Synthesizing Highly Expressive SQL Queries from Input-Output Examples

Abstract

SQL is the de facto language for manipulating relational data. Though powerful, SQL queries can be difficult to write SQL queries due to their highly expressive constructs. Using the programming-by-example paradigm to help users write SQL queries is 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 the large space of SQL queries as a result of its rich set of operators.

In this paper, we present a new scalable and efficient algorithm for synthesizing 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. 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 queries with 6 nested subqueries.

1. Introduction

Relational databases serve an important role in modern data management, and SQL is one of the most commonly used query language to manipulate relational data. SQL can be used for many basic tasks, such as selecting columns from a table; its rich features also makes it commonly used in many 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 posts related to SQL. In fact, many problems are so common among end users that they are grouped with popular tags, such as "greatest-n-per-group," "argmax," "moving-average," etc.

Though the problems are challenging to solve for end users, many of them can be concisely specified using input-output examples, as observed in many StackOverflow posts. Given the recent advances in programming-by-example (PBE) systems [6, 7, 9, 11, 21], building a tool that helps users write SQL by soliciting input-output (I/O) examples from users would alleviate their need to learn complex SQL constructs and idioms.

Prior work [27, 31] has developed automatic synthesizers for SQL queries using I/O examples. However, it only handles a small subset of the language and does not cover the wide range of tasks that appear in practice. We observe that difficulty of developing automatically synthesizer for SQL queries is resulted from the following unique features of SQL. 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 since a table can contain hundreds to thousands scalar values as the result of joins and unions. Third, unlike spreadsheet data transformation languages, whose operators can be decomposed and learned independently 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 determined 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 [6, 7, 18, 23] would remain scalable in SQL.

Our key insight to address the aforementioned challenges is to develop a new abstraction, namely 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 within them are replaced with *holes* that can be instantiated with any valid predicate. As operators in abstract queries are no longer parameterized by predicates, the search space of abstract queries is significantly smaller than the original one.

With abstract queries, the SQL synthesis problem is decomposed in 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. Then, in the sec-

ond phase, we search predicates for each synthesized abstract query to instantiate them into desired SQL queries and rank the results when multiple candidate queries are found. To make the search for predicates in the second phase efficient, we cluster predicates according to semantic equivalence classes and encode tables using bit-vectors.

We implemented our algorithm in a PBE tool called SCYTHE. To evaluate SCYTHE, we collected 155 real-world benchmarks from Stack Overflow and 28 benchmarks from previous work [31]. Results showed that SCYTHE efficiently and effectively 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 600s compared to the enumerative search algorithm [17, 28] and outperformed SQLSynthesizer [31] in their benchmarks with 4 more cases solved.

In sum, our paper makes the following contributions:

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

2. Overview

We now demonstrate our algorithm with a running example.

Problem Statement. We formalize a question from a user 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 [6, 20, 29]. First, we ask users to provide constants that will be used in predicates to make the synthesis process efficient. 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 input-output 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 input-output pair.

Running Example. We combine two Stack Overflow posts into a running example to demonstrate the full features of our algorithm.[a] This example contains two input tables $I = \{T_1, T_2\}$, an output table T_{out} (as shown below), along with 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 performs three tasks. 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 with predicate "`uid = oid`". Note the use of the provided constants as part of the selection predicates in this case.

```
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.

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 (see Figure 2); other features such as projection and union are discussed later in Section 3.

```
--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.

2.1 Enumerative Search with Equivalence-class

Before explaining our algorithm, we first discuss the enumerative search approach, a class of widely adopted algo-

[a] http://stackoverflow.com/questions/39761697/how-to-inner-join-three-tables-and-return-maximum-of-value-in-third
http://stackoverflow.com/questions/14995024/selecting-quarters-within-a-date-range?rq=1

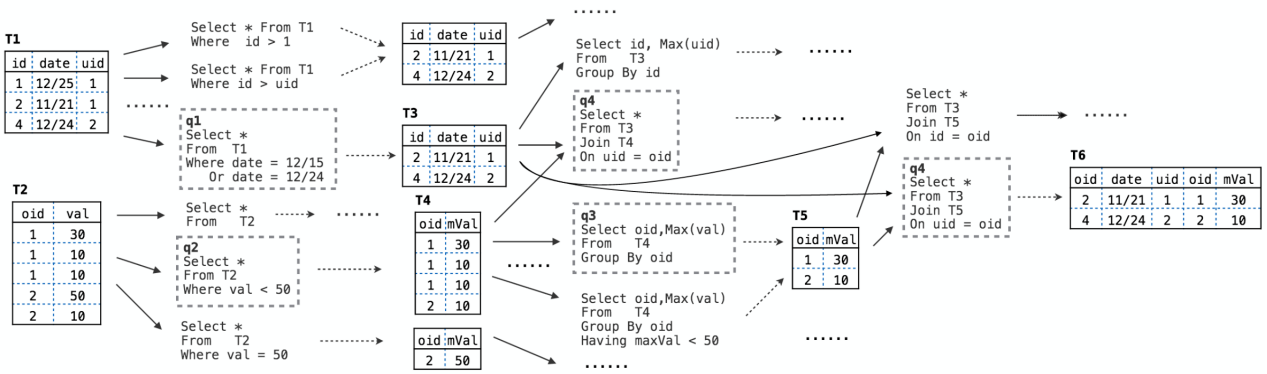


Figure 3: Equivalence-class based enumerative search algorithm, the subtree (queries in dash boxes) from T_1, T_2 to T_6 corresponds to the solution shown in Figure 1.

rithms used to solve many synthesis problems. Such algorithms enumerate all programs in the program space of a given depth limit and retain only those that are consistent with given input-output (I/O) examples. Previous enumerative synthesizers [1, 17, 28] 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 and then group them based on an equivalence metric, and move on to the next search space. 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 algorithm much more scalable than through simple enumeration.

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 examples, and each iteration enumerates all queries that can be constructed from these equivalence class representations using one of the operators in Figure 2. However, when applied to SQL, this algorithm performs inefficiently since grouping queries into equivalence classes cannot effectively compress the search space. There are a number of reasons for this. First, the major complexity of query synthesis comes from the generation of a large number queries per-stage rather than the number of stages. For instance, the number of intermediate queries generated at the last stage of Figure 3 is 554,856 even just within 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 up to a thousand cells (e.g., tables evaluated from nested `Joins`). Evaluating queries and memoizing tables during the search process constribute to large algorithmic overhead for the algorithm.

2.2 Our Approach

Our key insight for the challenges is to design a language for *abstract queries* to break down the synthesis process; abstract queries resemble SQL queries except they can contain uninstantiated operators (e.g., filter predicates) called *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 of the rest.
2. Synthesizing operators to fill the holes in the 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 decomposition makes the synthesis problem tractable.

2.2.1 The Language of Abstract Queries

The grammar for abstract queries is similar to 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 evaluting SQL queries. For instance, an abstract `Select` or `Join` query is evaluated as SQL query with the predicate hole replaced with the predicate `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 away.

2.2.2 Problem 1: Searching for Valid Abstract Queries

We search for abstract queries that contain the output relation by enumerating them based on their grammar. Fig-

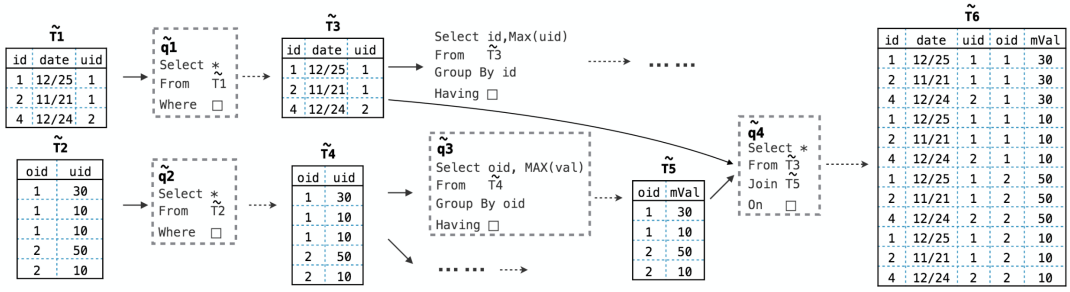


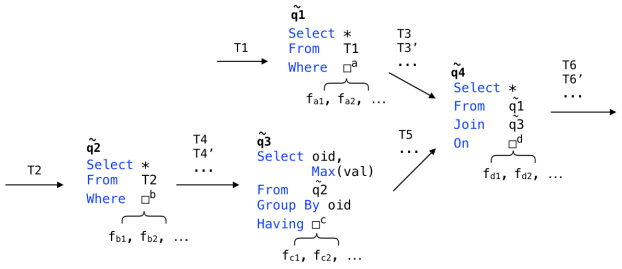
Figure 4: Searching for candidate abstract queries, where dash line arrows show evaluation of abstract queries. The tree from $\tilde{T}_1, \tilde{T}_2$ to $\tilde{T}_6$ corresponds to a candidate abstract query. Note the reduction in the number of tables and queries as compared to Figure 3.

ure 4 shows the search process for the running example. The search process is similar to that in Figure 3 except that its targets are abstract queries whose evaluation result contains the output example. Table $\tilde{T}_6$ in the figure fully contains the output relation, and the tree of queries from input tables to $\tilde{T}_6$ froms a candidate abstract query. All abstract queries that do not contain T_{out} are removed.

At this point, 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 to 1,889 tables and 42,680 cells for the enumerative search in Figure 3. Furthermore, the over-approximation property prunes away as much 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 $\square^{a-d}$ (examples shown in Figure 5) to instantiate the abstract query to a SQL query that evaluates to T_{out}.



$\square^a$	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.

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

Locally grouping candidate predicates. First, we group predicate candidates for each abstract subquery into equivalence classes. The idea is that if two predicate candidates behave the same on the evaluation result of a given abstract query, [b] then it does not matter how the other remaining holes in the abstract query are instantiated. Hence, we only need to retain one such predicate as a representative of the equivalence class. [c]

For example, the two candidate predicates "`True`" and "`maxVal <= 50`" for $\square^c$ in $\tilde{q}_3$ in Figure 5 produce the same results when evaluated on $\tilde{T}_5$ in Figure 4; hence, only one of them needs to be retained. In total, the 4,692 predicate candidates for the hole $\square^c$ in Figure 5 are grouped into 21 equivalence classes, with a reduction rate of 200× as compared to 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 any of its instantiation T: 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 as a pair $(\tilde{T}_4, [11101])$; ($\tilde{T}_4$ is the evaluation result of the abstract query $\tilde{q}_2$ in Figure 4), as rows 0,1,2,4 are present in T_4. Likewise, T_{out} can be encoded as a bit-vector $[010000000001]$ on $\tilde{T}_6$ in Figure 4.

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

[c] Our algorithm also expands representatives to all predicates in the group for candidate queries we synthesized to ensure the algorithm's completeness.

Encoding tables using bit-vectors has two benefits. First, since the effect of each predicate can be encoded as a bit-vector, the predicate synthesis problem is reduced to a bit-vector search. Second, operators on bit-vectors are cheaper to evaluate than those on tables since as many bit-vector operators require no materialization of tables, as we show in Section 5.2); therefore, using bit-vectors greatly improves the search efficiency.

With these optimizations, our predicate search algorithm enumerates over equivalence classes of predicates that are encoded using bit-vectors. When finished, all candidate queries satisfying the given I/O examples are retained and returned to the user after ranking.

2.3 Ranking and Interaction

The provided I/O examples are often not complete specifications of the target query; therefore, our algorithm finds multiple consistent candidate queries for each problem. We use a heuristic to rank [6, 20] the queries returned from the main synthesis algorithm, based on simplicity, naturalness and constant coverage, as discussed in Section 5.3. We present the top-ranked queries to the user, who can provide more samples and re-run our 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 represented as a schema-content pair (Figure 6), where the schema is a list of name-type pairs and the content is a list of rows. Values we support are either typed scalars or `null`. Additionally, as we adopt *bag-semantics* [14] for SQL, 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 the purpose of 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 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), and the schema of $T_1 \cup T_2$ is the same as T_1.

T	::=	$\texttt{Table}(schema, content)$	(Table)
$schema$	::=	$[c_1 : \tau_1, ..., c_m : \tau_m]$	(Schema)
$content$	::=	$[r_1, ..., r_n]$	(Content)
r	::=	$[v_1, ..., v_m]$	(Row)
τ	::=	`int` | `double` | `string` | `date` | `time`	(Type)

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

SQL The grammar of SQL is defined in Figure 7: 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.

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

Figure 7: SQL grammar where T ranges over input tables, c ranges over column names, $const$ ranges over constant values, and $name$ ranges over newly introduced table names. Bar notation is used to represent repetitive elements.

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

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

We omitted 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 the key to the query synthesis problem decomposition. 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 The evaluation rules for the abstract queries are shown in Figure 9, and the over-approximation property for these rules (as presented in Section 2.2.1) is formally defined below (Property 1).

The evaluation rules work as follows to ensure their satisfaction of the over-approximation property.

- The evaluation result of a table is the table itself.
- When evaluating a `Join` or a `Select` abstract query, the result is same as evaluating it with the predicate `True`.
- For a `LeftJoin` abstract query, we first compute the evaluation result of its abstract subqueries (T_1, T_2), then return the union of the following two table: 1) the left join of T_1, T_2 and 2) the left join result of T_1 and an empty table whose schema is same as T_2. The second part is to ensure that the over-approximation property is always satisfied no matter how $\tilde{q}_2$ is instantiated.
- For an `Aggr` abstract query, we first evaluate the inner abstract subquery into a table T, then we compute the aggregation result for *all* tables that are contained by T with `Having` clause set to `True` and finally union the result. The reason behind this rule is that the grouping result is dependant to how its abstract subquery is instantiated and we need to consider all possibilities to ensure the over-approximation property. Fortunately, although the number of tables contained by T is exponential to the row size, the total number of different actual aggregation values is much smaller (e.g., the different aggregation result of `Max`, `Min`, `Count` are in fact polynomial) and we can optimize the computation accordingly. An example of this rule is the evaluation from $\tilde{q}_3$ to $\tilde{T}_5$ in Figure 4.
- Evaluating other abstract queries is simply to evaluate them as concrete SQL queries, as the over-approximation property propagates automatically.

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, and proof for each row in any of the instantiation of the subquery is contained in its evaluation result. □

5. Synthesis Algorithm

With the design of the language of abstract queries, our synthesis algorithm is presented in Algorithm 1. Given an example containing input tables I, output table T_{out} and a constant set C, the `Synthesis` algorithm constructs a set of candidate queries within the given time limits. For each $depth$, the algorithm first synthesizes all abstract queries that can potentially be instantiated into candidate queries (line 5), then for each synthesized abstract query $\tilde{q}$, construct all of its instantiations that are consistent with the input-output example (lines 6,7), and finally selects all synthesized

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

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

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

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

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

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

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

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

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

queries whose score is beyond a system predefined threshold and returned the ranked candidates to the user (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

Given an input-output example I, T_{out}, C and a search depth $depth$, the goal is to synthesize a set of queries with the satisfaction of the following completeness condition.

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 by $S_{\tilde{q}}$.

Our algorithm achieves this completeness condition with the help of the over-approximation property of the abstract queries: as long as all abstract queries whose evaluation result contains T_{out} are included in the result, our algorithm will not miss any abstract queries that can potentially be

instantiated into queries that are consistent with the input-output example.

The abstract query synthesis 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, in 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.

Note that the enumerative search approach is applicable in this phase since the size of the abstract query space is much smaller than that of concrete query space, as all predicates are kept as holes. Besides ensuring completeness, the algorithm also effectively prunes abstract queries that cannot be instantiated into candidate queries help of the abstract evaluation rules (as will be shown in Section 6), which helps reduce the search space for the second phase algorithm to increase the overall algorithm efficiency.

1 SynthesisAbstractQuery$(I, T_{\text{out}}, depth)$:
// 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 $d \leftarrow 1, S_T \leftarrow I, M \leftarrow \emptyset$;
3 while $d \leq depth$:
4 $S_{\tilde{q}} \leftarrow S_{\tilde{q}} \cup$ EnumOneStepAbstractQuery(S_T);
5 $M \leftarrow M \cup \{(T, \tilde{q}) \mid T = \widetilde{eval}(\tilde{q}) \wedge \tilde{q} \in S_{\tilde{q}}\}$;
6 $S_T \leftarrow \{T \mid \tilde{q} \in S_{\tilde{q}} \wedge T = \widetilde{eval}(\tilde{q})\}$;
7 $d \leftarrow d + 1$;
8 $candidates \leftarrow \{T \mid T \in S_T \wedge T_{\text{out}} \subseteq T\}$;
9 return DecodeToAbstractQuery$(candidates, M)$;

Algorithm 2: The abstract query synthesis algorithm.

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

Proof Sketch. This condition is guaranteed since 1) if q is a query that is consistent with the input output example, the abstract evaluation result of $\tilde{q}$ contains the output example according to Property 1, and 2) algorithm 2 searches for all abstract queries whose evaluation result contains T_{out} so that $\tilde{q}$ is contained in the synthesis result. □

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 be used to solve this problem (Algorithm 3). First, the simple algorithm first searches (with memoization, as in Algorithm 2) for all tables 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 during the search process, the algorithm generates queries from the output table, based on the memoization result (function `GenQuery` in line 3). The enumeration rules are shown in Algorithm 4. 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$(\tilde{q}, I, T_{\text{out}}, C)$:
//Input: an abstract query $\tilde{q}$, input output example I, T_{out}, C
//Output: candidate queries instantiated from $\tilde{q}$.
1 $S_T \leftarrow$ DFS$(\tilde{q}, I, C)$;
2 if $T_{out} \in S_T$:
3 return GenQuery(T_{out});
4 return $\emptyset$;

Algorithm 3: A simple predicate synthesis algorithm.

DFS(T) : return $\{T\}$;
DFS$(\texttt{Proj}(\bar{c}, \tilde{q}))$: return $\{[\![\texttt{Proj}(\bar{c}, T)]\!] \mid T \in \texttt{DFS}(\tilde{q})\}$;
DFS$(\texttt{Dedup}(\tilde{q}))$: return $\{[\![\texttt{Dedup}(T)]\!] \mid T \in \texttt{DFS}(\tilde{q})\}$;
DFS$(\texttt{Select}(\tilde{q}, \square))$:
$F \leftarrow$ EnumAllPredicates$(\texttt{Select}(\tilde{q}, \square), I, C)$;
return $\{[\![\texttt{Select}(T, f)]\!] \mid T \in \texttt{DFS}(\tilde{q}) \wedge f \in F\}$;
DFS$(\texttt{Join}(\tilde{q}_1, \tilde{q}_2, \square))$:
$F \leftarrow$ EnumAllPredicates$(\texttt{Join}(\tilde{q}_1, \tilde{q}_2, \square), I, C)$;
return $\{[\![\texttt{Join}(T_1, T_2, f)]\!] \mid T_i \in \texttt{DFS}(\tilde{q}_i) \wedge f \in F\}$;
DFS$(\texttt{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square))$:
$F \leftarrow$ EnumAllPredicates$(\texttt{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square), I, C)$;
return $\{[\![\texttt{Aggr}(\bar{c}, c_t, \alpha, T, f)]\!] \mid T \in \texttt{DFS}(\tilde{q}) \wedge f \in F\}$;
DFS$(\texttt{Union}(\tilde{q}_1, \tilde{q}_2)) := \{[\![\texttt{Union}(T_1, T_2)]\!] \mid T_i \in \texttt{DFS}(\tilde{q}_i)\}$;
DFS$(\texttt{LeftJoin}(\tilde{q}_1, \tilde{q}_2, \bar{c} = \bar{c}'))$:
return $\{[\![\texttt{LeftJoin}(T_1, T_2, \bar{c} = \bar{c}')]\!] \mid T_i \in \texttt{DFS}(\tilde{q}_i)\}$;
DFS$(\texttt{Rename}(\tilde{q}, name, \bar{c}))$:
return $\{[\![\texttt{Rename}(T, name, \bar{c}))]\!] \mid T \in \texttt{DFS}(\tilde{q}_i)\}$

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.

As presented in Section 2, the simple algorithm is prohibitively expensive to be used in practice due to the challenges came 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 [d].

Our algorithm takes advantages of the over-approximation properties of abstract abstract to speed up the algorithm: 1)

[d] 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.

locally group predicate candidates into equivalence classes for each hole to reduce the number of predicates to be enumerated and 2) encode intermediate results into bit-vectors to increase search efficiency.

Predicate Enumeration and Grouping Given an abstract query $\tilde{q}$ constructed from `Select`, `Join`, `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 that we can group these candidate predicates without losing completeness of the filter behaviors is based on Property 2, as filter predicates in each equivalence class behaves indistinguishably in all possible instantiations of $\tilde{q}$ (no matter how abstract 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 is 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. Firstly, 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 with all predicates from `EnumAllPredicates` as in Algorithm 4, we only need to search with the predicate representatives returned by `EnumAndGroupPred` so that the search space size is reduced (we also expand representatives to all predicates in their equivalence classes for candidate queries are found by the algorithm, to ensure the completeness of syntactically different queries).

Encoding tables into bit-vectors The second optimization is the encoding of the intermediate results in the search process, to avoid the expensive computation and memoization caused by inefficient table representation. Suppose q is a query instantiated from an abstract query $\tilde{q}$, $T_0 = [\![q]\!]$ and $T = \widetilde{eval}(\tilde{q})$, then according to the over approximation property (Property 1), we have $T_0 \subseteq T$ and 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 based on T, and the result is a table that only contains rows whose index bit is marked as '1' in T .

```
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.

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

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

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

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

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

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

Figure 10: Operations on bit-vectors, where & refers to bit-'and'.

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 for all bit-vectors that can be constructed from instantiations of $\tilde{q}$ and generate queries from bit-vectors whose decoding result is T_{out}.

Besides making memoization more efficient, this reduction also brings us the opportunity to simplify the computation, defined by Algorithm 4, as many of the operations can be simplified into operations on bit-vectors that does not require the materialization of the tables (Figure 10). For example, when we search for bit-vectors resulted from a `Join` query, we no longer need to create a table from Cartesian product, but simply compute bit-vectors using bit-wise crossproduct operation in Figure 10.

Additionally, our algorithm also conducts a pruning with output table T_{out} in the bit-vector decoding process (line 3 of Algorithm 6): our algorithm precomputes all bit-vectors that potentially encodes T_{out} based on the evaluation result of $\tilde{q}$, and use this to avoid the materialization of all tables represented by the bit-vectors.

PredSynthesis$(\tilde{q}, I, T_{out}, Const)$:

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

Algorithm 6: The Predicate Synthesis Algorithm.

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

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

Proof Sketch. The proof can be achieved by induction done by induction on $\tilde{q}$, and for each abstract query constructor, prove that for each table T in $\texttt{DFS}(\tilde{q})$ there exists at least one bit-vector $\beta \in \texttt{BVDFS}(\tilde{q})$ that can be decoded to T when decoding on $\widetilde{eval}(\tilde{q})$. □

At the end of this phase, our algorithm returns all possible instantiations of candidates and the queries that are consistent with the input-output example, and they are passed to the ranking and user interaction phase of our algorithm.

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 that is consistent with the input-output example whose nested subquery depth is d, then given unlimited timeout q is able to be found by Algorithm 1 in the d-th iteration.

Proof Sketch. This theorem is the direct conclusion of Lemma 2 and Properties 2,3, as 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 ensures that the two optimizations does not change the result of the predicate enumeration algorithm (Algorithm 3). □

5.3 Ranking & User Interaction

Our algorithm adopts a heuristic based ranking algorithm to rank candidate queries returned from the main synthesis algorithm, where queries are ranked by three criteria: *simplicity*, *naturalness* and *constant coverage*: 1) we prefer simpler filter predicates than more complex ones, e.g. we give primitive predicates formed by column comparison a lower cost than those formed by `Exists` clause or compound predicates, 2) we prefer queries containing more natural predicates, e.g. predicate "`T1.id=T2.id`" is preferred over "`T1.id>T2.value`" since equi-join predicates are more commonly seen in practice and 3) we prefer queries that have a better coverage of constants than those not.

BVDFS(T) : return Encode(T, T);
BVDFS$(\texttt{Proj}(\bar{c}, \tilde{q}))$: return BVDFS$(\tilde{q})$;
BVDFS$(\texttt{Dedup}(\tilde{q}))$:
  $T \leftarrow \widetilde{eval}(\texttt{Dedup}(\tilde{q}))$, $B \leftarrow$ BVDFS$(\tilde{q})$;
  return $\{\texttt{Encode}([\![\texttt{Dedup}(\texttt{Decode}(\beta, T))]\!], T) \mid \beta \in B\}$;
BVDFS$(\texttt{Select}(\tilde{q}, \square))$:
  $B \leftarrow \{\texttt{EncodeFiltersToBV}(\texttt{Select}(\tilde{q}, \square), f) \mid f \in F\}$;
  $B_1 \leftarrow$ BVDFS$(\tilde{q})$;
  return $\{\beta \& \beta_1 \mid \beta \in B \wedge \beta_1 \in B_1\}$;
BVDFS$(\texttt{Join}(\tilde{q}_1, \tilde{q}_2, \square))$:
  $B \leftarrow \{\texttt{EncodeFiltersToBV}(\texttt{Join}(\tilde{q}_1, \tilde{q}_2, \square), f) \mid f \in F\}$;
  $B_1 \leftarrow$ BVDFS$(\tilde{q}_1)$, $B_2 \leftarrow$ BVDFS$(\tilde{q}_2)$;
  return $\{(\beta_1 \times \beta_2) \& \beta \mid \beta_i \in B_i \wedge \beta \in B\}$;
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 $\{\beta_1 \& \beta \mid \beta_1 \in B_1 \wedge \beta \in B\}$;
BVDFS$(\texttt{Union}(\tilde{q}_1, \tilde{q}_2))$:
  $B_1 \leftarrow$ BVDFS$(\tilde{q}_1)$, $B_2 \leftarrow$ BVDFS$(\tilde{q}_2)$;
  return $\{\beta_1 +\!\!+ \beta_2 \mid \beta_1 \in B_1 \wedge \beta_2 \in B_2\}$;
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 $\{\texttt{Encode}([\![\texttt{LeftJoin}(T_1, T_2, \bar{c} = \bar{c}')]\!], T) \mid T_i \in S_{Ti}\}$;
BVDFS$(\texttt{Rename}(\tilde{q}, name, \bar{c}))$: return BVDFS$(\tilde{q})$;
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 reduced version of Algorithm 4 that searches over bit-vectors. (All BVDFS functions are passed with the input table I and constant set C, we omitted them in the algorithm for simplicity consideration.)

After ranking all candidate queries, our algorithm returns top queries to the user as the synthesis result. When none of the top queries is accepted by user, we ask the user to 1) provide a new example to re-run the synthesis algorithm or 2) provide the aggregation functions that the query may use to the system for the purpose or better ranking.

As will be shown in our evaluation (Section 6), although our algorithm is not equipped with advanced ranking and interaction system [13, 21], it is able to effectively rank the correct solution among top 5 of the candidates for a large amount of real world benchmarks. Furthermore, as our main synthesis algorithm is orthogonal to the aforementioned advanced algorithm algorithm and interaction model, and they can be adopted in practice to further increase system effectiveness.

6. Evaluation

We have 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 eval-

uation is performed on a quad-core Intel Core i7 2.67GHz CPU with 4GB memory for the Java VM.

6.1 Implementation Optimization

We adopts the following two additional optimizations in our implementation. First, our algorithm avoids 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 avoid 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 perform the grouping of predicates for abstract queries during the first phase (directly when they are enumerated) and make 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[e] and 28 benchmarks from [31]. Each benchmark comes with an input-output example pair together with a reference solution (accepted answers on Stack Overflow or solutions provided in [31]). 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 constants, 15 benchmarks contain 2 constants and 2 benchmarks with 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 example sampled 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"[f], 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 contains input-output examples. We exclude posts that are 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 started with `SET` or `UPDATE`).
- *Recent posts* (so-recent): These 51 benchmarks are *all* 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.[g] 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.

	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: Statistics of the benchmarks and the the number of benchmarks can be solved by different algorithm given 600s timeout. 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).

ASE'13 benchmarks We obtained an additional 28 benchmarks from SQLSynthesizer [31], containing 5 forum posts and 23 textbook questions.

6.3 Evaluation Process

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

For the ASE'13 benchmark, we also report the comparison of our algorithm against SQLSynthesizer [31] based on their paper report.[h] SQLSyntheiszer is an PBE system that synthesizes SQL queries using decision trees: queries in their grammar has a fixed template "`Select` $\bar{c}$ `From` $\bar{T}$ `Where` p `Group By` $\bar{c}$ `Having` p" and 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 run 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 terminates within the time limit, we check the returned candidate queries against the reference solution: we mark the problem "solved" if the reference solution (or a semantically equivalent one determined manually) is among the top 5 returned result. If the algorithm fails to terminate or the correct query is not among top 5, we either 1) manually modify the original example based on english description shown in the posts and re-evaluate the algorithm on the new example or 2) extract the aggregation functions from the post and supply it to the algorithm (if exists) to the algorithm and rerun the algorithm with only provided aggregation functions. If

[e] http://stackoverflow.com/

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

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

[h] We didn't compare our algorithm against SQLSynthesizer on the Stack Overflow benchmarks as the tool is not publicly available.

the algorithm remains failing after this interaction, we mark the problem "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 (with 114 within 10 seconds), while ENUM only solved 92 cases within the 600 time limit (with 18 within 10 seconds). Cases that SCYTHE can solve but ENUM cannot are typically those containing higher subquery nesting level or large example size. A comparison between SCYTHE and ENUM on benchmarks that both algorithm can solve shows that SCYTHE is on average 57× faster (ranging from 7-200× faster).

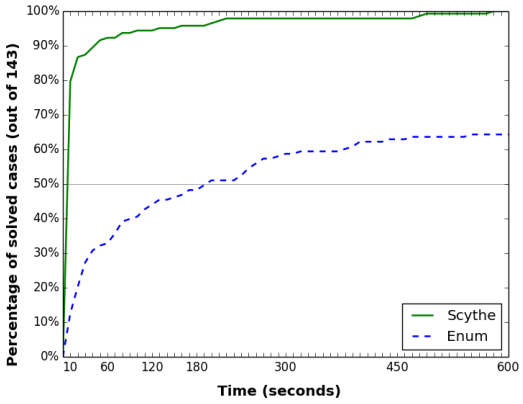


Figure 12: The percentage of benchmarks can be solved with the increasing time limit specified by x-axis (out of the 143 cases that SCYTHE can successfully solve).

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

6.5 Algorithm Statistics

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

Abstract Grammar We measure 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 are inconsistent with the output.

	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.

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 complete 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 measure the effectiveness of the predicate synthesis algorithm (Section 5.2) in SCYTHE by measuring the reduction rate due to enumerating predicates that are 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, need to specify aggregation functions for 9 cases, and provide both extra information 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 find 1 out of the 50 cases that SCYTHE failed to solve is failed due to our interaction model design limitation, as the query can only be constraint with at least two input-output example pairs at the same time, while our system allows only one input-output pair at a time.

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

6.7 Failure Analysis

We classify the 50 cases that SCYTHE cannot 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 requires our algorithm work cooperatively with synthesizers from other domains, e.g. arithmetic expression synthesizer [22] or string solvers [6, 8], which we consider as future work.

There are also 15 cases SCYTHE cannot solve due to scalability issue, and those cases features queries that contain highly nested subqueries (>6). Among them we noticed that a majority (9 cases) are the textbook questions from the ASE'13 benchmarks, which are designed for teaching purposes and rarely appear in online forums as we observed.

We also found 1 failure case is caused by the unsupported SQL feature (keyword `Limit`) and 1 caused by interaction model limitation. Note that although other operators like `Except`, `All` does not directly appear in our grammar, they can be desugared into queries written with the keyword `Exists` so that our language is able to express.

6.8 Threat of Validity

Our main study is based on the benchmarks from Stack Overflow, but it is possible that these benchmarks are not representative of all end-user tasks in practice. In analyzing the ranking effectiveness of SCYTHE, we only measured the number of cells needs to be added to the original example but not how hard it might be for end-users to come up with such additional examples. Studying end-user behavior can help us better address this problem.

7. Related Work

Programming by Examples SCYTHE is inspired by Programming by Examples (PBE) applications where users specify a program using input-output examples, and the system subsequently synthesizes a program that is consistent with the examples. This approach has been used to synthesize data transformation programs [6, 7, 21, 23], data extraction scripts [11], map-reduce programs [24], data structure transformation [30] and also SQL query synthesis [27, 31].

The two previous programming by examples systems for SQL queries are SQLSynthesizer [31] and Query By Output [27]. Both of them use a template based approach to synthesize queries, where queries are restricted to a fixed template and the systems complete holes in the template using a decision tree designed for the template to learn to fill the holes in the template. Due to the expressiveness restriction on the templates, neither approach provides support for several types of subquery nesting (e.g. aggregation after aggregation, join on multiple aggregation result) or advanced features like `Left Join`, `Union`, or `Exists` which are necessary for solving real world cases. In contrast, our approach is not template specific and is able to support a wider range of SQL features used in real world.

Program Synthesis Algorithms Inductive program synthesis algorithms [1, 2, 5] can be roughly classified into the following five categories: 1) enumerative search algorithms [4, 16, 17], 2) constraint-solver aided synthesis [25, 26], 3) type-directed synthesis [3, 15], 4) version space algebra based inference [6, 10, 18], and 5) stochastic search [19].

Our synthesis algorithm is most closely related to enumerative search algorithms [4, 17], which synthesize programs by enumerating programs within the program space specified by the language grammar. These algorithms are typically optimized using equivalence-class reduction, where intermediate results are grouped together based on their evaluation results to avoid re-visiting behaviorally equivalent programs. Our algorithm differs from existing approaches as we use abstract queries to decompose the search process to address the challenges caused by richness nature of SQL. The decomposition allows us to design better abstractions and more efficient search algorithms for the sub-problems and increase synthesis efficiency.

Interactive Refinement Different approaches has been proposed to improve the ranking algorithm and user interaction interface to make it easier for PBE systems to locate correct programs from the candidate set, including FlashProg [13], NaLIR [12] and BlinkFill [20]. SCYTHE can potentially integrate such techniques to enhance the quality of the synthesized programs.

8. Conclusion

In this paper, we presented a system called Scythe that can efficiently synthesize SQL queries from input-output exam-

ples. The key idea of our approach is to design an abstract language of queries to decompose to original complex synthesis problem into easier-to-solve subproblems. The evaluation of Scythe on a set of 193 real-world benchmarks shows that Scythe can effectively and efficiently solve real world SQL query synthesis tasks.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

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

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

[23] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. *ACM SIGPLAN Notices*, 51(1):343–356, 2016.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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