Automatically Synthesizing SQL Queries from Input-Output Examples

Sai Zhang
University of Washington
Joint work with: Yuyin Sun
Goal: making it easier for non-expert users to write correct SQL queries

- Non-expert database end-users
  - Business analysts, scientists, marketing managers, etc.

This paper: bridge the gap!
Find the name and the maximum course score of each student enrolled in more than 1 course.

The correct SQL query:

```
SELECT name, MAX(score)
FROM student, enrolled
WHERE student.stu_id = enrolled.stu_id
GROUP BY student.stu_id
HAVING COUNT(enrolled.course_id) > 1
```
Existing solutions for querying a database

• General programming languages
  + powerful
  − learning barriers

• GUI tools
  + easy to use
  − limited in customization and personalization
  − hard to discover desired features in complex GUIs
Our solution: programming by example

Table: student

<table>
<thead>
<tr>
<th>name</th>
<th>stu_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
</tr>
<tr>
<td>Dan</td>
<td>4</td>
</tr>
</tbody>
</table>

Table: enrolled

<table>
<thead>
<tr>
<th>stu_id</th>
<th>course_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>504</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>505</td>
<td>99</td>
</tr>
<tr>
<td>2</td>
<td>504</td>
<td>96</td>
</tr>
<tr>
<td>3</td>
<td>501</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>502</td>
<td>88</td>
</tr>
<tr>
<td>3</td>
<td>505</td>
<td>68</td>
</tr>
</tbody>
</table>

Output table

<table>
<thead>
<tr>
<th>name</th>
<th>MAX(score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>100</td>
</tr>
<tr>
<td>Charlie</td>
<td>88</td>
</tr>
</tbody>
</table>

SELECT name, MAX(score)
FROM student, enrolled
WHERE student.stu_id = enrolled.stu_id
GROUP BY student.stu_id
HAVING COUNT(enrolled.course_id) > 1
How do end-users use SQLSynthesizer?

Real, large database tables → Desired output result

Small, representative
Input-output examples

SQLSynthesizer

SQL?
SQLSynthesizer’s advantages

• Fully automated
  – Only requires input-output examples
  – No need of annotations, hints, or specification of any form

• Support a wide range of SQL queries
  – Beyond the “select-from-where” queries [Tran’09]
Outline

• Motivation

• A SQL Subset
  • Synthesis Approach
  • Evaluation
  • Related Work

• Conclusion
Designing a SQL subset

- 1000+ pages specification
- PSPace-Completeness [Sarma’10]
- Some features are rarely used

SQLSynthesizer’s focus: a widely-used SQL subset
How to design a SQL subset?

• Previous approaches:
  – Decided by the paper authors [Kandel’11] [Tran’09]

• Our approach:
  – Ask experienced IT professionals for the most widely-used SQL features
Our approach in designing a SQL subset

1. **Online survey**: eliciting design requirement
   - Ask each participant to select *10 most widely-used* SQL features
   - Got 12 responses

2. **Designing** the SQL subset
   Supported SQL features
   
   1) `SELECT.. FROM...WHERE`
   2) `JOIN`
   3) `GROUP BY / HAVING`
   4) Aggregators (e.g., MAX, COUNT, SUM, etc)
   5) `ORDER BY`
   
   Supported in the previous work [Tran’09]

3. **Follow-up interview**: obtaining feedback
   - Ask each participant to rate the *sufficiency* of the subset
   
   ![Rating Scale]
   
   Average rating: 4.5
   
   Not sufficient at all
   
   Completely sufficient
Our approach in designing a SQL subset

1. **Online survey**: eliciting design requirement
   - Ask each participant to select 10 most widely-used SQL features
   - Got 12 respondents

1. **Designing** the SQL subset
   - Supported SQL features:
     - SELECT... FROM... WHERE
     - JOIN
     - GROUP BY / HAVING
     - ORDER BY
     - Aggregators (e.g., MAX, COUNT, SUM, etc)
   - Supported in the previous work [Tran’09]

2. **Follow-up interview**: obtaining feedback
   - Ask each participant to rate the sufficiency of the subset
   - Average rating: 4.5

The SQL subset is enough to write most common queries.
Outline

• Motivation
• Language Design
• Synthesis Approach
• Evaluation
• Related Work
• Conclusion
SQLSynthesizer Workflow

Select the desired query, or provide more examples

Input tables  Output table
**SQLSynthesizer Workflow**

Select the desired query, or provide more examples

Input - Output Examples

SQLSynthesizer

Examples

Input tables

**A SQL query**

Output table
**SQLSynthesizer Workflow**

Select the desired query, or provide more examples

Input-Output Examples → SQLSynthesizer → Queries

- Combine
- Filter
- Project

Input tables → Output table
SQLSynthesizer Workflow

A complete SQL:

```
SELECT name, MAX(score)
FROM student, enrolled
WHERE student.stu_id = enrolled.stu_id
GROUP BY student.stu_id
HAVING COUNT(enrolled.course_id) > 1
```
Multiple solutions

Input tables

Query 1
Query 2
Query 3
...

Output table
Computes all solutions, ranks them, and shows them to the user.
Key techniques

1. **Combine:**
   Exhaustive search over legal combinations
   (e.g., cannot join columns with different types)

2. **Filter:**
   A machine learning approach to infer query conditions

3. **Project:**
   Exhaustive search over legal columns
   (e.g., cannot apply \texttt{AVG} to a string column)
Learning query conditions

Cast as a rule learning problem:

**Finding rules that can perfectly divide a search space into a positive part and a negative part**

- Positive part: Rows contained in the output table
- Negative part: The rest of the rows
- Search space: All rows in the joined table
Search space: the joined table

Table: student

<table>
<thead>
<tr>
<th>name</th>
<th>stu_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
</tr>
<tr>
<td>Dan</td>
<td>4</td>
</tr>
</tbody>
</table>

Table: enrolled

<table>
<thead>
<tr>
<th>stu_id</th>
<th>course_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>504</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>505</td>
<td>99</td>
</tr>
<tr>
<td>2</td>
<td>504</td>
<td>96</td>
</tr>
<tr>
<td>3</td>
<td>501</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>502</td>
<td>88</td>
</tr>
<tr>
<td>3</td>
<td>505</td>
<td>68</td>
</tr>
</tbody>
</table>

The joined table

<table>
<thead>
<tr>
<th>Name</th>
<th>stu_id</th>
<th>course_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>504</td>
<td>100</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
<td>505</td>
<td>99</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>504</td>
<td>96</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>501</td>
<td>60</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>502</td>
<td>88</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>505</td>
<td>68</td>
</tr>
</tbody>
</table>

Join on the stu_id column (inferred in the Combine step)
Finding rules selecting rows contained in the output table

The joined table

<table>
<thead>
<tr>
<th>name</th>
<th>stu_id</th>
<th>course_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>504</td>
<td>100</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
<td>505</td>
<td>99</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>504</td>
<td>96</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>501</td>
<td>60</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>502</td>
<td>88</td>
</tr>
<tr>
<td>Charlie</td>
<td>4</td>
<td>505</td>
<td>68</td>
</tr>
</tbody>
</table>

Output table

<table>
<thead>
<tr>
<th>name</th>
<th>MAX(score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>100</td>
</tr>
<tr>
<td>Charlie</td>
<td>88</td>
</tr>
</tbody>
</table>
Finding rules selecting rows containing the output table

The joined table

<table>
<thead>
<tr>
<th>name</th>
<th>stu_id</th>
<th>course_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>504</td>
<td>100</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
<td>505</td>
<td>99</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>504</td>
<td>96</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>501</td>
<td>60</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>502</td>
<td>88</td>
</tr>
<tr>
<td>Charlie</td>
<td>4</td>
<td>505</td>
<td>68</td>
</tr>
</tbody>
</table>

rules?

<table>
<thead>
<tr>
<th>name</th>
<th>stu_id</th>
<th>course_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>504</td>
<td>100</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
<td>505</td>
<td>99</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>501</td>
<td>60</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>502</td>
<td>88</td>
</tr>
<tr>
<td>Charlie</td>
<td>4</td>
<td>505</td>
<td>68</td>
</tr>
</tbody>
</table>
Finding rules selecting rows containing the output table

The joined table

<table>
<thead>
<tr>
<th>name</th>
<th>stu_id</th>
<th>course_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>504</td>
<td>100</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
<td>505</td>
<td>99</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>504</td>
<td>96</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>501</td>
<td>60</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>502</td>
<td>88</td>
</tr>
<tr>
<td>Charlie</td>
<td>4</td>
<td>505</td>
<td>68</td>
</tr>
</tbody>
</table>

No good rules!
Solution: computing additional features

- Key idea:
  - **Expand** the search space with additional features
  - **Enumerate** all possibilities that a table can be aggregated
  - **Precompute** aggregation values as features

Suppose grouping it by **stu_id**

<table>
<thead>
<tr>
<th>name</th>
<th>stu_id</th>
<th>course_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>504</td>
<td>100</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
<td>505</td>
<td>99</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>504</td>
<td>96</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>501</td>
<td>60</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>502</td>
<td>88</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>505</td>
<td>68</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>505</td>
<td>88</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>505</td>
<td>88</td>
</tr>
</tbody>
</table>

The joined table
Finding rules *without* additional features

The joined table

<table>
<thead>
<tr>
<th>name</th>
<th>stu_id</th>
<th>course_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>504</td>
<td>100</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
<td>505</td>
<td>99</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>504</td>
<td>96</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>501</td>
<td>60</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>502</td>
<td>88</td>
</tr>
<tr>
<td>Charlie</td>
<td>4</td>
<td>505</td>
<td>68</td>
</tr>
</tbody>
</table>

No good rules!
Finding rules with additional features

The joined table after the table is grouped by **stu_id**

<table>
<thead>
<tr>
<th>name</th>
<th>stu_id</th>
<th>course_id</th>
<th>score</th>
<th>COUNT(course_id)</th>
<th>MIN(score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>504</td>
<td>100</td>
<td>2</td>
<td>99</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
<td>505</td>
<td>99</td>
<td>2</td>
<td>99</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>504</td>
<td>96</td>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>501</td>
<td>60</td>
<td>3</td>
<td>60</td>
</tr>
<tr>
<td>Charlie</td>
<td>3</td>
<td>502</td>
<td>88</td>
<td>3</td>
<td>60</td>
</tr>
<tr>
<td>Charlie</td>
<td>4</td>
<td>505</td>
<td>68</td>
<td>3</td>
<td>60</td>
</tr>
</tbody>
</table>

**COUNT(course_id) > 1**

(after grouping by **stu_id**)

```
SELECT   name, MAX(score)
FROM     student, enrolled
WHERE     student.stu_id = enrolled.stu_id
GROUP BY  student.stu_id
HAVING    COUNT(enrolled.course_id) > 1
```
Ranking multiple SQL queries

- Occam’s razor principle: rank simpler queries higher
  - A simpler query is less likely to overfit the examples

- Approximate a query’s complexity by its text length

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>Bob</td>
<td>20</td>
<td>99</td>
</tr>
<tr>
<td>Charlie</td>
<td>30</td>
<td>99</td>
</tr>
</tbody>
</table>

Input table: student

Output table

Query 1: `select name from student where age < 30`
Query 2: `select name from student where name = 'Alice' || name = 'Bob'`
Outline

• Motivation
• Language Design
• Synthesis Approach
• Evaluation
• Related Work
• Conclusion
Research Questions

- **Success ratio** in synthesizing SQL queries?
- What is the **tool time cost**?
- How much **human effort** is needed in writing examples?
- **Comparison** to existing techniques.
Benchmarks

• **23** SQL query related exercises from a classic textbook
  – *All* exercises in chapters 5.1 and 5.2

• **5** forum questions
  – Can be answered by using standard SQL
    (Most questions are vendor-specific)
  – 2 questions contain example tables
Evaluation Procedure

- Rank of the correct SQL query
- Tool time cost
- Manual cost
  - Example size, time cost, and the number of interaction rounds
  (All experiments are done by the second author)
Results: success ratio

The correct query ranks 1st in all succeeded questions

Fail on 8 questions
Succeed on 20 questions

Fail on 8 exercises
Succeed on 15 exercises

Succeed on 5 forum questions

Require writing **sub-queries**, which are not supported in SQL Synthesizer
Result: tool time cost

- On average, 8 seconds per benchmark
  - Min: 1 second, max: 120 seconds
  - Roughly proportional to the #table and #column
Results: manual cost

- **Example size**
  - 22 rows, on average (min: 8 rows, max: 52 rows)

- **Time cost in writing examples**
  - 3.6 minutes per benchmark, on average
    (min: 1 minute, max: 7 minutes)

- **Number of interaction rounds**
  - 2.3 rounds per benchmark, on average
    (min: 1 round, max: 5 rounds)
Comparison with an existing approach

- Query-by-Output (QBO) [Tran’09]
  - Support simple “select-from-where” queries
  - Use data values as machine learning features

Table: student

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>Bob</td>
<td>20</td>
<td>99</td>
</tr>
<tr>
<td>Charlie</td>
<td>30</td>
<td>80</td>
</tr>
</tbody>
</table>

Output table

```
select name from student
where age < 30

name
Alice
Bob
```

```
select age, max(score) from student
group by age

<table>
<thead>
<tr>
<th>age</th>
<th>MAX(score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>30</td>
<td>80</td>
</tr>
</tbody>
</table>
```
- Many realistic SQL queries use aggregation features.
- Users are unlikely to get stuck on simple “select-from-where” queries.
Experimental Conclusions

- Good success ratio (71%)
- Low tool time cost
  - 8 seconds on average
- Reasonable manual cost
  - 3.6 minutes on average
  - 2.3 interaction rounds
- Outperform an existing technique
  - Success ratio: QBO (7%) vs. SQLSynthesizer (71%)
Outline

• Motivation
• Language Design
• Synthesis Approach
• Evaluation
• Related Work
• Conclusion
Related Work

• **Reverse engineering SQL queries**

  Query-by-Examples [Zloof’75]
  
  *A new GUI with a domain-specific language to write queries*

  Query-by-Output [Tran’09]
  
  *Uses data values as features, and supports a small SQL subset.*

  View definition Synthesis [Sarma’10]
  
  *Theoretical analysis, and is limited to 1 input/output table.*

• **Automated program synthesis**

  PADS [Fisher’08], Wrangler [Kandel’11], Excel Macro [Harris’11],

  SQLShare [Howe’11], SnippetSuggest [Khoussainova’11],

  SQL Inference from Java code [Cheung’13]

  – *Targets different problems, or requires different input.*
  – *Inapplicable to SQL synthesis*
Outline

• Motivation
• Language Design
• Synthesis Approach
• Evaluation
• Related Work
• Conclusion
Contributions

• A programming-by-example technique
  – Synthesize SQL queries from input-output examples
  – Core idea: using machine learning to infer query conditions

• Experiments that demonstrate its usefulness
  – Accurate and efficient
    • Inferred correct answers for 20 out of 28 SQL questions
    • 8 seconds for each question
  – Outperforms an existing technique

• The SQLSynthesizer implementation
  [http://sqlsynthesizer.googlecode.com](http://sqlsynthesizer.googlecode.com)
[Backup Slides]
The most widely-used SQL features

### Number of votes

- **SELECT... FROM..**: 21 features
- **GROUP BY**: The standard `select .. from.. where.. feature`
- **ORDER BY**: Aggregation features
- **COUNT**: Joining features
- **DISTINCT**: Existential features
- **AVG**: Value matching features
- **SUM**:
- **IN**:
- **MAX**:
- **MIN**:
- **FULL JOIN**:
- **INNER JOIN**:
- **HAVING**:
- **LEFT JOIN**:
- **BETWEEN**:
- **LIKE**:
- **NOT NULL**:
- **UNION**:
- **NOT EXIST**:
- **RIGHT JOIN**
**Design a SQL subset**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Covered features</th>
<th>Uncovered features</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT... FROM..</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>COUNT</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>AVG</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>SUM</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>IN</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>MAX</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>MIN</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>FULL JOIN</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>INNER JOIN</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>HAVING</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>LEFT JOIN</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>LIKE</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>NOT NULL</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>UNION</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>NOT EXIST</td>
<td>🟢</td>
<td>🔴</td>
</tr>
<tr>
<td>RIGHT JOIN</td>
<td>🟢</td>
<td>🔴</td>
</tr>
</tbody>
</table>

**Covered 15 features**

- SELECT... FROM..
- GROUP BY
- ORDER BY
- COUNT
- DISTINCT
- AVG
- SUM
- IN
- MAX
- MIN
- FULL JOIN
- INNER JOIN
- HAVING
- LEFT JOIN
- BETWEEN
- LIKE
- NOT NULL
- UNION

**Sub-query**

**Special joins**

**Wildcard matching**