Synthesizing Structurally Rich SQL Queries

Abstract

SQL is central to relational database systems and has a large number of users. Though being expressive and powerful, SQL is hard to write since its declarative language design forces users to write structurally complex queries for many common tasks, such as moving average or argmax. As we have observed in many online Q&A forums that users often asks help from experts by presenting input-output examples, it would be desirable to build a programming-by-example (PBE) system for SQL to increase the usability of the language. However, due to the structurally rich nature of SQL, none of the existing synthesis algorithm can be satisfyingly adopted to solve SQL query synthesis task.

To address this synthesis challenge, we designed 1) a symmetry-reduced subset of SQL as the target synthesis language, and 2) a new inductive synthesis algorithm that uses abstract representations to increase the efficiency of search space traversal. Our key insight in designing the synthesis algorithm is that we can use abstract representations to reduce the original search problem and bridge the information from the input examples and output example so that we can perform searching and pruning in a more efficient way. With this new synthesis algorithm, we have built a PBE system Scythe and evaluated it on a set of 60 benchmarks from the online Q&A forum. The evaluation result indicates that our algorithm is efficient enough to enable us to solve most problems within seconds.

1. Introduction

Relational database serves an important role in modern data management, and SQL is one of the most commonly used query language. While SQL is designed declaratively and easy-to-use for many basic tasks (e.g. selecting some columns or rows from a table), its also forces users to write structurally complex queries with nested subqueries for some other tasks that cannot be easily expressed declaratively in SQL (e.g. the ArgMax problem, querying the row with greatest or least values for each group). As a result, correctly formulating such queries remains highly challenging for non-expert users: in Stack Overflow, there is a frequently asked question tag "greatest-n-per-group"[a] with more than 1,600 posts only for the SQL ArgMax problem, and there are even more questions posted under tags "sql","mysql" or "postgresql" asking for help on writing such structural complex queries.

In many online Q&A forums like Stack Overflow and Tek-Tips, users ask questions by providing input-output examples to experts: they briefly describe their question, and provide several input tables and an output table to demonstrate the query behavior. Especially for tasks involving analytical queries (e.g., argmax, moving average), users choose to demonstrate their problems using examples since pure English descriptions are often too abstract or too ambiguous for expert to correctly understand. As a result, existing natural language query systems [20] cannot solve these problems and it would be highly desirable to develop programming-by-example (PBE) systems to enable automatic synthesis of queries from these examples to help solve the challenge in SQL formulation.

Although many successful PBE systems have been built for different domains including spreadsheet data transformation [12; 30; 14; 16] and text extraction [18], none of existing PBE systems for SQL [37; 35; 14] can satisfyingly solve the problem, since the subset of SQL used in these systems is too restrict to solve many real world problems.

The challenge of building programming-by-example systems for SQL is that synthesizing structurally rich SQL queries (e.g., queries with aggregations, subqueries) using traditional synthesis algorithms is prohibitively expensive. The three most common approaches to synthesis are enumerative, constraint solver-based, and version space algebra (VSA)-based. Enumerative search approaches [26; 25; 9] requires evaluating all intermediate enumerated results to enable memoization and equivalent-class partitioning, but SQL queries are relatively expensive to evaluate (comparing arithmetic expression) and more memory demanding to memoize (since intermediate results are tables). VSA-based approaches [27] requires that operators in the target language can be considered independently to each other to work efficiently, as that is a necessary condition for the algorithm to efficiently learn each operator independently from the output example. Finally, constraint-solver based approaches employ SMT solvers to synthesize programs, yet they do not work well in SQL since no efficient solver has been developed for the theory of relations.

Our key insight to the challenges brought up by the structural complexity of SQL language is that we built intermediate abstract representations to step-wisely reduce the problem into a simpler problem that we could easily utilize information from both input and output examples to efficiently prune and traverse each search space efficiently to find candidate queries.

First, we designed the SynthSQL grammar, a SQL grammar with reduced syntactical symmetries, to reduce the problem of searching SQL queries to searching SynthSQL queries that are consistent with input-output examples. Second, we designed a *forward abstraction algorithm* and a data structure *summary table* to divide the query space into numerous subspaces and store them compactly to represent all possible queries that can be built from user inputs. Particularly, this intermediate representation helps reduce the original problem of searching consistent queries into a simpler problem of searching for bitvectors and integer lists satisfying some property. Third, for each subspace, we designed a data structure *trace* and a *backward inference algorithm* that uses information from the output example to further simplify the search problem in a way that the output example can be used in pruning the search space. Last, we designed a *merge algorithm* that uses information stored in both forward-backward abstract representations to efficiently derive queries that are consistent with input-output examples from each subspace.

Based on the core synthesis algorithm described above, we built a programming-by-example system Scythe to address SQL query synthesis problem. Scythe also contains a heuristic ranking algorithm to rank the candidates, and an interface that allows user

[a] http://stackoverflow.com/questions/tagged/greatest-n-per-group

to modify input-output examples to refine the results when none of the top ranked queries are correct. Scythe performs efficiently in practice and is able to solve a rich set of problems in the real world.

Contributions This paper makes the following contributions:

- We devised a new SQL grammar SynthSQL that reduces symmetries in SQL to increase search efficiency.
- We designed new synthesis algorithms that use abstract representations to reduce the original synthesis problem to simpler search problem to overcome the challenges in synthesizing structurally complex SQL queries.
- We implemented our algorithm, and evaluated it on a set of 60 real world benchmarks collected from online Q&A forums. The evaluation result shows that our synthesis algorithm is very practical and can solve 55 real world problems within seconds.

2. Motivating Examples

To better understand the SQL synthesis task, we present two representative motivating example picked up from Stack Overflow to demonstrate the key challenges in synthesizing structurally rich SQL queries and our insights in addressing these challenges.

EXAMPLE 1 Our first example is an ArgMax question. In this example, the user intends to find rows from input table whose `Value` field equals to the maximum `Value` below 8 for each `User` group. The user's post[b] contains the following problem descriptions, and an expert provided a solution sometime after the problem was posted, and the expert used a subquery in the `Join`-clause to write the correct answer.

- *Description*: "I have a table (left) and I want to select the maximum value lower than the threshold 8 for each user, I want result to be as follows (right)."
- *Input example (left) and output example (right)*:

User	Phone	Value
Peter	0	1
Peter	456	2
Peter	456	3
Paul	456	7
Paul	789	10

col1	col2	col3
Peter	456	3
Paul	456	7

- *Solution provided by an expert*:

```
Select t1.User, t1.Phone, t1.Value
From   (Select   User, Max(Value) As MaxValue
        From     input
        Where    Value < 8
        Group By User) As t2 Join input As t1
Where  t1.User = t2.User And t1.Value = t2.MaxValue
```

As shown by the expert, to solve the problem we first need to use an aggregation query to calculate the maximum `Value` under 8 for each `User`, and join the aggregation result with the input table on `User` and `Value` fields to find the phone number associated with these maximum `Value`.

EXAMPLE 2 This is an example that involves using a subquery in the `Exists`-clause to write the correct answer. In this example, the user intends to find phone calls in the `Call` table such that its `Id` is not in the `Phone_Book` table. The user's post[c] contains the following problem descriptions, and an expert provided the following solution.

- *Description*: "How do I find out which calls were made by people whose `Phone_Number` is not in the `Phone_Book` table?"
- *Input Example*: The input example contains two tables: `Phone_Book` (left) and `Call` (right).

Id	Date	Phone
1	0945	1111
2	0950	2222
3	1045	3333

Id	Name	Phone
1	John	1111
2	Jane	2222

- *Output Example*:

Id	Date	Phone
3	1045	3333

- *Solution provided by an expert*:

```
Select *
From   Call As t1
Where  Not Exists
  (Select *
   From   Phone_Book As t2
   Where  t2.Phone = t1.Phone)
```

The solution is to use `Select` query on the `Call` table, and the filter uses a `Not Exists`-clause to check whether there exists a rows in the table `Phone_Book` such that the `Phone` value is equal to the current `Phone` number.

From these examples (and more in our benchmark, as will be shown in Section 10), we made the following observations:

- Users ask questions by providing input-output example pairs (where the input example can contain multiple tables, and the output example is a single table) as well as a set of constants and aggregators in text (e.g., integer "8", aggregator "`Max`" in the first example), and the desirable query typically contains no arbitrary constants or aggregators outside of user provided ones.
- Although the output example can be simple (e.g., output tables in both examples are simply subtables of the input tables), the correct queries can be very different and contain complex query structures (e.g., subquery, aggregation, `Exists`-clause).

These observations motivate that our synthesis system should 1) use input-output example as well as constants, aggregators provided by the user to guide the search direction and 2) search in a large search space considering rich features in SQL even when the output example is simple.

However, the search space can be very large when rich features (subquery, aggregation etc) are considered. For example, given the query skeleton (a query whose filter predicates in `On`, `Where` and `Having` are represented as holes) in Figure 1, even when we restrict filters that can be filled in the holes to conjunctive filter predicates with maximum length 2, ignoring `Exists`-clauses, the number of syntactically different queries is already greater than 2,682,720. For queries that contain multiple `Join` or `Exists` clauses, the number of syntactically different queries represented by such skeletons can be as high as 10^{30}. Furthermore, due to the simplicity of the output examples, it is typically very hard to learn directly from the output example about the structure of the direct queries, which makes the search space very challenging for pruning purposes.

2.1 Our Approach

Our key insight in solving the aforementioned challenges is to build intermediate abstractions to reduce the original problem of searching SQL queries to some simpler search problems that can be efficiently guided by the output examples. Concretely, we solve the original problem with the following reduction steps.

Problem Reduction 1. Our first reduction of the search problem is that we find a core grammar of SQL language, SynthSQL, that contains less symmetry as our target language to reduce the size of

[b] http://stackoverflow.com/questions/33063073/postgresql-max-value-for-every-user

[c] http://stackoverflow.com/questions/367863/sql-find-records-from-one-table-which-dont-exist-in-another

```
Select ...
From (Select   User, Max(Value) As MaxValue
      From     input
      Where    Value < 8
      Group By User
      Having   ...) As t2
Join (Select * From input Where ...) As t1
On   ...
```

Figure 1. A query skeleton for the running example, where the ellipsis refers to holes in the skeleton (both in projection predicates and filter predicates).

our search space. SynthSQL grammar contains the following two features: 1) queries in SynthSQL are also SQL [23] queries and they can be directly executed in SQL interpreters, 2) compared to SQL, SynthSQL grammar contains additional syntactical restrictions to reduce symmetry in the search space (i.e., having fewer semantic equivalent queries).

For example, SynthSQL does not support casting a table to a value, and the following query (which is also a correct solution for EXAMPLE 1) will not be searched during the synthesis process.

```
Select t1.User, t1.Phone, t1.Value
From   input As t1
Where  t1.Value = (
          Select Max(Value)
          From   input
          Where  Value < 8 And User = t1.User)
```

However, as we will show in Section 4, benefiting from the semantics-preserving SQL rewriting rules [4], many of unsupported operators (e.g., `In`, `All`, casting table to values) can be desugared to operators in SynthSQL. In other words, the above query can be rewrote into a `Join` query, as shown in the solution of EXAMPLE 1. As a result, SynthSQL remains highly expressive and can capture many real world cases.

With the definition of the LigthtSQL, we can present our synthesis problem as below:

- *(Problem 1).* Given the grammar of SynthSQL, search for all queries that are consistent with input-output examples.

Problem Reduction 2. Our second reduction of the problem is to use an intermediate abstraction, query skeleton, to reduce the problem of searching syntactical components (filter predicates and projection statements) of SynthSQL queries into the problem of searching the values of these syntactical components.

The key idea of this reduction is based on the observation that many syntactically different filters have the same value in a query skeleton. For example, for the query skeleton in Figure 1, the predicate "`t1.Value=t2.MaxValue`" has the same effect as "`t1.Value>=t2.MaxValue`" when they are used in the `On`-clause. If we can avoid this type of repetition during the search process, search efficiency will be significantly boosted. In fact, especially for '`Exists`'-predicates, the reduction rate between the number of syntactically different primitive filter predicates[d] and the number of different values in a query skeleton can be as high as $1000\times$.

Our system performs this reduction in the following two steps. First, our system enumerates all different query skeletons and divides the whole search space of SynthSQL queries into numerous subspaces, each specified by a query skeleton. Second, we encode each primitive filter predicate into a bitvector and encode each projection statement as a list of integers based on their values in the query skeleton, and thus reduce the search problem to a search problem in the domain of bitvectors and integer lists.

Example Figure 2 shows three examples of how this reduction is performed. For the first query skeleton, $\tilde{Q}_1$=''`Select ... From input Where ...`'' (Figure 4), we use the table `input` in the skeleton (we call such table the *core table* of a skeleton) to encode all primitive filter predicates and projection statement. For example, the three filter predicates "`Phone < Value`", "`Phone <= Value`", and "`Phone < 8`" all evaluate to the bitvector $[10000]$ since only they evaluate to `True` on the first row of T_1 and `False` on all other rows. The filter predicate "`Value < 8`" evaluates to $[11110]$ since it evaluates to `True` on the first four rows but `false` on the last one.

In addition, each projection clause has a natural encoding to integer lists based on the indexes of the columns in the projection statement: the statement "`Select User, Phone`" can be mapped to an integer list $[1, 2]$, since `User` is the first column of `input` and `Phone` is the second column of the table `input`, and another projection statement "`Select User, Value, Phone`" is mapped to an integer list $[1, 3, 2]$.

Figure 2(b,c) are two other examples demonstrating the encodings for skeletons of an aggregation query and a join query skeleton. For aggregation queries, only primitive filter predicates in the `Having` clause can be encoded (as the aggregation result is dependent to filter predicates inside group by and thus they are part of the skeleton) and they are encoded based on the table evaluated from the query inside. For join queries, primitive filter predicates from both subqueries and `On`-clause are encoded based on the Cartesian product result of the two tables in the sub queries.

Based on these encodings, we have the following correspondences between the original problem and the new problem: 1) a primitive filter predicate is mapped as a bitvector β (and filtering a table T with β is to extract rows in T whose index bit in β is set to 1), 2) conjunctions of filter predicates are mapped to conjunctions of bitvectors (i.e., combining a list of bitvectors using the bitwise 'and' operator, for example the conjunction of $\{[110], [011]\}$ is the bitvector $[010]$), and 3) a projection statement corresponds to an integer list $\gamma = [i_1, .., i_n]$ (and projecting a table with γ is to select columns in the table, whose indexes are in γ). The original problem is then reduced as follows (Problem 1' below is an alternative presentation of Problem 1):

- *(Problem 1').* For each query skeleton, search for all pairs consisting of 1) a list primitive filter predicates and 2) a projection statement, such that filling the holes with these filter predicates and projection statements can generate a query that are consistent with input-output examples.
- *(Problem 2).* For each query skeleton, search for all pairs of form $(\bar{\beta}, \gamma)$ consisting of 1) a list of bit vectors (encoded from primitive filter predicates) and 2) an integer list (encoded from a projection statement) satisfying the following checking condition: filtering the core table of the skeleton with the conjunction of all bitvectors and projecting the result with the γ can produce T_{out} (the output example).

With this reduction, as long as we can find all such pairs consisting of a list of bitvectors and an integer list, we can decode them to obtain filter predicates projection statements and then further generate all candidate queries within the search space.

Example The pair $(\{[0000100100], [101010010101]\}, [1, 2, 3])$ for the problem in Figure 2(c) (highlighted as red) is a pair that can generate the output example in EXAMPLE 1 (by filtering the core table $T_1 \times T_2$ with the combination result of the two bitvectors and projecting with the the filter). And the pair can be decoded to the following two predicates in SQL.

[d] We use "primitive filter predicates" to refer to non-conjunctive filters, such as "`x < y`", "`Exists (...)`" and their negations. Since disjunction tend to be uncommonly used, SynthSQL does not provide its support and all filter predicates can be viewed as conjunctions of primitive filter predicates.

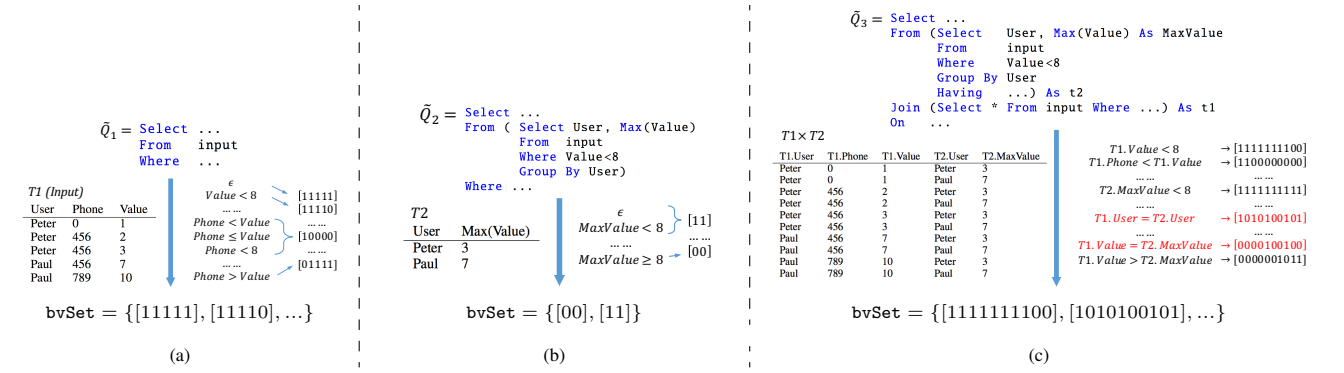

Figure 2. Encoding primitive filter predicates in query skeletons into bit vectors, based on their values on the core table of the skeleton.

- Filter: "`T1.Value=T2.MaxValue And T1.User=T1.User`"
- Projection: "`Select T1.User, T1.Phone, T1.Value`"

By filling the holes with these predicates, we can generate the desired query as presented in the solution of Example 1.

Problem Reduction 3. The third part of our problem reduction is to use information from the output example to simplify the checking condition in the reduced problem 2 to further increase the search efficiency.

The main drawback of problem 2 is that we need to enumerate and evaluate all $(\bar{\beta}, \gamma)$ pairs on the core table of a query skeleton to select candidates, and this process can be computationally expensive due to the large number of $(\bar{\beta}, \gamma)$ pairs and the extensive evaluation time for checking table equivalence.

Our solution to this problem is to evaluate from the output example and the core of the skeleton a set of bitvector-integer list pairs of form $P = \{\langle\beta, \gamma\rangle, ...\}$, containing all possible ways to obtain the output example T_{out} from the core table T by filtering and projection, (i.e., a bitvector β' and an integer list γ' can filter and project T to T_{out} if and only if the pair $\langle\beta', \gamma'\rangle \in P$).

Example For example, between the table T_1 and T_{out} in Figure 2(c), the set $P_1 = \{\langle[00110], [1, 2, 3]\rangle\}$ contains all possible ways to obtain T_{out} from T_1: the only way is to filter it with the bv-filter [00110] followed by a projection on the first three columns. Similarly, such set for T_2 and T_{out} is $P_2 = \emptyset$, since there is no way to evaluate T_2 to T_{out} by only using filtering and projection.

As a result, we can use the set P inferred between the core table of a skeleton and the T_{out} to further reduce the synthesis problem.

- *(Problem 3).* For each query skeleton, searching for all pairs of form $(\bar{\beta}, \gamma)$ consisting of 1) a list of bit vectors (encoded from primitive filter predicates) and 2) a integer list (encoded from a projection statement) that satisfies the following checking condition: suppose β' is the result generated by the conjunction of $\bar{\beta}$, then the pair $\langle\beta', \gamma\rangle \in P$, inferred between the core table of the skeleton and T_{out}.

With this reduction, we can efficiently use such set P to guide searching $(\bar{\beta}, \gamma)$ pairs in a query skeleton, for example, if $P = \emptyset$, we can directly prune the whole search space of the query with such skeleton, and furthermore, with a set $P \neq \emptyset$, we only need to search for combinations of bitvectors whose conjunction result is contained in a pair in P, and we can get γ for free.

2.2 System Pipeline

Based on these insights, we have designed algorithms and data structures to support the reduction and the final stage search process.

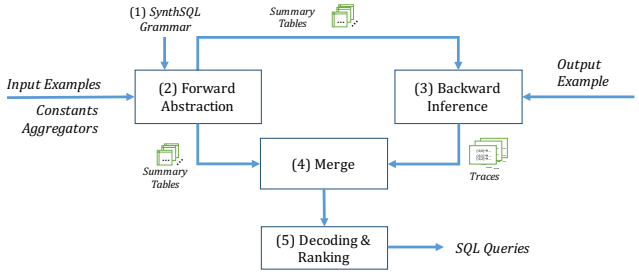

Figure 3. System overview.

As shown in Figure 3, the system synthesizes queries within SynthSQL grammar, and it takes four inputs: 1) a set of input examples (denoted as I), 2) a set of constants (optional) 3) a set of aggregators (optional), and 4) an output example (denoted as T_{out}). The output of the systems is a list of ranked queries that are consistent with the given input-output examples.

The system first takes the inputs from the user and uses the forward enumeration algorithm to lazily enumerate all query skeletons in the space of SynthSQL queries, and compactly stores them using the summary table data structure in order to perform the second reduction of our problem. Then for each query skeleton, the backward inference algorithm infers the set P between the core table of the skeleton and the output example (and compactly stores them as *traces* for the third reduction. Third, our algorithm uses a *merge* algorithm to efficiently search aforementioned $(\bar{\beta}, \gamma)$ pairs for each skeleton using the information from the the last two modules. Last, all these $(\bar{\beta}, \gamma)$ pairs are decoded into SQL queries and returned to the user after ranking them based on a heuristic.

3. Preliminaries

Before introducing our approach, we present terminologies used in this paper and formally define the representation of table.

- Our system inputs includes 1) a set of input tables, 2) a set of constants, 3) a set of aggregators, and 4) an output table. We use the term *Input-output examples* to refer to the tables that are provided to the system by an user: the input example is a set of tables, denoted as I, and the output example is a single table, denoted as T_{out} (our system takes only one input-output example pair at a time).
- A *consistent query* refers to a query Q that is consistent with the provided input-output examples, i.e., $Q(I) = T_{out}$.

$$
\begin{array}{rll}
T & ::= & (schema, content) & \text{(Table)} \\
schema & ::= & [c_1 : \tau_1, ..., c_m : \tau_m] & \text{(Schema)} \\
content & ::= & [r_1, ..., r_n] & \text{(Content)} \\
r & ::= & [v_1, ..., v_m] & \text{(Row)} \\
\tau & ::= & int \mid double \mid string \mid time \mid date & \text{(Value Types)} \\
\beta & ::= & [b_1 ... b_n] & \text{(Bv-Filters)} \\
\gamma & ::= & [i_1, ..., i_k] & \text{(Projectors)}
\end{array}
$$

$$
\begin{aligned}
&T_1 \times T_2 := (\texttt{schema}(T_1) +\!\!+ \texttt{schema}(T_2), \\
&\qquad [r_1 +\!\!+ r'_1, ..., r_1 +\!\!+ r'_{n_2}, ..., r_{n_1} +\!\!+ r'_1, ..., r_{n_1} +\!\!+ r'_{n_2}]) \\
&\qquad\quad where\ \texttt{Content}(T_1) = [r_1, ..., r_{n_1}] \\
&\qquad\quad\ \ and\ \texttt{Content}(T_2) = [r'_1, ..., r'_{n_2}] \\
&\texttt{filter}(T, \beta) := \\
&\qquad (\texttt{schema}(T), [r_i \mid \forall\, r_i \in \texttt{content}(T)\ and\ \beta[i] = 1]) \\
&\texttt{proj}(T, \gamma) := \\
&\qquad (\texttt{subList}(\texttt{schema}(T), \gamma),\ [\texttt{subList}(r, \gamma) \mid \forall\, r \in \texttt{content}(T)]) \\
&\texttt{deDup}(T) := (\texttt{schema}(T), \texttt{toList}(\{r_i \mid \forall\, r_i \in \texttt{content}(T)\})) \\
&\texttt{subList}(L, [i_1, ..., i_k]) := [v_i \mid v_i \in L\ and\ i \in [i_1, ..., i_k]]
\end{aligned}
$$

Figure 4. Table representation and operations on tables, where '$+\!\!+$' is the list-append operator, $\oplus$ is the bitwise-'and' operator, $\texttt{subList}(L, \gamma)$ extracts elements in L according to indexes stored in γ, and $\texttt{toList}(S)$ converts the set S to a list.

- Bit-wise and operator $\oplus$ is defined as below[e], and is promoted to bitvectors as $[b_1...b_n] \oplus [b'_1...b'_n] = [c_1...c_n]$ *where* $c_i = b_i \oplus b'_i$. Furthermore, we use $\bigoplus \bar{\beta}$ to refer to $\bigoplus_{\beta \in \bar{\beta}} \bar{\beta}$.

$$b_1 \oplus b_2 = \begin{cases} 1 & b_1 = 1\ or\ b_2 = 1 \\ 0 & otherwise \end{cases}$$

3.1 Relational Table

The definition of relational tables is presented in Figure 4. Relational table is defined as a pair of *schema* and *content*. The schema of a table is a list of pairs of column name and column type, the i-th element of the schema $c_i : \tau_i$ specifies that column i has name c_i, and all values in column i of the table are of type τ. The content of a table is a list of rows, and each row is a list of values, where the size of each row is required to be the same as the size of the schema. We use $\texttt{schema}(T)$ and $\texttt{content}(T)$ to refer to the schema and content of a table T, respectively.

For a table with n rows and m columns, the rows are indexed from 1 to n and columns from 1 to m, and we use coordinate (i, j) to refer to the cell in the i-th row and j-th column of the table.

Besides, we also define bv-filters (β) and projectors (γ) on a table: the former is a bitvector while the latter is a list of integers. The function $\texttt{filter}(T, \beta)$ applies a bv-filter $\beta = [b_1...b_n]$ to a table T, in a way that row i is kept if and only if b_i equals to 1, and the function $\texttt{proj}(T, \gamma)$ refers to projecting table T with a projector γ, in a way that column i is kept if and only if $i \in \gamma$. Other operators on tables include Cartesian product '$\times$', and `deDup`, which removes duplicate rows in a table.

Since we use bag semantics instead of set semantics (following SQL conventions), duplicate rows are allowed in a table and two tables are equivalent if and only if their contents are contained in each other, and the same rows in two tables have the same multiplicity (although row order in the table does not necessary need to be the same for equivalence).

[e] We use the notation '$\oplus$' to represent bit-and for bv-filters in order to distinguish it from logical 'and' operators used in other places of the paper.

4. SynthSQL Grammar

4.1 Syntax

The syntax of SynthSQL is presented in Figure 5. As presented in Section 2, SynthSQL is a subset of the SQL grammar with reduced symmetry while remains expressive enough to capture many real world problems. Many SQL queries using operators outside of SynthSQL grammar can be semantically-equivalently reduced to queries in SynthSQL using SQL rewriting rules.

The top level construct of a SynthSQL query Q is either an intermediate query q or a projection query built from q. An intermediate query q can be one of 1) named table reference, 2) filter query, 3) join query, 4) aggregation query and 5) rename query (where the table name and all columns are required to be specified). For SynthSQL, we use the term *primitive filter predicates* refers to filters of the form "$v\ binop\ v$" or "$(\textbf{not})\ \textbf{exists}\ q_e$", and use the term *conjunction of filter predicates* to refer to filters of the form "$f\ \textbf{and}\ f$".

Besides the above definitions, SynthSQL has the following additional restrictions on filter formulation.

- For primitive join predicates of the form $v_1\ binop\ v_2$ in a query construct "$q_1\ \textbf{join}\ q_2\ \textbf{on}\ f$", v_1 must refer to a column in q_1 while v_2 must refer to a column in q_2.
- For primitive filter predicates of form $v_1\ binop\ v_2$ in a `Having`-clause of an aggregation query, v_1 can only refer to an aggregation column $\alpha(c'_i)$.
- At least one of filter predicates of form $v_1\ binop\ v_2$ in a nested-subquery in `Exists`-clause have the property that v_1 refers a correlated column in outer level query.

$$
\begin{array}{rcll}
Q & ::= & q & \\
 & \mid & \textbf{select}\ (\textbf{distinct})\ c_1, ..., c_n\ \textbf{from}\ q & \text{(Projection)} \\
q & ::= & T & \text{(Table Reference)} \\
 & \mid & \textbf{select}\ *\ \textbf{from}\ T\ \textbf{where}\ f & \text{(Filtering)} \\
 & \mid & q_1\ \textbf{join}\ q_2\ \textbf{on}\ f & \text{(Join)} \\
 & \mid & \textbf{select}\ c_1, ..., c_n, \alpha(c'_1), ..., \alpha(c'_m)\ \textbf{from}\ q\ \textbf{group by}\ c_1, ..., c_n\ \textbf{having}\ f & \text{(Aggregation)} \\
 & \mid & q\ \textbf{as}\ name[c_1, ..., c_n] & \text{(Rename)} \\
f & ::= & f\ \textbf{and}\ f & \\
 & \mid & v\ binop\ v & \\
 & \mid & (\textbf{not})\ \textbf{exists}\ q_e & \\
q_e & ::= & \textbf{select}\ *\ \textbf{from}\ T\ \textbf{where}\ f & \\
v & ::= & c & \\
 & \mid & constant & \\
\alpha & ::= & max, min, sum, avg, count, concat & \\
binop & ::= & =, >, <, >=, <=, <> &
\end{array}
$$

Figure 5. SynthSQL syntax, where meta-variable c ranges over column names, T ranges over table names from user inputs, *name* ranges over newly assigned table names.

Although SynthSQL grammar contains only a restricted set operators in SQL, it remains very expressive, benefiting from semantics-preserving query rewriting rules [4]. For example, although SynthSQL does not directly support '`In`', '`All`' operators in filter, or casting table to values, SQL queries written with these syntax constructs can be transformed to queries in SynthSQL, and thus are still supported. For example, subquery-unnesting rewriting rule [10] is able to convert table to value casting queries to queries that have subqueries in the `Join`-clause.

These restrictions on the syntax of SynthSQL reduce *symmetry* in the search space, i.e., there are fewer queries that are semantically equivalent but with different syntactical forms in the search

space comparing to SQL, which reduce the complexity of the process of search space traversal.

Limitations There are also queries in SQL that cannot be semantics-equivalently transformed to SynthSQL and these queries are not in the query search space of SynthSQL: e.g., `Union`, `Outer Join`, disjunction filter predicates and arithmetic expressions. However, as we can see in evaluation, these operator occurs less common in online question posts and our language support is expressive enough to capture most real-world use cases.

4.2 Semantics

Since SynthSQL queries are also native SQL queries, they can be interpreted with SQL interpreters. Particularly, we use bag-semantics of SQL in this paper, which is formally defined by Negri et al. [23], and commonly used in many modern database systems including MySQL, SQL Server, etc.

5. Synthesis Algorithm

We first present our synthesis algorithm. The algorithm takes four inputs: 1) a set of input example tables I, 2) an output example table T_{out}, 3) a set of constants $consts$, and 4) a set of aggregators. The algorithm synthesizes a list of ranked queries in SynthSQL grammar that are consistent with the input-output example. The synthesis algorithm contains the following steps.

1. *(Forward Abstraction)* The algorithm iterates over the complexity of the skeletons, starting from 1, and then enumerates all possible query skeletons within the complexity level that can be built from the input tables, constants and aggregators provided by user (line 4). (Since there are infinitely many queries consistent with input-output examples, we use the variable *complexity* to decide when to communicate the results to the user.)
2. *(Backward Inference)* For each summary table ρ generated from forward enumeration, infer the trace σ between $\texttt{coreTable}(\rho)$ (i.e., the core table of ρ) and the output example T_{out} (line 7).
3. *(Merge)* Use the trace σ to guide the search in the subspace specified by the summary table ρ to obtain a set of filter-projector pairs in ρ, which are encodings of all consistent queries in the subspace specified by ρ (line 8).
4. *(Decode)* Decode each $(\bar{\beta}, \gamma)$ pair on the summary table ρ to obtain a list of candidate queries that are consistent with the given input-output example (lines 9,10).
5. *(Ranking and User Interaction)* Rank all candidate queries synthesized in this complexity level (if not empty) and present them to the user (lines 11,13). If the user accepts the answer, the algorithm terminates, otherwise continue searching for more complex queries in the search space (lines 13). (A user can also provide new examples to the system to restart the synthesis process on the new inputs, when none of these synthesized queries is correct.)

In the following sections, we formally define the intermediate data structures and elaborate on the synthesis modules used in the algorithm.

6. Forward Abstraction

The first module of our algorithm is forward abstraction, where we enumerate all query skeletons in the search space and construct summary tables from them to divide the space of SynthSQL into subspaces and reduce the problem of searching queries within the subspace into a simpler problem of searching for bv-filters and projectors on a table.

```
1  Synthesis(I, T_out, consts, aggregators)
2    complexity ← 1
3    while True:
4      S ← forwardEnum(I, consts, aggregators, complexity)
5      candidates ← ∅
6      foreach ρ ∈ S:
7        σ ← backwardInference(coreTable(ρ), T_out)
8        R ← merge(ρ, σ)
9        foreach (β̄, γ) ∈ R:
10         candidates ← candidates ∪ decode(ρ, β̄, γ)
11     candidates ← rank(candidates)
12     if candidates ≠ ∅:
13       feedback ← communicate(candidates)
14       if feedback = 'Accepted':
15         break;
16     complexity ← complexity + 1
```

Figure 6. Synthesis Algorithm.

$$
\begin{array}{rcll}
\rho & ::= & \texttt{ST}_\texttt{p}(T, B, Q) & \textit{(Primitive Summary Table)} \\
 & | & \texttt{ST}_\texttt{c}(\rho_1, \rho_2, B) & \textit{(Compound Summary Table)} \\
B & ::= & \{\beta_1, ..., \beta_k\} & \textit{(Bv-filter set)}
\end{array}
$$

$$\texttt{coreTable}(\texttt{ST}_\texttt{p}(T, B, Q)) := T$$

$$\texttt{coreTable}(\texttt{ST}_\texttt{c}(\rho_1, \rho_2, B)) := \texttt{coreTable}(\rho_1) \times \texttt{coreTable}(\rho_2)$$

$$\texttt{projSet}(\rho) := \{[i_1, ..., i_k] \mid 1 \le k, \; 1 \le i_k \le \texttt{colNum}(\texttt{coreTable}(T))\}$$

$$\texttt{bvSet}(\texttt{ST}_\texttt{p}(T, B, Q)) := B$$

$$\texttt{bvSet}(\texttt{ST}_\texttt{c}(\rho_1, \rho_2, B)) := B \cup \{\texttt{pullUpL}(\beta, T_1, T_2) \mid \beta \in \texttt{bvSet}(\rho_1)\} \cup \{\texttt{pullUpR}(\beta, T_1, T_2) \mid \beta \in \texttt{bvSet}(\rho_2)\} \quad \textit{where } T_i = \texttt{coreTable}(\rho_i)_{i=1,2}$$

$$\texttt{pullUpL}([b_1...b_n], T_1, T_2) := [b_{11}...b_{1m}...b_{n1}...b_{nm}], \quad \textit{where } b_{ij} = b_{i,(i\in[1,n],j\in[1,m])} \textit{ and } m = \texttt{rowNum}(T_2)$$

$$\texttt{pullUpR}([b_1...b_m], T_1, T_2) := [b_{11}...b_{1m}...b_{n1}...b_{nm}], \quad \textit{where } b_{ij} = b_{j,(i\in[1,n],j\in[1,m])} \textit{ and } n = \texttt{rowNum}(T_1)$$

Figure 7. Definition of summary tables and operations on them, where $maxLen$ stands for the length limit for conjunctive filters (predefined in the system).

6.1 Data Structure: Summary Table

Summary tables are defined in Figure 7 and they are the data structures used for representing these subspaces.

A primitive summary table $\rho = \texttt{ST}_\texttt{p}(T, B, Q)$ represents the subspace specified by a query skeleton of form "`Select...From Q Where...`", where Q is either an aggregation query or a reference to an input table, and it consists of three components: 1) T is the table evaluated from the query Q, 2) B is a bv-filter set, containing values of all primitive filter predicates that can be filled into the hole in the query skeleton and 3) the query Q inside the skeleton.

A compound summary table $\rho = \texttt{ST}_\texttt{c}(\rho_1, \rho_2, B)$ captures all queries with the skeleton "`Select...From` $\tilde{Q}_1$ `Join` $\tilde{Q}_2$ `On...`", where $\tilde{Q}_1$ is the skeleton represented by ρ_1, $\tilde{Q}_2$ is the skeleton represented by $\tilde{Q}_2$ (where the projection clauses of both sub-summary tables are '`Select *`', according to the SynthSQL grammar). The bv-filter set B stores values of all primitive join predicates that can be filled in the `On` clause, evaluated on $\texttt{coreTable}(\rho)$ (which is defined as Cartesian product of core tables of sub summary tables).

6.2 Enumeration Algorithm

The forward enumeration starts from input tables, constant set and aggregator set provided by the user, and a complexity control

```
1  forwardEnum(I, consts, aggregators, k)
2    if k = 1:
3      return enumFilteringQuery(I, consts, maxLen)
4    S ← forwardEnum(I, consts, aggregators, k − 1)
5    S_A ← enumAggrQuery(S, aggregators)
6    S_J ← ∅
7    forall (i, j) s.t. i + j = k :
8      S_1 ← forwardEnum(I, consts, aggregators, i),
9      S_2 ← forwardEnum(I, consts, aggregators, j),
10     S_J ← S_J ∪ enumJoinQuery(S_1, S_2)
11   return S_A ∪ S_J
```

Figure 8. Enumeration algorithm for complexity level k.

integer, and it returns all query summary tables of complexity level k tables that can be constructed form these inputs.

The input of the forward enumeration algorithm (Algorithm 8) includes 1) a set of input example I, 2) a set of constants, 3) a set of aggregators and 4) an integer k, specifying the complexity level of the summary tables to be enumerated. And the enumeration algorithm returns all summary tables of complexity level k: if k equals 0, the result would be all summary tables built from input examples (line 3), otherwise, the algorithm builds summary tables from aggregation queries or join queries by recursively invoking the enumeration function (lines 4-11).

The three enumeration modules used in the algorithm are presented below (Figure 9).

- `enumFilteringQuery` enumerates and encodes primitive filters for all tables provided by the user, including both binary comparison predicates (i.e., v_1 *op* v_2) and exists-filters. Binary comparison predicates are generated by `enumPrimFilters`, where v_1 is a column name, and v_2 can be either a column name or a constant with same type as v_1 from *consts*. For `Exists`-predicates, to avoid re-enumerating these clauses, instead of directly enumerating them, we first construct a sketch of the clause using the function `enumExistsSkeleton`, and instantiate them to filter predicates by substituting columns in the table with holes in a skeleton query to form a correlated column reference.
- `enumAggrQuery` enumerates queries with aggregations with given summary tables and aggregators. It first instantiates all conjunctive bv-filters represented by each summary table (lines 3,4), then evaluate each bv-filter on the core table to obtain a table T. The algorithm then enumerates and evaluates all aggregation queries that can be build from T, and build a primitive summary table to store the enumeration result.
- `enumJoinQuery` takes two sets of summary tables as input and build compound summary tables from them. For every ρ_1, ρ_2 such that each ρ one summary table set, the algorithm enumerates all primitive join predicates between these two sub-summary tables and encodes them on $\texttt{coreTable}(\rho_1) \times \texttt{coreTable}(\rho_2)$ as bit vectors to build a summary table.

At the end of the forward enumeration process, a set of summary tables is generated, and with these abstract representations, we can perform the following reduction on the original search problem of consistent query within the search space of SynthSQL.

6.3 Problem Reduction

Before introducing how the original problem can be reduced with summary tables, we first introduce one key operator on a summary table ρ: $\texttt{bvSet}(\rho)$. The function $\texttt{bvSet}(\rho)$ computes the values of all primitive filter predicates that can be filled in the query skeleton: for primitive summary tables (evaluated on $\texttt{coreTable}(row)$). For primitive summary tables, the result is the set B stored in the summary table, since B already contains the values of all filter predicates that can be filled in the "`Where...`" clause of the skeleton. While for compound summary tables, since the set B only stores the values of join predicates encoded on $\texttt{coreTable}(\rho)$ but not filter predicates inside sub summary tables, we need to generate the encoding of these filter predicates from the two sub summary tables.

We solve this problem by using the two "`pullUp`" operators: since these filter predicates are already encoded in sub summary tables (evaluated on $\texttt{coreTable}(\rho_1)$ and $\texttt{coreTable}(\rho_2)$), we can reuse these encodings to generate their corresponding encodings on $\texttt{coreTable}(\rho)$. The correctness of these "`pullUp`" operators are based on the following two equations, which indicates that as long as β is an predicate encoded on one of the sub summary table, the pulling up results (e.g., $\texttt{pullUpL}(\beta, T_1, T_2)$) has the same effect as β in the query skeleton, and thus it forms a correct encoding on $\texttt{coreTable}(\rho)$.

$$\texttt{filter}(T_1 \times T_2, \texttt{pullUpL}(\beta, T_1, T_2)) = \texttt{filter}(T_1, \beta) \times T_2$$
$$\texttt{filter}(T_1 \times T_2, \texttt{pullUpR}(\beta, T_1, T_2)) = T_1 \times \texttt{filter}(T_2, \beta)$$

In addition, the function $\texttt{projSet}(\rho)$ enumerates all combinations of integers between 1 and $\texttt{colNum}(\texttt{coreTable}(\rho))$, representing all projectors on the core table of ρ.

Problem 2. Given a summary table ρ, the problem of traversing the subspace to find all candidate queries that are consistent with input-output examples is reduced to the problem of finding all $(\bar{\beta}, \gamma)$ pairs such that:

$$\bar{\beta} \subseteq \texttt{bvSet}(\rho),$$
$$\gamma \in \texttt{projSet}(\rho),$$
$$\texttt{proj}(\texttt{filter}(\texttt{coreTable}(\rho), \bigoplus \bar{\beta}), \gamma) = T_{out}.$$

And if we can find all these $(\bar{\beta}, \gamma)$ pairs, we can 1) decode each primitive filter and the projection predicate to a syntactical predicate, and 2) fill the holes in the skeleton with these decoded predicates to generate queries that are consistent with input-output examples. The correctness of this reduction is presented by the following lemma.

Lemma 1. *(Complete of Summary Tables) Suppose Q is a query expressible in SynthSQL within inputs $(I, consts, aggregators)$, then there exists a summary table ρ from the enumeration result $\texttt{forwardEnum}(I, consts, aggregators, k)$, for some integer k, s.t. exists $\bar{\beta} \subseteq \texttt{bvSet}(\rho)$, $\gamma \in \texttt{projSet}(\rho)$ and the equation $\texttt{proj}(\texttt{filter}(T, \bigoplus \bar{\beta}), \gamma) = \texttt{eval}(Q)$ holds (where $T = \texttt{baseTable}(\rho)$).*

Proof. We first prove that for each constructor of q in $SynthSQL$ grammar, there exists a summary table ρ and a set of bv-filter $\bar{\beta} \in \texttt{bvSet}(\rho)$ such that $\texttt{filter}(T, \bigoplus \bar{\beta}) = \texttt{eval}(Q)$. This proof can be done by induction on the constructor of q in $SynthSQL$ grammar:

- If $q := T_0$ (the first constructor of q) where $T_0 \in I$ (the input table set), it is obvious that $\texttt{ST}_\texttt{p}(T_0, B, T_0)$ is in the result set of $\texttt{enumFilteringQuery}(I, const, holes)$ and the filters $\bar{\beta} = \{[11...1]\}$ satisfy the property (which is the default empty filter added into the set B).
- If $q :=$ **select** $*$ **from** T_0 **where** f. Then consider the summary table $\rho = \texttt{ST}_\texttt{p}(T_0, B, T_0)$. If the filter f is a primitive filter f_0, then there exists $\beta_0 \in B$ such that $\texttt{eval}(T_0, f_0) = \beta_0$ as the encoding filters will encode all primitive filters into bv-filters and we have $\rho, \bar{\beta} = \{\beta_0\}$ satisfying the result. If f is a compound primitive filter, i.e. $f = f_1$ **and** ... **and** f_n, we can find the bv-filter set $\bar{\beta} = \{\beta_1, ..., \beta_n\}$ satisfying the property, where b_i is the encoding result of f_i on T_0.
- If q is an aggregation query (with nested subquery q_0), by induction, there exists a summary table ρ_0 and a set of bv-filters $\bar{\beta}_0 \in \texttt{bvSet}(\rho_0)$ s.t. $\texttt{filter}(T_0, \bigoplus \bar{\beta}_0) = \texttt{eval}(q_0)$.

Since all group by fields are enumerated in `enumAggrQuery` on $\texttt{eval}(q_0)$, there exists a summary table $\texttt{ST}_\texttt{p}(T_1, B, Q)$ such that T_1 is the aggregation result of T_0 on the group by fields. Similar to the second case, each filter in **having** clause has a corresponding encoding and the property is satisfied.

- If q is a join query of the form $q = q_1$ **join** q_2 **where** f, by induction, there exists $\rho_1, \rho_2, \bar{\beta}_1, \bar{\beta}_2$, such that $\texttt{eval}(q_i) = \texttt{filter}(T_i, \bigoplus \bar{\beta}_i)$. Since a compound summary table $\rho = (\rho_1, \rho_2, B)$ is generated in `enumCompoundTable` algorithm and $\texttt{eval}(\text{“}q_1 \textbf{ join } q_2\text{”}) = \texttt{filter}(T_1 \times T_2, \bigoplus \bar{\rho}'_1 \oplus \bigoplus \bar{\rho}'_2)$, where $\bar{\beta}'_1 = \{\texttt{pullUpL}(\beta_1, T_1, T_2) \mid \beta_1 \in \bar{\beta}_1\}$ ($\texttt{pullUpR}$ for $\bar{\beta}'_2$). Furthermore, since the filter f has a set of corresponding encodings $\bar{\beta}$ in B, we have the property that $\texttt{eval}(q) = \texttt{filter}(\texttt{eval}(\text{“}q_1 \textbf{ join } q_2\text{”}), \bigoplus \bar{\beta})$ and then we can further expand the expression into $\texttt{eval}(q) = \texttt{filter}(\texttt{filter}(T_1 \times T_2, \bigoplus \bar{\rho}'_1 \oplus \bigoplus \bar{\rho}'_2), \bigoplus \bar{\beta}) = \texttt{filter}(T_1 \times T_2, \bigoplus \bar{\rho}'_1 \oplus \bigoplus \bar{\rho}'_2 \oplus \bigoplus \bar{\beta})$. As a result, the property also holds for this case.

With all three cases proved, the lemma is proved. □

Challenges in Problem 2. Although Problem 2 is a simpler search problem comparing to the original problem, directly enumerating all these pairs remains challenging, since 1) the number of all $(\bar{\beta}, \gamma)$ pairs we need to visit in the search space is large, and 2) the computation we need for checking the whether a pair $(\bar{\beta}, \gamma)$ is a candidate relatively expensive (as we need to evaluate all pairs on the table T_1 and compare the result against T_{out}). These challenges exist mainly because T_{out} is not effectively used in the search process. In the next section, we introduce how T_{out} can be effectively used to further reduce the complexity of the search problem.

7. Backward Inference

To overcome the challenges in Problem 2, our key insight is to build a simpler checking condition in a backward manner, i.e., using the output example, substitute the original one to enable more efficient pruning and checking. Concretely, given a summary table ρ, the backward inference algorithm contains two main components:

- For each summary table ρ, we build the trace σ (as will be shown later) between $\texttt{coreTable}(\rho)$ and the output example T_{out} to capture how $\texttt{coreTable}(\rho)$ can be filtered and projected to generate T_{out}.
- From each trace σ, we infer a set $P = \{\langle \beta, \gamma \rangle, ...\}$ such that a pair $(\bar{\beta}, \gamma)$ that satisfies the original checking condition if and only if $\langle \bigoplus \bar{\beta}, \gamma \rangle \in P$. And we use this condition to further reduce the original search problem.

7.1 Data Structure: Trace

Trace is the data structure built between two tables T_1 and T_2, and it compactly represents all possible ways to obtain T_2 from T_1 with `filter` and `proj` operators defined in Figure 4.

The definition of trace in presented is Figure 10: a trace σ built from two tables T_1 and T_2 is a set of mappings between cell coordinates in T_2 and cell coordinates in T_1. Each mapping $\omega \mapsto \Omega$ maps a cell ω in T_2 to a set of cells $\{\omega'_1, ..., \omega'_m\}$ in T_1, where $\omega = (i, j)$ represent the coordinate of a cell in T_2 (i for its row index and j for its column index) and $\omega'_1, ..., \omega'_m$ are cells in T_1 such that values in these cells are equal to the value in ω.

7.2 Inference Algorithm

The algorithm to build a mapping instance M from two tables T_1 and T_2 is presented in Figure 11. The algorithm iterates over all cells in T_2, and for each cell ω, it searchs all cells $\omega'_1, ..., \omega'_k$ in T_1 such that their values are equal to the value in ω and add a mapping $\omega \mapsto \{\omega'_1, ..., \omega'_k\}$ into the mapping instance.

7.3 Problem Reduction

The key operation on a trace σ is $\texttt{fullInstantiate}(\sigma)$: the operator extract a set of filter-projector pairs[f] P from σ, where each pair $\langle \beta, \gamma \rangle = \langle [b_1...b_n], [i_1, ..., i_m] \rangle \in P$ contains a bv-filter and a projector on T_1, and it satisfies the equation $\texttt{proj}(\texttt{filter}(T_1, \beta), \gamma) = T_2$. Furthermore, the set $P = \texttt{fullInstantiate}(\sigma)$ is *complete* if it contains all $\langle \beta, \gamma \rangle$ pairs satisfying the equation above, as presented below in Lemma 2.

Lemma 2. *(Complete Representation of Mapping Instances) Given two tables T_1 and T_2, suppose M is the mapping instance built from T_1 and T_2 using the backward inference algorithm defined in Figure 11, β and γ are two bitvectors such that $|\beta| = \texttt{rowNum}(T_1)$, $|\gamma| = \texttt{colNum}(T_1)$, then $\texttt{proj}(\texttt{filter}(T_1, \beta), \gamma) = T_2$ if and only if $\langle \beta, \gamma \rangle \in \texttt{fullInstantiate}(M)$.*

Proof. The proof of the lemma contains the following two parts:

- “⇒”: Suppose $\beta = [i_1, ..., i_n]$ and $\gamma = [j_1, ..., j_m]$ is an bv-filter and projector pair such that ,$proj(\texttt{filter}(T_1, \beta), \gamma) = T_2$. By definition of `proj` and `filter`, we know that for each $k \in [1, ..., n], l \in [1, ..., m], \texttt{val}((i_k, j_l), T_1) = \texttt{val}((k, l), T_2)$. Thus, for all pairs (i_k, j_l), $(i_k, j_l) \in \texttt{Img}(\sigma_{T_1,T_2}((i, j)))$ and $\beta\ \gamma$ can be derived from the backward inference algorithm.
- “⇐”: Suppose $\beta = [i_1, ..., i_n]$ and $\gamma = [j_1, ..., j_m]$ are generated from backward inference algorithm. By definition, for each $k \in [1, ..., n], l \in [1, ..., m]$, $\texttt{val}((i_k, j_l), T_1) = \texttt{val}((k, l), T_2)$, and thus filtering T_1 using β and projecting the result with γ results in T_2.

With both directions of “if” and “only if” proved, the lemma is proved. □

Problem 3. Given a summary table ρ, and the trace σ inferred between $\texttt{coreTable}(\rho)$ and T_{out}, Problem 2 can be further reduced to the problem of finding all $(\bar{\beta}, \gamma)$ pairs such that:

$$\bar{\beta} \subseteq \texttt{bvSet}(\rho),$$
$$\gamma \in \texttt{projSet}(\rho),$$
$$\langle \bigoplus \bar{\beta}, \gamma \rangle \in \texttt{fullInstantiate}(\sigma).$$

Challenges in problem 3. Although the second reduction can efficiently help guide the search process of $(\bar{\beta}, \gamma)$ in a space summary ρ, directly instantiating all $\langle \beta, \gamma \rangle$ from ρ from σ when the core table of the ρ is the Cartesian product of two (or more) tables (e.g., ρ_3 in Figure 2(c)) can be prohibitively expensive, as the set $P = \texttt{fullInstantiate}(\sigma)$ can contain an extremely large number of elements in some cases.

As an illustration, consider the following table fraction formed by the Cartesian product of an input example with a table T' with r rows. (the input and output table are from Example 1 of Section 2)

	...	...	...	...
r rows	Peter	456	3	...
	...	...	...	...
	Peter	456	3	...
	Paul	456	7	...
r rows	...	...	...	...
	Paul	456	7	...
	...	...	...	...

The problems stems from the Cartesian product operation. Originally, the target filter-projector set P_1 built between T_1 and the output example (T_{out}) contains exactly 1 element (as

[f] We use $\langle ..., ... \rangle$ to represent these pairs to enable distinguishing them against the $(\bar{\beta}, \gamma)$ pairs mentioned in the last section.

```
enumFilteringQuery(I, consts, holeNum)
// Input: input table set (I), constant set (consts), and an integer (holeNum)
// indicating the number of correlated values in exists-clauses
  S ← ∅;
  K ← enumExistsSkeleton(I, consts, holeNum);
  foreach T ∈ I:
    F ← enumFilters(schema(T), schema(T) ∪ consts);
    F ← F ∪ enumExistsFilters(schema(T), K, holeNum);
    B ← encodeFilters(F, T);
    S ← S ∪ {ST_p(T, B, T)};
  return S;

enumAggrQuery(S, aggregators, consts)
// Input: summary tables, aggregators, and constant set.
  R ← ∅;
  foreach ρ ∈ S:
    forall β̄ ⊆ bvSet(ρ):
      T ← TFilter(coreTable(ρ), ⨁ β̄);
      foreach {c_1, ..., c_k} ⊆ columns(T):
        foreach c_t ∈ columns(T), α ∈ aggregators:
          Q' ←"Select c_1, ..., c_k, α(c_t) ;
               From decode(ρ, β, [1, ..., colNum(T)]);
               Group By c_1, ..., c_k";
          T' ← eval(Q');
          F ← enumFilters({c_t}, schema(T) ∪ consts);
          B ← encodeFilters(F, T);
          R ← R ∪ {ST_p(T, B, Q')};
  return R;

enumJoinQuery(S_1, S_2, consts)
// Input: two summary table sets S_1, S_2, constant set (consts)
  R ← ∅;
  foreach ρ_1 ∈ S_1, ρ_2 ∈ S_2:
    T_1 ← BaseTable(ρ_1), T_2 ← BaseTable(ρ_2);
    F ← enumFilters(schema(T_1), schema(T_2));
    B ← encodeFilters(F, T_1 × T_2);
    R ← R ∪ {ST_c(ρ_1, ρ_2, B)};
  return R;
```

```
enumExistsSkeleton(I, consts, holeNum)
// Input: input tables, constant set, number of holes in exists-clause.
if holeNum = 0: return ∅;
  H ← ∅;
  foreach τ ∈ Types:
    H ← H ∪ {h_1 : τ, ..., h_holeNum : τ};
  return enumFilteringQuery(I, H ∪ consts, 0);

enumFilters(L, R)
// Input: two sets of values L and R and a boolean
  F ← ∅;
  foreach v_1 : τ_1 ∈ L, v_2 : τ_2 ∈ R;
      if τ_1 ≠ τ_2 : continue;
      foreach op ∈ lookUpOperators(τ_1):
        F ← F ∪ {v_1 op v_2};
  return F;

enumExistsFilters(V, K)
// Input: table T, a set of constants consts, and a boolean
  F ← ∅
  foreach κ ∈ K :
    [h_1 : τ_1, ..., h_n : τ_n] ← holes(κ);
    forall [v_1 : τ'_1, ..., v_n : τ'_n] ⊆ V:
      if exists τ_i ≠ τ'_i(i ∈ [1, n]):
        continue;
      Q ← [h_1 ↦ v_1, ..., h_n ↦ v_n]κ;
      F ← F ∪ {"Exists Q"};
  return F;

encodeFilters(F, T)
// Input: a set of filters on T, and a table T
  B ← ∅, n ← rowNum(T);
  foreach f ∈ F:
    β ← [b_1...b_n] where b_i = eval(f, r_i)_{i∈[1,n]};
    B ← B ∪ {β};
  return B;
```

Figure 9. Modules in forward enumeration algorithm, where we use function $\texttt{eval}(Q)$ to refer to evaluating a query Q using SynthSQL interpreter, $\texttt{eval}(f, r)$ to refer to evaluating a filter on a table row with SQL interpreter, and $\texttt{decode}(\rho, \beta, \gamma)$ is the function to decode bv-filter β and projector γ on a summary table ρ to a query, as will be shown in Figure 14.

$$\sigma_{T_1,T_2} ::= \{\omega \mapsto \Omega \mid \omega \in \texttt{coord}(T_2), \Omega = \{\omega' \in \texttt{coord}(T_1) \mid \texttt{val}(T_1, \omega') = \texttt{val}(T_2, \omega)\}\}$$

$$\texttt{fullInstantiate}(\sigma) ::= \{(\beta, \gamma) \mid \texttt{bitvecToList}(\beta) = [i_1, ..., i_{n_2}],\ \gamma = [j_1, ..., j_{m_2}],\ \forall x \in [i_1, ..., i_{n_2}], y \in [j_1, ..., j_{m_2}].(i_x, j_y) \in \sigma(x, y)\}$$

$$\texttt{partialInstantiate}(\sigma, split) ::= \{(\beta, \gamma) \mid \texttt{bitvecToList}(\beta) = [i_1, ..., i_{n_2}], \gamma = [j_1, ..., j_{m_2}], \forall x \in [i_1, ..., i_{n_2}], y \in [j_1, ..., j_{m_2}].(i_x, j_y) \in \sigma(x, y), \exists\, k.(j_k > split)\ and\ \exists\, k.(j_k \leq split)\}$$

$$\texttt{bitvecToList}([b_1...b_n]) ::= [i \mid b_i = 1]$$

Figure 10. The definition of trace.

```
backwardInference(T_1, T_2)
  M ← ∅
  foreach i = 1 to rowNum(T_2):
    foreach j = 1 to colNum(T_2):
      ω ← (i, j);
      Ω ← {ω' | ω' in T_1 ∧ val(ω, T_2) = val(ω', T_1)};
      M ← M ∪ {ω ↦ Ω};
  return M;
```

Figure 11. Inferring a mapping instance from T_1 to T_2.

$P_1 = \{\langle [00110], [1, 2, 3] \rangle\}$). However, after Cartesian product, each row in input will have r duplicates in first 3 columns, and the number of filter-projector pairs between $T_1 \times T'$ and T_{out} is increased to $r^{r_{out}}$ (or $2^{r_{out} * r}$ if we consider projection with '`Select Distinct`', r_{out} is the row number of the output table).

As we will show in the next section, instead of directly instantiating all $\langle \beta, \gamma \rangle$ pairs using $\texttt{fullInstantiate}(\sigma)$, the merge algorithm can use ρ to guide instantiate process of σ to avoid the problem listed above.

8. Merge Algorithm

As shown in Figure 6, the merge algorithm plays a crucial role in the synthesis pipeline: after obtaining a summary table ρ and a trace σ built from $\texttt{coreTable}(\rho)$ and the output example T_{out}, the merge algorithm then heuristically searches for all $(\bar{\beta}, \gamma)$ pairs on ρ with guide of σ.

Before presenting the merge algorithm, we first introduce several extra operations on both summary tables and mapping instances that are used in defining the merge algorithm.

Auxiliary Functions The two auxiliary functions defined on a summary table ρ are presented in Figure 12, including 1) pushing down a bv-filter in a compound summary table to a bv-filter in its left summary table, 2) heuristically enumerating $(\bar{\beta}, \gamma)$ in ρ pairs with the guidance of a target filter-projector pair set P.

```
1  pushDownL(β, T1, T2)
2    β' ← [ ];
3    for i = 1 to rowNum(T1):
4      b_i ← 0;
5      forall j = 1 to rowNum(T2):
6        b_i ← b_i ∨ β[i * rowNum(T2) + j];
7      β' ← β ++ [b];
8    return β;
```

```
1  guidedEnumFilterCombST(ρ, target)
2    if ρ = ST_p(T, B, Q):
3      Combs ← guidedCombineST(B, target, maxLen);
4      return {β̄ | β̄ ∈ Combs, (⨁ β̄) ∈ target};
5    if ρ = ST_c(ρ1, ρ2, B):
6      T1 ← baseTable(ρ1);
7      T2 ← baseTable(ρ2);
8      R1 ← {β̄1 | β̄ ⊆ bvSet(ρ1), β̄1 = pullUpL(β̄, T1, T2),
                ∃β0 ∈ target.((⨁ β̄1) ⊕ β0 = β0)};
9      R2 ← {β̄1 | β̄ ⊆ bvSet(ρ2), β̄1 = pullUpR(β̄, T1, T2),
                ∃β0 ∈ target.((⨁ β̄1) ⊕ β0 = β0)};
10     Combs ← guidedCombineST(B, target, maxLen);
11     R ← {β̄1 ++ β̄2 ++ β̄ | β̄1 ∈ R1, β̄2 ∈ R2, β̄ ∈ Combs,;
         ∃β0 ∈ target.((⨁ β̄1) ⊕ (⨁ β̄2) ⊕ (⨁ β̄) ⊕ β0 = β0)};
12     return R;
```

```
1  guidedCombineST(B, target, k)
2    if k = 1 :
3      return {[β] | β ∈ B and ∃β0 ∈ target.(β ⊕ β0 = β0)};
4    Combs ← guidedCombineST(B, target, k − 1);
5    R ← {[β1] ++ β̄ | β1 ∈ B, β̄ ∈ Combs,
                ∃β0 ∈ target.(β1 ⊕ β0 = β0)};
6    return R;
```

Figure 12. Extra operators on summary tables.

The first function $\texttt{pushDownL}(\beta, T_1, T_2)$ can be treated as the inversion of the function $\texttt{pullUpL}(\beta, T_1, T_2)$. The input of the pushDown function include a bv-filter β (of size $\texttt{rowNum}(T_1) * \texttt{rowNum}(T_2)$) and two tables T_1, T_2. The output of the function is a bv-filter β' (of size $\texttt{rowNum}(T_1)$) on T_1, satisfying the property that $\texttt{deDup}(\texttt{proj}(\texttt{filter}(T_1 \times T_2, \beta), \gamma)) = \texttt{deDup}(\texttt{filter}(T_1, \beta))$, where γ is a bitvector representing projection on all fields of T_1 and deDup represents the function that removes all duplicated rows in a table.

The second operator is $\texttt{guidedEnumFilterCombST}(\rho, P)$: it takes in a summary table ρ and a set of target bv-filter set B, and searches for all bv-filter lists $\bar{\beta}$ in ρ, such that $\bigoplus \bar{\beta} \in B$.

The key idea of the function is to prune all combinations $\bar{\beta}$ such that $\forall \beta_0 \in B.((\bigoplus \bar{\beta}) \oplus \beta_0 \neq \beta_0)$ (i.e. there exists some bit in β_0 that is 1 but the corresponding bit in $\bigoplus \bar{\beta}$ is 0) during the search process: since $\forall \beta_0 \in B.((\bigoplus \bar{\beta}) \oplus \beta_0 \neq \beta_0)$ indicates that we cannot find other filters to combine with $\bar{\beta}$ such that the combination result is in B, and we can safely prune out such combinations.

8.1 Merge Algorithm

With the auxiliary functions above, we now define the merge algorithm. As shown in Figure 13, the merge algorithm takes in a summary table ρ and the trace σ inferred between $\texttt{coreTable}(\rho)$ and T_{out}, and returns a set of pairs of form $(\bar{\beta}, \gamma)$, where the first element is a list of bv-filters on ρ and each β in the list is a primitive filter from $\texttt{bvSet}(\rho)$, while the second element is a projector on $\texttt{coreTable}(\rho)$. Particularly, each pair $(\bar{\beta}, \gamma)$ in the result set has the property: $\texttt{deDup}(\texttt{proj}(\texttt{filter}(\texttt{coreTable}(\rho), \bigoplus \bar{\beta}), \gamma)) = \texttt{deDup}(T_{out})$, and these pairs are passed to the next stage (decoding) to obtain all SynthSQL queries that are consistent with input-output examples within the subspace of ρ.

The merge algorithm takes the following steps to obtain the result. When the summary table is a primitive summary table, all $\langle \beta, \gamma \rangle$ pairs are directly instantiated from the program trace σ (line 3), and are directly used to guide searching candidate filter lists in ρ (lines 4-5) to obtain all candidate $(\bar{\beta}, \gamma)$ pairs in ρ. When the summary table is a compound summary table of form $\texttt{ST}_c(\rho_1, \rho_2, B)$, the merge algorithm uses two steps to overcome the challenges listed in problem 2:

- *Step 1.* Instantiates only $\langle \beta, \gamma \rangle$ pairs from σ such that γ involves columns from both T_1 and T_2, using the function $\texttt{partialInstantiate}(\sigma, bound)$ with $bound$ set to $\texttt{colNum}(T_1)$. With this set P instantiated from σ, we use P to guide search filter combinations in ρ as in primitive summary tables (lines 10-11). This step instantiates all $(\bar{\beta}, \gamma)$ pairs such that γ involves columns from both sub summary tables and $\texttt{coreTable}(\rho)$ can be filtered with $\bigoplus \bar{\beta}$ and projected with γ to obtain T_{out} (line 12).
- *Step 2.* In the second step, our goal is to find all $(\bar{\beta}, \gamma)$ pairs in ρ such that γ involves only columns in the core table of the left summary table (we name these pairs type-2 pairs on ρ) to ensure completeness of the search.

  Since directly instantiating all $\langle \beta, \gamma \rangle$ pairs whose γ involves only columns of the left summary table is prohibitively expensive, instead of directly searching them in ρ, we construct a new summary table ρ' such that each type-2 $(\bar{\beta}, \gamma)$ pairs in ρ can be mapped to a $(\bar{\beta}, \gamma)$ pair in ρ', and we can achieve the goal of the search by searching over all candidate $(\bar{\beta}, \gamma)$ pair in ρ' and mapping back to type-2 pairs in ρ.

  This encoding and decoding process consists of the following steps: 1) first, we enumerate combination of filters that do not use filters in ρ_1 (lines 14,15), 2) then we push down these filters to filters on T_1 using pushDownL function, (we also use the mapping for decoding purpose) (line 16), and build ρ_1' by extending ρ_1 with these pushed down filters, and 3) finally, we recursively call the merge function on ρ_1' to enumerate filter combinations (which contains much less pairs comparing to those in ρ) (lines 18-20) and use the map_1, map_2 to map each $(\bar{\beta}, \gamma)$ pair found in ρ_1' to pairs in ρ using mapBack (line 22).

  The correctness of the algorithm (in terms of completeness) is based on the property that there exists a map between type-2 $(\bar{\beta}, \gamma)$ pairs in ρ and $(\bar{\beta}, \gamma)$ pairs in ρ_1'. As a result, the result in line 22 contains complete $(\bar{\beta}, \gamma)$ pairs in ρ that can transform $\texttt{coreTable}(\rho)$ to T_{out}.

After invoking the merge function between each pair of summary table and trace, we obtain all $(\bar{\beta}, \gamma)$ on the summary tables that can generate T_{out} within the search space. As shown in Figure 6, the next step is to decode and rank the queries to recommend to the user.

Lemma 3. *(Soundness of type-2 operations) Given the two summary tables $\rho = \texttt{ST}_c(\rho_1, \rho_2, B)$ and ρ_1' in the merge algorithm, if a filter-projector pair $(\bar{\beta}, \gamma)$ is a candidate pair in ρ whose γ involves only columns in $\texttt{coreTable}(\rho_1)$, then there exists a pair $(\bar{\beta}', \gamma')$ in ρ_1', satisfying the property below and $(\bar{\beta}, \gamma)$ is in the coding result set of $(\bar{\beta}', \gamma')$.*

$$\texttt{deDup}(T_{out}) = \texttt{deDup}(\texttt{proj}(\texttt{filter}(\texttt{coreTable}(\rho_1'), \bigoplus \bar{\beta}'), \gamma))$$

Proof. For each $\beta \in \bar{\beta}$, if β is a filter generated from pulling up filters in ρ_1 ($\beta = \texttt{pullUpL}(\beta_0, T_1, T_2)$), where β_0 is a filter from ρ_1 and T_1 is the core table of ρ_1, we have $\beta_0 \in \texttt{bvSet}(\rho_1)$. With this relations, we can decompose $\bar{beta}$ into $\bar{\beta}_a$ and $\bar{\beta}_b$ where the former are those satisfying the above descriptions and the latter are the rest.

```
1  merge(ρ, σ)
   // Input: summary table, mapping instance M built from ρ and T_out.
2    if ρ = ST_p(T, B, Q) :
3      P ← fullInstantiate(σ)
4      C ← guidedEnumFilterCombST(ρ, {β | ∃ (β, γ) ∈ P})
5      return {(β̄, γ) | β̄ ∈ C and (⨁ β̄, γ) ∈ P}
6    if ρ = ST_c(ρ1, ρ2, B) :
7      T1 ← coreTable(ρ1)
8      T2 ← coreTable(ρ2)
9      // only instantiate pairs whose projector γ involves columns in both sub
       summary table.
10     P ← partialInstantiate(σ, colNum(T1))
11     C ← guidedEnumFilterCombST(ρ, {β | ∃ (β, γ) ∈ P})
12     result ← {(β̄, γ) | β̄ ∈ C and (⨁ β̄, γ) ∈ P}
13     // Build a new summary table ρ'1 by pushing down filters from bvSet(ρ2)
       and bvSet(ρ) to ρ1, in a way that by searching ρ'1, we can decode the search
       result to obtain all candidate (β̄, γ) pairs in ρ whose γ involves only columns
       in ρ1.
14     R2 ← {[pullUpR(βi, T1, T2) | βi ∈ β̄] | β̄ ⊆ bvSet(ρ2)}
15     Combs ← {β̄ | β̄ ⊆ B, |β̄| = maxLen)}
16     map1 ← {β' ↦ β̄2 ++ β̄ | β̄2 ∈ R2, β̄ ∈ Combs,
                 β' = pushDownL(⨁ β̄2 ⊕ ⨁ β̄, T1, T2)}
17     map2 ← {β ↦ [β'] | β ∈ bvSet(ρ1),
                 β' = promoteL(β, T1, T2)}
18     ρ'1 ← extendPrimBV(ρ1, dom(map1))
19     σ'1 ← backwardInference(coreTable(ρ'1, T_out))
20     R'1 ← merge(ρ'1, σ'1)
21     result ← result ∪ {(β̄', γ) | (β̄, γ) ∈ R'1,
                 β̄' ∈ mapBack(β̄, map1 ∪ map2),
                 proj(filter(T, ⨁ β̄), γ) = T_out}
22     return result

1  extendPrimBV(ρ, B')
2    if ρ = ST_p(T, B, Q):
3      return ST_p(T, B ∪ B', Q)
4    if ρ = ST_c(ρ1, ρ2, B):
5      return ST_c(ρ1, ρ2, B ∪ B')

1  mapBack([β1, ..., βn], map)
2    return {β̄'1 ++ ... ++ β̄'n | ∀i = 1, ..., n.(∃ βi ↦ β̄'i ∈ map)}
```

Figure 13. Merge Algorithm.

According to the merge algorithm, we have a $\beta_x \in \texttt{dom}(map_1)$ such that filtering T_1 with β_x is equal to projection the result of filtering $T_1 \times T_2$ with $\bigoplus \bar{\beta}_b$ by including only columns in T_1.

Thus, let $\bar{\beta}' = \bar{\beta}_a +\!\!+ \{\beta_x\}$, the formula is then satisfied. □

Lemma 4. *(Soundness of Merge) Given a summary tables $\rho = \texttt{ST}_c(\rho_1, \rho_2, B)$ and the trace σ inferred between $\texttt{coreTable}(\rho)$ and T_{out}, suppose a filter-projector pair $(\bar{\beta}, \gamma)$ satisfies the property that $\langle \bigoplus \bar{\beta}, \gamma \rangle \in \texttt{fullInstantiate}(\sigma)$, then $(\bar{\beta}, \gamma)$ is contained in the result of $\texttt{merge}(\rho, \gamma)$.*

Proof. This is obvious given Lemma 3: for projections that involves both columns from ρ_1 and ρ_2, filter-projector pairs are directly included in the result, and for pairs that involes only columns from ρ_1, according to Lemma 3, they can be discovered by R' in the algorithm and mapping them back into the compound summary table makes the set complete. □

9. Decoding and Ranking

Decoding For each summary table ρ after the merge algorithm, we obtain the set of all $(\bar{\beta}, \gamma)$ pairs from the merge algorithm such that $\texttt{proj}(\texttt{filter}(\texttt{coreTable}(\rho), \bigoplus \bar{\beta}), \gamma) = T_{out}$. Thus, by decoding each $(\bar{\beta}, \gamma)$ pairs on ρ into queries, we obtain the set of all candidate queries within the query space defined by ρ, whose evaluation results are all T_{out}.

The decoding algorithm is presented in Figure 14: for a primitive summary table ρ (lines 3-9), we re-enumerate primitive filters

```
1  decode(ρ, β̄, γ)
2    c̄ ← [ci | ci ∈ schema(coreTable(ρ)) and i ∈ γ]
3    if ρ = ST_p(T, B, Q):
4      n ← |β̄|
5      F1, ..., Fn ← ∅
6      foreach i = 1, ..., n:
7        Fi ← {f | eval(f, T) = βi}
8        //βi is the i-th element bv-filter in β̄
9      return {q | q ="Select c̄
                   From Q Where f1 And...And fn",
                 fi ∈ Fi(i = 1...n) }
10   if ρ = ST_c(ρ1, ρ2, B):
11     // Decoding two sub summary tables into queries.
12     β̄1 ← [β | ∃β' ∈ β̄.(β ∈ bvSet(ρ1) ∧ β' = pullUpL(β))]
13     β̄2 ← [β | ∃β' ∈ β̄.(β ∈ bvSet(ρ2) ∧ β' = pullUpR(β))]
14     S1 ← decode(ρ1, (β̄1, ∗))
15     S2 ← decode(ρ2, (β̄2, ∗))
16     β̄ ← [β | β ∈ β̄ ∧ β ∈ B]
17     n ← |β̄|
18     // Decoding join predicates.
19     F1, ..., Fn ← ∅
20     foreach i = 1, ..., n:
21       Fi ← {f | eval(f, T) = βi}
22       //βi is the i-th element bv-filter in β̄
23     return {q | q ="Select c̄ From q1 Join q2
                   On f1 And ... And fn",
                 ∀ i = 1...n.(fi ∈ Fi ∧ q1 ∈ S1 ∧ q2 ∈ S2)}
```

Figure 14. Decoding Algorithm, where we use "$*$" to refer to the projector "`Select` $*$".

on the $\texttt{coreTable}(\rho)$ to decode each bv-filter to a primitive filter predicate on T, and further obtain a set of candidate queries by filling the holes of the query skeleton with these primitive filter predicates (line 9). For compound summary tables, the decoding process is similar: we first recursively invoke decoding algorithm on each sub summary tables (lines 12-15), then re-enumerate primitive join predicates to decode each bv-vector to a primitive join predicate[g] (lines 16-21), and the result is a set of join queries generated by filling the holes with the decoded sub summary tables and join predicates, as presented in line 23.

Lemma 5. *(Completeness of the synthesis algorithm) Suppose Q is a query expressible in SynthSQL using the given input examples and constraints, such that Q is consistent with the input output examples, i.e., $Q(I) = T_{out}$, then after decoding phase in the synthesis algorithm, Q is in the candidates set in Figure 6.*

Proof sketch. This lemma is the immediate result from combining all of the previous lemmas.

Ranking We use a heuristic ranking strategy to rank the queries generated from the decoding algorithm before presenting the synthesis result to the user, by providing a score for each query based on the following three strategies.

- *Simplicity*: Queries with simpler structural and simpler queries are scored higher.
- *Coverage*: Queries with better coverage on the provided constraints (constants and aggregators) are awarded higher score.
- *Naturalness*: More natural join predicates are awarded with higher score, e.g., join on same key are awarded with higher score.

[g] In our implementation, each bv-filter is stored with its metadata (i.e., from which summary table was the bv-filter constructed), and thus bv-filters with same value but in different summary tables can be distinguished in lines 12,13,16.

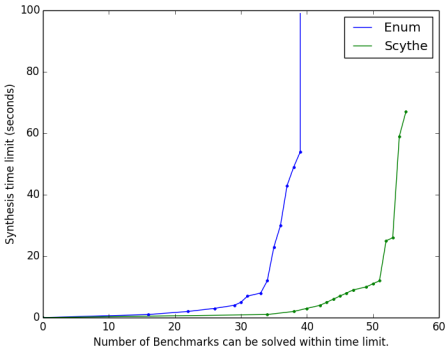

Figure 15. The number of benchmarks can be solved with the increasing time limit specified by y-axis.

With these three ranking rules, each query in the candidate query list obtains a score, and top rated queries are suggested to the user as the synthesis result.

User Interaction When none of the recommended queries are correct (i.e., although these queries are consistent with the input-output examples, they return wrong answer when a user evaluates them on larger input tables), the user can either 1) reject the answer and ask the system to continue searching for more structurally complex queries (as shown in Figure 6), 2) provide a new input-output example from scratch to the system to restart synthesis or 3) modify the existing input-output tables (add, delete some rows in the input tables or modify some cells in the original table and revise the output example correspondingly) to rerun the synthesis process.

As we will show in the evaluation section, with our basic ranking algorithm and interaction model, users can typically obtain a correct query among the top 3 ranked query with at most 2 rounds of interaction, and our algorithm is very practical in practice for solving end-user querying problem.

10. Evaluation

To evaluate the effectiveness of our approach in practice, we have implemented our algorithm as a prototype PBE system named Scythe[h] and evaluated it on a set of 60 benchmarks collected from online help forums. Our evaluation aims at answering the following four questions.

- *Q1*: Does the hypothesis that users do provide constants and aggregators when asking questions hold widely in reality?
- *Q2*: Is SynthSQL grammar expressive enough to capture many interesting real-world use case?
- *Q3*: What is the performance of Scythe comparing against existing algorithms?
- *Q4*: Is the ranking algorithm and interaction model effective enough for finding correct users among query candidates?

Evaluation Setup Besides implementating Scythe, we have also implemented a PBE system Enum that uses enumerative search approach with equivalent-class partitioning [26] as the core algorithm for performance comparison purpose. Both systems have the same target language SynthSQL, same ranking algorithm and interaction model. Scythe and Enum are implemented in Java and are evaluated on a laptop with Intel Core i5 2.7GHz CPU, 8GB of RAM, and the JVM maximum heap size is set to 4GB.

[h] Our project and evaluation benchmarks are open-sourced and its website has been redacted due to the double-blinded review process.

Benchmarks Our benchmarks consist of 60 questions collected from Stack Overflow, under "sql" tags[i]. We choose Stack Overflow posts as our evaluation source since 1) these posts are real-world questions, 2) accepted expert answers are included in the posts, so that we can use them to check the correctness of our synthesized queries, and 3) problems in Stack Overflow are more diverse, benefiting from the none-duplicate policy in Stack Overflow. We collect these questions by looking for highly-voted (with more than 50 upvotes) and recent (asked within 6 months) posts that contain input-output examples. Particularly, we exclude posts where 1) input-output examples are incorrect or incomplete, 2) questions are downvoted by forum experts (scored less than -5) due to vagueness and 3) questions that are targeting for system specific functions (e.g., '`FOR XML PATH`' in SQLServer).

From these benchmarks, 47 out of 60 questions involve using subqueries: 36 questions require using subqueries in the join-clause, 11 requires using subqueries in exists-clause. Comparing to SqlSynthesizer [37] that does not support subqueries (and is only able to express 13 out of 60 cases), our system captures a wider range of queries in practice. The number of input tables provides in these posts varies between 1-4, the row sizes of the tables vary between 3-12 and the columns sizes vary between 2-9.

Answer to Q1: All but two of the question posts contain constants or aggregators that are used in solution query appearing in the posts. This validates our assumption that we can obtain constants and aggregators from end users. In these two cases, the users indirectly provide these the aggregator '`count`' and constant '1' using the English word 'duplicates' or 'top' (we manually add the constants and aggregators for these two benchmark for performance evaluation).

Answer to Q2: Among these 60 benchmarks, SynthSQL grammar is able to expressive correct answers of 55 benchmarks. The five inexpressible cases requires using advanced operators in SQL that cannot be desugared to SynthSQL grammar, including arithmetic expression (2/5), partitioning (1/5), disjunctive filter predicates (1/5) and pivoting (1/5). Although imperfect, SynthSQL grammar is able to capture many real world cases including argmax, moving average, duplicate processing etc.

Performance We ran the 55 benchmarks that are expressible by SynthSQL on both Scythe and Enum. Figure 15 shows the number of benchmarks that can be solved by each algorithm (x-axis) with increasing time limit (y-axis). As shown in the figure, Scythe can solve all 55 benchmarks within 90 (and 51/55 within 15 seconds) seconds while Enum can totally only solve 38 of them and it is not able to solve more benchmarks even after increased time limits. Furthermore, we computed the average speedup of Scythe comparing to Enum on the 38 benchmarks that both algorithms can solve, and Scythe has an average speedup of 17.1× (with minimum 1.12× and maximum 422×).

User Interaction We evaluated the effectiveness of our heuristic-based ranking algorithm based on how many rounds does the user need to interact with the database system before a correct answer (comparing against the forum expert answer) is among top 3 of the candidate list. Among 55 benchmarks, 47 benchmarks can be solved without providing new input-output examples, and the rest 8 of them cannot be solved with only one input-output example pair due to underfit. Particularly, these 8 benchmarks can be solved by modifying the original input-output examples (i.e. modify the examples so that new examples can better specify the original problem), we measure the difficulty of this interaction by calculating the number of rows needed to be modified (including addition a new row, deleting an existing row, and modifying values of cells in

[i] http://stackoverflow.com/questions/tagged/sql

the row) in the input-output examples. For these 8 examples, the average number of rows needed to be modified is 4.6, which validates that our interaction model remains very effective for end-users.

11. Related Work

Program Synthesis Algorithms Inductive program synthesis algorithms [3; 1; 11] can be roughly classified into the following five categories: 1) enumeration algorithms [26; 25; 9], 2) constraint-solver aided synthesis [34; 33], 3) type-directed synthesis [24; 8], 4) version space algebra based inference algorithms [17; 27; 12], and 5) stochastic search [28].

Among them, our synthesis algorithm is most closely related to the enumeration approach and version space algebra synthesis algorithms. The former technique [26; 9] typically enumerates programs in a given program space specified by language syntax, and memorize enumerated results as values to avoid re-visiting visited search subspace. The latter [27; 18] uses version space algebra to efficiently maintain programs inferred from output examples using inverse semantics of the language.

Our algorithm differs from these two approaches as we use abstract data structures in both forward enumeration and backward inference phases, and we designed a merge algorithm to efficiently instantiate programs from these abstract data structures to obtain queries that are consistent with input-output examples. The abstract representation and efficient merge algorithm play a key role in addressing the challenge brought by the structure-rich nature of the SQL language, which cannot be solved efficiently by existing synthesis algorithm.

Programming by Examples Our system is inspired by Programming by Examples (PBE) applications where users specify a program using input-output examples, and the system synthesizes a program that is consistent with the examples. Different PBE systems have been developed to solve problems in data transformation [12; 30; 31; 14], data extraction [18], map-reduce program synthesis [32], data structure transformation [36] and also SQL query synthesis [37; 35]. Particularly, the two previous programming by examples systems for SQL queries developed by Zhang et al. [37] and Tran et al. [35] restrict SQL syntax by disallowing subqueries due to algorithm efficiency issues. With our new synthesis algorithm, we are able to synthesize a richer set of SQL queries that are used in practice while maintaining synthesis efficiency.

SQL Query Usability Besides PBE systems, different approaches [5; 6; 21; 2] have been proposed to increase the usability of the SQL language. One technique to increase SQL usability is by designing graphics query interfaces [21; 2; 7; 15; 20]. These graphical interfaces are often good a certain domains benefiting from a set of predefined operators to support commonly used queries in a the domain. Our system are not domain-specific and are more general in increasing SQL usability for end-users. Another line of prior work is supporting natural language interface for database systems (NLIDB) [20; 13], where users specify a query with natural language sentences, and the system translates the sentences into a SQL query. NLIDB systems are good at generating structural complex lookup queries from well formed descriptions but less satisfying in analytical queries in which their descriptions are difficult to form using natural utterances.

Interactive Refinement Different approaches has been proposed to increase the ranking algorithm and user interaction interface to make PBE system easier to locate correct programs from the candidate set, including FlashProg [22], NaLIR [19] and BlinkFill [29]. Our system can be potentially integrated with these techniques to enhance the quality of the synthesized programs.

12. Conclusion

In this paper, we presented a new synthesis algorithm for synthesizing structurally rich SQL queries and built a practical PBE system Scythe based on the algorithm. Our algorithm solves the challenges in synthesizing SQL queries by using abstract representations to step-wisely reduce the original problem of searching SQL queries to simpler problem of searching bitvectors and integer lists. We have evaluated our system on 60 real-world benchmarks collected from online Q&A forums and the result indicates that our system is very practical and being able to solve most problems within seconds.

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek 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] Sonia Bergamaschi, Francesco Guerra, Matteo Interlandi, Raquel Trillo-Lado, and Yannis Velegrakis. Quest: a keyword search system for relational data based on semantic and machine learning techniques. *Proceedings of the VLDB Endowment*, 6(12):1222–1225, 2013.

[3] Rastislav Bodík and Barbara Jobstmann. Algorithmic program synthesis: introduction. *STTT*, 15(5-6):397–411, 2013.

[4] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[5] Alvin Cheung and Armando Solar-Lezama. Computer-assisted query formulation. *Foundations and Trends in Programming Languages*, 3(1):1–94, 2016.

[6] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14, 2013.

[7] Jonathan Danaparamita and Wolfgang Gatterbauer. Queryviz: helping users understand sql queries and their patterns. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 558–561. ACM, 2011.

[8] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve 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.

[9] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663. ACM, 2014.

[10] Richard A Ganski and Harry KT Wong. Optimization of nested sql queries revisited. In *ACM SIGMOD Record*, volume 16, pages 23–33. ACM, 1987.

[11] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24. ACM, 2010.

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

[13] Sumit Gulwani and Mark Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 803–814. ACM, 2014.

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

[15] Sean Kandel, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and Paolo Buono. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization Journal*, 10(4):271–288, 2011.

[16] Dileep Kini and Sumit 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.

[17] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

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

[19] Fei Li and Hosagrahar V Jagadish. Nalir: an interactive natural language interface for querying relational databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 709–712. ACM, 2014.

[20] Fei Li and HV Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[21] Edward Lu and Ras Bodik. Quicksilver: Automatic synthesis of relational queries. Master's thesis, EECS Department, University of California, Berkeley, May 2013.

[22] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit 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.

[23] Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. Formal semantics of sql queries. *ACM Transactions on Database Systems (TODS)*, 16(3):513–534, 1991.

[24] Peter-Michael Osera and Steve 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.

[25] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *ACM SIGPLAN Notices*, volume 49, pages 396–407. ACM, 2014.

[26] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar 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.

[27] Oleksandr Polozov and Sumit 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.

[28] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGPLAN Notices*, 48(4):305–316, 2013.

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

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

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

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

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

[34] Emina Torlak and Rastislav 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.

[35] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 535–548. ACM, 2009.

[36] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat 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.

[37] Sai Zhang and Yuyin 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.