# SQUARES : A SQL Synthesizer Using Query Reverse Engineering

Pedro Miguel Orvalho Marques da Silva
pedro.orvalho@tecnico.ulisboa.pt

INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Portugal
OutSystems, Lisboa

November 2019

**Abstract**

Nowadays, with the massive amount of data that data analysts have to deal with, they frequently find tables with interesting data and they do not know how these tables were generated from a database. Hence, there is an increasing need for systems capable of solving the problem of *Query Reverse Engineering* (QRE). Given a database $\mathcal{D}$ and an output table $\mathcal{Q}(\mathcal{D})$, these systems have to find a query $\mathcal{Q}$, such that, when running $\mathcal{Q}$ on $\mathcal{D}$, the result is equal to $\mathcal{Q}(\mathcal{D})$. QRE is a subfield of Program Synthesis.

The goal of Program Synthesis is to automatically generate programs that satisfy a given high-level specification. Since the 60's, Program Synthesis is a well-studied problem, and has been considered the Holy Grail of Computer Science. Until now, program synthesizers have been using a single tree representation to represent programs. We propose a novel enumeration-based SQL synthesizer SQUARES, that uses a new line representation where we represent each program line with its own subtree.

Experimental results on the synthesis of SQL queries, show that the proposed line-based encoding allows a faster enumeration of programs when compared to the usual tree-based encoding. Moreover, while the tree-based encoding does not scale beyond a small number of operations, the new line-based encoding allows finding programs with a larger sequence of operations. Experimental results on the synthesis of SQL queries from OutSystems show that SQUARES outperforms Scythe, a state-of-the-art SQL synthesizer.

**Keywords:** Program Synthesis, Query Reverse Engineering, Satisfiability Modulo Theories (SMT), Enumeration-Based Program Synthesis, SQL

## 1. Introduction

The goal of Program Synthesis is to automatically generate programs that satisfy a given high-level specification [16]. Since the 60's, Program Synthesis is a well-studied problem, and has been considered the Holy Grail of Computer Science [10, 14].

Nowadays, the use of input-output examples as specifications is a common approach [10]. Even though these specifications are incomplete (i.e., a program may satisfy the specification but may not be the program that the user desires), these are easy to create and can be used to solve many real-world applications. This subfield of Program Synthesis is known as Programming By Example (PBE) and it has received more attention in the last decade. PBE has been used to automate tedious tasks in a plethora of applications, such as string manipulations in spreadsheets [8, 26], table reshaping [6] and SQL queries [33, 35, 37].

Even though there are many approaches to program synthesis [10], one of the most common approaches is to perform an enumerative search over the space of programs that satisfy the specifications [6, 7, 18]. Fig. 1 shows the high-level architecture of enumeration-based program synthesizers. They take as input the specification that describes the intention of the user (e.g., input-output examples) and a domain-specific language (DSL) that defines the search space. Program synthesizers typically enumerate programs in increasing order of the number of DSL components. For each candidate program $\mathcal{P}$, they check if $\mathcal{P}$ satisfies the specification. If this is the case, then a program consistent with the specification has been found. Otherwise, the program synthesizer learns a reason for failure and enumerates the next candidate program.

Since the beginning of the 21$^{st}$ century, with the Big Data revolution, several companies started having trouble with the management of their databases,

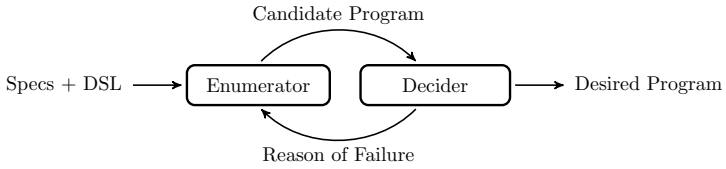


Figure 1: Enumeration-based Program Synthesis

concerning the massive amount of data that suddenly appeared. Although most users know how to make a description of what they want, or what the task should do, sometimes they do not know how to express it in a query language, such as SQL. On that account, more and more systems started to appear in order to help end-users query a relational database [15, 30, 31, 37]. This subfield of Program Synthesis became known as Query Synthesis, where the goal is to find the query desired by the user [30].

This thesis is concerned with the problem of Query Synthesis from input-output examples, most commonly known as Query Reverse Engineering (QRE) which is a subfield of Programming By Example. Given a database $\mathcal{D}$, and an output table $\mathcal{Q}(\mathcal{D})$, the goal is to synthesize a query $\mathcal{Q}$, such that, when running $\mathcal{Q}$ on $\mathcal{D}$ the result is $\mathcal{Q}(\mathcal{D})$ [30]. We focus on a new system, SQUARES, whose objective is to solve QRE for a subset of SQL.

### 1.1. Motivating Example

Suppose that a user wants to synthesize an SQL query using examples. In particular, given tables *supplier* and *parts*, with the schema "supplier(id: integer, sname: string)" and "parts(id: integer, pname: string)", the user wants to find the *names* of parts, *pnames*, for which there is some supplier [1]. This could be accomplished with the following SQL query:

```
SELECT pname
FROM parts, supplier
WHERE parts.id = supplier.id
```

To enumerate the space of programs that satisfy the specifications, program synthesizers must first construct an underlying representation of the feasible space. Figure 2 shows the typical tree representation used by program synthesizers [2, 6, 7], for the above query example. Each node can be a library component or a terminal symbol. Program synthesizers can then traverse the space of possible candidates by enumerating all possible trees of a given depth. However, for approaches that rely on logical deduction, the space of feasible programs is encoded a priori by using either a Boolean Satisfiability (SAT) or a Satisfiability Modulo Theory (SMT) formula [6, 7]. A common approach to encode all feasible programs is to represent them using a $k$-tree, where each node has exactly $k$ children and $k$ is the largest number of parameters of the functions in our library of components. Figure 2 shows an example of a 3-tree where each node has 3 children. A complete program corresponds to assigning a label to each node. Components that may have less than 3 parameters (e.g., `SELECT`), will have the empty label $\varepsilon$ assigned to their unused children.

A large downside of a $k$-tree representation is the exponential growth of the size of the tree with respect to its depth. For instance, Figure 2 would need 40 nodes to represent the search space for 3 lines of code with $k = 3$. If we consider programs with 10 lines of code with $k = 4$, then we would need to build a tree with 1,398,101 nodes. Since the encoding's complexity depends on the number of nodes, this makes it intractable to enumerate the search space of candidate programs using an SMT encoding.

### 1.2. Contributions

In this work, we propose a new line representation illustrated in Figure 3, where we represent each program line with its own subtree and add additional constraints to connect the multiple subtrees. For the above SQL query, we would only need 12 nodes using a line-based representation instead of the 3-tree representation's 40 nodes. When considering programs with 10 lines of code and $k = 4$, the line-based representation only needs 50 nodes instead of the 1,398,101 nodes required by the tree-based representation.

We also present a novel program synthesizer, SQUARES, that uses Enumeration-based Program Synthesis. SQUARES was developed on top of a state-of-the-art synthesis framework Trinity [18]. Trinity uses the traditional tree-based representation of a program, while SQUARES incorporates our new line representation illustrated in Figure 3.

SQUARES' goal is to synthesize SQL queries from input-output examples, i.e., solve the problem of Query Reverse Engineering. We gathered SQL instances previously used by well-known SQL synthesizers [6, 33, 37] and some query examples used in OutSystems' Engineering Database [22]. We evaluate our system with these instances and compare its performance against Scythe [33], a state-

[1] This corresponds to exercise 5.2.1 from a classic textbook on databases [24]

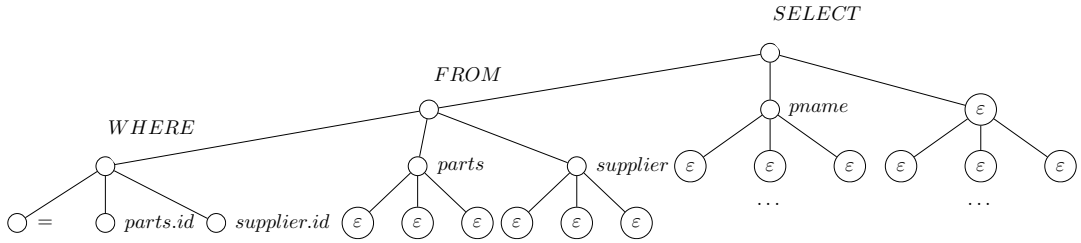


Figure 2: Tree-based representation of the search space

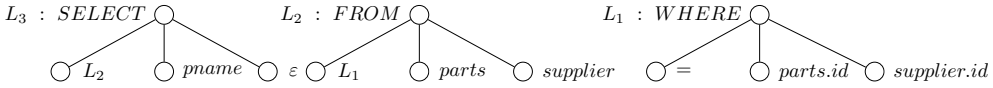


Figure 3: Line-based representation of the search space

of-the-art synthesizer presented at PLDI'17, whose goal is to solve the problem of Query Reverse Engineering.

## 2. Preliminaries

This section provides some definitions and notation that will be used throughout the document.

We assume that the reader is familiar with the Boolean Satisfiability Problem. Otherwise, the interested reader is referred to the literature [1, 20] for more details.

**Definition 1** (**SAT Solver**). *A SAT Solver is a logic engine capable of deciding if a given propositional formula $\phi$ is satisfiable. In this case, the solver produces a truth assignment to the variables of $\phi$ such that $\phi$ evaluates to true. Otherwise, the solver returns that $\phi$ is unsatisfiable [17].*

**Definition 2** (**Satisfiability Modulo Theories (SMT)**). *The Satisfiability Modulo Theories (SMT) problem is a generalization of the SAT problem. Given a decidable first-order theory $\mathcal{T}$, a $\mathcal{T}$-atom is a ground atomic formula in $\mathcal{T}$. A $\mathcal{T}$-literal is either a $\mathcal{T}$-atom $t$ or its complement $\neg t$. A $\mathcal{T}$-formula is similar to a propositional formula, but a $\mathcal{T}$-formula is composed of $\mathcal{T}$-literals instead of propositional literals. Given a $\mathcal{T}$-formula $\phi$, the SMT problem consists of deciding if there exists a complete assignment over the variables of $\phi$ such that $\phi$ is satisfied. Depending on the theory $\mathcal{T}$, the variables can be of type integer, real, Boolean, among others [21].*

**Example 1.** *Let $\phi_1 = (x_1 \geq 0) \wedge (x_1 \leq 4) \wedge (x_2 \leq 2) \wedge (x_1 + x_2 = 5)$ be an SMT formula where $\mathcal{T}$ is the Linear Integer Arithmetic (LIA) theory. Clearly, $\phi_1$ is satisfiable and a possible solution would be $x_1 = 4, x_2 = 1$. Let $\phi_2 = (x_1 \geq 0) \wedge (x_1 \leq 3) \wedge (x_2 \leq 1) \wedge (x_1 + x_2 = 5)$ be an SMT formula also in the LIA theory. In this case, $\phi_2$ is unsatisfiable since there is no assignment to the problem variables such that $\phi_2$ is evaluated to true.*

**Definition 3** (**Table**). *A Table $\Gamma$ is a 3-tuple $(C, R, \psi)$. The number of columns is denoted by $C$, and the numbers of rows by $R$. $\Gamma_{c_1,r_1}$ is the element in the column $c_1$ and row $r_1$. A column of a table $\Gamma$ is normally called an attribute of $\Gamma$ [6].*

**Definition 4** (**Schema Graph**). *A database $\mathcal{D}$ is represented by its schema graph $\mathcal{G}_S = (\mathcal{N}_S, E_S)$. $\mathcal{N}_S$ is the set of nodes and $E_S$ denotes the set of edges. Each node in $N_S$ corresponds to a distinct table $\Gamma$ in $\mathcal{D}$. Each edge between two distinct tables $\Gamma_1$ and $\Gamma_2$ is in $E_S$ if a join is possible between these two tables, i.e. if both tables share an attribute [13].*

**Definition 5** (**Context-free Grammar (CFG)**). *A context-free grammar $\mathcal{G}$ is a 4-tuple $(\mathcal{V}, \Sigma, R, S)$, where $\mathcal{V}$ is the set of non-terminals symbols, $\Sigma$ is the set of terminal symbols, $R$ is the set of rules and $S$ is the start symbol. A CFG describes all the strings permitted in a certain formal language [11].*

**Definition 6** (**Domain-Specific Language (DSL)**). *A Domain-specific Language (DSL) is a tuple $(\mathcal{G}, Ops)$, where $\mathcal{G}$ is a context-free grammar ($\mathcal{G} = (\mathcal{V}, \Sigma, R, S)$) and Ops is the semantics of DSL operators. The CFG $\mathcal{G}$ has the rules to generate all the programs in the DSL. The semantics of DSL operators is necessary to analyze conflicts and make deductions [7].*

Each symbol $\sigma \in \Sigma$ corresponds to built-in DSL constructs (e.g., `SELECT`, `WHERE`, `FROM`), constants, variables or inputs of the system. Each production rule $p \in R$ has the form $p = (A \rightarrow \sigma(A_1, \ldots, A_m))$, where $\sigma \in \Sigma$ is a DSL construct and $A_1, \ldots, A_m \in \Sigma$ are symbols for the arguments of $\sigma$.

**Example 2.** *Consider the DSL D in Fig. 4, and suppose that a user wants to solve the query presented in section 1.1, i.e. she wants to find all the names of parts for which there is some supplier. The desired query from D is the following select_from(column(pname), join(parts, supplier)). This query is obtained using three production rules* $p_1 = \text{select\_from}$, $p_2 = \text{column}$ *and* $p_3 = \text{join}$.

**Definition 7** (**Program Space**). *Program space is the space with all possibilities for program candidates in a certain programming language. The program space grows exponentially with the size of the desired program [10].*

Program synthesizers search the space of programs described by a given domain-specific language (DSL).

**Definition 8** (**Synthesis Problem**). *Given* $(S, \mathcal{G}, Ops)$*, being* $S$ *a program's specification (e.g. input-output examples),* $\mathcal{G}$ *a CFG and Ops the semantics for a particular DSL, the goal of synthesis is to infer a program* $\mathcal{P}$ *such that (1) the program is produced by* $\mathcal{G}$*, (2) the program is correct with respect to Ops and (3)* $\mathcal{P}$ *is consistent with* $S$ *[6].*

$$(S, \mathcal{G}, Ops) \rightarrow \mathcal{P}$$

**Definition 9** (**Programming By Example (PBE)**). *Given* $(\mathcal{E}, \mathcal{G}, Ops)$*, being* $\mathcal{E} = (\mathcal{E}_{in}, \mathcal{E}_{out})$ *a set of input-output examples,* $\mathcal{G}$ *a grammar and Ops the semantics for a particular DSL, the goal of Programming by Example is to infer a program* $\mathcal{P}$ *such that (1) the program is consistent with* $\mathcal{G}$*, (2) the program is correct with respect to Ops and (3)* $\mathcal{P}(\mathcal{E}_{in}) = \mathcal{E}_{out}$ *[6].*

$$(\mathcal{E}, \mathcal{G}, Ops) \rightarrow \mathcal{P}$$

PBE is a special case of Program Synthesis, where the specification is described by a set of input-output examples.

**Definition 10** (**Query Reverse Engineering (QRE)**). *Let* $\mathcal{D}$ *be a database with schema graph* $\mathcal{G}$ *and let* $\mathcal{Q}(\mathcal{D})$ *be an output table, which is the result of running some unknown query* $\mathcal{Q}$ *on* $\mathcal{D}$*. The goal of QRE is to produce a query* $\mathcal{Q}$ *whose result is* $\mathcal{Q}(\mathcal{D})$*, given* $(\mathcal{G}, \mathcal{Q}(\mathcal{D}))$ *[30].*

$$(\mathcal{G}, \mathcal{Q}(\mathcal{D})) \rightarrow \mathcal{Q}$$

QRE is a special case of PBE, where the examples are constructed from database tables.

## 3. Background

This section provides some background on Program Synthesis and on the problem of Query Reverse Engineering.

### 3.1. Program Synthesis

To the best of our knowledge, the first references to Program Synthesis date back to the 60's [3, 32]. Since then a large body of research has been conducted regarding the problem of synthesizing programs automatically [10]. The problem of Program Synthesis has three fundamental characteristics [10]: program space, user intent and search technique.

#### 3.1.1. Program Space

The number of program candidates in the program space grows exponentially with the size of the desired program. Early applications for Program Synthesis were based in searches in an exponentially growing tree. Modern approaches use heuristics for cutting the tree, reducing the search space [12].

#### 3.1.2. User Intent

The second main issue of Program Synthesis is to completely understand the user's desire, given the specification provided by the user. In order to express the user's desire several different approaches take inputs such as: natural-language descriptions [5, 9, 35], a few input-output examples [7, 8, 19], partial programs [14, 27] or related programs [25, 30].

#### 3.1.3. Search Techniques

There are four main search techniques for Program Synthesis: constraint solving, deductive, enumerative and statistical [10]. However, it is possible to use a combination of these techniques [7, 20].

### 3.2. Enumeration-Based Program Synthesis

This section presents the idea behind several state-of-the-art synthesizers [6, 7, 18] that use enumeration-based Program Synthesis. These synthesizers [6, 7, 18] rely on logical deduction, the space of feasible programs must be encoded a priori by using either a Boolean Satisfiability (SAT) or a Satisfiability Modulo Theory (SMT) formula. As presented in Fig. 1, these systems take as input a set of input-output examples and a set of library components that define the search space and typically enumerate programs in increasing order of number of components.

#### 3.2.1. Tree-based Encoding

The tree-based encoding is currently used on several state-of-the-art synthesizers [6, 7, 18] to perform program enumeration [21]. Given a DSL, program synthesizers search for a program that is consistent with the input-output examples provided by

$$
\begin{array}{lcl}
table & \rightarrow & select\_from(cols,\ table) \mid join(table,\ table) \mid parts \mid supplier \\
cols & \rightarrow & column(col) \mid columns(col,\ cols) \\
col & \rightarrow & pname \mid sname \mid id \mid color \mid address \mid * \\
empty & \rightarrow & empty
\end{array}
$$

Figure 4: The grammar of a simple DSL for query synthesis; in this grammar, *table* is the start symbol. All joins are natural joins (",") between columns with the same name. Given as input the tables *supplier* and *parts*, with the schema "supplier(id: integer, sname: string)" and "parts(id: integer, pname: string)".

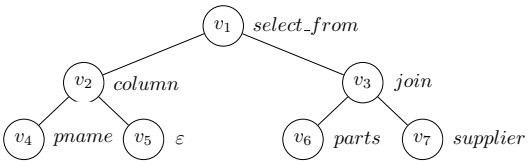


Figure 5: $K$-tree representation of the query presented in Example 2.

the user. For the search process to be complete, these frameworks use a structure capable of representing every possible program up to some given depth $n$. Let $k$ be the greatest arity among DSL constructs. For programs with $n-1$ production rules, synthesizers adopt a tree structure of depth $n$, referred to as $k$-tree, where each node has exactly $k$ children. For example, Fig. 5 illustrates a 2-tree of depth 2.

In order to perform program enumeration using the tree representation, program synthesizers can encode the tree as an SMT formula such that a model of the SMT formula encodes a concrete program by assigning a symbol to each node.

A detailed description of the SMT model follows. First, the encoding variables are introduced. Next, the constraints of the SMT model are presented.

3.2.2. Encoding Variables

Let $s$ be the length of the DSL's set of symbols, $s = |\Sigma|$. Let $id : \Sigma \rightarrow \mathbb{N}_0$ be a function that maps each symbol to an unique non-negative integer in a one-to-one mapping. As a result, this function provides a unique identifier (integer value between 0 and $s$) to each symbol in $\Sigma$. In our encoding, we assume that the empty production symbol ($\varepsilon$) is mapped to 0 (i.e. $id(\varepsilon) = 0$).

Consider the encoding for a program with a $k$-tree of depth $n$. Assume each node in the $k$-tree is assigned an unique index. Let $N$ be the set of all $k$-tree nodes indexes such that $N = I \cup L$ where $I$ denotes the set of internal node indexes and $L$ denotes the set of leaf node indexes. Let $C(i)$ denote the set of child indexes of node $i \in N$. Clearly, if $i$ is a leaf node ($i \in L$), then $C(i) = \emptyset$.

In our encoding we define the following variables: $V = \{v_i : 1 \leq i \leq |N|\}$ : each variable $v_i$ denotes the symbol identifier in node $i$ of the $k$-tree.

3.2.3. Constraints

Let $D$ be a DSL, $Prod(D)$ denotes the set of production rules in $D$ and $Term(D)$ the set of terminal symbols in $D$. Furthermore, let $Types(D)$ denotes the set of types used in $D$ and $Type(s)$ the type of symbol $s \in Prod(D) \cup Term(D)$. If $s \in Prod(D)$, then $Type(s)$ denotes the return type of production rule $s$.

To ensure that every program enumerated is well-typed the following constraints must be satisfied:

**Leaf Nodes.** The leaf nodes can only be assigned to terminal symbols because they have no children. Therefore, we define the following constraint:

$$\forall i \in L : \bigvee_{p \in Term(D)} v_i = id(p) \tag{1}$$

**Internal Nodes.** If a production rule $p$ is assigned to an internal node, then the type of its children nodes must match the types of parameters of $p$. Let $Type(p, j)$ denote the type of parameter $j$ of production rule $p \in Prod(D)$. If $j > arity(p)$, then $Type(p, j) = empty$. If $p$ is a terminal symbol, $p \in Term(D)$, then for every $j$, $Type(p, j) = empty$.

Let $\Sigma(Type(p, j))$ represent the subset of symbols in $\Sigma$ of type $Type(p, j)$.

$$\forall i \in I,\ j \in C(i),\ p \in \Sigma\ : \\ v_i = id(p) \Rightarrow \bigvee_{t \in \Sigma(Type(p,j))} v_j = id(t) \tag{2}$$

With constraint (2), all the programs generated will be well-typed since each node is only assigned to a production rule if its children have the correct type.

**Example 3.** *Consider again the query in Example 2. If the production select_from is assigned to the program's root, $v_1$, then $\Sigma(Type(select\_from, 1)) = \{column, columns\}$ and $\Sigma(Type(select\_from, 2)) = \{select\_from, join, parts, supplier\}$. The*

*following constraint must be satisfied:* $v_1 = id(select\_from) \Rightarrow \big(v_2 = id(column) \vee v_2 = id(columns)\big)$ *and* $v_1 = id(select\_from) \Rightarrow \big(v_3 = id(select\_from) \vee v_3 = id(join) \vee v_3 = id(parts) \vee v_3 = id(supplier)\big)$.

**Encoding Complexity.** Let $k$ be the greatest arity between DSL constructs and let $n$ denote the number of productions (lines of code) in a program. In terms of nodes complexity, the number of nodes used by tree-based enumeration increases exponentially with the number of productions, as follows:

$$\frac{k^{n+1} - 1}{k - 1} \tag{3}$$

### 3.3. Query Synthesis

Query Synthesis is a subfield of Program Synthesis. Given a database $\mathcal{D}$ and a query specification (e.g. input-output examples, natural language descriptions), the goal is to achieve the desired query.

**Query Reverse Engineering (QRE).** As defined in section 2, QRE is a subproblem of Programming By Example. QRE has numerous applications [31] like database usability, data analysis and data security.

The problem of QRE first appeared in 1975 by Zloof [38]. Zloof introduced a new language called "Query By Example" (QBE) [38, 39, 40] so the users who did not know how to program with SQL could query databases without having to comprehend SQL, just needing to learn QBE [38].

In the last decade several systems were developed trying to solve QRE: TALOS [30, 31], SQLSynthesizer [37], QFE [15], STAR [36], REGAL [28], FastQRE [13], REGAL+ [29], PALEO [23], Morpheus [6], Neo [7], Scythe [33, 34] and Trinity [18].

#### 3.3.1. Scythe

Wang et al. [33, 34] proposed *Scythe* at PLDI'17, a novel query-by-example synthesizer. Scythe can synthesize expressive SQL queries and is considered one of the best state-of-the-art synthesizers regarding SQL generation.

This framework can be divided into two major steps: synthesis and disambiguation. The synthesizer is the step responsible for generating a set of queries that are consistent with the specifications provided by the user. Afterwards, the synthesizer ranks all the generated queries (based on simplicity and naturalness) and passes them to the disambiguation module, where the user chooses which query is the desired one.

Regarding the synthesis process, Scythe decomposes this problem into two main parts: (1) synthesis of skeletons (i.e. abstract queries) and (2) synthesis of filter predicates.

Scythe starts by enumerating all possible query skeletons that can be obtained considering the schema graph (see Definition 4) provided by the user as input examples. Secondly, Scythe removes all skeletons whose schema graph does not correspond to the output example, i.e., skeletons that do not contain the output table are removed. Afterwards, all candidates use the input examples and generalize the output table, hence, the system just needs to find which predicates are going to be used. This search for predicates can be huge. Therefore, Wang et al. [33] proposed two optimizations: locally grouping candidate predicates and encode tables using bit-vectors. With these optimizations the synthesis process is considerably accelerated, which allow the synthesis of a wider range of SQL operators. For more details the interested reader is referred to the literature [20, 33].

## 4. SQUARES

This section provides the description of SQUARES, **A** **S**QL **S**ynthesizer **U**sing **Q**uery **R**everse **E**ngineering, an enumeration-based PBE system developed on top of a state-of-the-art synthesis framework, Trinity [18]. SQUARES' goal is to synthesize SQL queries from input-output examples.

SQUARES, like Trinity, receives as input a set of input-output examples, a DSL and an interpreter. SQUARES' synthesizer can be divided into two main components: enumerator and decider. The enumerator is responsible for enumerating all possible programs for the DSL, $\mathcal{D}$, provided as input. For each program $\mathcal{P}$, the interpreter runs $\mathcal{P}$ on the input examples and the decider compares if the output matches the expected one. If the output of $\mathcal{P}$ does not match, the decider produces a reason of failure which is used by the enumerator to prune all equivalent infeasible programs from the search space, like in Neo [7], afterwards, the next candidate program is enumerated. Otherwise, if the output of $\mathcal{P}$ matches the expected one, the synthesizer translates $\mathcal{P}$ to SQL and returns it.

For details about the structure of the input-output examples, the SQUARES' interpreter or the heuristics used to cut the program space, the interested reader is referred to the literature [20].

### 4.1. Domain-Specific Language (DSL)

This section describes the Domain-Specific Language (DSL) we use in SQUARES and from which we generate SQL. Since constructing a SQL grammar is very complex, we opted for creating a DSL inspired by the R language[2]. Having a query in R,

[2] https://www.r-project.org

$$
\begin{array}{lcl}
table & \rightarrow & input \mid inner\_join(table,\ table) \mid inner\_join3(table,\ table,\ table) \mid \\
 & & inner\_join4(table,\ table,\ table,\ table) \mid filter(table,\ filterCondition) \mid \\
 & & filters(table,\ filterCondition,\ filterCondition,\ op)) \mid \\
 & & summariseGrouped(table,\ summariseCondition,\ Cols) \mid \\
 & & anti\_join(table,\ table) \mid left\_join(table,\ table) \mid bind\_rows(table,\ table) \mid \\
 & & intersect(table,\ table) \\
tableSelect & \rightarrow & select(table,\ selectCols,\ distinct) \\
op & \rightarrow & Or \mid And \\
distinct & \rightarrow & true \mid false \\
empty & \rightarrow & empty
\end{array}
$$

Figure 6: SQUARES' DSL.

we can easily translate it to SQL as explained in the end of this section.

Fig. 6 presents our DSL with five distinct types: *table, tableSelect, op, distinct* and *empty*. The input's type is *table* and the output's type is *tableSelect*. Regarding the variety of production rules, we use the basic operations offered by R's *dplyr* [3] library (e.g. `inner_join, filter, summarise, left_join`). The terminals belonging to *filterCondition, summariseCondition, Cols, selectCols* (Fig. 6), are computed on the fly, because they depend on the input-output examples, as well as, the number of input tables.

**From R to SQL.** Each production rule present in our DSL, Fig. 6, has a direct translation (interpretation) to R [20]. Each production rule in R (e.g. `anti_join, summarise`) can be easily translated into several operators in SQL[4] (e.g. `anti_join` → `SELECT ... FROM... WHERE NOT EXISTS ...`). Therefore, we can generate programs with more SQL productions rules if we use a DSL for R and then translate the desired program to SQL, instead of generating a program directly from a SQL grammar. R's library *dbplyr* [5] has a built-in function, *show_query*, that receives a query in R and translates it automatically to SQL. On that account and for the sake of simplicity, we use this function to obtain SQL from our DSL.

### 4.2. Line-based Encoding

In this section we propose a new encoding to represent symbolic programs. Our goal is to represent a program as a sequence of lines where each line represents an operation in the DSL. Instead of using a single $k$-tree to represent a program, each line is represented as a tree with depth of 1.

Consider the program in Fig. 7. One can represent this program as three trees of depth 1 as shown in Fig. 8. Note that the result of the program is the value returned by the third tree.

[3]https://cran.r-project.org/web/packages/-dplyr/vignettes/dplyr.html

[4]https://dbplyr.tidyverse.org/articles/sql-translation.html

[5]https://cran.r-project.org/web/packages/-dbplyr/vignettes/dbplyr.html

$$
\begin{array}{lll}
L_1 & : ret_1 \leftarrow column(pname) \\
L_2 & : ret_2 \leftarrow join(parts,\ supplier) \\
L_3 & : ret_3 \leftarrow select\_from(ret_1,\ ret_2)
\end{array}
$$

Figure 7: The query *select_from(column(pname), join(parts, supplier))*, from Example 2, divided into three lines.

Observe that $ret_i$ is a new symbol that represents the return value of line $i$.

### 4.3. Encoding Variables

Recall that $D$ denotes a DSL, $Prod(D)$ the set of production rules in $D$ and $Term(D)$ the set of terminal symbols in $D$. Furthermore, $Types(D)$ denotes the set of types used in $D$ and $Type(s)$ the type of symbol $s \in Prod(D) \cup Term(D)$. If $s \in Prod(D)$, then $Type(s)$ denotes the return type of production rule $s$.

Consider the encoding for a program with $n$ lines where the maximum arity of the operators is $k$, then we have the following variables: (1) $O = \{op_i : 1 \leq i \leq n\}$ : each variable $op_i$ denotes the production rule used in line $i$; (2) $T = \{t_i : 1 \leq i \leq n\}$ : each variable $t_i$ denotes the return type of line $i$; (3) $A = \{a_{ij} : 1 \leq i \leq n, 1 \leq j \leq k\}$ : each variable $a_{ij}$ denotes the symbol corresponding to argument $j$ in line $i$;

### 4.4. Constraints

Besides the production rules $Prod(D)$ and terminal symbols $Term(D)$, we define one return symbol for each line in the program. Let $Ret = \{ret_i : 1 \leq i \leq n\}$ denote the set of return symbols in the program.

In our encoding, we define a different non-negative identifier for each symbol. Here, we extend the *id* function to also consider the symbols that represent the return value of each line. Let $Symbols = Prod(D) \cup Term(D) \cup Ret$ define the set of all symbols used in the program. Finally, let $id : Symbols \rightarrow \mathbb{N}_0$ and $tid : Types(D) \rightarrow \mathbb{N}_0$ be one-to-one mappings of symbols and types, respectively, to non-negative integer values.

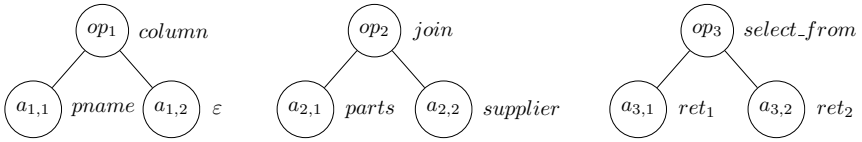


Figure 8: Each tree represents a production rule. The first tree represents line 1, the second tree represents line 2 and the third tree represents line 3. $ret_1$ (resp. $ret_2$) denotes the value returned in line 1 (resp. line 2).

**Operations.** First, the operations in each line must be production rules. Hence, we have the following set of constraints:

$$\forall 1 \leq i \leq n : \bigvee_{p \in Prod(D)} (op_i = id(p)) \tag{4}$$

The operation symbol used in each line implies the line's return type.

$$\begin{aligned} &\forall 1 \leq i \leq n, p \in Prod(D) : \\ &(op_i = id(p)) \Rightarrow (t_i = tid(Type(p))) \end{aligned} \tag{5}$$

Given a sequence of operations, the arguments of operation $i$ must either be terminal symbols or return symbols from previous operations. Hence, we have:

$$\begin{aligned} &\forall 1 \leq i \leq n, 1 \leq j \leq k : \\ &\bigvee_{s \in Term(D) \cup \{ret_r : r < i\}} (a_{ij} = id(s)) \end{aligned} \tag{6}$$

**Arguments.** The arguments for a given operation $i$ must have the same types as the parameters of the production rule used in the operation. Let $Type(p, j)$ denote the type of parameter $j$ of production rule $p \in Prod(D)$. If $j > arity(p)$, then $Type(p, j) = empty$. Hence, we have the following constraints when a return symbol is used as argument of an operation:

$$\begin{aligned} &\forall 1 \leq i \leq n, p \in Prod(D), \\ &1 \leq j \leq arity(p), 1 \leq r < i : \\ &((op_i = id(p)) \wedge (a_{ij} = id(ret_r))) \Rightarrow \\ &\Rightarrow (t_r = tid(Type(p, j))) \end{aligned} \tag{7}$$

A given terminal symbol $t \in Term(D)$ cannot be used as argument $j$ of an operation $i$ if it does not have the correct type:

$$\begin{aligned} &\forall 1 \leq i \leq n, p \in Prod(D), 1 \leq j \leq arity(p), \\ &s \in \{t \in Term(D) : Type(t) \neq Type(p, j)\} : \\ &(op_i = id(p)) \Rightarrow \neg(a_{ij} = id(s)) \end{aligned} \tag{8}$$

Since the arity of a given operation $i$ can be smaller than $k$, we must also have that the arguments above the production's arity must be assigned to the empty symbol:

$$\begin{aligned} &\forall 1 \leq i \leq n, p \in Prod(D), arity(p) < j \leq k : \\ &(op_i = id(p)) \Rightarrow (a_{ij} = id(empty)) \end{aligned} \tag{9}$$

$$\begin{aligned} L_1 &: ret_1 \leftarrow column(pname) \\ L_2 &: ret_2 \leftarrow join(parts,\ supplier) \\ L_3 &: ret_3 \leftarrow select\_from(ret_1,\ ret_2) \\ L_1 &: ret_1 \leftarrow join(parts,\ supplier) \\ L_2 &: ret_2 \leftarrow column(pname) \\ L_3 &: ret_3 \leftarrow select\_from(ret_2,\ ret_1) \end{aligned}$$

Figure 9: Two different ways of representing the program from Example 2 into three lines.

**Encoding Complexity.** Let $k$ be the greatest arity between DSL constructs and let $n$ denote the number of productions (lines of code) in a program. In terms of nodes complexity, we can observe a drastic difference between both types of enumeration, tree-based and line-based. The number of nodes used by line-based enumeration increases linearly, $(k + 1) \times n$, because the enumerator uses $n$ trees, with $k + 1$ nodes each, to represent a program with $n$ production rules.

### 4.5. Symmetric Programs

In line-based encoding, the number of models of the SMT formula is larger than the number of models in the corresponding tree-based encoding, since each program can have more than one representation, i.e. symmetric programs.

**Example 4.** *Consider the DSL in Fig. 4 and the program* `select_from(column( pname), join(parts, supplier))` *from Example 2. In tree-based encoding this program has a single representation shown in Fig. 5. However, for the same program, line-based encoding has two possible representations shown in Fig. 9.*

In order to enumerate the same number of models from the SMT formula in both types of enumeration, we need to find these symmetries and block

them. Otherwise, symmetric programs as the one in Fig. 9 will be enumerated and the synthesizer will have to check both programs. Therefore, if we have a model $\alpha$ of a line-based SMT formula and the synthesizer verifies that the corresponding program is not consistent with the input-output examples, then all models that encode programs symmetric to the one encoded by $\alpha$ can be blocked. A simple way to find these symmetries is through a Directed Acyclic Graph (DAG) of dependencies, where a vertex is defined for each program line, and edges correspond to the line dependencies in the program. After building the graph, one can enumerate all possible topological orders of vertexes in the dependency graph. Next, each program associated with a topological order is blocked in the SMT formula. Our experiments show that symmetry breaking does not improve the performance of line-based enumeration. The interested reader is referred to the literature [20, 21] for more details.

## 5. Experimental Results

SQUARES is implemented in Python and uses the Z3 SMT solver [4] with the theory of Linear Integer Arithmetic to check the satisfiability of formulas generated by our enumerator. We developed SQUARES on top of the Trinity [18] synthesis framework.

All of the experiments presented in this section were conducted on an Intel(R) Xeon(R) computer with E5-2630 v2 2.60GHz CPUs, using a memory limit of 64GB and a time limit of 3,600 seconds.

### 5.1. Line-based vs Tree-based Encodings

**Benchmark.** We started with an initial set of 23 SQL instances (corresponding to Sections 5.1.1 and 5.1.2 of the database textbook [24]), these instances were previously used by well-known SQL synthesizers [6, 33, 37]. Then we created variants of these instances resulting in a total of 55 instances. Since we want to study the performance of each encoding with respect to the size of the synthesized query, for each of these instances, we generate six different SMT formulas to search for programs that use exactly $n$ production rules from our DSL, for a total of 330 instances ($55 \times 6$, $1 <= n <= 6$).

**Performance.** Table 1 shows the number of instances solved by each model for a given number of lines in our DSL. The performance of both encodings is similar for programs with three or fewer lines of code. However, when the program size increases, the difference between these approaches becomes clear. The last column of Table 1, shows the percentage of solved instances by each approach for instances using more than three lines of code. The tree-based model only solves around 33% of the instances while line-based solves around 82%.

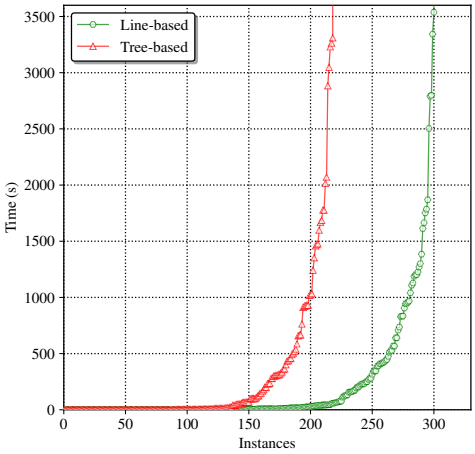


Figure 10: Tree-based vs Line-based Enumerators.

In terms of time spent in each instance, Fig. 10 shows a cactus plot. This plot shows the synthesis time ($y$-axis) against the number of instances solved ($x$-axis). Fig. 10 supports the results shown in Table 1. Additionally, this plot show that tree-based enumeration is, in general, significantly slower than line-based enumeration. These differences in time and number of instances solved, in particular for the instances with more than 3 lines, can be justified by the exponential number of variables and constraints required by tree-based enumeration.

### 5.2. SQL Generation

In the interest of evaluating SQUARES' SQL, we compared the performance of SQUARES against Scythe [33], a state-of-the-art SQL synthesizer presented in Section 3.3.1.

#### 5.2.1. Textbook Benchmark

**Benchmark.** We used the first 28 SQL queries of the database textbook (corresponding to Sections 5.1.1 and 5.1.2 and part of 5.1.3 of the database textbook [24]).

**Performance and Discussion.** Fig. 11 shows a cactus plot. Both systems had a similar performance being able to solve 19 instances. However, SQUARES is slightly faster than Scythe. This happens in 5 of the instances solved (almost 26%).

Regarding the SQL generation, both systems produce verbose SQL queries. We observed that normally Scythe produces queries with more inner *SELECT*s than SQUARES. Although both systems produce queries with more inner *SELECT*s than a human would write. However, some SQL dialects accept at most two tables in a *JOIN*. Therefore, in these cases the number of *SELECT*s would be higher. SQUARES provides a cleaner presentation of the SQL comparing to Scythe thanks to *sql-*

Table 1: Number of solved instances by each approach.

| Lines of Code | 1 | 2 | 3 | 4 | 5 | 6 | Total | % Solved | % Solved for LOC >= 4 |
|---|---|---|---|---|---|---|---|---|---|
| # Instances | 55 | 55 | 55 | 55 | 55 | 55 | 330 | | |
| Tree-based | 55 | 55 | 54 | 34 | 18 | 2 | 218 | 66.06% | 32.73% |
| Line-based | 55 | 55 | 54 | 49 | 48 | 39 | 300 | 90.91% | 82.42% |

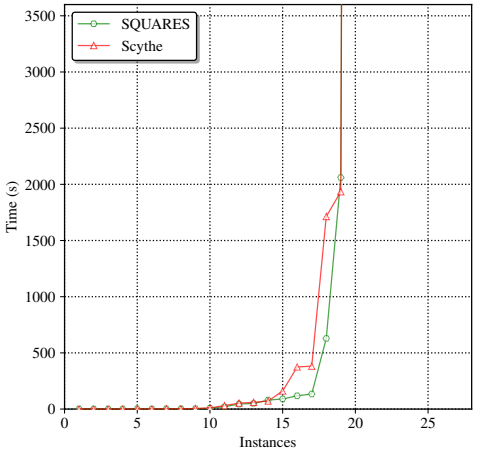


Figure 11: Performance of SQUARES and Scythe on 28 instances from a database textbook [24].

*parse* [6], a python library to parse SQL. With this library, we make the query more readable.

5.2.2. OutSystems Benchmark

**Benchmark.** In collaboration with Outsystems' engineers and using SQL Server Management Studio [7], OutSystems' main database $\mathcal{D}$, was tracked in order to collect all the queries that ran on $\mathcal{D}$ for seven hours in five different days. The six most used tables from the database where chosen. Hence, the examples with other tables were excluded. After we collected the queries that ran on $\mathcal{D}$ on those five days, we removed the copies and chose only the examples that use the six tables. We achieved a set of 20 queries.

**Performance and Discussion.** We observed that Scythe only solves two instances out of twenty (10 percent). On the other hand, SQUARES solved all twenty instances. The poor performance of Scythe can be explained by its problems with memory usage. We believe that Scythe encodes the tables' data into constraints. Each input table, used in the OutSystems' examples, has two thousand five hundred entries. Hence, this may be the reason that Scythe returns, in 90 percent of the instances, an *OutOfMemoryError*.

[6] https://pypi.org/project/sqlparse

[7] https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms

Regarding the generated SQL, it is difficult to compare since Scythe only solves two instances. In these two instances, both systems produce similar queries.

## 6. Conclusions and Future Work

In this work, we propose SQUARES, an enumeration-based program synthesizer whose goal is to solve the problem of SQL Synthesis by Example, also known as, Query Reverse Engineering (QRE).

Until now, enumeration-based program synthesizers [6, 7, 18] have been using a tree-based encoding (see Section 3.2.1) to represent the search space of possible programs. To the best of our knowledge, SQUARES is the first enumeration-based program synthesizer that uses a new representation of programs, where each program is represented as a sequence of lines [21].

Experimental results on the synthesis of SQL queries, show that the proposed line-based encoding allows a faster enumeration of programs when compared to the usual tree-based encoding.

We compared SQUARES against Scythe [33], a state-of-the-art QRE framework, in order to evaluate SQUARES in terms of SQL generation. We used instances from a database textbook [24] and instances from OutSystems' database. Concerning textbook instances, both systems show similar performances and produce similar queries. Regarding the OutSystems' instances, SQUARES shows great performance solving all of them. However, Scythe shows a weak performance on these instances which can be justified by its memory limitations.

As future work, it would be interesting to pursue several topics regarding our line-based encoding and our Domain-Specific Language (DSL).

With respect to our encoding, firstly, other symmetry breaking techniques should be tested, such as breaking symmetries through a lexicographic order. This type of symmetry breaking is expected to improve the performance of the proposed line-based encoding. Moreover, in order to evaluate our encoding in terms of scalability, it would be interesting to evaluate its performance generating programs with more than six lines of code.

In regard to our DSL, it should be extended in order to allow SQUARES to generate queries with a larger diversity of SQL operators.

## References


[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[2] K. Chandra and R. Bodík. Bonsai: synthesis-based reasoning for type systems. *PACMPL*, 2 (POPL):62:1–62:34, 2018.

[3] A. Church. Applications of recursive arithmetic to the problem of circuit synthesis. *Summaries of the Summer Institute of Symbolic Logic*, 1(1):3–50, 1957.

[4] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[5] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 345–356, 2016.

[6] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017.

[7] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435, 2018.

[8] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011.

[9] S. Gulwani and M. Marron. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 803–814, 2014.

[10] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[11] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[12] R. Joshi, G. Nelson, and Y. Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28 (6):967–989, 2006.

[13] D. V. Kalashnikov, L. V. S. Lakshmanan, and D. Srivastava. Fastqre: Fast query reverse engineering. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 337–350, 2018.

[14] A. S. Lezama. *Program synthesis by sketching*. PhD thesis, UC Berkeley, 2008.

[15] H. Li, C. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13):2158–2169, 2015.

[16] Z. Manna and R. J. Waldinger. Knowledge and reasoning in program synthesis. *Artif. Intell.*, 6(2):175–208, 1975.

[17] R. Martins, S. Joshi, V. M. Manquinho, and I. Lynce. Incremental cardinality constraints for maxsat. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 531–548, 2014.

[18] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig. Trinity: An extensible synthesis framework for data science. *PVLDB*, 12(12): 1914–1917, 2019.

[19] A. K. Menon, O. Tamuz, S. Gulwani, B. W. Lampson, and A. Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 187–195, 2013.

[20] P. Orvalho. Squares : A sql synthesizer using query reverse engineering. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, 2019.

[21] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. Encodings for enumeration-based program synthesis. In *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, pages 583–599, 2019.

[22] OutSystems. . `https://www.outsystems.com`, 2019. [Online; accessed 1-September-2019].

[23] K. Panev and S. Michel. Reverse engineering top-k database queries with PALEO. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016.*, pages 113–124, 2016.

[24] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[25] M. Schlaipfer, K. Rajan, A. Lal, and M. Samak. Optimizing big-data queries using program synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 631–646, 2017.

[26] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 343–356, 2016.

[27] A. Solar-Lezama. The sketching approach to program synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, pages 4–13, 2009.

[28] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. Reverse engineering aggregation queries. *PVLDB*, 10(11):1394–1405, 2017.

[29] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. REGAL+: reverse engineering SPJA queries. *PVLDB*, 11(12):1982–1985, 2018.

[30] Q. T. Tran, C. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 535–548, 2009.

[31] Q. T. Tran, C. Y. Chan, and S. Parthasarathy. Query reverse engineering. *VLDB J.*, 23(5): 721–746, 2014.

[32] R. J. Waldinger and R. C. T. Lee. PROW: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, pages 241–252, 1969.

[33] C. Wang, A. Cheung, and R. Bodík. 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, Barcelona, Spain, June 18-23, 2017*, pages 452–466, 2017.

[34] C. Wang, A. Cheung, and R. Bodík. Interactive query synthesis from input-output examples. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1631–1634, 2017.

[35] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: query synthesis from natural language. *PACMPL*, 1(OOPSLA):63:1–63:26, 2017.

[36] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 809–820, 2013.

[37] S. Zhang and Y. Sun. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 224–234, 2013.

[38] M. M. Zloof. Query by example. In *American Federation of Information Processing Societies: 1975 National Computer Conference, 19-22 May 1975, Anaheim, CA, USA*, pages 431–438, 1975.

[39] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA.*, pages 1–24, 1975.

[40] M. M. Zloof. QBE/OBE: A language for office and business automation. *IEEE Computer*, 14 (5):13–22, 1981.