/3276527

inputs from the provided query and test conditions to check whether the query can produce an ill-formed output on some input tables: for example, an application might return an error if a query in the application can return empty results or results containing NULL values when evaluated on a non-empty input employee relation, and these errors can be caught if we have corresponding test inputs.

Obviously, given the large number of possible tables for a database schema, exhaustively enumerating and explicitly storing them in program memory for query reasoning is infeasible. Hence, prior work has focused on reasoning about tables *symbolically*. For instance, Cosette [Chu et al. 2017b], a query equivalence checker, leverages Satisfiability Modulo Theories (SMT) solvers for bounded equivalence checking. Given two queries q_1 and q_2, Cosette encodes the outputs of both queries symbolically as an SMT formula and sends the formula to an SMT solver to either: (1) prove that the two formulas are semantically equivalent (and hence the input queries are equivalent), or (2) show that the two queries are inequivalent by finding an input table as the counterexample (i.e., the two queries will return different results when evaluated on it). Similar techniques have been employed in testing frameworks as well [Cheung et al. 2011; Tanno et al. 2015; Veanes et al. 2010]: these frameworks aim to generate test inputs from the given query and test conditions. These test conditions can be path conditions from a database application or unit test assertions for output properties that we are interested in.

While representing tables symbolically indeed allows such tools to reason about different database applications that arise in practice, we believe a substantial opportunity remains for further improvement. To our knowledge, none of such relational query reasoning tools leverage the properties of tables or the domain-specific aspects of relational query languages. An example property is the independence between groups in a query with Group By: in these queries, tuples are partitioned into different groups according to the value of the grouping keys, and different groups are reduced into single tuples independent of each other. As a result, without exploiting such properties, many existing tools explore an unnecessarily large number of tables during execution. As we will show, this significantly hampers such tools' ability to analyze more complex real-world database applications.

In this paper, we describe a way to *systematically identify and exploit* properties of tables and the SQL query language for relational query reasoning.[1] Specifically, we focus on scaling up SQL reasoning tasks aimed at finding input tables T_I for a given query q (or multiple queries) such that the output table T_O satisfies a property expressible using a subset of first-order logic with a single existential quantifier: $\exists t_O \in T.\psi(t_O, T_O)$ (i.e., there exists a tuple t_O in the output table T_O satisfying the property ψ). As we will discuss in section 2, while this subset does not include properties such as "the output table has exact size 5," it nonetheless allows us to check for properties such as the following: (1) "the output table contains a tuple with multiplicity greater than zero" (under bag semantics, multiplicity of a tuple if the number of times it appears in the table), which arise in many situations in the unit test generation task for database applications [Veanes et al. 2010]; (2) "the output table does not contain any attribute with value equals to NULL," which is useful for query optimization; and (3) "queries q_1 and q_2 outputs contain the same tuple t but with different multiplicities," meaning that the two queries are semantically different [Chandra et al. 2015].

To determine the validity of such properties, our key idea is to *backwardly* compute the *provenance* of tuples, i.e., the lineage of how a given tuple is derived from the input tables, using *the abstract*

[1]While we focus on SQL in this paper given its popularity, we believe the techniques described in this paper can be applicable to other relational query languages as well.

semantics of SQL to refine the space of input tables that symbolic reasoning tools need to consider. We next present two motivating examples to demonstrate our insights.

Motivating Example 1. We first show a simple example in the context of unit test generation [Veanes et al. 2010]. In this task, given the following query q parameterized with @x, we aim to either find an input table Bonus such that the output of q is non-empty for a given concrete parameter, or prove that no such input table exists. If such input exists for the given parameter, the input table, the parameter, and the query can be combined as a unit test for the database.

```
Select job, sal
From Bonus
Where sal <= @x
  And sal > 2;
```

Bonus$_1$

job	sal
2	11

Bonus$_2$

job	sal
3	5
3	5

Bonus$_3$

job	sal
2	11
3	5
3	5

Fig. 1. Given the query q, whether q would produce a non-empty output when applied to Bonus$_3$ is subsumed by whether q produces non-empty outputs when evaluated on Bonus$_1$ and Bonus$_2$.

Figure 1 shows a concrete example. The query q (parameterized with @x) filters tuples in table Bonus using the condition $\texttt{sal} > 2 \land \texttt{sal} < @\texttt{x}$. Assuming that the number of tuples considered by the query solver is bounded to k, the solver would encode the search space consisting of all tables with at most k tuples as a symbolic table and search for desirable assignments for all $2k$ attributes in the table, say by invoking an SMT solver.

However, we actually do not need to consider the full space of all tables with at most k tuples to search for the desirable table. Instead, given that the query q contains only one filter operation, we only need to consider the space of all tables with exactly *one* tuple (note that the tuple may appear multiple times in the table due to bag semantics). This is true because the query q does not introduce interactions among different tuples during computation, i.e., whether a tuple t in the input table would be included in the output does not depend on whether another tuple t' would be included or not. In other words, the *provenance* of an output tuple (a tuple in the output table) with job=j_1 and sal=s_1 is exactly those tuples in the input table with job and sal equal j_1 and s_1 respectively, but not any other tuples.

To leverage the provenance of tuples, observe that if we have already examined in the table search space that neither Bonus$_1$ nor Bonus$_2$ in Figure 1 is an input table that satisfies the test condition for the given @x (e.g., when $@\texttt{x} = 4$), we don't need to check Bonus$_3$. Meanwhile, if query q returns a non-empty output when applied to Bonus$_3$, then at least one of Bonus$_1$ or Bonus$_2$ would be a desirable input when searching in the space of all tables containing only one unique tuple. For example, if $@\texttt{x} = 10$, Bonus$_1$ is a valid unit test input, then smaller input tables such as Bonus$_2$ and Bonus$_3$ are valid test inputs as well.

Given this insight, we need to consider only tables containing k tuples where all tuples share the same job and sal values. The SQL solver only needs to find one concrete value for each field, rather than $2k$ different values for both fields in the table. As we will show in section 6, this reduction dramatically accelerates the test generation process, especially when the bound k is large.

Motivating Example 2. For bounded verification [Chu et al. 2017a] or query disambiguation [Chandra et al. 2015], the goal is either to prove the equivalence of two queries, q_1 and q_2, within a bounded space (i.e., q_1 and q_2 always return the same output when applied to the same set of input tables in the space), or to construct input tables that distinguish q_1 from q_2 (i.e., q_1 and q_2 return different results when evaluated on the same constructed distinguishing input tables).

Figure 2 shows two semantically equivalent queries, q_1 and q_3, and query q_2, which is inequivalent to them. Query q_1 first filters the input table `Bonus` by `sal > 5`, then groups the result by `job` and `dept` to calculate the maximum `sal` for each group, and finally keeps only the groups with `job` values less than 10. Query q_3 differs from q_1 only in its order of evaluating the filter predicate `job < 10` and grouping. Since the predicate refers to only columns appeared in the Group-By clause (i.e., `job`), q_1 and q_3 are equivalent. Query q_2 is semantically different from them since it groups tuples only by `job`, not by both `job` and `dept`.

```
-- q1                    -- q2                      -- q3
Select job, dept,        Select job, Max(dept),     Select job, dept,
       Sum(sal)                 Sum(sal)                   Sum(sal)
From Bonus               From Bonus                 From Bonus
Where sal > 5            Where sal > 5              Where sal > 5
Group By job, dept       Group By job                 And job < 10
Having job < 10;         Having job < 10;           Group By job, dept;
```

Fig. 2. Three queries with the relation $q_1 \equiv q_3 \not\equiv q_2$. Query q_3 differs from q_1 only by evaluating the filter `job < 10` before grouping. Since the predicate `job < 10` only refers to columns appeared in the Group By clause, this transformation is semantics preserving. q_2 is semantically different from q_1 since it groups tuples only by `job` but not both `job` and `dept`.

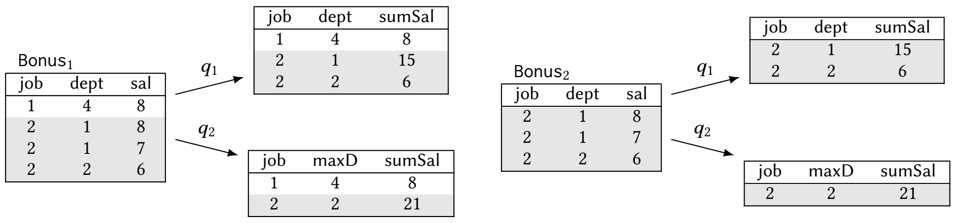
To check the equivalence between queries q_1, q_3 within the bounded space of all tables with at most k tuples, the solver first encodes the search space as a symbolic table and queries the underlying SMT solver to check whether q_1 and q_3 always produce the same output when applied to the symbolic input. In this example, the SQL solver faces the challenge to reason about grouping and aggregation: it needs to consider all 2^k possibilities of groups in both queries (or 2^k number of possibilities to partition the input table according to the grouping keys). Verification time increases exponentially as the bound increases.

Fortunately, we can also refine the search space in a way similar to that in the first case, after realizing that both queries do not introduce interplay among different (`job`, `dept`) groups during evaluation: different groups are aggregated independent of each other. In other words, each output tuple depends only on tuples in the input table belonging to the same (`job`, `dept`) group: the provenance of tuple $(job_1, dept_1, sumSal_1)$ are tuples in the input table satisfying `job` $= job_1$ and `dept` $= dept_1$).

Thus, we can restrict the search space to just tables with one (`job`, `dept`) group and restrict all tuples to satisfy `job < 10` and `sal > 5` (since no other tuples would pass through the filter to the output). The key insight behind this refinement process is this: assuming there exists an input table $\texttt{Bonus}_1$ such that q_1 and q_3 return different outputs T_{out1} and T_{out2} when applied to it, then there must be a tuple t whose multiplicity in T_{out1} differs from its multiplicity in T_{out2}. Then, we can also construct another input table $\texttt{Bonus}_2$, one that contains only tuples in $\texttt{Bonus}_1$ whose `job` and `dept` are the same as those in the tuple t, to reproduce the difference between q_1 and q_3. Apparently, since $\texttt{Bonus}_2$ contains just one (`job`, `dept`) group, it is included in the refined search space. On the other hand, if we prove that the two queries are equivalent in the refined search space, we also prove their equivalence in the original search space.

Similarly, to find a distinguishing input that distinguishes q_2 from q_1 in Figure 3, we can refine the search space in a similar fashion. Note that while q_2 could introduce interplay among tuples belonging to different (`job`, `dept`) groups since it groups only by `job` key (unlike q_1 and q_3), q_2 will not introduce interplay among tuples belonging to different `job` groups during evaluation (i.e., tuples in the input table belonging to different `job` groups won't be aggregated to the same tuple in the output). Thus, although we cannot restrict the new search space to tables with only one (`job`, `dept`) group, we can restrict it to tables consisting of tuples within one `job` group that satisfy `job < 10`, `sal > 5`. For example, $\texttt{Bonus}_1$ is a distinguishing input that distinguishes q_1 from q_2, as they both produce different results for tuples whose `job` = 2. Meanwhile, the other table $\texttt{Bonus}_2$ from the refined search space can also reproduce the difference between q_1 and q_2 based on their difference in multiplicities of tuples in the group with `job` equals 2. In fact, tuples in $\texttt{Bonus}_2$ are provenance tuples collected from $\texttt{Bonus}_1$ for the distinguishing output tuple $(2, 2, 8)$, which explains why $\texttt{Bonus}_2$ can reproduce the difference in multiplicities of $(2, 2, 8)$ in both query outputs, as does $\texttt{Bonus}_1$.

Since the refined search space no longer contains a table with multiple `job` groups for the solver to consider, the solver avoids the exponential encoding of the search space with respect to the `job` column, which provides a speedup in solving the problem.



job	dept	sal
1	4	8
2	1	8
2	1	7
2	2	6

job	dept	sumSal
1	4	8
2	1	15
2	2	6

job	maxD	sumSal
1	4	8
2	2	21

job	dept	sal
2	1	8
2	1	7
2	2	6

job	dept	sumSal
2	1	15
2	2	6

job	maxD	sumSal
2	2	21

Fig. 3. The two concrete tables $\texttt{Bonus}_1, \texttt{Bonus}_2$ are both distinguishing inputs that show the difference between q_1 and q_2 in Figure 2. Different colors in the table indicates different provenance groups in the input table.

Our approach. As mentioned above, in this paper we introduce a systematic approach for identifying and utilizing provenance information to refine the search space to scale up symbolic SQL reasoning. The key insight of our search space refinement algorithm is that we can perform a *symbolic provenance analysis* of the input queries to identify which tuples in the symbolic table alone are sufficient to prove or disprove the verification condition. Throughout this analysis, we construct a predicate ϕ that describes which tuples of an input table T are responsible for data generation or bounded verification.

We then use the provenance predicate ϕ to refine the original search space $\mathcal{S}$ into a new search space $\mathcal{S}'$ that is equivalent to $\mathcal{S}$ in terms of the verification condition: if there exists an input table $T \in \mathcal{S}$ satisfying the verification condition for the given queries, then there exists an input table $T' \in \mathcal{S}'$ that also satisfies the verification condition.

In particular, our provenance analysis uses only *the abstract semantics* of SQL to maintain efficiency, because computing the strongest provenance predicate for given queries requires full symbolic reasoning of the queries that can be as difficult as solving the reasoning task itself [Buneman et al. 2006]. For example, we over-approximate all aggregation functions as uninterpreted functions, so that the provenance tuples of tuple t_O (a tuple in the output table) are all tuples in the input table belonging to the same group as t_O (but in reality, certain aggregation functions like *max* depend only on the tuple in the input with the largest value). As we will show in section 5, such abstraction introduces a sound over-approximation of the query semantics that can be used to efficiently and soundly prune the search space for symbolic SQL reasoning. In the future, we could potentially redesign different abstractions to discover better trade-offs between analysis overhead and pruning effectiveness.

Furthermore, our space refinement process relies only on the semantics of the input queries but not the underlying symbolic reasoning tool's design and implementation, which allows it to be applied to speed up different symbolic SQL reasoning tools [Chu et al. 2017a; Shah et al. 2011; Veanes et al. 2010].

We evaluated the space refinement algorithm for three symbolic SQL reasoning scenarios: bounded verification (as utilized in verifying query rewriting rules), distinguishing input generation (for query disambiguation), and unit test generation (for testing frameworks). Using 61 real-world benchmarks, we compare the performance of SQL solvers using the search space with and without refinement. Results show that our refinement algorithm effectively speeds up the reasoning of a large class of queries used in real-world applications.

In sum, this paper makes the following contributions:

- We devised a new way to utilize the provenance of tuples to identify tables that are equivalent to each other with respect to the reasoning process of the given relational queries.
- We designed a space refinement algorithm that utilizes the provenance property to soundly prune the space of tables that need to be explored.
- We implemented our space refinement algorithm and evaluated it on various tools that apply symbolic reasoning to reason about relational queries. Results show that our search space refinement algorithm can effectively improve symbolic reasoning for SQL across different usage scenarios, and speeds up to 100× SQL solver reasoning for complex queries with aggregation.

We next review symbolic SQL reasoning (section 2), describe our approach with a query equivalence checking example (section 3), then formally introduce our space refinement algorithm (section 4, section 5), and finally evaluate our algorithm in the context of SQL equivalence checking and test generation (section 6).

2 PROBLEM DEFINITION

We start out by briefly reviewing backgrounds in symbolic SQL reasoning and defining the space refinement problem.

Table and SQL. Table is the first class value in SQL consisting of a schema and a bag of tuples [Negri et al. 1991].[2] A table schema defines the number of columns and the type of each column. A tuple t is a list of values with the same size as the schema. In our paper, we use $[\![T]\!]t$ to denote computing the multiplicity of t in table T: if t is in T, it returns the number of times t appears; otherwise, it returns 0. SQL queries are functions over tables, and we use $[\![q(\overline{T})]\!]$ to represent evaluating q against a list of input tables $\overline{T}$ (the result $[\![q(\overline{T})]\!]$ is a table T_{out}).

Under bag semantics, two tables are equal if and only if every tuple in them has the same multiplicities. Formally, table equality can be defined as $T_1 = T_2 \iff \forall t. [\![T_1]\!]t = [\![T_2]\!]t$. Two queries q_1, q_2 are equivalent if and only if evaluating them returns the same results for *all* possible input tables that are compatible with the schema. This equivalence relation can be defined as $q_1 \equiv q_2 \iff \forall \overline{T}.\ [\![q_1(\overline{T})]\!] = [\![q_2(\overline{T})]\!]$.

Symbolic SQL Reasoning. In this paper, we focus on SQL reasoning tasks in the form of finding input tables $\overline{T}_{\text{in}}$ for a given query q (or queries q_1, q_2) such that the output table $[\![q(\overline{T}_{\text{in}})]\!]$ satisfies a property in the form of $\Psi(T_{\text{out}}) = \exists t_O.\psi(t_O, T_{\text{out}})$, where the only use of T_{out} in ψ is to check multiplicity of t_O (i.e., used as $[\![T_{\text{out}}]\!]t_O$). This problem is solved by finding the satisfiability problem of $\exists \overline{T}_{\text{in}}.\Psi([\![q(\overline{T}_{\text{in}})]\!])$ using the formula Ψ above.

[2]There are other semantics of SQL, e.g., set semantics, but bag semantics is the most commonly implemented in modern commercial database systems.

- This formula restriction prevents us from reasoning about properties like "the output table T_{out} has exactly 5 tuples" where $\Psi(T_{\text{out}}) = (|T_{\text{out}}| = 5)$, or "every tuple in T_{out} has the same multiplicity" where $\Psi(T_{\text{out}}) = \exists m \forall t_O \in T_{\text{out}}.(\llbracket T_{\text{out}} \rrbracket t_O = m)$.
- On the other hand, we can still use the formula Ψ to express many practical reasoning tasks. For unit test generation, the property $\Psi(T_{\text{out}}) = \exists t_O.\llbracket T_{\text{out}} \rrbracket t > 0$ (there exists a tuple in the output table with multiplicity > 0). For equivalence checking, the property $\Psi(T_{\text{out1}}, T_{\text{out2}}) = \exists t_O.(\llbracket T_{\text{out1}} \rrbracket t_O \neq \llbracket T_{\text{out2}} \rrbracket t_O)$, i.e., exists a tuple t_O has different multiplicities in two query outputs.

As we will show later, our algorithm exploits the fact the we only need to find one t_O that satisfies the property ψ to witness the satisfaction of the property Ψ to conduct search space refinement.

Search Space Refinement. In this paper, we define the search space refinement problem as follows. Given a query q (or queries q_1, q_2), a property $\Psi(T_{\text{out}})$ and a search space $\mathcal{S}$ of input tables, we want to find a new search space $\mathcal{S}'$ such that $\mathcal{S}'$ is equivalent to $\mathcal{S}$: i.e., if there exists $\overline{T}_{\text{in}} \in \mathcal{S}$ s.t. $\Psi(\llbracket q(\overline{T}_{\text{in}}) \rrbracket)$ holds, then there exists $\overline{T}'_{\text{in}} \in \mathcal{S}'$ that also satisfies the property Ψ. Our goal is to construct $\mathcal{S}'$ such that $\mathcal{S}'$ is smaller than $\mathcal{S}$ and can be explored faster by the SQL solver.

In the rest of this paper, we explain our approach using the query equivalence checking problem (i.e., checking whether two queries q_1 and q_2 are semantically equivalent within some given space $\mathcal{S}$) as an illustrative example of reasoning tasks of the form $\Psi(T_{\text{out}}) = \exists t_O.\psi(t_O, T_{\text{out}})$. Given two queries q_1 and q_2, we call $\overline{T}_{\text{in}}$ a counterexample if $\llbracket q_1(\overline{T}_{\text{in}}) \rrbracket \neq \llbracket q_1(\overline{T}_{\text{in}}) \rrbracket$. If q_1 and q_2 are semantically inequivalent, then there must exist at least one tuple t such that $\llbracket q_1(\overline{T}_{\text{in}}) \rrbracket t \neq \llbracket q_2(\overline{T}_{\text{in}}) \rrbracket$ (i.e., its multiplicity differs in the outputs of q_1 and q_2). We call such t a *distinguishing output tuple* that demonstrates the semantic difference between the two queries.

3 OVERVIEW

We now use a concrete example for query equivalence checking to walk through our space refinement algorithm. As shown in Figure 4, our space refinement algorithm takes as input two queries q_1, q_2 and a search space $\mathcal{S}$, and it outputs the search space after refinement $\mathcal{S}'$ that is equivalent to $\mathcal{S}$ for the purpose of checking q_1 and q_2.

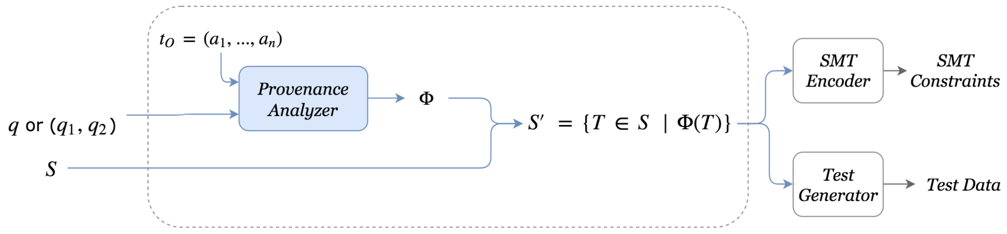


Fig. 4. The search space refinement algorithm (dotted box). The algorithm takes as input a query q (or two queries q_1, q_2) and a space of tables $\mathcal{S}$ for space refinement. It outputs a refined search space $\mathcal{S}'$ that is equivalent to $\mathcal{S}$. $\mathcal{S}'$ can be used in bounded verification or test data generation. Internally, the predicate Φ is computed using backward provenance analysis of the given query using a symbolic output tuple $t_O = (a_1, ..., a_n)$. The predicate is then used by the space refinement module to refine $\mathcal{S}$ into $\mathcal{S}'$.

To compute $\mathcal{S}'$, our algorithm contains the following two main modules:

- *(Provenance Analysis)* The provenance analysis process derives a predicate ϕ from the queries q_1, q_2 that describes a condition such that if two tables in the search space $\mathcal{S}$ satisfy it, they

The decomposition of q_1	Analysis steps	Provenance predicates
	(1) initialize	$\phi_{10}(t) = (t.\text{job} = t_O.\text{job} \wedge t.\text{dept} = t_O.\text{dept} \wedge t.\text{sum} = t_O.\text{sum})$
$q_1 = \text{Select}(q_{11}, \text{job} < 10)$	(2) $q_1 \to q_{11}$	$\phi_{11}(t) = (t.\text{job} = t_O.\text{job} \wedge t.\text{dept} = t_O.\text{dept} \wedge t.\text{sum} = t_O.\text{sum} \wedge t.\text{job} < 10)$
$q_{11} = \text{Aggr}(q_{12}, [\text{job}, \text{dept}], \text{Sum}(\text{sal}))$	(3) $q_{11} \to q_{12}$	$\phi_{12}(t) = (t.\text{job} = t_O.\text{job} \wedge t.\text{dept} = t_O.\text{dept} \wedge t.\text{job} < 10)$
$q_{12} = \text{Select}(T, \text{sal} > 5)$	(4) $q_{12} \to T$	$\phi_1(t) = (t.\text{job} = t_O.\text{job} \wedge t.\text{dept} = t_O.\text{dept} \wedge t.\text{job} < 10 \wedge t.\text{sal} > 5)$

$$t_O \longleftarrow [\![q_1(T)]\!]^{\phi_{10}} \longleftarrow [\![q_{11}(T)]\!]^{\phi_{11}} \longleftarrow [\![q_{12}(T)]\!]^{\phi_{12}} \longleftarrow T^{\phi_1}$$

Fig. 5. The analysis process of which tuples in input T contribute to the multiplicity of t_O in the output $[\![q(T)]\!]$. The analysis result is shown as a chain: for each arrow, the right hand side tuples evaluated from each subexpression determine the multiplicities of tuples on the left hand side. We use $[\![q(T)]\!]^{\phi}$ to denote tuples in $[\![q(T)]\!]$ satisfying the predicate ϕ. For example, $[\![q_1(T)]\!]^{\phi_{10}} \longleftarrow [\![q_{11}(T)]\!]^{\phi_{11}}$ indicates that tuples in $[\![q_{11}(T)]\!]$ satisfying ϕ_{11} determines the multiplicities of tuples in $[\![q_1(T)]\!]$ satisfying ϕ_{10}.

would be equivalent with respect to finding an distinguishing output tuple t_O. The predicate ϕ is computed by combining the provenance predicate ϕ_1 for t_O in q_1 (ϕ_1 determines which tuples in the input table T contribute to the multiplicity of the output tuple t_O) and the provenance predicate ϕ_2 for t_O in q_2.

- *(Space Refinement)* Using the provenance predicate ϕ, we construct a refinement predicate Φ describing what properties a table T need to satisfy to refine the search space and then use it to construct the refined search space $\mathcal{S}'$ from $\mathcal{S}$.

Example. We reuse the equivalence checking example between q_1 and q_2 shown in Figure 2 to demonstrate the space refinement algorithm. Given the queries q_1 and q_2 in Figure 2 and a search space $\mathcal{S}$, our goal is to determine whether q_1 and q_2 are equivalent within $\mathcal{S}$. As mentioned in section 1, since the two queries use different grouping keys, they are inequivalent, and our goal is to find a counterexample for them. An counterexample is shown in Figure 3. Note that both queries operate on input tables with schema Bonus(job, dept, sal).

3.1 Provenance Analysis

As shown in Figure 4, the first step is to perform a provenance analysis to determine which tuples in T contribute to the multiplicity of t_O in the outputs of q_1 and q_2 ($[\![q_1(T)]\!]$ and $[\![q_2(T)]\!]$). We start out by analyzing the provenance of t_O with regard to q_1.

We first convert the query into the sequential expressions shown in Figure 5 (left), where q_1 is computed from its subexpression q_{11}, $q_1 1$ is computed from q_{12}, and q_{12} is directly computed from input table T. The operator Select(q, f) represents filtering the result of q using the predicate f (for Having and Where) and Aggr represents the SQL operator Group By. The key idea behind provenance analysis process is as follows: given an expression $q = \text{op}(q')$ where q' is a subexpression of q, the key is to determine which tuples in $[\![q'(T)]\!]$ are sufficient to determine the multiplicities of (or, "contribute to") the tuples in $[\![q(T)]\!]$ that we are interested in. The goal is to "propagate" the deduced provenance relation from the outermost query (e.g., q_1) to the input table T, and subsequently to represent the provenance information as a predicate over tuples. The concrete analysis process is as follows:

- *(Step 1).* We start the analysis process by analyzing which tuples in $[\![q_1(T)]\!]$ contribute to the multiplicity of the output tuple t_O. Obviously, such tuples should have the same values for each attribute as t_O, and they can be represented as all tuples in $[\![q(T)]\!]$ satisfying the predicate $\phi_{10}(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{sum} = t_O.\mathsf{sum})$ ($t.c$ stands for referencing column c in t). We denote these tuples as $[\![q(T)]\!]^{\phi_{10}}$.
- *(Step 2).* For the expression $q_1 = \mathsf{Select}(q_{11}, \mathsf{job} < 10)$, we analyze which tuples in the subexpression result $[\![q_{11}(T)]\!]$ determine multiplicities of tuples in $[\![q(T)]\!]^{\phi_{10}}$. Since the latter determines the multiplicity of t_O in $[\![q_1(T)]\!]$, we can use this analysis result to propagate the provenance information one level back. Since q_1 contains the filter predicate $\mathsf{job} < 10$, only tuples in $[\![q_{11}(T)]\!]$ and satisfying both $\phi_{10}(t)$ and $\mathsf{job} < 10$ would contribute to $[\![q(T)]\!]^{\phi_{10}}$. Thus, we derive from ϕ_{10} and q_1 the predicate $\phi_{11}(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{sum} = t_O.\mathsf{sum} \wedge t.\mathsf{job} < 10)$ to describe these tuples in $[\![q_{11}(T)]\!]$; we denote them as $[\![q_{11}(T)]\!]^{\phi_{11}}$.
- *(Step 3).* Similarly, we analyze which tuples in $[\![q_{12}(T)]\!]$ contribute to $[\![q_{11}(T)]\!]^{\phi_{11}}$. Since q_{11} is an aggregation query that groups by the job and dept columns, each tuple t in $[\![q_{11}(T)]\!]^{\phi_{11}}$ is determined by all tuples in $[\![q_{12}(T)]\!]$ with the same job and dept values as t. Since ϕ_{11} states that target entries in $[\![q_{11}(T)]\!]^{\phi_{11}}$ are those belonging to the group $t_O.\mathsf{job}, t_O.\mathsf{dept}$ with $\mathsf{job} < 10$, we derive the predicate $\phi_{12}(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{job} < 10)$ to describe this group in $[\![q_{12}(T)]\!]$. The result $[\![q_{12}(T)]\!]^{\phi_{12}}$ then contains all tuples contributing to $[\![q_{11}(T)]\!]^{\phi_{11}}$.
- *(Step 4).* Last, we analyze which tuples in T contribute to $[\![q_{12}(T)]\!]^{\phi_{12}}$ from the expression $q_{12} = \mathsf{Select}(T, \mathsf{sal} > 5)$. Similar to step 2, the desired tuples are those satisfying both $\mathsf{sal} > 5$ and ϕ_{12}. We use the predicate $\phi_1(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5)$ to describe these target tuples.

Figure 5 summarizes the relationship between these predicates: T^{ϕ_1} determines $[\![q_{12}(T)]\!]^{\phi_{12}}$, $[\![q_{12}(T)]\!]^{\phi_{12}}$ determines $[\![q_{11}]\!]^{\phi_{11}}$, $[\![q_{11}]\!]^{\phi_{11}}$ determines $[\![q_1(T)]\!]^{\phi_{10}}$, and $[\![q_1(T)]\!]^{\phi_{10}}$ determines the multiplicity of t_O in the output $[\![q_1(T)]\!]$. It indicates that ϕ_1 specifies the provenance of t_O in T with respect to q_1.

Similarly, we also analyze q_2 using the same output tuple t_O to obtain the predicate $\phi_2(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5)$ describing the provenance of t_O in T with respect to q_2.

3.2 Space Refinement

After computing ϕ_1 and ϕ_2, we combine them to find the provenance of t_O with respect to both queries: the provenance tuples are those from T contained by both T^{ϕ_1} and T^{ϕ_2}. We can use the following ϕ to represent them.

$$\begin{aligned}
\phi(t) &= \phi_1(t) \vee \phi_2(t) \\
&= (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5) \\
&\quad \vee (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5) \\
&= (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5)
\end{aligned}$$

Since t_O used in the analysis is a symbolic tuple, the generated predicate ϕ generalizes to all possible instances of an output tuple. Thus, for any instance of t_O, T^{ϕ} specifies which tuples in T matters for the equivalence checking of q_1 and q_2 with respect to it: if t_O is a distinguishing output tuple for the two queries q_1, q_2 when evaluated on T, then T^{ϕ} is sufficient to replay this difference. In other words, any two input table T_1, T_2 sharing the same fragments that satisfy ϕ would be equivalent with respect to discovering the distinguishing output tuple t_O.

Using ϕ, we construct a new search space $\mathcal{S}'$ from $\mathcal{S}$ such that no two tables share the same provenance tuples for any instances of t_O. This space can be constructed by including only tables T such that $T^{\phi} = T$, so that $T_1^{\phi} = T_2^{\phi} \Rightarrow T_1 = T_2$ for each t_O. The new search space is defined using the following predicate Φ over tables derived from ϕ. (We use $\phi(t, t_O)$ to denote instantiating t, t_O with the provided arguments.)

$$\begin{aligned}\Phi(T) &= \exists t_O.\forall t \in T.\ \phi(t, t_O)\\ &= \exists t_O.\forall t \in T.\ (t.\text{job} = t_O.\text{job} \land t.\text{job} < 10 \land t.\text{sal} > 5)\end{aligned}$$

The predicate $\Phi(T)$ specifies that: (1) the table should contain only one job group, and (2) for each tuple t, $t.\text{job} < 10 \land t.\text{sal} > 5$. The new search space $\mathcal{S}'$ is then defined as $\{T \mid T \in \mathcal{S} \land \Phi(T)\}$. On the other hand, for each table $T \in \mathcal{S}$, T^{ϕ} is contained in $\mathcal{S}'$ for all t_O. Thus, if we can find a counterexample T in $\mathcal{S}$ with distinguishing output tuple t_O, we can also find T^{ϕ} in $\mathcal{S}'$ to reveal the same distinguishing output tuple. Thus, $\mathcal{S}'$ is a smaller yet equivalent space for checking the equivalence between q_1 and q_2.

Figure 3 illustrates the relationship between the new search space $\mathcal{S}'$ and $\mathcal{S}$. T_1 and T_2 are both counterexamples for q_1 and q_2 from $\mathcal{S}$, and they both generate the distinguishing output tuple $t_O = (2, 2, 21)$. Since $\mathcal{S}'$ disallows tuples with multiple groups in the job column, we do not need to consider T_1 in the equivalence checking process. This indicates we can use $\mathcal{S}'$ to speed up the equivalence checking for q_1 and q_2 due to its smaller size.

The refined search space $\mathcal{S}'$ can then be used to encode SMT formulas for bounded verification to determine query equivalence. For instance, encoding $\mathcal{S}'$ along with the queries to SMT formulas and solving them will show that q_1 and q_2 are indeed inequivalent.

4 DEFINITIONS

In this section we formally define SQL operators and the predicate language we use to describe provenance.

The SQL Language. Figure 6 shows the abstract syntax of SQL. A SQL query is formed by compositions of basic operators including Projection, Distinct (for de-duplication), Select (for filtering), Join, Aggr, LeftJoin, Union and Rename on top of base tables. The constructor Aggr$(\bar{c}, \bar{\alpha}, \bar{c}_t, q)$ corresponds to the aggregation query "Select $\bar{c}$, $\overline{\alpha(c_t)}$ From q Group-By $\bar{c}$ ", where $\bar{c}$ are group by keys, $\bar{c}_t$ are columns involved in aggregation, and $\bar{\alpha}$ are aggregation functions used for each column. The constructor Proj$(\bar{c}, q)$ corresponds to the projection query "Select $\bar{c}$ From q", Distinct(q) corresponds to "Select Distinct $*$ From q", and others directly corresponds to their concrete forms.

$$\begin{array}{rcll}
q & ::= & T \mid \text{Proj}(\bar{v}, q) \mid \text{Rename}(q, name, \bar{c}) \mid \text{Select}(q, f) \mid \text{Join}(q_1, q_2) & \text{(Query)}\\
 & \mid & \text{Aggr}(\bar{c}, \bar{\alpha}, \bar{c}_t, q) \mid \text{Distinct}(q) \mid \text{Union}(q_1, q_2) \mid \text{LeftJoin}(q_1, q_2, f) & \\
f & ::= & \text{True} \mid \text{False} \mid v\ op\ v \mid f \text{ And } f \mid f \text{ Or } f \mid \text{Not } f & \text{(Filters)}\\
v & ::= & c \mid const \mid \text{null} \mid \text{expr}(\bar{v}) & \text{(Values)}\\
\alpha & ::= & \text{Max} \mid \text{Min} \mid \text{Sum} \mid \text{Count} \mid \text{Count-Distinct} & \text{(Aggregators)}\\
op & ::= & = \mid > \mid < \mid <= \mid >= \mid <> & \text{(Operators)}
\end{array}$$

Fig. 6. The abstract syntax of SQL, where c ranges over column names, v over values, q over queries and f over filter predicates. We use expr for arithmetic operations.

Predicates. Figure 7 defines the predicate language for describing data provenance. A predicate ϕ is formed from compositions of primitives $v\ op\ v$, where a value v is either a reference of a tuple ($t.i$ represents the i-th element in the tuple t, $t.c$ is the same but refers the tuple entry by name c), an expression composed from operators like $+$, $-$ or function application. In our paper, symbolic tuples in predicates are allowed, and we use the notation $\phi(t_1, t_2)$ to denote substituting symbolic tuples with input tuples t_1, t_2.

$$\begin{array}{rcl} \phi & := & \text{true} \mid \text{false} \mid \phi \land \phi \mid \phi \lor \phi \mid \neg\phi \mid v\ op\ v \\ v & := & t.i \mid t.c \mid const \mid \text{expr}(\bar{v}) \\ op & := & = \mid > \mid < \mid <= \mid >= \mid <> \end{array}$$

Fig. 7. The predicate language, where t ranges over tuples and c over column names.

We also use the notation T^{ϕ} to denote filtering a table by keeping only those tuples in T that satisfy $\phi(x)$. Formally, $T^{\phi} = \{\!\{t \mid t \in T \land \phi(t)\}\!\}$ (we use $\{\!\{\cdot\}\!\}$ to represent a bag).

5 ALGORITHM

We next formally describe the space refinement algorithm. First, we introduce the provenance analysis algorithm (subsection 5.1). Then, we present how we use the analysis result to conduct search space refinement for the query equivalence checking problem (subsection 5.2). Without loss of generality, we assume queries refer only to one input table T to simplify notation.

5.1 Symbolic Provenance Analysis

Given a query q, the provenance analysis algorithm computes the provenance of a symbolic tuple t_O with respect to q. Taking q and t_O as input, the analysis returns a predicate ϕ describing which tuples in input table T contribute to the multiplicity of t_O in the query output.

Definition 5.1. (Provenance Predicate) ϕ is a provenance predicate for query q if the following property is satisfied:

$$\forall t_O.\forall T.\ [\![q(T)]\!]t_O = [\![q(T^{\phi(t,t_O)})]\!]t_O.$$

In other words, the multiplicity of the tuple t_O in the output of q is unchanged even though input table T is restricted to only tuples t that satisfy property $\phi(t, t_O)$. By definition, a query q has multiple provenance predicates, including the trivial predicate true. To compute a non-trivial predicate, we perform a backward analysis on q by propagating constraints from a tuple t_O in the query output back to the input, as shown in section 3.

5.1.1 The Algorithm. Key to the backward analysis is constructing a predicate ϕ that describes which tuples in T contribute to the multiplicity of the output tuple $t_O \in [\![q(T)]\!]$. To do so, the backward analysis first constructs a predicate over q to specify which tuples in q determine the multiplicity of t_O in the output and then propagates it to its subexpressions until reaching the input table T (the leaf subexpression).

Provenance Analysis. We compute the provenance predicate of a query q with respect to a symbolic output tuple $t_O \in [\![q(T)]\!]$ in the following three steps:

(1) *(Initialization).* Construct the predicate ϕ_0 to be $\phi_0 = \bigwedge_{i=1}^{n}(t.i = t_O.i)$. This initial predicate states that the provenance of t_O in the output table $[\![q(T)]\!]$ is itself.

(2) *(Propagation).* For each query $q_1 = \text{op}(q_2)$ (or $q_1 = \text{op}(q_2, q_3)$) where op is a SQL operator defined in Figure 6, we propagate the provenance predicate ϕ_1 over q_1 to its subquery q_2. The result is a predicate ϕ_2 over q_2 specifying which tuples in $[\![q_2(T)]\!]$ are sufficient to decide multiplicities of tuples in $[\![q_1(T)]\!]^{\phi_1}$. Propagation rules are shown in Figure 8.
- We use the notation $(q_1 \sim \phi_1)$ to describe that ϕ_1 is the provenance predicate computed at the query q_1 (i.e., $[\![q_1(T)]\!]^{\phi_1}$ is the provenance of the output tuple t_O with respect to $[\![q_1(T)]\!]$), and we use $(q_1 \sim \phi_1) \rightsquigarrow (q_2 \sim \phi_2)$ to describe the propagation of ϕ_1 over q_1 to the predicate ϕ_2 over the subquery q_2.
- Given a query $q_1 = \text{op}(q_2, q_3)$ and a predicate ϕ_1 over q_1, the rule $(q_1 \sim \phi_1) \rightsquigarrow (q_2 \sim \phi_2) \land (q_3 \sim \phi_3)$ produces ϕ_2 and ϕ_3 specifying which tuples in $[\![q_1(T)]\!]$ and $[\![q_2(T)]\!]$ determine

multiplicities of tuples in $[\![q_1(T)]\!]^{\phi_1}$. The conjunction states that $[\![q_2(T)]\!]^{\phi_2}$ and $[\![q_3(T)]\!]^{\phi_3}$ together determine $[\![q_1(T)]\!]^{\phi_1}$.

The propagation process terminates when all queries in the expression are T (i.e., reaching leaf nodes of the AST), since the remaining AST depth of the algorithm decreases over analysis steps. The propagation process can be presented as $(q \sim \phi_0) \rightsquigarrow \cdots \rightsquigarrow \bigwedge_k (T \sim \phi_k)$, where the final state shows which tuples in T determine multiplicities of tuples in $[\![q(T)]\!]^{\phi_0}$. The conjunction results from the fact that table T may appear multiple times in different subqueries of q.

(3) *(Merge).* The final step is to resolve the expression $\bigwedge_k (T \sim \phi_k)$ to obtain a provenance predicate ϕ over T s.t. T^{ϕ} determines $[\![q(T)]\!]^{\phi_0}$. According to the merge rule (Figure 9), we construct ϕ as $\phi = \bigvee_k \phi_k$. The correctness of ϕ is shown by Lemma 5.4.

Figure 8 and Figure 9 show the analysis rules. We describe these rules below.

- *(Select).* Given a query $q = \mathsf{Select}(q_1, f)$ and a predicate ϕ, the predicate ϕ_1 over q_1 is the conjunction of ϕ with a predicate formed by replacing every column reference c_i in the filter predicate f with $t.i$. The idea is that tuples in $[\![q_1(T)]\!]$ satisfying $f \wedge \phi$ alone are sufficient to determine $[\![q(T)]\!]^{\phi}$. For example, if $q = \mathsf{Select}(q_1, c_1 < 5)$ and $\phi = (t.2 > 1)$, then the tuples in $[\![q_1(T)]\!]$ satisfying $\phi_1 = (t.2 > 1 \wedge t.1 < 5)$ determine tuples in $[\![q(T)]\!]$ satisfying ϕ.
- *(Projection).* Given a query $q = \mathsf{Proj}(\bar{v}, q_1)$ and a predicate ϕ, we compute the predicate ϕ_1 by substituting column references in ϕ with expressions specified by $\bar{v}$, followed by abstracting column names using tuple expression $t.i$. For example, given a query $q = \mathsf{Proj}(c_3 + 2, c_1,\ q_1)$ (corresponding to Select $c_3 + 2, c_1$ From q_1) and $\phi = (t.1 = 1 \wedge t.2 > 1)$, the output predicate ϕ_1 for q_1 is $\phi_1 = (t.3 + 2 = 1 \wedge t.1 > 1)$. The rationale is that the multiplicity of the tuple (x_1, x_2) in $[\![q(T)]\!]$ is determined by all tuples t in $[\![q_1(T)]\!]$ such that $t.3 + 2 = x_1$ and $t.1 = x_2$.
- *(Join).* To propagate ϕ through $q = \mathsf{Join}(q_1, q_2)$, the rule breaks ϕ into $\phi_1 \wedge \phi_2 \wedge \phi_3$, where ϕ_1 contains terms that involve only columns from q_1, ϕ_2 contains terms that involve only columns from q_2, and ϕ_3 contains terms that that involve both. Then, ϕ_1 is propagated to q_1, ϕ_2 is propagated to q_2, and ϕ_3 is discarded. Discarding ϕ_3 allows the analysis continues independently in different branches, thus reducing the complexity of the resulting predicate. This approximation retains the soundness of the analysis: since $[\![q_1(T)]\!]^{\phi_1} \times [\![q_2(T)]\!]^{\phi_2}$ subsumes $\left([\![q_1(T)]\!]^{\phi_1} \times [\![q_2(T)]\!]^{\phi_2}\right)^{\phi_3}$ and the latter is sufficient to determine $[\![q(T)]\!]^{\phi}$, $[\![q_1(T)]\!]^{\phi_1} \times [\![q_2(T)]\!]^{\phi_2}$ is sufficient to determine $[\![q(T)]\!]^{\phi}$ as well. On the other hand, discarding ϕ_3 weakens the provenance predicate we obtain in the final result, which could limit the pruning power of the final provenance predicate. We leave a detailed discussion of the trade-off between analysis performance and pruning power to the end of this section. For example, the propagation of $\phi = (t.1 = a_1 \wedge t.2 > t.3 \wedge r.3 < 5)$ through query $q = \mathsf{Join}(q_1, q_2)$ is computed by first constructing $\phi_1 = (t.1 = a_1)$ for q_1, $\phi_2 = (t.1 < 5)$ for q_2, and discarding $\phi_3 = (t.2 > t.3)$ (since it refers to columns from both subqueries).
- *(Aggregate).* To propagate a predicate ϕ over an aggregation query $q = \mathsf{Aggr}(\bar{c}, \bar{\alpha}, \bar{c}_t, q_1)$ to its subquery q_1, we first split ϕ into $\phi_1 \wedge \phi_2$, where column references in ϕ_1 are limited to grouping columns $\bar{c}$, and then set ϕ_1 as the target predicate for q_1. Similar to the rule for Join, this is an approximation of the aggregation semantics, it guarantees that all tuples in each group satisfy ϕ_1 are retained so that $[\![q_1(T)]\!]^{\phi_1}$ is sufficient to compute aggregation results in $[\![q(T)]\!]^{\phi}$. We discard ϕ_2 to retain analysis efficiency.
- *(Union, Rename).* The predicates ϕ_1, ϕ_2 for the subqueries are the same as ϕ.
- *(LeftJoin).* Unlike for Join, the predicate for the subquery q_2 in LeftJoin is set to true instead of ϕ_2. This difference is introduced by the non-monotonicity of Left Join, i.e., given T =

$$\boxed{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1) \land \ldots}$$

$$\frac{q = T}{(q \sim \phi) \rightsquigarrow (T \sim \phi)}\ (\text{Table}) \qquad \frac{q = \mathsf{Select}(q_1, f) \quad \mathsf{schema}(q_1) = \bar{c}}{(q \sim \phi) \rightsquigarrow \left(q_1 \sim \phi \land \left[c_i \mapsto t.i\right] f\right)}\ (\text{Select})$$

$$\frac{q = \mathsf{Distinct}(q_1)}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi)}\ (\text{Distinct}) \qquad \frac{q = \mathsf{Proj}(\bar{v}, q_1) \quad \mathsf{schema}(q_1) = \bar{c}}{(q \sim \phi) \rightsquigarrow \left(q_1 \sim \left[t.i \mapsto \left(\left[c_j \mapsto t.j\right] v_i\right)\right] \phi\right)}\ (\text{Proj})$$

$$\frac{\begin{array}{c} q = \mathsf{Join}(q_1, q_2) \quad \phi = \phi_1 \land \phi_2 \land \phi_3 \quad |\mathsf{schema}(q_1)| = n_1 \\ \mathsf{colRef}(\phi_i) \cap \mathsf{schema}(q_i) = \emptyset \ _{(i=1,2)} \end{array}}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1) \land (q_2 \sim [t.i \mapsto t.(i - n_1)]\phi_2)}\ (\text{Join})$$

$$\frac{q = \mathsf{Union}(q_1, q_2)}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi) \land (q_2 \sim \phi)}\ (\text{Union}) \qquad \frac{q = \mathsf{Rename}(q_1, \mathit{name}, \bar{c})}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1)}\ (\text{Rename})$$

$$\frac{q = \mathsf{LeftJoin}(q_1, q_2, f) \quad \phi = \phi_1 \land \phi_2 \land \phi_3 \quad \mathsf{colRef}(\phi_i) \cap \mathsf{schema}(q_i) = \emptyset \ _{(i=1,2)}}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1) \land (q_2 \sim \mathsf{true})}\ (\text{LeftJoin})$$

$$\frac{\begin{array}{c} q = \mathsf{Aggr}(\bar{c}, \bar{\alpha}, \bar{c}_t, q_1) \\ \phi = \phi_1 \land \phi_2 \quad \mathsf{colRef}(\phi_1) \subseteq \{\bar{c}\} \end{array}}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1)}\ (\text{Aggr}) \qquad \frac{(q_i \sim \phi_i) \rightsquigarrow \bigwedge_k \left(q'_{ik} \sim \phi'_{ik}\right)}{\bigwedge_i (q_i \sim \phi_i) \rightsquigarrow \bigwedge_{i,k} \left(q'_{ik} \sim \phi'_{ik}\right)}\ (\text{Step})$$

Fig. 8. Propogation rules. Each propagation step $(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1)$ computes the constraint ϕ_1 that specifies which tuples in $[\![q_1(T)]\!]$ are sufficient to determine the multiplicities of tuples in $[\![q(T)]\!]$ that satisfy constraint ϕ. The auxiliary function colRef returns the column a predicate refers to, and the function schema extracts the output schema of a query.

$$\frac{}{\bigwedge_i (T \sim \phi_i) \rightsquigarrow (T \sim \bigvee_i \phi_i)}\ (\text{Merge})$$

Fig. 9. The merge rule. It computes the provenance constraint of the table T by merging contraints obtained from backward analysis.

$[\![\mathsf{LeftJoin}(T_1, T_2)]\!]$, if we remove a tuple from T_2 (denoted as T_2^-), the result $[\![\mathsf{LeftJoin}(T_1, T_2^-)]\!]$ is not subsumed by T. The enforcement of using the predicate true for q_2 in our rule design is a conservative way to preserved all tuples contributing to tuples in $[\![q(T)]\!]$.

5.1.2 Properties of the Analysis Algorithm. We now demonstrate properties of the provenance analysis algorithm. Lemma 5.2 states that each step of the analysis correctly propagates the provenance predicate to its subquery (subqueries). Lemma 5.3 states that weakening any provenance predicate generated by the algorithm still results in a provenance predicate. The weakening property allows us to generate not necessarily the strongest but sound constraints, and it guarantees the correctness of the merge rule, where predicates ϕ_k's are weakened to their disjunction $\bigvee_k \phi_k$. Finally, Lemma 5.4 shows that the provenance analysis algorithm returns a provenance predicate over T for the symbolic output tuple t_O.

Lemma 5.2. *For each propagation rule $(q \sim \phi) \rightsquigarrow \bigwedge_k (q_k \sim \phi_k)$, the following property holds:*

$$\forall T, T'. \left(\bigwedge_k \left(\llbracket q_k(T) \rrbracket^{\phi_k} = \llbracket q_k(T') \rrbracket^{\phi_k}\right)\right) \Longrightarrow \left(\llbracket q(T) \rrbracket^{\phi} = \llbracket q(T') \rrbracket^{\phi}\right)$$

Proof Sketch. By design (details are shown in the rule explanations in subsection 5.1), every rule guarantees that tuples excluded from $\llbracket q_1(T) \rrbracket$ by ϕ_i do not contribute to multiplicities of tuples in $\llbracket q(T) \rrbracket^{\phi}$. □

Lemma 5.3 (Weakening). *For each propagation rule $(q \sim \phi) \rightsquigarrow \bigwedge_k (q_k \sim \phi_k)$, if $\forall t.(\phi_k \Rightarrow \phi'_k)$ holds for all k, then the following property holds:*

$$\forall T, T'. \left(\bigwedge_k \left(\llbracket q_k(T) \rrbracket^{\phi'_k} = \llbracket q_k(T') \rrbracket^{\phi'_k}\right)\right) \Longrightarrow \left(\llbracket q(T) \rrbracket^{\phi} = \llbracket q(T') \rrbracket^{\phi}\right)$$

Proof Sketch. The weakening property for monotonic queries operators (Select, Proj, Join) is obvious, since appending extra rows not satisfying the original constraint do not affect $\llbracket q(T) \rrbracket^{\phi}$. For non-monotonic operators Aggr, the rule ensures that the whole a whole group would either all be included or none get included for each Group-by group; therefore, relaxing the constraint ϕ_i does not introduce new entries in the same group. For LeftJoin, the rule disallows propagation of the predicate to the right hand side query q_2 (the predicate is true, which can no longer be weakened); therefore, no new tuples with null placeholders will be introduced in the result. □

Lemma 5.4 (Soundness). *For a given query q whose output schema size is n, let $\phi_0 = \bigwedge_{i=1}^{n}(t.i = t_O.i)$, assume $(q \sim \phi_0) \rightsquigarrow \cdots \rightsquigarrow \bigwedge_k (T \sim \phi_k)$ where t_O is a symbolic output tuple; then, $\phi = \bigvee_k \phi_k$ is a provenance predicate over T with respect to the tuple t_O in $\llbracket q(T) \rrbracket$.*

Proof Sketch. By induction on the propagation rules, we can apply Lemma 5.2 to show that $T^{\phi_1}, \ldots, T^{\phi_n}$ together determine the multiplicity of t_O in $\llbracket q(T) \rrbracket$. Then by the weakening property, we can weaken every ϕ_i to $\bigvee_k \phi_k$ while retaining soundness. Thus, $\bigvee_k \phi_k$ is a provenance predicate over T for $\llbracket q(T) \rrbracket^{\phi_0}$ (i.e., the tuple t_O in $\llbracket q(T) \rrbracket$). □

5.1.3 The Effect of Analysis Approximation. Note that our inference algorithm provides a sound provenance predicate but *does not* produce the strongest provenance predicate at each analysis step, i.e., the property in Lemma 5.2 does not hold if we flip the "$\Longrightarrow$" into "$\Longleftarrow$". This is because the propagation process does not use the full semantics of SQL in the analysis process for the purpose of improving analysis efficiency. For example, the Join propagation rule does consider join predicates that refer to columns in both tables (by discarding ϕ_3); as a result, $\llbracket q_1(T)^{\phi_1} \rrbracket$ and $\llbracket q_2(T)^{\phi_2} \rrbracket$ can include tuples that have no matching tuples in the other table. Similarly, the Aggr propagation rule does not consider the semantics of aggregation functions, and it conservatively keeps all tuples in $\llbracket q_1(T) \rrbracket$ that are in the same groups as tuples in $\llbracket q(T) \rrbracket^{\phi}$. This design decision trades-off between the cost of provenance analysis and the effectiveness of the generated search space: we could ask a solver to find the strongest provenance predicate for the tuple t_O in $\llbracket q(T) \rrbracket$, but its computation time would be the same as directly asking the solver to solve the equivalence problem, making the analysis pointless. As we will show in section 6, although the generated provenance predicate is not the strongest, it is highly effective in speeding up various symbolic SQL reasoning tasks.

5.2 Space Refinement

We now introduce how to refine the search space for the query equivalence checking problem between two queries q_1 and q_2 using their provenance predicates ϕ_1 and ϕ_2 over input table T.

Merge Predicates. We first merge ϕ_1 and ϕ_2 to form a provenance predicate that is sound for both tables using the MergeRule in Figure 9. Since ϕ_1 specifies which tuples in T decide the multiplicity of t_O in $[\![q_1(T)]\!]$, and ϕ_2 specifies which tuples in T decide the multiplicity of t_O in $[\![q_1(T)]\!]$, combining them yields a predicate $\phi = \phi_1 \vee \phi_2$ that is sufficient to determine the multiplicity of t_O in both outputs of q_1 and q_2, as guaranteed by the weakening property (Lemma 5.3). Formally, the merged predicate ϕ holds the following property (recall that ϕ is shorthand for $\phi(t, t_O)$, which is a function over t_O).

$$\forall T, t_O. \left([\![q_1(T)]\!]t_O = [\![q_1(T^\phi)]\!]t_O\right) \wedge \left([\![q_2(T)]\!]t_O = [\![q_2(T^\phi)]\!]t_O\right)$$

Space Refinement. Given a table T and an output tuple t_O, the provenance predicate ϕ is a predicate over tuples that determines which tuples in T are sufficient to decide the multiplicity of t_O in both query outputs. Next, we lift ϕ from a predicate over tuples to a predicate over tables to refine the search space. Specifically, we define:

$$\Phi(T) = \exists t_O . \forall t \in T \ . \ \phi(t, t_O)$$

as a *table predicate.* Given a table search space $\mathcal{S}$, we construct a new search space $\mathcal{S}'$ using Φ as follows:

- If the space $\mathcal{S}$ satisfies the property that $\forall T, T' \ . \ T' \in \mathcal{S} \wedge T \subseteq T' \Rightarrow T \in \mathcal{S}$ (i.e., $\mathcal{S}$ is closed under containment):
$$\mathcal{S}' = \{T \in \mathcal{S} \mid \Phi(T)\}$$
- Otherwise, we first construct the closure $\mathcal{S}^* = \{T \mid \exists T' \in \mathcal{S} \wedge T \subseteq T'\}$ and then apply the above with $\mathcal{S} = \mathcal{S}^*$.

Intuitively, the new search space $\mathcal{S}'$ contains only one representative from each equivalence class of tables with respect to input queries: i.e., for each instantiation of output tuple t_O, if $T_1^\phi = T_2^\phi$ (ϕ also instantiated with t_O), only one table remains in the new search space $\mathcal{S}'$. For the second case, we construct the closure of $\mathcal{S}^*$ since the fragment contributing to an output tuple may not be contained in $\mathcal{S}$. For example, if $\mathcal{S}$ contains only tables with exactly 2 distinct tuples (and not containing those with 1 tuple), the table fragment identified by a provenance predicate might consist of only one tuple and not be contained in $\mathcal{S}$.

As shown in section 6, we use Φ to identify equivalent tables in the search space $\mathcal{S}$ to speed up query equivalence checking. We formally state the relationship between $\mathcal{S}$ and $\mathcal{S}'$ below.

Lemma 5.5 (Removing Redundancy). *Given q_1, q_2, a search space $\mathcal{S}$, and $\mathcal{S}'$ is the refined search space constructed from $\mathcal{S}$ using a table constraint. If there exists a table $T \in \mathcal{S}$ such that $[\![q_1(T)]\!] \neq [\![q_2(T)]\!]$, then there also exists a table $T' \in \mathcal{S}'$ such that $[\![q_1(T')]\!] \neq [\![q_2(T')]\!]$.*

Proof Sketch. Assume that $T \in \mathcal{S}$ is a counterexample with distinguishing output t_O for q_1 and q_2 (with multiplicities m_1, m_2, respectively). Then, $T^{\phi(t,t_O)}$ is a table from $\mathcal{S}'$. According to Def. 5.1, applying q_1, q_2 on $T^{\phi(t,t_O)}$ also results in m_1, m_2 as the multiplicities of t_O. This shows that $T^{\phi(t,t_O)}$ is also a counterexample for q_1, q_2. □

This property guarantees that if we fail to find a counterexample in $\mathcal{S}'$ for two queries q_1 and q_2, then q_1 and q_2 are guaranteed to be equivalent in $\mathcal{S}$. We show how to encode the refined search space in Appendix A.

6 EVALUATION

We evaluate the effectiveness of our space refinement algorithm on three tools that reason about tables and queries for different purposes: (1) bounded verification [Chu et al. 2017a], (2) test data generation (in the context of mutation testing[Chandra et al. 2015], auto-grading [Gupta et al. 2010]) and unit test data generation [Veanes et al. 2010], and (3) concolic testing [Tanno et al. 2015]. In each scenario, we run the tool on benchmarks extracted from the original paper with and without space refinement, and compare performance differences. In addition, we run each experiment twice with Cosette encoding [Chu et al. 2017a] and Qex encoding [Veanes et al. 2010] to demonstrate that our space refinement algorithm is general to different underlying solver implementations.

6.1 Experiment 1: Bounded Verification

We first study the space refinement algorithm on bounded verification. To do so, we use 46 benchmarks collected from the 232 test cases for the SQL rewrite rules in the Apache Calcite project,[3] an open source query optimization framework used by many database systems. Cases excluded from the benchmark are either those testing non-SQL feature or those containing features that are currently not support by Cosette and Qex (e.g., Partition, Order-By, Case and In). These 46 benchmarks contain non-trivial use of SQL operators: 29 cases contain queries with more than 5 subqueries, and 41 cases involve tables with more than 9 columns.

For each benchmark and each encoding method (i.e., Cosette and Qex), we chose the verification bound as the size of the maximum search space that the solver can completely explore within 600 seconds without space refinement. We next re-run the solver on the same bound but with space refinement. We measure the search space based on the number of symbolic values used in the encoding and report the relative performance with and without space refinement for each encoding approach.

This experiment helps us answer how the refinement algorithm affects the bounded verification process of different types of query, different search space size and different underlying solver choices.

Conclusion 1. Queries with aggregations benefit most from space refinement.

As shown in Table 1, 19 out of the 21 cases with aggregations show significant speedup in the bounded verification task, using both Qex and Cosette encodings. The medium is a 48× speedup for Cosette and a 58× speedup for Qex. The speedup mainly comes from the reduction of the number of groups that the solver needs to consider while using the refined search space. In such cases, the refinement algorithm determines it is sufficient to encode only a small number of groups to prove the equivalence between the given queries. The two cases that didn't benefit from the refinement algorithm are *boolean queries*, i.e., queries of the form "Select 1 From . . . ". In these cases, query inputs only affect how many "1"s are returned, and the provenance analysis algorithm can not propagate the initial predicate to its subqueries in a non-trivial way. As a result, the space refinement algorithm determines that *all* tuples in the input table are necessary to determine the output tuple (the tuple (1)) multiplicity.

For the other 25 cases that do not involve aggregation, only 6 cases display significant speedup. The other 19 cases are either unaffected or slowed down (minor slow-down with less than 10% time difference). These 25 cases are unions of conjunctive queries (UCQs) constructed from Select, Join, and Union operators. In these cases, since the query semantics does not introduce interactions among tuples, the underlying symbolic evaluator and SMT solver can also exploit the independence among different tuples when solving generated constraints. As a result, such queries benefit less

[3] `https://calcite.apache.org`

from space refinement. The speedup achieved in the 6 non-aggregation cases comes from backward constant propagation, i.e., the propagation constants appeared in query predicates to input tables; this propagation allows us to preassign values to certain parts of the symbolic table, which reduces the number of symbolic values need to encode the search space. For example, given the query "Select . . . Where $c = 10$", the analysis algorithm propagates the constant 10 from the predicate to the input table through a provenance predicate $t.1 = 10$, which frees us from using symbolic values for the column c in the input table.

Conclusion 2. The benefits of space refinement generalize to different encoding methods and different query sizes.

While the speedup varies for different encoding methods, i.e., Cosette v.s. Qex, whether a pair of queries benefits from space refinement is not affected. All cases in Table 1 with over 2× speedups display under Cosette encoding also display a noticeable improvement under Qex encoding. Also, compared to query structural differences (e.g., whether the target query uses aggregation or contains constants), the differences in query sizes have little influence on the amount of speedup gained from space refinement.

Illustrative Examples. In the following, we provide two examples to demonstrate the strengths and limitations of the space refinement algorithm. Both examples run on the following two tables:

```
Dept(deptno:int, name:str)
Emp(empno:int, ename:str, job:int, mgr:int, hiredate:int,
    comm:int, sal:int, deptno:int, slacker:int)
```

- *(PushFilterPastAggGroupSets2).* In this example, the refinement algorithm produces the provenance predicate $\phi(t, t_O) = (t.\text{name} = \text{“}Charlie\text{''} \land t.\text{deptno} = t_O.2)$. The predicate determines that (1) we need to consider only the group with name '*Charlie*', and (2) we only need to consider one department group. In this way, the solver no longer needs to consider the all possible ways to group the name and deptno columns (which is exponential to the input table size). This result in a speedup of 160×, as shown in Table 1.

```
-- q1
Select name, deptno, Count(*)
From Dept
Group By name, deptno
Having name = 'Charlie';
```

```
-- q2
Select t2.name, t2.deptno, Count(*)
From   (Select name, deptno
        From Dept) As t2
Where  t2.name = 'Charlie'
Group By t2.name, t2.deptno;
```

- *(TransitiveInferenceJoin3Way).* This example shows a boolean query that does not benefit from space refinement. Since both these query outputs are tables consisting of tuples with content 1, our algorithm generates only a trivial provenance predicate $\phi(t, t_O) = (t.\text{deptno} > 7 \land t_O.1 = 1)$ that cannot effectively reduce complexity of encoding the search space.

```
-- q1
SELECT 1
FROM (SELECT * FROM emp
      WHERE emp.deptno > 7) AS t
INNER JOIN emp AS EMP0
ON t.deptno = EMP0.deptno
INNER JOIN emp AS EMP1
ON EMP0.deptno = EMP1.deptno;
```

```
-- q2
SELECT 1
FROM (SELECT * FROM emp
      WHERE  deptno > 7) AS t1
INNER JOIN (SELECT * FROM emp
            WHERE deptno > 7) AS t2
ON t1.deptno = t2.deptno
INNER JOIN (SELECT * FROM emp
            WHERE deptno > 7) AS t3
ON t2.deptno = t3.deptno;
```

In sum, the search space refinement algorithm effectively speeds up bounded verification of an important fraction of complex queries.

6.2 Experiment 2: Test Data Generation

Our second experiment studies how the space refinement algorithm can be used to improve test data generation tasks, including: (1) generating inputs to disambiguate non-equivalent query pairs (for mutation testing and auto-grading) and (2) generating unit test inputs for the given query such that the query returns a non-empty output [Veanes et al. 2010].

In this experiment, we use 13 query disambiguation benchmarks and 2 unit test generation benchmarks from prior work. In the benchmark collection phase, we exclude cases whose distinguishing input tables are those with only 1 tuple, as all such cases can be solved within 0.2 seconds by current solvers and do not present scalability challenges in terms of search space size. For each benchmark, we measure the time each solver it takes (Cosette, Qex) to find the first desirable model with and without search space refinement.

Conclusion 3. The benefit of space refinement is limited when the target model size is small.

As shown in Table 2, the refined space results in speedups for 7 cases under Qex encoding and 6 cases under Cosette encoding. These cases are harder cases whose solutions require more tuples. Other cases are either unaffected or show an insignificant (<0.5 seconds) slowdown. The speedup is limited in these benchmarks because most cases requires tables only with 2 distinct tuples to disambiguate. As a result, the benefit of space size reduction does not compensate for the overhead of encoding the refinement predicate.

6.3 Experiment 3: Concolic Testing

Last, we demonstrate that the space refinement algorithm can speed up cocolic testing of relational queries. In this experiment, we manually translate the following two pairs of representative SQL queries (q_1, q_2 and q_3, q_4) into Java programs and call the CATG concolic testing engine to test whether the two queries are equal.

```
-- q1                    -- q2                  -- q3                    -- q4
SELECT id, SUM(val)      SELECT fid,year        SELECT fid,year          SELECT fid,year
FROM Flight              FROM Flight            FROM Flight              FROM Flight
WHERE year > 2010        WHERE year > 2010      WHERE year > 2010        WHERE year > 2015
GROUP BY fid;            GROUP BY fid, year;    AND fid = carrier;       AND fid = carrier;
```

Given a pair of queries, we create a Java snippet shown below, where the two queries take as input a randomly initialized concolic table. Then, we run the CATG concolic testing engine on the Java snippet and log the time spent by the concolic tester to run 20 test iterations. We set the size of the symbolic input table (the number of tuples in the table) as a variable and study the performance of the concolic tester.

```
... // input and query definition
if (tableEqual(q1.execute(), q2.execute()) {
    System.out.print("Reach EQ Branch");
} else {
    System.out.print("Reach NEQ Branch");
}
```

Conclusion 4. The benefit of space refinement in concolic testing increases as input space size increases.

Table 1. The evaluation result for bounded verification on Calcite benchmarks. Column #sq refers to the number of subqueries in the target query for measuring complexity, #SV refers to the number of symbolic values used in encoding the search space, and $t_{\mathcal{S}}$, $t_{\mathcal{S}'}$ refer to the time spent by the solver without and with space refinement.

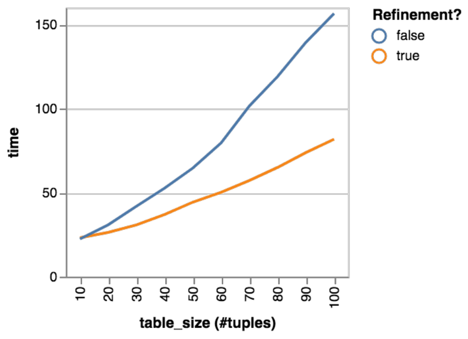
Calcite (with aggregates)	#sq	Qex Encoding				Cosette Encoding			
		#SV	$t_{\mathcal{S}}$(s)	$t_{\mathcal{S}'}$(s)	Speedup	#SV	$t_{\mathcal{S}}$(s)	$t_{\mathcal{S}'}$(s)	Speedup
PushFilterPastAgg	3	14	33.44	0.26	130.0	15	107.61	0.26	410.0
PushFilterPastAggTwo	3	16	195.55	0.69	280.0	19	154.43	0.56	280.0
AggConstKeyRule3	3	63	115.1	0.72	160.0	59	92.89	0.47	200.0
PullFilterThroughAgg	3	63	58.92	0.69	86.0	68	119.08	0.7	170.0
PushFilterPastAggGroupSets2	3	16	58.67	0.73	81.0	21	121.18	0.74	160.0
PullFilterThroughAggGroupSets	3	54	49.59	0.58	86.0	59	59.97	0.57	100.0
PullAggThroughUnion	6	45	61.83	1.06	58.0	50	92.28	1.15	81.0
AggProjectPullUpConsts	2	45	84.72	1.47	58.0	28	36.55	0.61	60.0
PushAvgThroughUnion	6	63	86.58	3.05	28.0	68	188.93	3.79	50.0
PushAggThroughJoin1	6	33	221.71	5.65	39.0	38	263.84	5.5	48.0
PushFilterPastAggGroupSets1	2	24	163.09	1.97	83.0	27	60.4	1.73	35.0
AggConstKeyRule2	2	108	69.82	2.68	26.0	113	76.72	2.63	29.0
PushFilterPastAggThree	2	117	128.88	3.75	34.0	113	87.62	3.07	29.0
PushAvgGroupSetsThroughUnion	6	63	215.38	3.47	62.0	59	54.38	2.05	27.0
PushAggThroughJoin3	5	35	32.14	3.19	10.0	38	28.32	2.47	11.0
AggProjectMerge	2	99	149.11	2.55	58.0	82	120.19	12.21	9.8
AggGroupSetsProjectMerge	2	99	148.58	2.58	58.0	82	120.35	12.26	9.8
PushAggThroughJoinDistinct	6	35	146.25	11.69	13.0	29	57.0	6.29	9.1
AggConstKeyRule	2	54	97.12	3.8	26.0	50	12.89	1.95	6.6
TransitiveInferAgg	6	63	69.06	69.56	0.99	59	42.0	40.44	1.0
TransitiveInferJoin3wayAgg	9	45	106.76	107.36	0.99	59	35.56	37.64	0.94

Calcite (without aggregates)	#sq	#SV	$t_{\mathcal{S}}$(s)	$t_{\mathcal{S}'}$(s)	Speedup	#SV	$t_{\mathcal{S}}$(s)	$t_{\mathcal{S}'}$(s)	Speedup
PullConstIntoProject	2	126	98.26	0.52	190.0	122	71.05	0.5	140.0
PullConstIntoFilter	3	171	45.45	0.88	52.0	176	54.96	0.89	62.0
RemoveSemiJoinFilter	5	79	55.14	0.85	65.0	38	0.38	0.04	9.7
RemoveSemiJoinRightFilter	7	46	57.31	1.36	42.0	26	0.35	0.04	7.8
MergeJoinFilter	5	46	54.54	0.44	120.0	37	1.12	0.15	7.4
MergeFilter	3	290	5.73	6.88	0.83	283	10.65	7.03	1.5
TransitiveInferUnion3way	13	54	106.9	108.8	0.98	32	0.95	0.96	1.0
PushJoinThroughUnionOnRight	10	72	62.92	63.3	0.99	59	7.73	7.72	1.0
PullConstThroughUnion3	6	1188	7.15	7.07	1.0	1157	7.52	7.36	1.0
TransitiveInferJoin	6	90	140.83	142.33	0.99	68	47.35	46.93	1.0
PushJoinCondDownToProject	6	222	37.77	38.37	0.98	227	38.45	37.28	1.0
TransitiveInferUnion	9	63	79.72	78.27	1.0	41	11.99	11.93	1.0
TransitiveInferJoin3way	9	81	195.8	200.27	0.98	41	0.59	0.59	1.0
TransitiveInferUnionAlwaysTrue	10	45	113.56	113.5	1.0	32	0.91	0.91	1.0
TransitiveInferProject	6	90	145.19	145.35	1.0	68	52.94	52.16	1.0
TransitiveInferComplexPredicate	7	63	100.72	101.25	0.99	32	172.81	170.94	1.0
RemoveSemiJoinRight	6	101	69.02	68.49	1.0	103	71.41	70.37	1.0
TransitiveInferConjInPullUp	6	72	69.42	69.3	1.0	41	7.08	7.04	1.0
PushJoinThroughUnionOnLeft	10	72	78.67	78.44	1.0	59	10.11	10.05	1.0
ExtractJoinFilterRule	4	418	18.12	18.52	0.98	423	17.48	17.25	1.0
TransitiveInferPullUpThruAlias	6	45	8.99	9.18	0.98	41	107.75	105.92	1.0
SemiJoinReduceConsts	8	234	44.52	43.97	1.0	239	44.57	44.88	0.99
PushProjectPastSetOp	6	972	8.17	8.51	0.96	959	8.14	8.44	0.96
PullConstThroughUnion	6	918	8.39	8.96	0.94	910	8.55	8.98	0.95
PullConstThroughUnion2	5	909	8.43	8.68	0.97	905	10.15	11.2	0.91

Table 2. The evaluation result for test data generation benchmarks. Column #sq refers to the number of subqueries in the target query for measuring complexity, #SV refers to the number of symbolic values used in encoding the search space, and t_S, $t_{S'}$ refer to the time spent by the solver without and with space refinement.

Case	#Q	Qex Encoding				Cosette Encoding			
		#SV	t_S(s)	$t_{S'}$(s)	Speedup	#SV	t_S(s)	$t_{S'}$(s)	Speedup
mutant-1	8	16	0.28	0.34	0.83	18	0.49	0.08	6.0
mutant-2	4	15	2.06	0.98	2.1	17	2.65	1.83	1.4
mutant-3	7	36	12.46	8.3	1.5	43	37.74	27.8	1.4
mutant-4	7	30	7.26	6.93	1.0	37	7.16	6.69	1.1
mutant-5	8	38	0.33	0.33	1.0	45	0.65	0.65	1.0
mutant-6	7	18	1.59	1.61	0.99	25	4.3	4.27	1.0
hw-1	5	8	0.46	0.47	0.97	12	1.08	1.07	1.0
mutant-7	13	6	9.12	9.46	0.96	5	0.56	0.56	1.0
mutant-8	5	8	0.3	0.29	1.0	15	0.27	0.27	1.0
hw-2	5	8	0.31	0.31	1.0	12	0.24	0.24	0.99
mutant-9	5	16	0.15	0.14	1.1	23	0.19	0.2	0.95
hw-3	4	21	53.51	17.99	3.0	18	2.2	2.31	0.95
hw-4	4	16	1.69	0.58	2.9	18	6.85	7.58	0.9
unit-test-1	4	13	0.12	0.1	1.1	13	0.22	0.16	1.3
unit-test-2	4	14	3.55	1.98	1.8	8	0.46	0.39	1.2

In both examples, the concolic test engine successfully found input tables to cover both branches in the Java snippet. As shown in Figure 10, both examples indicate that the refined search space enables the concolic test engine to run faster in the testing process. In the first example (q_1, q_2), the provenance predicate $\phi(t, t_O) = (t.\text{fid} = t_O.1 \wedge t.\text{year} > 2010)$ restricts the choices of the grouping key fid to be the same across the table. In the second example, the provenance predicate determines that it is sufficient to make both fid and year to be the same. Thus, the concolic test engine benefits from the smaller size of the refined search space to generate inputs faster.



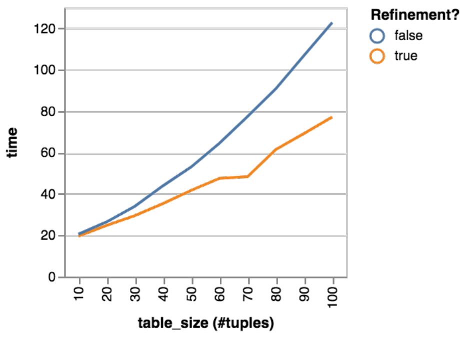


Fig. 10. The experiment for comparing query equivalence between q_1, q_2 (left) and q_3, q_4 (right).

7 RELATED WORK

SQL Equivalence. SQL query equivalence is a problem that has been extensively study in the database theory community [Chaudhuri and Vardi 1993; Cohen 2009; Cohen et al. 2007; Sagiv and Yannakakis 1980]. In general, query equivalence is undecidable, and subsequent study aims to identify decidable subsets of SQL and build decision procedures for them. Known decidable SQL subsets include conjunctive queries (CQ) [Chaudhuri and Vardi 1993], CQ with union (UCQ) [Sagiv and Yannakakis 1980], CQ with linear arithmetic [Cohen 2009], and CQ with *one* aggregate in the outer-most layer [Cohen et al. 2007].

Tools for reasoning about query equivalence include HottSQL [Chu et al. 2017b], a proof environment built on top of Coq for interactive SQL proof that targets a different verification method, and Cosette [Chu et al. 2017a], an SMT-based verifier for bounded verification. As shown in our case study, our space refinement algorithm can be used to improve symbolic reasoning of SQL equivalence.

[Schäfer and de Moor 2010] describes a type inference algorithm that maps datalog queries into type programs formulated in (decidable) monadic datalog to check Datalog query containment. These types are abstractions of the program semantics, and they can be used to optimize query execution as well as query containment checking. In particular, containment checking within the type language is sound but incomplete. Their abstraction is a sound approximation of all possible input tables. We also map programs (i.e., queries) into constraints for equivalence checking but with different goals: we are interested in producing symbolic constraints that can be used to generate concrete test cases or counterexamples in addition to proving query equivalence. As a result, our approach uses the provenance of query outputs to speed up symbolic reasoning from existing solvers, rather than approximating programs semantics.

Test Data Generation. Tools for generating test data for SQL queries include Qex [Veanes et al. 2009, 2010], XData [Gupta et al. 2010] and Tesma [Tanno et al. 2015]. Qex is a SMT-based tool for generating unit tests from a given query and test assertion, where the goal is to construct unit tests from the data, query and assertion. XData is a mutation testing tool for SQL: given an input query, XData generates mutations of the query and then asks the underlying SMT solver to construct a distinguishing input to kill the mutant. It then adds the generated distinguishing input to the test suite for database testing or grading [Bhangdiya et al. 2015]. Tesma is a concolic test engine that allows testing database applications under the concolic test framework. As shown in our evaluation, our space refinement algorithm can be applied to speedup test data generation.

Symmetry Breaking. Both Qex and Cosette include symmetry breaking modules for compiling into SMT formulas, similar to traditional symmetry breaking techniques [Crawford et al. 1996; Déharbe et al. 2011] used in constraint solving. Our approach instead utilizes high-level program semantics and breaks symmetry by identifying table equivalence given the semantics of the input queries. Also, our approach is specific to queries but generalizable to underlying encoding methods, which allows the opportunity to combine it with other lower-level symmetry breaking techniques [Torlak and Bodík 2014].

Provenance Analysis. Provenance analysis has been studied in both scientific computation [Bose and Frew 2005] and the database community [Buneman et al. 2006, 2001; Cui et al. 2000]. Data provenance has been applied to incremental view updates and data filtering [Green et al. 2007] to improve database performance. Our symbolic provenance analysis resembles these approaches, but our algorithm generates predicates to describe the provenance relation for *symbolic output tuples*, and we extend provenance reasoning to multiple queries simultaneously to solve the query equivalence problem. Our symbolic provenance analysis is a backward abstract analysis that feeds its result to improve the forward concrete analysis (e.g., compiling queries to SMT formulas, generating test cases). Previous work used backward analysis to summarize information that is not readily available to forward analysis, making the latter more efficient [Chandra et al. 2009; Duesterwald et al. 1995; Horwitz et al. 1995].

Semantics Abstraction. Semantics abstraction [Feng et al. 2017; Feser et al. 2015; Polikarpova et al. 2016; Wang et al. 2017] is also used in program synthesis to speed up (program) search space traversal, where program synthesizers compute space refinement constraints using abstract semantics of the target language to perform search space pruning. In program synthesis, such space refinement constraints are computed from (1) user specifications of the target program (logic formulas [Polikarpova et al. 2016] or input-output examples [Feng et al. 2017; Feser et al. 2015; Wang et al. 2017]) and (2) currently synthesized partial programs. The refinement constraints capture properties of partial programs and allow the synthesizer to partition and prune the search space before reaching complete programs. In symbolic reasoning tasks, provenance predicates are computed from the verification condition and the target programs, which are then used to break semantic symmetry in the (table) search space. While different, studying the relationship between the two types of refinement constraints in these scenarios offers an interesting future work.

Verification of Database Applications. Mediator [Wang et al. 2018] is a tool for reasoning about database applications with updates. The SparkLite verifier [Grossman et al. 2017] is an SMT-based tool for checking MapReduce program equivalence. Both approaches check program equivalences by inferring program invariants. Our space refinement algorithm currently can only speedup the type of assertions defined in section 2 but not general invariants. Generalizing our symmetry breaking algorithm to richer assertions is an interesting future work.

8 CONCLUSION

In this paper, we introduced a space refinement algorithm for symbolic SQL reasoning. At the algorithm's core are: a symbolic provenance analysis module that analyzes which tuples in the input table contribute to the multiplicities of the target tuples in query outputs, and a space refinement module that refines the search space with the provenance information. Our experiments on bounded SQL verification, test data generation and concolic testing show that the refined search space effectively speeds up the symbolic reasoning process.

A SEARCH SPACE ENCODING

We discuss how Qex and Cosette represent the search space, and how we encode the refined search space in these tools.

Representation of $\mathcal{S}$. Qex encodes the search space based on the exact number of tuples allowed in the table, and the search space of all tables containing k tuples is encoded as a list of tuples where each tuple is a list of symbolic values. For example, given the schema Bonus(Job:int, Dept:int, Sal:int), the space of all tables with 3 tuples is encoded as the symbolic table $\texttt{Bonus}_Q$ in Figure 11, where each value s_{ij} in the table is a symbolic integer. To encode the search space consisting of all tables with at most k tuples, Qex would iteratively increase the number of tuples from 0 to k and check the verification condition separately. Since the order of tuples in a table does not matter, Qex adopts a set of encoding constraints to break the encoding symmetry by asserting a canonical order of the tuples in the table. These constraints avoid the solver to encode the same table multiple times with a different order of tuples in the content.

Cosette differs from Qex by explicitly encoding the multiplicities of the tuples in the symbolic table, and a symbolic table with k entries in Cosette represents the search space of all tables with at most k distinct tuples. The symbolic table $\texttt{Bonus}_C$ in Figure 11 shows how Cosette encodes all tables with the schema Bonus containing at most 3 different tuples, and Cosette adopts a similar encoding constraints as Qex to reduce encoding symmetry. Compared to Qex, Cosette's encoding approach has the benefit of being able to compress the encoding for tables containing multiple identical tuples, e.g., a table with 100 identical tuples is represented with only one symbolic tuple with multiplicity 100. On the other hand, the use of multiplicity also makes representing tables without repeating tuples less compressed.

Bonus_Q

job	dept	sal
s_{11}	s_{12}	s_{13}
s_{21}	s_{22}	s_{23}
s_{31}	s_{32}	s_{33}

encoding constraints:
$(s_{21}, s_{22}, s_{23}) \geq (s_{11}, s_{12}, s_{13})$
$\wedge\ (s_{31}, s_{32}, s_{33}) \geq (s_{21}, s_{22}, s_{23})$

Bonus_C

job	dept	sal	mult
s_{11}	s_{12}	s_{13}	m_1
s_{21}	s_{22}	s_{23}	m_2
s_{31}	s_{32}	s_{33}	m_3

encoding constraints:
$(m_1 \geq 0) \wedge (m_2 \geq 0) \wedge (m_3 \geq 0)$
$\wedge (s_{21}, s_{22}, s_{23}) \geq (s_{11}, s_{12}, s_{13})$
$\wedge (s_{31}, s_{32}, s_{33}) \geq (s_{21}, s_{22}, s_{23})$

Fig. 11. The encoding of the search space of all tables containing 3 tuples by Qex (Bonus_Q), and the encoding of the search space of all tables containing at most 3 tuples in Cosette (Bonus_C). Both encodings adopt a set of encoding constraints to reduce the encoding symmetry.

Representation of $\mathcal{S}'$. The reduced search space generated from $\mathcal{S}$ is encoded similarly. We encode the refined search space $\mathcal{S}'$ by adding additional assertions to the encoding constraints.

We first introduce a new symbolic tuple $t_O = (a_1, \ldots, a_n)$, where n is the number of columns in the output of q_1 and q_2, and then assert the constraint $\phi(t, t_O)$ for each t_O that encodes $\mathcal{S}$.

For example, assume the provenance of a query whose output has two columns (denoted as $t_O.1$ and $t_O.2$) is $\phi(t, t_O) = (t.\texttt{Job} = t_O.1 \wedge t.\texttt{Sal} > 5)$ (for the Bonus table above). To encode the refined search space $\mathcal{S}'$ using ϕ, we first introduce two new symbolic values (a_1, a_2) to model t_O, and add the following assertion in addition to the encoding constraint:

$$(s_{11} = a_1 \wedge s_{12} > 5) \wedge\ (s_{21} = a_1 \wedge s_{22} > 5) \wedge\ (s_{31} = a_1 \wedge s_{32} > 5)$$

Alternatively, we can also simplify the encoding by eliminating redundant symbolic values, e.g., replacing all s_{i1} with a_1 in the example above.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1651489, OAC-1739419, CCF-1139138, CCF-1337415, NSF ACI-1535191, and NSF 16-606; DARPA award FA8750-16-2-0032 and FA8750-14-C-0011; DOE award DE-SC0016260 and FOA-0000619; the Intel-NSF CAPA center; the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; as well as gifts from Google, Intel, Mozilla, Nokia, Qualcomm, and Huawei. We would also like to thank Shumo Chu, Dan Suciu, and Emina Torlak for insightful discussions in early stage of the project, as well as anonymous reviewers for their valuable comments on paper revising.

REFERENCES

Amol Bhangdiya, Bikash Chandra, Biplab Kar, Bharath Radhakrishnan, KV Maheshwara Reddy, Shetal Shah, and S Sudarshan. 2015. The XDa-TA system for automated grading of SQL query assignments. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on.* IEEE, 1468–1471.

Rajendra Bose and James Frew. 2005. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.* 37, 1 (2005), 1–28. https://doi.org/10.1145/1057977.1057978

Peter Buneman, Adriane Chapman, and James Cheney. 2006. Provenance management in curated databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006.* 539–550. https://doi.org/10.1145/1142473.1142534

Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*. 316–330. https://doi.org/10.1007/3-540-44503-X_20

Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. 2015. Data generation for testing and grading SQL queries. *VLDB J.* 24, 6 (2015), 731–755. https://doi.org/10.1007/s00778-015-0395-0

Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 363–374. https://doi.org/10.1145/1542476.1542517

Surajit Chaudhuri and Moshe Y. Vardi. 1993. Optimization of *Real* Conjunctive Queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC, USA*. 59–70. https://doi.org/10.1145/153850.153856

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2011. Partial Replay of Long-running Applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. 135–145.

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017a. Cosette: An Automated Prover for SQL. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017b. HoTTSQL: proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 510–524. https://doi.org/10.1145/3062341.3062348

E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387.

Sara Cohen. 2009. Equivalence of queries that are sensitive to multiplicities. *VLDB J.* 18, 3 (2009), 765–785. https://doi.org/10.1007/s00778-008-0122-1

Sara Cohen, Werner Nutt, and Yehoshua Sagiv. 2007. Deciding equivalences among conjunctive aggregate queries. *J. ACM* 54, 2 (2007), 5. https://doi.org/10.1145/1219092.1219093

James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. 1996. Symmetry-Breaking Predicates for Search Problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996*. 148–159.

Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.* 25, 2 (2000), 179–227. https://doi.org/10.1145/357775.357777

David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. 2011. Exploiting Symmetry in SMT Problems. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*. 222–236. https://doi.org/10.1007/978-3-642-22438-6_18

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1995. Demand-driven Computation of Interprocedural Data Flow. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 37–48. https://doi.org/10.1145/199448.199461

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 422–436. https://doi.org/10.1145/3062341.3062351

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 229–239. https://doi.org/10.1145/2737924.2737977

Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. 2007. Update Exchange with Mappings and Provenance. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. 675–686. http://www.vldb.org/conf/2007/papers/research/p675-green.pdf

Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. 2017. Verifying Equivalence of Spark Programs. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 282–300. https://doi.org/10.1007/978-3-319-63390-9_15

Bhanu Pratap Gupta, Devang Vira, and S. Sudarshan. 2010. X-data: Generating test data for killing SQL mutants. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. 876–879. https://doi.org/10.1109/ICDE.2010.5447862

Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. 1995. Demand Interprocedural Dataflow Analysis. In *SIGSOFT '95, Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington, DC, USA, October 10-13, 1995*. 104–115. https://doi.org/10.1145/222124.222146

Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. 1991. Formal Semantics of SQL Queries. *ACM Trans. Database Syst.* 16, 3 (1991), 513–534. https://doi.org/10.1145/111197.111212

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 522–538. https://doi.org/10.1145/2908080.2908093

Yehoshua Sagiv and Mihalis Yannakakis. 1980. Equivalences Among Relational Expressions with the Union and Difference Operators. *J. ACM* 27, 4 (1980), 633–655. https://doi.org/10.1145/322217.322221

Max Schäfer and Oege de Moor. 2010. Type inference for datalog with complex type hierarchies. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 145–156. https://doi.org/10.1145/1706299.1706317

Shetal Shah, S. Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, and Devang Vira. 2011. Generating test data for killing SQL mutants: A constraint-based approach. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*. 1175–1186. https://doi.org/10.1109/ICDE.2011.5767876

Haruto Tanno, Xiaojing Zhang, Takashi Hoshino, and Koushik Sen. 2015. TesMa and CATG: automated test generation tools for models of enterprise applications. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 717–720.

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 '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 530–541. https://doi.org/10.1145/2594291.2594340

Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. 2009. Symbolic Query Exploration. In *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*. 49–68. https://doi.org/10.1007/978-3-642-10373-5_3

Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL Query Explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. 425–446. https://doi.org/10.1007/978-3-642-17511-4_24

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 452–466. https://doi.org/10.1145/3062341.3062365

Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2018. Verifying equivalence of database-driven applications. *PACMPL* 2, POPL (2018), 56:1–56:29. https://doi.org/10.1145/3158144