Automatically Synthesizing SQL Queries from Input-Output Examples

Sai Zhang Yuyin Sun
Computer Science & Engineering
University of Washington, USA
{szhang, sunyuyin}@washington.edu

***Abstract*—In the age of big data, a large number of computer end-users, such as research scientists and business analysts, need to frequently query a database, yet lack the programming knowledge to do such tasks smoothly. In this paper, we present a *programming by example* technique that permits end-users to automate such querying tasks. Our technique takes from users an input and output example of how the database should be queried, and synthesizes a SQL query that reproduces the example output from the example input. Later, when the synthesized SQL query is applied to another, potentially larger, database with a similar schema as the example input, the synthesized SQL query produces a corresponding result that is similar to the example output. Our technique has several notable features: it only needs small input-output examples to infer a desirable SQL query, it is fully automated and does not require users to provide annotations/hints of any form, and it can rank multiple possible solutions to provide to users the most likely result.**

Our technique has been implemented as an open-source programming tool. In our preliminary evaluation, our prototype tool has synthesized correct answers for 5 out of 6 SQL exercises from a classic database textbook, and has been used to solve 5 non-trivial problems raised by real-world users from popular online forums, including ones that receive no human replies.

I. Introduction

The big data revolution over the past few years has resulted in significant advances in digitization of massive amounts of data and accessibility of computational devices to massive proportions of the population. A perennial challenge faced by many enterprise nowadays is the management of their increasingly large and complex databases, which can contain hundreds and even thousands of tables.

A. End-users' Difficulties in Using SQL

Although the relational database management system (RDBMS) and the de facto language (SQL) are perfectly adequate for many end-users' needs [16], the costs associated with deployment and use of database software and SQL are prohibitive.

The problem is exacerbated by the fact that many end-users have myriad diverse backgrounds including research scientists, business analysts, commodity traders, human resource managers, finance professionals, and marketing managers. Increasingly, those end-users who need to query databases to analyze the data are not professional programmers, but are experts in some other domains. They need to ask a variety of questions on their data and use the answer to support their business decisions. For example, as pointed out by [10], conventional RDBMS software remains underused in the long tail of science: the large number of users, such as the research scientists who are in relatively small labs and individual researchers, have limited IT funding, staff and infrastructure yet collectively produce the bulk of scientific knowledge.

For a large number of end-users, they learn and use SQL queries in the following typical ways: they browse textbook or online resources to learn basic idioms of SQL. They try to write experimental SQL queries and execute them on sample databases to explore the data itself. They modify the written queries to derive new queries by adding or removing snippets: predicates in the `where` clause, tables in the `from` clause, or columns in the `select` clauses. However, such practice is inefficient and time-consuming. A key challenge is that many end-users can clearly understand their goals but simply can not write a correct SQL query for their tasks, either due to the syntax complexity of the language itself, or the structure complexity of the underlying databases, or others. For many end-users, they need an *intelligent* tool to assist them to perform database query tasks. A highly accessible tool that end-users can describe their needs, and use to connect their intentions to executable SQL queries would be best.

B. Existing Solutions

Graphical User Interface (GUI) and *general programming languages* are two state-of-the-art approaches in helping end-users perform database queries. However, both of them are far from satisfactory.

Many RDBMS come with a well-designed GUI with tons of features. However, a GUI often does not permit users to personalize a database's functionality for user-specific tasks. On the other hand, as a GUI supports more and more customization features, users may struggle to discover those features, which can significantly degrade its usability.

General programming languages, such as SQL, Java (with JDBC), or other domain specific languages, serve as a fully expressive medium for communicating a user's intention to a database. However, learning a practical programming language (even a simplified, high-level domain specific language) often requires a substantial amount of time and energy that a typical end-user would not prefer, and should not be expected, to invest.

C. Synthesizing SQL Queries from Examples

After carefully studying how end-users were describing their encountered SQL query problems on online help forums, we observed that one of the most common ways for end-users to express their intents is using input-output examples. Although input-output examples may lead to underspecification, they serve as a straightforward way of describing *what* the task is and a natural interface that a tool can provide assistance.

In this work, we present a technique for synthesizing SQL queries from input-output examples. In particular, our technique takes example input table(s) and output table from the end-users, and then automatically infers a SQL query (or multiple queries, if exist) that queries the input database and returns the output example. If the inferred SQL query is applied to the example input, then it produces the example output, and if the SQL query is applied to other similar inputs (potentially much larger tables), then the SQL query produces a corresponding output.

Our technique follows a general methodology for designing systems supporting programming by examples [15]. It includes the following three major steps: (1) identify a language subset (of SQL) that is expressive to describe a large proportion of database queries; (2) design an algorithm that can efficiently infer programs in the language from input-output examples; and (3) develop a ranking strategy that presents the most likely program to the users, when multiple solutions exist.

Supported SQL Subset. The full SQL 93 specification contains over 1000 pages, and it is infeasible and unnecessary to support all language features in practice. When selecting a supported SQL subset, there is a tradeoff between the expressiveness of the selected language subset, and the complexity of finding simple consistent solutions within that language's search space. In general, the more expressive a search space, the harder the task of finding consistent hypotheses within that search space.

During our study of end-user's real problems, we also carefully studied what would be the solution to users' desired query, and found that the most common desired queries can be composed by a handful of basic SQL constructs, which form the target language of this work (Section II).

As shown in our experiments (Section IV), the supported subset, though far from completion, is expressive enough to describe many query tasks succinctly, while at the same time concise enough to be amenable for efficient inference.

SQL Query Inference Algorithm. Our inference algorithm, as described in Section III, consists of two major steps. The first step, called *SQL Skeleton Creation*, scans the example input and output, and *guesses* what a desirable SQL query may look like. The output of this step is a set of *incomplete* SQL query skeletons. A SQL query skeleton is an incomplete SQL query, which contains unknown parts. The second step, called *SQL Query Completion* takes as inputs a created SQL skeleton, fills unknown parts with possible solutions, and then produces a set of syntactically-valid SQL queries. In particular, when performing SQL query completion, our technique uses a rule learning algorithm to infer the conditions (Section III-C1), and uses type-directed search to infer possible aggregates in a query (Section III-C2).

Query Candidate Ranking Strategy. Input-output examples provided by end-users are likely to be under-specified. It is entirely possible that multiple queries can be synthesized to satisfy the provided examples. We address this problem by developing a ranking scheme that ranks the possibly multiple queries consistent with the given input-output examples. Our ranking scheme, detailed in Section III-D is inspired by the Occam's razor principle, prefers a smaller and simpler solution if other aspects being equal.

D. Technique Evaluation

The goal of this work is developing a practical SQL query synthesis technique that is capable of helping end-users writing a wide range of SQL queries. The technique aims to replace the role of the forum expert, which not only removes human from the loop, but also enables end-users to solve their problems in a few seconds as opposed to a few days (while waiting for an expert's reply).

Thus, we implemented our technique in an open-source tool and evaluated it in two ways. First, we picked up 6 SQL exercises from a classic database textbook [24]. SQL exercises from a database textbook often cover the most widely-used SQL features that a course instructor wishes students to master. Those exercises can be served as golden test to evaluate the expressiveness of our supported SQL subset. Second, we collected 5 non-trivial SQL problems raised by real end-users on online help forums, and tested whether our technique can effectively synthesize desirable SQL queries.

As a result, our technique successfully synthesized 5 out of 6 textbook exercises and solved all 5 forum problems, within a very small amount of time (1 minute per exercise/problem, on average).

E. Contributions

This paper makes the following contributions:

- **Problem.** To the best of our knowledge, we are the first to address the SQL query synthesis problem from examples across multiple tables.
- **Technique.** We show to automatically synthesize SQL queries from input-output examples (Section III).
- **Tool.** A practical tool that implements the proposed technique (available at: http://sqlsynthesizer.googlecode.com).
- **Evaluation.** An empirical evaluation of the tool implementation on a number of SQL exercises from a classic textbook [24] and SQL problems raised by real users on online forums (Section IV).

II. Language and Example

In this section, we first present the supported SQL subset, and then describe a motivating example which could be expressed in the supported SQL subset.

$\langle query \rangle ::=$ SELECT $\langle expr \rangle^+$ FROM $\langle table \rangle^+$
WHERE $\langle cond \rangle^+$
GROUP BY $\langle column \rangle^+$ HAVING $\langle cond \rangle^+$

$\langle table \rangle ::= atom$

$\langle column \rangle ::= \langle table \rangle.atom$

$\langle cond \rangle ::= \langle cond \rangle$ && $\langle cond \rangle$
| $\langle cond \rangle$ || $\langle cond \rangle$
| ($\langle cond \rangle$)
| $\langle cexpr \rangle \langle op \rangle \langle cexpr \rangle$

$\langle op \rangle ::=$ = | > | <

$\langle cexpr \rangle ::= const \mid \langle column \rangle$

$\langle expr \rangle ::= \langle cexpr \rangle \mid count(\langle column \rangle)$
$\mid sum(\langle column \rangle) \mid max(\langle column \rangle) \mid min(\langle column \rangle)$

Fig. 1. Syntax of the supported SQL subset.

Student_key	Student_name	Level
0001	Adam	senior
0002	Bob	junior
0003	Erin	senior
0004	Rob	junior
0005	Dan	senior
0006	Peter	senior
0007	Sai	senior

Fig. 2. An example input table 'student' for the SQL question described in Section II-B.

A. Supported SQL Subset

Figure 1 defines the syntax of the supported language, which is a subset of the standard SQL 93 language. This language subset supports common query operations across multiple tables (i.e., `select ... from ... where..`), and conjunction of predicates. The language subset shares the same semantics with the standard SQL language. Differing from language subsets used in the existing work [5], our language subset supporting joining operations across multiple tables, and includes widely-used database operations such as `group by`, `order by`, and `having`, as well as a few common aggregation functions such as `count`, `sum`, `max`, and `min`.

As we will show in Section IV, this subset, though far from completion, is expressive enough for answering many evaluated SQL exercises from a textbook and some real problems from online forums.

B. Motivating Example

Consider the following SQL question picked up from a classic database textbook [24]: *given a `student` table (Figure 2) and an `enrolled` table (Figure 3), find out the name and max score of the students whose level is senior and enrolled in more than 3 courses*.

The question's description is quite simple. For a novice user, although they have a clear intention of what the query should do, the answer (Figure 5) may not be that straightforward.

Despite the possible difficult in writing a correct SQL query, a user could still easily draw two input tables (Figure 2 and Figure 3) and one output table (Figure 4) that fulfill the SQL question.

Student_key	Course_key	Score
0001	001	4
0001	002	2
0002	001	3
0002	002	2
0002	003	3
0003	002	1
0004	001	4
0004	003	4
0005	002	5
0005	003	2
0005	004	1
0006	002	4
0006	004	5
0007	001	2
0007	003	3
0007	004	4

Fig. 3. An example input table 'enrolled' for the SQL question described in Section II-B.

Student_name	Max_score
Dan	5
Sai	5

Fig. 4. An example output table for the SQL question described in Section II-B.

In the `student` table, column `Student_key` with `String` type serves as the primary key. In the `enrolled` table, both columns `Student_key` and `Course_key` are two foreign keys, and column `Score` with `Integer` type keeps students' scores on their enrolled courses.

In the output table, the first column `Student_name` comes from table *student*, and the second column `Max_score` is a aggregation attribute.

Having the input and output examples, our technique successfully synthesizes the desirable SQL query as shown in Figure 5.

III. Technique

We next describe our SQL query synthesis technique in detail. How to infer the desirable query for the examples shown in Section II-B will be discussed thoroughly in this section.

A. Overview

Figure 6 sketches the general workflow of our technique. At a high level, our technique consists of three major steps: Query Skeleton Creation(Section III-B), SQL Query Completion (Section III-C), and Query Candidate Ranking (Section III-D). Specifically, our approach takes from users input-output examples. It first infers a partially-complete query skeleton. The inferred query skeleton, though contains incomplete parts that can not be yet decided, serves as a good reference for further synthesizing a complete SQL query. After that, our technique uses two techniques to complete a SQL skeleton. In

```
select student.Student_name, max(enrolled.Score)
from student, enrolled
where student.Student_key = enrolled.Student_key
      and student.Level = 'senior'
group by student.Student_key
having count(enrolled.Course_key) > 3
```

Fig. 5. A SQL query to solve the SQL question described in Section II-B. When applied to the input tables in Figure 2 and Figure 3, it produces the result table in Figure 4.

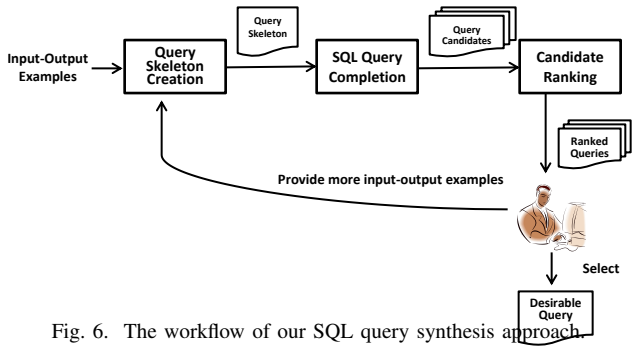

Fig. 6. The workflow of our SQL query synthesis approach.

particular, it uses an advanced rule learning algorithm from the machine learning community to infer query conditions (Section III-C1), and employs type-directed search to figure out possible aggregation expressions (Section III-C2). The SQL Query Completion step produces a list of syntactically-valid query candidates that satisfy the provided input-output examples. However, it is possible that multiple query candidates can be synthesized based on the provided input-output example. To deal with that, our technique ranks all generated candidates, and provides users a ranked list of SQL queries with the simplest ones on the top.

End-users can use our approach to obtain SQL query to transform multiple, huge database tables by constructing small, representative input and output example tables. On some examples, we speculate that our approach may produce a SQL query that satisfies the input and output examples given by the user, but does not address the intention that the user wants. To address this issue, we adapt a simple interaction model from [15] to ask users to investigate the results of an output SQL query and report any discrepancy. In this case, the user can refine the inferred SQL query by providing a more informative input-output example (or multiple input-output examples that together describe the required behavior) that demonstrate the behavior on which the originally-inferred SQL query behaves incorrectly.

B. Query Skeleton Creation

In this step, our technique scans the provided input and output examples, guesses what a target SQL query might look like, and infers a set of query skeletons that capture the basic query structures.

A query skeleton is an incomplete SQL query, which captures three basic query structures that could often be decided after a simple scan over the examples: tables used in a result SQL query, table columns used to join input tables, and table columns for projecting the query results. Other parts in a SQL query such as conditions (including selection and having conditions), and aggregations are left as unkown, and will be determinated in the next step (Section III-C).

Step 1: Determining the Table Set. In practice, end-users are unwilling to provide more than enough inputs. Therefore, every input table specified in the example is expected to be used to construct a desirable SQL query. Based on this observation, we assume that every input table should be used as least once in the query. By default, the table set T used in the result SQL query contains all given input tables. However, it is possible that a single table will be used for multiple times. Our technique does not forbid this case, rather, we adopt a single heuristic to estimate the used table set: if the same column from an input table appears more than once in the example output, we add the input table the same number of times to the used table set.

Step 2: Determining Joining Columns. Given two arbitrary tables, there exist many ways to join them. Enumerating all possible joining conditions may introduce a huge number of joining conditions and would quickly become intractable. We observe that, in practice, two tables are often joined via the following three cases: (1) tables are joined on their primary keys with the same (or compatible) data types, such as joining a *student* table with an *enrolled* table with the *student_id* column. It does not make any sense to join two tables on a Integer column and a String column; (2) tables are joined using columns with the same name, such as joining a *student* table with a *enrolled* table on the *student_name* column; and (3) two columns that have the data type, and have a large portion of overlapped values in their corresponding input tables can be used as a joining condition. It is straightforward to check the first two cases to identify possible joining columns. For the third case, our technique scans the given input tables to check "value similarity" between two arbitrary columns, and selects columns whose "value similarity" is above a fixed threshold as joining columns.

Step 3: Determining Output Columns. To identify output table columns on which the querying result would be projected, our technique checks whether each output table column name appears in any input tables. If so, we used the matched column from the input table as the output column. Otherwise, the output column must be produced by using aggregation. Our technique keeps track of those aggregation columns and search for proper aggregates in the next phase (Section III-C2).

In summary, this step infers three parts as a query skeleton: tables used in constructing a SQL query, joining conditions to connect the input tables, and a list of columns to project the output results.

1) Example: For our motivating example, from the output table (Figure 4), our technique identifies that column `Student_name` comes from table `student` and column `Max_Score` is a new one which must be created by using aggregation.

The created query skeleton is shown in Figure 7. As we can see from Figure 7, there are three unknown structures represented by <Aggregation> or <Conditions> in red color, which will be filled in the next phase.

```
select    Student.Student_name, <Aggregation>
from      student, enrolled
where     student.Student_key = enrolled.Student_key
          and  <Conditions>
group by  Student.Student_name
having    <Conditions>
```

Fig. 7. The SQL skeleton created for the motivating example in Section II-B.

C. SQL Query Completion

The SQL skeleton produced by the first step, though incomplete, serves as a good reference in inferring complete and valid SQL queries. In this step, our technique the remaining incomplete parts: conditions and aggregates, by rule-based learning and type-directed search, respectively.

1) Learning Conditions: The problem of learning query *conditions* can be cast as finding appropriate *rules* that can perfectly divide the whole searching space into positive part and negative part. In our context, the searching space contains all tuples generated by joining the input tables, the positive part are all tuples in the output table, and the negative part are the rest tuples.

Techniques to efficiently find rules from data has been studied extensively by many machine learning researchers [4], [9], [22]. Generally, in the machine learning community, there are major two paradigms of rule learning:

1. Decision-tree learning [22]. Given the positive and negative examples, one can generate a decision tree, and then extract decision-tree-splitting conditions from the root to all positive leaf. The extracted conditions can be viewed as the selection criteria that isolates the output data from the input searching space.

2."Divide-and-conquer" strategy [29]. Unlike decision-tree learning, instead of learning a full tree, this methodology repeatedly determines the most powerful rules for the dataset that can maximally separate the positive examples from the negative ones, until no more positive examples are available.

However, for our problem, both methodologies can not be directly applied. Learning rules via decision-tree learning is restricted to conjunctive rules, which is insufficient for many real scenarios. On the other hand, the generalization ability of the "divide-and-conquer" strategy is quite limited due to overpruning and covering heuristic.

To overcome these two limitations, in our technique, we adapt the PART learning algorithm [9], which combines both rule learning paradigms above. Notably, PART is capable of giving an more accurate and expressive rule set. Specifically, it utilizes the "divide-and-conquer" strategy in that it repeatedly builds rules and removes the instances it covers until no examples are left. When creating each rule, it employs a pruned decision tree built from current set of instance and only makes the leaf with the largest coverage into the resulting rules, without keeping the whole learned tree in memory.

Although PART is a promising algorithm for learning rules from data, there are two challenges when applying it to our problem:

group by	aggregation
C_1	COUNT(C_2), MAX(C_3), MIN(C_3), AVG(C_3)
C_2	COUNT(C_1), MAX(C_3), MIN(C_3), AVG(C_3)

TABLE I
THE GENERATED AGGREGATION FEATURES FOR A TABLE WITH 3 COLUMNS: C_1, C_2, AND C_3, IN WHICH COLUMNS C_1 AND C_2 ARE STRING TYPE AND COLUMN C_3 IS INTEGER TYPE.

predicate	comparison result
$C_1 = C_2$	0
$C_1 < C_2$	1
$C_1 > C_2$	0
$C_3 = C_4$	1
$C_3 < C_4$	0
$C_3 > C_4$	0

TABLE II
THE GENERATED COMPARISON FEATURES FOR A TABLE WITH 4 COLUMNS: C_1, C_2, C_3, AND C_4, IN WHICH COLUMNS C_1 AND C_2 ARE STRING TYPE, AND COLUMNS C_3 AND C_4 ARE INTEGER TYPE. COLUMNS WITH THE SAME TYPE ARE COMPARABLE, SUCH AS C_1 AND C_2, AND C_3 AND C_4.

Challenge 1. How to represent tuples. In PART, an example data point is represented by a single feature vector. Therefore, we must transform tuples into appropriate feature representation. To do so, a straightforward way is simply using concrete values in a tuple as a feature vector. However, doing so loses much useful structure information needed in a SQL query.

We encode existing domain knowledge about SQL query as additional features, such as, (1) Aggregation, including `COUNT`, `MAX`, `MIN` and `AVG`, whose results might be used in query condition; and (2) Comparison results between two comparable columns. The above two additional knowledge encoding permits our technique to make use of correlations between columns, rather than only values from each isolated and sequential columns.

We add two types of new attributes into feature representation of tuples.

1) ***Aggregation Features***. Aggregation features are the aggregation results grouped by each `String` type column over every other columns. Table I shows an example.
2) ***Comparison Features***. Comparison feature is the result of comparing two comparable columns. We consider two possible values: $\{1,0\}$, which represents means whether two columns under comparison satisfy the predicate or not, respectively. Table II shows an example.

Combining using concrete tuple values, aggregation features, and comparison features, our technique is able to extract expressive feature representation for tuples, and permits users to encode domain knowledge and structural information about a SQL query.

Challenge 2. How to use the learnt rules to complete a SQL query. PART may return rules including three types of features. For example, it returns the following rules for our motivating example in Section II-B:

```
COUNT(enrolled.Course_key) > 2      && student.level
='senior'
```

We need to split the returned rule into two parts: the query selection condition, and the having condition. To achieve this,

we use P_o to denote predicates related to original features directly derived by the concrete tuple values, P_a to denote predicates related to aggregation features and P_c to denote predicates related to comparison features. For predicate in P_o, we use it to fill condition holes in select clause by comparing selected column with a constant value. For predicate in P_a, we can use it to fill aggregation holes in having clause.

For our motivating example, $P_o = \{$`student.level ='senior'`$\}$, $P_a = \{$`COUNT(enrolled.Course_key) > 2`$\}$.

After that, our technique adds a having condition such as "`having COUNT(enrolled.Course_key)`" to the SQL query skeleton, and use predicates in P_c to fill the selection condition.

2) Searching for Aggregates: The last step in completing a SQL query is searching for aggregates as the projection columns. Our technique uses a type-directed searching strategy to do this. The whole searching space includes all possible combinations of table columns and the five supported aggregates (see Figure 1). In type-directed search, we leverage the following information to prune the potential space:

1) The output values' type in the result column must be compatible with the aggregate's return type. For instance, if an output column is String type, it must not use aggregates that always return an Integer type, such as `count` and `sum`.
2) When using arithmetic aggregates such as `max` and `min`, the values in the output column must have appeared in the input table.

In our experience, the type-directed searching strategy significantly reduces the searching space and makes our tool find the desirable aggregates faster.

D. Query Candidate Ranking

After the above two steps, SQL queries satisfying the given input-output examples will be returned. To reduce the effort in inspecting the results and selecting a desirable query, we devise a strategy to put the most likely query near the top of the return list.

Our strategy is based on Occam's razor to compute a cost for each returned query, and prefers queries with lower costs. For a SQL query, each table appearing in it introduces a cost C_t and each appearing condition introduces an additional cost C_p. The total cost of a query is computed by: $n_t \cdot C_t + n_p \cdot C_p$, where n_t is the number of tables and n_p is the number of predicates used in the query. Our technique ranks queries based on their costs in an increasing order to ensure that a simpler query often ranks higher. Figure 8 shows an example to illustrate this ranking strategy.

Input table: **student**

name	**score**
Bob	4
Dan	5
Jim	2

Query output

name
Bob
Dan

Query 1:

```
select name from student where score > 3;
```

Query 2:

```
select name from student where name = `Bob` or name = `Dan`
```

Fig. 8. The illustration of our query candidate ranking strategy. Both Query 1 and Query 2 can transform the example input to the example output. However, based on our ranking strategy, Query 1 is ranked higher, since it contains less conditions and is simpler than Query 2. An inferred query containing more conditions like Query 2 is more likely to overfit the given examples.

IV. Evaluation

We implemented our technique in an open-source programming tool. Our tool employs WEKA [14] to implement the PART learning algorithm to extract rules, and uses MySQL [21] as the backend database engine to check the validity of each output query. When a SQL query is synthesized, our tool executes the synthesized query on the backend database, and checks whether the output query result matches the given example output.

We next describe the evaluation of our prototype tool.

A. Research Questions

We aim to investigate the following research questions:

- Is the supported SQL subset expressive enough to describe queries that real end-users require?
- Can our technique efficiently find a SQL query from small input-output examples?

B. SQL Query Synthesis Scenarios

To answer these two questions, we collected a set of SQL query synthesis scenarios in the following two ways. First, we picked up 6 SQL exercises from a classic database textbook [24]. Those 6 SQL exercises cover many commonly-used SQL features, which a database course instructor think would be the most useful parts, and expect her students to get familiar with. Second, we collected 5 SQL problems raised by real end-users from popular online help forums [6], [26], [28].

For a given exercise or problem, if it has already been associated with input-output examples (e.g., those provided by users from online forums), we directly apply our tool on the existing examples. Otherwise, we manually provide small concise and representative input-output examples for our tool. If for a given exercise or problem, our tool inferred a SQL query that does not behave as we expected when applied to other inputs, we will manually find an input on which the SQL query misbehaves and reapplied our tool to the new input. We repeat this process until our tool infers a desirable SQL query.

	#Total
Textbook exercises	6

TABLE III
SUBJECT PROGRAMS AND THEIR CONFIGURATION PROBLEMS.

C. Experimental Results

For each SQL synthesis scenario, we measured two results. The time cost in producing a desirable SQL query by our tool, and the time we spent in writing input-output examples. Table III summarizes our experimental results.

As shown in Table III, our technique synthesized expected SQL queries for 10 out of 11 scenarios within a very small amount of time (less than 1 minute each). The human effort spent in providing input-output examples is also very limited: it took us less than 5 minutes for one scenario.

1) A Real Example: We next describe a real SQL problem picked up from an online SQL help forum[1]. The thread was started by a novice user, who needed help to write a SQL query to get result from three input tables. In the help thread, the novice user described his required query in a few paragraphs of English, but also include several small, representative input-output examples as shown in Figure 9, to better express his intention. This thread receives no replies as of May 2012, and we speculated that writing a SQL query to join three tables to produce certain output results can be non-trivial.

We ran our tool on the input-output examples in Figure 9. The tool produced 6 valid answers in less than 1 minutes, all of which satisfy the given examples. The highest ranked SQL query is shown in Figure 10, which is quite unintuitive to write. The SQL query in Figure 10 first joins three input tables on columns `T1.Column2`, `T2.Column2`, `T2.Column1`, and `T3.Column1` using some selected columns, and then aggregates the results based on column `Table2.Column3`'s value. Finally, it returns the minimal values of columns `T1.Column1`, `T1.Column4`, and `T3.Column2` from each aggregated group as the results.

D. Experimental Conclusions

Our initial experimental results are promising. It shows that the supported SQL subset is expressive enough to describe a variety of useful database queries. It also demonstrates the usefulness of our SQL query synthesis technique. In particular, our SQL query synthesis technique only requires a small amount of human effort and small input-output examples. It infers desirable SQL queries on both database textbook exercises and online forum problems, even solving problems that receive no reply from popular online forums.

V. RELATED WORK

Our work can be differentiated from existing work in three major ways: the scope of the problem addressed, the technique used, and the evaluation methodology. We next discuss two categories of closely-related work on reverse query processing and automated program synthesis.

Add this as related work [3]

[1]http://forums.tutorialized.com/sql-basics-113/join-problem-147856.html

The input Table1:

Column1	Column2	Column3	Column4
101	2001	3020	01-01-11
101	2001	3002	02-01-11
101	2001	3001	03-01-11
102	2002	3002	01-01-11

The input Table2:

Column1	Column2	Column3
20011	2001	200131
20012	2001	200132
20013	2001	200133

The input Table3:

Column1	Column2
20011	Site
20012	Site
20013	Site

The output table:

101	200131	01-01-11	Site
101	200132	01-01-11	Site
101	200133	01-01-11	Site

Fig. 9. Input-output examples taken from an online SQL help forum thread. Our technique automatically synthesizes a SQL query as shown in Figure 10 that produces the output table from the three input tables.

```
select min(T1.Column1), T2.Column3,
       min(T1.Column4), min(T3.Column2)
from Table1 T1, Table2 T2, Table3 T3
where T1.Column2 = T2.Column2 and T2.Column1 = T3.Column1
group by T2.Column3
```

Fig. 10. The highest ranked SQL query inferred by our technique from the input-output examples in Figure 9.

A. Reverse Query Processing

In the database community, there is a rich body of work [5], [27], [30] that share the same broad principle of "reverse query processing". Zloof's work on *Query by Example* (QBE) [30] provided an intuitive form-based Graphical User Interface for writing database queries, which is completely different than ours.

The problem of *Query By Output* (QBO) [27] shares a similar goal with QBE, but takes a quite different approach. In QBO, we are given a view: V and n relations: $R_1, R_2, ..., R_n$ and asked to return a query Q such that $Q(R_1, R_2, ..., R_n) = V$. However, QBO is cast as a classification problem (into two classes: tuples in V, and tuples not in V) and decision trees are used to find a compact query. A decision tree is constructed in a greedy fashion by determining a "good" predicate to split the tuples into two classes, which form the root nodes of two decisions trees (each of which is constructed in a recursive fashion). The QBO technique presented in [27] infers select-project-join queries. Such queries can only be applied to satisfy a limited number of examples, and cannot be applied to many of the other real world cases that we studied (e.g., a query with an aggregation operator like our motivating example in Figure 9).

Recently, the *View Definitions Problem* (VDP) has been addressed by Sarma et al. [5]. VDP is restricted to find the

most succinct and accurate view definition, when the view query is restricted to a specific family of queries. VDP is the special case of QBO where $n = 1$ and there are no joins nor projections. The high-level goal of this work is similar to our approach, but there are some key differences: (1) VDP techniques infer a relation from a large representative example view, while we infer a SQL query from a set of few input-output examples (which is a critical usability aspect for end-users who lack adequate SQL experience). (2) the VDP technique in [5] does not consider join or projection operations and assumes the output view to be a strict subset of the input database with the same schema, while our work considers both join and projection operations.

In the data mining community, a slightly related problem to ours is the *redescription mining* problem [23]. At an abstract level, our work is different from these redescription mining techniques in several ways. First, the goal of redescription mining is to find different subsets of data that afford multiple descriptions across multiple vocabularies covering the same dataset, while our goal is to infer a SQL statement to obtain the desirable querying behavior. Second, we are concerned with a fixed subset of the data (i.e., the output example table). Third, none of the existing approaches for redescription mining account for structural (relational) information in the data.

Other query recommendation systems proposed in the literature may achieve a similar goal of helping end-users write better SQL queries, but they rely on information that we cannot assume access to in many realistic scenarios: a query log [18], or user history and preferences [16].

B. Automated Program Synthesis

Program synthesis is the problem of creating a program in some underlying language from specifications that can range from logical declarative specifications to examples or demonstrations [12], [15], [25]. This topic has been studied intensively in Artificial Intelligence and Programming Language communities with the goal of easing the burden of algorithm designers, software developers, or end-users. Interest in it has grown rapidly in recent years [11], and many approaches have been proposed.

Example-based program synthesis techniques [1], [2], [7], [8], [17], [19], [20] take textual input-output examples and infer programs that transform text. However, existing techniques focused on synthesizing programs for string expressions [12], [25], spreadsheet layout transformation [15], , geometry constructions [13], thus can not be directly applied to our SQL query synthesis domain, due to different abstractions, granularities, and tasks.

VI. Conclusions and Future Work

General purpose programming platforms have never been easy. Many end-users who are non-professional programmers are still mostly stucked with the process of *how* to accomplish a certain task by receiving step-by-step, detailed, and syntactically correct instructions, instead of simply describing *what* the task is. *Programming by examples* has the potential to change this landscape, when targeted for the right set of people for the right set of problems.

In this paper, we studied the problem of automated SQL query synthesis from simple input-output examples, presented a practical technique to infer desirable SQL queries that fulfill end-users' intentions, and developed a usable programming tool. To demonstrate usefulness of our technique, we applied our tool to 6 SQL exercises from a classic database textbook as well as 5 non-trivial SQL problems from online forums by real end-users. The results are promising, showing that our technique can automate a variety of database query tasks.

The source code of our tool implementation is available at: http://sqlsynthesizer.googlecode.com

Besides general issues such as performance and ease of use, our future work will concentrate on the following topics:

Enrich the supported SQL subset. The technique presented in this paper focuses on a widely-used SQL subset. It may miss some important language features such as nested queries, and thus fail to infer meaningful queries in some scenarios when such missing features must be used. We plan to remedy this problem by enriching the supported SQL subset, and designing a corresponding algorithm to synthesize more general solutions.

Illustration of the synthesis steps. Besides producing a final result, end-users may also be interested in knowing how a SQL query is inferred. Showing detailed inference steps (visually) not only makes the tool more usable, but only permits end-users to better understand the whole process and spot possible errors earlier. To do so, we plan to investigate how to apply recent advance in data visualization [17] to the context of program synthesis.

Noise detection and tolerance in users' inputs. The current technique requires users to provide noise-free input-output examples. Even in the presence of a small amount of user-input noises (e.g., a typo), the inference algorithm will declare failure when it fails to learn a valid SQL query. To overcome this limitation, we plan to design a more robust inference algorithm that can attempt to identify and tolerate user-input noises, and even suggest a fix to the noisy example.

Generalization. We have applied our synthesis technique to one specific, but important problem. There is a rich history of end-user programming problems being cast as program synthesis (e.g., [11]). In future work, we plan to formulate other tasks, such as repetitive file operations, and to apply our technique to them.

References

[1] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *Proc. VLDB Endow.*, 2(1):514–525, Aug. 2009.

[2] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 193–204, New York, NY, USA, 2010. ACM.

[3] A. Cheung, A. Solar-Lezama, and S. Madden. Inferring sql queries using program synthesis. *CoRR*, abs/1208.2013, 2012.

[4] W. W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*, ICML'95, pages 115–123. Morgan Kaufmann, 1995.

[5] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 89–103, New York, NY, USA, 2010. ACM.

[6] Database Journal. http://forums.databasejournal.com/.

[7] K. Fisher. Learnpads: Automatic tool generation from ad hoc data. In *In SIGMOD*, 2008.

[8] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 421–434, New York, NY, USA, 2008. ACM.

[9] E. Frank and I. H. Witten. Generating accurate rule sets without global optimization. In *Proceedings of the 15th International Conference on Machine Learning*, ICML'98, pages 144–151. Morgan Kaufmann, 1998.

[10] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Rec.*, 34(4):34–41, Dec. 2005.

[11] S. Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM.

[12] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[13] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 50–61, New York, NY, USA, 2011. ACM.

[14] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[15] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 317–328, New York, NY, USA, 2011. ACM.

[16] B. Howe, G. Cole, N. Khoussainova, and L. Battle. Automatic example queries for ad hoc databases. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1319–1322, New York, NY, USA, 2011. ACM.

[17] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 3363–3372, New York, NY, USA, 2011. ACM.

[18] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: context-aware autocompletion for sql. *Proc. VLDB Endow.*, 4(1):22–33, Oct. 2010.

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

[20] T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 527–534, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[21] MySQL. http://www.mysql.com.

[22] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[23] N. Ramakrishnan, D. Kumar, B. Mishra, M. Potts, and R. F. Helm. Turning cartwheels: an alternating algorithm for mining redescriptions. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 266–275, New York, NY, USA, 2004. ACM.

[24] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. Addison-Wesley (3rd Edition), 2007.

[25] R. Singh and S. Gulwani. Learning semantic string transformations from examples. In *Proceedings of the 37st International Conference on Very Large Data Bases*, VLDB '2012, New York, NY, USA, 2012. ACM.

[26] StackOverflow. http://www.stackoverflow.com.

[27] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 535–548, New York, NY, USA, 2009. ACM.

[28] Tutorialized Forums. http://forums.tutorialized.com.

[29] y Giulia Pagallo, D. Haussler, and P. Rosenbloom. Boolean feature discovery in empirical learning. *Machine Learning.*, 5(1):71–99, Mar 1990.

[30] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, pages 1–24, New York, NY, USA, 1975. ACM.