Automatically Synthesizing SQL Queries from Input-Output Examples

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

***Abstract*—Many computer end-users, such as research scientists and business analysts, need to frequently query a database, yet lack enough programming knowledge to write a correct SQL query. To alleviate this problem, we present a *programming by example* technique (and its tool implementation, called SQLSynthesizer) to help end-users automate such query tasks. SQLSynthesizer takes from users an example input and output of how the database should be queried, and then 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.**

We evaluated SQLSynthesizer on XXX SQL exercises from a classic database textbook and XXX SQL questions raised by real-world users from online forums. SQLSynthesizer synthesizes correct queries for XXX textbook exercises and XXX forum questions, and it does so with small examples.

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 key challenge faced by many enterprise or computer end-users nowadays is the management of their increasingly large and complex databases.

Motivation. Although the relational database management system (RDBMS) and the de facto language (SQL) are perfectly adequate for most end-users' needs [15], the costs associated with deployment and use of database software and SQL are prohibitive. For example, as pointed out in [9], 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 training, staff and infrastructure yet collectively produce the bulk of scientific knowledge.

The problem is exacerbated by the fact that many end-users have myriad diverse backgrounds including business analysts, commodity traders, human resource managers, finance professionals, and marketing managers. Those end-users are not professional programmers, but are experts in some other domains. They need to retrieve a variety of information from their database and use the information to support their business decisions. Although most end-users can clearly describe *what* the task is, they are often stuck with the process of *how* to write a correct database query (i.e., a SQL query) even after receiving step-by-step, detailed, and syntactically correct instructions. Thus, typical end-users often need to seek information from online help forums, or ask SQL experts. This process can be repetitive, laborious, and frustrating. To assist non-expert end-users in conducting database query tasks, a highly accessible tool that can be used to "describe" their needs and "connect" their intentions to executable SQL queries would be highly desirable.

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

Many RDBMS come with a well-designed GUI with tons of features. However, a GUI is often fixed, and does not permit users to personalize a database's functionality for their query 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 query languages, serve as a fully expressive medium for communicating a user's intention to a database. However, general purpose programming languages have never been easy for end-users who are not professional programmers. Learning a practical programming language (even a simplified, high-level domain specific language, such as MDX [20]) often requires a substantial amount of time and energy that a typical end-user would not prefer, and should not be expected, to invest.

Our Solution: Synthesizing SQL Queries from Input-Output Examples In this paper, we present a technique (and its tool implementation, called SQLSynthesizer) to automatically synthesize SQL queries from input-output examples[1]. Although input-output examples may lead to underspecification, writing them, as opposed to writing declarative specifications or imperative code of any form, is one of the most straightforward ways to describe *what* the task is and is more natural for a typical end-user to use. If the synthesized 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.

[1]The SQL queries in this paper refer to the read-only database queries, which do not modify the database content.

SQLSynthesizer is designed to be used by non-expert database end-users when they do not know how to write a correct SQL query. End-users can use SQLSynthesizer to obtain a SQL query to transform multiple, huge database tables by constructing small, representative input and output example tables. We also envision SQLSynthesizer to be useful in an online education setting (i.e., an online database course). Recently, several education initiatives such as EdX, Coursera, and Udacity are teaming up with experts to provide high quality online courses to several thousands of students worldwide. One challenge, which is not present in a traditional classroom setting, is to provide answers on questions raised by a large number of students. A tool, like SQLSynthesizer, that has the potential of answer SQL query related questions would be useful.

Inferring SQL queries from examples is challenging, primarily for two reasons. First, the standard SQL language is inherently complex; a SQL query can consist of many parts, such as joins, aggregates, the `GROUP BY` clause, and the `ORDER BY` clause. Searching for a SQL query to satisfy the given input-output examples, as proved by Sarma et al. [4], is a PSPACE-hard problem. Thus, a brute-force approach such as exhaustively enumerating all syntactically-valid SQL queries and then filtering away those do not satisfy the examples would quickly become intractable in practice. Second, a SQL query has a rich set of operations: it needs to be evaluated on *multiple* input tables; it needs to perform data grouping, selection, and ordering; and it needs to project data on certain columns as the output. All such operations must be handled properly.

To make SQL query synthesis from examples feasible in practice, our SQLSynthesizer technique focuses on a widely-used SQL subset (Section III), and uses three steps to link a user's intention to a desirable SQL query:

- **Skeleton Creation.** SQLSynthesizer scans the given input-output examples and heuristically identify the table joins and projection columns in the result query. Then, it creates an incomplete SQL query (called, query skeleton) to capture the basic structure of the result query.
- **Query Completion.** SQLSynthesizer uses a rule-learning algorithm from the machine learning community to infer a set of accurate and expressive rules, which transform the input example into the output example. Then, it searches for possible aggregates and columns in the `ORDER BY` clause; and outputs a list of syntactically-valid candidate queries.
- **Candidate Ranking.** If multiple SQL queries satisfy the given input-output examples, SQLSynthesizer employs the Occam's razor principle to rank more likely queries higher in the output.

Compared to previous approaches [3], [4], [26], [28], SQLSynthesizer has two notable features:

- **It is fully automated.** Besides an example input and output pair, SQLSynthesizer does not require users to provide annotations or hints of any form. This distinguishes our work from competing techniques such as specification-based query inference [28] and query synthesis from imperative code [3].
- **It supports a wide range of SQL queries.** Similar approaches in the literature support a small subset of the SQL language; most of them can only infer simple select-from-where queries on a single table [3], [4], [4], [26], [28]. By contrast, SQLSynthesizer significantly enriches the supported SQL subset. Besides supporting the standard select-from-where queries, SQLSynthesizer also supports many other important SQL features, such as table joins, aggregates (e.g., `MAX`, `MIN`, `SUM`, and `COUNT`), the `GROUP BY` clause, the `ORDER BY` clause, and the `HAVING` clause.

Evaluation. We evaluated SQLSynthesizer's generality and accuracy in two aspects. First, we used SQLSynthesizer to solve XXX SQL exercises from a classic database textbook [22]. Textbook exercises are good resource to evaluate SQLSynthesizer's generality, since they are often designed to cover a wide range of SQL features. Some exercises are even designed on purpose to cover some less realistic, corner cases in using SQL. Second, we evaluated SQLSynthesizer on XXX SQL query related questions collected from popular online help forums, and tested whether SQLSynthesizer can synthesize correct SQL queries for them.

As a result, SQLSynthesizer successfully synthesized queries for XXX out of XXX textbook exercises and all XXX forum problems, within a very small amount of time (XXX minute per exercise or problem, on average). SQLSynthesizer's accuracy and speed make it an attractive approach to help end-users write SQL queries.

Contributions. This paper makes the following contributions:

- **Technique.** We present a technique that automatically synthesizes SQL queries from input-output examples (Section IV).
- **Implementation.** We implemented our technique in a tool, called SQLSynthesizer (Section V). It is available at: http://sqlsynthesizer.googlecode.com.
- **Evaluation.** We applied SQLSynthesizer to XXX textbook exercises and XXX forum questions. The experimental results show that SQLSynthesizer can synthesize a wide range of SQL queries, and it does so with small examples (Section VI).

II. Illustrating Example

We use an example, described below, to illustrate the use of SQLSynthesizer. The example is taken from a classic database textbook [22] (Chapter 5, Exercise 1) and has been simplified for illustration purpose[2].

Find the name and the maximum course score of each senior student enrolled in more than 2 courses.

Despite the simplicity of the problem description, writing a correct SQL query can be non-trivial for a non-professional end-user. Although most users can clearly understand the question, they must choose the right SQL features and use them correctly.

[2]This exercise defines 2 tables: `student` and `enrolled`. The `student` table contains three columns: `student_id`, `name`, and `level`. Table `enrolled` contains three columns: `student_id`, `course_id`, and `score`.

student_id	name	level
1	Adam	senior
2	Bob	junior
3	Erin	senior
4	Rob	junior
5	Dan	senior
6	Peter	senior
7	Sai	senior

student_id	course_id	score
1	1	4
1	2	2
2	1	3
2	2	2
2	3	3
3	2	1
4	1	4
4	3	4
5	2	5
5	3	2
5	4	1
6	2	4
6	4	5
7	1	2
7	3	3
7	4	4

```
SELECT student.name, MAX(enrolled.score)
FROM student, enrolled
WHERE student.student_id = enrolled.student_id
      and student.level = 'senior'
GROUP BY student.student_id
HAVING COUNT(enrolled.course_id) > 2
```

name	max_score
Dan	5
Sai	5

(a) Two input tables: student (Left) and enrolled (Right) **(b)** A SQL query inferred by SQLSynthesizer **(c)** An output table

Fig. 1. Example input/output tables and the SQL query synthesized by SQLSynthesizer to solve the problem in Section II. In this example, users provide SQLSynthesizer with two input tables (shown in (a)) and an output table (shown in (c)). SQLSynthesizer automatically synthesizes a SQL query (shown in (b)) that transforms the two input tables into the output table.

Users can use our SQLSynthesizer technique to obtain the desirable query. As illustrated in Figure 1, to use SQLSynthesizer, an end-user only needs to provide it with some small, representative example input and output tables (Figures 1(a) and 1(c)). Then, SQLSynthesizer works in a fully-automatic, push-button way in inferring a SQL query that satisfies the given example input and output.

The SQL query, shown in Figure 1(b), first joins two tables on the common `student_id` column, and then groups the joined result by the same `student_id` column. Further, the query selects all senior students (using a query condition in the `WHERE` clause) who has been enrolled in more than 2 courses (using a condition in the `HAVING` clause). Finally, the query projects the result on the `student.name` column and uses the `MAX` aggregate to compute the maximum course score.

III. A SQL Subset Supported in SQLSynthesizer

The problem of finding SQL queries satisfying a given input-output example pair is PSPACE-hard [4]. To make the problem tractable, instead of supporting all features in the standard SQL language, SQLSynthesizer focuses on a widely-used SQL subset using which a large class of query tasks can be performed. Unfortunately, when designing the SQL subset, we found that no systematic study has ever been conducted to this end, and little empirical evidence has ever been provided on which SQL features are widely-used in practice. Without such empirical knowledge, deciding which SQL subset to support remains difficult.

To address this challenge and reduce our personal bias in language design, we first conducted an online survey to ask experienced IT professionals about the most widely-used SQL features in writing database queries (Section III-A). Then, based on the survey results, we designed a SQL subset (Section III-B). We also sent the designed SQL subset to the survey participants and conducted a series of follow-up email interviews to confirm whether our design would be sufficient in practice.

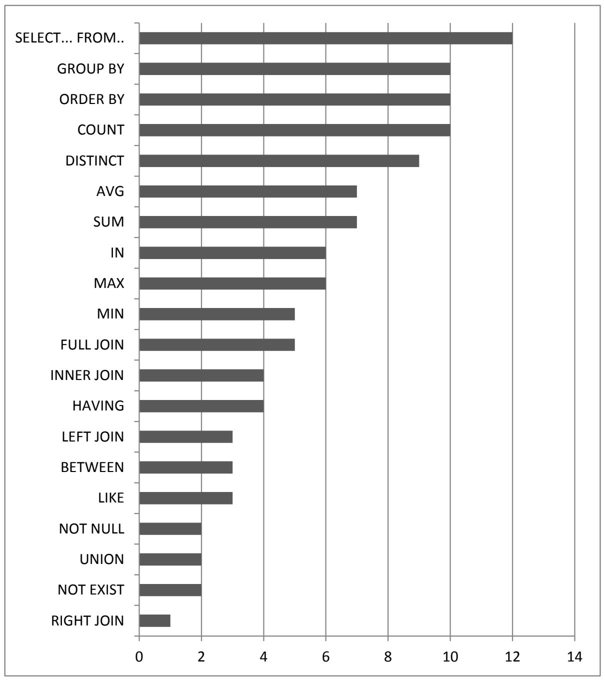


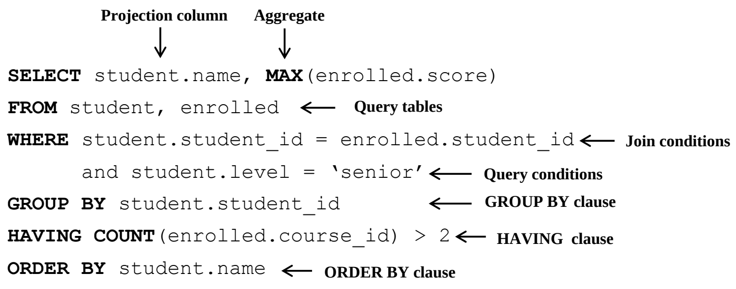
Fig. 2. Survey results of the most widely-used SQL features in writing a database query. There were 12 participants in the survey, and each participant was asked to select the top 10 widely-used SQL features. SQL features with no selection are omitted in this Figure for brevity.

A. *Online Survey: Eliciting Design Requirements*

Our online survey consists of 6 questions that can be divided into three parts. The first part includes simple demographic questions about participants. In the second part, participants were asked to select the top 10 most widely-used SQL features in their minds. Instead of directly asking participants about the SQL features, which might be vague and difficult to respond, we presented them a list of *all* standard SQL features in writing a query. Additionally, participants were asked to report their own experience in writing SQL queries in the third part of the

```
  ⟨query⟩ ::=  SELECT ⟨expr⟩+ FROM ⟨table⟩+
               WHERE ⟨cond⟩+
               GROUP BY ⟨column⟩+ HAVING ⟨cond⟩+
               ORDER BY ⟨column⟩+
  ⟨table⟩ ::=  atom
 ⟨column⟩ ::=  ⟨table⟩.atom
   ⟨cond⟩ ::=  ⟨cond⟩ && ⟨cond⟩
             | ⟨cond⟩ || ⟨cond⟩
             | ( ⟨cond⟩ )
             | ⟨cexpr⟩ ⟨op⟩ ⟨cexpr⟩
     ⟨op⟩ ::=  = | > | <
  ⟨cexpr⟩ ::=  const | ⟨column⟩
   ⟨expr⟩ ::= ⟨cexpr⟩ | COUNT(⟨column⟩) | COUNT(DISTINCT ⟨column⟩)
             | SUM(⟨column⟩) | MAX(⟨column⟩) | MIN(⟨column⟩)
```

Fig. 3. Syntax of the supported SQL subset in SQLSynthesizer: *const* is a constant value and *atom* is a string value, representing a table name or a column name.



```
SELECT student.name, MAX(enrolled.score)
FROM student, enrolled
WHERE student.student_id = enrolled.student_id
      and student.level = 'senior'
GROUP BY student.student_id
HAVING COUNT(enrolled.course_id) > 2
ORDER BY student.name
```

Fig. 4. An example query using the SQL subset defined in Figure 3.

survey.

We sent out invitation to the graduate mailing list at University of Washington, and posted our survey on professional online forums (e.g., StackOverflow). As of April 2013, we received 12 responses. On average, the respondents have 9.5 years of experience in software development (max: 15, min: 5), and 5.5 years of experience in using database (max: 10, min: 2). In addition, two participants identified themselves as database professionals.

Figure 2 summaries the survey results.

B. Language Syntax

Based on the survey results, we design a SQL subset whose syntax is shown in Figure 3. Its semantics are the same as the standard SQL language, and are omitted in this section for brevity.

The supported SQL subset is a subset of the standard SQL language. It covers all top 10 most widely-used SQL features voted by the survey participants, except for the `IN` keyword in Figure 2. In addition, the SQL subset supports the `HAVING` keyword since `HAVING` is often used together with the `GROUP BY` clause. The SQL subset, though by means complete in writing all possible queries, has significantly enriched the SQL subset supported by the existing query inference work [4], [26]. Besides supporting the standard select-from-where queries as in [4], [26], our SQL subset also supports table joins, aggregates (i.e., `COUNT`, `MAX`, `MIN`, and `AVG`), the `GROUP BY` clause, the `ORDER BY` clause, and the `HAVING` clause. For readers who are not familiar with the basic SQL idioms, we show an example query using our SQL subset in Figure 4, and annotate it with important concepts.

When designing this SQL subset, we only considered standard SQL features, while excluding user-defined functions and vendor-specific features, e.g., the `TOP` keyword supported in Microsoft SQLServer. We discarded some standard SQL features, primarily for three reasons. First, some features are designed as syntactic sugar to make a SQL query easier to write; and thus can be safely removed without affecting a language's functionality. For example, the `BETWEEN` keyword checks whether a given value is within a specific range, and can be simply replaced by two query conditions. Similarly, the `NOT NULL` keyword is also omitted. Second, some features, such as `FULL JOIN`, `LEFT JOIN`, and `RIGHT JOIN`, provide special ways to join tables, and are less likely to be used by non-expert end-users. Third, other features, such as `IN`, `UNION`, and `NOT EXIST`, are used to write sub-queries, which are the major source of the PSPACE-hardness in inferring a SQL query [4]. We exclude them in order to make the synthesis problem more tractable. The `LIKE` is also discarded, since it is used for string wildcard matching, and can also lead to **[[xx]]**

C. Follow-up Interviews: Feedback about the SQL Subset

After proposing the SQL subset in Figure 3, we performed follow-up email interviews to gain participants' feedback about it. Participants were first asked to rate the expressiveness of the SQL subset in Figure 3 in writing real-world database queries, on a 6-point scale (5-completely sufficient; 0-not sufficient at all; and in-between values indicating intermediate sufficiency), and then to provide their comments.

On average, the rating of this SQL subset is 4.5. Most of the participant rated it 5, or 4. Only one participant rated it 3, because this participant misinterpreted the designed syntax and thought it does not support table joins.

Overall, based on the feedback by experienced IT professionals, we think our SQL subset is reasonably expressive for writing database queries that most end-users need.

IV. Technique

This section first gives an overview of SQLSynthesizer's workflow and high-level algorithm in Section IV-A, and then explains SQLSynthesizer's three steps in details (Section IV-B, Section IV-C1, and Section IV-D).

A. Overview

Figure 5 illustrates SQLSynthesizer's workflow. SQLSynthesizer consists of three steps: (1) the "Skeleton Creation" step (Section IV-B) infers a set of query skeletons from the given examples; (2) the "Query Completion" step (Section IV-C) infers the missing parts in each query skeleton and outputs a list of syntactically-valid queries that satisfy the provided example input and output; and (3) the "Candidate Ranking" step (Section IV-D) ranks all synthesized SQL queries and place the

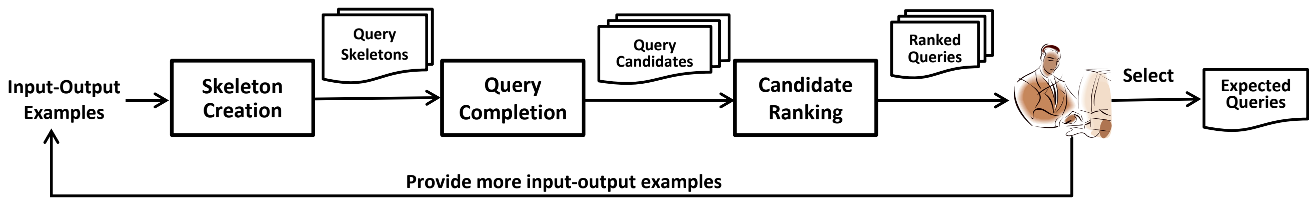


Fig. 5. Illustration of SQLSynthesizer's workflow of synthesizing SQL queries from input-output examples.

Input: example input table(s) T_I, example output table T_O
Output: a ranked list of SQL queries

```
synthesizeSQLQueries($V_{old}$, $V_{new}$, $T$)
 1: $queryList \leftarrow$ an empty list
 2: $skeletons \leftarrow$ createQuerySkeletons($T_I$, $T_O$)
 3: for each $skeleton$ in $skeletons$ do
 4:     $conds \leftarrow$ inferConditions($T_I$, $T_O$, $skeleton$)
 5:     $aggs \leftarrow$ searchForAggregates($T_I$, $T_O$, $skeleton$, $conds$)
 6:     $columns \leftarrow$ searchForOrderBys($T_O$, $skeleton$, $aggs$)
 7:     $queries \leftarrow$ buildQueries($skeleton$, $conds$, $aggs$, $columns$)
 8:     for each $query$ in $queries$ do
 9:         if isValidOnExamples($query$, $T_I$, $T_O$) then
10:             $queryList$.add($query$)
11:         end if
12:     end for
13: end for
14: rankQueries($queryList$)
15: return $queryList$
```

Fig. 6. Algorithm for synthesizing SQL queries from input-output examples.

more likely ones near the top. Users can inspect the query list and select a query from it. If SQLSynthesizer produces SQL queries that satisfy the input and output examples, but does not address the intention that the user wants; SQLSynthesizer can be used interactively by asking users to give more informative examples and then refine the inferred queries.

Using the SQL query in Figure 4 as an example, the "Skeleton Creation" step infers its project column, query tables, and join conditions; and the "Query Completion" step infers the remaining parts (i.e., aggregates, query conditions, the `GROUP BY` clause, the `HAVING` clause, and the `ORDER BY` clause).

Figure 6 sketches SQLSynthesizer's high-level algorithm. Line 2 corresponds to the first "Query Skeleton Creation" step. Lines 3 – 13 correspond to the second "Query Completion" step, in which SQLSynthesizer searches for the query conditions (line 4)[3], aggregates (line 5), and columns in the `ORDER BY` clause (line 6). SQLSynthesizer then assembles a list of candidate SQL queries (line 7), and validates their correctness on the examples (lines 8 – 11). Line 14 corresponds to the "Query Ranking" step.

B. Skeleton Creation

A query skeleton is a partially-complete SQL query used to capture the very basic structure of the result query. It consists of three parts: query tables, query join conditions, projection columns. To create it, SQLSynthesizer performs a simple scan over the examples, and uses several heuristics to determine each part.

[3]Including conditions used in the `HAVIGN` clause.

Determining query tables. A typical end-user is often unwilling to provide more than enough example input. Thus, we assume every example input table should be used (at least once) in the result query. By default, the query tables are all example input tables. Yet it is possible that one input table will be used multiple times in a query (e.g., in the case of self join). SQLSynthesizer does not forbid this case; it uses a heuristic to estimate the query tables. If the same column from an input table appears N ($N > 1$) times in the output table, it is highly likely that the table containing that column will used multiple times in the query, such as joined with different tables. Thus, SQLSynthesizer replicates the input table N times in the query table set.

Determining join conditions. There are many ways to join all query tables; enumerating all possibilities may lead to a large number of join conditions and would quickly become intractable. SQLSynthesizer uses two rules to join two tables in the most common ways in practice. For the case of multiple tables, SQLSynthesizer repeatedly joins two tables until all tables get joined. In the first rule, SQLSynthesizer seeks to join tables on columns with the same name and the same data type. For example, in Figure 1, the `student` table is joined with the `enrolled` table on the `student_id` column, which exists in both tables and has the same data type. If such columns do not exist in between two query tables, SQLSynthesizer uses the second rule to join tables on the columns with the same data type (even with different names). For example, suppose the column name `student_id` in the `student` table of Figure 1(a) is changed to `student_key`, SQLSynthesizer will no longer find a column with the same name and data type between table `student` and table `enrolled`. In this case, SQLSynthesizer will identify three possible join conditions by only considering columns with the same data type: `student_key = student_id`, `student_key = course_id`, and `student_key = score`; and then creates three skeletons, each of which uses one join condition.

Determining projection columns. SQLSynthesizer scans each column in the output table, and checks whether the column name appears in any of the input tables. If so, SQLSynthesizer uses the matched column from the input table as the projection column in the skeleton. Otherwise, the column in the output must be produced by using an aggregate. Take the output table in Figure 1 as an example, SQLSynthesizer identifies the `name` column is from the `student` table and the `max_score` column

```
SELECT      student.name, <Aggregate>
FROM        student, enrolled
WHERE       student.student_id = enrolled.student_id
            and  <Query Condition>
GROUP BY    < Column Name(s)>
HAVING      <Query Condition>
ORDER BY    <Column Name(s)>
```

Fig. 7. A query skeleton created for the motivating example in Figure 1. The missing parts (in red) will be completed in Section IV-C.

must be created by using an aggregate over some column.

Note that, if multiple columns (with the same name and data type) in different input tables has the same name as a column in the output table, SQLSynthesizer can safely uses an abitrary one; since columns with the same name and data type will be used in join conditions and are guaranteed to have the same value in the jo **[[xx]]**

As mentioned earlier, SQLSynthesizer can create multiple skeletons for a given example input-output pair. Each skeleton shares the same query tables and project columns, but differs in the join conditions. For the example in Figure 1, SQLSynthesizer creates one query skeleton shown in Figure 7.

C. Query Completion

In this step, SQLSynthesizer analyzes each created query skeleton , and completes the missing query conditions (Section IV-C1), aggregates (Section IV-C2), and the `ORDER BY` clause (Section IV-C3). SQLSynthesizer outputs a list of syntactically-correct SQL queries that satisfy the given example input and output.

1) Inferring Query Conditions: SQLSynthesizer casts the problem of *inferring query conditions* as *learning appropriate rules* that can perfectly divide a search space into a positive part and a negative part. In our context, the search space is all tuples from joining all query tables; the positive part includes all tuples in the output table; and the negative part includes the rest tuples.

The standard way for rule learning is using a decision-tree-based algorithm. However, a key challenge is how to design a good feature set. Existing approaches [26] simply use tuple values in the input table(s) as features, and limits their abilities in inferring more complex rules as query conditions. In particular, merely using tuple values as features can only infer conditions that compares a column value with a constant (e.g., `student.level = 'senior'`), but fails to infer conditions using aggregates (e.g., `COUNT(enrolled.course_id) > 2`), or conditions comparing the values of two table columns (e.g., `enrolled.course_id > enrolled.score`). Figure 9 shows an example, in which the expected query condition uses the `COUNT` aggregate.

SQLSynthesizer addresses this challenge by adding two types of additional features to each tuple. When inferring rules, SQLSynthesizer uses the existing tuple values and the newly-added features together as the feature set.

- **Aggregation Features**. For each table column, SQLSynthesizer groups all tuples by that column's value, and then applies every applicable aggregate (i.e., `COUNT`, `COUNT DISTINCT`, `MAX`, `MIN`, and `AVG` for a numeric type column; and `COUNT`, and `COUNT DISTINCT` for a string type column) to each *remaining* column and computes the aggregation result. The "Aggregation Features" part in Figure 8 shows an example.
- **Comparison Features**. For each tuple, SQLSynthesizer compares the values of every two type-comparable columns, and records the comparison results (1 or 0) as features. The "Comparison Features" part in Figure 8 shows an example.

[[The incerasing number of features, can be falsified quickly]]

Using both tuple values and the enhanced features, SQLSynthesizer employs a variant of the decision tree algorithm, called PART [8], to infer a set of rules as query conditions. Compared to the original decision tree algorithm [], PART has two notable features. First, it uses a "divide-and-conquer" strategy to repeatedly construct rules and remove tuples that have already been covered until no tuples are left, and thus is more efficient. Second, when constructing each rule, PART uses a pruned decision tree built from current set of tuples and only makes the leaf with the largest coverage into the resulting rules, without keeping the whole learned tree in memory. This permits PART to consume less memory than the original decision tree algorithm.

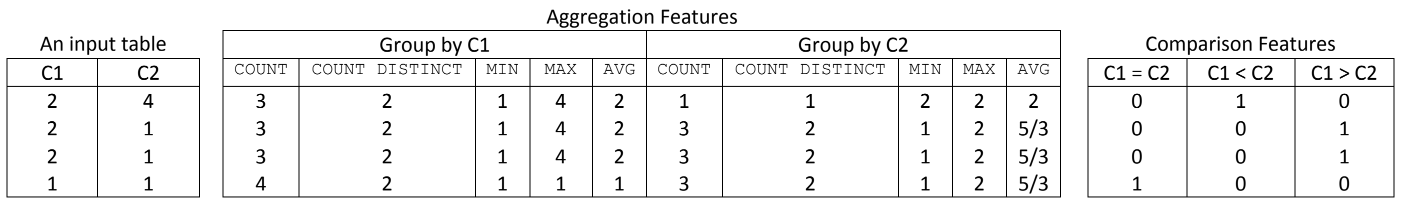
SQLSynthesizer next splits the inferred rules into two parts: it puts conditions using aggregates to the `HAVING` clause and puts other conditions to the `WHERE` clause. This is becuase according to the SQL specification, query conditions using aggregates are valid only when they are used *after* the `GROUP BY` clause. For the example conditions inferred in Figure 9, SQLSynthesizer puts `student.level ='senior'` to the query condition part in the `WHERE` clause, `COUNT(enrolled.course_id) > 2` to the query condition part in the `HAVING` clause, column `student_id` to the `GROUP BY` clause.

2) Searching for Aggregates: For each column that is produced by an aggregate, SQLSynthesizer repeatedly applies each applicable aggregate on every type-compatible table column to test its correctness. SQLSynthesizer adapts two rules to reduce the search space:

- The data values in an output column must be compatible with an aggregator's return type. For instance, if an output column contains float values, it cannot be produced by using `COUNT` or `COUNT DISTINCT`, or it cannot be produced by using the `MAX` aggregator over a column with Integer type.
- When using the `MAX` and `MIN` aggregator, each value in the output column must appear in the input table.

We found both rules are useful to speed up the search for appropriate aggregates.

3) Searching for columns in the `ORDER BY` column: SQLSynthesizer scans data values for each column of the output table. If the data values in a column are sorted, SQLSynthesizer append the column name to the `ORDER BY` clause.



An input table

C1	C2
2	4
2	1
2	1
1	1

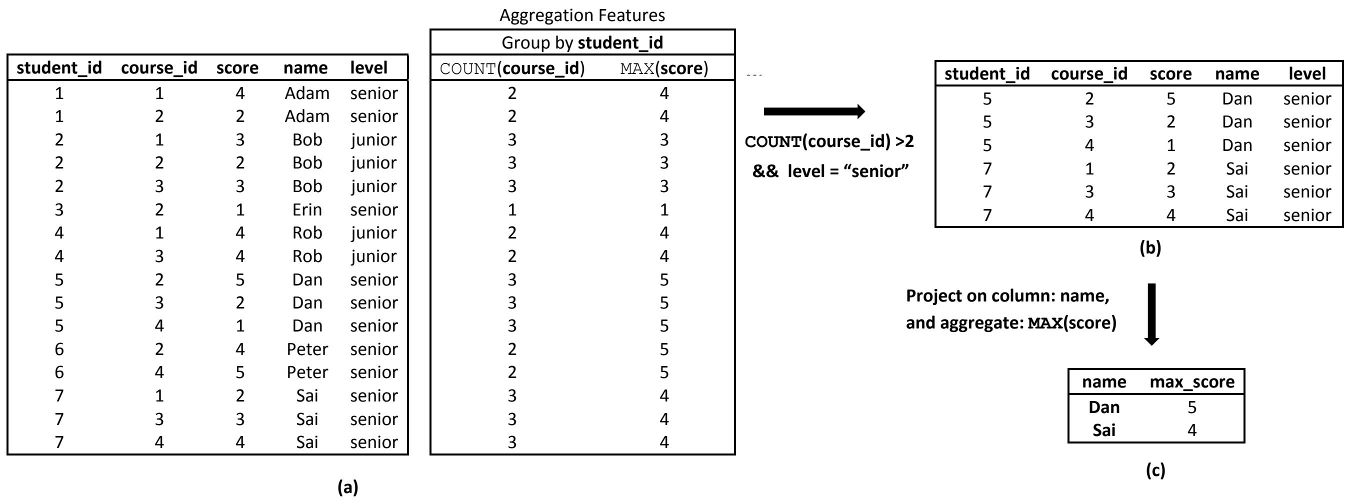
Aggregation Features

Group by C1					Group by C2				
COUNT	COUNT DISTINCT	MIN	MAX	AVG	COUNT	COUNT DISTINCT	MIN	MAX	AVG
3	2	1	4	2	1	1	2	2	2
3	2	1	4	2	3	2	1	2	5/3
3	2	1	4	2	3	2	1	2	5/3
4	2	1	1	1	3	2	1	2	5/3

Comparison Features

C1 = C2	C1 < C2	C1 > C2
0	1	0
0	0	1
0	0	1
1	0	0

Fig. 8. Illustration of the aggregation features and the comparison features enriched by SQLSynthesizer. (Left) An example input table with two columns: C1 and C2. (Center) The aggregation features enriched by SQLSynthesizer for the input table. (Right) The comparison features enriched by SQLSynthesizer for the input table.



student_id	course_id	score	name	level
1	1	4	Adam	senior
1	2	2	Adam	senior
2	1	3	Bob	junior
2	2	2	Bob	junior
2	3	3	Bob	junior
3	2	1	Erin	senior
4	1	4	Rob	junior
4	3	4	Rob	junior
5	2	5	Dan	senior
5	3	2	Dan	senior
5	4	1	Dan	senior
6	2	4	Peter	senior
6	4	5	Peter	senior
7	1	2	Sai	senior
7	3	3	Sai	senior
7	4	4	Sai	senior

Aggregation Features

Group by **student_id**	
COUNT(**course_id**)	MAX(**score**)
2	4
2	4
3	3
3	3
3	3
1	1
2	4
2	4
3	5
3	5
3	5
2	5
2	5
3	4
3	4
3	4

student_id	course_id	score	name	level
5	2	5	Dan	senior
5	3	2	Dan	senior
5	4	1	Dan	senior
7	1	2	Sai	senior
7	3	3	Sai	senior
7	4	4	Sai	senior

name	max_score
Dan	5
Sai	4

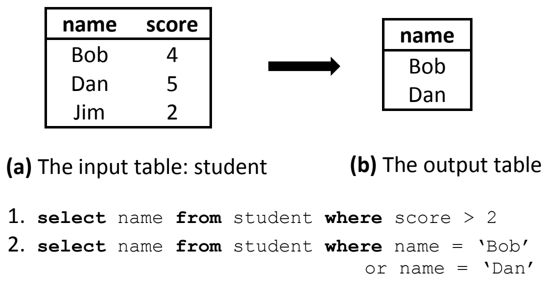
Fig. 9. Illustration of how additional features added by SQLSynthesizer helps in inferring query conditions. (a) shows SQLSynthesizer enriches the original table (Left: the result of joining table `student` with table `enrolled` on the `student_id` column) with additional features. For brevity, only relevant aggregation features are shown. Using the added aggregation features, SQLSynthesizer infers two query conditions that transform the original table into the table show in (b). Note that, without the aggregation features enhanced by SQLSynthesizer, a learning algorithm will *fail* to learn the above conditions. (c) shows the output table, which is produced by projecting the table in (b) on its column: name, and an aggregate: MAX(score).

D. Candidate Ranking

It is possible that multiple SQL queries satisfying the given input-output examples will be returned. This may adversely impact end-users who want to perform simple query tasks but now need to select the query of their intent. To alleviate this problem, we employ the Occam's razor principle, which states that the simplest explanation is usually the correct one, to rank a more likely query higher in the output list. A simpler query is less likely to overfit the given examples than a complex query, even when both of them can transform the example input to the example output.

A SQL query is simpler than another one if it uses fewer query conditions (including conditions in the `Having` and `from` clauses) or the expressions (including aggregates) in each query condition are pairwise simpler (e.g., expression `Count(student_id)` is simpler than `Count(Distinct student_id)`. Simpler query conditions suggests the extraction logics are more common and general.

In our implementation, SQLSynthesizer computes a cost for each query, and prefers queries with lower costs. The cost for a SQL query is computing approximately by summarizing the number of conditions, aggregates, and other expressions appearing in the `group by` and `order by` clauses. This heuristic, though fairly simple, has been observed to work well. Figure 10 shows an example.



name	score
Bob	4
Dan	5
Jim	2

name
Bob
Dan

(a) The input table: student **(b)** The output table

```
1. select name from student where score > 2
2. select name from student where name = 'Bob'
                               or name = 'Dan'
```

Fig. 10. Illustration of SQLSynthesizer's query candidate ranking heuristic. SQLSynthesizer produces two queries for the given input-output examples. Based on the heuristic in Section IV-D, the first query differs from the second query by using simpler conditions, and thus is ranks higher.

E. Discussion

Soundness and Completeness. The SQLSynthesizer technique is neither sound nor complete. The primary reason is that several steps (e.g., the Query Skeleton Creation step in Section IV-B) use heuristics to infer possible joins, tables used in the `from` clause, and columns in the `select` clause. Such heuristics are necessary, since they provide a good approximate solution to the problem of finding SQL queries from examples, which has been proved to be PSPACE-hard and is thus intractable in practice. Although SQLSynthesizer cannot guarantee to infer correct SQL queries for all cases, as demonstrated in Section VI, we find SQLSynthesizer is useful in synthesizing a wide variety of queries in practice.

[[no title]]

V. Implementation

We implemented the proposed technique in a tool, called SQLSynthesizer. SQLSynthesizer uses the built-in PART algorithm implementation in the Weka toolkit [13] to learn query conditions (Section IV-C). SQLSynthesizer also uses MySQL [21] as the backend database to validate the correctness of each synthesized SQL query. Specifically, SQLSynthesizer first populates the backend database with the given input tables; when a SQL query is synthesized, SQLSynthesizer executes the query on the database to observe whether the output matches the given output. Our implementation is publicly available at: http://sqlsythesizer.googlecode.com

VI. Evaluation

We evaluated four aspects of SQLSynthesizer's effectiveness, answering the following research questions:

- What is the success ratio of SQLSynthesizer in synthesizing SQL queries? (Section VI-C1).
- How long does it take for SQLSynthesizer to synthesize a SQL query (Section VI-C2).
- How much human effort is needed to write sufficient input-output examples for SQL synthesis (Section VI-C3).
- How does SQLSynthesizer's effectiveness compare to existing SQL query inference techniques (Section VI-C4).

A. Benchmarks

We collected benchmarks from two sources:

- We selected *all* SQL query related exercises (XXX in total) from a classic database textbook [22]. All exercises are from Chapter 5, which systematically introduces the SQL language. Textbook exercises are good resource to evaluate SQLSynthesizer's generality, since such exercises are often designed to cover a wide range of SQL features. Some exercises are even designed on purpose to cover some less realistic, corner cases in using SQL. As shown in Figure 11, each textbook exercises involves at least 3 tables. It was unintuitive for us to write the correct query by simply looking at the problem description in the exercise.
- We searched SQL query related questions raised by real-world database users from 3 popular online forums [5], [25], [27]. We focused on questions about using standard SQL features rather than vendor-specific SQL features. We excluded questions that were vaguely described or obviously wrong, and discarded questions that had been proved to be unsolvable by using SQL (e.g., computing a transitive closure). We collected XXX recent forum questions related to writing a SQL query. **[[merge same types. exclude jdbc, why relative few]]**

B. Evaluation Procedure

We used SQLSynthesizer to solve each textbook exercise and forum question. If an exercise or problem was associated with example input and output, we directly applied SQLSynthesizer on those examples. Otherwise, we manually wrote some example input and output. To reduce the bias in writing examples, all examples are written by a different graduate student (whose research field is not database-related) from University of Washington other than SQLSynthesizer's developers.

We checked SQLSynthesizer's correctness by comparing its output with the expected SQL queries. Specifically, for textbook exercises, we compared SQLSynthesizer's output with their correct answers; for forum questions, we manually wrote the correct SQL query and then determined whether SQLSynthesizer can produce it.

For some textbook exercises and forum questions, SQLSynthesizer inferred a SQL query that satisfied the input-output examples, but did not behave as we expected when applied to other inputs. We manually found another input on which the SQL query mis-behaved and re-applied SQLSynthesizer to the new input. We repeated this process and recorded the number of interactions until SQLSynthesizer synthesized a desirable SQL query.

All experiments were run on a 2.67GHz Intel Core PC with 4GB physical memory (2GB was allocated for the JVM), running Windows 7.

C. Results

Figure 11 summarizes our experimental results.

1) Success Ratio: As shown in Figure 11, SQLSynthesizer synthesized expected SQL queries for XXX out of XXX the textbook exercises, and XXX out of XXX the forum questions. **[[why the technique can work, why some problem can not be solved]]**

2) Performance: We measured SQLSynthesizer s performance by recording the average time cost in producing a ranked list of SQL queries. As shown in Figure 11, the performance of SQLSynthesizer is reasonable. On average, it uses less than XXX minutes to produce the results in one interactive round. Most of the time is spent querying the backend database to validate the correctness of each synthesized SQL query.

3) Human Efforts: We measured the human efforts taken to use SQLSynthesizer in two ways. First, the time cost to write input-output examples. Second, the number of interactive rounds in invoking SQLSynthesizer to synthesize the desirable SQL queries.

As shown in Figure 11, human efforts spent in providing input-output examples are very limited: on average, it took less than 5 minutes for one benchmark. **[[explain some abnormal points]]**

The number of interactive rounds is a measure of the generalization power of the conditional learning part of the algorithm and the ranking scheme. We observed that the tool typically requires just XXX rounds of interaction, when the user is smart enough to give an example for each input format (which

Benchmarks			SQLSynthesizer					Query by
ID	Source	#Input Tables	Example Size	Rank	Tool Cost (s)	Cost in Writing Examples (s)	#Iterations	Output [26]
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							
1	Textbook Ex 5.1.1							

Fig. 11. Experimental results in synthesizing SQL queries. Column “Benchmarks” describes the characteristics of our benchmarks. Sub-column “#Input Tables” shows the number of input tables in each benchmark. Column “SQLSynthesizer” shows SQLSynthesizer’s results in synthesizing SQL queries. Sub-column “Example Size” shows the number of rows in all example input and output tables. Sub-column “Rank” shows the absolute rank of the desirable SQL query in SQLSynthesizer’s output. Sub-column “Tool Time Cost (s)” shows **[[]]**. Sub-column “#Iterations” shows the number of interactive rounds in using SQLSynthesizer to obtain the desirable SQL query. Column “Query by Output” shows the results of using a previous technique, called *Query by Output* (QBO) []. Since **[[treated as a special case]]**, we omit other. “Y” means QBO produces the desirable SQL queries, while “N” means QBO fails to produce the desirable SQL queries.

typically range from 1 to 3) to start with. This was indeed the case for most cases in our benchmarks, even though our algorithm can function robustly without this assumption. The maximum number of interactive rounds required in any scenario was **[[XXX]]** (with 2 to 3 being a more typical number). **[[the largest table]]** The maximum number of examples required in any scenario over all possible interactions was 10.

4) Comparison with an Existing Technique: .

We compared SQLSynthesizer with *Query By Output* (QBO), a data-driven approach to infer SQL queries [26]. We chose QBO because it is the most recent technique and also one of the most precise SQL query inference techniques in the literature. QBO requires similar input as SQLSynthesizer, and uses a decision-tree-based algorithm **[[explaining what is QBO]]** However, QBO cannot infer SQL queries using **[[aggregates]]**

The experimental results of QBO is shown in Figure 11 (Column “Query by Example”). For all XXX database exercises and XXX forum questions, QBO produces correct answers for XXX and XXX of them, respectively. QBO fails to synthesize desirable SQL queries for other benchmarks, because it **[[the reasons]]**.

We did not compared SQLSynthesizer with other related techniques [], for three reasons. **[[reasons]]**

D. Experimental Discussion

Limitations. The experiments indicate three limitations of our technique. First, some query tasks cannot be formulated by our SQL subset (Section III) due to some unsupported features, such as nested queries. This limitation is expected; and our future work should address this by including more SQL features in SQLSynthesizer. Second, on some examples, the learned query conditions, though correct, are not precise enough; and require users to provide more informative examples. Take the example input and output in Figure 10 as an example, SQLSynthesizer produces a SQL query `select name from student where score > 2` to satisfy the examples. However, if the condition of the expected query is `score > 3`, users must provide one more tuple in the input table, such as ”Chris, 3” (a tuple with value “Chris” in the name column and “3” in the score column), while keeping the output table the same, to guide SQLSynthesizer to learn the correct query condition. Third, SQLSynthesizer 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), SQLSynthesizer will declare failure when it fails to infer 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.

Threats to Validity. There are three major threats to validity in our evaluation. First, the XXX textbook exercises and XXX forum questions, though covering a wide variety of SQL features, may not be representative enough. Thus, we can not claim the results can be generalized to an arbitrary use-case scenario. Second, we only compared SQLSynthesizer with the *Query by Output* technique [26]. Using other query inference or recommendation techniques might achieve different results. Third, our experiments focus on evaluating SQLSynthesizer's generality and accuracy. Even though all experiments are carried out by a different person other than SQLSynthesizer's developers, it is unknown about SQLSynthesizer's general usability in practice. To address this issue, we plan to conduct a user study in our future work.

Experimental Conclusions. We have three chief findings: **(1)** The supported SQL subset in SQLSynthesizer is expressive enough to describe a variety of database queries. **(2)** SQL-Synthesizer can efficiently synthesize desirable SQL queries with a small amount of human efforts and small input-output examples. **(3)** SQLSynthesizer produces better results than an existing technique (*Query by Output* [26]).

VII. Related Work

This section discusses two categories of closely-related work on reverse engineering SQL queries and automated program synthesis.

A. Reverse Engineering SQL Queries

Reverse engineering SQL queries is a well-known technique in the database community [4], [26], [28] to enhance a database system's usability. Zloof's pioneering work on *Query by Example* (QBE) [28] provided a high-level query language and a form-based Graphical User Interface (GUI) for writing database queries. To use the QBE system, users need to learn its own query language, formulate a query with the language, and fill in the appropriate skeleton tables on the GUI. By contrast, SQLSynthesizer mitigates such learning curves by only requiring users to provide some representative examples to describe their query intentions

Tran et al. [26] proposed a technique, called *Query by Output* (QBO), to find a set of semantically-equivalent SQL queries to a given input query. QBO is related to but significantly differs from SQLSynthesizer in two aspects. First, QBO has a rather different goal and requires different inputs : it takes as inputs a database, an SQL query, and the query's output (on the database); and computes one or more equivalent queries that produce the same output on the input database. Second, QBO can only infer simple select-project-join queries, while excluding many useful SQL features, such as aggregates, the `Having` clause, and the `Group by` clause. As we demonstrated in in Section VI, queries inferred by QBO cannot be applied to many of the other real-world cases. QBO's key limitation stems from the fact that it only considers existing tuple values in the input tables as features, when learning a set of classification rules as query conditions. By contrast, SQLSynthesizer remedies this limitation by enhancing the existing tuple values with two kinds of additional features (Section IV-C1).

Recently, Sarma et al. [4] studied the *View Definitions Problem* (VDP). VDP aims to find the most succinct and accurate view definition, when the view query is restricted to a specific family of queries. VDP can be solved as a special case in SQLSynthesizer where there is only one input table and one output table. Furthermore, the main contribution of Sarma et al's work is the complexity analysis of three variants of the view definitions problem; there is no tool implementation or empirical studies to evaluate the proposed technique.

B. Automated Program Synthesis

Program synthesis [10] is a useful technique to create an executable program in some underlying language from specifications that can range from logical declarative specifications to examples or demonstrations [1], [2], [6], [11], [14], [16], [18], [19], [23]. It has been used recently for many applications such as synthesis of efficient low-level code [24], data structure manipulations [7], geometry constructions [12], snippets of excel macros [14], relational data representations [1], [2] and string expressions [11], [23].

The PADS [7] system takes a large sample of unstructured data and infers a format that describes the data. Related, the Wrangler tool, developed in the HCI community, provides a visual programming-by-demonstration interface to table transformations for data cleaning [16]. These two techniques, though well-suited for tasks like text extraction, are inapplicable to synthesizing a database query, since they use completely different abstractions than SQL and lack the support for many database operations like table joining, aggregations, etc.

Harris and Gulwani described a system for learning excel spreadsheet transformation macros from an example input-output pair [14]. Given one input table and one output table, their system can infer an excel macro that filters, duplicates, and/or reorganizes table cells to generate the output table. SQLSynthesizer differs in multiple respects. First, excel macros have significantly different semantics than the SQL language. An excel macro can express a variety of table transformation operations (e.g., table re-shaping), but are not capable to formulate database queries. Second, Harris and Gulwani's approach treats table cells as atomic units, and thus has different expressiveness than SQLSynthesizer. For instance, their technique can generate macros to transform one table to another, but cannot join multiple tables or group query results by certain table columns.

Some recent work proposed query recommendations systems to reduce the obstacles to using relational databases [15], [17]. SQLShare [15] is a cloud-based service that allows users to upload their data and SQL queries. Each uploaded query is saved as a view, allowing other users to compose and reuse. SnipSuggest [17] is a SQL autocompletion system. As a user types a query, SnipSuggest mines existing query logs to recommend relevant clauses or SQL snippets (e.g., the table names for the `from` clause) based on the partial query that the user has typed so far. Compared to SQLSynthesizer, both SQLShare and SnipSuggest assume the existence of a comprehensive query log (produced either by the user herself or other users) that contains valuable information. However, such assumption often does not hold for many database users in practice. SQLSynthesizer eliminates this assumption and infers SQL queries by using user-provided examples.

Cheung et al [3] presented a technique to infer SQL queries from imperative code. Their technique identifies fragments of application logic (written in an imperative language like Java) that can be pushed into SQL queries. Compared to SQLSynthesizer, their work is designed for developers to improve a database application's performance, rather than helping non-expert end-users write correct SQL queries from scratch. If an end-user wishes to use their technique to synthesize a SQL query, she must write a snippet of imperative code to describe the query task. Comparing to providing example input and output as required by SQLSynthesizer, writing correct imperative code can be too challenging for a typical end-user, and thus would substantially degrade the technique's usability in practice.

VIII. Conclusion and Future Work

This paper studied the problem of automated SQL query synthesis from simple input-output examples, and presented a practical technique (and its tool implementation, called SQLSynthesizer). SQLSynthesizer is motivated by the growing population of non-expert database users, who need to query on their databases, but have difficulty with SQL. We have shown that that SQLSynthesizer is able to synthesize a variety of SQL queries, and it does so with small input-output examples. We view SQLSynthesizer as an important step toward making databases more usable. The source code of our tool implementation is available at: http://sqlsynthesizer.googlecode.com

For future work, we are interested in conducting a user study to evaluate SQLSynthesizer's usability. We also plan to explore applications of this technique.

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] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] MDX: A query language for OLAP databases. http://msdn.microsoft.com/en-us/library/gg492188.aspx.

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

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

[23] 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.

[24] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

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

[26] 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.

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

[28] 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.