PATSQL: Efficient Synthesis of SQL Queries from Example Tables with Quick Inference of Projected Columns

Keita Takenouchi
NTT DATA
Tokyo, Japan
Keita.Takenouchi@nttdata.com

Takashi Ishio
Nara Institute of Science and Technology
Nara, Japan
ishio@is.naist.jp

Joji Okada
NTT DATA
Tokyo, Japan
Joji.Okada@nttdata.com

Yuji Sakata
NTT DATA
Tokyo, Japan
Yuji.Sakata@nttdata.com

ABSTRACT

SQL is one of the most popular tools for data analysis, and it is now used by an increasing number of users without having expertise in databases. Several studies have proposed programming-by-example approaches to help such non-experts to write correct SQL queries. While existing methods support a variety of SQL features such as aggregation and nested query, they suffer a significant increase in computational cost as the scale of example tables increases. In this paper, we propose an efficient algorithm utilizing properties known in relational algebra to synthesize SQL queries from input and output tables. Our key insight is that a projection operator in a program sketch can be lifted above other operators by applying transformation rules in relational algebra, while preserving the semantics of the program. This enables a quick inference of appropriate columns in the projection operator, which is an essential component in synthesis but causes combinatorial explosions in prior work. We also introduce a novel form of constraints and its top-down propagation mechanism for efficient sketch completion. We implemented this algorithm in our tool PATSQL and evaluated it on 226 queries from prior benchmarks and Kaggle's tutorials. As a result, PATSQL solved 68% of the benchmarks and found 89% of the solutions within a second. Our tool is available at https://naist-se.github.io/patsql/.

PVLDB Reference Format:
Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. PATSQL: Efficient Synthesis of SQL Queries from Example Tables with Quick Inference of Projected Columns. PVLDB, 14(11): 1937 - 1949, 2021.
doi:10.14778/3476249.3476253

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/NAIST-SE/PATSQL.

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476253

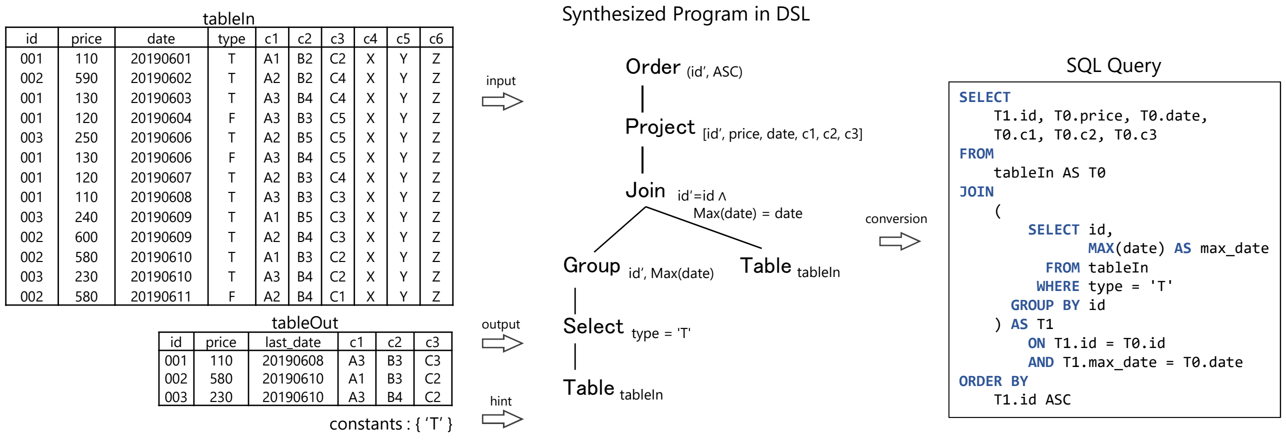
1 INTRODUCTION

SQL is a query language that manages data in relational databases, and it is commonly used in a wide range of software systems ranging from web applications to banking systems. With the growing popularity of data analysis in recent years, an increasing number of users without having expertise in databases are using SQL [22]. However, it is not easy for such non-experts to write SQL queries since they need to express a variety of analytical needs.

Programming-by-example (PBE) is a program synthesis technique that automatically synthesizes programs from input and output (I/O) examples. PBE is known to be a practical approach to help non-experts to implement programs [15, 16]. In the context of SQL, queries are automatically synthesized from I/O tables that the user provides as an example. In recent years, several studies have proposed techniques to automatically synthesize SQL queries [7, 31, 37, 43] or table manipulation programs [5, 12] from I/O tables. SCYTHE [37] synthesizes SQL queries that support highly expressive features such as projection, join, grouping, aggregation and union. SCYTHE even supports the synthesis of nested queries, which have other queries inside. These features enable users to obtain practical queries that gain insights from accumulated data.

However, these methods commonly have serious performance issues depending on the scale of I/O tables. The reason is that these algorithms need to enumerate a large number of candidates for each part of a program. For example, SCYTHE suffers an exponential increase in computational cost as the number of columns increases since it needs to enumerate all the permutations of them. However, the schemas of tables in real-world databases are not small, and thus the user fails to obtain queries in most of the practical scenarios.

In this paper, we propose a sketch-based algorithm that synthesizes SQL queries from I/O tables. The illustrative example is shown in Figure 1. Our algorithm is efficient in terms of the execution time and the scale of supported tables. While it requires fewer hints (i.e. constants used in a query) than in prior work [37], it maintains the high expressiveness of synthesized queries including aggregations, nested queries and window functions. To the best of our knowledge, this is the first SQL synthesizer that supports window functions such as cumulative sum and rank of each row, which have gained more popularity in database communities over the past years. Note that our algorithm does not depend on the column names because



tableIn

id	price	date	type	c1	c2	c3	c4	c5	c6
001	110	20190601	T	A1	B2	C2	X	Y	Z
002	590	20190602	T	A2	B2	C4	X	Y	Z
001	130	20190603	T	A3	B4	C4	X	Y	Z
001	120	20190604	F	A3	B3	C5	X	Y	Z
003	250	20190606	T	A2	B5	C5	X	Y	Z
001	130	20190606	F	A3	B4	C5	X	Y	Z
001	120	20190607	T	A2	B3	C4	X	Y	Z
001	110	20190608	T	A3	B3	C3	X	Y	Z
003	240	20190609	T	A1	B5	C3	X	Y	Z
002	600	20190609	T	A2	B4	C3	X	Y	Z
002	580	20190610	T	A1	B3	C2	X	Y	Z
003	230	20190610	T	A3	B4	C2	X	Y	Z
002	580	20190611	F	A2	B4	C1	X	Y	Z

tableOut

id	price	last_date	c1	c2	c3
001	110	20190608	A3	B3	C3
002	580	20190610	A1	B3	C2
003	230	20190610	A3	B4	C2

constants : { 'T' }

SQL Query

```
SELECT
    T1.id, T0.price, T0.date,
    T0.c1, T0.c2, T0.c3
FROM
    tableIn AS T0
JOIN
    (
        SELECT id,
               MAX(date) AS max_date
          FROM tableIn
         WHERE type = 'T'
       GROUP BY id
    ) AS T1
        ON T1.id = T0.id
        AND T1.max_date = T0.date
ORDER BY
    T1.id ASC
```

Figure 1: The overview of our approach. It synthesizes a program in our DSL, and converts it into an SQL query.

they do not necessarily indicate the correspondence between input and output columns.

To achieve the efficiency, we integrate properties known in relational algebra into sketch-based program synthesis. Our key insight is that the projection operator (i.e. `Select` keyword in SQL) in a program sketch can be lifted above other operators by applying transformation rules in relational algebra without changing the semantics of the program. By leveraging this insight, we can avoid a combinatorial explosion that existing methods suffer during the completion of the projected columns.

We implemented this algorithm in our tool PATSQL. To evaluate it, we used 193 SQL queries from Stack Overflow and a textbook on databases, as in prior work. We also collected 33 queries from Kaggle's tutorials, which deal with real table schemas used for data analysis. The result shows PATSQL significantly outperforms a state-of-the-art method SCYTHE in terms of the execution time and the scalability of I/O tables it can handle. In particular, PATSQL solved 68% of the benchmarks while SCYTHE solved 57% of them. Moreover, PATSQL found 136 solutions (89% of the solved benchmarks) within a second while SCYTHE found only 28 solutions.

The main contributions of this paper are as follows.

- We propose a novel technique that synthesizes SQL queries from I/O tables. It has strengths in both the execution time and the scale of I/O tables it can handle. It also supports the high expressiveness of synthesized queries including grouping, aggregation, nested query and window functions only with hints about used constants.
- We describe the synthesis algorithm that focuses on the efficient completion of the projected columns by leveraging properties known in relational algebra.
- We propose a user interface that is inspired by the concept of live programming. The user can get real-time feedback on the synthesized queries each time s/he updates tables.
- We implement the algorithm in our tool PATSQL and evaluate it on various benchmarks including queries that we collected from Kaggle's tutorials. The results show that PATSQL significantly outperforms a state-of-the-art algorithm SCYTHE in terms of the execution time and the scalability of I/O tables.

2 OVERVIEW

In this section, we provide an overview of our approach with an illustrative example and define the problem we solve in this paper.

2.1 Illustrative Example

We show an example to illustrate the usefulness of our PBE method. Suppose that a non-expert user wants to extract data from a relational database. A table named `tableIn` keeps track of the price history of items, and the schema consists of *item ID*, *price*, *updated date*, *type*, and other six columns *c1*, . . . ,*c6*. Now the user wants to know *the latest price and the updated date along with the values in c1, c2 and c3 for each item whose type is "T"*, but has difficulty writing such a query since the user is not familiar with the syntax and semantics of SQL.

Instead of writing the query from scratch, the user can use our tool PATSQL. First, the user gives an example of the I/O tables that should be satisfied by the query. The user also needs to provide hints for synthesis, i.e., the constants used in predicates in the query. The I/O tables and hint are shown on the left of Figure 1. Here `tableIn` and `tableOut` are the input and output tables, respectively. The constant value "T" is given as a hint. Then, PATSQL synthesizes a program in our DSL (see Section 3 for the details) that satisfies the I/O tables and makes use of the hints provided. In this case, PATSQL synthesizes the program shown in the center of Figure 1. This program first filters out the records that do not belong to the type "T" from the input table, and it calculates the maximum date for each item. Then, it joins the aggregation result and the input table with two key pairs, and it extracts the desired columns and sorts records. Finally, PATSQL converts the synthesized program in DSL into a SQL query and returns it as a result. The synthesized SQL query is shown on the right of Figure 1. The user can extract the desired records by executing it on the original table. Since the query has a nested structure having aggregation with grouping, it is not easy for the user to write it without the aid of PATSQL. Note that the conversion from our DSL to SQL is straightforward because our DSL is based on relational algebra, which is a theoretical foundation for SQL. Hence, we omit its details in this paper and focus on the synthesis algorithm in our DSL.

$$
\begin{array}{rcl}
\langle table \rangle & ::= & \texttt{Table}(tname) \\
& | & \texttt{Order}(\langle table \rangle, [\langle key \rangle_1, \ldots, \langle key \rangle_n]) \\
& | & \texttt{Distinct}(\langle table \rangle) \\
& | & \texttt{Project}(\langle table \rangle, [\langle col \rangle_1, \ldots, \langle col \rangle_n]) \\
& | & \texttt{Select}(\langle table \rangle, \langle pred \rangle) \\
& | & \texttt{Group}(\langle table \rangle, [cname_1, \ldots, cname_m], [\langle gc \rangle_1, \ldots, \langle gc \rangle_n]) \\
& | & \texttt{Window}(\langle table \rangle, [\langle win \rangle_1, \ldots, \langle win \rangle_n]) \\
& | & \texttt{Join}(\langle table \rangle, \langle table \rangle, \langle pairs \rangle) \\
& | & \texttt{LeftJoin}(\langle table \rangle, \langle table \rangle, \langle pair \rangle) \\
\langle key \rangle & ::= & (\langle col \rangle, \texttt{Asc}) \mid (\langle col \rangle, \texttt{Desc}) \\
\langle col \rangle & ::= & cname \mid \langle gc \rangle \\
\langle gc \rangle & ::= & \langle agg \rangle(\langle col \rangle) \\
\langle win \rangle & ::= & (\langle func \rangle, col, [cname_1, \ldots, cname_m], \langle key \rangle) \\
\langle pairs \rangle & ::= & \langle pair \rangle \land \cdots \land \langle pair \rangle \\
\langle pair \rangle & ::= & \langle col \rangle = \langle col \rangle \\
\langle pred \rangle & ::= & \langle clause \rangle \lor \cdots \lor \langle clause \rangle \\
\langle clause \rangle & ::= & \langle prim \rangle \land \cdots \land \langle prim \rangle \\
\langle prim \rangle & ::= & \langle col \rangle\ \langle binop \rangle\ const \mid \texttt{IsNull}(cname) \mid \texttt{IsNotNull}(cname) \\
\langle agg \rangle & ::= & \texttt{Max} \mid \texttt{Min} \mid \texttt{Count} \mid \texttt{Sum} \mid \texttt{Avg} \mid \texttt{CountDistinct} \\
& | & \texttt{ConcatComma} \mid \texttt{ConcatSpace} \mid \texttt{ConcatSlash} \\
\langle func \rangle & ::= & \texttt{Max} \mid \texttt{Min} \mid \texttt{Count} \mid \texttt{Sum} \mid \texttt{Rank} \\
\langle binop \rangle & ::= & = \mid < \mid <= \mid > \mid >= \mid <>
\end{array}
$$

Figure 2: The grammar of our DSL. *tname* denotes the name of a input table. *cname* denotes a column name. *const* denotes a constant value. *n* is the size of a vector, and *m* is the size of grouping keys, which is limited to two or less.

In general, the schemas of example tables given to a PBE tool should be the same as the schemas of the original tables stored in a database. This is because queries synthesized from simplified tables can cause SQL errors when it is executed in the database. For example, suppose the user deletes the columns c1, c2 and c3 from the output example in Figure 1 because the user mistakenly thinks these columns do not affect the structure of a synthesized query. Then, a PBE tool may well return the following query, which is much simpler than the desired query in Figure 1.

```
SELECT id, min(price), max(date)
FROM tableIn WHERE type = 'T' GROUP BY id
```

After obtaining this query, the user adds the columns c1, c2 and c3, the values of which the user wants to see, and then the user creates the following query.

```
SELECT id, min(price), max(date), c1, c2, c3
FROM tableIn WHERE type = 'T' GROUP BY id
```

However, an SQL error occurs when the query is executed in the database because the columns c1, c2 and c3 are not included in the grouping key. To avoid such undesired situations, we assume that example tables given to our tool preserve the original schemas.

2.2 Problem Definition

The input of our algorithm is a tuple $(\mathcal{E}, C)$, where $\mathcal{E} = (\vec{T}_{\text{in}}, T_{\text{out}})$ is an example of input tables $\vec{T}_{\text{in}}$ and an output table T_{out}, and $C = \{v_1, \ldots, v_k\}$ is a set of typed constants. The schema of each column has a name and a type. The types consist of `Str`, `Int`, `Dbl` and `Date`. In this paper, we propose an algorithm that takes $(\mathcal{E}, C)$ as input, and returns a program p in our DSL that satisfies $p(\vec{T}_{\text{in}}) = T_{\text{out}}$, where the constants used in p are included in C. Here the equality operator (=) compares two tables by treating records as a list if T_{out} has at least one sorted column, and treating them as a multiset otherwise. That is, we synthesize programs with sort operators whenever possible. Of course, this policy possibly synthesizes extra sort operators, but it is important to support the sort operator since `ORDER BY` clause is often used in real-world SQL queries [43].

The limitation of asking the user to provide the constants is the same as that of SCYTHE [37]. The constants are used not only for improving the synthesis performance but for directly reflecting the user's intention. We believe this limitation is not an obstacle in practice because it is known that users asking SQL questions on Stack Overflow are usually ready to provide such constants even when they do not know how to write correct queries [37]. In addition to constants, SCYTHE requires hints about aggregation functions while PATSQL does not. One of the PATSQL's advantages is that it works with a more limited kind of user hints than in prior work [31, 37].

3 DOMAIN-SPECIFIC LANGUAGE

In this section, we describe our domain-specific language (DSL) that determines the search space of synthesis problems. The DSL is a kind of extended relational algebra with additional operators such as window functions. Before describing each operator in our DSL, we emphasize that DSLs for program synthesis need to be carefully designed with the trade-off between expressiveness and efficiency of synthesis [14, 40]. Following this policy, our DSL is designed to support as many SQL features as possible while allowing the synthesis algorithm to leverage properties of relational algebra.

Figure 2 shows the grammar, and the start symbol is $\langle table \rangle$. We refer to the elements other than `Table` in the right-hand side of the rule $\langle table \rangle$ as *operators*. Each operator is a function that takes one or more tables as input and returns a table.

The semantics of the operators is as follows. Here we use a vector such as $\vec{c}$ to represent multiple elements. `Project`$(T, \vec{c})$ extracts columns $\vec{c}$ from table T. `Select`$(T, pred)$ selects rows in T that satisfy a predicate *pred*. This predicate is in a conjunctive normal form, namely an AND of ORs. `Group`$(T, \vec{c}, \vec{gc})$ groups rows in T by keys $\vec{c}$ and returns a new table, whose columns consist of the keys $\vec{c}$ and aggregation results $\vec{gc}$. `Window`$(T, \vec{w})$ appends the resulting columns of window functions $\vec{w}$ to T. Each column w consists of a window function, a target column, partitioning keys and a sort key. `Join`$(T_1, T_2, pairs)$ executes inner join on T_1 and T_2 with key *pairs*. Likewise `LeftJoin`$(T_1, T_2, pair)$ executes left join with a key *pair*. `Distinct`(T) removes duplicated rows in T. `Order`$(T, \vec{key})$ sorts rows in T according to key columns and directions $\vec{key}$.

Our DSL supports a variety of SQL features used in practice. It supports `SELECT`, `WHERE`, `GROUP BY`, `JOIN`, `LEFT JOIN`, `DISTINCT`,

$$\begin{aligned}\langle s\rangle ::= &\ \texttt{Table}(\square) \mid \texttt{Order}(\langle s\rangle,\square) \mid \texttt{Distinct}(\langle s\rangle)\\ &\mid \texttt{Project}(\langle s\rangle,\square) \mid \texttt{Select}(\langle s\rangle,\square) \mid \texttt{Group}(\langle s\rangle,\square,\square)\\ &\mid \texttt{Window}(\langle s\rangle,\square) \mid \texttt{Join}(\langle s\rangle,\langle s\rangle,\square) \mid \texttt{LeftJoin}(\langle s\rangle,\langle s\rangle,\square)\end{aligned}$$

Figure 3: The grammar of the sketches in our DSL. The start symbol is $\langle s\rangle$.The symbol '□' denotes an uninstantiated part.

ORDER BY and HAVING. It also supports operators that can be expressed by other operators such as EXISTS (expressed by JOIN), NOT EXISTS (expressed by LEFT JOIN and IS NULL [9]), and RIGHT JOIN (expressed by LEFT JOIN). BETWEEN and IN in predicates can also be rewritten using OR. In summary, our DSL supports 17 out of 20 keywords that are most popular among SQL users according to the questionnaire survey in 2013 [43]. Besides, it supports window functions and nested queries since DSL operators can have other operators as their children. On the other hand, we do not support UNION operator. We discuss this limitation in Section 5.3.

4 PRELIMINARIES

In this section, we introduce key concepts that are used in our synthesis algorithm.

4.1 Sketch and its Completion

We refer to a program with uninstantiated parts as *sketch*, and an uninstantiated part is denoted as the symbol '□'. Specifically, the sketches in our algorithm are the languages that are produced by the grammar of Figure 3. In the syntax tree of a sketch, a leaf node corresponds to a Table while a non-leaf node corresponds to an operator. We call *sketch s' is a child of sketch s* when the node for s' is a child of the node for s in the corresponding syntax tree. Also, *sketch completion* is an action that fills '□'s with concrete structures. Our synthesis algorithm generates sketches and tries to complete each of them. Creating a sketch determines the operators used in the resulting program and enables us to validate its structure. Here we define the size of a sketch as follows. The size of the operators other than Window is one. The size of Window is two because the functionality is more complicated than the other operators, namely partitioning cells, applying window functions, and appending the results to the original table. We then define Size(s) function as the sum of each operator's size in a sketch s.

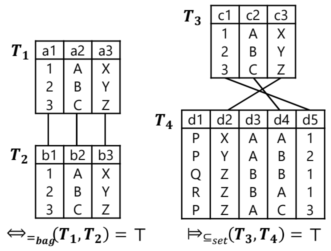
For instance, the illustrative example in Figure 1 employs the sketch s = Order(Project(Join(Group(Select(Table(□), □), □), Table(□),□), □), □), and Size(s) returns five. The results of sketch completion of s include the program in the center of Figure 1.

4.2 Table Inclusion Relation φ

In our synthesis algorithm, the knowledge of the output table is propagated in the form of our original constraint. The constraint denoted as $\varphi(T,T')$ is an inclusion relation between two tables T and T', and it returns true ($\top$) or false ($\bot$). Figure 4 shows the definition. The relation φ is in one of the three states: $\Leftrightarrow$, $\Mapsto$ and $\top$. The relation $\Leftrightarrow_R\ (T,T')$ means that the columns in T and T' correspond to each other in their order. $\Mapsto_R\ (T,T')$ means that there exist at least one column in T' that correspond to each column in T.

$$\begin{aligned}\varphi(T,T') &::= \ \Leftrightarrow_R (T,T') \mid \ \Mapsto_R (T,T') \mid \top\\ R &::= \ =_{\text{bag}} \mid \ \subseteq_{\text{bag}} \mid \ =_{\text{set}} \mid \ \subseteq_{\text{set}}\\ \Leftrightarrow_R (T,T') &\leftrightarrow |T| = |T'| \land \forall i \in [1,|T|].\ R(T[i],T'[i])\\ \Mapsto_R (T,T') &\leftrightarrow \forall i \in [1,|T|].\ \exists i' \in [1,|T'|].\ R(T[i],T'[i'])\end{aligned}$$

Figure 4: The definition of our table relation φ. T and T' refer to tables. $|T|$ is the number of columns in T, and $T[i]$ is the i-th column in T.



T_1
T_2
T_3
T_4
$\Leftrightarrow_{=_{bag}}(T_1,T_2) = \top$
$\Mapsto_{\subseteq_{set}}(T_3,T_4) = \top$

a1	a2	a3
1	A	X
2	B	Y
3	C	Z

b1	b2	b3
1	A	X
2	B	Y
3	C	Z

c1	c2	c3
1	A	X
2	B	Y
3	C	Z

d1	d2	d3	d4	d5
P	X	A	A	1
P	Y	A	B	2
Q	Z	B	B	1
R	Z	B	A	1
P	Z	A	C	3

Figure 5: Examples of our table relation φ. The edges between columns represent the column correspondence that is needed to hold a relation φ.

Specifically, we calculated the column mapping by enumerating the column pairs of the two tables. The parameter R in $\Leftrightarrow_R$ and $\Mapsto_R$ is a binary relation between columns, and holds a type of comparison: equality (=) or inclusion ($\subseteq$), and a treatment of cells: multiset (bag) or set (set).

We show examples of these relations in Figure 5. By using the tables T_1 and T_2, the predicate $\Leftrightarrow_{=_{\text{bag}}} (T_1,T_2) = \top$ holds since the column relations $=_{\text{bag}}(\text{a1},\text{b1})$, $=_{\text{bag}}(\text{a2},\text{b2})$ and $=_{\text{bag}}(\text{a3},\text{b3})$ are true, and the columns correspond to each other in their order. On the other hand, the relation $\Mapsto_{\subseteq_{\text{set}}} (T_3,T_4) = \top$ holds since the column relations $\subseteq_{\text{set}}(\text{c1},\text{d5})$, $\subseteq_{\text{set}}(\text{c2},\text{d4})$ and $\subseteq_{\text{set}}(\text{c3},\text{d2})$ are true, and there exists a column in T_4 that corresponds to each column in T_3.

The constraint φ can be represented as a tuple (M,R) except for the case of $\top$, where $M \in \{\Leftrightarrow, \Mapsto\}$ and $R \in \{=_{\text{bag}}, \subseteq_{\text{bag}}, =_{\text{set}}, \subseteq_{\text{set}}\}$. In the rest of this paper, we denote $\varphi(T,T')$ as $\varphi = (M,R)$ for short when the tables T and T' are obvious from the context. For example, we use notations such as $\varphi = (\Leftrightarrow, =_{\text{bag}})$ or $\varphi = (\Mapsto, \subseteq_{\text{set}})$.

5 SYNTHESIS ALGORITHM

In this section, we describe the details of our synthesis algorithm. Figure 6 shows the top-level algorithm. This algorithm takes as input an I/O example $\mathcal{E} = (\vec{T}_{\text{in}}, T_{\text{out}})$ and constants C, and returns a synthesized program. The algorithm steps are executed as follows. We first initialize a set of sketches S with a singleton having the sketch Table(□) or Order(Table(□), □), depending on whether T_{out} is sorted or not (lines 1-4), and iterate the following operations. In each iteration, we retrieve a sketch s with the minimum size of the sketches in S (lines 6-7). For each sketch s, we assign a table name to each '□' in Table(□) by calling the function AssignTables (line 8). Then, we complete all of the remaining '□'s in the sketch by calling CompleteSketch (line 9). When the completion succeeds

```
SYNTHESIZE($\mathcal{E}, C$)
Input $\mathcal{E} = (\vec{T}_{\text{in}}, T_{\text{out}})$: input and output tables
      $C$: constants used in predicates
Output a synthesized program
 1: if IsSorted($T_{\text{out}}$) then
 2:     $S \leftarrow$ {Table(□)}
 3: else
 4:     $S \leftarrow$ {Order(Table(□), □)}
 5: while true do
 6:     choose $s \in S$ s.t. $\forall t \in S$. Size($s$) $\leq$ Size($t$)
 7:     $S \leftarrow S \setminus \{s\}$
 8:     for $s' \in$ AssignTables($s, \vec{T}_{\text{in}}$) do
 9:         for $p \in$ CompleteSketch($s', T_{\text{out}}, C$) do
10:             if $p(\vec{T}_{\text{in}}) = T_{\text{out}}$ then
11:                 return $p$
12:     $S \leftarrow S \cup$ ExpandSketch($s$)
13: return $\perp$
```

Figure 6: The top-level synthesis algorithm

and a program p is found, we check whether the output table is equal to the evaluation result of p (line 10). If the check succeeds, we return the program p as a result (line 11). Otherwise, we generate additional sketches from the sketch s by calling ExpandSketch (line 12). The operations on lines 5-12 are iterated until a solution is found. Note that this algorithm never stops when there are no solutions in the search space. Therefore, a timeout should be set when it is provided to users in practice.

We show an example of how the top-level algorithm works by using the illustrative example in Figure 1. We start with a singleton with the sketch s_0 = `Order(Table(□), □)` since the output table `tableOut` is sorted by the `id` column. First, we try to complete the '□'s in the sketch s_0, but cannot find a program consistent with the I/O tables. Then, we expand the sketch s_0 by calling ExpandSketch function and obtain new sketches including s_1 =`Order(Select(Table(□), □)), □)`. We choose the sketch s_1 as the next candidate, but we fail to complete the sketch s_1 and then expand s_1 to obtain new sketches. After repeating the operations, we find the sketch s in Section 4.1 and successfully complete it. As a result, we can obtain the solution program consistent with the I/O tables, i.e. the program in the center of Figure 1.

Our algorithm returns a program that has the minimum size of the possible solutions since it tries to complete sketches in ascending order of the sketch size. This design is based on Occam's razor, i.e., the hypothesis that the simplest solution is most likely to be correct. This strategy is also helpful for users to understand synthesized queries. In contract, there is prior work that finds multiple candidates and returns top-k solutions based on criteria other than the size of programs [15, 37]. To make our PBE tool interactive, we return a simple program that satisfies the given specification as quick as possible, rather than returning more sophisticated programs by taking more synthesis time. In addition, the tolerable waiting time for computer response is known to be about two seconds [29, 33]. Thus, the synthesis time is preferable to be less than two seconds.

Table 1: The restriction of parent-child relations between operators in a sketch. Rows and columns represent parents and children, respectively. The combinations ✓ are allowed.

	`Order`	`Distinct`	`Project`	`Select`	`Group`	`Window`	`Join`	`LeftJoin`	`Table`
`Order`	X_3	✓	✓	✓	✓	✓	✓	✓	✓
`Distinct`	X_1	X_3	✓	✓	✓	✓	✓	✓	✓
`Project`	X_1	X_2	X_3	✓	✓	✓	✓	✓	✓
`Select`	X_1	X_2	X_3	X_3	✓	✓	✓	✓	✓
`Group`	X_1	X_2	X_3	✓	✓	✓	✓	✓	✓
`Window`	X_1	X_2	X_3	✓	✓	✓	✓	✓	✓
`Join`	X_1	X_2	X_3	X_3	✓	✓	✓	✓	✓
`LeftJoin`	X_1	X_2	X_3	✓	✓	✓	✓	✓	✓

5.1 Sketch Generation

We describe the details of ExpandSketch(s). This function takes a sketch s and returns sketches that are created by appending an operator to s. Here we refer to the elements other than `Table` in the right-hand side of $\langle s \rangle$ in Figure 3 as *sketch constructors*. This algorithm inserts each sketch constructor above each position of `Table` in the sketch s. First, we find a `Table(□)` in s and replace it with a sketch constructor. At this point, the entire sketch contains one or more $\langle s \rangle$ symbols. Then, we replace all of the $\langle s \rangle$s with `Table(□)`s. For example, when the input sketch is s = `Project(Table(□), □)`, the result of ExpandSketch(s) includes the following sketches.

- `Project(Select(Table(□), □), □)`
- `Project(Group(Table(□), □, □), □)`
- `Project(Join(Table(□), Table(□), □), □)`

Here we restrict the combinations of the operators that can appear in a sketch. Table 1 shows the parent-child relations that are allowed in a syntax tree. The combinations marked as '✓' are allowed whereas 'X' are not.

We exclude the combinations marked as X_1 in Table 1 because the operators except `Order` do not preserve the order of records, and therefore the order determined by `Order` is meaningful only when it is at the top of a sketch. We also exclude X_2 because it is considered to be rare in real queries. In fact, there exist 1,446 SQL queries having `DISTINCT` in Spider benchmark [42], which consists of practical SQL queries from various domains, and none of them violate the combination X_2. Additionally, we exclude X_3 because sketches including the combinations are not in *normal form*. That is, when a program p is obtained from a sketch that is not in normal form, we can always obtain a program equivalent to p from a sketch in normal from. These properties are based on transformation rules in relational algebra. For example, the following rules are known [8, 13].

- `Project(Project(`T, c_1`),` c_2`)` $\longrightarrow$ `Project(`T, c_2`)`
- `Select(Select(`T, p_1`),` p_2`)` $\longrightarrow$ `Select(`$T, p_1 \land p_2$`)`

These rules mean that a program that repeats the same operator can always be expressed as a program without repetition, and hence it is sufficient to generate only sketches without repetition in such cases. Besides, the following rules are known.

- `Select(Project(`T, c`),` p`)` $\longrightarrow$ `Project(Select(`T, p`),` c`)`

$$\textsc{Propagate}(\varphi, \texttt{Order}) = \varphi$$

$$\textsc{Propagate}(\varphi, \texttt{Distinct}) = \begin{cases} (M, =_{\text{set}}) & \text{if } \varphi = (M, =_{\text{bag}}) \\ (M, \subseteq_{\text{set}}) & \text{if } \varphi = (M, \subseteq_{\text{bag}}) \\ \varphi & \text{otherwise} \end{cases}$$

$$\textsc{Propagate}(\varphi, \texttt{Project}) = \begin{cases} (\Mapsto, R) & \text{if } \varphi = (\Leftrightarrow, R) \\ \varphi & \text{otherwise} \end{cases}$$

$$\textsc{Propagate}(\varphi, \texttt{Select}) = \begin{cases} (M, \subseteq_{\text{bag}}) & \text{if } \varphi = (M, =_{\text{bag}}) \\ (M, \subseteq_{\text{set}}) & \text{if } \varphi = (M, =_{\text{set}}) \\ \varphi & \text{otherwise} \end{cases}$$

$$\textsc{Propagate}(\_, \texttt{Group}) = \top \qquad \textsc{Propagate}(\_, \texttt{Window}) = \top$$
$$\textsc{Propagate}(\_, \texttt{Join}) = \top \qquad \textsc{Propagate}(\_, \texttt{LeftJoin}) = \top$$

Figure 7: Definition of the propagation function

- `Join(Project(`T_1, c`),` T_2, p`)` $\longrightarrow$ `Project(Join(`T_1, T_2, p`),` c'`)`

These are rules for moving `Project` above `Select` and `Join`. The same rule is also applicable for `Group`, `Window` and `LeftJoin`. As a consequence, the combinations in Table 1 allow only sketches with at most one `Project` and with `Project` above the operators other than `Order` and `Distinct`. Also, the sketch constructors `Order`, `Distinct` and `Project` can appear at the top of the sketch in this order. Importantly, this restriction leads to the efficient completion of the sketch `Project(`s'`, □)`. Note that the restrictions X_1 and X_3 do not decrease the expressiveness of synthesized queries.

Finding an appropriate program structure from the given I/O specification is essentially difficult in most domains [16], and our domain of SQL is not an exception. Concretely, the worst-case complexity of the sketch generation algorithm is exponential to the size of the sketch. The reason is that, when creating sketches with the size of n, we need to insert each sketch constructor into the sketches with the size of $n-2$ (for `Window` constructor) or $n-1$ (for the other constructors). The mitigation of the computational cost will help us to find complex program structures.

5.2 Sketch Completion

First, we describe the function $\textsc{AssignTables}(s, \vec{T}_{\text{in}})$ used in the top-level algorithm in Figure 6. This function takes as input a sketch s and the input tables $\vec{T}_{\text{in}}$, and returns sketches with the tables' name filled. Specifically, it assigns a table name in $\vec{T}_{\text{in}}$ to each `Table(□)` in the sketch s. Here all of the input tables must be used to complete the sketch. This limitation is based on the assumption that the user does not give input tables that are not used in a resulting query, and it is effective for excluding sketches that do not meet the user's intention. For example, when a sketch s = `LeftJoin(Table(□), Table(□), □)` and $\vec{T}_{\text{in}} = \{T_1, T_2\}$, the result of $\textsc{AssignTables}(s, \vec{T}_{\text{in}})$ includes the following sketches.

- `LeftJoin(Table(`T_1`), Table(`T_2`), □)`
- `LeftJoin(Table(`T_2`), Table(`T_1`), □)`

Note that the sketch `LeftJoin(Table(`T_1`), Table(`T_1`), □)` is not included since this does not use the input table T_2.

Next, we explain the function $\textsc{CompleteSketch}(s, T_{\text{out}}, C)$. This function takes as input a sketch s, the output table T_{out} and constants C, and returns a set of programs. It fills '□'s in the sketch s by using constants in C, and thus constructs programs whose evaluation result can be T_{out}. Figure 8 shows the algorithms for sketch completion. The function $\textsc{CompleteSketch}(s, T_{\text{out}}, C)$ is the entry point, and it invokes auxiliary functions named $\textsc{Complete}$ for recursively completing a sketch by propagating a constraint φ. The completion algorithms are similar to those used in Morpheus [12] with two main differences. First, we prune the search space based on table inclusion relation φ while Morpheus relies on the metadata of tables and employs an SMT solver. Second, the completion algorithm of projection sketches is novel and efficient.

Before describing the details of each function in Figure 8, we start with the notations that are commonly used in these functions. We use $[\![p]\!]$ to refer to the table that can be gained by evaluating a program p. The function $\textsc{Cols}(T)$ returns the columns in table T. $\textsc{Propagate}(\varphi, op)$ takes a constraint φ and an operator type op, and returns a constraint φ'. Figure 7 shows the definition of $\textsc{Propagate}$. It calculates a precondition φ' from the postcondition φ of the sketch completion. Accurately, suppose we have a sketch s and its child s', and the programs p and p' are obtained by completing s and s', respectively. When the sketch s has the operator op, $\textsc{Propagate}(\varphi, op)$ returns a constraint φ' such that $\varphi'(T_{\text{out}}, [\![p']\!]) = \top$, which is a necessary condition for $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ to hold. For example, suppose a sketch s = `Select(`s'`, □)` is to be completed, and the postcondition is $\varphi = (\Mapsto, =_{\text{set}})$. This means that $[\![p]\!]$ needs to have a column equivalent to each column in T_{out} as set ($=_{\text{set}}$). Then, $\textsc{Propagate}(\varphi, \texttt{Select})$ returns $\varphi' = (\Mapsto, \subseteq_{\text{set}})$ as a precondition for the completion of s. The reason is that `Select` operator does not yield any new values, and therefore $[\![p']\!]$ needs to have a column that is superset ($\subseteq_{\text{set}}$) of each column in T_{out}.

We explain each part of the algorithm in Figure 8. For the completion of `Table(name)`, there are no '□'s to be filled, and only pruning is performed using the propagated constraint φ. Here the evaluation result $[\![p]\!]$ is the same as one of the input tables. Therefore, the pruning essentially compares an input table T_{in} and the output table T_{out}. This is the first pruning that we perform before filling '□'s in a sketch, and here we validate whether the entire sketch can satisfy the I/O tables. For example, if T_{out} has a cell with a value v that does not exist in T_{in}, we can discard the sketch `Project(Select(Table(`T_1`), □), □)` before filling the remaining '□'s. The reason is that the propagated constraint $\varphi = (\Mapsto, \subseteq_{\text{bag}})$ can detect the fact that no matter how the remaining '□'s are filled, the value v will not be yielded by `Select` or `Project`. This pruning is similar to the validation of a sketch in Morpheus [12]. While it uses an SMT solver to validate a sketch, our algorithm uses a propagated constraint φ, which considers an inclusion relation between the output and intermediate tables.

For the completion of `Order(`s'`, □)`, we first complete the child s' and then fill the target sketch with sort keys. The function $\textsc{SortKeys}$ infers sort keys from the columns in T_{out}. Accurately, we find a column c_1 sorted in ascending or descending order from the columns in T_{out}. If the values in c_1 are duplicated, we group the records by the key of c_1. Then, we again find sort keys for each group. If c_2 is a valid key for all the groups, we obtain a composite key $[c_1, c_2]$. By

```
$\underline{\textsc{CompleteSketch}}(s, T_{\text{out}}, C)$
Input $s$: a sketch, $T_{\text{out}}$, $C$: constants
Output a set of programs
  $\varphi_0 \leftarrow (\Leftrightarrow, =_{\text{bag}})$
  return $\textsc{Complete}(s, T_{\text{out}}, C, \varphi_0)$

$\underline{\textsc{Complete}}(\texttt{Table(name)}, T_{\text{out}}, C, \varphi)$
  $p \leftarrow \texttt{Table(name)}$
  if $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ then
      return $\{p\}$
  else
      return $\emptyset$

$\underline{\textsc{Complete}}(\texttt{Order}(s', \square), T_{\text{out}}, C, \varphi)$
  // always $\varphi = (\Leftrightarrow, =_{\text{bag}})$
  $P \leftarrow \emptyset$
  for $p' \in \textsc{Complete}(s', T_{\text{out}}, C, \varphi)$ do
      $keys \leftarrow \textsc{SortKeys}(T_{\text{out}}, [\![p']\!])$
      $p \leftarrow \texttt{Order}(p', keys)$
      // always $\varphi(T_{\text{out}}, [\![p]\!]) = \top$
      $P \leftarrow P \cup \{p\}$
  return $P$

$\underline{\textsc{Complete}}(\texttt{Distinct}(s'), T_{\text{out}}, C, \varphi)$
  $P \leftarrow \emptyset$
  $\varphi' \leftarrow \textsc{Propagate}(\varphi, \texttt{Distinct})$
  for $p' \in \textsc{Complete}(s', T_{\text{out}}, C, \varphi')$ do
      $p \leftarrow \texttt{Distinct}(p')$
      if $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ then
          $P \leftarrow P \cup \{p\}$
  return $P$
```

```
$\underline{\textsc{Complete}}(\texttt{Project}(s', \square), T_{\text{out}}, C, \varphi)$
  // always $\varphi = (\Leftrightarrow, R)$
  $P \leftarrow \emptyset$
  $\varphi' \leftarrow (\Rrightarrow, R)$ // $= \textsc{Propagate}(\varphi, \texttt{Project})$
  for $p' \in \textsc{Complete}(s', T_{\text{out}}, C, \varphi')$ do
      for $i \in 1 \ldots |T_{\text{out}}|$ do
          $R_i \leftarrow \{i' \mid R([\![p']\!][i'], T_{\text{out}}[i])\}$
      for $cols \in R_1 \times \cdots \times R_{|T_{\text{out}}|}$ do
          $p \leftarrow \texttt{Project}(p', cols)$
          if $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ then
              $P \leftarrow P \cup \{p\}$
  return $P$

$\underline{\textsc{Complete}}(\texttt{Select}(s', \square), T_{\text{out}}, C, \varphi)$
  $P \leftarrow \emptyset$
  $\varphi' \leftarrow \textsc{Propagate}(\varphi, \texttt{Select})$
  for $p' \in \textsc{Complete}(s', T_{\text{out}}, C, \varphi')$ do
      for $cond \in \textsc{Conds}([\![p']\!], C)$ do
          $p \leftarrow \texttt{Select}(p', cond)$
          if $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ then
              $P \leftarrow P \cup \{p\}$
  return $P$

$\underline{\textsc{Complete}}(\texttt{Window}(s', \square), T_{\text{out}}, C, \varphi)$
  $P \leftarrow \emptyset$
  for $p' \in \textsc{Complete}(s', T_{\text{out}}, C, \top)$ do
      $wins \leftarrow \textsc{Wins}([\![p']\!])$
      $p \leftarrow \texttt{Window}(p', wins)$
      if $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ then
          $P \leftarrow P \cup \{p\}$
  return $P$
```

```
$\underline{\textsc{Complete}}(\texttt{Group}(s', \square, \square), T_{\text{out}}, C, \varphi)$
  $P \leftarrow \emptyset$
  for $p' \in \textsc{Complete}(s', T_{\text{out}}, C, \top)$ do
      $aggs \leftarrow \textsc{Aggs}([\![p']\!])$
      for $cols$ s.t. $cols \subseteq \textsc{Cols}([\![p']\!]) \wedge |cols| \leq 2$ do
          $p \leftarrow \texttt{Group}(p', cols, aggs)$
          if $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ then
              $P \leftarrow P \cup \{p\}$
  return $P$

$\underline{\textsc{Complete}}(\texttt{Join}(s'_1, s'_2, \square), T_{\text{out}}, C, \varphi)$
  $P \leftarrow \emptyset$
  $P'_1 \leftarrow \textsc{Complete}(s'_1, T_{\text{out}}, C, \top)$
  $P'_2 \leftarrow \textsc{Complete}(s'_2, T_{\text{out}}, C, \top)$
  for $(p'_1, p'_2) \in P'_1 \times P'_2$ do
      for $pairs \in \textsc{Pairs}([\![p'_1]\!], [\![p'_2]\!])$ do
          $p \leftarrow \texttt{Join}(p'_1, p'_2, pairs)$
          if $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ then
              $P \leftarrow P \cup \{p\}$
  return $P$

$\underline{\textsc{Complete}}(\texttt{LeftJoin}(s'_1, s'_2, \square), T_{\text{out}}, C, \varphi)$
  $P \leftarrow \emptyset$
  $P'_1 \leftarrow \textsc{Complete}(s'_1, T_{\text{out}}, C, \top)$
  $P'_2 \leftarrow \textsc{Complete}(s'_2, T_{\text{out}}, C, \top)$
  for $(p'_1, p'_2) \in P'_1 \times P'_2$ do
      for $pair \in \textsc{Pair}([\![p'_1]\!], [\![p'_2]\!])$ do
          $p \leftarrow \texttt{LeftJoin}(p'_1, p'_2, pair)$
          if $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ then
              $P \leftarrow P \cup \{p\}$
  return $P$
```

Figure 8: Algorithms for sketch completion

finding sort keys in the same manner, we obtain a composite key $[c_1, c_2, \ldots, c_n]$. Finally, we convert it to the corresponding columns in $[\![p']\!]$. Note that pruning with the constraint φ is not needed here because the child's constraint $\varphi(T_{\text{out}}, [\![p']\!]) = \top$ holds and hence the predicate $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ is always true.

For the completion of `Distinct`(s'), we complete the child sketch and discard invalid programs since there are no '$\square$'s to be filled,.

For the completion of `Project`$(s', \square)$, we fill the sketch with projected columns. This is one of the most distinctive parts of our algorithm. Importantly, owing to the restriction of sketch structures in Table 1, the constraint φ always has the form of $\varphi = (\Leftrightarrow, R)$ when this function is invoked. First, we obtain the program p' by completing the child of the sketch. Then, for each i-th column in T_{out}, we calculate the set R_i of indices i' such that the i'-th column in $[\![p']\!]$ corresponds to the i-th column in T_{out} by relation R. Note that every R_i cannot be empty because the child's constraint $\varphi' = (\Rrightarrow, R) \wedge \varphi'(T_{\text{out}}, [\![p']\!]) = \top$ holds. Finally, we enumerate the elements in the cartesian product of $R_1, \ldots, R_{|T_{\text{out}}|}$ as the candidates for projected columns. The bottlenecks of this part are (1) the calculation of each R_i and (2) the calculation of the cartesian product of $R_1, \ldots, R_{|T_{\text{out}}|}$. The computational complexity of (1) is $O(\|T_{\text{out}}\|\|[\![p']\!]\|)$, where $\|T\|$ is the number of the cells in table T. The worst-case complexity of (2) is exponential, but the computation does not cause a combinatorial explosion in most cases because the size of each R_i is generally small.

In contrast, prior work [12, 37] does not expect that the columns in $[\![p']\!]$ correspond to those in T_{out} in their order (i.e., '$\Leftrightarrow$' in our work) when `Project` sketch is being completed. Therefore, it needs to enumerate all the permutations of the columns in $[\![p']\!]$ as in Figure 9. The computational cost grows exponentially as the number of the columns in $[\![p']\!]$ increases, which leads to a scalability issue in the overall algorithm.

For the completion of `Select`$(s', \square)$, we fill the sketch with predicates. The function $\textsc{Conds}(T, C)$ returns the predicates using columns in table T and constants in C. These predicates conform to the rule $\langle pred \rangle$ in the grammar of Figure 2, and they have consistent types between columns and constants. To enumerate the predicates efficiently, the information about remaining rows is encoded into a *bit array*, which is introduced in SCYTHE [37]. That is, we calculate a bit array b for each predicate, where $b[i] = 1$ if the i-th row in T remains, and $b[i] = 0$ otherwise. Specifically, we enumerate predicate candidates in the order of $\langle prim \rangle$, $\langle clause \rangle$ and $\langle pred \rangle$. A predicate for $\langle clause \rangle$ is a disjunction of $\langle prim \rangle$s, and a $\langle pred \rangle$ is a conjunction of $\langle clause \rangle$s. When calculating the remaining rows as a result of these predicates, we perform the operations $\vee$ and $\wedge$ over bit arrays instead of executing compound predicates over an instantiated table. Since we try to find a single program that satisfies the I/O tables, we discard predicates that have the same bit array as previously enumerated ones.

For the completion of `Window`$(s', \square)$, we fill the sketch with the columns of window functions. The function $\textsc{Win}(T)$ enumerates

```
COMPLETE(Project($s'$, □), $T_{\text{out}}$, $C$, $\varphi$)
1: $P \leftarrow \emptyset$
2: for $p' \in$ COMPLETE($s'$, $T_{\text{out}}$, $C$, $\varphi$) do
3:     for $cols \subseteq$ COLS($[\![p']\!]$) do
4:         for $cols' \in$ PERMUTE($cols$) do
5:             $p \leftarrow$ Project($p'$, $cols'$)
6:             if $\varphi(T_{\text{out}}, [\![p]\!]) = \top$ then
7:                 $P \leftarrow P \cup \{p\}$
8: return $P$
```

Figure 9: The completion of Project sketch in prior work

the columns in the form of $\langle win \rangle$ in the grammar of Figure 2, i.e., a window function, a target column, partitioning keys and a sort key. The number of these combinations may be large in some cases, and the table obtained by $[\![p]\!]$ possibly has a large number of columns. However, this does not significantly affect the overall performance since our algorithm is resistant to the increase in column size. This strategy enables our algorithm to do without user hints as to which functions should be used. Of course, due to this strategy, synthesized programs may contain useless columns that do not affect the behavior. Hence, we eliminate such columns after we have found a correct program.

For the completion of `Group`(s', □, □), we fill the sketch with grouping keys and aggregation columns. Similar to the completion of `Window` sketch, the function AGGS($[\![p']\!]$) enumerates the combinations of the columns in $[\![p']\!]$ and the aggregation functions. We then enumerate grouping keys, the size of which is less than or equal to two. We discuss this limitation in Section 5.3.

For the completion of `Join`(s'_1, s'_2, □) and `LeftJoin`(s'_1, s'_2, □), we fill the sketch with key pair(s). These sketches have two children s'_1 and s'_2, and we begin with completing them to obtain the sets P'_1 and P'_2 of programs. Then, we fill join predicates for each pair of the programs $p'_1 \in P'_1$ and $p'_2 \in P'_2$. The function PAIRS(T_1, T_2) uses columns in T_1 and T_2 to create the predicates in the form of $\langle pairs \rangle$ in Figure 2. In particular, it first enumerates the predicates for $\langle pair \rangle$, and then combines them using $\wedge$ to generate the conjunctive predicates for $\langle pairs \rangle$. Similarly, PAIR(T_1, T_2) returns a set of single pairs in T_1 and T_2 for the completion of `LeftJoin` sketches. In these processes, we use bit arrays to efficiently enumerate the predicates as in the completion of `Select` sketches. In addition to calculating remaining rows efficiently, we reduce the cost for instantiating the cross product of two tables, as in SCYTHE.

5.3 Adaptability to Grammar Extensions

Our algorithm can easily support additional syntax features for predicates and functions because it does not depend on the concrete semantics of them. For example, we can easily support the aggregation function `STDEV`, which calculates the statistical standard deviation for each group, by adding it to the list of aggregation functions. Also, supporting `LIKE` in predicates is straightforward if the user provides pattern strings used in `LIKE` predicates as part of constants C.

In contrast, it is difficult for our algorithm to naively support the `UNION` clause, which is an operator to combine the records in two tables. Because the positions of `Union` and `Project` cannot be safely interchanged in sketch structures, we can no longer fix the position of `Project` above other operators as the combinations in Table 1. Hence, filling `Project`(s', □) with projected columns leads to a significant increase in computational cost as in prior work. However, in general, the `UNION` clause is used just for combining the results from multiple subqueries. We believe that the user does not have difficulty combining such queries by using the `UNION` keyword as long as our method synthesizes each query. It is also difficult to naively support queries with a large number of grouping keys. The reason is that a slight change in key selection can have a significant effect on yielded values, and we cannot find desired grouping keys without executing queries. Thus, allowing an arbitrary number of grouping keys results in a combinatorial explosion during the completion of `Group` and `Window` sketches. This is the reason for which we limit the size of grouping keys to two or less in Figure 2.

6 IMPLEMENTATION

We implemented this algorithm in Java as a tool PATSQL and optimized it in several points to reduce the search space on sketch structures. First, we do not enumerate the sketches that are "symmetric" with the previously explored ones. Accurately, we do not distinguish the sketches in the form of `Join`(s_1, s_2, □) and `Join`(s_2, s_1, □) since the programs derived from them are semantically equivalent. Second, we do not enumerate the sketches that do not contain `Select` operator when the constants C are not empty.

We provide a user interface that is inspired by the concept of *live programming*. Live programming is an interactive programming environment, where the user can get real-time feedback on the behavior of a program each time s/he updates the code [23]. Similarly, we implemented an interactive user interface for PBE, where the user can get real-time feedback on the synthesized program each time s/he updates the I/O example. This interface enables the user to start with a simple example and progressively create more complex examples, as long as the synthesis time is reasonably short.

7 EVALUATION

To evaluate PATSQL, we perform experiments to answer the following research questions.

RQ1: Is PATSQL more effective for synthesizing complex SQL queries than prior methods?
RQ2: Do our algorithm improvements perform better than the naive algorithm?
RQ3: What kind of queries does PATSQL fail to synthesize?
RQ4: Can PATSQL handle large I/O tables better than prior methods?

To answer the questions, we synthesize a wide variety of SQL queries by using PATSQL and other methods. The experiments are conducted on a machine with 2.20 GHz Intel Xeon CPU and 8 GB of physical memory running the Windows 10 OS.

7.1 Benchmarks

To investigate the effectiveness of PATSQL on practical SQL queries, we collected 226 queries in total from the following benchmarks.
ase13: These 28 queries were extracted from a textbook for database systems. The I/O tables were introduced for the evaluation of SQLSynthesizer [43] and also used for SCYTHE study [37]. These

Table 2: The number of benchmarks solved by different algorithms. The "No." column represents the number of queries. The "#Col" and "#Cell" columns represent the average number of columns and cells, respectively.

Benchmark	No.	#Col	#Cell	PatSQL	PatSQL$_5$	Scythe	BaseLine
`ase13`	28	3.2	50.5	**25 (89%)**	21 (75%)	15 (54%)	17 (60%)
`so-top`	57	3.0	17.2	**42 (74%)**	**42 (74%)**	40 (70%)	30 (53%)
`so-dev`	57	4.9	26.1	**47 (82%)**	46 (81%)	46 (81%)	38 (67%)
`so-rec`	51	5.5	33.7	20 (39%)	17 (33%)	**27 (53%)**	13 (25%)
`kaggle`	33	12.1	164.5	**19 (58%)**	18 (55%)	0 (0%)	1 (3%)
total	226	-	-	**153 (68%)**	144 (63%)	128 (57%)	99 (44%)

queries range from basic ones that non-experts may have difficulty writing, to complex ones that combine standard SQL features.

`so-top`, `so-dev`, `so-rec`: The three benchmarks have 57, 57 and 51 queries, respectively. They were extracted from Stack Overflow. The I/O tables were introduced in the evaluation of Scythe. The posts in these benchmarks are questions about SQL programming. The benchmark `so-top` consists of posts with more than 30 votes, and it represents common and practical issues that developers are faced with. The benchmarks `so-dev` and `so-rec` are questions that were posted during the development of Scythe. These posts have considerably fewer votes than `so-top`, and the intention of the posts tends to be ambiguous. Also, some posts were modified after Scythe study. Since Scythe study does not publish queries marked as solutions, we determined the solutions based on the latest posts and published them in our Github repository. Note that the median first response time for the posts took 12 minutes, and the median acceptance time took 81 minutes.

`kaggle`: These 33 queries were extracted from SQL tutorials in Kaggle, which is an online community for data science. In particular, we collected queries from tutorials and exercises in "Intro to SQL" [1] and "Advanced SQL" [2]. These include various queries that are common in the context of data analysis. Importantly, the queries handle real tables that are published as datasets in Kaggle. This benchmark is mainly used for evaluating the effectiveness of the various PBE methods on I/O tables having full-scale schemas.

7.2 Compared Systems

We compare PatSQL with the following methods.

Scythe is a state-of-the-art tool that synthesizes complex SQL queries form I/O tables [37]. In particular, it supports the synthesis of nested query along with grouping and aggregation by enumerating *abstract queries* in a bottom-up manner. There are several differences in specifications between PatSQL and Scythe. For instance, while PatSQL finds a single solution as a result, Scythe finds five solutions that have high scores based on a ranking heuristic. Also, Scythe requires hints about which aggregation functions should be used in the resulting query, in addition to constants.

PatSQL$_5$ is a top-5 implementation of PatSQL that returns five solutions based on a ranking heuristic. This algorithm is used for a fair comparison with Scythe since it also returns five solutions as a result. The ranking heuristic is similar to Scythe. Namely, it considers the simplicity of synthesized queries and the coverage of given constants, and it finds five best solutions among programs of the same size. This algorithm is also used for investigating the effectiveness of PatSQL's strategy that finds a single solution.

BaseLine is a baseline algorithm that has the same search space as PatSQL. Specifically, BaseLine synthesizes SQL queries in our DSL in Figure 2, and the overall algorithm is almost the same as PatSQL. The difference is, while completing the sketch $\texttt{Project}(s', \square)$, it uses a brute-force search algorithm (Figure 9), which is highly inefficient way for finding the projected columns, as opposed to our algorithm (Figure 8). Also, BaseLine requires hints about which aggregation and window functions should be used. Without the hints, the computationan becomes intractable since the completion of `Project` sketches needs to calculate the permutations of a large number of columns, which we have appended in the completion of `Group` and `Window` sketches.

7.3 Evaluation Process

We performed the synthesis of the benchmark queries by using PatSQL, Scythe, PatSQL$_5$ and BaseLine. For the benchmarks other than `kaggle`, we reused the I/O tables that were created in Scythe study [37] for a fair comparison between these methods. It is important to use similar tables for each method since the performance can depend on the size of tables. The columns related to the query are extracted in advance, and hence the schemas of the I/O tables are not necessarily the same as original ones. For the `kaggle` benchmark, we created I/O tables for each query. Specifically, the tables were created by a third person, who is familiar with SQL but was not involved in the development of our synthesis algorithm. Here the schemas of I/O tables are the same as those of the original tables in datasets. Thus, the tables in the `kaggle` benchmark have an average of 12.1 columns, which is larger than the average of 4.3 columns in the other four benchmarks. In general, using original schemas in I/O examples is preferable for users since there is no need to decide which columns to be used while creating tables. We provided each algorithm with hints about the constants used in predicates, and we also provided Scythe and BaseLine with hints about aggregation and window functions. Scythe required a total of 110 hints about aggregate functions, some of which are used in the same query.

The correctness of the synthesized queries was verified by at least two people familiar with SQL. When the synthesized query did not have the same semantics as the solution, we updated the I/O tables to make the intention clearer mainly by adding rows. We continued to update the tables and hints until a solution was found or a timeout occurred. When a solution was found, we reported the synthesis time of the last execution. Since the synthesis time

[1]https://www.kaggle.com/learn/intro-to-sql
[2]https://www.kaggle.com/learn/advanced-sql

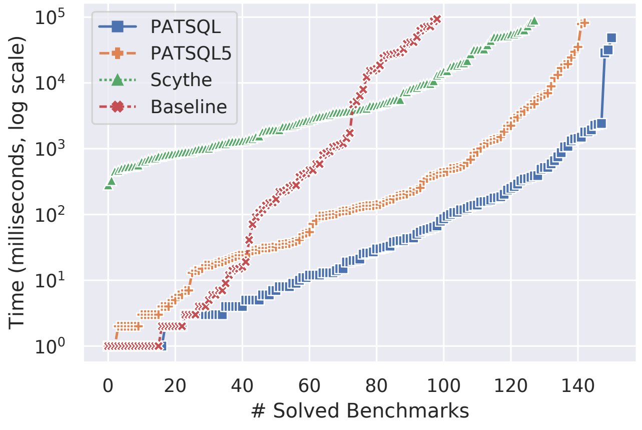


Figure 10: The synthesis time for benchmarks.

increases monotonically during the iterations, the time required for the last execution is a good indicator for the total time for multiple iterations. For the evaluation of Scythe and PatSQL$_5$, we checked if a desired query was included in the five queries returned as a result. We did not check the correctness of the Scythe's results for the benchmarks ase13 and so-top because the correctness had been carefully verified in Scythe study. Since we focus on an interactive usage of a PBE tool as mentioned in Section 6, we did not count the number of the updates performed during the synthesis process. For the same reason, we performed each synthesis with a time limit of 100 seconds, which is shorter than that in Scythe study.

7.4 Comparison to Prior Work

We first compare the number of benchmarks solved PatSQL and Scythe. For the benchmarks other than kaggle, PatSQL solved 134 benchmarks while Scythe solved 128 (see Table 2). In particular, PatSQL succeeded in synthesizing 20 queries from I/O tables with a larger number of cells that Scythe failed to solve due to scalability issues. Additionally, PatSQL succeeded in synthesizing a query with the window function RANK(), which Scythe does not support. In contrast, PatSQL failed to synthesize 15 queries that Scythe succeeded in. Six cases were due to non equi-join, which uses inequality operators such as '<' in a JOIN condition, and four cases were due to the UNION clause. The other causes include unsupported predicates and the order in which sketches were generated. For the kaggle benchmark, PatSQL solved 19 benchmarks while Scythe did not solve any of them. Since the benchmark handles I/O tables having full-scale schema, Scythe caused combinatorial explosions in the enumeration of abstract queries. In contrast, PatSQL succeeded in synthesizing such queries due to improvements in our algorithm to support I/O tables having larger schemas. In summary, the expressiveness of PatSQL is comparable to that of Scythe, and PatSQL can synthesize queries from I/O tables having larger schemas that Scythe fails to deal with. Because of its highly scalable nature, PatSQL was able to synthesize more queries within a time limit. In addition, we count the number of the rows in I/O tables for PatSQL and Scythe to find a solution. We found that the rows required for PatSQL is on average 0.56 more than those required for Scythe. This means PatSQL does not impose a large overhead

Table 3: The causes of PatSQL's failure. The 'f-elem' row means unsupported syntax features, and 'f-struct' means complex structures of desired queries that can be represented in DSL but cannot be found within a time limit.

	ase13	so-top	so-dev	so-rec	kaggle	total
f-elem	2	14	5	29	10	60
f-struct	1	1	5	2	4	13
total	3	15	10	31	14	73

on the user. When we compare PatSQL$_5$ to Scythe, both of which return top five solutions from the same size programs, PatSQL$_5$ still outperforms Scythe in the number of solved benchmarks (Table 2).

Next, we compare PatSQL to Scythe in terms of the execution time. PatSQL synthesized 102 of solved benchmarks within 0.1 seconds and 136 within a second, while Scythe synthesized 28 within a second and 100 within ten seconds (Figure 10). The faster execution time of PatSQL is useful for interactive scenarios where the user modifies the I/O example step by step. Also, 96% of the response time taken by PatSQL was shorter than two seconds, i.e. the tolerance waiting time for web response (Section 5), and then much shorter than 12 minutes, i.e. the median time for the first response in Stack Overflow (Section 7.1). Thus, the contributions of PatSQL are shown to be valuable for the practical application of PBE methods. When we compare PatSQL$_5$ to Scythe in terms of the execution time, PatSQL$_5$ still outperforms Scythe (Figure 10). This shows that PatSQL's efficiency is significant, even taking into account the difference in the number of solutions returned.

We compare the performance of PatSQL and PatSQL$_5$. The solved benchmarks by PatSQL$_5$ was slightly fewer than that of PatSQL (Table 2). The execution time of PatSQL$_5$ was about 10 times slower than PatSQL (Figure 10). This result is as expected since PatSQL$_5$ has a larger search space than PatSQL to find multiple candidate programs. We found that 90% of the I/O examples required for PatSQL$_5$ were the same as those required for PatSQL. In other words, an interface that returns a single solution does not impose significant additional overhead on the user. Thus, we can say that the search priorities based on program simplicity in PatSQL is effective as well as the ranking function in PatSQL$_5$. In general, a top-k algorithm only works well when it can find the desired solution as one of the candidates and rank it within top k among the candidates. Since there can be an exponential number of candidates in the search space, top-k algorithms tend to require additional examples to resolve ambiguities, as in the algorithm that finds a single solution. Hence, the search strategy that finds a single solution as quickly as possible would be suitable in many practical applications.

For **RQ1**, the experimental result suggests that PatSQL outperforms a state-of-the-art algorithm Scythe in terms of the execution time and the scalability of I/O tables while maintaining the expressiveness of synthesized queries. We also show the effectiveness of PatSQL's strategy that finds a single solution.

7.5 Comparison to Baseline

To illustrate the effectiveness of our improvements in the algorithm, we compare the number of benchmarks solved by PatSQL

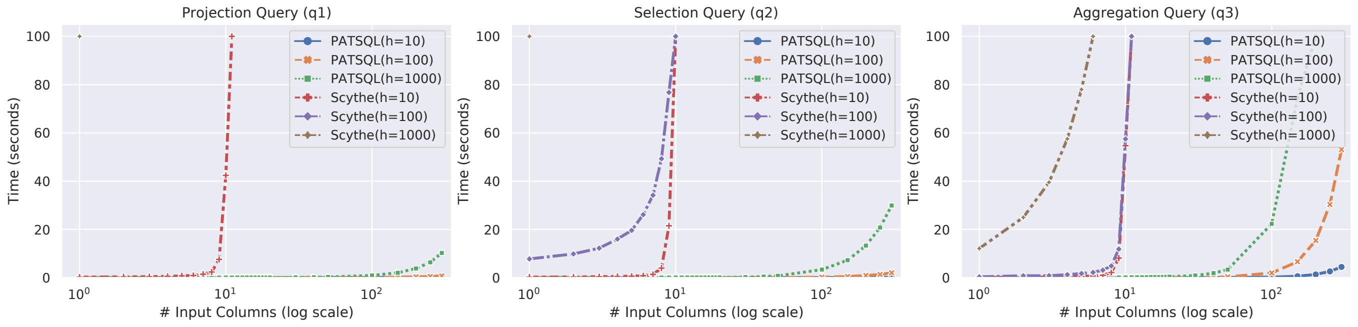


Figure 11: The synthesis time in a controlled scalability experiment. "h"s in legends represent the number of input rows. For instance, "(h=10)" means that the number of input rows is 10.

and BaseLine. As a result, BaseLine solved 99 out of the 226 benchmarks while PatSQL solved 153 out of them (Table 2). Recall that the difference between PatSQL and BaseLine is in the completion of `Project` sketches. Therefore, the failed benchmarks by BaseLine were caused by inefficient performance in the completion of `Project` sketches. That is, the completion algorithm seen in prior work significantly decreases the performance in cases with large tables, and our improvements can work efficiently in such cases. This difference leads to a 24% $(= (153 - 99)/226)$ improvement in the number of solved benchmarks. Next, we compare the execution time. As a result, BaseLine solved 67 benchmarks within a second while PatSQL solved 136 within a second (Figure 10). The result shows that our improvements significantly reduce the execution time. Note that other improvements such as pruning by the table constraint φ and the restriction on sketch structures in Table 1 are not crucial for the performance improvement, but they are required to realize the efficient completion of `Project` sketches.

For **RQ2**, the experimental result shows that our improvements in the algorithm improve the overall performance significantly. We also show that the efficient completion of `Project` sketches is crucial for the overall performance.

7.6 Failed Cases

To answer **RQ3**, we analyzed the 73 cases that PatSQL failed to solve. We classified the failure causes into two categories: *f-elem* and *f-struct*, and we found that 60 cases are in f-elem and 13 cases in f-struct (Table 3). The 'f-elem' row represents the number of queries that PatSQL failed to synthesize as the syntax features are not supported as of now. These include pivot operations (7 cases), `UNION` (6 cases), timestamp operations (5 cases), non equi-join (4 cases), `LIKE` (3 cases), `CASE` (3 case) and other 15 SQL features. In contrast, the 'f-struct' row represents the number of failed cases due to the complex structures of desired queries. In particular, two cases in `kaggle` benchmark require sketches whose sizes are five, while the other cases require more than five. PatSQL cannot find relatively small sketches for `kaggle` benchmark because it needs to process larger I/O tables during the synthesis. Although the computational cost of PatSQL does not increase significantly with respect to the scale of I/O tables, it certainly does increase in a polynomial order. Therefore, PatSQL could not search a sufficient number of sketches before a timeout occurred.

7.7 Controlled Scalability Experiment

To answer **RQ4**, we performed a controlled experiment that examines the difference between PatSQL and Scythe in synthesis time when the size of I/O tables increases. We used the following SQL queries as synthesis solutions in this experiment.

(q1) `SELECT * FROM table;`
(q2) `SELECT * FROM table WHERE c1 = 'T';`
(q3) `SELECT COUNT(*) FROM table;`

The queries q1, q2 and q3 represent the simplest queries for projection, selection and aggregation, respectively. We then crafted different I/O tables consisting of unique values to synthesize each of the queries. We employed different settings for the size of I/O tables. Specifically, the number of rows can be 10, 100 or 1,000 while the number of columns can be 1 to 300. For each pair of row and column sizes, we measured synthesis time taken to find the solution by PatSQL and Scythe. The timeout is set to be 100 seconds as in the other experiments.

Figure 11 shows the result. For the projection query q1, the time taken by PatSQL grows gradually as the column size increases. This is the case even when the number of rows is larger, such as 1,000 rows. In contrast, Scythe has difficulty finding a solution when the column size is 10 or more even though the number of rows is 10. Scythe was unable to find any solution for the case of 100 and 1,000 rows. For the other two queries q2 and q3, PatSQL was able to handle a large number of rows and columns that Scythe was not able to handle. We also found that the time taken by PatSQL gradually increased with the column size while that of Scythe increased significantly. In summary, this experiment shows PatSQL's strength in the scalability of I/O tables.

8 RELATED WORK

Programming by Example (PBE) is a technique that synthesizes programs from given I/O examples, and has been studied intensively in recent years [15, 16, 28]. PBE has been applied to help non-experts in a wide range of domains such as string manipulation [10, 14], data migration [39, 40], data extraction [24] and MapReduce program [34]. Several studies have proposed techniques that synthesize expressive SQL queries from I/O tables [7, 31, 37, 43]. SQLSynthesizer [43] employs a kind of the decision tree algorithm to construct appropriate predicates in the WHERE clause. SqlSol [7] uses an off-the-shelf SMT solver to build the entire query by encoding SQL components and tables into logic constraints. SQL-Synthesizer and SqlSol differ from our technique in that they do not support nested subqueries. SCYTHE [37] enumerates abstract queries in a bottom-up manner and instantiates each of them by encoding tables in bit-vectors. SQUARES [31] is an SQL synthesizer developed on top of a state-of-the-art synthesis framework Trinity [27]. SCYTHE and SQUARES have similar performance on small examples. SQUARES performs better on tables having a large number of rows than SCYTHE, but it requires more types of hints including aggregation functions and attribute names. We cannot expect from a non-expert user to provide millions of rows in order to know how to write a correct query. Our tool PATSQL significantly outperforms SCYTHE, as evidenced in Section 7, by utilizing properties in relational algebra. Our algorithm is also closely related to techniques that deal with table structures as I/O examples such as data frame manipulation [5, 12], tensor manipulation [32] and data visualization [38]. These algorithms focus on methods that leverage the properties of table structures to enable efficient synthesis. Examples of them include pruning by table inclusion relations [37, 38], pruning by constraints on table metadata such as the number of columns and rows [12] and machine learning by transforming a table into a graph structure [5]. Also, many of them adopt the concept of sketch, which determines and validates the program structure before enumerating concrete programs, as our algorithm does. Thanks to these improvements, a rich set of syntax features and complex program structures have been supported. However, these existing techniques suffer exponential increases in computational cost as the number of columns increases. This challenge makes it difficult to apply the PBE methods in practical scenarios. Note that, to avoid such combinatorial explosions, some algorithms [5, 31, 43] are based on the assumption that the corresponding column names are always the same, which we believe is not practical especially when the user is not an expert. To address these issues, PATSQL deals with I/O tables having full-scale schemas with no constraints on the column names.

Query by Example (QBE) is a technique for formulating database queries from several examples of records (and counterexamples in some cases) that the user wants to retrieve from tables [11]. QBE has been actively studied, and a variety of techniques have been proposed [11, 20, 26, 36]. In a typical setting for QBE, the target table is an existing one in a database, and therefore executing a query may take long time when the table has a large number of records. Thus, these algorithms need to deal with the scalability as the number of rows in the table increases, which our algorithm does not need to focus on. Also, some techniques assume that helpful information such as used constants or used tables is not available. Despite these strict limitations, SQUID [11] restores queries with aggregation and grouping by pre-computeing statistics of semantic properties to construct an abduction-ready database. Although a rich set of syntax features has been supported, there are no QBE methods that support more advanced features such as nested query or window functions as far as we know.

Synthesis Algorithms. A wide variety of improvements in program synthesis algorithm have been proposed. Our algorithm is categorized as an enumerative synthesis algorithm, which is one of the most successful strategies [16]. Smith et al. [35] proposed a program synthesis technique that only enumerates programs in *normal form* to reduce the search space. The normal forms are computed from a set of rewrite rules on program structures. Although this concept is similar to our restriction on sketch structures in Table 1, we focus on restricting program structures for enabling efficient sketch completion, rather than reducing the search space on program structures. We also highlight recent advances in synthesis techniques that employ machine learning or stochastic models [2, 4, 21, 25, 28, 30]. For example, SKETCHADAPT [30] prioritizes sketches that may lead to a desired query based on a recurrent neural network model, which has learned patterns from codebases. Introducing such sketch priorities into our algorithm would be straightforward since our sketch completion depends only on the target sketch.

Natural Language Interface (NLI) is another approach than PBE for helping non-experts to write queries [1]. In particular, a variety of techniques for translating specifications in natural language into SQL queries have been studied [6, 17, 19, 41]. For example, SyntaxSQLNet [41] is a text-to-SQL generator that employs an SQL specific syntax tree-based decoder. It supports a rich set of SQL features such as aggregation, union and nested query. However, specifications in natural language tend to be ambiguous compared to I/O examples. To address this issue, DUOQUEST [3] aims to combine PBE and NLI approaches to meet user's practical demands. We believe each advance in PBE and NLI technologies will also contribute to advances in such dual-specification approaches.

9 CONCLUSION

We have presented an SQL synthesizer called PATSQL, which synthesizes expressive SQL queries from input and output tables. PATSQL is the first SQL synthesizer that integrates properties known in relational algebra into sketch-based program synthesis. PATSQL employs a novel form of constraints and its top-down propagation mechanism for efficient sketch completion. We have shown that PATSQL outperforms a state-of-the-art algorithm SCYTHE in both the execution time and the scalability of input and output tables even though PATSQL does not employ hints about aggregation functions required by SCYTHE.

An immediate future direction is to develop synthesis algorithms that support additional syntax elements and are able to find more complex program structures such as advanced SQL queries seen in Kaggle's tutorial. Another direction is to work with I/O examples that may contain incorrect values. In recent years, the program synthesis from such noisy data has been studied in several domains [10, 18]. It will also be interesting to consider the performance of synthesized queries although the performance can depend on the database indexes tuned for target tables. More broadly, we believe that it is important to find out the requirements for the practical use of PBE tools through user studies and continue to improve the algorithms and user interfaces.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number JP20H05706.

REFERENCES

[1] Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. 2019. A comparative survey of recent natural language interfaces for databases. *The VLDB Journal* 28 (2019), 793–819.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* 51, 4, Article 81 (2018).

[3] Christopher Baik, Zhongjun Jin, Michael Cafarella, and H. V. Jagadish. 2020. Duoquest: A Dual-Specification System for Expressive SQL Queries. *CoRR* abs/2003.07438 (2020).

[4] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proceedings of the International Conference on Learning Representations*.

[5] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proceedings of the ACM on Programming Languages (OOPSLA)* 3, 1–27.

[6] Ben Bogin, Jonathan Berant, and Matt Gardner. 2019. Representing Schema Structure with Graph Neural Networks for Text-to-SQL Parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4560–4565.

[7] Lin Cheng. 2019. SqlSol: An accurate SQL Query Synthesizer. In *International Conference on Formal Engineering Methods*. 104–120.

[8] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2012. Inferring SQL Queries Using Program Synthesis. *CoRR* abs/1208.2013 (2012).

[9] Jan Van den Bussche and Stijn Vansummeren. 2009. Translating SQL into the relational algebra. *Course notes, Hasselt University and Université Libre de Bruxelles* (2009).

[10] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning*. 990–998.

[11] Anna Fariha and Alexandra Meliou. 2019. Example-Driven Query Intent Discovery: Abductive Reasoning Using Semantic Similarity. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1262–1275.

[12] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 422–436.

[13] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book, Second Edition*. Prentice Hall Press, Upper Saddle River, NJ, USA.

[14] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[15] Sumit Gulwani and Prateek Jain. 2019. Programming by Examples: PL meets ML. *Dependable Software Systems Engineering* (2019).

[16] Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119.

[17] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4524–4535.

[18] Shivam Handa and Martin C. Rinard. 2020. Inductive program synthesis over noisy data. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 87–98.

[19] Po-Sen Huang, Chenglong Wang, Rishabh Singh, Wen tau Yih, and Xiaodong He. 2018. Natural Language to Structured Query Generation via Meta-Learning. In *The 16th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

[20] Dmitri V. Kalashnikov, Laks V.S. Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *Proceedings of the 2018 International Conference on Management of Data*. 337–350.

[21] Neel Kant. 2018. Recent Advances in Neural Program Synthesis. *CoRR* abs/1802.02353 (2018).

[22] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2018. Data Scientists in Software Teams: State of the Art and Challenges. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1024–1038.

[23] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The road to live programming: insights from the practice. In *Proceedings of the 40th International Conference on Software Engineering*. 1090–1101.

[24] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA, 542–553.

[25] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 436–449.

[26] Denis Mayr Lima Martins. 2019. Reverse engineering database queries from examples: State-of-the-art, challenges, and research opportunities. *Information Systems* 83 (2019), 89–100.

[27] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An Extensible Synthesis Framework for Data Science. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1914–1917.

[28] Aditya Krishna Menon, Omer Tamuz , Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. 2013. A Machine Learning Framework for Programming by Example. In *Proceedings of the 30th International Conference on Machine Learning*. 187–195.

[29] Fiona Nah. 2003. A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait?. In *The Americas Conference on Information Systems*. 2212–2222.

[30] Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2019. Learning to Infer Program Sketches. In *Proceedings of the 36th International Conference on Machine Learning*. 4861–4870.

[31] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco Manquinho. 2020. SQUARES : A SQL Synthesizer Using Query Reverse Engineering. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2853–2856.

[32] Kensen Shi, David W. Bieber, and Rishabh Singh. 2020. TF-Coder: Program Synthesis for Tensor Manipulations. *CoRR* abs/2003.09040 (2020).

[33] Alyssa C. Smith, Brandon C. W. Ralph, Jeremy Marty-Dugas, and Daniel Smilek. 2019. Loading… loading… The influence of download time on information search. *PLOS ONE* 14 (2019), 1–24.

[34] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 326–340.

[35] Calvin Smith and Aws Albarghouthi. 2019. Program Synthesis with Equivalence Reduction. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. 24–47.

[36] Wei Chit Tan, Meihui Zhang, Hazem Elmeleegy, and Divesh Srivastava. 2017. Reverse Engineering Aggregation Queries. *Proceedings of the VLDB Endowment* 10, 11, 1394–1405.

[37] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 452–466.

[38] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2020. Visualization by Example. In *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages*.

[39] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration Using Datalog Program Synthesis. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1006–1019.

[40] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-Example. *Proceedings of the VLDB Endowment* 11, 5 (2018), 580–593.

[41] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-Domain Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 1653–1663.

[42] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.

[43] Sai Zhang and Yuyin Sun. 2013. Automatically Synthesizing SQL Queries from Input-output Examples. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. 224–234.