Synthesizing Highly Expressive SQL Queries from Input-Output Examples

Chenglong Wang Alvin Cheung Rastislav Bodik
{clwang, akcheung, bodik}@cs.washington.edu
University of Washington, USA
http://scythe.cs.washington.edu

Abstract

SQL is the de facto language for manipulating relational data. Though powerful, SQL queries can be difficult to write due to their highly expressive constructs. Using the programming-by-example paradigm to help users write SQL queries presents an attractive proposition, as evidenced by online help forums such as Stack Overflow. However, developing techniques to synthesize SQL queries from input-output (I/O) examples has been difficult due to SQL's rich set of operators.

In this paper, we present a new scalable and efficient algorithm to synthesize SQL queries from I/O examples. Our key innovation is the development of a language for abstract queries, i.e., queries with uninstantiated operators, that can express a large space of SQL queries efficiently. Using abstract queries to represent the search space nicely decomposes the synthesis problem into two tasks: (1) searching for abstract queries that can potentially satisfy the given I/O examples, and (2) instantiating the found abstract queries and ranking the results. We implemented the algorithm in a new tool, called SCYTHE, and evaluated it on 193 benchmarks collected from Stack Overflow. Our results showed that SCYTHE efficiently solved 74% of the benchmarks, most in just a few seconds. Queries synthesized by SCYTHE range from simple ones involving a single selection to complex ones with six levels of nested queries.

CCS Concepts • **Software and its engineering** → **Programming by example**; • **Information systems** → **Structured Query Language**

Keywords Program Synthesis, Query by Example, SQL

1. Introduction

Relational databases serve an important role in modern data management, and SQL remains one of the most commonly used query languages to manipulate relational data. SQL can be used for many basic tasks, such as selecting columns from a table; its rich features also make it useful for solving complex data manipulation tasks, such as computing arg max and joining multiple tables together with aggregates. However, the various operators available in SQL and the many ways that they can be combined to form advanced idioms (e.g., correlated subqueries, unions, nested queries, groupings, various types of joins, etc) make the language difficult to master, as evidenced by over 10,000 Stack Overflow SQL related posts. In fact, many problems are so common among end users that they are grouped with popular tags, such as "greatest-n-per-group," "argmax," and "moving-average."

These are basically tags used in stackOverflow

Though end users find solving these problems to be challenging, they can often specify the problems using input output (I/O) examples, as observed in many Stack Overflow posts. Given the recent advances in programming-by-example (PBE) systems [12, 13, 15, 17, 29], building a tool that helps users write SQL by soliciting I/O examples from users would alleviate their need to learn complex SQL constructs and idioms.

Prior work [36, 40] has developed automatic synthesizers for SQL queries using I/O examples. However, it handles only a small subset of the language and does not cover a wide range of practical tasks. We observe that the difficulty in developing automatic synthesizer for SQL queries results from several of its unique features. First, the space of SQL queries is huge: many SQL operators (selection, projection, joins, grouping, etc) are parameterized by predicates, and they can be composed with each other. Second, the first-class value in SQL, table, is a type of compound value that is expensive to compute and memoize: tables computed from joins and unions can contain hundreds to thousands scalar values. Third, unlike spreadsheet data transformation languages, whose operators can be decomposed and inferred

backwardly from output examples, SQL operators cannot be trivially decomposed and learned in this way. For the example in Figure 1, the `Join` predicates cannot be inferred independently from its nested subqueries as they jointly affect the output table. Thus, it is unclear whether the "divide-and-conquer" synthesis algorithms used in other domains [12, 13, 26, 32] would remain scalable in SQL.

Our key insight to address the above challenges is to develop a new abstraction – the language of abstract queries – to decompose the originally challenging synthesis problem. Abstract queries in this language are syntactically similar to SQL queries except that filter predicates are replaced with *holes* that can be instantiated with any valid predicate. Since operators in abstract queries are no longer parameterized by predicates, the search space of abstract queries is significantly reduced than the original one. Furthermore, the language contains a set of evaluation rules: given an abstract query, the rules evaluate it into a table that over-approximates the results of all queries that can be instantiated from it, allowing us to prune the search space earlier.

With this abstraction, we decompose the SQL synthesis problem into two phases. In the first phase, we search for abstract queries that can potentially be instantiated into SQL queries that satisfy the given I/O examples, and we prune away others based on their evaluation result. Then, in the second phase, we search for proper predicates for each synthesized abstract query to instantiate it into desired SQL queries and return top candidates to the user. To make the predicate search process efficient, we cluster predicates into semantically equivalence classes and encode tables using bit-vectors.

We implemented our algorithm in a PBE tool called SCYTHE. To evaluate SCYTHE, we collected 165 real-world benchmarks from Stack Overflow and 28 benchmarks from previous work [40]. Results showed that SCYTHE quickly and precisely solved more than 74% of the benchmarks and 80% out of these solved benchmarks were solved within a few seconds. Our algorithm solved 51 more cases within 600 seconds compared to the enumerative search algorithm [25, 37] and outperformed SQLSynthesizer [40] on their project benchmarks with 4 more cases solved.

In sum, our paper makes the following contributions:

- We present a new approach to decompose the SQL query synthesis problem. Our key innovation is the design of the language of abstract queries and rules to evaluate them. (Section 4)
- We describe efficient synthesis algorithms optimized using properties of abstract queries to solve the two subproblems after decomposition, i.e., searching for abstract queries and instantiating them. (Section 5)
- We implemented our algorithm in a PBE system SCYTHE and evaluated it on 193 real-world benchmarks. Results show that SCYTHE makes substantial improvement compared to the enumerative search algorithm and other prior tools for synthesizing SQL queries. (Section 6)

the JOIN operation's condition (the predicate T3.uid = T5.oid) depends on the results of both subqueries . We cannot determine the effect of the JOIN just by looking at one subquery without considering the other.

2. Overview

We first demonstrate our algorithm with a running example.

Problem Statement. We formalize a user's query as a triple (I, T_{out}, C), where $I = \{T_1, ..., T_n\}$ stands for input tables, T_{out} stands for the output table, and $C = \{v_1, ..., v_k\}$ stands for a set of predicate constants. We seek to synthesize a query q such that $q(I) = T_{\text{out}}$ with the additional constraint that all constants used in predicates in q must come from C.

Note two special characteristics of our problem formalization that are tailored to relational queries in constrast to PBE systems in other domains [12, 28, 38]. First, we ask users to provide constants that will be used in predicates to make the problem less ambiguous as well as to boost the search efficiency. Based on our study of Stack Overflow questions, we find that users are usually willing to provide such constants along with table examples (e.g., return values that are between dates "12/24" and "12/25") because failing to provide them would result in a degree of ambiguity that must be resolved in a dialogue with experts.

Second, our problem formulation allows only one I/O example pair (I, T_{out}): the rationale is that users tend to resolve the ambiguities in the provided example by revising the example rather than creating a completely new one, so our algorithm needs to accept only one I/O pair.

Running Example. We combine two Stack Overflow posts into a running example to demonstrate the full features of our algorithm.[1] This example contains two input tables $I = \{T_1, T_2\}$, an output table T_{out} (as shown below), and constants {"12/25", "12/24", 50}.

T_1

id	date	uid
1	12/25	1
2	11/21	1
4	12/24	2

T_2

oid	val
1	30
1	10
1	10
2	50
2	10

T_{out}

c_0	c_1	c_2	c_3	c_4
1	12/25	1	1	30
4	12/24	2	2	10

The Solution. Figure 1 shows one correct solution. The query contains three steps. It (1) selects the rows in T_1 whose `date` column is "12/25" or "12/24", (2) groups T_2 on `oid` and calculates the maximum `val` below 50 for each group, and (3) joins the results computed from steps 1 and 2 using the predicate `uid = oid`. Note the use of the provided constants as part of the selection predicates in this case.

Subset of SQL Used in This Section. To focus on the key ideas without loss of generality, we restrict our synthesis algorithm to consider only a subset of SQL operators: selection, join, and aggregation (Figure 2); other features such as projection and union are discussed later in Section 3.

[1] http://stackoverflow.com/questions/39761697, http://stackoverflow.com/questions/14995024

```
Select *
From   (Select *
        From   T1
        Where  T1.date = 12/24
               Or T1.date = 12/25) T3
Join   (Select   oid, Max(val)
        From     (Select *
                  From   T2
                  Where  T2.val < 50) T4
        Group By oid) T5
On     T3.uid = T5.oid
```

Figure 1: A solution for the running example.

```
--select        --join          --aggregation
Select *        Select *        Select    c,α(c_t)
From   q        From   q1       From      q
Where  f        Join   q2       Group By  c
                On     f        Having    f
```

Figure 2: A subset of SQL grammar; q refers to an input table or a nested subquery, f refers to a predicate, α refers to an aggregation function, and c refers to a column name.

"aggregates" data from multiple rows into a single result, eg. SUM(), COUNT()

2.1 Enumerative Search with Equivalence-Class

Before explaining our algorithm, we first discuss the enumerative search approach, which is a class of widely adopted algorithms used to solve many synthesis problems. Such algorithms enumerate all programs in the program space of a given depth limit and retain only those consistent with the provided I/O examples. Previous enumerative synthesizers [2, 25, 37] adopt the concept of equivalence classes to optimize the search process: they group programs into equivalence classes based on their behaviors on the input example to compress the search space. The search then proceeds iteratively: within each stage, the algorithms enumerate all programs in the current search space, group them based on an equivalence metric, and proceed to the next stage. Assume that r is the average reduction rate from programs to values per stage; the total reduction rate at stage d is r^d, which makes the algorithms much more scalable than simple enumeration.

"equivalence-class based" refers to the idea that queries are grouped based on the output they produce when run against the input tables. If two queries produce the same output, they are conside red equivalent.

Figure 3 illustrates how this technique is applied to SQL query synthesis to solve the running example. Queries are grouped into equivalence classes based on their evaluation results on the input example, and each iteration enumerates all queries that can be constructed from these equivalence class representatives using an operator from Figure 2. However, when applied to SQL, this algorithm performs inefficiently, because grouping queries into equivalence classes cannot effectively compress the search space. There are a number of reasons for this. First, the main complexity of query synthesis comes from the generation of a large number queries per stage rather than a large number of stages. For instance, the number of intermediate queries generated at the last stage of Figure 3 is 554,856 even if we only consider the grammar shown in Figure 2. More importantly, grouping queries into equivalence classes represented by tables cannot effectively reduce search complexity: since tables are compound values that may contain thousands of cells (e.g., tables evaluated from nested `Joins`), evaluating queries and memoizing tables during the search process both contribute to large algorithmic overhead.

2.2 Our Approach

Our key insight for the challenges is to design a language of abstract queries to break down the synthesis process. Abstract queries resemble SQL queries except that they can contain uninstantiated filter predicates in the form of *holes*. This language lets us decompose the original synthesis problem into the following two subproblems that can be efficiently solved:

1. Synthesizing all abstract queries that can *potentially* be instantiated into queries satisfying the given I/O examples and pruning away the rest.
2. Searching for predicates to fill holes in these abstract queries, instantiating them into concrete ones, and determining which ones are consistent with the I/O examples.

We next describe the language of abstract queries and how this decomposition makes the synthesis problem tractable.

2.2.1 The Language of Abstract Queries

The grammar for abstract queries resembles the SQL grammar shown in Figure 2, except that all filter predicates (in `Where`, `Having`, `On` clauses) are replaced with holes "□" (we use $\tilde{q}$ to refer to abstract queries). As in SQL, abstract queries can also be composed (as in the case of `Join`).

Evaluating abstract queries is similar to evaluating SQL queries. For instance, an abstract `Select` or `Join` query is evaluated as a SQL query with its predicate hole replaced with `True`. We define the formal evaluation rules in Section 4. All evaluation rules satisfy the following *over-approximation property*: asumme $\tilde{q}$ is an abstract query; then, for any concrete query q instantiated from $\tilde{q}$, i.e., with all holes replaced with any syntactically valid predicates, the result of q is contained in the result of $\tilde{q}$. Thus, any abstract query whose result does not contain the output example will not lead to a valid query and can be pruned.

2.2.2 Problem 1: Searching for Valid Abstract Queries

We first search for abstract queries whose evaluation results contain the output example via enumerative search. Figure 4 shows the search process for the running example. The search process is similar to that shown in Figure 3, yet with different grammar and evaluation rules. Table $\tilde{T}_6$ in the figure contains the output example, so that the tree of queries from input tables to $\tilde{T}_6$ forms a candidate abstract query. All abstract queries that do not contain T_{out} are removed.

Since abstract queries do not contain filter predicates, far fewer intermediate tables are generated: for the running example, only 105 different intermediate tables (in total 2,710 cells) are generated in the last stage of Figure 4 compared

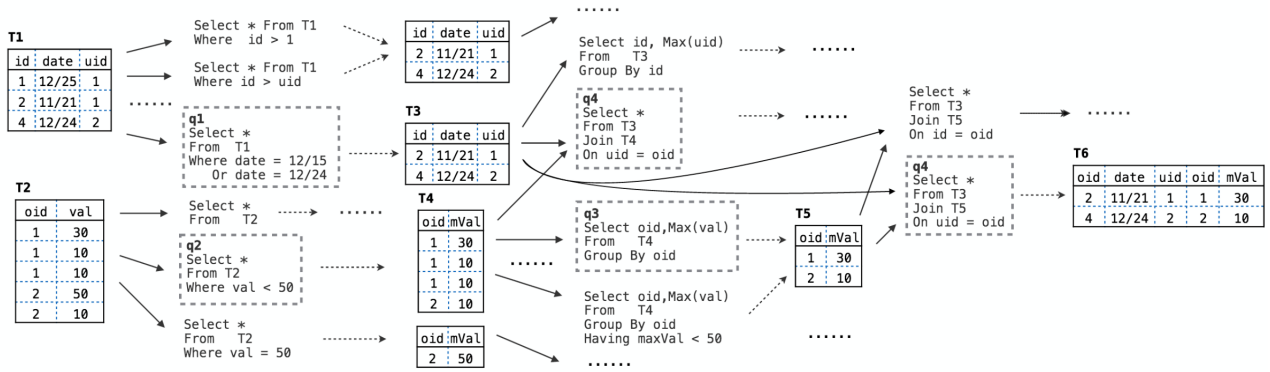


Figure 3: Equivalence-class based enumerative search algorithm, the subtree (queries in dash boxes) from T_1, T_2 to T_6 corresponds to the solution shown in Figure 1.

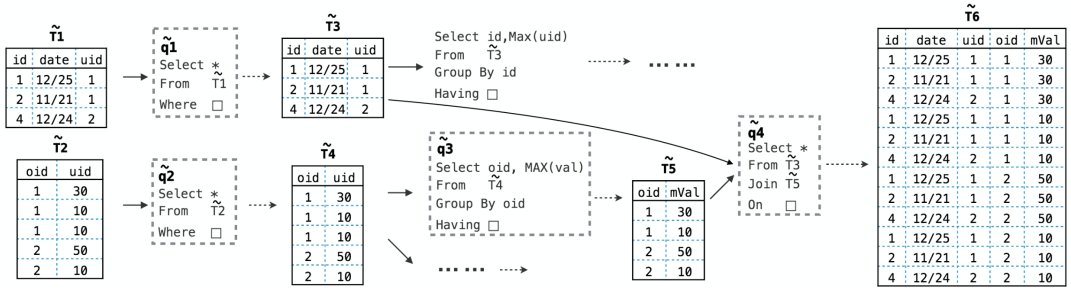


Figure 4: Searching for candidate abstract queries, where dash line arrows show evaluation of abstract queries. The tree from $\tilde{T}_1, \tilde{T}_2$ to $\tilde{T}_6$ corresponds to a candidate abstract query. Note the reduction in the number of tables and queries as compared to Figure 3.

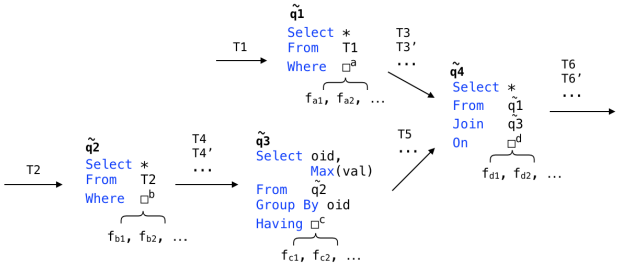
to 1,889 tables and 42,680 cells for the search process in Figure 3. Furthermore, the over-approximation property prunes as many as 90% of the abstract queries generated in the last stage.

2.2.3 Problem 2: Predicate Synthesis

Once candidate abstract queries are identified, we synthesize predicates to instantiate each of them. Specifically, for the running example, we need to find predicates for holes $\square^{a-d}$ (examples shown in Figure 5) to instantiate the abstract query to a SQL query whose evaluation result is T_{out}.

This search remains highly challenging since: (1) the number of candidate predicates is huge (4,692 syntactically different predicates for $\square^c$ alone, due to the large number of compound predicates built from conjunction/disjunction/negation) and (2) evaluating and memoizing intermediate tables remain expensive. We use two optimizations to address these issues.

Locally grouping candidate predicates. First, for each subquery of the given abstract query, we group its candidate predicates into equivalence classes. The idea is that if two candidate predicates behave the same on the evaluation result of a given abstract subquery,[2] their behaviors on the whole



$\square^a$	f_{a1} : date>=12/24 f_{a2} : date=12/25 Or date=12/24	$\square^b$	f_{b1} : T2.val<50 f_{b2} : T2.val<>50 .....
$\square^c$	f_{c1} : True f_{c2} : maxVal<50	$\square^d$	f_{d1} : T3.id=T5.uid f_{d2} : T3.oid=T5.uid .....

Figure 5: The abstract query in Figure 4 and candidate predicates for each hole.

abstract query remains identical, no matter how other holes in the query are instantiated. Hence, we need to retain only one such predicate as a representative of the equivalence class.[3]

For example, the two candidate predicates "`True`" and "`maxVal <= 50`" for $\square^c$ in $\tilde{q}_3$ in Figure 5 produce the same

[2] The behavior of a predicate on a table means evaluating the table and the predicate as a simple `Select` query.

[3] To ensure the algorithm's completeness, for each synthesized candidate query, our algorithm generates all different versions of the query by replacing predicate equivalence class representatives with all predicates in the group.

results when evaluated on $\tilde{T}_5$ in Figure 4; hence, only one of them needs to be retained to reduce search complexity. In total, the 4,692 candidate predicates for the hole $\square^c$ in Figure 5 are grouped into 21 equivalence classes, with a reduction rate of 200× compared to direct enumeration.

Encoding tables using bit-vectors. Our second optimization encodes intermediate tables using bit-vectors to improve search efficiency. The insight stems from the over-approximation property of evaluating abstract queries. Because of that, when searching for the instantiation of an abstract query $\tilde{q}$, we can use its abstract evaluation result $\tilde{T}$ together with a bit-vector β to represent the evaluation result of every instantiation of $\tilde{q}$: the size of β is same as the number of rows in $\tilde{T}$, and the i-th bit in β represents whether the the row i in $\tilde{T}$ appears in T.

For example, table T_4 in Figure 3 (evaluated from q_2) can be represented using the pair $(\tilde{T}_4, [11101])$ ($\tilde{T}_4$ is the evaluation result of the abstract query $\tilde{q}_2$ in Figure 4), since rows 0,1,2,4 of $\tilde{T}_4$ appear in T_4. Likewise, $T_{\texttt{out}}$ can be represented as the pair $(\tilde{T}_6, [010000000001])$ as shown in Figure 4.

Encoding tables into bit-vectors has two benefits. First, the intermediate results of the predicate search process can be fully represented using bit-vectors to reduce the memoization overhead, since the tables evaluated from abstract queries are shared among many and bit-vectors are cheap to memoize. Second, we can optimize operators on queries into bit-vectors operators benefiting from this representation; since many bit-vector operators require no materialization of tables, the computation overhead is also significantly reduced, as we will show in Section 5.2.

With these optimizations, the predicate synthesis algorithm efficiently searches for instantiations of abstract queries that are consistent with the provided I/O example. These candidate queries are later ranked and returned to the user.

2.3 Ranking and Interaction

Since the provided I/O example often does not completely specify a task, our algorithm returns multiple candidate queries that are consistent with the example. We use a heuristic to rank [12, 28] the queries returned from the main synthesis algorithm based on simplicity, naturalness and constant coverage, as will be discussed in Section 5.3. We present top-ranked queries to the user, who can provide a new example to the system and re-run the tool if needed.

3. The SQL Language

We introduce the definition of tables and briefly review our target language SQL in this section.

Table A table is a pair consisting of schema and content (Figure 6), where the schema is a list of name-type pairs and the content is a list of rows. Values we support include typed scalars or `null`. Additionally, as we adopt *bag-semantics* [22] for SQL, where duplicate rows are allowed in tables, and the equivalence between two tables is defined as bag equivalence, i.e., two tables are equal iff they mutually contain each other regardless of row ordering.

For readability, we use the notation $T_1 \subseteq T_2$ to represent that all rows in T_1 are contained by T_2, and the multiplicity of each row in T_1 is smaller or equal than its multiplicity in T_2.We also use $T_1 \cup T_2$ to refer to the union of contents in T_1 and T_2 (when their schema type are compatible); the schema of $T_1 \cup T_2$ is the same as T_1.

$$
\begin{array}{rcll}
T & ::= & \texttt{Table}(schema, content) & \text{(Table)} \\
schema & ::= & [c_1 : \tau_1, ..., c_m : \tau_m] & \text{(Schema)} \\
content & ::= & [r_1, ..., r_n] & \text{(Content)} \\
r & ::= & [v_1, ..., v_m] & \text{(Row)} \\
\tau & ::= & \texttt{int} \mid \texttt{double} \mid \texttt{string} \mid \texttt{date} \mid \texttt{time} & \text{(Type)}
\end{array}
$$

c1,c2, .. are column names and tau is the datatype

Figure 6: The definition of tables and auxiliary functions on tables. Metavariable c ranges over column names and v ranges over values.

SQL Figure 7 presents the grammar of SQL: a query q is formed by one of `Projection`, `Dedup`, `Select`, `Join`, `Aggr`, `LeftJoin`, `Union` or `As` constructors. The constructor $\texttt{Aggr}(\bar{c}, c_t, \alpha, q, f)$ corresponds to the aggregation query `Select` $\bar{c}, \alpha(c_t)$ `From` q `Group By` $\bar{c}$ `Having` f, the constructor $\texttt{Proj}(\bar{c}, q)$ corresponds to the projection query `Select` $c_1, ..., c_n$ `From` q, and others can be directly mapped to their concrete forms.

We omit the evaluation rules for SQL in our formal definition due to space limitations. We use the notation $[\![q]\!]$ to denote evaluating the query q into a table.

4. The Language of Abstract Queries

The language of abstract queries is key to the decomposition of the query synthesis problem. Figure 8 presents its grammar: an abstract query in the language is similar to a concrete SQL query except that filter predicates are replaced by uninstantiated holes ($\square$). Abstract queries and concrete queries have the following *instantiation/abstraction* relation.

Definition 1. *(Instantiation and Abstraction)* Given an abstract query $\tilde{q}$ and a query q, we call q an instantiation of $\tilde{q}$, if there exists a substitution $\phi = \{\square_1 \mapsto f_1, ..., \square_k \mapsto f_k\}$ (where k is the number of holes in $\tilde{q}$), such that substituting holes in $\tilde{q}$ with ϕ results in q, i.e., $\tilde{q}/\phi = q$. We also call $\tilde{q}$ the abstraction of q (by definition, only one abstraction exists for a query q).

Evaluation Rules Figure 9 shows evaluation rules for abstract queries, and the over-approximation property for these rules (as presented in Section 2.2.1) is formally defined below (Property 1). These rules are designed to ensure the satisfaction of the over-approximation property:

- Evaluating a table results in the table itself.
- When evaluating an abstract `Join` or `Select` query, the result is obtained by evaluating it with the predicate `True`.

$$
\begin{array}{rcll}
q & ::= & T & \text{(Named Table)} \\
 & | & \texttt{Proj}(\bar{c}, q) & \text{(Projection)} \\
 & | & \texttt{Dedup}(q) & \text{(De-duplication)} \\
 & | & \texttt{Select}(q, f) & \text{(Select)} \\
 & | & \texttt{Join}(q_1, q_2, f) & \text{(Join)} \\
 & | & \texttt{Aggr}(\bar{c}, c_t, \alpha, q, f) & \text{(Aggregation)} \\
 & | & \texttt{Union}(q_1, q_2) & \text{(Union)} \\
 & | & \texttt{LeftJoin}(q_1, q_2, \bar{c} = \bar{c}') & \text{(Left Join)} \\
 & | & \texttt{Rename}(q, \mathit{name}, \bar{c}) & \text{(Rename)} \\
f & ::= & \texttt{True} \mid v\ \mathit{binop}\ v & \text{(Predicates)} \\
 & | & \texttt{Exists}\ q \mid \texttt{Is Null}\ c & \\
 & | & f\ \texttt{And}\ f \mid f\ \texttt{Or}\ f \mid \texttt{Not}\ f & \\
v & ::= & c \mid \mathit{const} \mid \texttt{null} & \text{(Values)} \\
\alpha & ::= & \texttt{Max} \mid \texttt{Min} \mid \texttt{Avg} \mid \texttt{Count} \mid \texttt{Sum} & \text{(Aggregators)} \\
 & | & \texttt{Count-Distinct} \mid \texttt{Concat} & \\
\mathit{binop} & ::= & = \mid > \mid < \mid <= \mid >= \mid <> &
\end{array}
$$

Figure 7: SQL grammar. The metavariable T ranges over tables, c ranges over column names, *const* ranges over constant values, and *name* ranges over fresh table names. Bar notation is used to represent repetitive elements.

$$
\begin{array}{rcll}
\tilde{q} & ::= & T & \text{(Named Table)} \\
 & | & \texttt{Proj}(\bar{c}, \tilde{q}) & \text{(Projection)} \\
 & | & \texttt{Dedup}(\tilde{q}) & \text{(De-duplication)} \\
 & | & \texttt{Select}(\tilde{q}, \square) & \text{(Select)} \\
 & | & \texttt{Join}(\tilde{q}_1, \tilde{q}_2, \square) & \text{(Join)} \\
 & | & \texttt{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square) & \text{(Aggregation)} \\
 & | & \texttt{Union}(\tilde{q}_1, \tilde{q}_2) & \text{(Union)} \\
 & | & \texttt{LeftJoin}(\tilde{q}_1, \tilde{q}_2, \bar{c} = \bar{c}') & \text{(Left Join)} \\
 & | & \texttt{Rename}(\tilde{q}, \mathit{name}, \bar{c}) & \text{(Rename)} \\
\alpha & ::= & \texttt{Max} \mid \texttt{Min} \mid \texttt{Avg} \mid \texttt{Count} \mid \texttt{Sum} & \text{(Aggregators)} \\
 & | & \texttt{Count-Distinct} \mid \texttt{Concat} &
\end{array}
$$

Figure 8: The grammar of abstract queries. The symbol "$\square$" refers to an uninstantiated predicate, and the metavariable c ranges over column names.

- Given an abstract `LeftJoin` query, we first compute the evaluation results T_1, T_2 of its abstract subqueries and then return the union of (1) the left join result of T_1, T_2 and (2) the left join result of T_1 and an empty table whose schema is the same as T_2's. The second part ensures that the over-approximation property is satisfied no matter how $\tilde{q}_2$ is instantiated.
- Given an abstract `Aggr` query, we first evaluate the inner abstract subquery into a table T; we next compute the aggregation result for *all* tables that are contained by T with `Having` clause set to `True`, and we finally union the results. We must consider all possibilities in the rule to ensure the over-approximation property, since the grouping result is dependent to how its abstract subquery is instantiated.
- Evaluation rules for other abstract queries resemble their concrete version, since the over-approximation property propagates automatically from their subqueries.

Complexity and Optimization We measure the complexity of evaluating an abstract query using the number of SQL operators executed in the evaluation process and the output table's size. Assume the abstract queries are evaluated on an input table (or input tables) with r rows and c columns in total. From the rules in Figure 9, the worst-case measures for evaluating abstract `Aggr` queries are both exponential in r since we must compute the aggregation result of all tables contained by the input table and union the results, while the worst-case measures for all other abstract queries are polynomial in $r \times c$. Thus, the bottleneck of evaluating abstract queries lies in evaluating `Aggr` subqueries in an abstract query.

$$
\widetilde{eval}(T) = T \qquad \widetilde{eval}(\texttt{Proj}(\bar{c}, \tilde{q})) = [\![\texttt{Proj}(\bar{c}, \widetilde{eval}(\tilde{q}))]\!]
$$

$$
\widetilde{eval}(\texttt{Dedup}(\tilde{q})) = [\![\texttt{Dedup}(\widetilde{eval}(\tilde{q}))]\!]
$$

$$
\widetilde{eval}(\texttt{Select}(\tilde{q}, \square)) = [\![\texttt{Select}(\widetilde{eval}(\tilde{q}), \texttt{True})]\!]
$$

$$
\widetilde{eval}(\texttt{Join}(\tilde{q}_1, \tilde{q}_2, \square)) = [\![\texttt{Join}(\widetilde{eval}(\tilde{q}_1), \widetilde{eval}(\tilde{q}_2), \texttt{True})]\!]
$$

$$
\widetilde{eval}(\texttt{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square)) =
$$

$$
\texttt{let}\ T_0 = \widetilde{eval}(\tilde{q})\ \texttt{in}\ \left[\!\!\left[\texttt{Dedup}\left(\bigcup_{T \subseteq T_0} [\![\texttt{Aggr}(\bar{c}, c_t, \alpha, T, \texttt{True})]\!]\right)\right]\!\!\right]
$$

$$
\widetilde{eval}\ (\texttt{Union}(\tilde{q}_1,\ \tilde{q}_2)) = [\![\texttt{Union}(\widetilde{eval}(\tilde{q}_1), \widetilde{eval}(\tilde{q}_2))]\!]
$$

$$
\widetilde{eval}(\texttt{LeftJoin}(\tilde{q}_1, \tilde{q}_2, \bar{c} = \bar{c}')) =
$$

$$
\texttt{let}\ T_1 = \widetilde{eval}(\tilde{q}_1), T_2 = \widetilde{eval}(\tilde{q}_2), T_3 = [\![\texttt{Select}(T_2, \texttt{False})]\!]
$$

$$
\texttt{in}\ [\![\texttt{LeftJoin}(T_1, T_2, \bar{c} = \bar{c}')]\!] \cup [\![\texttt{LeftJoin}(T_1, T_3, \bar{c} = \bar{c}')]\!]
$$

$$
\widetilde{eval}(\texttt{Rename}(\tilde{q}, \mathit{name}, \bar{c})) = [\![\texttt{Rename}(\widetilde{eval}(\tilde{q}), \mathit{name}, \bar{c})]\!]
$$

Figure 9: The evaluation rules for abstract SQL. Notation $[\![q]\!]$ refers to evaluating a concrete query based on SQL semantics.

Fortunately, we can optimize the evaluation rules for abstract `Aggr` queries formed with many commonly used aggregation functions, including `Max`, `Min`, `Count` and `Count Distinct` to avoid the bottleneck. Since `Max` and `Min` return only existing values from the input tables, the output table size of an abstract `Aggr` query that uses such aggregates is polynomial to the size of its input table. This property lets us to simplify the evaluation rule into $[\![\texttt{Dedup}(T)]\!]$ (where T is the evaluation result of the inner abstract subquery) without violating the over-approximation property. An example is the evaluation of $\tilde{q}_3$ to $\tilde{T}_5$ in Figure 4. Similarly, the only new values produced by `Count` and `Count-Distinct` are column counts, and the evaluation rules for abstract queries containing them can likewise be simplified.

Property 1. *(Over Approximation)* Given an abstract query $\tilde{q}$ and a query q instantiated from $\tilde{q}$, $[\![q]\!] \subseteq \widetilde{eval}(\tilde{q})$.

Proof Sketch. By induction on the abstract query constructors, we can prove that every row in any instantiation of the abstract query is contained in its evaluation result. □

5. Synthesis Algorithm

We now introduce our synthesis algorithm (Algorithm 1). Given an example containing input tables I, output table T_{out},

and a set of constants C, the `Synthesis` algorithm constructs a set of candidate queries within the given time limits. For each *depth*, the algorithm first searches for all abstract queries that can potentially be instantiated into candidate queries (line 5); then, for each synthesized abstract query $\tilde{q}$, it constructs all of instantiations of $\tilde{q}$ that are consistent with the I/O example (lines 6,7); finally, the algorithm selects all synthesized queries whose score is beyond a system predefined threshold, and returns candidates to the user after ranking (lines 9,10).

```
1  Synthesis(I, T_out, C):
   // Input: input tables I, output table T_out, constants C
   // Output: Queries that are consistent with the input-output examples.
2    depth ← 1;
3    while timeout() = false:
4      S_q ← ∅;
5      S_q̃ ← SynthesizeAbstractQuery(I, T_out, depth);
6      foreach q̃ in S_q̃:
7        S_q ← S_q ∪ PredSynthesis(q̃, I, T_out, Const);
8      candidates ← {q | q ∈ S_q ∧ Score(q) > threshold};
9      if candidates ≠ ∅:
10       return Rank(candidates);
11     depth ← depth + 1;
12   return ∅;
```

Algorithm 1: The main synthesis algorithm.

5.1 Abstract Query Synthesis

The first part of the algorithm is abstract query synthesis, the goal of which is presented by the completeness condition below.

Definition 2. *(Completeness Condition)* Given an example (I, T_{out}, C) and search depth d, suppose $S_{\tilde{q}}$ is the set of abstract queries returned by the abstract query synthesis algorithm, then for all q in the query space that are consistent with the input output example whose length is with d, its corresponding abstract query $\tilde{q}$ is contained in $S_{\tilde{q}}$.

Our algorithm achieves this goal with the help of the over-approximation property of the abstract queries: as long as every abstract query whose evaluation result contains T_{out} is included in the result, our algorithm will not miss any abstract queries that can potentially be instantiated into queries that are consistent with the I/O example.

The algorithm (Algorithm 2) adopts an enumerative search approach for all abstract queries satisfying this condition: starting from the input tables I with depth $d = 1$, the algorithm iteratively (1) enumerates all abstract queries can be constructed from tables in S_T (by iterating over all query constructors for all tables in S_T) (line 4), (2) maintains a mapping between the abstract evaluation result of these abstract queries and their syntactical form using the map M (line 5), and (3) updates the S_T with newly generated tables (line 6). When the given search depth is reached, on lines 8-9, the algorithm retrieves all tables that fully contains T_{out} and decodes them into abstract queries with the help of the mapping M (by recursively substituting intermediate tables with their corresponding abstract subqueries). At the end of the phase, the algorithm returns a set of abstract queries.

The enumerative search approach can be applied efficiently in this phase since the size of the abstract query space is much smaller than that of the concrete query space, as all predicates are kept as holes. On the other hand, Algorithm 2 has one bottleneck: the number of tables enumerated at each stage is exponential to the maximum column size of input tables since our algorithm enumerates all possible grouping by columns for each aggregation query to ensure completeness. Fortunately, the input examples provided by the user are typically reasonably small and a majority of these abstract queries are immediately pruned in this phase if they cannot be instantiated into candidate queries. As a result, the number of abstract queries sent to the second phase algorithm is sufficiently small to ensure that the overall algorithm runs efficiently.

```
1  SynthesisAbstractQuery(I, T_out, depth):
   // Input: input output tables (I, T_out), search depth (depth)
   // Output: all abstract queries constructed from I within depth depth,
   // whose evaluation result fully contains T_out.
2    d ← 1, S_T ← I, M ← ∅;
3    while d ≤ depth:
4      S_q̃ ← S_q̃ ∪ EnumOneStepAbstractQuery(S_T);
5      M ← M ∪ {(T, q̃) | T = ẽval(q̃) ∧ q̃ ∈ S_q̃};
6      S_T ← {T | q̃ ∈ S_q̃ ∧ T = ẽval(q̃)};
7      d ← d + 1;
8    candidates ← {T | T ∈ S_T ∧ T_out ⊆ T};
9    return DecodeToAbstractQuery(candidates, M);
```

Algorithm 2: The abstract query synthesis algorithm; the subroutine `EnumOneStepAbstractQuery` enumerates all abstract queries that can be directly constructed table(s) in S_T.

Lemma 1. Algorithm 2 is complete (Definition 2).

Proof Sketch. This condition is guaranteed since (1) if q is a query consistent with the I/O example, its evaluation result contains the output example according to Property 1, and (2) Algorithm 2 searches for every abstract queries whose evaluation result contains T_{out}. □

5.2 Predicate Synthesis

Given an abstract query synthesized by the previous algorithm, the predicate synthesis algorithm synthesizes predicates for the abstract query to instantiate it into candidate queries. We first present a simple (but inefficient) algorithm that can solve this problem (Algorithm 3). First, the simple algorithm searches (with memoization, as in Algorithm 2) for all tables that can be obtained from queries instantiated from the abstract query $\tilde{q}$, by enumerating all predicates that can be filled into the predicate holes (line 1). Then, if the output table is found in the search process, the algorithm generates queries from the output table, according to its memoization result (function `GenQuery` in line 3). Algorithm 4 shows the

enumeration rules. The function `EnumAllPredicates` in the rule enumerates all possible syntactically different filter predicates for `Select`, `Join` and `Aggr` abstract queries, by iterating over all valid predicates defined by the SQL grammar (Figure 7).

```
SimplePredSynthesis(q̃, I, T_out, C):
// Input: an abstract query q̃, input output example I, T_out, C
// Output: candidate queries instantiated from q̃.
1  S_T ← DFS(q̃, I, C);
2  if T_out ∈ S_T:
3    return GenQuery(T_out);
4  return ∅;
```

Algorithm 3: A simple predicate synthesis algorithm.

```
DFS(T) : return {T};
DFS(Proj(c̄, q̃)) : return {⟦Proj(c̄, T)⟧ | T ∈ DFS(q̃)};
DFS(Dedup(q̃)) : return {⟦Dedup(T)⟧ | T ∈ DFS(q̃)};
DFS(Select(q̃, □)) :
  F ← EnumAllPredicates(Select(q̃, □), I, C);
  return {⟦Select(T, f)⟧ | T ∈ DFS(q̃) ∧ f ∈ F};
DFS(Join(q̃1, q̃2, □)) :
  F ← EnumAllPredicates(Join(q̃1, q̃2, □), I, C);
  return {⟦Join(T1, T2, f)⟧ | Ti ∈ DFS(q̃i) ∧ f ∈ F};
DFS(Aggr(c̄, ct, α, q̃, □)) :
  F ← EnumAllPredicates(Aggr(c̄, ct, α, q̃, □), I, C);
  return {⟦Aggr(c̄, ct, α, T, f)⟧ | T ∈ DFS(q̃) ∧ f ∈ F};
DFS(Union(q̃1, q̃2)) := {⟦Union(T1, T2)⟧ | Ti ∈ DFS(q̃i)};
DFS(LeftJoin(q̃1, q̃2, c̄ = c̄′)) :
  return {⟦LeftJoin(T1, T2, c̄ = c̄′)⟧ | Ti ∈ DFS(q̃i)};
DFS(Rename(q̃, name, c̄)):
  return {⟦Rename(T, name, c̄))⟧ | T ∈ DFS(q̃i)}
```

Algorithm 4: The DFS algorithm searching for all tables that can be evaluated from queries instantiated from an abstract query $\tilde{q}$. C and I refer to constants and input tables from the user. The function `EnumAllPredicates` enumerates all candidate predicates for the given abstract query.

Though simple, without any optimization, this algorithm is prohibitively expensive to be used in practice due to the challenges that arise from: (1) the large number of filter predicates to be searched, and (2) the expensive process of evaluating and memoizing tables during the search process.[4]

Our algorithm takes advantages of the over-approximation properties of abstract queries to speed up the algorithm by (1) locally grouping predicate candidates into equivalence classes for each hole to reduce the number of predicates to be enumerated and (2) encoding intermediate results into bit-vectors to increase search efficiency.

Predicate Enumeration and Grouping Given an abstract query $\tilde{q}$ constructed from `Select`, `Join` or `Aggr` that contain a predicate hole, the predicate enumeration algorithm (Algorithm 5) enumerates all filter predicates that can be filled into the hole of $\tilde{q}$ (lines 4, 10), groups them into equivalence classes based on their behavior on the evaluation result of $\tilde{q}$, (i.e., $\widetilde{eval}(\tilde{q})$) (lines 7-8, lines 11-12), and returns the representatives of all predicate groups.

The reason we can group these candidate predicates without losing completeness of the filter behaviors is shown below in Property 2. The key idea is that filter predicates in each equivalence class behaves indistinguishably in all possible instantiations of $\tilde{q}$ (no matter how subqueries of $\tilde{q}$ are instantiated).

Property 2. *(Predicate Equivalence)* Let $\tilde{q}$ be an abstract query formed by one of `Select`, `Join`, `Aggr` constructor with a hole $\square_0$, $T = \widetilde{eval}(\tilde{q})$, and f_1, f_2 be two predicate candidates for $\square_0$ that are equivalent on T, i.e., $[\![\texttt{Select}(T, f_1)]\!] = [\![\texttt{Select}(T, f_2)]\!]$. Then, for any ϕ, a substitution of holes in subqueries in $\tilde{q}$, we have $[\![\tilde{q}/(\phi \circ \{\square_0 \mapsto f_1\})]\!] = [\![\tilde{q}/(\phi \circ \{\square_0 \mapsto f_2\})]\!]$.

Proof Sketch. First, we have $T_0 = [\![\tilde{q}/(\phi \circ \{\square_0 \mapsto \texttt{True}\})]\!] \subseteq T$ according to the over-approximation property of abstract evaluation. Since f_1, f_2 are equivalent on T, they are also equivalent on any table that is contained by T, i.e., they are also equivalent on T_0, so that the equation is satisfied. □

With this property, instead of needing to search all predicates from `EnumAllPredicates` as in Algorithm 4, we only need to search predicate representatives returned by `EnumAndGroupPred`, which reduces the search space size. Note that when candidate queries are synthesized. We also expand representatives to all predicates in their equivalence classes to obtain different versions of the candidates, which ensures the completeness of syntactically different queries.

```
1  EnumAndGroupPred(q̃, Const, I):
   // Input: an abstract subquery q̃, constants Const, input tables I
   // Output: representative predicates
2    T ← eval~(q̃);
3    V ← schema(T) ∪ Const;
4    primitives ← EnumPrimitivePred(V, I);
5    rep ← ∅;
6    foreach p ∈ primitives:
7      if ∄f ∈ rep.([⟦Select(T, f)⟧] = [⟦Select(T, p)⟧]):
8        rep ← rep ∪ {p};
9    compound ← EnumCompoundPred(rep);
10   foreach p ∈ compound:
11     if ∄f ∈ rep.([⟦Select(T, f)⟧] = [⟦Select(T, p)⟧]):
12       rep ← rep ∪ {p};
13   return rep;
```

Algorithm 5: Predicate Enumeration Algorithm. The function `EnumPrimitivePred` enumerates primitive predicates using given values V and tables (tables are used for enumerating `Exists` predicates) and the function `EnumCompoundPred` generates compound predicates (`and`, `or`, `not`) from given predicates.

Encoding Tables into Bit-Vectors The second optimization is to encode of intermediate results in the search process to avoid the expensive computation and memoization caused

[4] Not doing so will make algorithm even less efficient as the number of different programs in the space is several magnitudes more than the number of their evaluation results.

$$\texttt{Encode}(T_0, T) = [b_1, ..., b_n]_n \ \textit{where}\ n = \texttt{rowNum}(T) \wedge\ T_0 \subseteq T$$

$$b_i = \begin{cases} 1 & \texttt{if}\ T[i] \in T_0 \\ & \quad \wedge\ \texttt{occur}(T[i], T[1, ..., i-1]) < \texttt{occur}(T[i], T_0) \\ 0 & \textit{otherwise} \end{cases}$$

$$\texttt{Decode}([b_1, ..., b_n]_n, T) = \texttt{Table}(\texttt{schema}(T), [r_i \mid r_i \in T \wedge b_i = 1]) \quad \textit{where}\ n = \texttt{rowNum}(T), i \in [1, n]$$

$$[b_1, ..., b_n]_n\ \&\ [b'_1, ..., b'_n]_n = [b_1 \& b'_1, ..., b_n \& b'_n]$$

$$[b_1, ..., b_n]_n +\!\!+ [b'_1, ..., b'_m]_m = [b_1, ..., b_n, b'_1, ..., b'_m]$$

$$[b_1, ..., b_n]_n \times [b'_1, ..., b'_m]_m = [c_{i*m+j} \mid c_{i*m+j} = b_{i+1} \& b_j]_{m \times n} \quad \textit{where}\ i \in [0, n-1], j \in [1, m]$$

Figure 10: Bit-vector operators, where & refers to bit-'and', $+\!\!+$ refers to list concatenatior and $\times$ refers crossproduct operator.

by inefficient table representation. Suppose q is a query instantiated from an abstract query $\tilde{q}$, $T_0 = [\![q]\!]$ and $T = \widetilde{eval}(\tilde{q})$; according to Property 1, we have $T_0 \subseteq T$, so that we can represent T_0 as a bit-vector based on T. As shown in Figure 10, we use a bit-vector β of length $\texttt{rowNum}(T)$ to represent T_0: we mark the i-th bit b_i as 1, if row i of T is also a row in T_0 (the second condition ensures that duplicates are correctly handled). Also, we can decode a bit-vector β of length $\texttt{rowNum}(T)$ into a table together with T, and the result is a table that contains only rows whose index bit is marked as '1' in T.

With this encoding, we reduce the original predicate synthesis problem (Algorithm 3) into a search problem whose intermediate results are bit-vectors, (Algorithm 6): given an abstract query $\tilde{q}$, our goal is to search over the space of bit-vectors that encodes instantiations of $\tilde{q}$ for ones that can be decoded to T_{out}.

Besides making memoization more efficient, this reduction also brings us the opportunity to simplify the computation shown in Algorithm 4, since many operators on table can be simplified into operators on bit-vectors that require no materialization of the tables (Figure 10). For example, when computing a bit-vector representation of a `Join` query result, we do not need to instantiate the Cartesian product table; instead, we only need to merge bit-vectors from its subqueries using the bit-wise crossproduct operator shown in Figure 10.

Additionally, our algorithm also conducts pruning in the bit-vector decoding process (line 3 of Algorithm 6) using output table T_{out}. Our algorithm precomputes a set S containing all bit-vectors that encode T_{out} based on the evaluation result of $\tilde{q}$; thus, we only need to check whether a bit-vector is in S to determine whether it leads to a candidate, which avoids the materialization of all tables in the last step.

The correctness of the search algorithm is ensured by the property below. The property suggests that given an abstract query $\tilde{q}$, if a table T can be found by the original algorithm, the new algorithm that operates on bit-vectors is also able to find a bit-vector whose decoding result is T, so that no table that the original algorithm found is missed after optimization.

$\underline{\texttt{PredSynthesis}}(\tilde{q}, I, T_{out}, Const)$:
1 $B \leftarrow \texttt{BVDFS}(\tilde{q}, I, Const)$;
2 $T \leftarrow \widetilde{eval}(\tilde{q})$;
3 $candidates \leftarrow \{\beta \mid \beta \in B \wedge \texttt{Decode}(\beta, T) = T_{out}\}$;
4 return $\texttt{GenQuery}(candidates)$;

Algorithm 6: The Predicate Synthesis Algorithm.

Property 3. *(Soundness of Encoding)* Given an abstract query $\tilde{q}$, we have $\texttt{DFS}(\tilde{q}) = \{\texttt{Decode}(\beta, \widetilde{eval}(\tilde{q})) \mid \beta \in \texttt{BVDFS}(\tilde{q})\}$.

Proof Sketch. The proof can be achieved by induction on constructors of $\tilde{q}$: for each abstract query constructor, we can prove that for each table T in $\texttt{DFS}(\tilde{q})$ there exists at least one bit-vector $\beta \in \texttt{BVDFS}(\tilde{q})$ whose decoding result on $\widetilde{eval}(\tilde{q})$ is T. □

At the end of this phase, our algorithm returns all possible instantiations of candidate abstract queries that are consistent with the I/O example. These queries are passed to the ranking and user interaction phase of our algorithm.

$\underline{\texttt{BVDFS}}(T)$: return $\texttt{Encode}(T, T)$;
$\underline{\texttt{BVDFS}}(\texttt{Proj}(\bar{c}, \tilde{q}))$: return $\texttt{BVDFS}(\tilde{q})$;
$\underline{\texttt{BVDFS}}(\texttt{Dedup}(\tilde{q}))$:
  $T \leftarrow \widetilde{eval}(\texttt{Dedup}(\tilde{q}))$, $B \leftarrow \texttt{BVDFS}(\tilde{q})$;
  return $\{\texttt{Encode}([\![\texttt{Dedup}(\texttt{Decode}(\beta, T))]\!], T) \mid \beta \in B\}$;
$\underline{\texttt{BVDFS}}(\texttt{Select}(\tilde{q}, \square))$:
  $B \leftarrow \{\texttt{EncodeFiltersToBV}(\texttt{Select}(\tilde{q}, \square), f) \mid f \in F\}$;
  $B_1 \leftarrow \texttt{BVDFS}(\tilde{q})$;
  return $\{\beta \& \beta_1 \mid \beta \in B \wedge \beta_1 \in B_1\}$;
$\underline{\texttt{BVDFS}}(\texttt{Join}(\tilde{q}_1, \tilde{q}_2, \square))$:
  $B \leftarrow \{\texttt{EncodeFiltersToBV}(\texttt{Join}(\tilde{q}_1, \tilde{q}_2, \square), f) \mid f \in F\}$;
  $B_1 \leftarrow \texttt{BVDFS}(\tilde{q}_1)$, $B_2 \leftarrow \texttt{BVDFS}(\tilde{q}_2)$;
  return $\{(\beta_1 \times \beta_2) \& \beta \mid \beta_i \in B_i \wedge \beta \in B\}$;
$\underline{\texttt{BVDFS}}(\texttt{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square))$:
  $B \leftarrow \{\texttt{EncodeFiltersToBV}(\texttt{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square), f) \mid f \in F\}$;
  $S_T \leftarrow \{\texttt{Decode}(\beta, \widetilde{eval}(\tilde{q})) \mid \beta \in \texttt{BVDFS}(\tilde{q})\}$;
  $T \leftarrow \widetilde{eval}(\texttt{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square))$;
  $B_1 \leftarrow \{\texttt{Encode}([\![\texttt{Aggr}(\bar{c}, c_t, \alpha, t, \texttt{True})]\!], T) \mid t \in S_T\}$;
  return $\{\beta_1 \& \beta \mid \beta_1 \in B_1 \wedge \beta \in B\}$;
$\underline{\texttt{BVDFS}}(\texttt{Union}(\tilde{q}_1, \tilde{q}_2))$:
  $B_1 \leftarrow \texttt{BVDFS}(\tilde{q}_1)$, $B_2 \leftarrow \texttt{BVDFS}(\tilde{q}_2)$;
  return $\{\beta_1 +\!\!+ \beta_2 \mid \beta_1 \in B_1 \wedge \beta_2 \in B_2\}$;
$\underline{\texttt{BVDFS}}(\texttt{LeftJoin}(\tilde{q}_1, \tilde{q}_2, \bar{c} = \bar{c}'))$:
  $T \leftarrow \widetilde{eval}(\texttt{LeftJoin}(\tilde{q}_1, \tilde{q}_2, \bar{c} = \bar{c}'))$;
  $S_{T1} \leftarrow \{\texttt{Decode}(\beta, \widetilde{eval}(\tilde{q}_1)) \mid \beta \in \texttt{BVDFS}(\tilde{q}_1)\}$;
  $S_{T2} \leftarrow \{\texttt{Decode}(\beta, \widetilde{eval}(\tilde{q}_2)) \mid \beta \in \texttt{BVDFS}(\tilde{q}_2)\}$;
  return $\{\texttt{Encode}([\![\texttt{LeftJoin}(T_1, T_2, \bar{c} = \bar{c}')]\!], T) \mid T_i \in S_{Ti}\}$;
$\underline{\texttt{BVDFS}}(\texttt{Rename}(\tilde{q}, name, \bar{c}))$: return $\texttt{BVDFS}(\tilde{q})$;
$\underline{\texttt{EncodeFiltersToBV}}(\tilde{q})$:
  $T \leftarrow \widetilde{eval}(\tilde{q})$, $r \leftarrow \texttt{rowNum}(T)$;
  $F \leftarrow \texttt{EnumAndGroupPred}(\tilde{q}, I, Const)$;
  return $\{[b_1 ... b_r]_r \mid f \in F \wedge b_i = \texttt{Eval}(f, T[i])\}$;

Algorithm 7: The optimized version of Algorithm 4 that searches over bit-vectors instead of tables. (All `BVDFS` functions are passed with the input table I and constant set C, we omitted them in the algorithm for simplicity consideration.)

We present the main theorem (completeness property) of our synthesis algorithm below.

Theorem 1. *(Completeness)* Given an example (I, T_{out}, C), suppose q is a query consistent with the provided I/O example with subquery depth is d; then, given unlimited timeout, Algorithm 1 can find q in the d-th iteration.

Proof Sketch. This theorem is the direct conclusion of Lemma 2 and Properties 2,3: the former ensures that the abstract query $\tilde{q}$ corresponding to q is included in the result at depth d of Algorithm 2, and the latter two ensure that the two optimizations do not change the result of the predicate enumeration algorithm (Algorithm 3). □

5.3 Ranking and User Interaction

After obtaining candidate queries from the main synthesis algorithm, our system heuristically scores and ranks them before returning them to users. Queries are scored based on the following criteria:

- *Simplicity.* Queries with simpler structures and filter predicates are scored higher than more complex ones. For example, queries with predicates formed by column comparisons are scored higher than those containing compound predicates or predicates formed by `Exists`. Similarly, queries with fewer nested subqueries are scored higher.
- *Predicate naturalness.* Queries containing more natural predicates are scored higher, e.g., the predicate `T1.id=T2.id` has a higher score than `T1.id>T2.value` since equi-join predicates are more commonly seen in practice.
- *Constant coverage.* Queries with a better coverage of constants are scored higher than those that lacks such coverage.

After ranking, our system returns the top candidates to the user. If none of top queries is correct, the user can return our system to refine the result by providing new I/O examples or aggregation functions that the query could use.

In general, crafting a new I/O example from scratch can be costly and ineffective at refining synthesis results. However, in the case of SQL, the new I/O example can be easily created by incrementally modifying the old one. As often observed from Stack Overflow, users typically create new examples by appending new rows or modifying contents of a few table cells in previously provided tables.

While our current ranking model is simple, it can already effectively rank the correct solution among top 5 of the candidates for a relatively large number of real-world benchmarks (see Section 6). In addition, our synthesis algorithm is orthogonal to ranking algorithms and interaction models. Hence, it can be integrated with other interaction models [20, 29, 31] to further increase usability. For example, our system can be integrated with COSETTE [6, 7], a solver for SQL queries, to further reduce the need to provide further I/O examples: given the top k synthesized queries, COSETTE can be used to compute a distinguishing set of input tables S such that applying each of the top k queries on S yields different query output. Using distinguishing tables, the user only need to choose the correct result that matches the input example to obtain the correct candidate query.

6. Evaluation

We implemented our algorithm in Java as a system called SCYTHE. In this section, we report the evaluation of SCYTHE on a set of 193 real-world benchmarks. The evaluation was performed on a quad-core Intel Core i7 2.67GHz CPU with 4GB memory for the Java VM.

6.1 Implementation Optimization

We adopted the following two additional optimizations in our implementation. First, our implementation avoided enumerating semantically equivalent queries, i.e., queries that are equivalent on all possible inputs, as much as possible to increase search efficiency. For example, we avoided enumerating both "`Select T1.a, T2.b From T1 Join T2`" and "`Select T1.a, T2.b From T2 Join T1`" by restricting the order of tables in `Join`. Second, we performed the grouping of predicates for abstract queries during the first phase (directly when they are enumerated) and made the grouping result sharable among different abstract queries, since different abstract queries may contain same abstract subqueries.

6.2 Benchmarks

We collected 193 benchmarks for evaluation, including 165 benchmarks from Stack Overflow and 28 benchmarks from prior work [40]. Each benchmark includes an input-output example pair and a reference solution (accepted answers on Stack Overflow or solutions provided in [40]). The statistics of benchmarks are shown in Figure 11 (for example size) and Figure 13 (for feature statistics). Besides, among the 193 benchmarks, there are 61 benchmarks contain constants provided by the user: 44 benchmarks contain exactly 1 constant, 15 benchmarks contain 2 constants, and 2 benchmarks contain 4 constants.

Stack Overflow Our main evaluation benchmarks are the 165 Stack Overflow posts, divided into three groups.

- *Development set* (so-dev): These 57 benchmarks are collected from posts under tags "sql", "moving-average", "great-n-per-group" in our development time.
- *Top-voted posts* (so-top): These 57 benchmarks are *all* top-voted (i.e., vote greater than 30) Stack Overflow posts containing input-output examples and are not marked as "duplicate posts,"[5] featuring most common tasks that end-users have trouble with. In our collection process, we *exhaustively* went through the search result and picked all posts about SQL programming that contained input-output

[5] With the search term "*[sql] is:question score:30.. lastactive:5y.. hasaccepted:yes duplicate:no*".

	No.	Size	SCYTHE	ENUM	Zhang.
so-dev	57	31.6	55 (96%)	33 (58%)	-
so-top	57	24.7	41 (72%)	33 (58%)	-
so-recent	51	34.6	29 (57%)	18 (35%)	-
ase13	28	56.8	18 (64%)	8 (29%)	15(53%)
total	193	34	143 (74%)	92 (48%)	-

Figure 11: Benchmark statistics and the the number of benchmarks that can be solved by different algorithms given 600 seconds limit. The "Size" column shows the average size of the benchmark examples (the size for a benchmark refers to the the number of cells in the input output tables plus the number of provided constants).

examples. We excluded posts that were not related to SQL programming (e.g., how to avoid SQL inject attack) or posts about how to write queries to update database (e.g., queries that start with `SET` or `UPDATE`).

- *Recent posts* (so-recent): These 51 benchmarks are *all* the posts containing input-output examples posted during the 14 day period between 2016-10-09 and 2016-10-23, with additional constraints that the posts should contain an accept answer and they are not marked as duplicate posts.[6] These tasks are more specialized and typically involve more complex features compared against top-voted questions, featuring long-tail end-user SQL problems. The collection process are the same as that for top voted posts.

ASE'13 Benchmarks We obtained an additional 28 benchmarks from SQLSynthesizer [40], containing 5 forum posts and 23 textbook questions.

6.3 Evaluation Process

We compared SCYTHE to the implementation of the equivalence-class based enumerative search algorithm described in Section 2 (ENUM). The same algorithm was used to rank the output in both cases.

For the ASE'13 benchmark, we also compared our algorithm to SQLSynthesizer [40] based on their paper report.[7] SQLSyntheiszer is a PBE system that synthesizes SQL queries using decision trees: queries in its grammar has a fixed template "`Select` $\bar{c}$ `From` $\bar{T}$ `Where` p `Group By` $\bar{c}$ `Having` p"; SQLSynthesizer heuristically enumerates tables ($\bar{T}$) to be joined and then adopts a decision tree designed for the template to learn column names and predicates to fill the the template.

We ran each benchmark using the different algorithms by feeding the provided input-output example provided by the user, subject to a 600 seconds time limit. If the algorithm terminated within the time limit, we checked the returned candidate queries against the reference solution: we marked the problem "solved" if the reference solution (or a semantically equivalent one determined manually) was among the top 5 returned result. If the algorithm failed to terminate or the correct query was not among top 5, we either 1) manually modified the original example based on the text description shown in the posts and re-evaluated the algorithm on the new example or 2) extracted the aggregation functions from the post and supplied it to the algorithm (if exists) and reran it with only provided aggregation functions. If the algorithm continued to fail after this interaction, we marked the problem as "failed". The statistics we collected during the evaluation process are reported below.

6.4 Number of Solved Benchmarks

Figure 11 shows the number of benchmarks solved by different algorithms, Figure 12 shows the performance comparison between SCYTHE and ENUM, and Figure 13 shows the statistics of the queries synthesized by SCYTHE.

SCYTHE solved 143 cases within the 600 seconds time limit (114 within 10 seconds). ENUM solved only 92 cases within the 600 seconds time limit (18 within 10 seconds). Cases that SCYTHE solved but ENUM did not are typically those containing higher subquery nesting levels or large example sizes. A comparison between SCYTHE and ENUM on benchmarks that both algorithms solved shows that SCYTHE is on average 57× faster (ranging from 7-200× faster).

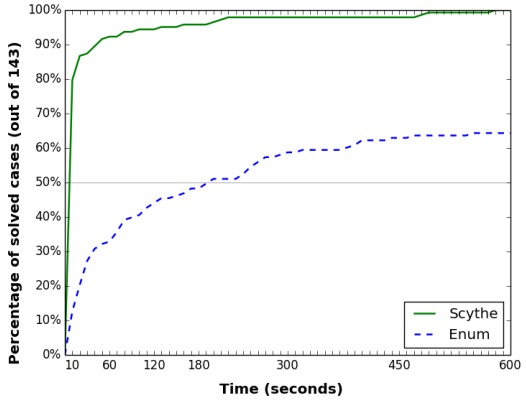


Figure 12: The percentage of benchmarks solvable with the increasing time limit specified by the x-axis (out of the 143 cases that SCYTHE successfully solved).

For the ASE'13 benchmark, SCYTHE solved 3 more cases than SQLSynthesizer since our algorithm supports more operators (`Exists`, `Left Outer Join`, and `Union`). Furthermore, although SCYTHE searched over a significantly larger space of queries (as it supports more types of subquery nesting, more operators, and have no template restriction in comparison to SQLSynthesizer), it showed no compromise in performance: for the 18 benchmarks that SCYTHE solved, 12 were solved in 10 seconds, and 6 of them were solved with over 10 seconds but within 120 seconds. In comparison, SQLSynthesizer solved 14 cases within 10 seconds and 1 with 120 seconds.

[6] With search term "*table result [sql] score:0.. is:question created:2016-10-09..2016-10-23 duplicate:no hasaccepted:yes*"

[7] We did not compare our algorithm against SQLSynthesizer on the Stack Overflow benchmarks as the tool is not publicly available.

	so-dev	so-top	so-recent	ase13	total
Join	40	31	18	10	90
Aggr	43	39	19	13	113
Left-Join	2	1	3	1	7
Union	1	0	9	1	11
Feature(>1)	35	30	18	9	92

Figure 13: Statistics for the occurrences of advanced SQL in the solutions synthesized by SCYTHE. Row "Feature(>1)" refer to queries containing more than one of the advanced operators shown above.

6.5 Algorithm Statistics

We present several statistics of our algorithm to demonstrate how different parts of our algorithm contributes to the overall efficiency improvements.

Abstract Grammar We measured the effectiveness of the abstract query synthesis algorithm in SCYTHE (Section 5) by measuring (1) the number of intermediate results generated by SCYTHE compared to ENUM, and (2) the search space reduction rate resulted from pruning away abstract queries that were inconsistent with the output.

For the first question, on the benchmarks that both algorithms can solve, the average number of tables generated by SCYTHE is 996, while that generated by ENUM is on average 7× more (ranging from 1.5-36×). Furthermore, for the 50 cases that only SCYTHE successfully solved, the average number of intermediate tables is 27,832 (max 471,316), thus the reduction is essential to allow SCYTHE to find answers to these more complex cases.

For the second question, pruning with abstract grammar reduces the size of search space by a factor of 2145× (ranging from 1.1-169,028×). The reduction rate is typically higher for cases whose output table size is larger or those whose solution contains aggregation subquery.

Predicate Synthesis We measured the effectiveness of the predicate synthesis algorithm (Section 5.2) in SCYTHE by measuring the reduction rate due to enumerating predicates that were in the same equivalence class rather than all syntactically equivalent ones. For each predicate hole in an abstract query, the average reduction rate from candidate predicates to their equivalence classes is 45,179× (ranging from 51-2,170,150×).

6.6 Effectiveness of Ranking

Next, among the 143 cases that SCYTHE solved, we need to provide additional input-output examples for 22 cases, specify aggregation functions for 9 cases, and provide both extra information (a second example or aggregation functions) for 3 cases to help SCYTHE disambiguate consistent queries. For every one of the cases that requires additional input-output examples, the average number of cells added to example (considering cells added to both input and output example) is 7.8 (maximum 17).

Besides, we also found 1 out of the 50 cases that SCYTHE failed to solve was failed due to our interaction model design limitation, as the query can only be constraint with at least two I/O example pairs at the same time, while our system allows only one I/O pair at a time.

Thus, though imperfect, our ranking algorithm remains effective in finding correct solutions from the large number of candidates.

6.7 Failed Cases

We classified the 50 cases that SCYTHE failed to solve into the following five categories to summarize the limitations of our algorithm: (1) cases requiring adding other standard SQL features to our grammar [f-exp1], (2) cases requiring additional non-standard SQL features (e.g., arithmetic expressions, date/string transformations, pivoting, window functions or dense ranking) [f-exp2], (3) cases that our language is able to express but failed due to scalability issues (e.g., increasing the level of nested queries) [f-scale], and (4) cases expressible but the corrected answer is unable to be disambiguated even after providing new examples [f-rank].

	f-exp1	f-exp2	f-scale	f-rank	total
so-dev	1	0	1	0	2
so-top	0	16	0	0	16
so-recent	0	17	5	0	22
ase13	0	0	9	1	10
total	1	33	15	1	50

Figure 14: Benchmarks that SCYTHE fails to solve.

As shown in Figure 14, a major fraction of the unsolvable cases (33 cases) are due to non-standard SQL features. Adding support for these specialized features requires that our algorithm work cooperatively with synthesizers from other domains, e.g., arithmetic expression synthesizer [30] or string solvers [12, 14], which we consider as future work.

There are also 15 cases SCYTHE failed to solve due to scalability. Those cases either requires a solution with highly nested subqueries (> 5) or contains large I/O example table size (> 60 cells). They meet the bottleneck of our synthesis algorithm, making the pruning costly and relatively ineffective. We noticed that a majority of them (9 cases) are the textbook questions from the ASE'13 benchmarks, which are designed for teaching purposes and appear rarely in online forums.

We also found 1 failure case caused by the unsupported SQL feature (keyword `Limit`) and 1 caused by interaction model limitation. Note that although other operators like `Except`, `All` do not directly appear in our grammar, they can be reformulated into queries written using the keyword `Exists` such that they can be expressed using our language.

6.8 Threat of Validity

Our main study and evaluation were based on the benchmarks from Stack Overflow, but it is possible that these benchmarks

do not representative all end-user tasks in practice. In analyzing the ranking effectiveness of SCYTHE, we measured only the number of cells needs to be added to the original example, not how hard it might be for end-users to come up with such additional examples. Studying end-user behavior could help us better address this problem.

7. Related Work

Programming by Examples SCYTHE is inspired by Programming by Example (PBE) applications: users of such systems specify a program using I/O examples, and the system subsequently synthesizes a program consistent with the examples. This approach has been used to synthesize data transformation programs [5, 12, 13, 29, 32], data extraction scripts [17], map-reduce programs [1, 33], data structure transformations [39] and also SQL queries [4, 36, 40].

SQLSynthesizer [40] and Query By Output [36] are two PBE systems for SQL queries. Both of them synthesize queries using the decision tree algorithm: the systems come with a set of templates and they complete holes in the template using the decision tree algorithm. Since the decision tree algorithm is limited by the facts that (1) the algorithm needs to build a different decision tree for every query template and (2) the decision tree size for a template is exponential to the number of rows and columns of the I/O example. Thus, neither systems support advanced SQL features like `Left Join`, `Union`, `Exists` and free-form subquery nesting, which is essential in solving real-world problems. In contrast, due to the scalability improvement of our two-phase synthesis algorithm, SCYTHE supports a wider range of SQL features used in practical settings.

Program Synthesis Algorithms Inductive program synthesis algorithms [2, 3, 11] can be roughly classified into the following five categories: (1) enumerative search algorithms [10, 18, 24, 25, 37], (2) constraint-solver aided synthesis [34, 35], (3) type-directed synthesis [9, 23], (4) version space algebra based inference [12, 16, 26], and (5) stochastic search [27].

Our synthesis algorithm is most closely related to enumerative search algorithms, which were adopted in Transit [37], Lenses [25], CodeHint [10] and AlphaRegex [18]. The first three systems are optimized using the concept of equivalence-class reduction where intermediate results are grouped together based on their evaluation results so that behaviorally equivalent programs are visited only once. Our algorithm differs from them in our development of abstract queries to decompose the search process, which is essential to solve the challenge of memoizing tables to scale up the synthesis algorithm. Besides, AlphaRegex, an enumerative synthesizer for regular expressions whose pruning strategy resembles the first phase of our algorithm, enumerates all regular expressions within the search space to find those consistent with user provided examples; it prunes intermediate regex skeletons based on their under/over-approximation to increase scalability. Our first phase algorithm differs in how over-approximations are computed: evaluating query skeletons requires the design of the abstract query language with different evaluation rules, while evaluating regular expression skeletons can be achieved purely using built-in operators.

Interactive Refinement Statistical ranking algorithms [28, 31] and interactive disambiguation interfaces [19, 20] have been proposed to improve PBE system accuracy. SCYTHE can potentially integrate such techniques to enhance the quality of the synthesized programs.

Other Related Techniques *Inductive logic programming* (ILP) [8, 21] is an approach adopted by the AI community for learning general logic representations from demonstrations: given N I/O examples together with their logical representations, ILP algorithms learn the set of programs that generalizes all examples using bottom-up searches. However, ILP cannot be directly applied to SQL query synthesis. First, it requires multiple examples to learn a general form. In our case, each task is specified using only one I/O pair, and tables cannot be trivially decomposed into smaller I/O examples for query synthesis (since doing so will likely change the user's intention). Second, using ILP requires logical representations of each I/O example, and constructing them from I/O examples for SQL is highly non-trivial, as shown by Polozov and Gulwani [26].

8. Conclusion

In this paper, we presented a system called SCYTHE, which efficiently synthesizes SQL queries from I/O examples. The key idea of our approach is the design an abstract language of queries to decompose the original complex synthesis problem into easier-to-solve subproblems. The evaluation of SCYTHE on a set of 193 real-world benchmarks shows that it can effectively and efficiently solve real world SQL query synthesis tasks.

Acknowledgements

This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1651489, and CNS-1563788; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; and gifts from Adobe, Amazon, and Google.

References

[1] M. B. S. Ahmad and A. Cheung. Leveraging parallel data processing frameworks with verified lifting. In *Proceedings of the Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.*, pages 67–83, 2016.

[2] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[3] R. Bodík and B. Jobstmann. Algorithmic program synthesis: introduction. *STTT*, 15(5-6):397–411, 2013. doi: 10.1007/s10009-013-0287-9.

[4] A. Cheung and A. Solar-Lezama. Computer-assisted query formulation. *Found. Trends Program. Lang.*, 3(1):1–94, June 2016. ISSN 2325-1107. doi: 10.1561/2500000018. URL `https://doi.org/10.1561/2500000018`.

[5] A. Cheung, A. Solar-Lezama, and S. Madden. Using program synthesis for social recommendations. In *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 1732–1736, 2012.

[6] S. Chu, C. Wang, K. Weitz, and A. Cheung. 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*, 2017. URL `http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf`.

[7] S. Chu, K. Weitz, A. Cheung, and D. Suciu. Hottsql: Proving query rewrites with univalent SQL semantics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '17 (To Appear)*, 2017.

[8] L. De Raedt. Inductive logic programming. In *Encyclopedia of Machine Learning*, pages 529–537. Springer, 2011.

[9] J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: A type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 802–815, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837629.

[10] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663. ACM, 2014.

[11] S. Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 13–24. ACM, 2010.

[12] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[13] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–328. ACM, 2011.

[14] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 105–116. ACM, 2009.

[15] D. Kini and S. Gulwani. Flashnormalize: Programming by examples for text normalization. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 776–783. AAAI Press, 2015. ISBN 978-1-57735-738-4.

[16] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[17] V. Le and S. Gulwani. Flashextract: A framework for data extraction by examples. In *ACM SIGPLAN Notices*, volume 49, pages 542–553. ACM, 2014.

[18] M. Lee, S. So, and H. Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 70–80. ACM, 2016.

[19] F. Li and H. V. Jagadish. NaLIR: an interactive natural language interface for querying relational databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 709–712. ACM, 2014.

[20] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 291–301. ACM, 2015.

[21] S. Muggleton, R. Otero, and A. Tamaddoni-Nezhad. *Inductive Logic Programming*, volume 38. Springer, 1992.

[22] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of sql queries. *ACM Transactions on Database Systems (TODS)*, 16(3):513–534, 1991.

[23] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 619–630, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2738007.

[24] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *ACM SIGPLAN Notices*, volume 49, pages 396–407. ACM, 2014.

[25] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310. ACM, 2016.

[26] O. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126. ACM, 2015.

[27] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. *ACM SIGPLAN Notices*, 48(4):305–316, 2013.

[28] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations.

[29] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment*, 5 (8):740–751, 2012.

[30] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *International Conference on Computer Aided Verification*, pages 634–651. Springer, 2012.

[31] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*, pages 398–414. Springer, 2015.

[32] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. *ACM SIGPLAN Notices*, 51(1):343–356,

2016.

[33] C. Smith and A. Albarghouthi. Mapreduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–340. ACM, 2016.

[34] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[35] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594340.

[36] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 535–548. ACM, 2009.

[37] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: Specifying protocols with concolic snippets. *SIGPLAN Not.*, 48(6):287–296, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462174.

[38] X. Wang, S. Gulwani, and R. Singh. Fidex: Filtering spreadsheet data using examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 195–213, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984030.

[39] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 508–521. ACM, 2016.

[40] S. Zhang and Y. Sun. Automatically synthesizing sql queries from input-output examples. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 224–234. IEEE, 2013.