# Interactive Query Synthesis from Input-Output Examples

Chenglong Wang, Alvin Cheung, Rastislav Bodik
University of Washington
{clwang, akcheung, bodik}@cs.washington.edu

## ABSTRACT

This demo showcases SCYTHE, a novel query-by-example system that can synthesize expressive SQL queries from input-output examples. SCYTHE is designed to help end-users program SQL and explore data simply using input-output examples. From a web-browser, users can obtain SQL queries with SCYTHE in an automated, interactive fashion: from a provided example, SCYTHE synthesizes SQL queries and resolves ambiguities via conversations with the users.

In this demo, we first show SCYTHE how end users can formulate queries using SCYTHE; we then switch to the perspective of an algorithm designer to show how SCYTHE can scale up to handle complex SQL features, like outer joins and subqueries.

## 1. INTRODUCTION

Emerging online open databases, e.g., Google Scholar, Microsoft Academics, and data collecting tools [1] make data more accessible than ever before. Such tools benefit not only software developers but also many end-users like business analysts and data scientists. However, when the high-volume data collected with these tools is stored in relational databases, end-users must master the skill of writing SQL queries to retrieve or analyze data, which is considered challenging [8, 7, 5]. The challenges mainly result from the fact that the solutions to many common tasks (e.g., ArgMax, moving average) involve complex SQL features, like subqueries and aggregation.

Many approaches have been developed over the past decades to address this end-user programming problem: they provide users with alternate interfaces to construct SQL to ease the programming process. Two emerging ones are the query-by-natural-language (QBNL) interface [5] and the query-by-example (QBE) interface [7, 8].

While these systems can solve many practical problems, their expressiveness remains relatively limited for many analytical tasks, e.g., computing moving averages, removing duplicates, and calculating ArgMax. This is because: (1) many analytical tasks are difficult to describe concisely and unambiguously using pure natural language, (2) while examples are often considered expressive for specifying tasks, existing QBE systems do not scale sufficiently for complex queries, and (3) user-supplied specifications are typically highly ambiguous, making disambiguating candidate queries a highly nontrivial task.

Our system, SCYTHE, with innovated algorithms, is capable of synthesizing expressive SQL queries from input-output (I/O) examples without suffering from the preceding limitations. SCYTHE has the following features:

1. *An efficient synthesis algorithm.* SCYTHE's design includes a new synthesis algorithm that efficiently synthesizes complex queries from I/O examples. The improvement of algorithm scalability lets SCYTHE support a larger set of SQL grammar, including nested subqueries, outer joins, and unions. Besides, SCYTHE can synthesize many complex queries from I/O examples in just a few seconds. To gain these improvements, our algorithm separates the synthesis process of filter predicates (in `Where`, `On` clauses) from that of higher level query operators, like `Join` and `Group By`. This separation brings us new opportunities to optimize the solvers for subproblems.

2. *Conversational disambiguation.* Since I/O examples are incomplete specifications of user tasks, the synthesizer may return multiple semantically different queries consistent with the given examples. When such ambiguities arise, SCYTHE carefully crafts a new input example and solicits the desired output from the user. The new I/O pair let SCYTHE disambiguate previously generated queries.

3. *Visualization*: SCYTHE contains a visualization module to run synthesized queries on the database and build visualizations. These visualizations aims to help end-users interpret synthesized queries.

In the following, we first overview our system (Section 2) and then demonstrate its features (Section 3).

## 2. SYSTEM OVERVIEW

Figure 1 shows the architecture of SCYTHE, which consists of four modules: (1) the user interface, (2) the synthesizer, (3) the disambiguation module, and (4) the database connector. We introduce the workflow of SCYTHE in this section and leave algorithm details to Section 3.



*SIGMOD'17, May 14 - 19, 2017, Chicago, IL, USA*



ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3058738

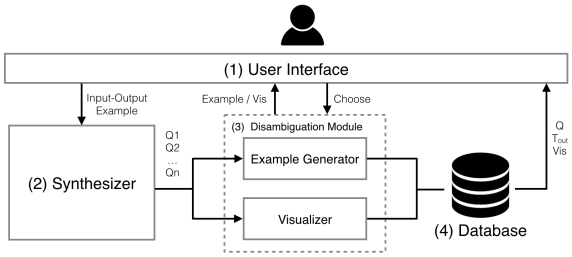


**Figure 1: The Overview of Scythe.**

A user starts a query formulation task through Scythe's web-based interface (Figure 2) by submitting an I/O example. The input example consists of a set of tables $I = \{T_1, ..., T_m\}$ and the output example is a table $T_{\text{out}}$. Scythe seeks to synthesize a SQL query $Q$ such that returns $T_{\text{out}}$, when executed on $I$. The schema of these input tables should reflect those of the database tables.

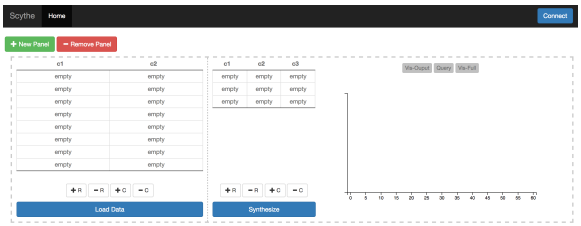


**Figure 2: The Scythe Interface.**

Given the I/O example, the *synthesizer* searches for consistent queries within the SQL grammar and ranks results based on simplicity and naturalness. Note that while all synthesized queries behave the same on the provided example, they may behave differently when executed on the database since the user's example incompletely specifies the task.

When ranking cannot disambiguate synthesized queries, the *disambiguation module* interacts with the user. Given the top $n$ synthesized queries ($n$ being a user-configured parameter), the *example generator* computes a new small input example $I'$ on which the $n$ queries produce different results (such $I'$ is called an *distinguishing input* for the queries). Tables in $I'$ have the same schema as those in user provided input example $I$ but have different contents. The disambiguation module then asks the user to choose the corresponding solution for $I'$, and Scythe uses the provided choice to attempt to resolve the ambiguity. During the this phase, the *visualizer* generates visualizations of query results on the database to assist the user to understand the queries, and users can also choose the desired results based on these different visualizations.

After the resolution of all ambiguities, Scythe runs the synthesized query on the database. The query, its results, and the visualization of the result are returned to the user through the interface.

# 3. DEMONSTRATED FEATURES

Our Scythe demonstration contains three parts: (1) solving Stack Overflow problems, (2) exploring Google Scholar data, and (3) illustrating Scythe's synthesis algorithms.

## 3.1 Solving Stack Overflow Problems

The first example asks Scythe to solve SQL programming problems for end users as an alternative to the use of Stack Overflow. Specifically, we demonstrate how to use Scythe to solve a popular post (with over 500,000 views and 550 upvotes) about the "arg-max" problem collected from Stack Overflow.

Example[1]. Having a table with columns `id`, `rev` and `content`, the user wants to select rows containing the maximum `rev` for each `id`. Besides a brief task description, the user provided the following I/O example in the post.

Input:

| id | rev | content |
|---|---|---|
| 1 | 1 | A |
| 2 | 1 | B |
| 1 | 2 | C |
| 1 | 3 | D |

Output:

| id | max_rev | content |
|---|---|---|
| 1 | 3 | D |
| 2 | 1 | B |

*Solution.* A forum expert provided the following query (Q1) to solve the problem:

```
-- Q1
Select a.id, b.max_rev, a.contents
From (Select id, Max(rev) As max_rev
      From input
      Group By id) b
Inner Join input As a
On a.id = b.id And a.rev = b.max_rev
```

This solution first uses a subquery to calculate the maximum `rev` for each `id` and then joins the result with the `input` table to get `content` values associated with maximum `rev` values for each row.

Instead of writing a post and spending time conversing with forum experts to disambiguate the question, we can solve the problem using Scythe. To use Scythe, we feed the I/O example in Example through the interface and invoke the synthesizer. Scythe then returns all queries that are consistent with the example. Incorrect queries, such as the two below, may also be included since the example is not a complete specification.

```
-- Q2
Select b.*
From (Select id, Max(rev)
      From input
      Group By id) AS a
Join (Select * From input
      Where id <> rev) AS b
Where a.max_rev = b.rev
```

```
-- Q3
Select b.*
From (Select id, Max(rev)
      From input
      Group By id) AS a
Join input AS b
Where a.max_rev = b.rev
  And a.id <= b.id
```

Scythe heuristically ranks synthesized queries based on *simplicity* and *naturalness* (how often operators in the synthesized query come together in real-world cases), and it eliminates low scored queries (e.g., those with 3 joins are less likely to be the correct solution for this example since much simpler, more natural queries have already been found). The synthesizer then forwards remaining queries with similarly high scores (i.e., the correct answer and the preceding two incorrect queries) to the example generator for disambiguation. The example generator produces the following distinguishing input (left). The right hand table shows the results of $Q_1, Q_2, Q_3$ on the new example (note that they behave the same on the original input example).

[1]http://stackoverflow.com/questions/7745609.

**Figure 3: Exploring Google Scholar data.**

| id | rev | content |
|---|---|---|
| 1 | 1 | A |
| 2 | 1 | B |
| 1 | 2 | C |
| 1 | 3 | D |
| 2 | 3 | E |

$Q_1$ :

| | | |
|---|---|---|
| 1 | 3 | D |
| 2 | 3 | E |

$Q_2$ :

| | | |
|---|---|---|
| 1 | 3 | D |
| 2 | 3 | E |
| 1 | 3 | D |
| 2 | 3 | E |

$Q_3$ :

| | | |
|---|---|---|
| 1 | 3 | D |
| 1 | 3 | D |
| 2 | 3 | E |

Since only the $Q_1$ output matches the task specification above, we select it for disambiguation. As a result, SCYTHE returns $Q_1$ as the final result.

**What will visitors see and do?** The preceding is but one sample scenario of how visitors can interact with SCYTHE. We will prepare 10 additional problems collected from the most recent/top-rated Stack Overflow posts for the visitors to try. In addition, visitors can also try their own I/O examples.

## 3.2 Exploring Google Scholar Data

Our second demonstration showcases SCYTHE's ability to help users with data exploration tasks. In particular, we pose a question previously studied by Chasins et al. [2], *"What is the distribution of career peaking times for computer scientists?"*. We use Google Scholar data to answer the question: the data is a table of schema `(author, paper, year, citation)` with ∼3.5 million tuples, collected using the web scraping tool Ringer [1].

The "peak time" for researchers can be defined as "the year one's most cited paper is published.", and the task can be formulated as computing the distribution of the number of years it takes for researchers to publish the most cited paper since the beginning of their careers.

To solve this task, we feed the input-output example shown in Figure 3 (1) to SCYTHE. The output example contains three columns: name of the author (`c1`), the year first paper was published (`c2`), and the year most cited paper was published (`c3`). SCYTHE synthesized query Q4 (shown below) from the example: the first `Join` calculates the year with maximum citation and the second one add the year of the first published paper to the table. We omit the disambiguation phase for this example since it is the same as the one in the previous example.

```
-- Q4
Select a.name, c.min_year, b.year
From (Select name, Max(citation)
      From T Group By name) AS a
Join T As b
Join (Select name, Min(year)
      From T Group By name) AS c
Where a.max_citation = b.citation
  And a.name = b.name And a.name = c.name
```

To visualize the distribution, we choose SCYTHE's histogram with the $x$-axis mapping to `c3 - c2` and the $y$-axis mapping to the count. The chart, shown in Figure 3-1 (right), shows that most researchers publish their most cited papers within the first 20-30 years of their careers, which could initially suggest that researchers peak early in their career.

However, the conclusion may be imprecise: the "peak time" alone does not reflect the relative position of the peak overall in the career span of the researchers. Thus, fewer researchers peak later (in 50-60 years) may be caused by the fact that fewer people have such long career spans.

To understand the distribution of researcher career length, we provide a new I/O example to the system. The input example is the same as the previous one, but the output example is different: it contains columns including (1) names of authors, (2) the year their first paper was published, and (3) the year their last paper was published (see Figure 3-2 left). We set the visualization histogram to have the $x$-axis map to the career length and the $y$-axis map to the count of the given career length.

The solution for the problem include the query Q5 below its corresponding visualization in Figure 3-2 (right). From the visualization, we learn that more people have a career span of 20-30 years than 50-60 years, indicating that the previous answer is misleading.

```
-- Q5
Select name, Max(year), Min(year)
From T Group By name
```

To better answer the question, we combine the two queries shown before to build a new histogram that shows both peak time and career length. While making a new example to capture both values can be difficult (for both SCYTHE and the user) given the complexity of the task, we make the new example from the two previous output examples, and SCYTHE reuses the previously synthesized queries to gain efficiency.

The final example is shown in Figure 3-3 (left). The input example contains columns (1) name, (2) the year fist paper is published, (3) the year the last paper was published, (4) the year most cited paper was published, (5) the career length, and (6) the peak time. The output example is a table containing the count for each career length - peak time combination.

```
-- Q6
Select career_len, peak_time, Count(*)
From (Select a.name, min_year, max_year, mc_year,
             max_year - min_year As career_len,
             mc_year - min_year As peak_time
      From Q4 As a Join Q5 As b
      Where a.name = b.name)
Group By career_len, peak_time
```

Query $Q_6$ shows the solution: it first combines $Q_4$ and $Q_5$ by joining the career length and peak time for each researcher; it then calculates the number for each combination with an aggregation. The visualization of the query result, shown in Figure 3-3 (right), uses the pre-built histogram for 3-dimensional data with $x$-axis mapped for the length of the career, $y$-axis mapped to the peak time and the grayscale of the circles shows the count.

From this third diagram, we learn an interesting fact: researchers' peak time grows as their career lengthens, with many exceptions for individuals. This conclusion matches that by Chasins et al. [2].

**What will visitors see and do?** In this part, we will preload the Google Scholar data into SCYTHE and demonstrate the example described. We then ask visitors to solve other tasks, e.g., the distribution of the number of paper published each year.

## 3.3 Inside Scythe

In this part, we showcase how SCYTHE's internal components works in synthesizing queries from I/O examples.

**The Synthesizer.** The synthesis algorithm is the key to SCYTHE's improvements over prior QBE systems. As previously noted, SCYTHE decomposes the query synthesis problem into two parts: (1) synthesizing query skeletons (queries whose predicates are placed as holes, denoted as "`??`"), and (2) synthesizing filter predicates to fill these holes. Given an example $(I, T_{\text{out}})$, SCYTHE first enumerates all query skeletons that can be constructed from $I$ without considering filter predicates in `Where`, `On`, or `Having` clauses. We refer to these skeletons as abstract queries. Below is the abstract query corresponding to $Q_1$ shown in Section 3.1.

```
-- Abstract Q1
Select a.id, b.max_rev, a.contents
From (Select id, Max(rev) As max_rev
      From input Where ??
      Group By id Having ??) b
Inner Join (Select * From input Where ??) As a
On ??
```

When SCYTHE finishes enumerating all abstract queries, it estimates whether whether each abstract query can possibly be instantiated into the output example $T_{\text{out}}$ by filling certain predicates into the holes, and SCYTHE prune away all those cannot after estimation. A simple way is to prune abstract queries whose schema do not match $T_{\text{out}}$.

In the second phase, our algorithm attempts to synthesize predicates for all remaining queries. Since the skeletons are fixed, predicate synthesis can be optimized. One optimization involves grouping predicates that behave same on the skeleton into equivalence classes so that we need not search over all syntactically different predicates but only those behaves differently. For example, two predicates "`id = rev`" and "`id <= rev`" behave the same on the input example in EXAMPLE (Section 3.1); therefore, we need not keep both during the search process.

As a result, the overall synthesis algorithm is greatly accelerated, which lets SCYTHE to support a wider range of SQL queries without compromising performance. In our experiment of SCYTHE on 193 benchmarks collected from Stack Overflow, SCYTHE can solve 143 of them, most within seconds. Visitors will experience the performance and expressiveness in the demo session.

**The Disambiguation Module.** The example generator that computes distinguishing input for top-ranked queries is layered on the COSETTE SQL solver [3]. Given two queries, COSETTE either computes a distinguishing input for them when they are not equivalent or generates a proof of equivalence. SCYTHE invokes COSETTE for each pair of the top-ranked queries to find the distinguishing input. Visualizations in SCYTHE are prebuilt using D3[2], and users can change parameters to customize data visualizations.

**What will visitors see?** We will demonstrate in detail our synthesis algorithm on EXAMPLE in Section 3: how abstract queries are enumerated, how undesirable ones are pruned away, how predicates are synthesized, and how distinguishing input is computed. We will show how the example is solved in a pipeline with intermediate results.

# 4. RELATED SYSTEMS

QBO [7], QFE [6] and SQLSynthesizer [8] are previous QBE systems built for end-user SQL programming. SCYTHE differs from them in the synthesis algorithm and the disambiguation module with wider range of SQL support. Nalir [5] and Nlyze [4] are QBNL systems for translating texts to SQL queries. QBNL systems are better in Q&A style questions while QBE systems are in analytical tasks; combining two approaches together is part of our future work.

# 5. CONCLUSION

SCYTHE is a novel QBE system that can synthesize highly expressive SQL queries from I/O examples. SCYTHE is automated and interactive, and our demo showcases how SCYTHE can help end users solve real-world data analytics and exploration tasks.

**Acknowledgement** This work is supported in part by NSF Grants CCF-1139138, CCF-1337415, CNS-1563788, ACI-1535191 and IIS-1546083, a Grant from U.S. DOE under Award Number FOA-0000619, Grants from DARPA FA8750-14-C-0011, FA8750-16-2-0032, and FA8750-16-2-0032, DOE award DE-SC0016260, as well as gifts from Google, Adobe, Amazon, Intel, Mozilla, Nokia, and Qualcomm.

# 6. REFERENCES

[1] S. Barman, S. Chasins, R. Bodik, and S. Gulwani. Ringer: web automation by demonstration. In *SPLASH*, pages 748–764. ACM, 2016.

[2] S. Chasins, S. Barman, R. Bodik, and S. Gulwani. Browser record and replay as a building block for end-user web automation tools. In *WWW*, pages 179–182. ACM, 2015.

[3] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for sql. 2017.

[4] S. Gulwani and M. Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*, pages 803–814. ACM, 2014.

[5] F. Li and H. V. Jagadish. Nalir: an interactive natural language interface for querying relational databases. In *SIGMOD*, pages 709–712. ACM, 2014.

[6] H. Li, C.-Y. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *Proceedings of the VLDB Endowment*, 2015.

[7] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD '09*, USA. ACM.

[8] S. Zhang and Y. Sun. Automatically synthesizing sql queries from input-output examples. In *ASE*. IEEE, 2013.

[2] http://d3js.org/.