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 the programming knowledge to do such tasks smoothly. 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 input and output example 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 infers correct queries for XXX textbook exericses and XXX forum questions, including one question that received no replies.

I. Introduction

The big data revolution over the past few years has resulted in significant advances in digitization of massive amounts of data and accessibility of computational devices to massive proportions of the population. A key challenge faced by many enterprise or computer end-users nowadays is the management of their increasingly large and complex databases.

A. End-users' Difficulties in Writing SQL Queries

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 research scientists, 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 query a variety of information on their database and use the information to support their business decisions.

To learn how to query a RDBMS, most end-users often refer to a textbook or online resources to first get familiar with the basic idioms of the SQL language. Then, they may try to write some experimental queries, execute them on a sample database to observe the output, and subsequently revise the queries (if the output is not desirable). However, such practice is inefficient and time-consuming. Non-professional end-users are often stucked with the process of *how* to accomplish a certain task by receiving step-by-step, detailed, and syntactically correct instructions. Even though most end-users can clearly describe *what* the task is, they simply can not get the SQL query correct, either due to the syntax complexity of the SQL language itself, or the structure complexity of the databases. To write a correct SQL query, end-users often need to seek information from online help forums, or ask SQL experts. This process can be repetative, laborious, and frustrating. To assist them 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 highly desirable.

B. 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 specific tasks. On the other hand, as a GUI supports more and more customization features, users may struggle to discover those features, which can significantly degrade its usability.

General programming languages, such as SQL, Java (with JDBC), or other domain specific 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.

C. *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. SQLSynthesizer takes example input table(s) and output table from the end-users, and then automatically infers a SQL query (or multiple queries, if exist) that queries against the input tables and returns the output example. If the inferred SQL query is applied to the example input, then it produces the example output; and if the SQL query is applied to other similar inputs (potentially much larger tables), then the SQL query produces a corresponding output.

Although input-output examples may lead to underspecification, writing them, as opposed to writing declarative sepcifications or transformation scripts of any forms, is a more straightforward way of describing *what* the task is. SQLSynthesizer uses such input-output examples as a natural interface to "understand" an end-user's intention and provide corresponding assistance.

SQLSynthesizer is designed to be used by non-professional or novice database end-users when they do not know how to write a desirable SQL query. It aims to help end-users solve their problems by replacing the role of the SQL expert. End-users can use SQLSynthesizer to obtain SQL query to transform multiple, huge database tables by constructing small, representative input and output example tables. We also envision this technique 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 can help answer SQL problems would be useful.

Inferring SQL queries from examples can be challenging. This is primarily because **[[some reasons.]]**. SQLSynthesizer focuses on an **[[important]]** SQL subset (described in 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 to determine the table set, joining columns, and output columns that might be used in the result query. Then, it creates an incomplete SQL query (called, query skeleton) to capture the basic structure of the result query.
- **Condition Inference.** 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; and outputs a set of syntactically-valid SQL queries.
- **Solution Ranking.** If multiple SQL queries satisify the given input-output examples, SQLSynthesizer employs the Occam's razor principle to rank more likely queries higher in the result.

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

- **It is fully automated.** Besides input-output examples, SQLSynthesizer does not require users to provide annotations or hints of any form. This distinguishes our work from competing techniques such as interactive query writing [28] and programmer-assisted SQL sythensis [3].
- **It supports a wide range of SQL queries.** Similar approaches in the literature support a small subset of the SQL language, such as data selection (from a single table) [4], [28] and table joining [3], [4], [26]. By contrast, SQLSynthesizer significantly enriches the supported SQL subset. Besides supporting the standard data selection and table joining operations, SQLSynthesizer also supports aggregate functions (i.e., `MAX()`, `MIN()`, `SUM()`, and `COUNT()`), group by operation, order by operation, and the `HAVING` statement.

D. *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]. Exercises from a textbook are good resource to evaluate SQLSynthesizer's generality, since textbook exercises are often designed to cover a wide range of SQL features. (Some exercises are even designed on purpose to cover less realistic, corner cases in using SQL.) Second, we evaluated SQLSynthesizer on 5 SQL problems collected from online help forums, and tested whether SQLSynthesizer can synthesize desirable SQL queries for those real problems.

As a result, SQLSynthesizer successfully synthesized XXX out of XXX textbook exercises and solved 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.

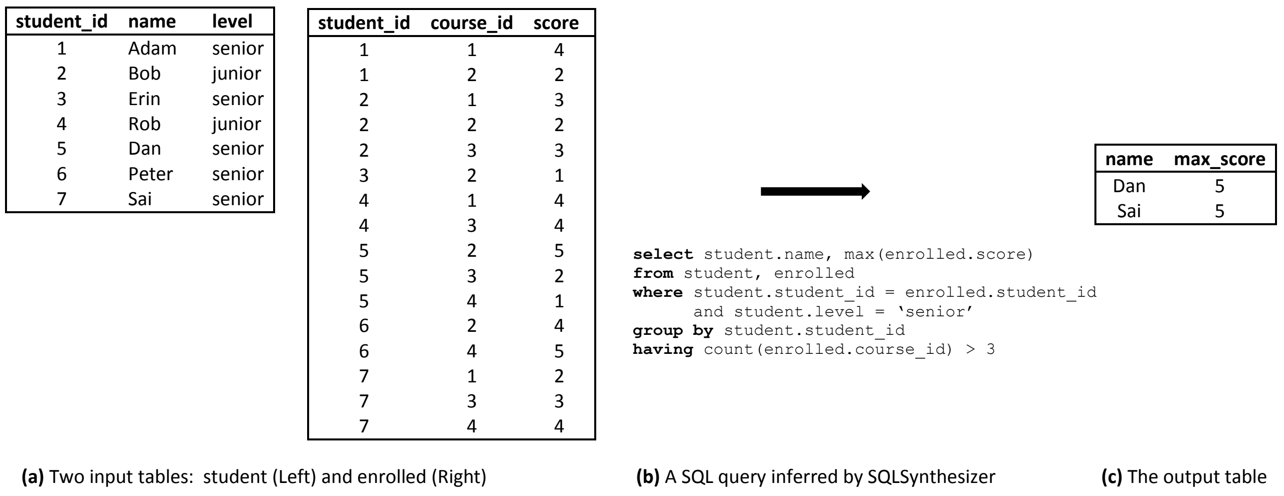
E. *Contributions*

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 SQL exercises and XXX SQL-related problems from online forums. The results show that SQLSynthesizer can synthesize a wide range of SQL queries. (Section VI).

II. Illustrating Example

We use the example in Figure 1 (described below) to illustrate the use of SQLSynthesizer. This example is adapted from a SQL exercise from a classic database textbook [22] (Chapter 5, Exercise 1), and has been simplified for illustration purpose.



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) > 3
```

name	max_score
Dan	5
Sai	5

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

Fig. 1. Example input-output tables and the SQL query sythensized by SQLSynthesizer. In this example, users provide SQLSynthesizer with two input tables (shown in (a)) and an output table (shown in (c)); SQLSynthesizer automatically infers a SQL query (shown in (b)) that transforms the two input tables into the output table.

Find the name and the maximum course score of all senior students who are enrolled in more than 2 courses. [1]

The desirable SQL query, as SQLSynthesizer inferred in Figure 1(b), first joins the `student` and `enrolled` tables, then groups by the joined table by the `student_id` column and selects students enrolled in more than 3 courses (the `having` statement). After that, this query further selects students whose level is `senior` and uses the `max` aggregator function to compute the maximum course score.

Despite the simplicity of the problem description, to write the desirable SQL query, end-users must manually connect their understanding of the task with specific SQL language features that can be used to fullfil the task. Such a process can be non-trivial for novice users. In addition, to the best of our knowledge, none of the existing SQL synthesis techniques [3], [4], [26], [28] can help uers write the above query**[[right place?]]**

As illustrate in Figure 1, an alternative approach to write this query is to provide SQLSynthesizer with some representative input-output examples; and let SQLSynthesizer automatically automatically infer the query. **[[transition]]**

III. Supported SQL Subset

The full SQL language[2] contains **[[.intractable.]]** keywords. It is infeasible to infer the whole language**[[revise]]**. **[[We must focus on]]** Thus, the first step is to identify a widely-used SQL subset using which a large class of query tasks can be performed. Unfortunately, no systematic study has ever been conducted to this end, and little empirical evidence has ever been provided on which SQL features are important in

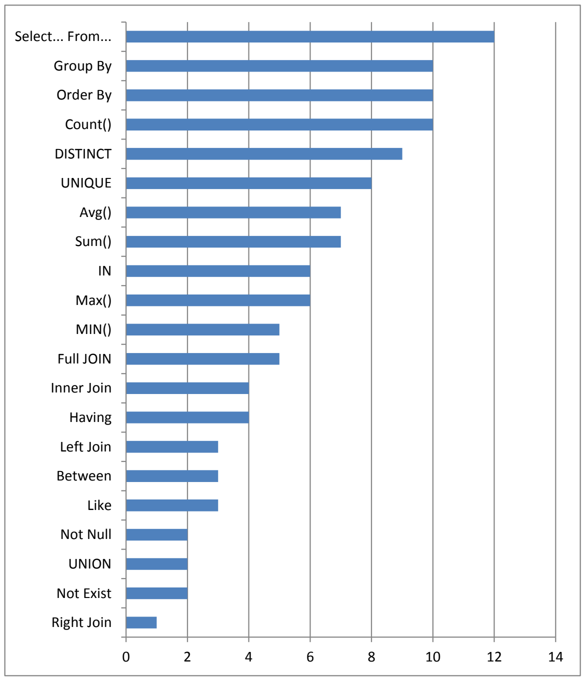


Fig. 2. Survey results of the most important SQL features in writing a database query. There were 12 participants in the survey, and each participant was asked to select the top 10 most important SQL features. SQL features with no selection **[[x]]** are omitted for brevity.

practice. Without such empirical knowledge, deciding which SQL subset to support in SQLSynthesizer remains difficult.

[[Existing work some existing work, that the author decided the language subset]]

To address this issue, we first conducted an online survey to ask IT professionals about the most important and widely-used SQL features in writing database queries (Section III-A). Then, based on the survey results, we designed a SQL subset that supports many database query tasks in practice

[1]Shown in Figure 1, the `student` table contains three columns: `student_id`, `name`, and `level`. Table `enrolled` contains three columns: `student_id`, `course_id`, and `score`.

[2]Here, we refer to the latest SQL 93 standard [].

```
 ⟨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⟩
             | NOT EXIST (⟨query⟩)
    ⟨op⟩ ::=   = | > | <
 ⟨cexpr⟩ ::=   const | ⟨column⟩
  ⟨expr⟩ ::= ⟨cexpr⟩ | count(⟨column⟩)
             | sum(⟨column⟩) | max(⟨column⟩) | min(⟨column⟩)
```

Fig. 3. Syntax of the supported SQL subset in SQLSynthesizer. This subset covers the top XXX **[[refer to figure 2]]** in Figure 2

(Section III-B). After that, we sent the designed SQL subset to the survey participants and conducted a series of follow-up email interviews to confirm our design.

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 most important 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 survey.

We sent out invitation to graduate mailing lists 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

We now present a SQL subset that can express database query tasks required by real users. Figure 3 shows the language syntax.

The supported SQL subset is a subset of the SQL 93 language, and shares the same semantics with the standard SQL language. Even though this subset cannot describe all query tasks against a database, it covers XXX out of XXX **[[features]]**. In particular, this SQL subset significantly enriches the SQL features used in existing query inference work [4]. **[[xx]]** It supports database query across multiple tables, conjuction of query conditions **[[others...]]** joining operations across multiple tables, and includes widely-used database operations such as `group by`, `order by`, and `having`, as well as a few common aggregation functions such as `count`, `sum`, `max`, and `min`.

When designing this SQL subset, we used the following rationale. First, we only considered standard SQL features, and excluded some vendor-specific features, such as the `top` keyword in Microsoft SQLServer. Second, we **[[discard fuzzy matching, like, untractable, string manipulation]]**. Third, we exclude a few SQL features that can have equivalent replacement **[[xx]]** in the current SQL subset. For example, the `Not Null` (followed by a column name) keyword can be simply replaced by a query condition to check the column is not `Null`. Nested queries are also omitted, since it can be re-written using conditions **[[xx]]**. Fourth, we exclude less useful **[[features]]** 5. omit between, discard Left Joint, Right Join, only remaining Full Join

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 the tailored SQL subset. Participants were first asked to rate the expressiveness of the SQL subset in Figure **??** 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 rate it 5, or 4. Only one participant rates it 3. He mistakenly understood....**[[]]**

One participant complained the SQL subset lacks the support of column re-naming

Based on the feedback, we think the language subset is sufficient ...

IV. Technique

This section first explains SQLSynthesizer's 3 steps, as introduced in Section I and illustrated in Figure 4; and then gives SQLSynthesizer's full algorithm in Section IV-D

A. Query Skeleton Creation

In this step, SQLSynthesizer performs a simple scan over the provided input-output examples to capture the basic structure of the desirable SQL query.

A query skeleton is an incomplete SQL query, consisting of three basic parts: the table set used in the query, the table columns used to join tables, and the table columns used to project the query results. Besides, other query parts, such as query conditions and aggregation operators, are left as unknown and will be determined later.

We next describe how SQLSynthesizer determines the table set, joining columns, and projection columns used in a query.

Determining the Table Set. End-users are unwilling to provide more than enough inputs. Thus, every table in the input example is expected to be used in the desirable SQL query. Based on this observation, we assume that every input table should be used *as least* once in the result query. By

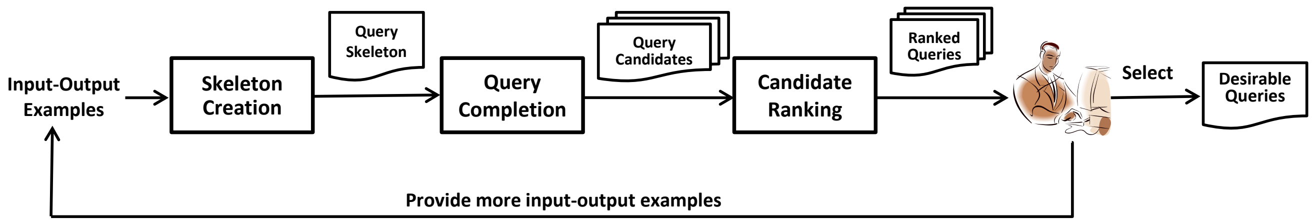


Fig. 4. Illustration of SQLSynthesizer's workflow in inferring SQL queries from input-output examples. SQLSynthesizer consists of 3 steps: (1) The "Skeleton Creation" step (Section IV-A) infers a partially-complete SQL query as a skeleton from the given examples, (2) The "Query Completion" step (Section IV-B) uses a rule-learning algorithm [] and type-directed search to produces a list of syntactically-valid query candidates that satisfy the provided input-output examples. , and (3) The "Candidate Ranking" step (Section IV-C) ranks all inferred SQL query candidates and provides users a ranked list of SQL queries with the likely ones on the top. Users select the desirable SQL queries from the produced ranked query list. On some examples, 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 provide more informative examples and then refine the SQL queries.

default, the table set contains all given input tables. However, it is possible that one input table will be used for multiple times. SQLSynthesizer does not forbid this case, rather, it uses a heuristic to estimate the table set: if the *same* column from an input table appears more than once in the example output, we add the input table to the table set the same number of times.

[[we view it as a strong indicator that this table will be joined multiple times and add it to our table set T using an alias]]

Determining Joining Columns. Given a set of tables, there are many ways to join them. Enumerating all possibilities can introduce a huge number of joining conditions and would quickly become intractable. To prune the search space, we use three simple but effective effective rules. First, tables are often joined on their primary keys with the same data types, such as joining the `student` table with the `enrolled` table on the `student_id` column (Figure 1). By contrast, it is unlikely to join two tables with an Integer column and a String column. Second, tables are often joined on columns with the same name, such as joining the `student` table with the `enrolled` table on the `student_name` column. Third, it is only meaningful to join two tables on columns that have the same data type and some overlapped values.

SQLSynthesizer restricts the search space in uses the above three rules **[[need to revise]]**

[[need to implement above.]] [[mention how many skeletons will be created]]

Determining Output Columns. For each column in the output table, SQLSynthesizer checks whether its name appears in any of the input table. If so, SQLSynthesizer uses the matched column from the input table as the output column. to the output column set. Otherwise, the output column must be produced by using aggregation operators. **[[same names]]** Consider the example in Figure 1, SQLSynthesizer determines that column `name` comes from the `student` table, while column `max_Score` must be created by using an aggregation operator. **[[If there is no column name]] [[check the values in the output column]]**

[[It is possible that multiple skeleton can be created. add an algorithm here.]]

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

Fig. 5. The SQL skeleton created for the motivating example in Figure 1.

Figure 5 shows the created query skeleton for the motivating example in Figure 1. In this skeleton, three unknown structures represented by <Aggregation> or <Conditions> are in red, and will be filled in the next phase. **[[revise text]]**

[[how to create group by]]

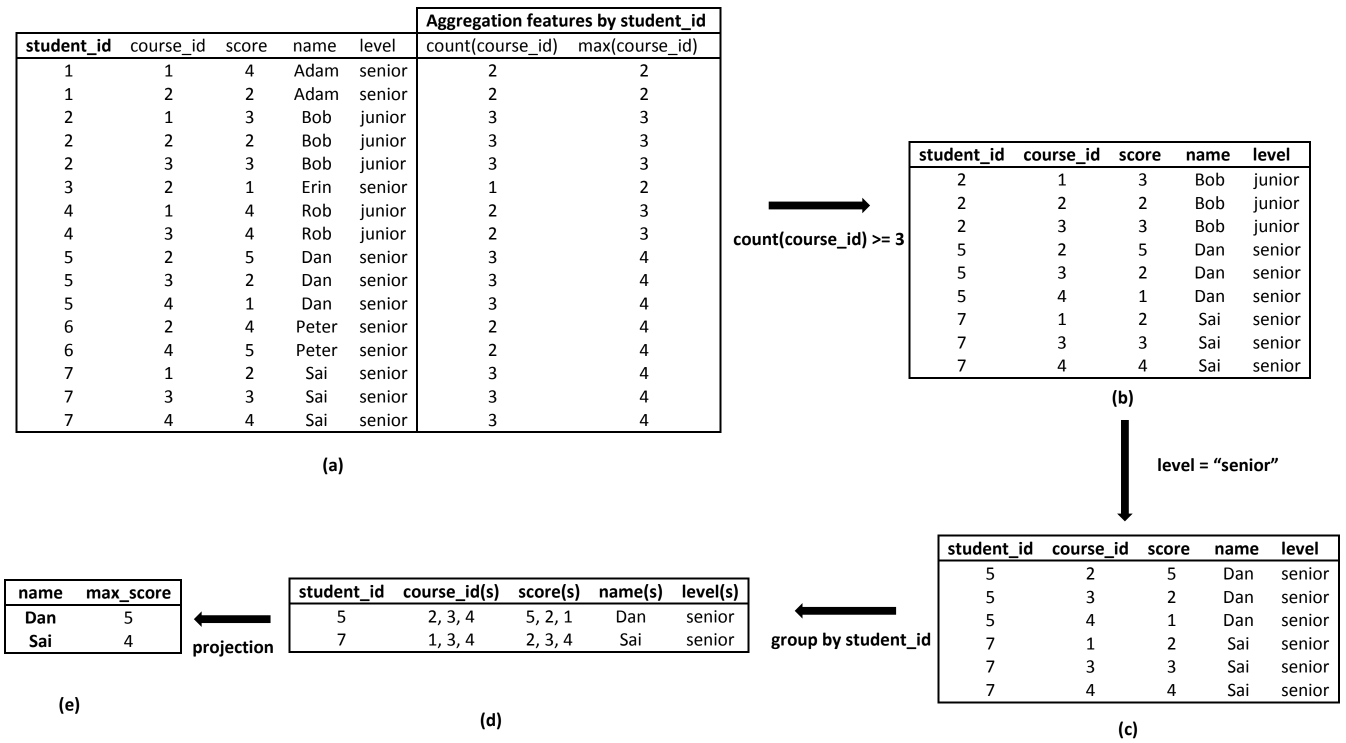
B. SQL Query Completion

In this step, SQLSynthesizer takes as input a query skeleton, infers the missing parts, namely query conditions (Section IV-B1), aggregates (Section IV-B2), and order-by clauses (Section IV-B3); and then produces a set of syntactically-correct SQL queries.

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

SQLSynthesizer uses the PART learning algorithm [8] from the machine learning community to infer a set of accurate and expressive rule set. Compared to other well-known learning algorithms like decision-tree learning or **[[add]]**, PART has two notable features. **[[unclear]]** First, it utilizes the "divide-and-conquer" strategy to repeatedly build rules and remove the instances it covers until no examples are left. When creating each rule, it employs a pruned decision tree built from current set of instance and only makes the leaf with the largest coverage into the resulting rules, without keeping the whole learned tree in memory.

A key challenge in using the PART algorithm is how to devise meaningful features. Existing approaches [] simply uses concrete values in a tuple as features. However, doing so loses



(a)

student_id	course_id	score	name	level	Aggregation features by student_id	
					count(course_id)	max(course_id)
1	1	4	Adam	senior	2	2
1	2	2	Adam	senior	2	2
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	2
4	1	4	Rob	junior	2	3
4	3	4	Rob	junior	2	3
5	2	5	Dan	senior	3	4
5	3	2	Dan	senior	3	4
5	4	1	Dan	senior	3	4
6	2	4	Peter	senior	2	4
6	4	5	Peter	senior	2	4
7	1	2	Sai	senior	3	4
7	3	3	Sai	senior	3	4
7	4	4	Sai	senior	3	4

(b)

student_id	course_id	score	name	level
2	1	3	Bob	junior
2	2	2	Bob	junior
2	3	3	Bob	junior
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

(c)

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

(d)

student_id	course_id(s)	score(s)	name(s)	level(s)
5	2, 3, 4	5, 2, 1	Dan	senior
7	1, 3, 4	2, 3, 4	Sai	senior

(e)

name	max_score
Dan	5
Sai	4

Fig. 6. Illustration of xxx

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

Fig. 7. The generated aggregation features for a table with 3 columns: C_1, C_2, and C_3, in which columns C_1 and C_2 are String type and column C_3 is Integer type.

much useful structure information needed in a SQL query. **[[not enough, no support for max, min, sum]]** Thus, we must transform tuples into appropriate feature representation.

SQLSynthesizer addresses this problem by encoding two types of additional features for each tuple:

- **Aggregation Features**. For each column, SQLSynthesizer adds Aggregation features are the aggregation results grouped by each `String` type column over every other columns. Table 7 shows an example. (1) Aggregation, including `COUNT`, `MAX`, `MIN` and `AVG`, whose results might be used in query condition. **[[cannot infer group by column1, column2]]**
- **Comparison Features**. Comparison feature is the result of comparing two comparable columns. We consider two possible values: $\{1, 0\}$, which represents means whether two columns under comparison satisfy the predicate or not, respectively. Comparison results between two comparable columns. The above two additional knowledge encoding permits our technique to make use of correlations between columns, rather than only values from each isolated and sequential columns. Table 8 shows an example.

Combining using concrete tuple values, aggregation features,

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

Fig. 8. The generated comparison features for a table with 4 columns: C_1, C_2, C_3, and C_4, in which columns C_1 and C_2 are String type, and columns C_3 and C_4 are Integer type. Columns with the same type are comparable, such as C_1 and C_2, and C_3 and C_4.

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

[[should give an example here of why such feature is useful]]

The PART algorithm returns rules representing three types of features. **[[which 3 types?]] [[move below text to the example figure]]** For example, it returns the following rules for our motivating example in Section II:

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

SQLSynthesizer next splits the returned rules into two parts: the query selection condition, and the having condition. SQLSynthesizer treats rules derived from the value features as the query conditions (`COUNT(enrolled.course_id) > 2` for the example of Figure 1), and rules derived from the aggregation features as the having condition (`&& student.level ='senior'` for the example of Figure 1). **[[explain why?]]**

2) Searching for Aggregates: For each column produced by an aggregation operator, the whole search space includes all possible combinations of table columns and the five supported aggregation operators (see Figure 3). SQLSynthesizer leverages the following two observations to further reduce the search space:

- The data type of an output column must be compatible with the aggregation operator's return type. For instance, if an output column has the String type, it must not use aggregation operators (e.g., `count` and `sum`) that returns an Integer.
- If an arithmetic aggregation operator, such as `max` and `min`, is used, each value in the output column must has appeared in the input table.

3) Searching for Order By Columns: After the query conditions and aggregates are inferred, SQLSynthesizer observes data values in each output table column. If the data values in a column appear are sorted, SQLSynthesizer append the column name to the `Order By` clause.

C. Query Candidate Ranking

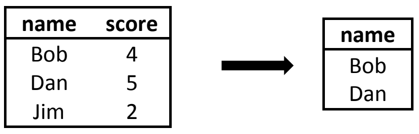
It is possible that multiple SQL queries satisfying the given input-output examples will be returned. This may adversely impact end-users who want to write simple query tasks but now may require to provide more bits for disambiguation of their intent (which manifests in the need to provide more examples and more rounds of interaction). To alleviate this problem, we employ the Occam's razor principle, which states that the simplest explanation is usually the correct one, to rank the more likely queries higher in the output list. We define a comparison scheme between different SQL queries by defining a partial order between them. Some of these choices are subjective, but have been observed to work well.

A SQL query is simpler than another one if it uses smaller number of query conditions (including `having` and `order by`) or the expressions in each query condition are pairwise simpler. 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 by summarizing the text length of each query condition used. Figure 9 shows an example.

D. Put It All Together

Figure 10 summarizes the full algorithm in sythesizing SQL queries from input-output examples. SQLSynthesizer first creates a set of query skeletons based on the given input-output examples (line 2). Then, for each query skeleton, it searches for the query conditions (line 4), aggregates (line 5), and order-by columns (line 6) to construct a list of syntactically-correct SQL queries (line 7). SQLSynthesizer further validates each constructed SQL query on the given example input tables to verify whether it can produce the same example output table (line 9). If so, the constructed query is added to the result list



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 > 3
2. select name from student where name = 'Bob'
                                   or name = 'Dan'
```

Fig. 9. Illustration of SQLSynthesizer's query candidate ranking strategy. SQLSynthesizer produces two queries for the given input-output examples. Based on our heuristic (Section IV-C), the first query differs from the second query by using simpler conditions, and thus ranks higher (and is less likely to overfit the given examples).

```
Input: example input table(s) $T_I$, example output table $T_O$
Output: a ranked list of SQL queries
sythesizeSQLQueries($V_{old}$, $V_{new}$, $T$)
1:  queries ← an empty list
2:  skeletons ← createQuerySkeletons($T_I$, $T_O$)
3:  for each skeleton in skeletons do
4:      conditions ← inferQueryConditions($T_I$, $T_O$, skeleton)
5:      aggregates ← searchForAggregates($T_I$, $T_O$, skeleton, conditions)
6:      orderByColumns ← searchForOrderByColumns($T_O$, skeleton, aggregates)
7:      queryCandidates ← constructQueries(skeleton, conditions, aggregates, orderByColumns)
8:      for each query in queryCandidates do
9:          if validateOnDatabase(mathitquery, $T_I$, $T_O$) then
10:             queries.add(query)
11:         end if
12:     end for
13: end for
14: queries ← rankdQuery(queries)
15: return queries
```

Fig. 10. Algorithm for sythesizing SQL queries from input-output examples.

(line 10). Finally, the query list is sorted by using the Occam's razor principle (line 14), and returned to the end-users.

[[write an algorithm here?]]

E. Discussion

We next discuss some design issues in SQLSynthesizer.

Soundness and Completeness. The SQLSynthesizer technique is neither sound nor complete. The primary reason is that the Query Skeleton Creation step (Section IV-A) in SQLSynthesizer is based on heuristics. **[[such heuristics approximate the NP hard problems.]]** Thus, SQLSynthesizer cannot gurantee to infer correct SQL queries in all cases. Despite such limitations, as demonstrated in Section VI, SQLSynthesizer is still useful in sythesizing many SQL queries in practice.

What if the input-output examples are not representative enough? On some examples, SQLSynthesizer may produce a SQL query that satisfies the input-output examples specified by the user, but does not address the intention that the user wants.

To address this issue, we use a simple interaction model [14] to ask users to investigate the results of an output SQL query and report any discrepancy. After, the user can refine the inferred SQL query by providing a more informative input-output example that demonstrate the behavior on which the originally-inferred SQL query behaves incorrectly. As demonstrated in our evaluation (Section VI), such an interactive model works well in practice: **[[a few interations]]**

V. Implementation

We implemented the proposed technique in a tool, called SQLSynthesizer. SQLSynthesizer uses the Weka toolkit [13] to implement the rule-learning algorithm described in Section IV-B. **[[inspect the above]]**

SQLSynthesizer uses MySQL [21] as the backend database to check the correctness of each inferred SQL query. Specifically, SQLSynthesizer 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.

VI. Evaluation

We evaluate SQLSynthesizer's effectiveness in sythesizing SQL queries. SQLSynthesizer's effectiveness can be reflected by:

- the success ratio of sythesizing a variety of SQL queries (Section VI-C1).
- the time cost of sythesizing SQL queries (Section VI-C2).
- the human efforts to write input-output examples (Section VI-C3).
- comparison with a previous SQL query inference technique (Section VI-C4).

A. Benchmarks

We collected benchmarks from two sources:

- We picked up *all* SQL exericses (XXX in total) from a classic database textbook [22]. Exercises from a textbook are good resource to evaluate SQLSynthesizer's generality, since such exericses are often designed to cover a wide range of SQL features. Some exericses are even designed on purpose to cover less realistic, corner cases in using SQL. When writing **[[exclude sql update, delte]]**
- We searched SQL query-related questions raised by real-world database users from 3 popular online forums [5], [25], [27]. **[[focus on standard features, some of features, such questions reflect how ppl use sql in practice]]** As of April 2013, we collected XXX forum questions related to writing a SQL query.

B. Evaluation Procedure

For each textbook exericse and forum question, we use SQLSynthesizer to solve it. If an exericse or problem has been associated with input-output examples, we directly apply SQLSynthesizer on those existing examples. Otherwise, we manually write some examples based on our understanding. To reduce the bias in writing examples, all examples are writting by a graduate student from University of Washington other than SQLSynthesizer's developers.

We checked SQLSynthesizer's correctness by comparing its output with the desired SQL queries. **[[some exericses have answers, but some are not. If in different forms, we checked it semantic equivalence.]]**

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 reapplied SQLSynthesizer to the new input. We repeated this process until our tool inferred a desirable SQL query.

Our 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]]

We use a real SQL question from an online forum[3] to illustrate SQLSynthesizer's effectiveness. The question was started by a novice user, who needed help to write a SQL query to get result from three input tables. In this question, the novice user described his required query in a few paragraphs of English, but also include several small, representative input-output examples as shown in Figure 12, to better express his intention. This question receives no replies as of April 2013 and we speculated that writing a SQL query to join three tables to produce certain output results is non-trivial.

We ran SQLSynthesizer on the input-output examples in Figure 12. The tool produced 6 valid answers in less than 1 minutes, all of which satisfy the given examples. The highest ranked SQL query is shown in Figure 12, which is quite unintuitive to write. The SQL query in Figure 12 first joins three input tables on columns `T1.Column2`, `T2.Column2`, `T2.Column1`, and `T3.Column1` using some selected columns, and then aggregates the results based on column `Table2.Column3`'s value. Finally, it returns the minimal values of columns `T1.Column1`, `T1.Column4`, and `T3.Column2` from each aggregated group as the results.

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 interative round. Most of the time is spent querying the backend database to validate the correctness of each sythesized SQL query.

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

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							

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 sythesizing 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.

Column1	Column2	Column3	Column 4
101	2001	3020	01-01-11
101	2001	3002	02-01-11
101	2001	3001	03-01-11
102	2002	3002	01-01-11

Column1	Column2	Column 3
20011	2001	200131
20012	2001	200132
20013	2001	200133

Column1	Column 2
20011	Site
20012	Site
20013	Site

(a) Three input tables: T1 (top), T2 (left), and T3 (right)

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

(b) A SQL query inferred by SQLSythesizer

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

(c) The output table

Fig. 12. Input-output examples ((a) and (c)) taken from an online SQL help forum thread. SQLSynthesizer automatically sythensizes 6 SQL queries that can produce the output table from the three input tables. (b) shows the highest ranked 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 sythesize 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 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 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 sythesize desirable SQL queries for other benchmarks, because it **[[the reasons]]**.

D. Initial Experience

We deployed the beta-test version of SQLSynthesizer to a small number of developers and have been using it ourselves, and refining it, since early May 2012. Designing and deploying SQLSynthesizer, along with feedback from the handful of users, has helped us to better understand the issues and to improve the tool's design.

Here is one example piece of feedback from an external user, via private communication:

[[some quote from users]]

Prior to developing SQLSynthesizer, we studied over 100 forum posts about common problems about writing correct SQL queries. **[[most of them use examples]]** We anticipate that future user studies will identify additional strengths and weaknesses that will allow us to further improve SQLSynthesizer.

E. Experimental Discussion

Limitations. The experiments indicate several limitations of our technique. First, XXX Second, XXX.. **[[fill above]]**

Threats to Validity. There are three major threats to validity in our evaluation. First, the XXX textbook exercises and XXX forum problems may not be representative. Thus, we can not claim the results can be generalized to an arbitrary scenario. Second, we only compared SQLSynthesizer with **[[comparison results]]**. Using other query inference and recommendation techniques [] might achieve different results. Third, **[[about user study]]**

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)** SQLSynthesizer can efficiently synthesize desirable SQL queries on **[[how many]]** for both textbook exercises and online forum problems, with a small amount of human effort small input-output examples. **(3)** SQLSynthesizer produces better results than existing techniques []. **[[edit]]**

VII. Related Work

This section discusses two categories of closely-related work on reverse query processing and automated program synthesis.

A. Reverse Query Processing

Reverse query processing is a well-known problem in the database community [4], [26], [28]. Zloof's work on *Query by Example* (QBE) [28] provided a high-level query language and an intuitive form-based Graphical User Interface for writing database queries. To use the QBE system, users must formulate their queries with the provided language and fill in the appropriate skeleton tables on its interface. **[[exactly what we need to avoid.]]**

Tran et al. [26] proposed a technique, called *Query By Output* (QBO), to infer SQL queries from input-output examples. **[[say]]** However, their techniques can only infer simple select-project-join queries, and such queries can only be applied to satisfy the example in **[[some example]]**, and such queries cannot be applied to many of the other real-world cases that we studied (Section **??**). The key limitation of the QBO technique is that it only considers table values in the original input tables as features **[[need to read]]**

Recently, Sarma et al. [4] studied the *View Definitions Problem* (VDP). With a rather different goal than SQLSynthesizer, VDP aims to 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 1 input table with no joins nor projections. In addition, the main contribution of Sarma et al's work is the complexity analysis of three variants of the view definitions problem, and there are no tool implementationand empirical study for their proposed technique.

[[what is the limitation of non program sythesis, why sythensis would be useful.]]

B. Automated Program Synthesis

Program synthesis [10] aims 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. Users then manually define tools for data in the format. Similarly, the Wrangler tool, developed in the HCI community, provides a nice visual programming-by-demonstration interface to table transformations for data cleaning [16]. Theses techniques, though well-suited for their intended tasks, are insufficient for the database query tasks. For example, while text extraction and mass editing are quite valuable for data transformation, the above tools lack other needed operations such as table joinning, aggregation, and query condition inference. **[[explain]]**

Harris and Gulwani described a system for learning excel spreadsheet transformations macros from an example input-output pair [14]. Given input and output tables, 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 abstractions and semantics than the SQL language. An excel macro can express a variety of table re-shaping operations (e.g., **[[an example]]**), but are not suitable to formulate database queries. Second, Harris and Gulwani's approach treats table cells as atomic units, and thus has limited expressiveness. It can only transforms one table to another, but does not support queries involving table joining, aggregation, or column projection. **[[takk avbout the example]]**

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 saved query is also registered as a view, allowing other users to compose and reuse. SnipSuggest [17] is a SQL autocompletion system. As a user typs 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. Compard to SQLSynthesizer, both SQLShare and SnipSuggest hypothesize that existing query logs (by the user herself or other users) contains valuable information, as a user articulates an increasingly larger fragment of a query. However, such query logs are often not available to most users. SQLSynthesizer eliminates this assumption by directly asking users to draw the input-output examples. **[[the difficult of start writing queries.]]**

Cheung et al [3] presented a technique to infer SQL queries from imperative code. Their technique automatically 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 by re-writing fragments of logic code into SQL queries, rather than helping novice end-users write correct SQL queries from scratch. In addition, **[[focus on performance...]]**

VIII. Conclusions 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) that can infer desirable SQL queries that fulfill end-users' intentions. Our experimental results show that SQLSynthesizer is effective in automating a variety of database query tasks with small input-output examples.

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

Our future work will concentrate on the following topics:

Enrich the supported SQL subset. We plan to enrich the supported SQL subset by SQLSynthesizer, and design a corresponding algorithm to synthesize more general queries.

Illustration of synthesis steps. Besides producing a final result, end-users may also be interested in knowing how a SQL query is inferred step by step. Showing detailed inference steps not only makes SQLSynthesizer more usable, but also permits end-users to better understand the whole process and spot possible errors earlier. We plan to apply recent advance in data visualization [16] to the context of program synthesis.

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

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.