Synthesizing SQL Data Modification Queries from Input-Output Examples

Luca Schroeder Ezra Zigmond
Harvard University
{schroeder, ezigmond}@college.harvard.edu

Abstract

Users of relational databases often have trouble constructing SQL queries to perform their desired tasks. However, users are often able to provide input-output examples that provide a partial specification for their desired outcome. Accordingly, prior work has developed programming-by-example systems to help users craft database `SELECT` queries. A natural and useful extension of this technique is to automatically synthesize queries that modify data (that is, `UPDATE` and `DELETE` queries), as even users who are proficient in writing `SELECT` queries may struggle to write these modification queries.

In this paper, we present the first system for synthesizing SQL data modification queries from input-output examples. Our key insight is that it is possible to solve SQL data modification query synthesis problems by issuing multiple, independent calls to an existing `SELECT` synthesizer, which we can treat as a black box. This results in a simple synthesis algorithm that will benefit from future advances in `SELECT` synthesis technology. We have implemented our algorithm in a new tool called REAPER[1] which can solve a variety of interesting update problems in a few seconds.

1. Introduction

SQL is a widespread domain-specific language used to manipulate relational data. In a recent survey of developers conducted by Stack Overflow, SQL was the second-most used programming language among respondents, with over 51% of developers using SQL [19]. SQL has significantly different syntax than popular general-purpose programming languages and is primarily declarative rather than procedural. SQL-specific concepts and idioms (for example, joins, groupings, and nested queries) present a further barrier to learning SQL. Moreover, even a user who is proficient in issuing SQL `SELECT` queries may struggle to write correct `UPDATE` queries if they wish to modify existing data. This can be seen by examining Stack Overflow questions tagged "sql-update": users who appear competent in writing `SELECT` queries often provide either a syntactically invalid `UPDATE` query that they are unsure how to fix or just an example of a query that selects the "new" data they wish to replace some existing data with.[2] Further, experimenting with a possibly incorrect `UPDATE` presents a risk of irreparably corrupting data unlike testing out a `SELECT` query.

Though users may struggle to write SQL queries, they can usually provide input-output (I/O) examples to specify their intent. Recent work on programming-by-example (PBE) tools for SQL query synthesis [22, 24] have attempted to ease the pain of writing SQL `SELECT` queries. These systems solicit I/O examples from users and return queries satisfying the examples so that users need not learn the complexities of writing SQL queries themselves. We believe that this approach can be extended to helping users construct `UPDATE` and `DELETE` queries, which previous work has not addressed.

Our key insight is that we can treat an existing SQL `SELECT` synthesis system as a *black box* and use it to implement an algorithm for synthesizing `UPDATE` and `DELETE` queries from I/O examples. This approach has two distinct advantages: 1) implementing or modifying our algorithm does not require a deep knowledge of program synthesis techniques and 2) improvements to the quality of the synthesizer are orthogonal to our work and should automatically improve the quality of our results.

Based on this insight, we can decompose the `UPDATE` synthesis problem into two independent steps. First, we use a `SELECT` synthesizer to search for a predicate that correctly classifies the rows in the table based on whether or not they should be updated (that is, the `WHERE...` clause in an `UPDATE` query). Then, in the second step, we search for terms in the `SET` clause that will assign the correct new values to the rows (again using a `SELECT` synthesizer). The `DELETE` synthesis problem is a smaller version of this problem where we only need to learn a classifying predicate. On top of this, we implement several heuristics for manipulating the inputs to and outputs from the `SELECT` synthesizer which we have found significantly improve the quality of the final synthesis output.

We have implemented our algorithm and instantiated it with Scythe [22] as the underlying `SELECT` synthesizer in a PBE tool we call REAPER. We have evaluated REAPER qualitatively and found that it is able to solve a variety of interesting update problems in a matter of seconds. To the best of our knowledge, REAPER is the first PBE tool that supports synthesizing SQL data modification queries.

To recap, our work makes the following contributions:

- We describe an algorithm that decomposes the problem of SQL `UPDATE` and `DELETE` synthesis into multiple, independent calls to a `SELECT` synthesizer, which we treat as a black box.
- We present heuristics which we use to improve the quality and expressiveness of queries returned by a `SELECT` synthesizer.
- We implement our algorithm and heuristics as REAPER and qualitatively demonstrate that it is useful for synthesizing correct, readable `UPDATE` and `DELETE` queries.

2. Overview

We begin with a high-level description of our algorithm, motivating our approach through an illustrative example.

SQL Modification Queries. All synthesized `UPDATE` and `DELETE` queries are of the form given in Figure 1. Non-standard features such as multi-table modification queries (allowed in MySQL) and

[1] Our code can be found at https://github.com/ezig/Reaper

[2] e.g. https://stackoverflow.com/questions/34554337/

```
UPDATE t              DELETE t
SET s1, ..., sm       WHERE p
WHERE p
```

Figure 1. The grammar of synthesized SQL data modification queries; t is a table name, the s_i are SET clauses, and p is a predicate.

$$\texttt{SET}\text{ clause } s := (col = e)$$
$$e := const \mid col \mid (\texttt{SELECT}\dots)$$

Figure 2. A SET clause is the assignment of a column to a constant value, another column, or the result of a nested query.

use of the FROM clause in UPDATE queries (allowed in SQL Server) are not supported. Although these restrictions limit expressiveness, the queries we synthesize will be compatible with nearly all major database management systems (DBMSs). As many DBMSs are in common use [19], we believe such broad compatibility is important.

Synthesizing an UPDATE query involves learning a number of SET clauses (see Figure 2) and a predicate p in the WHERE clause (we call this the *classifier predicate*). Synthesizing a DELETE query involves learning just a classifier predicate p. DELETE synthesis is thus a smaller problem than UPDATE synthesis. For clarity, this section will largely focus on UPDATE synthesis. Section 2.4 discusses how our approach extends immediately to synthesizing DELETE queries.

Problem Statement. In the style of Wang et al. [22], we formalize a user's query as a 5-tuple $(I, T_{update}, T_{out}, K, C)$, where: $I = \{T_1, \dots, T_n\}$ is the set of input tables; $T_{update} \in I$ is the table to be updated; T_{out} is the desired state of T_{update} after the UPDATE query; K is a function from column names in T_{update} to sets of constant values; and C is a set of constants relevant to the classifier predicate. Conceptualizing a SQL data modification query as a function from sets of tables to sets of tables, the objective is to synthesize a query q such that $q(I) = q(\{T_1, \dots, T_{update}, \dots, T_n\}) = \{T_1, \dots, T_{out}, \dots, T_n\}$, with the constraint that all constant values used in q must have been provided by the user. In particular, we require that all constants that appear in the classifier predicate p must be from C and all constants in a SET clause $col = e$ must be from $K(col)$.

Although the requirement that the user provide all constants may initially seem burdensome, we believe it is reasonable and necessary in our scenario for several reasons. First, our synthesizer is designed for users who roughly know the semantics of the query they want (e.g. "update salary for employees who have been at the firm for > 10 years") but do not how to express this query in SQL syntax. It is thus realistic to expect that the user knows the special values (e.g. '10 years') relevant to their query. Second, in many cases user-provided constants are necessary to resolve ambiguities in small input-output examples. For instance, if the user provides a two-row example in which a 13-year employee gets a raise but a 2-year employee didn't, the synthesizer has no way of knowing which constant value between 2 and 13 years is the intended cutoff point. Third, user-provided constants boost efficiency as the synthesizer only has to enumerate over the structure of queries and does not have to enumerate all possible constant values (which may be infinite). Fourth, many complex and interesting SQL data modification queries require little or no constants at all.

```
UPDATE COMPACT_DISCS
SET label_id = (SELECT CD_LABELS.label_id
 FROM CD_LABELS
 WHERE CD_LABELS.company = 'Sarabande')
WHERE COMPACT_DISCS.cd_id = 115.0
```

Figure 3. Our synthesizer's solution for Example 1.

Example 1. We use a simple example from an introductory SQL textbook [16] (Chapter 8, Exercise 6, modified for brevity) in this section. The example consists of two input tables: $T_{\text{CD\_LABELS}}$, which lists record labels; and $T_{\text{COMPACT\_DISCS}}$, which lists compact discs produced by these record labels. The scenario is as follows: the user has just inserted a CD ('Orlando') into $T_{\text{COMPACT\_DISCS}}$ but made a data entry error, assigning the CD to Capitol Records instead of Sarabande. The user would like to correct the error, and wants to modify the incorrect label_id without first looking up the correct label_id value for Sarabande. The user provides the relevant cd_id (115) as a classifier predicate constant, and the name of the correct record company ('Sarabande') as a constant that may be useful in synthesizing the SET clause for the column label_id.

Using the 5-tuple notation introduced above, the user's query is $(\{T_{\text{CD\_LABELS}}, T_{\text{COMPACT\_DISCS}}\}, T_{\text{COMPACT\_DISCS}}, T_{out}, K, \{115\})$, where $K(\text{label\_id}) = \{\text{'Sarabande'}\}$ and $K = \emptyset$ everywhere else.

$T_{\text{CD\_LABELS}}$

label_id	company
834	Reprise
835	Capitol
836	Sarabande

$T_{\text{COMPACT\_DISCS}} = T_{update}$

cd_id	cd_title	label_id
113	Blue	834
114	Fundamental	835
115	Orlando	835

T_{out}

cd_id	cd_title	label_id
113	Blue	834
114	Fundamental	835
115	Orlando	**836**

Solution. Figure 3 shows the correct solution generated by our synthesizer, which matches the textbook solution. The query updates the row in $T_{\text{COMPACT\_DISCS}}$ where cd_id $= 115$, setting the label_id to Sarabande's label_id using a nested query. Note that the use of constant values in the generated query matches the constraint described above.

2.1 Our Approach

There has been recent work [22, 24] on PBE tools for SQL query synthesis. However, all of these research efforts have focused exclusively on synthesizing SELECT queries. Our key insight is that the problem of UPDATE and DELETE query synthesis can be decomposed into smaller, independent subproblems that can each be solved using a SELECT synthesizer:

1. Synthesizing the classifier predicate p.
2. Synthesizing each of the SET clauses s_i.

The following two sections justify the claim that these subproblems are independent and sketch the necessary SELECT synthesizer calls. We assume the SELECT synthesizer accepts queries of the form (I, T_{out}, C), where I is again the set of input tables, T_{out} is the desired output table from the SELECT query, and C is a set of predicate constants. This matches the interface of Scythe [22], the tool we extend in this paper.

2.2 Synthesizing the Classifier Predicate

Using a simple row-by-row comparison between T_{update} and T_{out}, we can determine the set of rows in T_{update} that are modified by

```
SELECT *
FROM COMPACT_DISCS
WHERE COMPACT_DISCS.cd_id = 115.0
```

Figure 4. The top result from the classifier predicate subproblem call to the SELECT synthesizer, for Example 1.

the UPDATE query. We consider a row modified if one or more of its fields is different between T_{update} and T_{out}. We assume that there are no "coincidental non-updates"; that is, there are no rows that were meant to satisfy the classifier predicate but for which the "correct" SET clauses coincidentally act as an identity transformation. We view coincidental non-updates as an example quality and user interaction issue orthogonal to the main contributions of this paper.

Given the rows that are modified by the query, synthesizing the WHERE clause that selects these rows for the update and synthesizing the SET clauses that apply the correct transformation to these rows are clearly two independent problems.

We conceptualize a SQL table as a multiset of rows and let $T_{modified} \subseteq T_{update}$ and $T'_{modified} \subseteq T_{out}$ denote the modified rows before and after (respectively) the UPDATE query executes. In Example 1, these two tables both consist of a single row:

$T_{modified} \subseteq T_{update}$

cd_id	cd_title	label_id
115	Orlando	835

$T'_{modified} \subseteq T_{out}$

cd_id	cd_title	label_id
115	Orlando	836

We ask the SELECT synthesizer to produce a query whose result is $T_{modified}$, given the input tables I and the classifier predicate constants C; in other words, we give the SELECT synthesizer the query $(I, T_{modified}, C)$. We filter out results that are not of the form SELECT * FROM T_update WHERE p, and then "lift" the classifier predicate p from the top candidate returned by the SELECT synthesizer and use it in our own result. Figure 4 shows the top candidate returned for the appropriate call for Example 1; the classifier predicate we lift is highlighted in blue.

2.3 Synthesizing SET clauses

Under the SQL standard, columns referenced on the right-hand side of a SET clause (either directly, or in a nested SELECT query) are the columns *before* the larger UPDATE query takes place. The query UPDATE t SET col1 = col2, col2 = col1, for example, would swap the values in col1 and col2. SET clauses can thus be synthesized not only independently of the classifier predicate but also independently of one another.[3]

For each column col in the original table, we first check whether all the values of col in $T'_{modified}$ are equal to a constant value $k \in K(\texttt{col})$. If so, the SET clause is simply $\texttt{col} = k$. If not, we then check whether the col column in $T'_{modified}$ is equal to another column col2 in $T_{modified}$, in which case the SET clause is $\texttt{col} = \texttt{col2}$. If col2 is col, the SET clause is simply the identity and we omit it in our synthesized query for brevity. Otherwise, the right-hand side of the SET clause must be the result of a nested SELECT query, and we ask the SELECT synthesizer for that query by posing it the problem $(I, \pi_{\texttt{col}}(T'_{modified}), K(\texttt{col}))$, where $\pi_{\texttt{col}}(T'_{modified})$ denotes the table consisting of just the column col in $T'_{modified}$. For Example 1, the SET clause for columns cd_id and cd_title is the identity, and since the single value of label_id in $T'_{modified}$ does not equal any value in the single-row $T_{modified}$ or any constant in $K(\texttt{label\_id}) = \{115\}$, we must make a call to the SELECT synthesizer. The resulting SET clause is shown in Figure 3.

[3] In MySQL, the order of the SET clauses does matter—SET clauses are evaluated one at a time and "generally" in left-to-right order [17], meaning the swap query above would likely produce two identical columns. This behavior is a significant departure from the standard and we therefore do not support MySQL.

$$
\begin{array}{rcll}
T & ::= & \texttt{Table}(schema, content) & \text{(Table)} \\
schema & ::= & [c_1 : \tau_1, \ldots, c_m : \tau_m] & \text{(Schema)} \\
content & ::= & [r_1, \ldots, r_n] & \text{(Content)} \\
r & ::= & [v_1, \ldots, v_m] & \text{(Row)} \\
\tau & ::= & \texttt{int} \mid \texttt{double} \mid \texttt{string} & \text{(Type)} \\
 & \mid & \texttt{date} \mid \texttt{time} &
\end{array}
$$

Figure 5. The formalization of SQL tables, following Scythe [22]. c denotes a column name and v denotes a value.

One may wonder why our algorithm uses C and K rather than a single "bag" of constants. This choice is in part due to the design of state-of-the-art SELECT synthesizers. Scythe [22], the SELECT synthesizer we build on, for example ranks synthesized queries based on whether or not they use all of the user-provided constants. Passing every user-provided constant to the synthesis of a single nested SET clause could result in a convoluted query that attempted to use every constant and it is thus necessary to separate the constants relevant to each individual SELECT synthesizer call. We view this preference towards queries in which all user-provided constants are used not as a quirk of Scythe but as a desirable property for ranking algorithms in general, since queries that leave constants unused intuitively are not fully capturing the user's intent. Separation of constants allows the user to more precisely specify their input-output example and allows us to provide better results as the SELECT synthesizer is passed only the appropriate constants.

2.4 Synthesizing DELETE Queries

So far our discussion has focused on UPDATE query synthesis. The insights here are however immediately applicable towards DELETE synthesis. For deletes, a user's query is a 4-tuple $(I, T_{delete}, T_{out}, C)$, where $T_{delete} \in I$ is the table whose rows are being deleted. Note that no function K is needed here as only a classifier predicate is being synthesized.

We find the rows $T_{modified} \subseteq T_{delete}$ which are deleted in the output, i.e. $T_{modified} = T_{delete} - T_{out}$. As before, we then pass the query $(I, T_{modified}, C)$ to the SELECT synthesizer and lift the result's classifier predicate for our DELETE query.

3. SQL Language

Before turning to a complete treatment of our synthesis algorithm, we briefly introduce the definition of SQL tables used in this paper and the grammar of SQL data modification queries we are able to synthesize, in Figures 5 and 6 respectively. We follow Wang et al. [22] in both cases.

A table is a $(schema, content)$ tuple, where $schema$ associates types with columns and $content$ is a multiset of rows. Standard multiset binary operators such as $=$ (equality), $\subseteq$ (subset), $\cup$ (union), and $\setminus$ (set difference) are defined to compare the content of tables when the tables have equivalent schemas. These operators are undefined when the operands are two tables with different schemas.

Thanks to the expressiveness of Scythe [22], the SELECT synthesizer we extend for the implementation of our synthesis algorithm, we are able to handle a large subset of the SQL language. The grammar outlined in Figure 6 is straightforward; the only constructor that requires some explanation is $\texttt{Aggr}(\vec{c}, c', \alpha, i, p)$, which maps to the SQL query SELECT $\vec{c}, c'$ FROM i GROUP BY $\vec{c}$ Having p.

$$
\begin{array}{rlr}
q ::= & \texttt{Update}(T, \vec{s}, p) & \text{(Queries)} \\
| & \texttt{Delete}(T, p) & \\
s ::= & c = v \mid c = c' & (\texttt{SET}\text{ clauses}) \\
| & c = \texttt{Select}(i, p) & \\
i ::= & T & \text{(Intermediates)} \\
| & \texttt{Projection}(\vec{c}, i) & \\
| & \texttt{Dedup}(i) & \\
| & \texttt{Select}(i, p) & \\
| & \texttt{Join}(i_1, i_2, p) & \\
| & \texttt{Aggr}(\vec{c}, c', \alpha, i, p) & \\
| & \texttt{Union}(i_1, i_2) & \\
| & \texttt{LeftJoin}(i_1, i_2, \vec{c} = \vec{c'}) & \\
| & \texttt{Rename}(i, name, \vec{c}) & \\
p ::= & \texttt{True} \mid binop(v, v) \mid \texttt{Exists}\ i & \text{(Predicates)} \\
| & \texttt{Is Null}\ c \mid p\ \texttt{And}\ p \mid p\ \texttt{Or}\ p & \\
| & \texttt{Not}\ p & \\
v ::= & c \mid const \mid \texttt{null} & \text{(Values)} \\
\alpha ::= & \texttt{Max} \mid \texttt{Min} \mid \texttt{Avg} \mid \texttt{Count} \mid \texttt{Sum} & \text{(Aggregators)} \\
| & \texttt{Count-Distinct} \mid \texttt{Concat} & \\
binop ::= & \texttt{=} \mid \texttt{>} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<>} & \text{(Comparators)}
\end{array}
$$

Figure 6. Grammar of synthesized SQL queries, extended from Scythe [22]. T ranges over tables, c ranges over column names, $const$ ranges over constant values, and $name$ ranges over table name strings. Vector notation is used to indicate sequences of one or more elements, e.g. $\vec{s}$ denotes a sequence of one or more SET clauses.

Note that neither our grammar for SET clauses nor Scythe's language of SELECTs has support for advanced SQL features like arithmetic expressions, string manipulations, or date expressions. This is not a fundamental limitation of our approach and these features could be supported by integrating existing synthesis tools [18].

4. Synthesis Algorithm

In this section we present our synthesis algorithm (Algorithm 1) in full. Again we focus on the UPDATE case for clarity. Given an input-output example $(I, T_{update}, T_{out}, K, C)$ and access to a SELECTSYNTHESIS procedure which takes queries of the form (I', T'_{out}, C') and returns a ranked list of candidate queries, UPDATESYNTHESIS returns a single query q or $null$ (if no solution was found).

The algorithm begins with a check that the input-output example is valid (lines 2-7): the schema of T_{update} and T_{out} must match and both tables must have the same number of rows. If this check fails there does not exist a solution to the input-output example, as UPDATE queries cannot alter a table schema or delete/insert rows.

If the check passes, we compute the table containing the rows in T_{update} that will be modified by the UPDATE query ($T_{modified}$). From this, we also compute the corresponding rows in T_{out} ($T'_{modified}$). This is done on lines 8-10 and is valid as T_{update} and T_{out} are guaranteed to have the same schema at this point. The classifier predicate p is then synthesized (line 11) as well as the SET clauses $\vec{s}$ (line 12); if either synthesis subroutine fails, we return $null$.

The synthesis of the classifier predicate (Algorithm 2) closely matches the overview given in Section 2. A call is made to the SELECT synthesizer for queries consistent with the input-output example $(I, T_{modified}, C)$. Of the candidates returned by the SELECT synthesizer, the classifier predicate of the top candidate matching the form (SELECT * FROM T_update WHERE p) is returned.

Algorithm 1 Synthesizing UPDATE Queries

```
1: function UPDATESYNTHESIS($I, T_{update}, T_{out}, K, C$)
2:     if $T_{update}.schema \neq T_{out}.schema$ then
3:         return $null$
4:     end if
5:     if $|T_{update}.content| \neq |T_{out}.content|$ then
6:         return $null$
7:     end if

8:     $T_{modified} \leftarrow T_{update} \setminus T_{out}$
9:     $T_{unmodified} \leftarrow T_{update} \setminus T_{modified}$
10:    $T'_{modified} \leftarrow T_{out} \setminus T_{unmodified}$

11:    $p \leftarrow$ SYNTHESIZEWHERE($I, T_{update}, T_{modified}, C$)
12:    $\vec{s} \leftarrow$ SYNTHESIZESETS($I, T_{modified}, T'_{modified}, K$)

13:    if $(p = null) \vee (\vec{s} = [\,])$ then
14:        return $null$
15:    end if

16:    return $update(T_{update}, \vec{s}, p)$
17: end function
```

Algorithm 2 Synthesizing the Classifier Predicate

```
1: function SYNTHESIZEWHERE($I, T, T_{modified}, C$)
2:     $candidates \leftarrow$ SELECTSYNTHESIS($I, T_{modified}, C$)

3:     for all $candidate$ in $candidates$ do
4:         $Select(i, p) \leftarrow candidate$
5:         if $i = T$ then
6:             return $candidate$
7:         end if
8:     end for

9:     return $null$
10: end function
```

To synthesize the SET clauses (Algorithm 3), we iterate over each column of $T'_{modified}$ as described in Section 2. $\sigma_i(col)$ is shorthand for the ith value in column col. Note that if the algorithm is unable to find a constant value, a column in $T_{modified}$, or a nested SELECT query that accounts for the values of a column in $T'_{modified}$, synthesis fails (lines 14-16).

4.1 Query Correlation

One subtlety of SQL UPDATE semantics which we have not considered so far is how a nested query in a SET clause is evaluated. If a SET clause sets a column col equal to a nested query, all rows that satisfy the classifier predicate will have the value of col set to *the first* result of the nested query. As we show in the following example, this behavior is often undesirable.

Example 2. Suppose there are two input tables: T_{MAIN} and T_{CACHED}, and that the cache table has been invalidated.[4] For each row in T_{CACHED}, we want to set its value equal to that in the row with the same id in T_{MAIN}.

[4] This example is adapted from the question posed in https://stackoverflow.com/questions/12394506/

Algorithm 3 Synthesizing SET Clauses

```
1: function SYNTHESIZESETS($I, T, T', K$)
2:     $\vec{s} \leftarrow [\,]$

3:     for all $col$ in $T'$ do
4:         if $\exists k \in K(col)$ s.t. $\forall i, \sigma_i(col) = k$ then
5:             $\vec{s}.append(col = k)$
6:         else if $\exists col2 \in T$ s.t. $\forall i, \sigma_i(col) = \sigma_i(col2)$ then
7:             $\vec{s}.append(col = col2)$
8:         else
9:             $candidates = $ SELECTSYNTHESIS$(I, T', C)$
10:            if $|T'.content| > 1$ then
11:                $candidates.map($CORRELATEQUERY$)$
12:                $candidates.filter(x => x \neq null)$
13:            end if

14:            if $candidates = [\,]$ then
15:                return $[\,]$
16:            end if

17:            $\vec{s}.append(col = candidates.top)$
18:        end if
19:    end for

20:    return $\vec{s}$
21: end function
```

```
UPDATE CACHED
SET id = (
    SELECT id
    FROM MAIN
    JOIN CACHED
    WHERE CACHED.id = MAIN.id
)
```

Figure 7. Possible incorrect solution to the problem in Example 2.

Formally, the problem is $(I, T_{\text{CACHED}}, T_{out}, K, \emptyset)$, where $I = \{T_{\text{MAIN}}, T_{\text{CACHED}}\}$ and $K(col) = \emptyset$ for all columns col. The tables are given below:

T_{MAIN}

id	value
031938	AAA
930111	BBB
000391	CCC
129078	DDD

$T_{\text{CACHED}} = T_{update}$

id	value
031938	NULL
129078	NULL

T_{out}

id	value
031938	**AAA**
129078	**DDD**

A user might erroneously produce the query shown in Figure 7. This will result in the following incorrect result (the incorrect tuple is shown in red):

T_{out}

id	value
031938	AAA
129078	AAA

since the nested query used in the SET clause evaluates to:

value
AAA
DDD

and only the first result (bolded) will be used to update the values in T_{update}.

Solution. Instead, to get the intended result, we must ensure that the rows in the nested query are correctly matched up with the rows in T_{update}. This can be achieved through the use of a *correlated subquery* [15], as demonstrated in Figure 8. The nested query is correlated because the column CACHED.id in the WHERE clause (emphasized in blue) refers to a column in the table being updated in the outer query.

```
UPDATE CACHED
SET id = (
    SELECT id
    FROM MAIN
    WHERE CACHED.id = MAIN.id
);
```

Figure 8. A correct solution to the problem in Example 2 using a correlated subquery. This is also the actual result returned by our system.

The nested queries synthesized by Algorithm 3 on line 9 are unaware of the UPDATE context in which they will be used (since the underlying SELECT synthesizer does not know anything about UPDATE synthesis) and can thus lead to errors like the one shown in Example 2. To remedy this, we generalize the solution we employed in that example to transform nested queries into correlated queries. Algorithm 4 shows the implementation of the CORRELATEQUERY function, which is applied to each nested query candidate on line 11 in the SET clause synthesis algorithm (Algorithm 3).

Algorithm 4 Correlating Candidate Queries

```
1: function CORRELATEQUERY($candidateQ, T_{update}$)
2:     $Projection(\vec{c}, q) \leftarrow candidateQ$
3:     $Join(i_1, i_2, p) \leftarrow q$

4:     if GETCOLUMNNAMES$(p) \cap T_{update}.schema \neq \emptyset$ then
5:         if $i_1 = T_{update}$ then
6:             return $Projection(\vec{c}, Select(i_2, p))$
7:         else if $i_2 = T_{update}$ then
8:             return $Projection(\vec{c}, Select(i_1, p))$
9:         end if
10:    end if

11:    return $null$
12: end function
```

Intuitively, this approach checks if the query is a projection on top of a JOIN on T_{update} (such as the query shown in Figure 7). If the structure matching fails on line 3, we return *null*. Next, on line 4 we check to make sure that the predicate in the JOIN references some column in T_{update} so that the resulting query will be a correlated subquery. Finally, we check to make sure that one of the tables in the JOIN was T_{update} and return a selection over the table that *is not* T_{update} using the predicate from the original JOIN.

Note that this approach is almost certainly not complete: there are likely many queries that could be transformed into correlated queries that are not of this exact form. However, we claim that this algorithm is sound in that the original query will evaluate to the same relation that the transformed query evaluate to in the context of the UPDATE query.[5] The synthesis may fail to find a solution for a particular input, but it will not make errors of the sort demonstrated in Figure 7. We have found qualitatively that this algorithm is able to correlate the sort of nested queries commonly found in UPDATE queries. However, it could be extended to consider other sorts of nested queries. There is not much prior work to draw on for this sort of transformation because it is really a form of query *de*optimization.[6]

4.2 Implementation

We implement our algorithm in a system called REAPER,[7] written in Java. The system is implemented on top of the SELECT synthesizer Scythe [22], although we treat Scythe as a black box to underscore that our algorithm could be implemented on top of *any* existing SELECT synthesizer satisfying the same interface. All told, the implementation is less than a thousand lines of code added to Scythe, including the extensions discussed in the following section. Scythe itself, by contrast, is over 10,000 lines of Java code. The brevity of REAPER illustrates the value of our approach: by leveraging existing SELECT synthesis tools, support for data modification queries can be achieved for a fraction of the work it would take to build a new system from scratch.

The performance of an implementation of our algorithm is almost entirely dependent on the underlying SELECT synthesizer and is thus orthogonal to the contributions of this paper. We did find, however, that REAPER was able to return solutions within a very reasonable amount of time (a few seconds). As the synthesis of the classifier predicate and the synthesis of each of the SET clauses is independent, there is an opportunity for parallelism to improve performance in future work.

5. Extensions

Lastly, we consider extensions to our basic algorithm, which we have found qualitatively improve the expressiveness and readability of synthesized queries. Because we treat the underlying SELECT synthesizer as a black box, these extensions operate either by manipulating the input to the black box synthesizer or transforming the output.

5.1 Example Transformation

The hand-constructed input-output examples received by a SQL query synthesizer will often have two key characteristics. First, the input and output tables will be small (perhaps a few rows). Second, the values in the tables will draw from a small range (say 1-10 for integer columns). While these concise examples are easy to read and write, they have an unfortunate side effect: as the following example will demonstrate, there will be many coincidences in the tables that a synthesizer may capitalize on, leading to results that vastly diverge from user intent.

Example 3. We use an example inspired by a Stack Overflow question.[8] The example consists of three input tables: $T_{\text{MATERIALS}}$, which lists materials; T_{PARTS}, which lists parts; and $T_{\text{PART\_MATERIALS}}$, which lists which materials a part needs (`p_id` $= 1$, `m_id` $= 1$ means the part 'sink' requires the material 'steel'). The scenario is as follows: the user wants to update the `qty` field of $T_{\text{MATERIALS}}$ such that it reflects the number of parts that use that material.

Formally, the problem is $(I, T_{\text{MATERIALS}}, T_{out}, K, \emptyset)$, where $I = \{T_{\text{MATERIALS}}, T_{\text{PARTS}}, T_{\text{PART\_MATERIALS}}\}$ and $K(col) = \emptyset$ for all *col*. Note that no constants are needed for this example.

$T_{\text{MATERIALS}} = T_{update}$

m_id	name	qty
1	Steel	0
2	Wood	0
3	Gold	0

T_{PARTS}

p_id	name
1	Sink
2	Chair
3	Ring

$T_{\text{PART\_MATERIALS}}$

p_id	m_id
1	1
2	1
2	2
3	3

T_{out}

m_id	name	qty
1	Steel	**2**
2	Wood	**1**
3	Gold	**1**

There are many coincidences in this example. The values in the qty column of T_{out} (which we want to synthesize a nested SELECT query to produce) are a count of the number of rows in $T_{\text{PART\_MATERIALS}}$ with a particular m_id. But the values in this column—1 and 2—are also m_ids of materials ('Steel' and 'Wood') and p_ids of parts ('Sink' and 'Chair'). As a result, when this query is passed to our base implementation, the result tries to perform a very complicated UPDATE such that the qty column is set to the p_ids 1 and 2.

Solution. To produce relevant results while still allowing the user to write small, simple input-output examples, we develop coincidence-removing example transformations. The idea is simple: allow users to add a small annotation ([id]) to table columns, and then map the values in annotated columns to random integers disjoint from the set of user-provided constants. Annotations can specify a transformation group (e.g. [id1]); all columns in a particular group will be transformed by the same function and it is guaranteed that the transformed values in a group will be disjoint from the transformed values of every other group. Transformation groups are necessary for examples with tables with foreign keys. The annotation is [id] because id columns are the most common example of columns where the specific values don't usually matter so long as they are consistently used. Our transformation preserves equality and order, which is useful if a JOIN with or MIN/MAX over an annotated column is necessary. For Example 3, the user might specify MATERIALS.m_id, PART_MATERIALS.m_id as transformation group 1 and PARTS.p_id, PART_MATERIALS.p_id as transformation group 2. A transformed version of Example 3 following these annotations is shown below:

$T_{\text{MATERIALS}} = T_{update}$

m_id	name	qty
3056	Steel	0
42214	Wood	0
60649	Gold	0

T_{PARTS}

p_id	name
7126	Sink
23178	Chair
91016	Ring

$T_{\text{PART\_MATERIALS}}$

p_id	m_id
7126	3056
23178	3056
23178	42214
91016	60649

[5] Note that based on line 10 of Algorithm 3 that candidate queries that result in 1 row, 1 column relations are not transformed into correlated queries. This is sound considering that the "top one" semantics for evaluating the nested query will produce the intended result in this case. In fact, some common types of UPDATEs are elegantly expressed using an uncorrelated nested query (see Figure 3). However, there is no guarantee that when the query is evaluated against the full database from which the I/O example was drawn that it will still result in a 1×1 relation. This means that queries returned by our system could fail when used in a more general context. Future work could ask users to mark certain columns as having schema uniqueness constraints and then conservatively reason about when a query is *guaranteed* to evaluate to a 1×1 relation for any tables satisfying the given schema.

[6] A naive DBMS may evaluate a correlated subquery once for each row in the outer relation, whereas a nested query only needs to be evaluated once.

[7] REAPER = Scythe + Tuple lifecycle management

[8] https://stackoverflow.com/questions/3949592/

```
UPDATE MATERIALS
SET qty = (SELECT T1.count_p_id
 FROM
  (SELECT
     PART_MATERIALS.m_id,
     COUNT(PART_MATERIALS.p_id) AS count_p_id
    FROM
     PART_MATERIALS
    GROUP BY
     PART_MATERIALS.p_id) AS T1
 WHERE T1.m_id = MATERIALS.m_id)
```

Figure 9. Our synthesizer's solution for the transformed version of Example 3.

```
SELECT T2.created_id
 FROM
  (SELECT TABLE1.id, TABLE1.subst_id,
    TABLE1.created_id, T1.id AS id1,
    T1.serial_id, T1.branch_id
  FROM TABLE1 JOIN
     TABLE2 As T1) AS T2
 WHERE T2.subst_id = T2.serial_id
```

Figure 10. Scythe output for a simple `JOIN`.

T_{out}

m_id	name	qty
3056	Steel	**2**
42214	Wood	**1**
60649	Gold	**1**

The result when this transformed example is passed to REAPER is shown in Figure 9 and precisely matches the user's intent. There is one caveat to note regarding the use of example transformation: as we transform only the column values but not any of the user-provided constants (again because the `[id]` should only be applied to columns where the specific values truly do not matter except for their equality and order), it is possible that the transformed example contains no solutions even if there was a solution to the untransformed example. Moreover, it is conceivable that the transformed example itself introduces coincidences and thus results in a query that is invalid over the original example. Because this transformation is therefore unsound in general, REAPER evaluates the synthesized query resulting from the transformed example on the untransformed input tables and verifies that it is indeed a solution to the user's problem; if it is not or there are no solutions to the transformed example, REAPER tries instead to synthesize a solution for the untransformed example as usual.

5.2 Rename Elimination

In developing REAPER, we found that the `SELECT` results returned from Scythe were often difficult to read due to renaming of intermediate results. Figure 10 provides a prototypical example of this problem, which we encountered in synthesizing a `SET` clause involving a nested query. If one examines the query closely and traces the renames through the nested query layers, it becomes clear that the query is equivalent to the simple equijoin shown in Figure 11.

Naturally, we prefer the query in Figure 11 to the one in Figure 10, since it is easier to read and therefore validate. Further, simpler queries are also more amenable to the transformations that our algorithm and its extensions perform (for example, it is obvious how to transform the query in Figure 11 into a correlated subquery that would be suitable for updating `TABLE1` or `TABLE2`).

```
SELECT TABLE1.created_id
 FROM
  TABLE1 JOIN
    TABLE2
 WHERE TABLE1.subst_id = TABLE2.serial_id
```

Figure 11. Semantically equivalent version of the query in Figure 10 after applying rename elimination.

```
UPDATE CUSTOMERS
SET firstname = 'John'
WHERE
  CUSTOMERS.id =
    (SELECT
     MAX(CUSTOMERS.id) AS max_id
     FROM CUSTOMERS)
```

Figure 12. An `UPDATE` query that has a nested `SELECT` in the `WHERE` clause.

To this end, we implement a set of heuristics for safely removing unneeded renames. These are applied to the results of the `SELECT` synthesizer as well as to the results of other query transformations (as the transformations may result in more renames being removable than before). We briefly describe the situations in which we remove renames and provide (without proof) an intuitive explanation for their validity:

- ***Within a `JOIN` over named tables***: the clause `TABLE2 as T1` in Figure 11 provides a good example of this case. Since it is possible to unambiguously refer to the columns of this `JOIN` using the fully qualified column name (e.g. `TABLE2.serial_id`), the rename is not required. The restriction that the `JOIN` term be a named table is necessary: in the case of a nested `SELECT`, the rename may be needed to refer to the columns generated by the intermediate result.
- ***`SELECT` immediately below a top-level `SELECT`***: the clause `... AS t2` in Figure 10 is a good example of this. The columns of this `SELECT` can be unambiguously referred to by their column names in the top-level `SELECT`, since we know that the nested result is not used higher up in any `JOIN`s or other operations that might necessitate the rename.

We have found qualitatively that these heuristics are enough to result in readable queries and facilitate other desired transformations. Further, the implementation of the tests is straightforward. If one desired to eliminate even more renames, a more nuanced approach that takes into account which columns of intermediate results are used in later parts of the query may be possible.

5.3 Nested Query Lifting

As Figure 6 shows, our base implementation of REAPER does not support classifier predicates p with nested queries, meaning we would be unable to produce queries like that in Figure 12. This restriction is simply because Scythe does not support such nested queries in `WHERE` clauses either. For `SELECT` synthesis this limitation is irrelevant as the same predicates can be expressed using a `JOIN` as shown in Figure 13. The same can be done for `UPDATE` queries in some DBMSs (e.g. MySQL) but is non-standard. Supporting these sorts of queries in general is important for the practical utility of our tool; when one considers `DELETE` queries in particular, which only have a classifier predicate, almost all interesting queries which might prompt a user to employ a synthesizer will have a nested query in the `WHERE` clause.

```
SELECT T1.max_id, CUSTOMERS.firstname
FROM
  (SELECT
    MAX(T2.id) AS max_id
   FROM CUSTOMERS AS T2) AS T1 JOIN
   CUSTOMERS
WHERE T1.max_id = CUSTOMERS.id
```

Figure 13. A `SELECT` query that achieves an equivalent `WHERE` clause to the query in Figure 12 using a `JOIN`.

```
SELECT *
FROM CUSTOMERS
WHERE
  CUSTOMERS.id =
    (SELECT
     MAX(CUSTOMERS.id) AS max_id
     FROM CUSTOMERS)
```

Figure 14. The `SELECT` query from Figure 13 after "nested query lifting."

To support nested queries in the classifier predicate for a limited number of cases, we use a technique we call "nested query lifting." The idea is as follows: if the classifier predicate in the original `SELECT` query (as in Figure 13) is a comparison between a column in T_{update} (`CUSTOMERS.id`) and a column in an intermediate result table generated in the `FROM` clause (`T1.max_id`), we can "lift" the query that produces the column of the intermediate result and use it in the predicate. Intuitively, this technique is thus analogous to inlining the definition of a local variable. The restrictions are that the outermost layer of the `FROM` clause in the original `SELECT` must be a join between T_{update} and the intermediate result table (`T1 JOIN CUSTOMERS`) and that the result of the nested query we are "lifting" is a 1-row, 1-column relation (else this technique is not well-founded).[9]

In the formal SQL grammar presented in Section 6, nested query lifting transforms queries of the form

$$Join(T_{update}, i, binop(T_{update}.\texttt{col1}, i.\texttt{col2}))$$

to queries of the form

$$\begin{aligned}&Select(\ T_{update},\\&\qquad binop(T_{update}.\texttt{col1}, \sigma_1(\texttt{Projection}(\texttt{col2}, i))))\end{aligned}$$

the latter of which passes the `SELECT * FROM T_update WHERE p` structure requirement imposed by the classifier predicate synthesis subroutine. The result of nested query lifting on the query in Figure 13 (after rename elimination) is shown in Figure 14.

6. Evaluation

Before we implemented out Reaper, we picked 5 examples of `UPDATE`/`DELETE` queries from a SQL textbook [16]. We also searched through posts and Stack Overflow and picked 10 interesting and complex examples that we felt were representative of user questions. The Stack Overflow questions were generally more sophisticated than the textbook examples. In total, we used 15 total examples (10 `UPDATES` and 5 `DELETES`) to evaluate Reaper and found that it was able to solve 10/15 correctly (corresponding to 8/10 `UPDATES`, 2/5 `DELETES`, 4/5 textbook questions, and 6/10 Stack Overflow questions).[10] For all examples tested, Reaper returned either a correct answer or a failure within a few seconds.

We attribute the three failed `DELETE` cases to limitations of Scythe. For two examples, the expected solutions involved `WHERE IN...` clauses in the `DELETE` queries, which Scythe does not support. This could be remedied by extending Scythe with support for this support of query, or implementing a transformation like nested query lifting (Section 5.3) that allows us to synthesis this sort of classifier predicate without modifying the `SELECT` synthesizer. The final case involves a simple disjunctive `WHERE` clause which Scythe failed to synthesize.

The two failed `UPDATE` cases are more interesting. In both cases, the intended solution involved `SET` clauses with reasonably simple nested queries. However, Scythe returned complicated queries which could not obviously be transformed into correlated nested queries for which our correlation procedure was inadequate (Algorithm 4). On the one hand, one might say that Scythe ought to have synthesized simpler queries with a top level `JOIN` that could have been correlated successfully. To this end, one could attempt to improve on `SELECT` synthesis or ranking algorithms to output simpler queries that can more easily be reasoned about. On the other hand, one might say that Reaper ought to be able to handle a more robust subset of the possible `SELECT` queries that the black box synthesis procedure might return. To address this, one could attempt to write more sophisticated transformations or correlation procedures. In either case, these cases highlight the boundary point between our algorithm and the black box select synthesizer as a likely cause for limitations in Reaper.

Overall, we are encouraged by the qualitative performance of Reaper, especially considering the simplicity of its implementation. We have found that, in practice, it is able to solve a variety of SQL synthesis problems that are representative of the sort of queries users struggle with. Future work could evaluate the practicality of creating a `SELECT` synthesizer that is aware of the semantics of SQL `UPDATE`s.

7. Related Work

Programming by Example. Our work is inspired by PBE systems, wherein users specify input/output examples and the system synthesizes a program compatible with those examples. Gulwani [10] provides a comprehensive overview of program synthesis and PBE techniques. PBE has been used to synthesize programs in such diverse domains as graphics [3, 9], data manipulation tasks [8, 18], and data structure transformations [23].

Scythe [22], SQLSynthesizer [24], and Query By Output [20] are existing PBE systems for SQL `SELECT` synthesis. We built our system on top of Scythe because it uses an algorithm based on the notion of *abstract queries* which is more efficient than previous approaches based on decision trees. Scythe also supports a richer set of SQL operations such as `UNION`, `EXISTS`, and free-form subquery nesting which were impractical in earlier systems due to scalability limitations. To the best of our knowledge, no previous PBE system has addressed synthesizing SQL data modification queries.

Relational Database Usability. Researchers have long recognized the need to address the usability limitations of relational databases [5]. Approaches to database usability have typically been in one of two areas: designing new query interfaces or considering context and personalization [12]. Our work, like other SQL PBE systems, provides an example-based query interface on top of SQL. Other proposed interfaces have included visual [2] or form-based

[9] This approach suffers from the same query generalization problem as our query correlation algorithm (see note 5).

[10] The examples are at https://github.com/ezig/Reaper/tree/master/data/reaper under "textbook" and "stackoverflow".

[4] interfaces as well as keyword-based [1] and natural language [14] approaches.

Usability approaches addressing context and personalization inherit from a long line of work on *cooperative answering* [6] which uses principles of conversational cooperation [7] studied in linguistics to enhance responses to queries. Koutrika and Ioannidis [13] create a user preference model for determining what query results a user will be interested in. Ioannidis and Viglas [11] propose conversational querying in which query results depend on the context of previous queries in the session.

8. Conclusion & Research Outlook

In this paper, we presented an algorithm for synthesizing SQL data modification queries on top of any existing SQL `SELECT` query synthesizer. The key idea is that `UPDATE` and `DELETE` query synthesis can be decomposed into several small problems, each of which can be solved by a `SELECT` synthesizer. We present a number of extensions to our basic algorithm which improve the expressiveness and readability of queries. We implement our ideas in REAPER, which can synthesize a wide range of `UPDATE` and `DELETE` queries in just a few seconds.

There are a number of directions for further study. First and foremost, there is much potential for improvement in the usability of REAPER. Our tool, for instance, currently only has a command-line interface; creating a rich, web-based user interface in the style of Wang et al. [21] may be a worthwhile investment. A web-based user interface could also let us display multiple candidates for the classifier predicate and each `SET` clause, as well as perhaps allow the user to mark coincidental non-updates and preview the `UPDATE` or `DELETE` query on their actual database. A second direction is to investigate the synthesis of sequences of data modification queries. That is, the user may provide us with an input-output example and we return a series of queries that take the database from the input state to the output state. Third, it is worth investigating whether access to richer schema information can automate annotation of input-output examples. In this paper, we treated a schema as just a mapping from column names to value types; knowledge of foreign key constraints for instance may help us infer transformation groups. Fourth, although there are sets of input-output examples that are used as benchmarks for `SELECT` synthesis, no such data sets exist for `UPDATE` or `DELETE` synthesis. Constructing such a benchmark would more rigorously evaluate the effectiveness of REAPER and future SQL data modification query synthesizers. Finally, as mentioned in Section 6, it is not a foregone conclusion that treating a `SELECT` synthesizer as a black box is the best approach to SQL data modification query synthesis. It is worth considering the potential advantages of creating a SQL `SELECT` system that is aware of `UPDATE` semantics.

Acknowledgments

We would like to thank Stephen Chong and Christos Dimoulas for their support and guidance throughout the development of this project.

References

[1] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. 2002. DBXplorer: enabling keyword search over relational databases. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (SIGMOD '02). ACM, New York, NY, USA, 627-627. DOI: https://doi.org/10.1145/564691.564782

[2] Francesca Benzi, Dario Maio, and Stefano Rizzi. Visionary: A Viewpoint-based Visual Language for Querying Relational Databases. *Journal of Visual Languages and Computing* 10, 2 (April 1999), 117-145. DOI: https://doi.org/10.1006/jvlc.1998.0102

[3] Salman Cheema, Sarah Buchanan, Sumit Gulwani, and Joseph J. LaViola, Jr.. 2014. A practical framework for constructing structured drawings. In *Proceedings of the 19th international conference on Intelligent User Interfaces* (IUI '14). ACM, New York, NY, USA, 311-316. DOI: http://dx.doi.org/10.1145/2557500.2557522

[4] Joobin Choobineh, Michael V. Mannino, and Veronica P. Tseng. 1992. A form-based approach for database analysis and design. *Commun. ACM* 35, 2 (February 1992), 108-120. DOI=http://dx.doi.org/10.1145/129630.129636

[5] C. J. Date. 1983. Database usability. In *Proceedings of the 1983 ACM SIGMOD international conference on Management of data* (SIGMOD '83). ACM, New York, NY, USA, 1-1. DOI: https://doi.org/10.1145/582192.582194

[6] Terry Gaasterland, Parke Godfrey, and Jack Minker. An overview of cooperative answering. *Journal of Intelligent Information Systems* 1, 2 (Oct. 1992), 123-157. DOI: https://doi.org/10.1007/BF00962280

[7] H. Paul Grice. 1975. Logic and Conversation. In *Syntax and Semantics*, Vol. 3: Speech Acts, ed. by Peter Cole and Jerry L. Morgan. Academic Press, New York, 1975, 41-58.

[8] Sumit Gulwani. 2011. 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). ACM, New York, NY, USA, 317-330. DOI: https://doi.org/10.1145/1926385.1926423

[9] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. 2011. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '11). ACM, New York, NY, USA, 50-61. DOI: http://dx.doi.org/10.1145/1993498.1993505

[10] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1-119. DOI: http://dx.doi.org/10.1561/2500000010

[11] Yannis E. Ioannidis and Stratis D. Viglas. 2006. Conversational querying. *Inf. Syst.* 31, 1 (March 2006), 33-56. DOI=http://dx.doi.org/10.1016/j.is.2004.09.002

[12] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. 2007. Making database systems usable. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (SIGMOD '07). ACM, New York, NY, USA, 13-24. DOI: https://doi.org/10.1145/1247480.1247483 2007.

[13] Georgia Koutrika and Yannis Ioannidis. Personalization of Queries in Database Systems. In *Proceedings of the 20th International Conference on Data Engineering* (ICDE '04). IEEE Press, Piscataway, NJ, USA. DOI: http://dx.doi.org/10.1109/ICDE.2004.1320030

[14] Fei Li and H. V. Jagadish. 2016. Understanding Natural Language Queries over Relational Databases. *SIGMOD Rec.* 45, 1 (June 2016), 6-13. DOI: http://dx.doi.org/10.1145/2949741.2949744

[15] Microsoft TechNet. Correlated Subqueries. Retrieved December 1, 2017 from https://technet.microsoft.com/en-us/library/ms187638.aspx.

[16] Andy Oppel and Robert Sheldon. 2009. *SQL: A Beginners Guide* (3rd ed.). McGraw-Hill, New York, NY.

[17] Oracle. MySQL :: MySQL 5.7 Reference Manual :: 13.2.11 UPDATE Syntax. Retrieved December 1, 2017 from https://dev.mysql.com/doc/refman/5.7/en/update.html.

[18] Rishabh Singh and Sumit Gulwani. 2012. Learning semantic string transformations from examples. *Proc. VLDB Endow.* 5, 8 (April 2012), 740-751. DOI: http://dx.doi.org/10.14778/2212351.2212356

[19] Stack Overflow. Stack Overflow Developer Survey 2017. Retrieved December 1, 2017 from https://insights.stackoverflow.com/survey/2017.

[20] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (SIGMOD '09), Carsten Binnig and Benoit Dageville (Eds.). ACM, New York, NY, USA, 535-548. DOI: http://dx.doi.org/10.1145/1559845.1559902

[21] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive Query Synthesis from Input-Output Examples. In

Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17). ACM, New York, NY, USA, 1631-1634. DOI: https://doi.org/10.1145/3035918.3058738

[22] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2017). ACM, New York, NY, USA, 452-466. DOI: https://doi.org/10.1145/3062341.3062365

[23] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '16). ACM, New York, NY, USA, 508-521. DOI: https://doi.org/10.1145/2908080.2908088

[24] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, USA, 224-234. DOI: https://doi.org/10.1109/ASE.2013.6693082