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. If 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 exercises from a classic database textbook and XXX forum questions about writing SQL queries. SQLSynthesizer synthesized correct answers for XXX textbook exercises and XXX forum questions, and it did 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. There is a growing population of non-expert database end-users, who need to perform analysis on their databases, but have limited programming knowledge.

Although the relational database management system (RDBMS) and the de facto query language (SQL) are perfectly adequate for most end-users' needs, the costs associated with use of database and SQL are non-trivial [16]. 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). Thus, non-expert 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 performing 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 valuable.

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

Many RDBMS come with a well-designed GUI with lots 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.

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 programming languages have never been easy for end-users who are not professional programmers. Learning a practical programming language (even a simplified domain specific language, such as MDX [21]) 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 [1] from input-output examples. 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 easiest ways for end-users to describe *what* the task is. If the synthesized SQL query is applied to the example input, then it produces the example output; and if the synthesized SQL query is applied to other similar input (potentially much larger tables), then it produces a corresponding output.

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 [7], Coursera [4], and Udacity [29] are teaming up with experts to provide high quality online courses to several thousands

[1] All queries mentioned in this paper refer to SQL queries that retrieve data from a database but do not update its content.

of students worldwide. One challenge, which is not present in a traditional classroom setting, is to provide answers to questions raised by a large number of students. A tool, like SQLSynthesizer, that has the potential of answering 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 a given example input and output pair, as proved by Sarma et al. [5], 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 becomes 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 operations; and it needs to project data on certain columns to form the output. All such operations must be inferred properly and efficiently.

To address these challenges and make example-based SQL query synthesis feasible in practice, SQLSynthesizer focuses on a widely-used SQL subset (Section III), and uses three steps to link a user's intention to a SQL query (Section IV):

- **Skeleton Creation.** SQLSynthesizer scans the given input-output examples and heuristically determines table joins and projection columns in the result query. Then, it creates an incomplete SQL query (called, a query skeleton) to capture the basic structure of the result query.
- **Query Completion.** SQLSynthesizer uses a machine learning algorithm to infer a set of accurate and expressive rules, which transforms the input example into the output example. Then, it searches for other missing parts in a query skeleton, and then builds a list of 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], [5], [27], [30], 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 [30] 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; and most of them can only infer simple select-from-where queries on a single table [3], [5], [5], [27], [30]. By contrast, SQLSynthesizer significantly enriches the supported SQL subset. Besides supporting the standard select-from-where queries, SQLSynthesizer also supports many advanced 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 [24]. We used textbook exercises because 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 tool for end-users to use.

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 is useful in synthesizing SQL queries 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 [24] (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 typical end-user. An end-user must carefully choose the right SQL features and use them correctly to fullfil the described task.

As an alternative, users can use SQLSynthesizer 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 to infer 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 results by the `student_id` column. Further, the query selects all senior students (using a query condition in the `WHERE` clause) who are enrolled in more than 2 courses (using a condition in the `HAVING` clause). Finally, the query projects the result on the

[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	5

(a) Two input tables: student (Left) and enrolled (Right)

```
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
```

(b) A SQL query inferred by SQLSynthesizer

name	max_score
Dan	5
Sai	5

(c) An output table

Fig. 1. Example input and output tables and the SQL query synthesized by SQLSynthesizer for solve the example described 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.

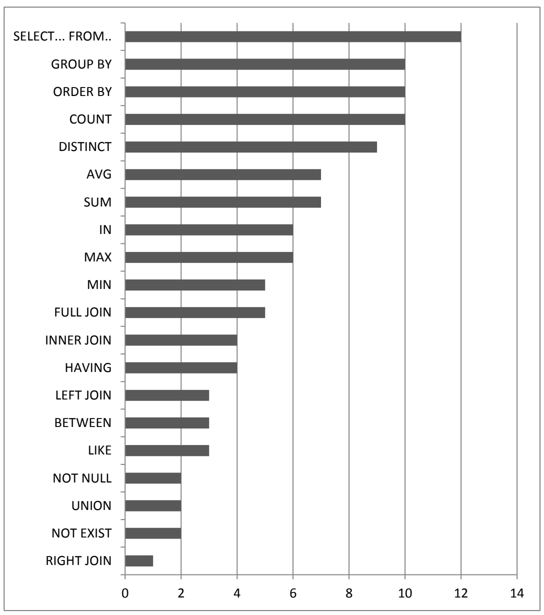

Fig. 2. Survey results of the most widely-used SQL features in writing a database query. There were 12 participants in the survey, and each participant was asked to select the top 10 widely-used SQL features. SQL features with no vote are omitted in this Figure for brevity.

$$
\begin{array}{rcl}
\langle query \rangle & ::= & \texttt{SELECT}\ \langle expr \rangle^{+}\ \texttt{FROM}\ \langle table \rangle^{+} \\
& & \texttt{WHERE}\ \langle cond \rangle^{+} \\
& & \texttt{GROUP BY}\ \langle column \rangle^{+}\ \texttt{HAVING}\ \langle cond \rangle^{+} \\
& & \texttt{ORDER BY}\ \langle column \rangle^{+} \\
\langle table \rangle & ::= & atom \\
\langle column \rangle & ::= & \langle table \rangle.atom \\
\langle cond \rangle & ::= & \langle cond \rangle\ \texttt{\&\&}\ \langle cond \rangle \\
& | & \langle cond \rangle\ \texttt{||}\ \langle cond \rangle \\
& | & \texttt{(}\ \langle cond \rangle\ \texttt{)} \\
& | & \langle cexpr \rangle\ \langle op \rangle\ \langle cexpr \rangle \\
\langle op \rangle & ::= & \texttt{=}\ |\ \texttt{>}\ |\ \texttt{<} \\
\langle cexpr \rangle & ::= & const\ |\ \langle column \rangle \\
\langle expr \rangle & ::= & \langle cexpr \rangle\ |\ \texttt{COUNT(}\langle column \rangle\texttt{)}\ |\ \texttt{COUNT(DISTINCT}\ \langle column \rangle\texttt{)} \\
& | & \texttt{SUM(}\langle column \rangle\texttt{)}\ |\ \texttt{MAX(}\langle column \rangle\texttt{)}\ |\ \texttt{MIN(}\langle column \rangle\texttt{)}
\end{array}
$$

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.

`student.name` column and uses the `MAX` aggregate to compute the maximum course score.

III. A SQL Subset Supported in SQLSynthesizer

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 empirical study has ever been conducted to this end, and little evidence has ever been provided on which SQL features are widely-used in practice. Without such knowledge, deciding which SQL subset to support remains difficult.

To address this challenge and reduce our personal bias in designing the language subset, 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). Later, we 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.

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 are 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 present them a list of *all* standard SQL features in writing a query. Additionally, participants are asked to report their own experience in using SQL in the third part of the survey.

We sent out invitation to the graduate students 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 had 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 subset of the standard SQL language, whose syntax is shown in Figure 3.

The supported SQL subset covers all top 10 most widely-used SQL features voted by the survey participants in Figure 2, except for the `IN` keyword. In addition, the SQL subset supports the `HAVING` keyword since `HAVING` is often used together with the `GROUP BY` clause. Our SQL subset, though by no means complete in writing all possible queries, has significantly enriched the SQL subsets supported by the existing query inference work [5], [27]. Besides being able to write the standard select-from-where queries as in [5], [27], our SQL subset also supports table joins, aggregates (e.g., `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 the SQL subset, we focused on standard SQL features, and excluded user-defined functions and vendor-specific features, such as 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 or nested queries, which are the major source of the PSPACE-hardness in query inference [5]. Related, the `LIKE` keyword is designed for string wildcard matching; determining its matching patterns requires systematic search and can be expensive in practice. Thus, for the sake of inference efficiency, we exclude these keywords.

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 sufficiency 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.

```
                 Projection column   Aggregate
SELECT student.name, MAX(enrolled.score)
FROM student, enrolled            ← Query tables
WHERE student.student_id = enrolled.student_id ← Join conditions
      and student.level = 'senior'  ← Query conditions
GROUP BY student.student_id         ← GROUP BY clause
HAVING COUNT(enrolled.course_id) > 2 ← HAVING clause
ORDER BY student.name              ← ORDER BY clause
```

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

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

Overall, based on the feedback by experienced IT professionals, we believe our SQL subset is usable for end-users in writing common database queries.

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-C, 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) creates 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 places the more likely ones near the top. Users can inspect the query list and select an expected query from it. If the synthesized SQL queries satisfy the example input and output, but do not address the user's intention, SQLSynthesizer can be used interactively by requesting more informative examples from the end-user and then update the result queries.**[[xx]]**

Take the query in Figure 4 as an example, the "Skeleton Creation" step infers its projection columns, 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

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

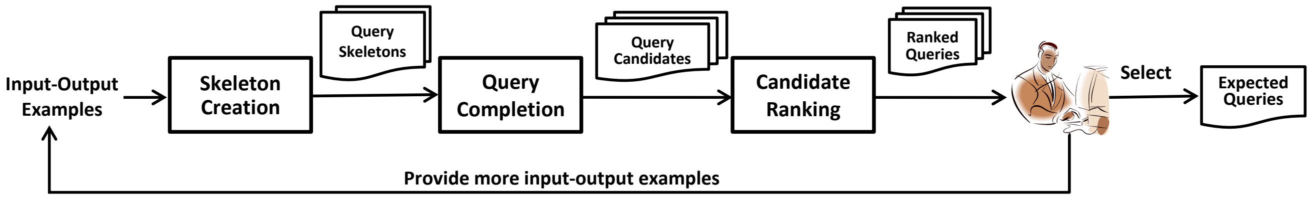

Fig. 5. Illustration of SQLSynthesizer's workflow of synthesizing SQL queries from input-output examples.

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

synthesizeSQLQueries(T_I, T_o)

1: *queryList* ← an empty list
2: *skeletons* ← createQuerySkeletons(T_I, T_O)
3: **for** each *skeleton* in *skeletons* **do**
4: *conds* ← inferConditions(T_I, T_O, *skeleton*)
5: *aggs* ← searchForAggregates(T_I, T_O, *skeleton*, *conds*)
6: *columns* ← searchForOrderBys(T_O, *skeleton*, *aggs*)
7: *queries* ← 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.

examples (lines 8 – 11). Line 14 corresponds to the "Query Ranking" step.

B. Skeleton Creation

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

Determining query tables. A typical end-user is often unwilling to provide more than enough example input. Thus, we assume every example input table is 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. 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 input table will be used multiple times in the query, such as being 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 yield a large number of join conditions and is feasible in practice. To determine the most likely join conditions among them, SQLSynthesizer uses two rules to repeatedly join two different tables in the query tables, until all tables get joined. First,

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

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, SQLSynthesizer uses the second rule to join tables on 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 in 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 an input table. If so, SQLSynthesizer uses the matched column from the input table as the projection column in the skeleton. Otherwise, the output column must be created by using an aggregate. Take the output table in Figure 1 as an example, SQLSynthesizer determines that the `name` column is from the `student` table and the `max_score` column is created by using an aggregate over some column.

For an example input and output pair, depending on the number of join conditions, SQLSynthesizer may create multiple skeletons, which share the same query tables and projection columns, but differ in the join condition. For the example in Figure 1, SQLSynthesizer creates one query skeleton shown in Figure 8.

C. Query Completion

In this step, SQLSynthesizer completes the missing parts in each query skeleton, and then outputs a list of SQL queries that satisfy the given example input and output pair.

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

An input table

C1	C2
2	4
2	1
2	1
1	1

Aggregation Features

Group by **C1**						Group by **C2**					
`COUNT(C2)`	`COUNT (DISTINCT C2)`	`MIN(C2)`	`MAX(C2)`	`SUM(C2)`	`AVG(C2)`	`COUNT(C1)`	`COUNT (DISTINCT C1)`	`MIN(C1)`	`MAX(C1)`	`SUM(C1)`	`AVG(C1)`
3	2	1	4	6	2	1	1	2	2	2	2
3	2	1	4	6	2	3	2	1	2	5	5/3
3	2	1	4	6	2	3	2	1	2	5	5/3
1	1	1	1	1	1	3	2	1	2	5	5/3

Comparison Features

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

Fig. 7. Illustration of two types of additional features added by SQLSynthesizer. (Left) An example input table with two columns: C1 and C2. (Center) The aggregation features added by SQLSynthesizer for the input table. (Right) The comparison features added by SQLSynthesizer for the input table. Take the first row in the input table as an example, when grouping the table by column C1 (with value 2), the count of values in the C2 column is 3; the count of distinct values in the C2 column is 2; the minimal value in the C2 column is 1, the maximal value in the C2 column is 4, the sum of values in the C2 column is 6, and the average value in the C2 column is 2.

part and a negative part. In our context, the search space is all tuples from joining 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, the key challenge is how to design a good feature set. Existing approaches [27] simply use tuple values in the input table(s) as features, and limits their abilities in inferring more complex query conditions. In particular, merely using tuple values as features can only infer conditions comparing 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 values of two table columns (e.g., `enrolled.course_id > enrolled.score`).

To address this challenge, SQLSynthesizer adds two types of additional features to each tuple, and uses the existing tuple values together with the additional features for rule learning.

- **Aggregation Features**. For each column in the joined table, SQLSynthesizer first tries to group all tuples by *each* tuple's value, and then applies every applicable aggregate[4] to each of the *remaining* columns to compute the corresponding aggregation result. The "Aggregation Features" part in Figure 7 shows an example.
- **Comparison Features**. For each tuple, SQLSynthesizer compares the values of every two type-compatible columns, and records the comparison results (1 or 0) as features. The "Comparison Features" part in Figure 7 shows an example.

SQLSynthesizer employs a variant of the decision tree algorithm, called PART [11], to learn a set of rules as query conditions. We choose PART because it uses a "divide-and-conquer" strategy to build rules incrementally, and thus is faster and consumes less memory than the original decision tree algorithm [23]. Using additional features added by SQLSynthesizer, the PART algorithm is able to discover rules that are hard to identify by merely using the original tuple values as features. Figure 9 shows how such additional features help in learning rules for the motivating example in Figure 1.

SQLSynthesizer next divides the learned conditions into two disjoint parts, and places each part to the appropriate clause. Specifically, SQLSynthesizer places conditions using aggregates to the `HAVING` clause, and places other conditions to the `WHERE` clause. This is based on the SQL language's specification: query conditions using aggregates are valid only when they are used *together with* the `GROUP BY` clause and *inside* the `HAVING` clause. Take the conditions inferred in Figure 9 as an example, SQLSynthesizer places the query condition: `student.level ='senior'` in the `WHERE` clause, places condition: `COUNT(enrolled.course_id) > 2` in the `HAVING` clause, and places column `student_id` to the `GROUP BY` clause.

2) Searching for Aggregates: For every column in the output table that has no matched column in the input table(s), SQLSynthesizer searches for the desirable aggregate by repeatedly applying each aggregate on every input table column; and then checking whether the aggregate (with the input table column) produces the same output as in the output table. To speed up the exhaustive search, SQLSynthesizer uses two sound heuristics to filter away infeasible combinations.

- SQLSynthesizer only applies an aggregate to its *type-compatible* table columns. Specifically, the data values in an output column must be compatible with an aggregate's return type. For instance, if an output column contains float values, it cannot be produced by using the `COUNT` or `COUNT DISTINCT` aggregates, or using the `MAX` aggregate over a column of integer type. On the other hand, some aggregates cannot be applied on table columns of certain types. For example, the `AVG` and `SUM` aggregates cannot be applied to columns of string type. SQLSynthesizer encodes such knowledge to prune the search space.
- SQLSynthesizer checks whether each value in the output column has appeared in the input table. If not, the output column cannot be produced by using the `MAX` or `MIN` aggregate.

For the example in Figure 1, SQLSynthesizer determines the `max_score` column in the output table is produced by using the `MAX(score)` aggregate.

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

For the output table in Figure 1, SQLSynthesizer determines no column should be added to the `ORDER BY` clause, since neither output column is sorted.

[4] `COUNT`, `COUNT DISTINCT`, `MAX`, `MIN`, `SUM`, and `AVG` for a numeric type column; and `COUNT`, and `COUNT DISTINCT` for a string type column

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

Aggregation Features: Group by **student_id** ...

(a)

COUNT(course_id) >2 && level = "senior"

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	5	Sai	senior

(b)

Project tuples on column: name, and aggregate: MAX(score)

name	max_score
Dan	5
Sai	5

(c)

Fig. 9. Illustration of how additional features added by SQLSynthesizer helps in inferring query conditions for the example in Figure 1. (a) shows SQLSynthesizer enriches the original tuple values (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). Without the aggregation features added 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 the `name` column and the `MAX(score)` aggregate.

D. Candidate Ranking

It is possible that multiple SQL queries satisfying the given input-output examples will be returned. To help end-users select their expected queries, SQLSynthesizer ranks more likely queries higher in the output. To do so, SQLSynthesizer employs the Occam's razor principle, which states that the simplest explanation is usually the correct one. A simpler query is less likely to overfit the given examples than a complex one, 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 or clause are pairwise simpler. For example, expression `COUNT(student_id)` is simpler than `COUNT(DISTINCT student_id)`. Simpler query conditions and expressions often suggest 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 computed by summarizing the costs of all conditions, aggregates, and other expressions appearing in the `GROUP BY` and `ORDER BY` clauses. (The cost of each condition, aggregate and expression is approximated by its length.) Figure 10 shows an example.

name	score
Bob	4
Dan	5
Jim	2

(a) An input table: student

name
Bob
Dan

(b) An 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 ranking heuristic. SQLSynthesizer produces two queries for the given examples. The first query differs from the second query in using a simpler condition, and thus is ranked 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. 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.

V. Implementation

We implemented the proposed technique in a tool, called SQLSynthesizer. SQLSynthesizer uses the built-in PART algorithm implementation in the Weka toolkit [14] to learn query conditions (Section IV-C). SQLSynthesizer also uses MySQL [22] 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 query result matches the given output.

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

Our benchmarks are shown in Figure 11.

- We used XXX SQL query related exercises from a classic database textbook [24]. These exercises are from Chapter 5, which systematically introduces the SQL language. We chose textbook exercises because they are 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. We used *all* exercises that can be answered using the standard SQL language features without any vendor-specific features or user-defined numeric functions. As shown in Figure 11, most textbook exercises involve at least 3 tables. It was unintuitive for us to write the correct query by simply looking at the problem description.
- We searched SQL query related questions raised by real-world database users from 3 popular online forums [6], [26], [28]. We focused on questions about how to use standard 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 non-trivial forum questions related to writing a SQL query (available at: [10]), among which two questions even did not receive any reply. Writing a good forum post is often harder than asking around a SQL expert (since the post has to clearly describe the problem), and these end-users had already tried but failed to find the correct SQL query before they wrote the post.

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 graduate student (whose research field is not database-related) from University of Washington rather than SQLSynthesizer's developers.

We checked SQLSynthesizer's correctness by comparing its output with the expected SQL queries. For textbook exercises, we compared SQLSynthesizer's output with their correct answers; for forum questions, we first checked SQLSynthesizer's output with the confirmed answer in the same post, if there is any. Otherwise, we manually wrote the correct SQL query and then compared it with SQLSynthesizer's output.

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

All experiments were run on a 2.67GHz Intel Core PC with 4GB physical memory, running Windows 7.

C. Results

Figure 11 summarizes our experimental results.

1) Success Ratio: SQLSynthesizer synthesized correct SQL queries for XXX out of XXX the textbook exercises, and all XXX forum questions. SQLSynthesizer failed to solve XXX textbook exercises, for two reasons. XXX exercises **[[why the technique can work, why some problem can not be solved]]**

We also observed that our ranking strategy (Section IV-D) was quite effective: for most benchmarks, it ranked the correct SQL query as one of top XXX suggestions.

2) Performance: On average, including benchmarks that SQLSynthesizer failed to produce a correct answer, SQLSynthesizer took less than XXX minutes in total to produce the results (max: xx, min: xx). For benchmarks on which SQLSynthesizer succeeded, the average time cost was xxx minutes (max: xxx, min: xxx). Most of the time is spent querying the backend database to validate the correctness of each synthesized SQL query. SQLSynthesizer's speed makes it an attractive tool to replace the role of the SQL experts, which enables end-users to solve their problems in a few minutes.

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

The human efforts spent in writing input-output examples are reasonable. For all succeeded benchmarks, it took less than 5 minutes on average to write examples for one benchmark (max: XXX, min: XXX); and the average example size is XXX (max: XXX, min: XX). To produce the correct SQL query, SQLSynthesizer typically requires just XXX rounds of interaction (max: XXX, min: XXXX). For most succeeded benchmarks, the number of interaction rounds ranges from XXX to XXX.

For those failed benchmarks, we observed that a typical user often gave up after XXX interactions. **[[xxx]]**

4) Comparison with an Existing Technique: .

We compared SQLSynthesizer with *Query By Output* (QBO), an approach to infer SQL queries [27] from examples. We chose QBO because it is the most recent technique and also one of the most accurate SQL query inference techniques in the literature. QBO requires an example input-output pair, and uses the decision tree algorithm to infer a query. However, QBO has three fundamental limitations. First, it can only join two tables on their key columns (annotated by users), and requires users to specify how to project the results by annotating the projection columns. Second, it uses the original tuple values in input tables as learning features, and thus can only infer simple query conditions. Third, QBO does not support many useful SQL features, such as aggregates, the `GROUP BY` clause, and the `HAVING` clause.

We implemented QBO as a special case of SQLSynthesizer, annotated each example table as required by QBO, and run it

Benchmarks			SQLSynthesizer					Query by
ID	Source	#Input Tables	Example Size	Rank	Time Cost (s)	Cost in Writing Examples (m)	#Iterations	Output [27]
1	Textbook Ex 5.1.1	4						
2	Textbook Ex 5.1.2	4						
3	Textbook Ex 5.1.3	4						
4	Textbook Ex 5.1.4	4						
5	Textbook Ex 5.1.5	4						
6	Textbook Ex 5.1.6	4						
7	Textbook Ex 5.1.7	4						
8	Textbook Ex 5.1.8	4						
9	Textbook Ex 5.1.9	4						
10	Textbook Ex 5.1.10	4						
11	Textbook Ex 5.1.11	4						
12	Textbook Ex 5.1.12	4						
13	Textbook Ex 5.2.1	3						
14	Textbook Ex 5.2.2	3						
15	Textbook Ex 5.2.3	3						
16	Textbook Ex 5.2.4	3						
17	Textbook Ex 5.2.5	3						
18	Textbook Ex 5.2.6	3						
19	Textbook Ex 5.2.7	3						
20	Textbook Ex 5.2.8	3						
21	Textbook Ex 5.2.9	3						
22	Textbook Ex 5.2.10	3						
23	Textbook Ex 5.2.11	3						
24	Forum Question 1							
25	Forum Question 2							
26	Forum Question 3							
27	Forum Question 4							
28	Forum Question 5							

Fig. 11. Experimental results. Column "Benchmarks" describes the characteristics of our benchmarks. Column "#Input Tables" shows the number of input tables in each benchmark. Column "SQLSynthesizer" shows SQLSynthesizer's results. Column "Example Size" shows the number of tuples (i.e., rows) in all example input and output tables. Column "Rank" shows the absolute rank of the correct SQL query in SQLSynthesizer's output. "**X**" means SQLSynthesizer fails to produce a correct answer. Column "Cost in Writing Examples (m)" shows the total time cost of writing sufficient examples in minutes. Column "#Iterations" shows the number of interactive rounds in using SQLSynthesizer to obtain the correct query. Column "Query by Output" shows the results of an existing technique, called *Query by Output* (QBO) [27]. In this column, "**Y**" means QBO produces the correct SQL query, and "**N**" means QBO fails to produce the correct query. Since QBO was implemented as a special case of SQLSynthesizer; its time cost is similar to SQLSynthesizer and is omitted for brevity.

on the same benchmarks. Its results are shown in Figure 11. For all XXX database exercises and XXX forum questions, QBO only produces correct answers for XXX and XXX of them, respectively. Without surprise, *all* benchmarks solved by QBO can also be solved by SQLSynthesizer. QBO's poor performance is primarily caused by its limited support for learning join conditions, query conditions, as well as many other SQL features.

We did not compared SQLSynthesizer with other related techniques [3], [15]–[17], for two reasons. First, some techniques such as [3], [16], [18], require completely different input (e.g., a query log [16], [18] or a snippet of Java code [3]) than SQLSynthesizer. Second, other techniques produce completely different output (e.g., an excel transformation macro [15], or a text editing script [17]) than SQLSynthesizer. All these factors make it hard to conduct a meaningful comparison.

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 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 to the input table, such as "Chris, 3" (a tuple with "Chris" in the name column and "3" in the score column), while keeping the output table unchanged, 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. To overcome this limitation, we plan to design a more robust inference algorithm that can 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 [27]. 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 rather than SQLSynthesizer's developers, it is unknown about SQLSynthesizer's general usability. A user study is needed to eliminate this threat.

Experimental Conclusions. We have three chief findings: **(1)** SQLSynthesizer is effective in synthesizing SQL queries with small input-output examples. **(2)** SQLSynthesizer is fast enough for practical use; and needs a small amount of human efforts in writing examples; **(3)** SQLSynthesizer produces significantly better results than an existing technique (*Query by Output* [27]).

VII. Related Work

We next discuss two categories of related work on reverse engineering SQL queries and automated program synthesis.

Reverse Engineering SQL Queries. Reverse engineering SQL queries is a technique in the database community [5], [27], [30] to improve a database's usability. Zloof's pioneering work on *Query by Example* (QBE) [30] 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 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 intentions

Tran et al. [27] 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 slightly 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 Section VI, QBO cannot infer queries for many use-case scenarios. Its key limitation stems from the fact that it only considers existing tuple values in the input tables as features when learning query conditions. By contrast, as shown in Section IV-C1, SQLSynthesizer addresses this limitation by enhancing the existing tuple values with two kinds of additional features, and thus is able to learn more complex conditions.

Recently, Sarma et al. [5] 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.

Automated Program Synthesis. Program synthesis [12] 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], [8], [13], [15], [17], [19], [20], [25]. It has been used recently for many applications.

The PADS [9] 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 interface for table transformations and data cleaning [17]. These two techniques, though well-suited for tasks like text extraction, cannot be used to synthesize a database query. This is because text extraction tasks use completely different abstractions than a database query task, and the existing tools like PADS and Wrangler lack the support for many database operations such as joins and aggregation.

Harris and Gulwani described a system for learning excel spreadsheet transformation macros from an example input-output pair [15]. Given one input table and one output table, their system can infer an excel macro that filters, duplicates, and/or re-organizes 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 aggregate query results by certain table columns.

Some recent work proposed query recommendations systems to enhance a database's usability [16], [18]. SQLShare [16] is a web service that allows users to upload their data and SQL queries, permitting other users to compose and reuse. SnipSuggest [18] 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 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 using user-provided examples.

Cheung et al. [3] presented a technique to infer SQL queries from imperative code. Their technique transforms fragments of application logic (written in an imperative language like Java) into SQL queries. Compared to SQLSynthesizer, their work aims to help developers 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, writing correct imperative code is too challenging for a typical end-user, and thus would inevitably degrade the technique's usability.

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 their databases, but have difficulty with SQL. We have shown that SQLSynthesizer is able to synthesize a variety of SQL queries, and it does so with small input-output examples. The source code of SQLSynthesizer 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] Coursera. https://www.coursera.org/.

[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] EdX. https://www.edx.org/.

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

[9] 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.

[10] Forum questions used in evaluation. http://code.google.com/p/sqlsynthesizer/source/browse/#svn\%2Ftrunk\%2Fsqlsynthesizer\%2Fdat\%2Fforum_questions.

[11] 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.

[12] 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.

[13] 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.

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

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

[23] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, Mar. 1986.

[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] Udacity. https://www.udacity.com/.

[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.